Contents

Chapter 1

Introduction

This book is an introduction to code generation. Code generation is the process by which the *backend* of a programming language compiler transforms a parsed and analysed program into object code. We shall assume that the means by which programs are parsed and analysed in the compiler *frontend* are already understood. Chapter 2 is a short chapter on semantic analysis, which is necessary only for consistency with the rest of the material.

Chapter 3 is a fairly comprehensive account of runtime organization, which describes the execution environment for which our compilers generate code. In particular, the different conventions for procedure calling and parameter passing are discussed in some detail.

Two different methods of code generation are discussed. The first method involves the generation of an intermediate form and the subsequent generation of code using an interpretive automaton. This method is quite widely used in production compilers, but has been little discussed in the literature. The other method of code generation is bottom-up tree rewriting. This method has a much shorter history, but is the most capable method currently known.

A number of chapters deal with the subject of code improvements, or *optimizations* as they are generally called. Another chapter deals with the critical issue of register allocation. The emphasis is on the global methods of register allocation, which depend on the same kind of computations as the so-called *global optimizations.*

1.1 Assumed background

It is assumed that the reader is familiar with fundamentals of the theory of programming languages, and is a practised programmer. A fair degree of familiarity with standard algorithms and data structures is also assumed. Most of the examples in the book which use high level programming language use the language *Modula-2*, however, for most purposes a familiarity with either *Ada* or *Pascal* would serve almost as well. It is also assumed that the reader has some familiarity with the structure and syntax of *ANSI C*, although expertise in programming in this language is not assumed.

Some aspects of compilation and runtime structure of the so-called *Object Oriented* languages are described also. In these sections, no prior familiarity with such languages is assumed.

The book uses an example intermediate form called *DCode* which is fully described in an appendix, and has a publicly available reference document. There is also, inevitably, a fair amount of assembly language. It is assumed that the reader is at least familiar with the concepts of assembly language, and of machine organization.

In some of the chapters of the book, some formal notations are necessary. Most of these notations are explained in place, and in any case rely only on mathematical constructs which should be familiar to every student of computer science. While formal material has been kept to a minimum, the remaining uses of mathematical notation are both necessary and helpful.

This book does not cover the theory of formal languages, parsing and tree building. It is assumed that the reader is either familiar with the theory and practice of such matters, or is prepared to abstract such things away and start with an already constructed abstract syntax tree.

1.2 The reference architecture

In order to be able to treat the subject in a structured manner, a **reference architecture** is introduced. This is the example which is treated in detail. Of course there are a large number of alternative architectures. In particular, many compilers fold together several layers of the reference architecture, or perform the operations in a different order.

The question of what is usually called *code optimization* is ignored for the moment. In any case, this name has been rather badly chosen. Although there are many optimality results which have been proved about code generation these do not relate to realistic practice. A much better name would be *code improvement*.

The positions at which code improvement transformations may be placed in the reference architecture are shown in figure 1.1. The subject of code improvement will be treated in detail in later chapters.

1.3 Intermediate Forms

Each of the layers in the reference architecture involve the definition of an intermediate representation of the program being compiled.

1.3.1 Abstract syntax tree

The form of the abstract syntax tree (*AST*) is not dealt with in detail in this book, being the concern of the compiler front-end. This book is concerned with the form of the *AST*, only to the extent that it is necessary to understand how to traverse the trees emitting the code of the intermediate representation (*IR*) proper. It suffices to note that the form will usually be a representation in

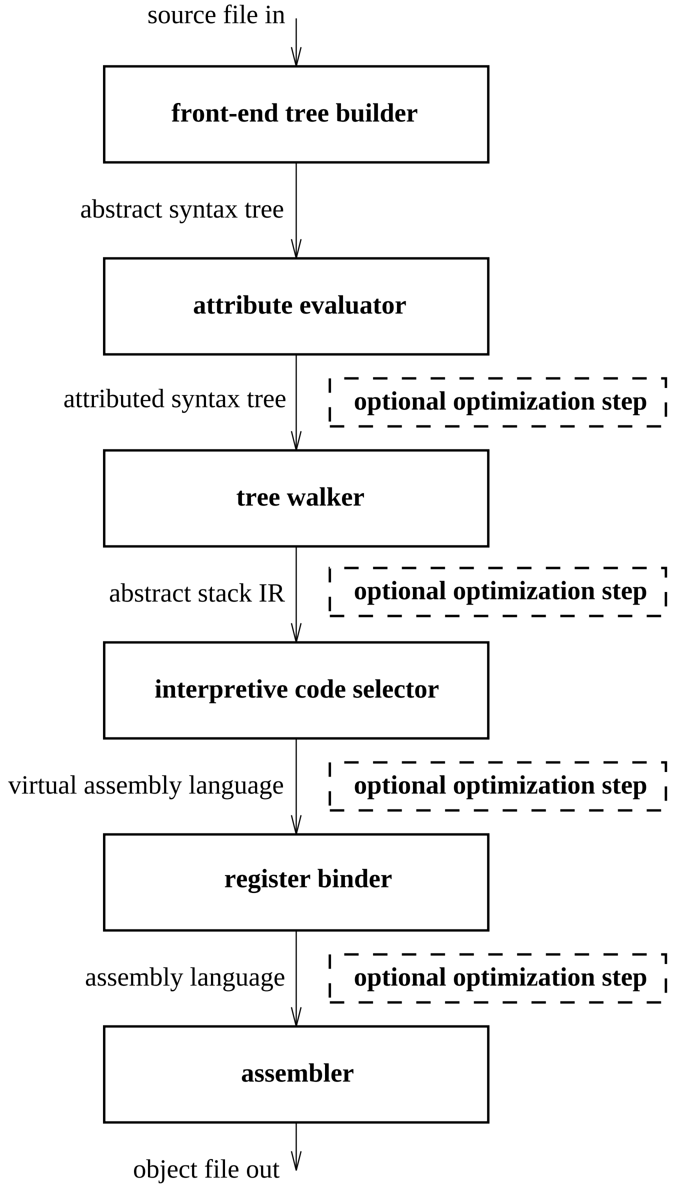


Figure 1.1: the reference architecture

dynamic memory of the whole program, or perhaps just one procedure at a time. The form will probably be language dependent, and we shall assume that any attributes which the treewalker requires are evaluated and stored in the treee during the static semantic analysis of the program.

1.3.2 Abstract stack intermediate representation

It is this level of representation of the program which is usually called **the** intermediate representation, or *IR*. The importance of this level is that it is here that the cleanest separation is possible of the language dependent and independent part of the processing, and the machine dependent and machine independent parts.

There are two main forms of *IR* in common use, but this book concentrates on just one. The chosen form is one based on the instruction set for a virtual machine based on a push-down stack. All evaluation of expressions is done on this *abstract stack*. We call this form an abstract stack form because it refers to an idealized stack machine, a *virtual machine* with most of the messy details of all real machines hidden away. In particular we use abstractions for such things as parameter passing, so that we may handle machine conventions which pass parameters in differing ways.

The particular form of abstract stack *IR* used here is called *DCode*. This form is used in all the **gardens point** compilers, and is fully specified in a maintained report on the internet. The *DCode* form is rather larger than one would ideally chose for purposes of exposition, possessing over one hundred instructions. However, against that must be balanced the advantage of being complete, and having proved adequate to the demands of multiple languages and multiple targets. Furthermore, the tools which are available for use with this book all use this *IR*.

There is a further option available for code selection, which uses the tree form itself as an intermediate representation. Such code selectors work by *bottom-up tree rewriting,* and provide the most powerful pattern matching method available at present. These code generators are treated in chapter 6.

1.3.3 Virtual assembly language

The result of code generation, in the reference architecture is **virtual assembly language** (*VAL*). This language is chosen at the convenience of the compiler writer. It is usually similar to the actual assembly language of the target machine, except that the number of registers is unlimited. This allows for the generation of code to be carried out separately from the allocation of registers.

In some cases, it is convenient to modify the instruction set of the *VAL* to allow for more convenient processing. For example, in the case of the *Intel-x86* architecture the same assembly language mnemonic "`movl`" (move long word) is used for both loading 32-bit registers *from* memory, and for storing such registers *to* memory. For this machine the processing of the code generator is simpler if the two cases are kept separate, by the use of two fictional instructions "`lw`" (load word), and "`sw`" (store word). It is a trivial matter to output the same final "`movl`" instruction for virtual instructions of both kinds.

The single assignment rule

Although it is not a necessary consequence of the use of *VAL* we shall also implement a restriction in our code generators which ensure that each virtual register has a single definition point. This rule must be strictly observed, since some of the simple register allocation strategies rely on this property. In cases where it is logically necessary to allocate different values to the same virtual register, we will use special "non-defining" assignments to mark the different semantics.

1.3.4 A note on assembly language syntax

Since we deal with a variety of different machine architectures in the examples, it is inevitable that we must deal with a variety of assembly language notations. Unfortunately, there is no standard way in which such languages are written. Even for a single machine architecture, such as *iapx386* there is no standard notation, and different assemblers for this architecture use different operand orders, and even different ways of specifying the operand size.

In order to minimize the problems in changing from one assembler to another this book adopts a uniform convention for all assembly languages. In this convention the operand size is implicit in the operation code, and the destination operand always comes last on the line.

This notation is different to that used by *MIPS* and by the widely-used *MASM* assembler for *iapx386*. It follows that the output which is observed from some of the tools which are described here will be different to that shown on the page. This is a pity, but is infinitely preferable to having to switch formatting conventions, sometimes even in mid-page.

The assembly language notation has the form —

`opcode` *input-operand(s) output-operand(s)*

Operands are comma-separated, and the lexical conventions treat everything on the line after a semicolon as a comment.

Addressing modes use symbolic names, numeric offsets, and address registers. Symbolic names are written as identifiers, with numeric offsets attached to the symbolic names, if any. Address register operands are shown in parentheses, comma separated in those cases where machine have multi-register addressing modes.

Thus the single address mode in *MIPS* (indexed) is denoted as in the following example —

```
lw globVar+24(r4),r5
```

which loads register-5 with the 32-bit word at the address —

> address of `globVar`, plus
> offset 24, plus
> contents of index register `r4`.

In the case of *iapx386*, the most general addressing mode (scaled, based, indexed) is denoted as in the following example —

```
lw globVar+24(esi,edi,4),eax
```

In this example, the 32-bit accumulator `eax` is loaded with the (long) word at address —

> address of `globVar`, plus
> offset 24, plus
> contents of base address register `esi`, plus
> contents of index register `edi`, scaled by four.

In the *Intel-x86* case, unused address register fields may be left empty in the comma separated list within the parentheses.

1.4 How to avoid code generation entirely

The strings in the *IR* contain all of the semantic content of the original source programs. It is possible to avoid any further work in language transformations in a few special cases. Firstly, it is possible for the final target machine to directly execute the *IR*. Historically, machines have been built which have stack-like architectures, and in these cases it is possible for the *IR* to be the actual machine language.

It is also possible to write an interpreter program which performs the instructions of the *IR* by manipulating a stack data structure at runtime. This concept was the basis of the *UCSD Pascal* series of compilers. A single front-end produced *P-Code*, and for each target machine it was only necessary to write an interpreter. The interpreter was usually written in assembly language.

Interpreters tend to have a fairly high overhead in their execution. For example, an interpreter executing *DCode* on an *Intel-x86* processor executes about 10% of the speed of native code on the same machine. This overhead is caused by the number of instructions of the interpretive *fetch-execute* loop. This loop repeatedly fetches the next instruction of the *IR*, and uses this as an index into a dispatch table. The dispatch table simply points to the machine code which manipulates the stack according to the actual instructions of each code.

Just as a matter of interest, figure 1.2 has a fragment of the *Intel-x86* interpreter for *DCode*, and the code for one simple instruction. In this case, 16-bit segment, *iap386* code is shown.

1.5 A little bit of duality

One of the recurring themes in code generation is the possiblity of replacing operations at compile time with operations at runtime, and conversely. Consider

```
     ; es:(di) points to the next DCode instruction
     ; ds:(si) points to the top of the evaluation stack (e-stack)
...
nextDCode:                    ; Fetch operation of interpreter
   movzbw es:(di),bx          ; get next DCode & zero extend to word
   inc   di                   ; index to next dcode
   jmp   cs:DcodeTable(bx,bx); jump to code through dispatch table
...
andWrd:  ; dcode  - bitwise AND of the top two words of the e-stack.
         ; method - pops the top two 32-bit words from the e-stack,
         ;          performs AND, pushes the result back on e-stack
   movl  ds:(si),eax          ; right op is popped from e-stack into eax
   subw  si,4                 ; adjust the e-stack pointer
   andl  eax,ds:(si)          ; AND with left operand
   jmp   nextDcode            ; jump to main interpreter loop
```

Figure 1.2: Sample interpreter code

the following example, that of generating code for a block copy operation. This operation occurs as a primitive in most *IR*s, arising from the assignment of entire variables, for example. How is the code to be generated for this operation, for machines which do not have such a block copy operation as a primitive instruction?

It is possible to perform such a block copy at runtime by two methods. The data may be copied word-by-word using a simple sequence of instructions, or the data may be copied using a loop. It would be normal for contemporary compilers to use the first method for small blocks, and the second for blocks larger than some particular size.

Let us look at the outline of the compiler code for producing this object code. It is assumed that the source and destination addresses are already loaded into two registers `src` and `dst` respectively —

```
load rSrc,rDst with source, destination addresses
IF size > inlineLimit THEN
    AllocateLabel(new);
    EmitLabel(new);
    emit "lw (rSrc),reg"
    emit "sw reg,(rDst)"
    emit "inc rDst"
    emit "inc rSrc"
    emit test and branch to new
ELSE (* do inline expansion *)
    offset := 0;
    WHILE size > 0 DO
        emit "lw offset(rSrc),reg"
        emit "sw reg,offset(rDst)"
        INC(offset); DEC(size);
    END; (* while *)
END; (* if *)
```

Notice that the first case has straight-line code at compile time, and a loop at runtime. The second case has a loop at compile time, but has straight-line code at compile time. Even the address increment instructions move between the two cases. Is this not an intriguing and beautiful duality?

On a much larger scale, runtime interpretation and interpretive code generation demonstrate the same duality. An interpreter manipulates a stack data structure at runtime, an interpretive code generator manipulates a stack data structure at compile time.

1.6 Compiler Bootstrapping

It is quite common for compilers for procedural languages to be written in the language which they compile. Although it is not necessary to do so, there are some good reasons to choose such a design. We shall deal with the arguments in favour of such a strategy at the end of this section, but for now we wish to address the obvious question *how does a compiler compile itself?*

Once a compiler has been compiled, and is reasonably correct, it is easy to see how it may compile its own source code. The problem is, how does one get to this point, starting from the original source code the first time. This is the *compiler bootstrapping* problem. There are two extreme scenarios, creating a compiler for a new language, and creating a compiler for an existing language, but on a new machine. A helpful notation for discussing this process is the T-diagram.

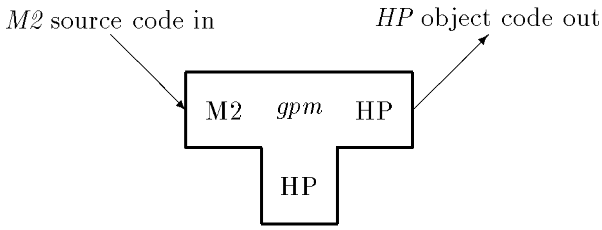
1.6.1 Introduction to T-diagrams

In the compiler literature it is usual to represent the process by which a compiler is modified to accept a new construct, and then modified again to *use* the new construct by means of “T” diagrams. These diagrams represent the *bootstrap* process by which a compiler written in its own language can be made to accept and use an enhanced language.

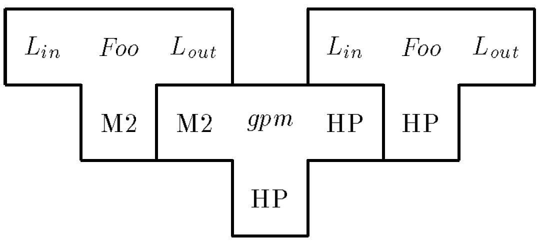
In "T" diagrams the left label is the language accepted as input. The right label is the language of output. The lower label is the language of implementation. The centre label is the program designation.

Example

The compiler *gpm* translates standard Modula-2 into HP object code, and is implemented in HP object code.

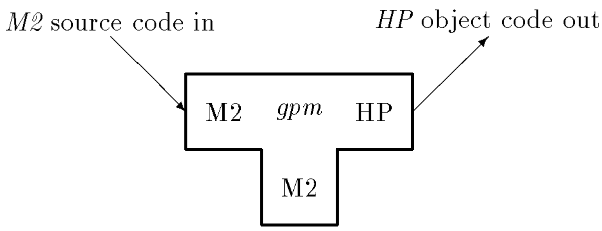


First we consider the situation where a compiler processes some arbitrary program. Later we consider the situation when a compiler compiles itself. Suppose that a program *Foo* translates some input language L_{in} into some output language L_{out}, and the program is implemented in Modula-2. If this program is compiled using *gpm*, the compiled version of the program performs the same translation, but is implemented in HP object code. We represent the compilation of this program thus—

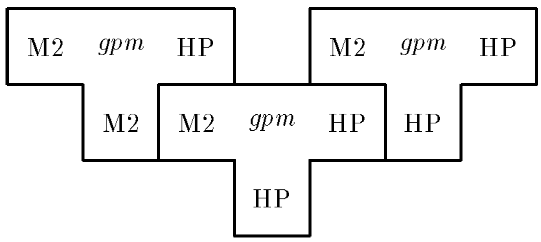


Note that the matching of languages on the ends of the "T" of the compiler. Compilation leaves the input-output languges of the T-diagram of its input unchanged, but changes the language of implementation. This is the basis of all correct translation— the meaning of the program is invariant, but the language of implementation is changed.

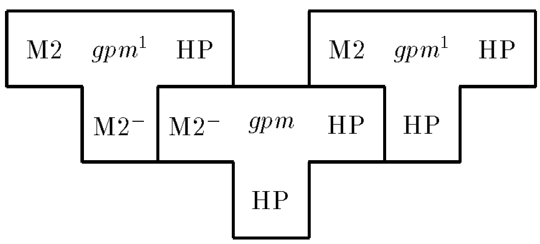
Now, *gpm* is itself written in Modula-2, and whenever it is modified or corrected it may be used to recompile itself. Here is the source code written in M2 and translating from M2 to HP.



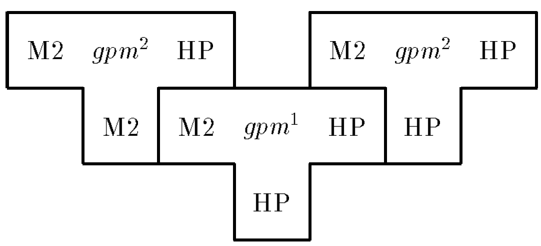
This recompilation process may be indicated as follows —



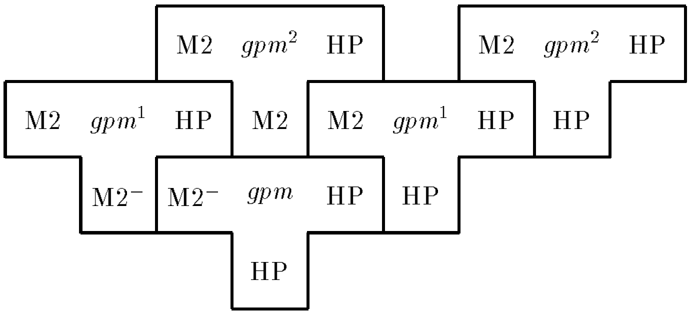
Suppose now that some error is found in the source code of *gpm*. In effect, we find that the current compiler does not correctly translate M2, but some subset $\mathrm{M2}^-$. The compiler source is corrected so that it does compile the complete language and must now be recompiled. Note that the corrected source gpm^1 must now be recompiled with the defective compiler to produce a correct compiler version. Of course the corrected source must not *use* that part of the language which the current compiler incorrectly translates or the following will not work.



Finally, if it is wished to make use of the now correctly translated construct in the source of the compiler, the compiler may be modified yet again (to gpm^2 say) and then recompiled once again.



This whole process may be represented as follows —

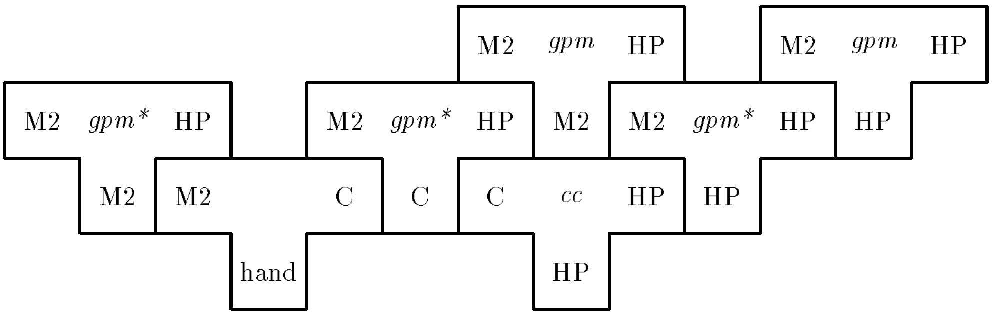


This is all very well. It is easy to see how that language extensions may be added in this gradual fashion, but where does the first version come from? There are basically two methods which may be used.

1.6.2 The hand-translation bootstrap

The compiler may be initially *hand translated* into some other language for which there is a compiler on the target machine. Usually the compiler is written in a simple subset of the language. Note that the translation does not have to be very efficient, just correct.

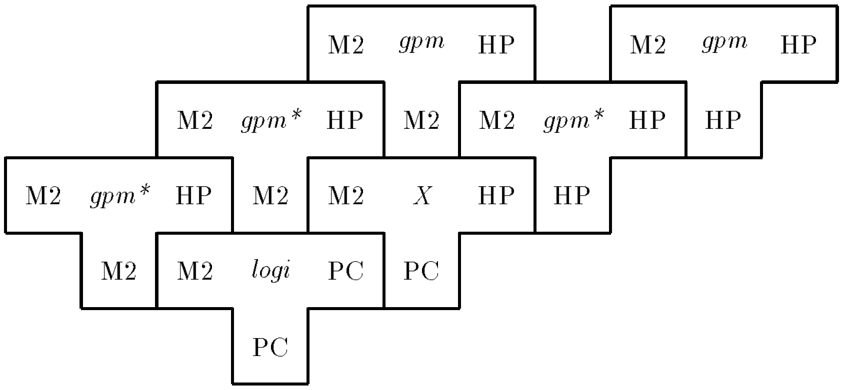
Suppose this path is followed via language C.



The subset compiler *gpm** is hand translated to C, and this source is compiled using the standard C compiler *cc* on an HP machine. The resulting executable produces HP object code, and is hosted on the HP machine. This version has all the limitations of the simplified source code version of the original compiler, together with any inefficiencies of the crude hand translation. This compiler is now used to compile the complete, original source of the compiler *gpm*, obtaining the final compiler at the right of the diagram.

1.6.3 The cross-bootstrap

In this next case it is assumed that the given language is available on a different machine. Suppose we have a compiler *logi* which accepts a dialect of M2 and runs on a PC. We proceed as follows.



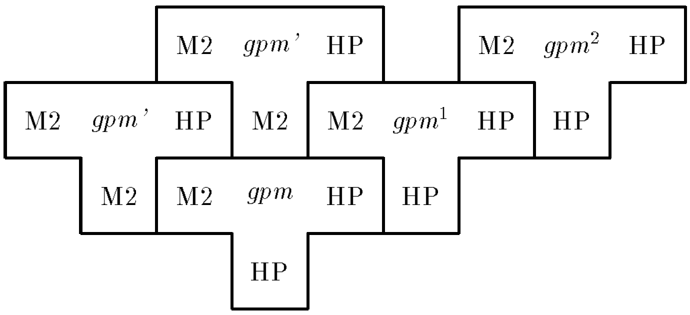
We must first check that the source of our new compiler *gpm* is written in the dialect of M2 which the PC compiler accepts. In the most general case the source may need to be modified to meet this condition. It may also need to be partitioned into small enough files that the PC can successfully compile it. We shall call this modified source *gpm**. Now, as the first step we compile the new source *gpm** using *logi*, obtaining the compiler marked X in the diagram. This is a cross compiler which compiles M2 to HP, but runs on a PC. We now pass the modified source of the compiler through compiler X, obtaining the HP-object files of the new compiler. These files must be carried onto an HP machine, either on a floppy disk or over a network, and linked on the target machine to produce an executable version of the compiler. The compiler can then be bootstrapped once again on the target machine.

1.6.4 Why write a compiler in its own language

Firstly, a compiler is a relatively complex program, and a check that it can repeatedly recompile itself is a good first check for correctness. Although the test that a compiler will compile itself does not substitute for thorough regression testing, it is a useful sanity check. Suppose that some minor modification has been made to the source code of a compiler. Before any further steps are taken it would be usual to compile the new code once, using the old compiler. The new compiler would then be used to compile itself, and the second generation compiler used to compile itself yet again. The linked executables of the second and third generation should be byte-for-byte identical. If this test is passed, then proper regression testing should be started.

Another reason often advanced for writing a compiler in its own language is the double benefit which is obtained when an improvement is made to the source code. Suppose, for example, that some improvement is made to the algorithms for code generation for a compiler written in its own language. We shall suppose that the improvement results in the production of faster object code. When the new source is compiled, the new compiler will produce faster object code for all of the programs which it compiles. If the new compiler now recompiles itself to produce a second generation of the modified compiler, it will produce the same faster object code as the first generation. However, since the second generation is a program compiled using the improved algorithm, it will itself run faster than the first generation of the new compiler.

Having given these two advantages for writing a compiler in its own language, it is only fair to report that there is also a significant disadvantage. This has to do with the difficulty of debugging the compiler in the event that a bootstrap fails. Suppose that we have the following sequence of compilations.



We shall suppose that we have a modified version of *gpm* which we shall call *gpm'*. We assume that the new source has some small and subtle error in it. We compile the new source using the existing version of *gpm*, obtaining the first generation compiler gpm^1. This compiler is actually incorrect, but appears to function correctly. It compiles itself without error, to produce the compiler marked gpm^2 in the diagram. The new compiler fails to function at all, crashing when it attempts even to compile *hello world*[1].

How is such a situation to be diagnosed? Looking at the point in gpm^2*'s* source at which it crashes is not directly useful. The crash point is not the place where the error is. The error is in the code which *translates* the construct which causes the program crash. This level of indirection in the cause and effect is a real challenge to the compiler writer trying to find the error in the source code.

To take a specific example, suppose that the source has code which incorrectly calculates the storage size of dynamically allocated records in the programs which it translates. The compiler gpm^1 will compile programs without crashing or giving any incorrect error message. The compiler gpm^1 does not incorrectly allocate its own storage, since it uses the correct algorithm of the compiler *gpm* which compiled it. However, all programs which the new compiler compiles

[1]For some reason, the first program compiled by any new compiler is almost always the canonical simplest useful program, which simply prints out "hello, world".

will incorrectly allocate storage. In particular, the second generation compiler *gpm*^{2} may crash in the middle of a memory allocation step when it finds its free list corrupted. The reasoning steps that need to be followed to diagnose this problem involve noticing that the program crashed during memory allocation, so the previous allocation step may have corrupted the free list. The corrupted free list probably resulted from an incorrectly sized allocation, so the code which computes object sizes may be incorrect.

More difficult still would be a case where the algorithm for computing object sizes is correct, but there is an error in some other code which is only exercised when translating some part of this correct algorithm. In this case, the problem would show up on the third, rather than the second generation.

1.7 Code Improvement

There are a number of places in the compilation process where improvements may be made in the code which is produced by our simple algorithms, at the cost of additional processing during the compilation itself. These code improvements may have as their objective either the reduction of memory use in the running program, or the improvement in speed of the final program. In recent years the emphasis has moved from the former to the latter, for most systems.

As an historical tradition, all such methods of code improvement are called *Optimizations*, even though almost none of the techniques actually guarantee optimality in the formal sense. The dotted boxes in our reference architecture figure 1.1 show the positions at which these so-called optimizations may be computed. In most cases, there is a choice in the position at which a particular optimization may be computed.

1.7.1 Safety and profitability

There are two important properties which any optimization technique should have. These are *safety* and *profitability*.

> **Definition:** An optimization is said to be **safe** if for every correct program the optimized code has equivalent input-output behaviour as the same program compiled without the optimization. In particular, the two compiled program versions must fail for exactly the same input data sets.
>
> **Definition:** An optimization is said to be **profitable** if the transformed program is an improvement over the original with respect to the objective of the optimization.

The underlying notion which the safety property tries to encapsulate is that correct programs will produce sensibly identical results when compiled with or without the optimization. It seems to be an overspecification to demand that incorrect programs also produce identical output after optimization. For example, suppose an incorrect program produces some output, and then fails with an array index violation. It may be the case that an optimized version of the same program crashes at a different point, perhaps earlier in the output sequence. What we would like to ensure is that the optimized program does not fail to detect the erroneous behaviour, and that the error message is still meaningful.

The notion of equivalent output cannot be sensibly applied to incorrect programs, as a simple example will illustrate. Suppose a particular language explicitly allows procedure parameters to be evaluated in any order, but a program uses parameter evaluations with side-effects, and relies on the parameters being evaluated left to right say. The program is thus incorrect. As a result of this false assumption by the programmer it may be that the optimized and unoptimized versions of this program assign differing values to a particular variable. If this value is used in a division operation, one version may fail with a divide-by-zero error and the other not. The failing version could be either the optimized or the unoptimized version.

This example shows that it is not possible to demand that optimized and unoptimized versions of a program fail on precisely the same sets of input data, if the program is incorrect.

Profitability is also a somewhat tricky concept. We might like to insist that *every* program is improved by the optimization, but this may be too demanding. Many optimizations improve *almost all* programs, but it is possible to find pathological examples for which the optimization actually makes things worse. It is thus sometimes necessary to demand some kind of probabilistic improvement in the program, rather than an absolute improvement in every case.

1.7.2 Space and time

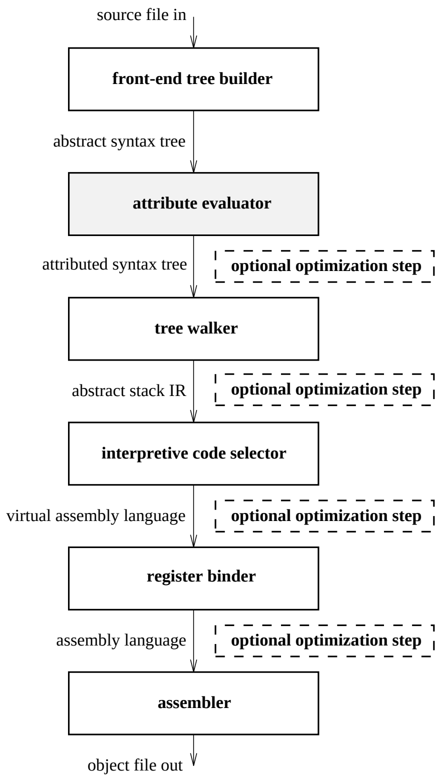
Optimization is always carried out with respect to some objective. The objective is invariably some weighted combination of program size and execution speed. It should be noted however that contemporary technology does not allow us to exactly predict the change in speed that a program will experience if any particular change is made to the code. When considering the effect of some particular code transformation, it is normal to assume that reducing the number of machine cycles which are executed in a particular computation will increase the speed of the program. This is *usually* the case, but there are difficult to predict effects which intrude if the change in the number of instructions affects the efficiency of the hardware cache, or the demand-paged memory management system.

In the past a reduction in the size of the executing program was sometimes the critical objective, and it was tolerable to allow speed to decrease to obtain a significant decrease in size. Using such an objective, for example, one might consider replacing the dispatch table for a *CASE* or *switch* statement with a sparse array structure. The vectored jump operation would then depend on some kind of search operation on the new structure, trading time for space.

In contemporary practice, speed is almost invariably weighted very heavily in the objective function for optimization. Thus it is common to trade off space in order to reduce the number of instructions which are executed for a particular computation. It is important to remember however, that extreme cases of program size increase may actually slow the program down, even if the number of instructions executed is reduced. This might happen, for example, if the increase in size causes the transformed program to exhibit seriously degraded cache behaviour.

Chapter 2

Attribute Evaluation



2.1 Abstract syntax trees

We shall assume that the text of the program source has been translated into an abstract syntax tree. The techniques which are involved in such tree-building are not treated here.

In order for a program to be correct, it must fulfill a number of conditions which are specified by the rules of the language in which it is encoded. Some of these rules are readily specified by context free syntax rules, such as the familiar *extended BNF*. Other rules are not able to be specified by context free grammars, and are arbitrarily referred to as *semantic rules*. These semantic rules contain conditions which must be satisfied if the program is to be well-formed. Some of these conditions are able to be checked by static analysis of the program, while others are only able to be checked by actually executing the program. We refer to these two kinds of rules as *static semantic rules*, and *dynamic semantic rules*.

The syntax tree, in its most rudimentary form, encodes information which corresponds to the abstract syntax of the source program. If the parser has been able to create a syntax tree according to the syntactic rules of the language, then we know that the source program is syntactically correct. However, before we can generate code for the source, it is necessary to perform the *static semantic checks*. These checks relate to certain relationships between the attributes of the various nodes present in the tree. The *dynamic semantic checks* can only be made at runtime.

As well as checking for correctness, we require certain attributes of the program to be evaluated before we can generate code. Some of these attributes arise naturally as a by-product of name-binding and type checking, while others have to be evaluated especially. For example, for each variable which appears in an expression we need to know whether the variable is local or static, so that we may generate the right kind of instruction sequence to fetch the value.

2.1.1 Defining abstract syntax

There are a number of formalisms which describe the structure and attributes of abstract syntax trees. These formalisms give us an agreed vocabulary to describe the essential aspects of the structure, without having to specify the details of implementation and data representation. We refer to *abstract* syntax, precisely because such details have been abstracted away. The formalism which we use here is *interface definition language*, (*IDL*), so called from its intended use in defining interchange forms between software tools.

IDL represents programs by trees. These trees are defined in terms of three kinds of entities: *classes, nodes* and *attributes*. A class is an aggregate type, with a finite set of alternative forms. Each of these alternative forms is a node of the class. Attributes are the components of the aggregate.

If it is helpful to think of a concrete representation of such structures, then in a *Pascal*-like language we have the following approximate correspondences. Trees correspond to pointer-linked, dynamically-allocated structures, with classes corresponding to variant record types. Nodes correspond to individual variants of the type, while each attribute would correspond to a field of the variant record type.

If *IDL* is implemented in an object oriented language, then classes correspond to single-level class hierarchies in the language. With such an implementation, each *IDL* class is implemented as an abstract classes, with each node corresponding to an immediate subclass of a single, parent class. In this case attributes are just instance variables of these subclasses.

Of course, *IDL* is an abstract notation, so that it is not necessary to implement the trees by means of pointer-linked, dynamic structures. For tool interchange, for example, *IDL* trees might be implemented in a flattened form in a file. Nevertheless, we shall find it convenient to represent trees diagramatically as though they were implemented as pointer-linked structures.

In *IDL*, classes are defined as an alternation of nodes. The syntax of the definition is —

Class-definition	→	*class-id* '`::=`' *node-id* { '|' *node-id* }';' .
Node-definition	→	*node-id* '`==>`' Attribute-def { ',' Attribute-def } ';' .
Attribute-def	→	*attr-id* ':' Type .
Type	→	*class-id* | '`seqOf`' *class-id* | *externally-defined-type* .

in the convention adopted here, classes have upper case names, nodes have mixed case names which begin with an upper case alphabetic character, while attribute names begin with a lower case alphabetic character. Attributes have a type which may be an externally defined type, such as *INTEGER*, or may be a class or a sequence of some class type. In the case that a class has a single member only, we may use the node identifier instead of the class identifier to denote the type of an attribute.

Consider the following fragment of *IDL*, which describes the structure of the *Modula-2 CASE* statement. A graphical representation of an implementation of the *IDL* is shown in figure 2.1.

```
STMT     ::= ... | CaseStmt | ...
CaseStmt ==> selector   : EXPR,
             cases      : seqOf Case,
             default    : seqOf STMT;
Case     ==> ranges     : seqOf Range,
             statements : seqOf STMT;
Range    ==> lower      : EXPR,
             upper      : EXPR;
```

One of the nodes belonging to the *STMT* class is *CaseStmt*. This node has three structural attributes: a selector, which belongs to the *EXPR* class, a sequence of *Case* nodes, and a default statement sequence.

Each *Case* node consists of a sequence of *Range*, and a sequence of *STMT*. A range consists of a lower and upper node of *EXPR* class. Of course, a range may consist of a single expression, in which case the upper expression has a distinguished **nil** value. The concrete syntax of *Modula-2* restricts some of the sequences to be non-empty, while others are permitted to be empty[1]. In section 2.4 we consider the additional attributes which need to be evaluated for this structure.

[1]To be specific, in *Modula-2* statement sequences and case sequences may be empty, but the range sequence for each case must contain at least one range.

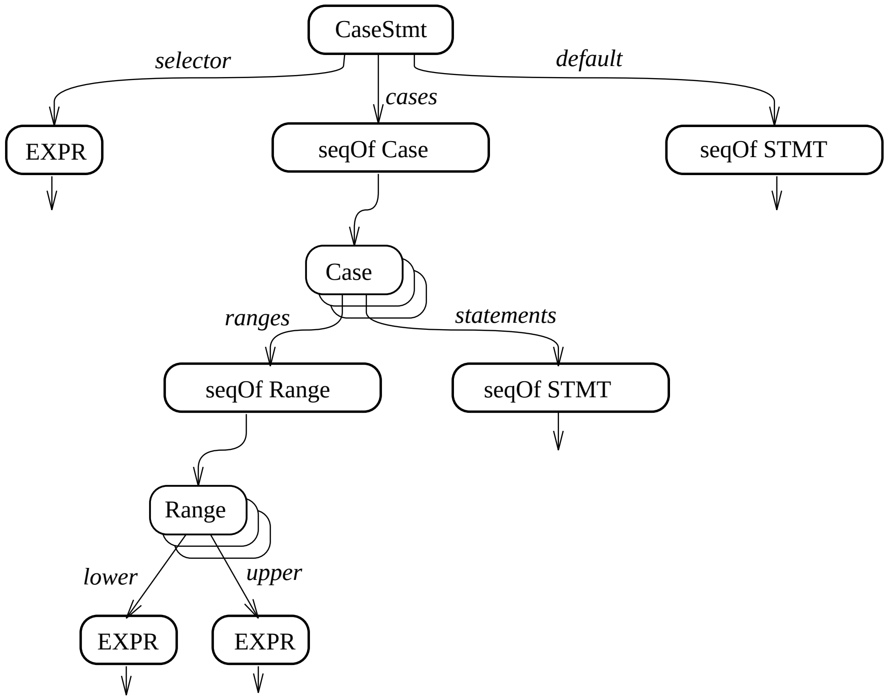


Figure 2.1: Abstract syntax representation of the *Modula-2 CASE* statement

2.2 Representing declared entities

Every declared entity in a program has an abstract representation which forms part of the abstract syntax of the language. For example we might define an abstract syntax for *Modula-2* for which a fragment of the *IDL* might appear as

```
ID_DESCRIPTOR ::= VarIdDesc | TypeIdDesc | ConstIdDesc
                | fieldIdDesc | ...
TypeIdDesc    ==> name     : IdOrd,
                  typType  : TY_DESCRIPTOR;
VarIdDesc     ==> name     : IdOrd,
                  varType  : TY_DESCRIPTOR,
                  offset   : INTEGER;   -- for local vars
ConstIdDesc   ==> name     : IdOrd,
                  conType  : TY_DESCRIPTOR,
                  conValue : LITERAL;
LITERAL       ::= ZZlit | RRlit | StrLit | ...
```

All declarations are represented by an *identifier descriptor*, with a separate node of the class representing variable declarations, constant declarations and so on. The type information for the descriptor is held in a separate structure, a node of the *type descriptor* class.

When the source is parsed, the types may be simple type denotations which are resolved to some actual type descriptor during name-binding. In the case of type definitions, the type descriptors themselves are constructed.

2.2.1 Representing types

Most of our languages of interest have a small number of primitive types, and type constructors which allow the definition of new types. These types are constructed from existing types, so that programs may construct directed graphs of types. Apart from the fact that the built-in types appear at the leaves, these type graphs have almost arbitrary structure.

A fragment of the abstract syntax for types in *Modula-2* might be

```
TY_DESCRIPTOR ::= TyNameDesc | ArrayDesc | RecordDesc | PointerDesc | ...
TyNameDesc    ==> name        : IdOrd;    -- all that is known so far
ArrayDesc     ==> name        : IdOrd,    -- possibly anonymous
                  size        : INTEGER,
                  indexType   : TY_DESCRIPTOR,
                  elementType : TY_DESCRIPTOR;
RecordDesc    ==> name        : IdOrd,    -- possibly anonymous
                  size        : INTEGER,
                  fieldSeq    : seqOf ID_DESCRIPTOR;
PointerDesc   ==> name        : IdOrd,    -- possibly anonymous
                  size        : INTEGER,
                  targetType  : TY_DESCRIPTOR;
```

The type descriptor for a named type has no attributes other than the name. Array descriptors have two attributes which denote the index type and the element type of the array. Both of these attributes are themselves type descriptors. Record descriptors have a sequence of field identifiers which are identifier descriptors.

It should be noted that there is no attempt made to ensure that the *AST* can only represent correct type declarations. For example, the *elementType* field of an array descriptor can legally be any node of the type descriptor class. It is a quite separate attribution task to check the validity of the declarations, and throw out impossible cases such as non-ordinal types used as array index types.

If all that has been done is to parse the declarations, then we shall still have a tree. It is after the names have been bound that the full graph structure becomes apparent.

Suppose we parse the type declarations —

```
TYPE Tree     = POINTER TO TreeNode;
     TreeNode = RECORD
                  key   : INTEGER;
                  left  : Tree;
                  right : Tree;
                END;
```

Figures 2.2 and 2.3 show the *AST* following parsing, and after the names have been bound. In these figures, the type descriptors have been shown shaded,

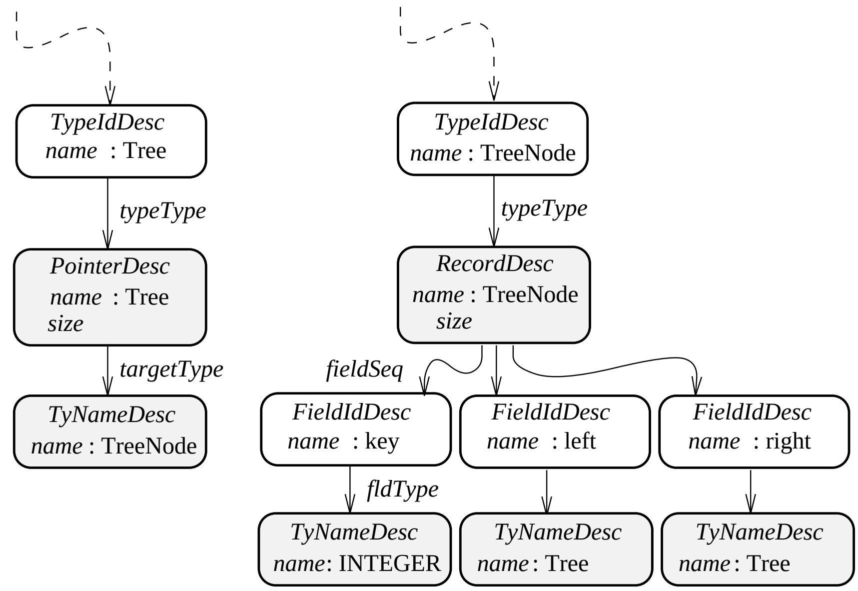


Figure 2.2: Abstract syntax trees of type declarations, before name binding

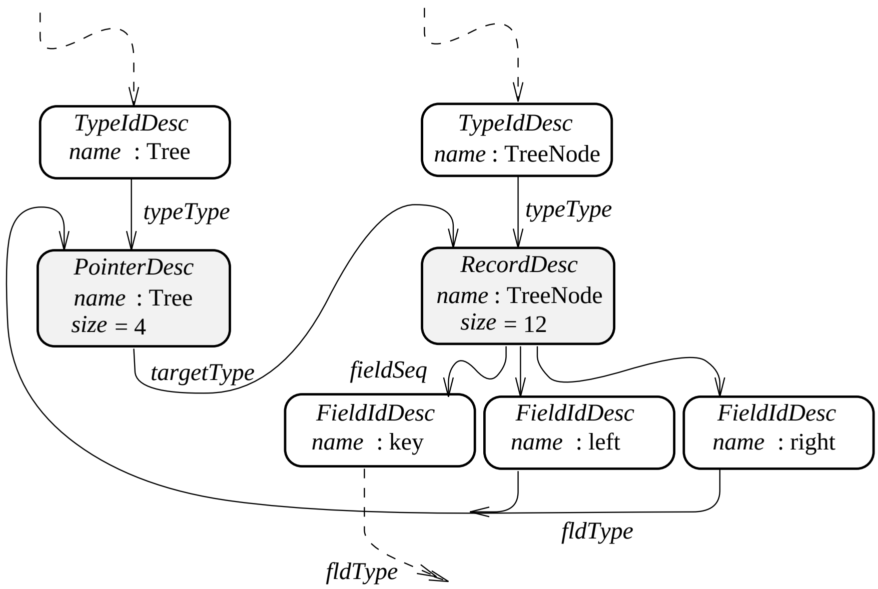


Figure 2.3: Abstract syntax trees after name binding, showing type graph

while the identifier descriptors are shown unshaded. Initially all occurrences of explicit typenames appear as references to *TyNameDesc* nodes. After name binding these attributes refer to the actual type descriptors which are bound to that name in the current scope. The type descriptor for the predeclared type *INTEGER* is not shown in figure 2.3.

In the case of languages which insist on *declaration before use in a declaration*, the name binding can take place as the tree is built. *Modula-2* insists on declaration before use in a declaration, except for pointer types. Thus for the declarations in the figures, the name *TreeNode* cannot be bound during parsing, so a place-keeper equivalent to the *TyNameDesc* node must be placed in the *AST*. When all of the declarations of the current scope are complete, then the name binding of the unknown types may be performed, so as to eliminate all such place-keeper nodes. By constrast, when the record declaration in the example is parsed the types of all fields are known, and may be bound immediately.

2.2.2 Representing scopes

In conventional block structured languages the scopes of a program form a tree-like structure. In languages such as *Modula-2* and *Ada* the situation is more complex, due to the explict control over visibility of names.

In principle, if a language possesses semantics which make it possible to perform all name binding during parsing, then scopes need not be preserved as a part of the *AST* representation of the program. In constrast, if further name binding may need to be performed during the attribution passes over the tree, then some persistent representation of each scope is necessary.

Most programs generate a large number of scopes, with each scope containing only a few names. A few scopes however will contain a substantial number of names. These characteristics are constraints on the data structure implementations which may sensibly be used. There are many different ways of solving this problem, so here we deal with just a representative example.

Since the same identifier may appear in several scopes, a global mapping from the character string representation of names to a unique ordinal is sensible. Names are mapped to their ordinal during scanning of the program, so that the laborious, per-character processing is performed only once for each name occurrence. Some kind of hash table provides fast processing to perform the mapping. The unique ordinal could be derived either from the hash-bucket number (for closed hash tables, for which the key is unique), or the string table "spelling" index at which the single copy of the name is stored.

It is possible to use the same mechanism for every instance of identifier lookup. If we define a *scope* node type in our *AST*, then we may define attributes of this type in record type descriptors, procedure descriptors, and imported module identifier descriptors. We may then use the same *Lookup* operation on this abstract data type to bind dot-qualified record field names, bind variable names, and find the names of imported objects specified in qualified form.

The scope type might be implemented several ways. Given the low number of average entries, it is even possible to consider linear lists. However, binary trees seem a somewhat more natural choice. If this option is chosen, it is important to ensure that the tree does not become unbalanced by entries being inserted in key order — for example, this choice would appear to rule out using spelling table index as a key, as insertions would usually appear in strictly ascending key order.

In any case, the ability to manipulate scopes as entire entities brings flexibility to the name binding process. Consider binding a name in a *Pascal* program, within a **with** statement. The order of name searching of the scopes will be: the scope of the fieldnames of the type of the with-selected record(s), the local variable scope and then successively more global scopes out to the predeclared scope. In this case, entry into the scope of a with statement simply adds the record type-descriptor fieldname scope to the head of the search list, and exit from the scope unlinks the record type-descriptor scope.

2.3 Static semantic checking

There are two main categories of static semantic attributes which we shall take as examples of the types of computation which need to be done as part of static semantic checking. Firstly there is the matching of declarations and uses, pervasive in all those languages which require the explicit declaration of named entities.

Secondly, for all those languages which are *strongly typed*, it is possible to compute an unique type for every datum, and for every expression. Typical semantic rules specify the allowable relationships between types of actual and formal parameters, and between the left and right-hand-sides of assignments.

There is considerable variation between languages in the complexity of the computation required to determine the type of expressions and designators. In some languages the type of an expression is easily computed from the declared type of every leaf in the expression tree taken separately, together with some simple transformational rules. *Modula-2* is a particularly simple example.

In other languages, the type of an expression can only be deduced from the declared types of the leaves, together with consideration of the contexts in which subexpressions occur. Features such as the possibility of overloading user-defined names, or the existence of implicit conversions considerably add to the complexity of semantic analysis for compilers and for human readers alike.

In object oriented languages, static semantic analysis poses slightly different problems because of the existence of limited forms of polymorphism[2]. In such cases the type rules tend to involve partial order relations between classes, rather than the simple equalities required by strongly typed languages.

In any case, almost all languages provide for some degree of operator overloading, at least for the built-in operators. For example in *Pascal* the plus sign can denote any one of: integer addition, real number addition, or set union. The problem of determining the meaning of an operator denotation, from the possible types of the operand expressions is called the *operator identification problem*.

[2]Polymorph — a species having more than one form.

2.4 Attribute evaluation

There are three different kinds of attributes which are known to *IDL*. There are the *structural attributes*, which represent the structure of the tree, as shown in the example in figure 2.1. There are also *semantic attributes*, which are evaluated according to the semantic rules of the language. Type information in an expression tree, is an easy example. Finally there are *code attributes* which are needed to enable code to be generated.

Attributes for the case statement

As an example of such *code attributes* consider the generation of code for figure 2.1. We wish to implement the case statement by means of a jump table. In order to do this, we need to know the lowest and highest values of the selector expression specified in any case range. Values which fall outside these bounds will select the default case, while values between these limits will vector through the table, as described on page 120. We would like to decorate the case statement node with two extra attributes —

```
CaseStmt ==> minRange   : INTEGER,
             maxRange   : INTEGER;
```

These attributes may be evaluated during a traversal of the sequence of range-expressions.

As an example of *semantic attributes*, for the case statement in *Modula-2* the rules insist that every expression in a range must be constant. These constant expressions must be compatible with the selector expression, which must be of some ordinal type. In every range, the upper constant must be greater than or equal to the lower constant. Finally all of the values specified or included in the ranges belonging to any one case statement must be distinct.

2.4.1 Attribute grammars

The evaluation of attributes is modelled by a formalism which describes a general method of computation on trees. This formalism is called an *attribute grammar*. Attribute grammars decompose global computations on trees, by specifying computations which are performed on local contexts. A local context consists of a single node, together with its children in the tree. Each such context corresponds to a part of the tree which corresponds to a single production-rule application in the context free grammar.

Inherited and synthesized attributes

Attributes are attached to syntactic categories of a context free grammar. We denote the b attribute of (some instance of) a syntactic category A as $A.b$, by analogy with the notation for record fields. In the event that a particular symbol appears more than once in a production, we add subscripts to occurences, numbering from left to right.

Attributes in an attribute grammar may be evaluated by two different kinds of local rule. We may have a rule which evaluates an attribute of the left-hand-side symbol in terms of attributes of the right-hand-side symbols. Alternatively, we may have a rule which evaluates an attribute of a right-hand-side symbol in terms of attributes of the left-hand-side symbol and other right-hand-side symbols. We call these two kinds of attributes *synthesized*, and *inherited* respectively.

Figure 2.4 illustrates the dependency for the two kinds of attribute.

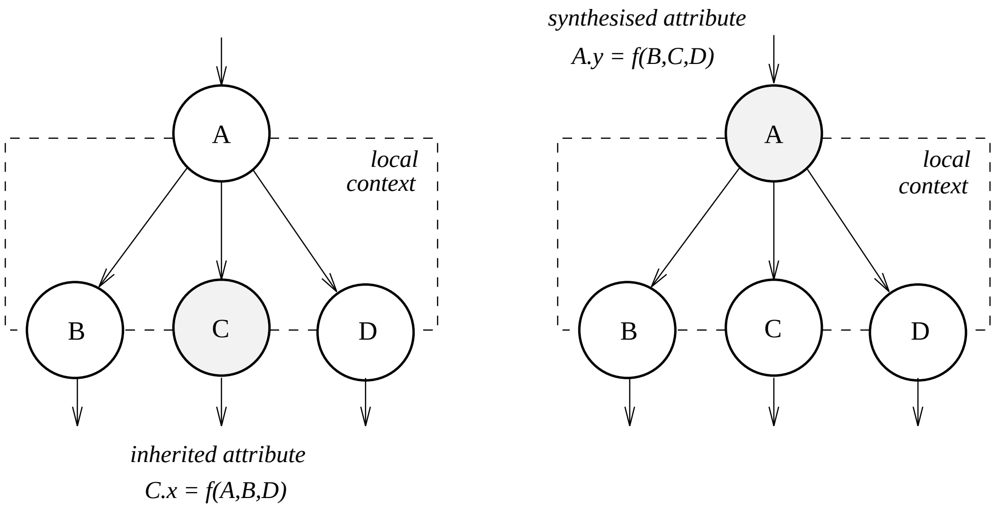


Figure 2.4: Inherited and synthesized attributes of a production $A \rightarrow BCD$

Inherited attributes are so called, because the value of the attribute is some function of information which is passed down the tree from the parent. Synthesized attributes are computed in the lower part of the local context, as some function of attributes of the child nodes. They are then passed up the tree to the parent node.

There is one final kind of attribute which is neither computed from information in the parent nor from information in the descendants. These are called *intrinsic* attributes. Such attributes apply to leaf nodes in a tree, and have values which are defined by the properties of the node itself. An example would be a leaf node in a constant expression-tree. The *value* attribute of a constant expression is synthesized for each node from the *value* attributes of the children. However, for leaf nodes, *value* is an intrinsic attribute which is placed in the node by the scanner and parser.

As an example of an inherited attribute, consider implementation of the *EXIT* statement in *Modula-2*. This statement causes control to jump past the end of the closest enclosing *LOOP* statement. Every loop statement needs an unique exit label to be assigned to it, and this exit-label will be inherited by all (recursively) nested statements other than loop statements. The following table sets out the attribution required for a fragment of the abstract statement grammar. Attribute equations are shown in braces at the right of the production to which they apply.

Stmt	$\rightarrow$	IfStmt	{ *IfStmt.exitLabel* = *Stmt.exitLabel* }
	$\mid$	WhileStmt	{ *WhileStmt.exitLabel* = *Stmt.exitLabel* }
	$\mid$	...	
	$\mid$	LoopStmt	{ *LoopStmt.exitLabel* = NewLabel() }
LoopStmt	$\rightarrow$	'`LOOP`' StatSeq '`END`'	{ *StatSeq.exitLabel* = *LoopStmt.exitLabel* }
StatSeq	$\rightarrow$	Stmt	{ *Stmt.exitLabel* = *StatSeq.exitLabel* }
	$\mid$	Stmt StatSeq	{ *Stmt.exitLabel* = $StatSeq_1$*.exitLabel* ; $StatSeq_2$*.exitLabel* = $StatSeq_1$*.exitLabel* }

Except for the loop statement, in every case the *exitLabel* attribute is inherited by a copy operation from the parent in the parse tree. In the case of the loop statement, we will assume that the label value is obtained from some external mechanism.[3]

The computation of the *value* attribute in constant expression trees is a simple example of the evaluation of synthesized attributes. As an example, consider a simple grammar with just two operators, "`+`" and "`*`".

Expr	$\rightarrow$	Expr '`+`' Term	{ $Expr_1.value = Expr_2.value + Term.value$ }
	$\mid$	Term	{ $Expr.value = Term.value$ }
Term	$\rightarrow$	Term '`*`' Factor	{ $Term_1.value = Term_2.value \times Factor.value$ }
	$\mid$	Factor	{ $Term.value = Factor.value$ }
Factor	$\rightarrow$	*number*	{ $Factor.value = number.value$ }
	$\mid$	'(' Expr ')'	{ $Factor.value = Expr.value$ }

In every case the value attribute of the left-hand-side symbol is computed from the values of the children. Note the use of the intrinsic attribute of the *number* leaf nodes, and the use of subscripts to disambiguate the repeated symbols.

2.4.2 Implementing attribute evaluation

Attributes are evaluated during one or more traversals of the *AST*. During a recursive traversal of the tree, we carry inherited attributes down the tree as the recursion advances, and carry synthesized attribute values back up the tree, as the recursion returns.

It is not possible to compute attributes for an arbitrary attribute grammar, since circular dependences of various kinds are a fatal circumstance. In fact, in general we would like to place a small bound on the number of separate traversals which are required to evaluate all attributes. Later we discuss tools which are

[3] If one wished to be a purist, and avoid the use of apparently external mechanisms such as calls to a label allocator, it is possible to allocate unique labels using local attribute computations alone. In effect, other "chain" attributes pass the highest allocated label number around the tree. This is best avoided, except on sytems which provide automatic attribution from specification.

able to create attribute evaluators from attribute grammar specifications, but for the moment we discuss hand-crafted evaluators.

As an example, we shall take a simple problem, that of allocating location offsets to local block variables in a block structured language. We also wish to compute the total space required for all the variables in a procedure.

An abstract syntax which models the situation is —

```
STMT     ::= ... | Compound | ...
ProcBody ==> declPart  : seqOf VarDecl, -- or empty
             stmtPart  : seqOf STMT,
             totalSize : INTEGER;       -- synthesized
Compound ==> declPart  : seqOf VarDecl, -- or empty
             stmtPart  : seqOf STMT;
VarDecl  ==> name      : IdType,
             size      : INTEGER,       -- intrinsic
             offset    : INTEGER;       -- inherited
```

Procedure bodies consist of a possibly empty declarative part, and a sequence of statements. These statements may include compound statements which in turn have declarative parts. The procedure body has a *total size* attribute which is synthesized. Each variable declaration has *name* and *size* attributes, which we shall assume are computed during tree-building. In a more general case, variables might also have an alignment constraint, but we shall ignore that here. We wish to compute the *offset* of each variable. These are inherited attributes.

Space has to be allocated for the variables of a block as the block is entered, and is reclaimed as the block is left. Looking ahead, figure 3.9 on page 52 illustrates an example of a procedure with blocks, and the storage use at various points in the procedure.

We shall traverse the trees defined by the above abstract syntax with a recursive procedure which will visit each statement node in turn, visiting sequences in left-to-right order. Inherited attributes will be passed as value parameters of the recursive procedure, while synthesized attributes will be returned in variable mode parameters. This is a general rule.

In order to evaluate the attributes defined in the *IDL*, we shall need to introduce some additional attributes of the nodes. In compound and procBody nodes, we shall introduce an inherited integer *hiTide* attribute, which holds the current allocated size. The traversal procedure is given in figure 2.5. Note that the inherited attribute *hiTide* is passed to the value parameter *tide*, while the synthesized attribute *totalSize* is returned by the variable parameter *max*.

The additional attributes which are introduced in the figure are rather specialized, in that they are simply temporary working values which are not required further once the traversal is complete. In fact all of the attributes computed in this figure can be computed in a single top-down, left-to-right traversal of the *AST*. Attributes with this property are said to be *L-attributed*. Such attributes may be computed on-the-fly, as the tree is constructed by a top-down recursive descent parser.

In *Modula-2* the evaluation of the type attribute for expression trees is almost trivial, as commented earlier. Type is a synthesized attribute of expression

```
PROCEDURE VisitProc(proc : ProcBody);

  PROCEDURE VisitStmt(stmt : STMT; tide: INTEGER; VAR max : INTEGER);
    VAR decl : VarDecl; next : STMT;
  BEGIN
    CASE statement type of stmt OF
    | ...
    | compound :
        FOR decl := every VarDecl in stmt^.declPart DO
          decl^.offset := tide; INC(tide,decl^.varType^.size);
        END;
        IF tide > max THEN max := tide END;
        FOR next := every STMT in stmt^.stmtPart DO
          VisitStmt(next,tide,max);
        END;
    END;
  END VisitStmt;

  VAR hiTide : INTEGER;  decl : VarDecl; next : STMT;

BEGIN (* VisitProc *)
  hiTide := 0;
  FOR decl := every VarDecl in proc^.declPart DO
    decl^.offset := hiTide; INC(hiTide,decl^.varType^.size);
  END;
  proc^.totalSize := hiTide; (* initialize *)
  FOR next := every STMT in proc^.stmtPart DO
    VisitStmt(next,hiTide,proc^.totalSize);
  END;
END VisitProc;
```

Figure 2.5: Evaluating offsets and size in block structured languages

nodes, and may be computed in a single pass over the tree. We shall consider a more general problem in section 2.5.

2.4.3 Attribute evaluation from specifications

`<< still to come >>`

2.5 Operator identification

In some languages the problem of identifying the meaning of an overloaded operator requires some relatively complex attribute evaluation. In order to motivate this topic, consider the following fragment of *Modula-2* —

```
VAR a,b,c,d : SHORTINT; (* 16-bit integers *)
  ...
  a := b + c + d;
```

Now, consider that some machines have instructions which are able to add 8, 16, or 32-bit integers. The question is, does the '+' sign mean short integer arithmetic, or does it mean 32-bit arithmetic, or do we have a choice?

Of course, there is no logical way to decide such a fine point of semantics. Unless the language standard says the meaning is implementation defined, the meaning must be as specified by the relevant standard. In this case the draft international standard for *Modula-2* (DIS 10514, section 6.4.1) says —

> A value of a subrange type is also a value of the host type of that subrange type, and the type of expressions that are value designators, or function calls, is defined in terms of the host type of any subrange type which might otherwise apply ...

In *Modula-2* there is only one integer type, with all other signed types being subranges of this *host type*. Expressions must thus be evaluated at full *INTEGER* width, with the result being range-checked against the declared range of the destination type.

Note that this choice in the language has some clear advantages, since the example expression will still be able to be computed, even if `(b+c)` is outside of the short integer range, but `(b+c+d)` is not. This choice of semantics thus avoids intermediate overflow errors. However, the choice is not without cost on some machine architectures. Since we are only permitted subranges of a single signed type, it is necessary to perform *all* arithmetic at the width of the widest supported type. On machines for which operations on the widest type attract some speed penalty, this choice of semantics may lead to a significant loss of performance.

The same example in language *C* has rather different properties. In *ANSI C* the *usual unary conversions* apply to each operand in such expressions, and if the types of the operands are different, then the *usual binary conversions* are then applied. In this example, only the unary conversions apply, and convert the `shortint` data to `int`, giving the same semantics as *Modula-2*, but without the overflow or range checks.

2.5.1 Automatic coercions

A *value coercion* is an operation by which a value of one type is converted to another type. Automatic coercions are those which are invoked by the compiler, without any explicit denotation in the source text. The implicit widening of subranges to their host type in *Modula-2* is a trivial example. The usual conversions of *C* are a more interesting case, since they involve representation changes between whole number and floating point types.

The possible coercions in *C* are shown in figure 2.6, where the unary conversions are shown as dotted lines, and the solid lines are the binary conversions.[4]

[4] In fact, this diagram glosses over some of the fine print in the *ANSI* standard. The figure is correct only if `sizeof(short int) < sizeof(int)` etc.

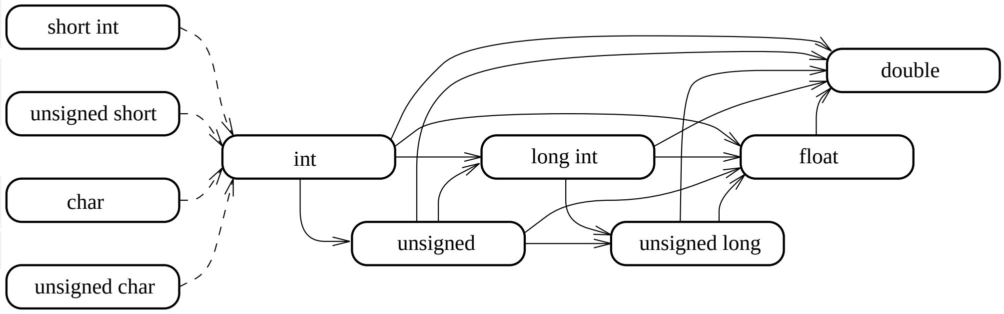


Figure 2.6: Usual unary and binary coversions in *ANSI C*

The binary conversions are used to try to make the types of the two operands of any binary operator the same. These conversions form a *coercion graph*. In general, such conversions should be value-preserving, or give a closest approximation in some defined sense. The conversions in figure 2.6 are not all value-preserving. In particular, for machines in which *long int* is 32-bits and *float* is single-precision *IEEE* format the conversion of large integers may round as much as 8-bits of significance from the exact value. Also, in *C* the conversion of negative integer values to unsigned types silently produces a result which is congruent to the exact value modulo 2^n, where n is the number of bits in the unsigned representation. In *Modula-2* the same conversion can never be implicit, but requires the use of the standard function *ORD* or *VAL*, and generates a runtime range check.

A different type of implicit conversion occurs when a value of one type is assigned to a variable of another type. These same rules apply to substitution of actual parameters to formals of another type. In *C* a value of *any* numeric type may be assigned to a variable of any other numeric type, again without any explicit denotation in the source code. In conversions from floating to integral types, the value is rounded towards zero, while for narrowing conversions the high significance bits are simply discarded. Note that these assignment conversions do not appear in the figure 2.6, and if present would make the graph cyclic.

All these rules, taken together, allow us to deduce the implicit coercions which take place in the assignment in the following program fragment —

```
int       a;
long int  b;
short int c;
double    d;
  ...
  a = b + c + d; /* tree says ((b+c)+d) */
```

The *AST* before and after attribution is shown in figure 2.7. In this figure the nodes labelled *lval* yield an *l-value*, that is, the address of the variable. The nodes labelled *rval* yield an *r-value*, that is, the value of the variable. In the right-hand figure, typed operator nodes are labelled with a *D, I, L, S* to indicate *double, integer, long integer, short integer* respectively. The coercion nodes are shown explicitly and have labels which indicate the conversion required. The short

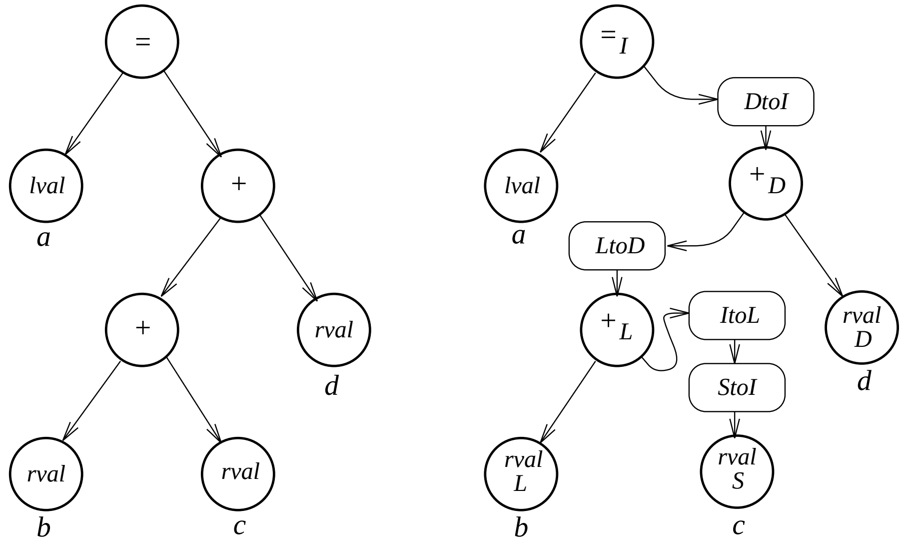


Figure 2.7: Abstract expression tree, and code tree showing explicit coercions

integer c is first converted to integer as required by the usual unary conversions, and then is converted to long integer to match the left operand of the lower addition node.

The top conversion in the figure is different to all others. It is an assignment conversion, which in this case does not correspond to any of the edges in figure 2.6.

A lattice coercion model

Examples of implicit coercion which are semantically well behaved require some restriction on the form of the implicit coercion graph. In particular, we would like all implicit conversions to be entirely value-preserving, and to induce a partial order on the types. The partial order relation $T_1 \subseteq T_2$ corresponds to the predicate *the values of* T_1 *are included in* T_2.

If for every pair of types there is some unique type which is the least upper bound of the types, then we say the types form a *lattice*. For any such pair of types the *least upper bound* is the least type to which both values may be converted without loss of accuracy or range. A reasonable semantic rule for such a system would be: for every binary operator, convert each operand to the least upper bound of the pair. Down conversions only take place over assignment, or by an explicit conversion function.

Figure 2.8 is a possible coercion model for an *Oberon-2*variant with tiny, short, integer and long integers, and short, real and long reals. The figure is value preserving for machines with 8, 16, 32 and 64-bit integers, and single, double and long double *IEEE* floating point reals. In this case, the addition of a 32-bit integer and a single precision floating point value would require both to be coerced to double precision. If the value is ultimately assigned to (say) a

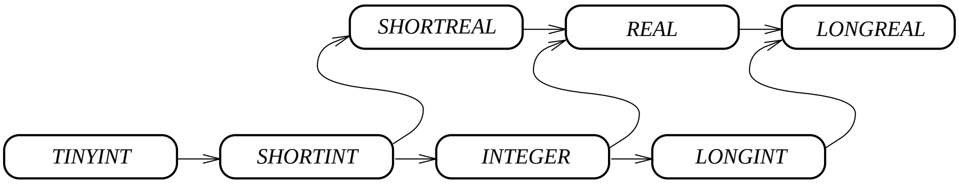


Figure 2.8: Value-preserving coercion lattice, for an *Oberon-2*variant

short real, then all loss of accuracy takes place at the assignment, rather than at intermediate stages.

An important characteristics of such value-preserving types lattices is that conversions may be done in one or more steps without affecting the final result. Thus if a 8-bit integer needs to be converted to a long double, and the machine architecture can do this transformation in one step, then we shall get the same result as if we follow the lattice step by step.

The *C* type coercion graph in figure 2.6 is a lattice with respect to the partial order induced by the possible coercions. However these coercions are not value preserving.

Resolution of overloading

In all of the cases considered so far, the need for conversions is discovered as a result of local attribute evaluation. However, there are cases where the need for coercions, or even the meaning of an operator, is only apparent after attribution of the complete tree. This happens in language *Ada*, when overloaded functions and operators are declared. How are we to identify the operators in such a case? We have somehow to find a unique, self-consistent identification which matches operand and result types, possibly with the assistance of implicit coercions.

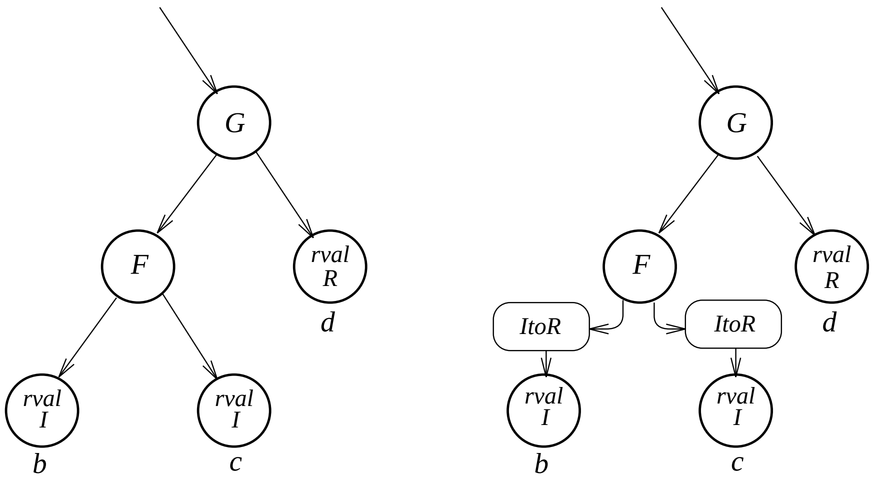


Figure 2.9: *AST* with overloaded operator, and resolved code tree

Suppose that we have a program which declares a binary operator G, and two different operators named F, with signatures

$$\begin{aligned} G &: R \times T_1 \rightarrow B \\ F &: R \times R \rightarrow T_1 \\ F &: I \times I \rightarrow T_2 \end{aligned}$$

where R, I and B are reals, integers and Boolean respectively, and T_1 and T_2 are some user defined types. The *AST* in figure 2.9 must be attributed as shown on the right of the figure, since this is the only consistent, type-correct interpretation of the tree.

All overloading problems of this kind may be solved in cases where the coercion graph is a lattice, by a two-pass attribution. We traverse the tree twice. On the first pass we compute a synthesised attribute which is the set of all possible types which the subtree might have, as a result of the allowed coercions, and all the matching choices of operators. If we reach the root of the tree with a set containing more than one type, then the tree is ambiguous. If the *possible type set* is a singleton, then we propagate an inherited, *demanded type* attribute down the tree, so as to identify each operator. In the example in the figure, the demanded types of the two operands of G are T_1 and R, thus selecting the first of the two F functions. This choice of meaning for F makes R the demanded types of the two operands of F, necessitating the coercions shown on the right.

This two-pass attribution can be used to solve problems of the same general kind, in which multiple interpretations are possible. In such cases, we must have a *cost function* associated with each rewriting step, and we seek to choose the lowest cost rewriting of the tree. This least-cost tree rewriting technique is seldom necessary for operator identification, but forms the basis of the most powerful method of instruction selection currently known. Chapter 6 has the details.[5]

All of the computations necessary for operator identification may be automatically produced from specifications. Such a tool forms part of the *Eli* compiler writing environment.

2.6 Rewriting the tree

As well as attributing the tree, it is usually necessary to rewrite the tree to take into account the result of evaluation. For example, in languages such as *Modula-2*, it is normal to fold the evaluation of constant expressions at compile time. In such cases, a possibly complex expression tree may be replaced by a single, literal leaf-node.

[5]If we look at figure 2.7 from this new perspective, we see that there are two consistent operator identifications. We could have identified both of the additions as floating point double operations, and coerced the values of b and c directly to double. As it happens, only the identification given in the figure is consistent with the C standard, which specifies the usual binary conversions on only one operand of a binary operator.

It is possible to adopt differing approaches to this rewriting. At one extreme, some schools of compiler construction suggest that a completely new and separate *code-tree* be constructed. At the other extreme, we might generate code by traversing the original *AST*, and restrict ourselves to simply annotating the *AST* in order to facilitate this. In the examples given in this chapter, we have assumed that additional nodes will be introduced into the *AST* as necessary, and that whole trees may be replaced by equivalent single nodes.

In either case, the objective of the rewriting step is to produce a tree which makes the code-generating traversal as simple as possible. We may think of the rewriting as some kind of transformation into a canonical form. Two small examples from *Modula-2* may help to make this clear.

Making standard functions canonical

Some of the built-in procedures and functions of *Modula-2* have alternate forms. For example *INC* and *DEC* have one-parameter and two parameter forms. Since we must check the actual parameters for correctness during type checking, it simplifies the subsequent code emission traversal if the single parameter form is transformed into the more general two-parameter form.

Essentially the same considerations apply in the case of obsolete syntax forms which are replaced by some preferred form. An example in *Modula-2* is the transformation of the obsolete *type transfer function* into the preferred form using *SYSTEM.CAST*.

Syntactically, the built-in functions and procedures appear as ordinary procedure invocations. The identity of the nodes is not known until after name binding has been completed. However, since these procedures usually do not give rise to a procedure call at runtime, it makes sense to replace the original procedure-call nodes by another node type. Since we have to make the separation between built-in procedures and others during static semantic analysis, it makes sense to not have to do so again during code generation.

Removing syntactic sugar

The replacement of the single parameter form of *INC* by the more general two-parameter form may be viewed as the removal of a layer of syntactic sugar, placed in the language for convenience without adding expressivity. Other examples exist also.

All of the explicit type conversions in *ISO Modula-2* are special cases of the *VAL* function. Thus we may transform all these special cases away, and leave only a single, parameterized coercion function in the tree for code generation.

The transformation of multi-dimensional array type declarations from the comma-separated list form to the *ARRAY OF ARRAY OF* form also falls in the category of removal of syntactic sugar. In this case however we can create the canonical form directly during tree construction.

2.7 Finding out more

`<< citations for idl, DIS10514, Eli, Oberon2, etc >>`

2.8 Exercises

2.1

2.2

Chapter 3

Runtime Organization

3.1 The runtime environment

In almost all cases, our compilers do not generate all of the code which is needed to implement any particular source program. Except for some specialized embedded systems, the linked object code does not interface to the bare hardware, but rather to a virtual machine consisting of the hardware, together with facilities provided by an operating system.

It is rather difficult to generalize about the runtime environment within which our programs are expected to work, since there is a great diversity. Different operating systems provide different facilities, and different levels of program protection, and the facilities required to support different languages vary somewhat also. In this chapter, in order to be concrete, we shall focus on a typical environment which has an underlying operating system with facilities similar to *UNIX*. Except for explicit digressions, we shall also concentrate on the facilities required for the support of procedural languages of the *ALGOL* family, that is, *Pascal, Modula-2, Ada, C* and related *object oriented languages*.

The operating system typically provides facilities for input and output, access to some kind of file system, and perhaps most important, resource management. The operating system is usually responsible for protecting the various memory segments from invalid access, and for protecting the memory belonging to one process from other processes.

Except for a few degenerate systems, such as *MS-DOS*, the operating system is able to exercise control over resources with the help of hardware mechanisms. The most common mechanism is the implementation of a multi-level machine in the hardware. A typical such machine has multiple operating states, *user state* and one or more *privileged states*. Certain instructions may only be executed in privileged states, and cause a trap to be taken if an attempt is made to execute them in user state. Furthermore the mapping between virtual addresses and physical addresses is modified during the change of state, so that it is possible to ensure that memory belonging to the operating system is inaccessible from code executing in user state. Such systems can provide protection for various regions of memory, usually on a *per-page* basis, where a page is of the order of

512 to 8192 bytes, depending on the particular system. Pages may have *read, write* and *execute* permissions.

The nature of the resource management provided by the runtime environment is an important parameter of our code generation, however a detailed discussion of this subject belongs properly in a text on operating systems theory, rather than a book on code generation.

3.1.1 Memory organization

Programs compiled from sources in procedural languages divide their memory use within a number of disjoint categories. These separate categories are chosen on the basis of the required lifetime of the data, and the required visibility of the data object.

UNIX systems provide four different *memory segments* as they are called. The *TEXT segment* is read-only, and contains the code of the user program. This protection is important, as it prevents erroneous data accesses from corrupting the code during its execution.

The *DATA segment* contains data, the content of which is specified prior to the start of the program. This segment is therefore suitable for constants and for initialized data. In some *UNIX* implementations there are two separate data segments, one of which has a read-only attribute, and is protected by the hardware's memory management, and another with the read-write attribute, which is used for initialized data. It is helpful if our code generators can preserve the distinction between read-only and read-write data declarations for these cases.

The *BSS segment* is used for uninitialized data, and is usually able to be varied in size during program execution. This is a read-write segment. Most systems clear the allocated memory to all zero bytes, during the allocation. The name, for obscure reasons, is an initialism for *Block Started by Symbol.* This segment is used for variables which are of static extent, and is also used for the *heap*, for languages which provide for dynamically allocated objects. Heap management is not treated here.

Finally, the *STACK segment* contains uninitialized data, and is used for the runtime stack of user programs. This is a read-write segment. Primitive systems without paging hardware may require the size of the stack segment to be specified at link-edit time, but it is usual for the stack to be allocated some minimal default size, and to expand automatically as the growth of the stack causes page faults to occur. Almost all contemporary systems have the stack segment growing from higher to lower addresses, that is, the stack is inverted. The *Hewlett-Packard Precision Architecture* is almost unique in that the stack grows upward in the address space.

Scope and extent

Scope and extent are the two attributes which control the visibility and lifetime of data in our languages. In some languages the two attributes are intertwined in such a way that the programmer may not control the two separately.

By *scope* we mean the textual range of program statements and declarations over which a declared name is associated with a particular object. By *extent* we mean the temporal range, at runtime, throughout which a declared datum is bound to a particular memory location.

Roughly speaking, there are four possible extents which a datum might have.

Automatic: a datum of automatic extent is allocated storage at the invocation of a procedure or function, or entry to a block. The space is automatically reclaimed when the procedure is completed, or control leaves the block

Static: a datum of static extent is allocated storage at the start of execution, and remains until the program terminates

Explicit: a datum of explicit extent is allocated storage as a result of a call to some allocation routine, such as *NEW* or `malloc`. The storage is explicitly deallocated by a corresponding call to *DISPOSE* or `free`

Persistent: a datum of persistent extent has a lifetime which persists beyond termination of the program. None of the common procedural languages support any persistent data, other than files

The scope of name declarations is usually delimited by some kind of enclosing syntactic structure in the source program. For example, in *Pascal* variables declared inside a procedure have a scope which is the whole of the text of the procedure in which they are declared. In *C* the scope of a variable name declared within a function extends from the point of declaration to the end of the function.[1] Procedural languages generally provide some or all of the following scope delimiting structures.

Block: a name of block scope is visible within the block in which it is declared

Procedure: a name of procedure scope is visible within the procedure in which it is declared

File: a name of file scope is visible within the whole of the file or module in which it is declared

Global: a name of global scope is visible throughout the whole of the program in which it occurs. For most languages, modules other than the declaring module must explicitly import the global name in order to make it visible

Apart from explicitly allocated objects, for which the responsibility is entirely with the programmer, the scope rules place a constraint on the extent of declared data. It is an obvious rule that the lifetime of a named datum must extend for at least as long as control remains within the control structure which delimits the scope. Thus data of block scope may have an automatic extent which is shorter than that of the surrounding procedure. Similarly, file scope and global scope data must have static extent.

[1]In both cases there is some fine print here, which has to do with declarations of other objects with the same name in nested scopes. These issues are discussed in detail later.

Within these constraints, the notions of scope and extent are independent, although as noted above, some languages do not allow independent specification. Figure 3.1, 3.2, and 3.3 show which combinations are available for three different languages. In all of these tables, the annotation *invalid combination* denotes combinations which would violate the constraint that extent must at least match scope. Other annotations are used for combinations which logically might exist but for which there is no syntax, or are forbidden by other language rules.

	Scope			
Extent	**Global scope**	**File scope**	**Procedure scope**	**Block scope**
Static	does not exist in *Pascal*	**var** declarations outside procedures	not possible in *Pascal*	*Pascal* does not have local blocks
Procedure-automatic	invalid combination	invalid combination	**var** declarations inside procedures	*Pascal* does not have local blocks
Block-automatic	invalid combination	invalid combination	invalid combination	*Pascal* does not have local blocks

Figure 3.1: Scope and extent for *Pascal*

In the case of *Pascal*, note that there are only two choices. Variables declared within procedures have procedure scope, and procedure-automatic extent. Variables declared outside of procedures have file scope, and static extent. Since standard *Pascal* programs are single, monolithic compilation units, the question of global scope does not arise, although most *Pascal* implementations provide some form of separate compilation as an extension, with explicit control over global visibility.

	Scope			
Extent	**Global scope**	**File scope**	**Procedure scope**	**Block scope**
Static	exported *VAR* declarations	non-exported *VAR* declarations	not possible in *Modula*	*Modula* does not have local blocks
Procedure-automatic	invalid combination	invalid combination	*VAR* declarations inside procedures	*Modula* does not have local blocks
Block-automatic	invalid combination	invalid combination	invalid combination	*Modula* does not have local blocks

Figure 3.2: Scope and extent for *Modula-2*

In *Modula-2*, the situation is slightly more flexible, as the import and export facilities allow control over the visibility of variables of static extent.

In language *C* there are many more possibilities. Variables declared outside of functions always have static extent, but may have global scope (the default) or be explicitly restricted to file scope. Variables declared within functions have function scope, and function-automatic extent as default. Function local variables may be explicitly given the static storage class, which allows them to preserve their values between invocations of the function to which they belong. Note that the use of this facility requires that static variables be given a predictable initial value, in this case zero.

Variables declared within a $\{ \cdots \}$ block have block scope, and block-automatic

	Scope			
Extent	**Global scope**	**File scope**	**Function scope**	**Block scope**
Static	default for file-level declarations	'`static`' file-level declarations	'`static`' declarations inside func's	'`static`' declarations inside blocks
Function-automatic	invalid combination	invalid combination	declaration default inside func's	not possible in *C*
Block-automatic	invalid combination	invalid combination	invalid combination	declaration default inside blocks

Figure 3.3: Scope and extent for *C*

extent. If they are declared with the static storage class then they have static extent.

Notice that in language *C* the "`static`" storage class specifier is ambiguous. Outside of functions its effect is to override the default visibility, reducing the *scope* from global to file-scope. Inside a function, the effect of the "`static`" storage class specfier is to override the default *extent*, changing that from automatic to static.

In *Modula-2*, variables of global scope are explicitly exported by the module in which they are declared, and become visible in other modules only after explicit import into those modules. In *C*, variables of global scope are implicitly exported by the absence of the "`static`" storage class specifier in the module in which they are declared. Other modules of the program can make these variables visible with an explicit "`extern`" declaration, or may use the "last resort" rule that any undeclared variable is assumed to be a global variable of type "`int`". Sensible *C* programmers do not use this implicit behaviour.

Storage management

In *Modula-2* we have symbolic constants, declared after the *CONST* keyword, and variables which the language does not guarantee are initialized. Programs may not refer to the address of a constant, nor pass it as the actual parameter to a formal parameter of *VAR* mode. Nevertheless, some constants do have an address at runtime, while others do not. Small integral constants such as 5, say, would never be placed in memory for any machine architecture which has instructions with immediate operands. All legal uses of such a constant would simply insert the constant as an immediate operand in the object code.

The situation is quite different in the case of structured constants such as character strings or the constant value constructors which appear in *ISO* draft standard *Modula-2*. In these cases an area of the data segment must be set aside for the constant, and suitably initialized. Note carefully that although legal source programs cannot mention the address of such a constant, the intermediate representation often manipulates the address. For example, the block copy operation which assigns a literal string to an array variable would typically need to be passed the address of the constant string. A correct code generator will never threaten the integrity of such a constant when the address is implicitly taken in this way. Nevertheless, it is sensible to allocate read-only storage for such constant data, for those machine architectures which provide such a facility.

In language *C* the situation is more complex. Originally *C* did not provide for manifest constants, and it was usual to use the facilities of the preprocessor to substitute literals for symbolic names.[2] Such macro literals do not have an address, and in the case of string literals each applied occurrence of the macro name may give rise to a new copy of the string at some new address.

With *ANSI C*, the situation is better resolved. File scope variable declarations may be given the "`const`" attribute, which gives some level of protection to the value. These "constant variables" are still data objects, and have a run-time address, which may even be referenced in the program. Conforming *C* compilers will check that no assignment is made to a datum declared with the const attribute, but cannot always ensure the integrity of values which have their addresses taken explicitly. It is therefore sensible to place all such constant variables in a read-only segment, if that is possible.

ANSI C also provides for scalar variables to be declared with the "`register`" storage class, or with the "`volatile`" attribute. Compilers are not obliged to place variables and parameters declared with register class in machine registers, and many compilers simply ignore such declarations. At best, such a storage class declaration is a hint to the compiler that the variable in question is likely to be heavily used. However, such a hint makes a nonsense of the notion of program portability, since the same code may be compiled on machines with very few registers, or with very many registers. In any case, an optimizing compiler can probably make a better choice of data to place in registers than most programmers.

The "`volatile`" marker has mandatory semantics. Such a declaration warns the compiler that the value may be changed without any apparent direct assignment in the source code. Such variables cannot take part in common subexpression elimination or value tracking optimizations. Every time the program calls for the value of such a variable to be used, the value *must* be read anew from memory. Consider a program which inspects a memory-mapped peripheral device register. Clearly such a variable cannot be used as part of a common subexpression, since its value may change without any action by the program. Similarly, programs with multiple threads and pre-emptive thread scheduling must treat variables which are shared between threads as volatile.

3.2 Procedure call and return

The main organizational principle for programs written in procedural languages is *procedural abstraction*. Procedures and functions are used to encapsulate complex operations, and are passed parameters to control their effect. We use the term "procedure" to mean either *pure procedures*, that is, procedures invoked solely for their side-effects, or to refer to value-returning functions.[3]

---

[2] Note the unfortunate effect that this has on program clarity, because of the different scope rules which apply to preprocessor declarations and data declarations.

[3] Note that in the *C* literature, all such routines are called "functions", with pure procedures being considered to be functions which return the "`void`" type.

In general, procedures do not encapsulate program state, although they may require local variables during their execution. Any such local data, together with parameter information and the information necessary to allow the procedure to return correctly, forms part of the procedure's *activation record*.

3.2.1 Register use conventions

Older machine architectures often provided machine registers with restricted or otherwise specialized purposes. Thus registers were reserved by the hardware as stack pointers, argument pointers, segment pointers, index registers, base registers and general purpose registers. It is rather more common in recent architectures to have *general register organization*, in which almost all the operand registers are able to be used for any computational purpose. Such a general register machine may allow registers to be used in arbitrary ways, but it is still necessary to have agreed conventions for register use. These rules are often called *software conventions* since they constrain the way that code is generated, but are not constraints which are inherent in the hardware architecture.

In order to see why these conventions are necessary, consider the separate compilation of modules, or even just the calling of library procedures. When compiling code for a procedure which calls a library routine, it is necessary to know which machine registers might be modified by the called routine, and which ones are guaranteed to have the same contents on their return. If parameters are passed to the called procedure in registers, it is necessary for caller and callee to agree on which registers to use, and the order in which the parameters appear in these registers. Return values from functions must also be passed in a register agreed between caller and callee.

As a first step, we distinguish between registers which are unchanged by procedure calls, and those which are possibly modified by such calls. We shall call registers which are threatened by procedure calls "*caller-saves registers*". Suppose that such a register happens to have a live[4] datum in it immediately before the call to some procedure. Since the called procedure is free to destroy the contents of the register it is the responsibility of the *caller* to save the value, and restore it after the call returns.

The other category of register, is those which are guaranteed to have their values preserved across the call and return of any procedure. We shall call these registers "*callee-saves registers*". If live values are in such registers at procedure calls, then the caller takes no action to save the values. However, in the called procedure, if it becomes necessary to use such a callee-saves register, then the called procedure must save the register itself, and restore the original value before returning to the caller.

To place all registers in either category is very inefficient. For example, if all registers were callee-saves, then a called procedure would have to save and subsequently restore however many registers were required to execute the procedure code. In many cases the saved registers would not contain live values, so the procedure entry and exit would carefully save and restore garbage values.

[4] By *live* we mean that the value in the register might be needed again after the call returns. Such values are said to be *live across the call*.

Having a convention in which all registers were caller-saves would not suffer from the same problem. At least on the calling side the code generator would be able to compute which registers contain live values, so garbage would not be saved and restored. However, making all registers caller-saves suffers from an equally silly inefficiency. The problem is that during code generation for the caller we are not aware of how many free registers the called procedure will actually need to use. Thus we may carefully save and restore values in order to free up registers which are never used by the callee.

It turns out that a partitioning of the register set between caller-saves and callee-saves, with approximately equal numbers in each category, is statistically better than either of the pure strategies mentioned before. The idea is as follows. During code generation our register allocator will try to keep values which are live across procedure calls in callee-saves registers. The caller-saves registers will be used for short-lived values which are not live across any call. In a called procedure, if only a few registers are needed, then the register allocator will use the caller-saves set first, since these are known to be free to use and modify at will. If it should turn out that more registers are needed, or the called procedure itself has values which are live across another procedure call, then some callee-saves registers must be freed.

For the kind of code generators that we consider here, and which buffer whole procedures until after register allocation has been performed, it is normal to save any callee-saves registers in the *procedure entry prolog.* This prolog performs the housekeeping associated with procedure entry. Similarly, the restoration of any callee-saves registers forms part of the *procedure exit epilog.* This epilog performs the housekeeping associated with the procedure exit.

3.2.2 Activation records

In languages which support the recursive calling of procedures each call of a procedure must have a new activation record allocated to it. Thus if recursion occurs, each invocation of a recursive procedure has a separate, private activation record. Languages which do not support recursion, such as *FORTRAN*, can allocate activation records statically. Even for recursive languages, it is possible to allocate most activation records statically, but it is not profitable to do so on *RISC* machines where access to static variables may be more costly than to variables in a local stack frame.

Logically, activation records contain several categories of data.

- procedure linkage information
- parameter data
- declared local variables
- workspace for the procedure

We deal with each of these categories in some detail later.

There is actually no reason that activation records must be allocated on a stack. Logically, at any point in execution of a program the activation records of all the currently active procedures form a linked list. At the head of list we have the activation record for the *currently executing procedure*. In the tail of the list we have the activation record of the *caller* of the current procedure, followed by the caller of the caller, and so on. If the current procedure calls another procedure, the activation record for that procedure is added to the list. If the current procedure returns, the head of the list is removed, the activation record of the caller becomes the head of the new list, and the caller becomes the currently active procedure.

However, since the addition and removal of activation records always takes place at the head of the list, it makes sense to specialize the list to a stack. Thus we speak of *stack frames* rather than activation records, almost as though using a stack was the only possible organization.

Explicitly linked activation records

There are two circumstances under which activation records may usefully or necessarily be implemented by means of an explicit linked list.

When a procedural language is implemented by tranlation to a lower level language, there are usually certain difficulties caused by the mismatch in the semantics of the source and target languages. Language *C* is almost the universal choice for target languages for this purpose, because of its wide availability, and the ease with which its type system may be breached. However, when *Pascal*-like languages are translated to *C* there is no simple way of expressing the scope rules of *Pascal* in *C* terms. In particular, in language *C* there is no facility for referring to the local data of lexically enclosing procedures, the so-called *uplevel data*. There are several tricks which may be used to circumvent this problem, and one of the simplest, conceptually, is to allocate activation records from the *heap* rather than use the language *C* stack. In this case uplevel activation records appear as explicit records in a linked list, and may be accessed by the normal *C* selection mechanisms.

The second circumstance under which explicitly linked activation records are found is in the implementation of those functional languages which provide for *continuations*. In these languages the strict order of procedure call and return is complicated by the possibility of suspending a procedure, and then reactivating it later. In such cases the activation records form a tree. In this tree, the currently active and suspended procedures are at the leaves, and the parent of each activation record is the record of its caller.

3.2.3 Stack organization

In the usual way of implementing procedural languages activation records are allocated in a *runtime stack*. When this is the case, we refer to the activation records as *stack frames*. The activation records exist in a separate memory segment, which typically has the benefit of some hardware support. In traditional machine architectures, one machine register was reserved for use as a *top-of-stack pointer*, (*sp*), and certain instructions use the stack pointer as an implicit operand.

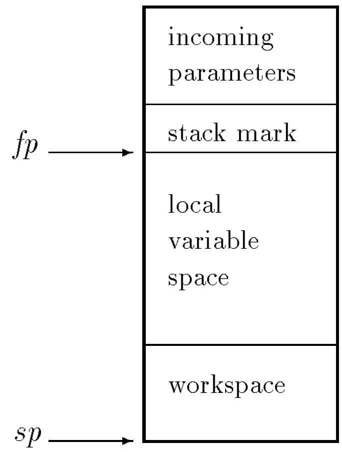


Figure 3.4: Stack frame with stack and frame pointers

Stack and frame pointers

The *Intel-386* is typical of these conventional architectures. The register "`esp`" (32-bit extended stack pointer) points to the top of the runtime stack, which grows downward in the address space.[5] The "`call`" and "`ret`" (return) instructions modify the stack pointer as side-effects of their execution, and "`push`" and "`pop`" instructions operate on this stack.

It is conventional to reserve a second hardware register as a pointer to a known location in the stack frame. This register is called the *frame pointer register*, (*fp*). In the *Intel-386* architecture, the "`ebp`" register is used as the frame pointer. In this architecture the top-of-stack may change during the execution of the procedure, for example, as procedure parameters are pushed, so the frame pointer provides a fixed reference in the frame throughout execution of the procedure.

The layout of stack frames, for such machine organizations is typically as shown in figure 3.4. There are four regions in the frame: the stack mark, which contains the linkage information, the incoming parameter area, the local variable area, and finally any workspace required by the procedure. The frame pointer register allows convenient access to the parameters and local data of the procedure. Parameters will have a positive offset from *fp* while local variables will have negative offsets.

The stack mark contains at least enough information to allow the procedure to return, restoring the stack state of the caller. For the *386* only two data are necessary: the address of the next instruction in the calling procedure, and the frame pointer value of the caller. For this machine the "`call` *proc-name*" instruction transfers control to the entry point of the nominated procedure, and

[5] Note the perversity of this convention. We call the limit of the stack "the top", despite the fact that in memory address terms it is actually the lowest address on the stack.

pushes the address of the instruction following the call (the *return address*) onto the stack. Figure 3.5 shows the stack state before, during and after procedure P calls procedure Q. In part (a) the stack frame of P is shown, with the frame pointer pointing to the stack mark, and the stack pointer to the 'top' of the stack.

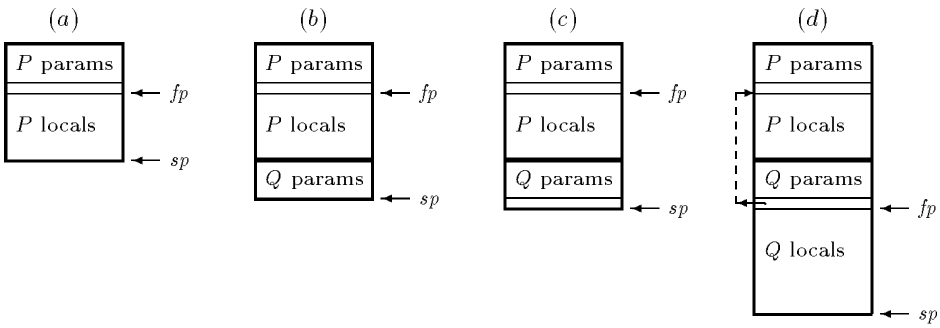


Figure 3.5: Procedure P calls procedure Q, with parameters

In part (b), the procedure P has pushed the parameters for Q onto the stack in preparation for the call. Note that the stack pointer has moved to accomodate the parameters, but the frame pointer is unchanged.

In part (c), the procedure P has executed the "`call`" instruction. The return address in the code of P is pushed onto the stack as a side-effect of the "`call`" instruction. Control has now passed to procedure Q, which proceeds to execute its *procedure entry prolog*.

The purpose of the prolog is to complete the creation of the stack mark and stack frame for Q. In this case the stack mark consists of just two data: the return address is already pushed, and a pointer to the previous stack mark must now be pushed also. Finally, the stack pointer must be adjusted downward, to make space for the local variables of Q, arriving at figure 3.5d. The *Intel-x86* assembler is —

```
Q:                                  ; start entry prolog of Q
        pushl   ebp                 ; push long frame pointer
        movl    esp,ebp             ; current sp becomes new fp
        subl    48,esp              ; make room for local vars
        save callee-saves registers, if needed
```

where we have assumed that in this example the total space required for variables and workspace is 48-bytes.

The value of the previous frame pointer, which is saved in the stack mark, is called the *dynamic link*. The linked list of stack marks which is thereby formed is called the *dynamic chain*.

The epilog with stack and frame pointers

The procedure return process retraces the steps of the call. The "`ret`" instruction pops the top-of-stack value into the *program counter* or *instruction pointer*, as the *Intel* documentation calls it. We must thus expose the return address by discarding the local variable space, and restoring the caller's frame pointer value. The instructions of the procedure exit epilog corresponding to the entry prolog example are —

```
xLab:                                  ; start exit epilog of Q
        restore callee-saves registers, if saved
        addl    48,esp                 ; remove local variable space
        popl    ebp                    ; restore caller's frame pointer
        ret                            ; pop return address and return
```

Finally, the used parameter values must be removed to get back to figure 3.5a. This may be done either by the returning procedure during the epilog, or by the caller after the call. We shall assume that the parameters are removed by the caller in this case. Typical code for the call of Q in procedure P might be —

```
        push Q's parameters onto the stack
        call    Q                      ; call the procedure
        addl    16,esp                 ; remove 16 bytes of parameters
```

where we have assumed that there were 16 bytes of parameters passed to Q in this example.

RISC stack frames

Typical *RISC* machine architectures encourage simplified call and return conventions, and pass at least the first few parameters in registers. Typically, they do not have instructions which perform side-effects, such as the adjustment of a stack pointer. Indeed, for many of these machines the use of particular registers as stack and frame pointers is only a matter of software convention, and is not "built-in" into the hardware architecture in any way. The details of parameter passing are given in the next section, however for the moment it suffices to note that non-leaf procedures, that is, procedure which call other procedures, must reserve a parameter assembly area on the top of their stack frame. This parameter assembly area must be large enough to accomodate the largest group of parameters which that particular procedure passes to any of its callees.

The *MIPS R3000* architecture is typical, and we shall use it as an example. A typical stack frame is shown in figure 3.6.

In this figure, the position of the stack mark is indicated by a *virtual frame pointer*. We would prefer not to waste a register with a second pointer into the stack frame, so if we can ensure that the size of the stack frame does not vary during procedure execution, we may address all data in the frame using offsets from the stack pointer.

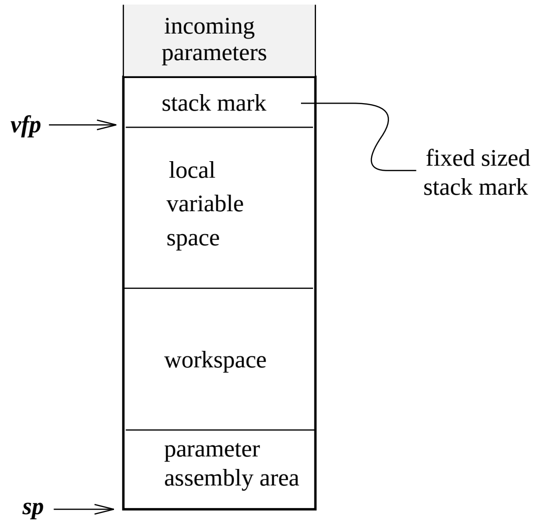


Figure 3.6: *RISC* stack frame with stack pointer and virtual frame pointer

It is convenient in such cases to refer to the *virtual frame pointer* in the *VAL*, since the final frame size is not known until after register allocation. The final translation to assembly language will make the transformation —

$$vfp \quad \rightarrow \quad (sp + \text{final-frame-size})$$

As it turns out, for the *MIPS* architecture, the stack mark is empty. The return address of the procedure call is moved into a general purpose register, and as we have just seen there is no frame pointer. Corresponding to figure 3.5, we have figure 3.7 in the *MIPS* case.

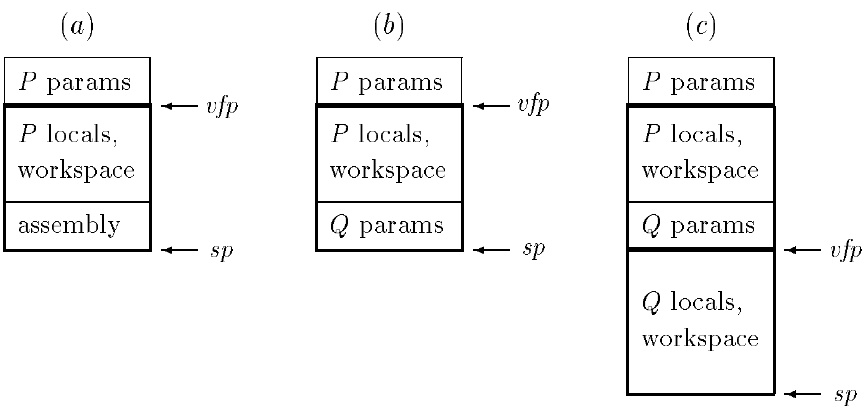


Figure 3.7: P calls Q, *MIPS* architecture version

In figure 3.7*a*, procedure P has a stack frame which consists of space for locals and workspace, and a parameter assembly area. P's own parameters are in the stack frame of P's caller.

In figure 3.7*b*, the parameters for Q have been placed into the parameter assembly area, and the "`jal`" (jump and link) instruction is executed. The return address is placed in r_{31}. The procedure prolog of Q moves the stack pointer to make room for P's locals and workspace, and for an outgoing parameter assembly area also if P is not a leaf procedure.

If P is a leaf procedure, the entry prolog is almost empty. There is nothing to do except to adjust the stack pointer. In fact, if P is a leaf procedure, has no local variables and does not run out of caller-saves registers during register allocation, then the stack frame is completely empty, and there is nothing to do in the prolog. In such a case, the epilog will simply be the return instruction —

```
xLab:                                 ; start exit epilog of Q
        jr      r31                   ; jump to return adr reg
```

In the more general, non-leaf case, or when a real stack frame does need to be constructed, the prolog must make space for the local variables, workspace and the outgoing parameter assembly area. Since any subsequent procedure call will destroy the return address in r_{31}, this register must be saved in the workspace, along with any other callee-saves registers which the procedure destroys. A typical prolog, in such a case might be —

```
Q:                                    ; start entry prolog of Q
        sub     sp,64,sp              ; make room for local vars
        save callee-saves regs, including r31, if needed
```

The procedure epilog reverses the above process. Any callee-saves registers are restored, the current stack frame is removed, and the procedure jumps to the return address —

```
        restore callee-saves regs, including r31, if saved
        add     sp,64,sp              ; remove stack frame
        jr      r31                   ; jump to return adr reg
```

Note that since the parameter assembly area is allocated as a fixed part of the stack frame, there is no code to remove the parameters after each procedure call.

Variable size stack marks

In some runtime conventions, registers are saved *inside* the stack frame, so that the size of the stack frame is not known until code generation is complete. In such cases, it is convenient to extend the concept of the virtual frame pointer one further step.

For such conventions, it is convenient to manipulate the virtual assembly language as though there are two pointers, one to each end of the stack mark. Parameters are accessed relative to the *virtual argument pointer* (*vap*), while

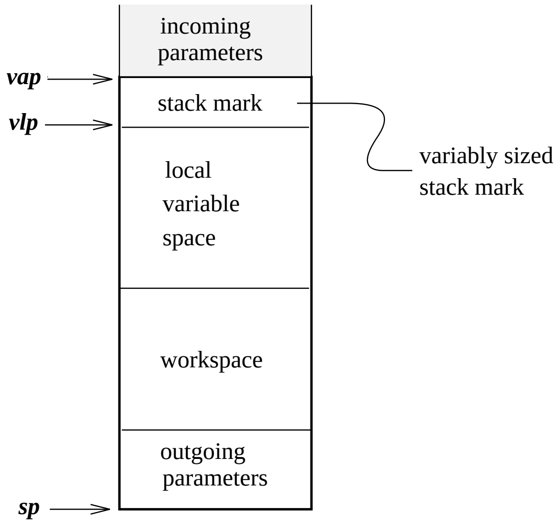


Figure 3.8: *RISC* stack frame with two virtual pointers

locals are accessed relative to the *virtual local pointer* (*vlp*). Figure 3.8 has the diagram which corresponds to figure 3.6 in this case.

Once code generation is complete, and the sizes of the frame and mark are known, references to the two virtual pointers are adjusted —

$$\begin{aligned} vlp &\rightarrow (sp + \text{final-frame-size}) \\ vap &\rightarrow (sp + \text{final-frame-size} + \text{final-mark-size}) \end{aligned}$$

Allocating space for block locals

Local blocks, as they appear in *C* and some other languages, allow for local variables to be declared within unnamed blocks. These blocks may be nested within each other, and provide for typical nested scope visibility rules. The semantics of such local variables are described by an abstraction in which the variables are automatically allocated when control enters the block, with the space being freed when the block is left. The effect of these semantics may be obtained without the complexity of actually extending and contracting the stack frame when such blocks are entered and left.

Since blocks which are not nested do not have variables of overlapping extent, it is possible to use the same space for the variables of non-overlapping blocks, as shown in figure 3.9. It is possible to find the smallest stack frame which will accomodate all block variables using a simple attribute evaluation. During traversal of the *AST* of the procedure, we evaluate two data. An inherited attribute carries the *currentFrameSize*, while a synthesised attribute *maxSize* tracks the maximum stack frame size.

Of course, it is not necessary to reuse the stack frame space in this way. For some backends, the task of code improvement will be simpler if only one

```
void Example(void)
{
    declare 24 bytes of variables
    {
        declare 20 bytes of variables
        {
            declare 24 bytes of variables
        }
    }
    {
        declare 32 bytes of variables
    }
}
```

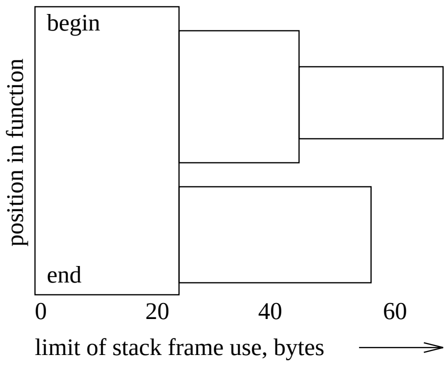


Figure 3.9: C function with local blocks, showing frame space allocation

logical variable is found at any particular offset. In such cases there is some benefit in not overlapping block variables, and simply allocating space to block variables just as if they were declared at the procedure level. Figure 3.10 shows a possible allocation, for the previous example, but without the overlapping of block variables.

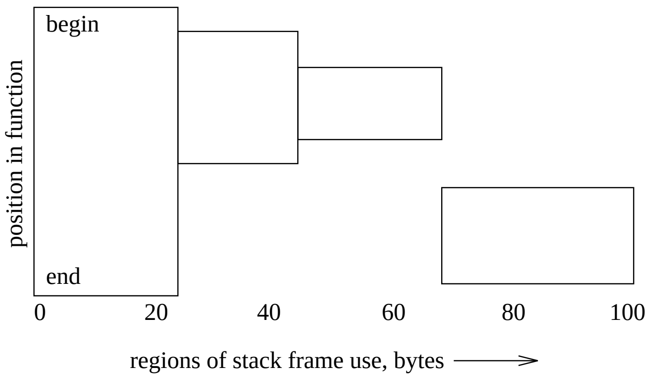


Figure 3.10: C function with local blocks, allocated with procedure-automatic extent

3.2.4 Parameter passing

There are two main methods by which parameters are passed to procedures. They may be pushed onto the runtime stack, or they may be assembled in fixed locations at the top of the caller's stack frame. We refer to these two methods as *stack-parameters*, and *fixed assembly*. Choosing a method has far-reaching consequences, since the way in which the front-end evaluates parameter expressions is distinctly different for the two cases. Finally, a variation of fixed assembly is used for those machines which, like *SUN microsystem's SPARC* architecuture, have *register windows*.

Parameter mode and structure

Within algol-family languages, there are several different parameter semantics. When an actual parameter is passed to a formal parameter which is specified as having *value mode semantics* the value of the expression is evaluated and passed to the formal. Within the called procedure, the formal is treated as an ordinary local variable and may be modified at will.

When an actual parameter is passed to a formal parameter which is specified as having *variable mode semantics*, the actual parameter must be a variable. Within the called procedure, the formal is treated as an ordinary variable, and may be read and written at will. However, after the return of the called procedure, the actual parameter variable will have the final value that was last assigned to it in the called procedure.

In language *Ada* formal parameters are specified as having *in, out* or *inout* semantics. The semantics of *in* mode parameters are not quite the same as value mode. The actual parameter expression is evaluated and passed to the formal. However, within the called procedure the formal is a local constant, and cannot be changed or even threatened. *Inout* mode has identical semantics to variable mode, while *out* mode carries the final value of the formal out to the actual variable, but does not bring the initial value of the actual into the procedure.

For simple scalar types and pointers, value parameters are always *passed by value*. That is to say, the expression is evaluated, and copied to the parameter location of the formal parameter. Variable mode parameters are usually *passed by reference*, in *Pascal*-style languages. When a parameter is passed by reference, the address of the actual variable is passed to the called procedure, creating an *alias* to the original variable. When code is generated to access such formal parameters, the compiler must perform an additional dereference of the pointer to reach the actual value.

In principle, variable mode parameters could be passed by copying into the formal at procedure invocation, and copying the final value out at the procedure return. The differences between the two methods only become apparent for pathological programs which, for example, pass the same variable to separate variable mode formal parameters. In such a case, if reference semantics are used then modifying one formal will change both, while if a copy-in, copy-out implementation is chosen then the two may be modified independently. Here is a *Modula-2* program which tests for the method of passing of variable mode parameters. Of course, *Modula-2* specifies passing by reference.

```
MODULE CopyOrRef;
  IMPORT InOut;
  VAR val : INTEGER;

  PROCEDURE Horrible(VAR a : INTEGER; VAR b : INTEGER);
  BEGIN
    INC(a); INC(b);
  END Horrible;

BEGIN
  val := 0;
  Horrible(val,val);
  IF    val = 1 THEN InOut.WriteString("Passed by copying");
  ELSIF val = 2 THEN InOut.WriteString("Passed by reference");
  ELSE               InOut.WriteString("Parameters are broken");
  END;
  InOut.WriteLn;
END CopyOrRef.
```

In *Ada* non-scalar parameters of any mode may be passed by reference, or by copying (copying in, out, or in and out, repsectively). *Ada* programs which depend on a particular method of parameter passing are incorrect.

The situation with structured parameters, either arrays or records, is somewhat different. In general, value mode parameters need to be copied by some kind of block copy operation, so that the formal may be modified without affecting the actual. However, the copying may be done either by the caller or by the callee. For arrays and records, there are at least three known variations.

Passed by reference, copied by the callee

In this case, the code of the caller passes a reference to the actual parameter, exactly as if the formal parameter was of variable mode. The callee then makes a copy as part of the procedure entry prolog. In this case, space for the copy is allocated as part of the local variable space of the called procedure. The called procedure can directly access the components of the local copy.

Copied by caller, passed by value

In this case, the caller copies the value into the reserved locations in the parameter area of the called procedure. As before, the called procedure can directly access the components of the formal parameter.

Copied by caller, passed by reference

With this convention, the caller copies the value into a temporary area in its own stack frame, and passes a reference to this temporary location to the called procedure. In this case, the called procedure must access the components of the formal indirectly, dereferencing the pointer which the caller placed in the parameter area. *SUN Microsystems C* compilers pass structures using this method.

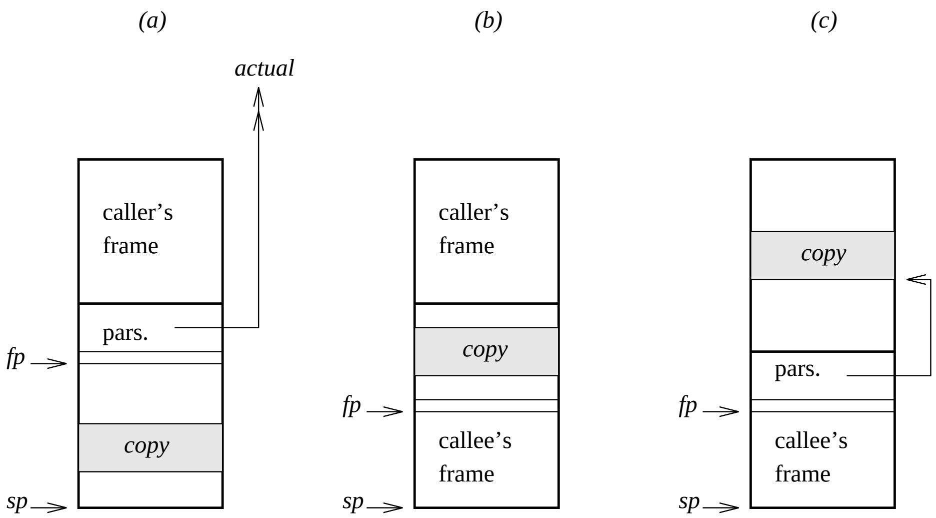


Figure 3.11: Copying structured value parameters
(a) Passed by reference, copied by called procedure
(b) Copied by caller, passed by value
(c) Copied by caller, passed by reference

The relationship between these three conventions is shown in figure 3.11, for a conventional stack organisation with a stack and frame pointer. The shaded area in each diagram is the copied value.

In part *a* the datum in the parameter area of the callee points to the actual parameter somewhere in memory. The callee has made a copy into its own stack frame, in the local variable space.

In part *b* the datum has been copied by the caller, with the value being placed in the parameter space of the callee's stack frame.

In part *c* the datum has been copied by the caller into its own workspace. A reference to this copy is placed in the callee's parameter area, so that the callee can access the value indirectly.

Constructed values

The discussion of passing structured values to value mode formal parameters has implicitly assumed that such values arise from reference to variables. However, structured values can arise as computed results in at least three instances in *Modula-2*: functions returning structured values, set operations on large (multi-word) sets, and non-constant value constructors.

```
PROCEDURE Foo(par : BigSet); ··· END Foo;

Foo(Set());          (* pass function return value as parameter *)

Foo(a + b);          (* pass expression result value as parameter *)

Foo(BigSet{i .. j}); (* pass constructed value as parameter *)
```

In each case, the value must be materialized in some memory location. In the case of parameter passing conventions which rely on the callee making the copy, it is not possible for the value to be constructed in place. For both of the other methods of parameter passing, it is possible to construct or return the value directly into the copy position, thus avoiding an additional block copy operation.

Open arrays

Pascal, *Modula-2* and *Ada* all allow for parameters to be arrays, the size of which is unknown to the called procedure. In *Pascal* two additional, hidden integer parameters are passed to the called procedure, specifying the upper and lower bounds of the *conformant array*, as it is called. In *Modula-2* formal open arrays are always indexed starting from zero, so a single, hidden cardinal *HIGH* value is passed for each dimension of the array.

Only two of the three parameter passing conventions of figure 3.11 may be used to pass open arrays. It is not convenient to pass the array by value, since the actual array value size would modify the runtime offsets of other parameters.

The array may be passed by reference, and copied by the caller. This entails a two-step procedure entry prolog. First the stack pointer must be adjusted to provide space for all the data the size of which is known at compile time. The stack must then be adjusted a second time, by a variable amount which depends on the actual *HIGH* value. The prolog must then execute a block copy operation to fetch the value. Finally, the reference to the actual parameter must be overwritten to point to the copy. The compiler cannot generate code to access the copy directly, since the offset of the copy is not known until runtime. Thus an indirect access must be used, see figure 3.12.

If this method of passing open arrays is chosen, it follows that the stack frame no longer has a size which is known at compile time. Hence either space for the copy must be allocated from outside the stack, or the possibility of saving a register and using a virtual frame pointer must be foregone, at least for procedures with value open array parameters.

As an alternative, the array may be copied by the caller, and then passed by reference. In this case, the stack frame of the caller must provide space for the copy. This is convenient, since in most cases the caller knows the size of the array at compile time, and hence can generate better quality copying code. The exceptional case arises when a procedure takes an open array formal and passes it on to another procedure. In just this case, the size of the copy is not known

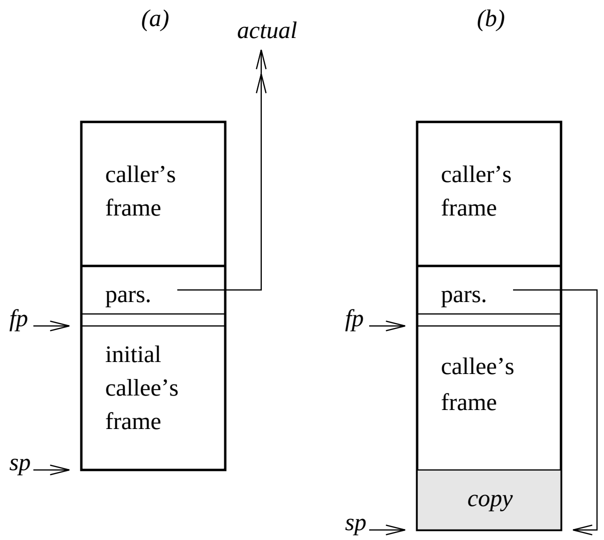


Figure 3.12: Copying an open array, before and after

by either the caller or the callee at compile time. For this case a copy location must be provided off-stack, or a variable size stack frame created.

A curious constraint appears in draft standard *Modula-2*. In this language, it is possible that the size of array elements may be different in the called and calling procedures. This means that the elements of open arrays of subrange types may need to be either widened or narrowed and range checked during the parameter copying process. This seems to constrain compilers to use the "copy by the caller, and pass by reference" convention of figure 3.11*c*, since only the caller knows the storage size of the elements of the actual array.

Pushing parameters on the stack

The traditional method of passing parameters on a runtime stack, is to push each of the parameters, in order, as they are computed. The parameters thus appear in the callee's parameter area with the first parameter to be pushed at the greatest offset from the stack mark. The last parameter to be pushed would be nearest to the stack mark.

Language *C* allows variable numbers of parameters, with optional parameters following mandatory parameters in the formal parameter list. Since the mandatory parameters must be at known offsets, it follows that optional parameters must be pushed onto the stack first, followed by the mandatory parameters at smaller, known offsets from the stack mark. It follows that when *C* implementations pass parameters by pushing them on the stack, they are pushed in right-to-left order. It also follows that the caller knows the size of the parameter area, but the callee does not. Thus the caller must be responsible for *cutting back the stack*, that is, for removing the parameters from the stack after the called procedure returns. These conventions: pushing parameters right-to-left, and cutting the stack in the caller, have come to be known as "*C calling conventions*".

Many *Pascal* compilers adopt different conventions, in which the parameters are evaluated and pushed left-to-right, and in which the callee is responsible for removing the parameters from the stack prior to the return instruction. These so-called *Pascal calling conventions* are awkward to implement on machines which do not provide hardware support in the form of a "return and remove N parameters" instruction. With our standard stack frame the parameters are below the return address in the frame, see figure 3.4. So if the machine does not have a parameterized return instruction the return address must be saved while the stack pointer is adjusted to remove the parameters.

Nested procedure calls do not cause any problems for those calling conventions which push parameters on the runtime stack. Consider the following call, which involves a nested function application —

```
WriteCard(ValueOf(str), width);
```

Suppose we wish to evaluate the above expression, using *C* calling conventions, where *ValueOf* takes an open array parameter.

We need to push the parameters to the call of "`WriteCard`" in right-to-left order. Therefore, the generated object code must first fetch and push the variable *width*, then we must evaluate and push the left hand parameter of the call. Recursively, in order to evaluate the nested function call we must push the parameters of the function, call the function, and push the returned value. The steps are as follows —

push the value of `width`	; *2nd param of WriteCard*;
push the HIGH value of `str`	; *2nd param of ValueOf*;
push the address of of `str`	; *1st param of ValueOf*;
call function `ValueOf`	
remove 2 params from stack	; *remove ValueOf params*
push result of `ValueOf`	; *1st param of WriteCard*
call procedure `WriteCard`	
remove 2 params from stack	; *remove WriteCard params*

We have assumed that the calling conventions expect *ValueOf* to copy the open array parameter, as shown in figure 3.12.

Note that in this example the evaluation of parameters is overlapped, but this causes no difficulty. When *ValueOf* is invoked it expects, and finds, two parameters. It neither knows nor cares that one of the parameters of the enclosing procedure call has been evaluated and is on the stack already.

In the case of structured parameters which are passed by value, as in figure 3.11*b*, a typical sequence of instructions would be to move the stack pointer to make room for the parameter value, then invoke a block-copy operation with the actual parameter value as source operand, and the top-of-stack as destination.

Passing parameters in registers

It has become common in modern machine conventions for the first few parameters of procedures to be passed in registers. This convention partly reflects the greater number of registers which are available in such machines. However, there are compelling performance advantages from such a choice, particularly for *RISC* architectures. In these architectures values are computed into registers, and must be loaded into registers before being used as operands in instructions. If parameters are passed in registers, then there are a statistically significant proportion of cases where it is possible to evaluate the argument into the parameter register, call the procedure, and have the callee use the value out of the register without ever moving the value into memory. Typical *RISC* machines pass the first four to six parameters in registers, placing any further parameters into a reserved *parameter assembly area* in the stack frame.

The price which is paid for the efficiency advantages of passing parameters in registers is some added complexity at compile time, as extra static analysis is necessary.

With fixed parameter assembly, the evaluation and placement of parameters for nested procedure calls cannot be overlapped. Let us suppose that the first two parameters are placed in r_4 and r_5 respectively, as is the case for *MIPS* architecure machines. For our previous example —

```
WriteCard(ValueOf(str), width);
```

this convention determines that *width* will be loaded into r_5, the *ValueOf* result into r_4, *HIGH*(*str*) into r_5, and the reference to *str* into r_4. Clearly the evaluation of these parameters, and their placement into the parameter registers cannot be done in an arbitrary order, since the same resource is used repeatedly.

In general, a safe strategy is to serialize all procedure calls. This may be done either by the frontend or the backend. In effect, the calls are unnested by performing the function evaluations first, and moving the function results to the parameter registers afterward. In later chapters we shall see that using *VAL*'s virtual registers to store function results while parameters are being evaluated solves the problem elegantly.

Notice that the problem with overlapped parameter evaluations does not inherently have to do with the use of registers for the parameters. The root cause is that the parameters are being passed in a shared resource.

In order to deal with an arbitrary number of parameters, we shall allocate a *parameter assembly area* at the top of the stack frame of the current procedure. This area must be large enough to contain the parameters of whichever called procedure requires the largest parameter space. This assembly size attribute must be evaluated by static analysis of the procedure body.

As far as front-ends are concerned, parameters are moved to known offsets in the parameter assembly area. Back-ends may very well decide that the first N bytes of this assembly area will be placed in machine registers, but the front-end may usually abstract this detail away.

Using fixed assembly on non *RISC* machines

There may be some advantage in adopting fixed parameter assembly conventions, even in the case of machines which traditionally push their parameters onto the stack. The *Intel-386* architecture is a typical example. In this machine, it is permissible to use the stack pointer as an index register, so parameters may be placed in a parameter assembly area relative to the "`esp`" register. Each such access is slightly faster than a "`pushl`", on average. Using fixed assembly also ensures that it is never necessary to waste cycles cutting back the stack.

However, the main advantage of adopting such conventions is to create fixed size stack frames, so that no register needs to be wasted on a frame pointer. In the *Intel* case this means the difference between having seven registers to allocate, rather than the usual six.

The only disadvantage of such a convention is the slightly greater analysis which is required at compile time, and the fact that debugger programs are able to supply far less information when programs use these conventions. For example, debuggers are unable to deduce the number of parameters by examining the instruction at the return address. Perhaps more serious is the fact that debuggers are unable to trace the dynamic call chain without special help, if there are no conventional stack frames. Possibly the best compromise is to used fixed assembly, but create conventional stack frames with a frame pointer register in the presence of the debugging –**g** command line option.

Passing parameters in register windows

Some *RISC* machines have registers which are arranged in overlapping windows. In the *SPARC* architecture there are 32 general purpose registers of which eight are global, and do not form part of the window. The other 24 registers include eight registers which overlap between the caller and callee procedures, and 16 fresh registers. The register windows form a circular buffer of perhaps four or eight register sets. These sets are used as a finite depth stack. If the stack overflows, the bottom set of 16 registers are saved to a reserved location in the corresponding stack frame. If the stack becomes empty during procedure return, then one or more saved windows are restored from their stack frames. Figure 3.13 shows the relationship between the register windows of a caller and callee procedures.

The region of overlap between a caller and callee register window is used to pass the first six parameters of the call. The output registers $o0$ to $o5$ hold the first six outgoing parameters, or more generally the first 48 bytes of the parameter assembly area. These six registers appear as input registers $i0$ to $i5$ in the register window of the called procedure.

For machines with register windows the passing of parameters involves the same considerations as for other machines with fixed parameter assembly. The major difference is that no special steps need to be taken to save and restore callee-saves registers. In effect, the outgoing local and input registers are callee-saves, since they are preserved across procedure calls. However, nothing need be done to save them, since they will be automatically saved if the circular buffer fills up, and in many cases will never be saved.

Of course, the stack frame must reserve space for the register window to be saved, just in case that should that prove to be necessary inside some dynamically nested procedure call. It is normal to allocate this space as part of the stack mark, so that there is a comparatively large gap between the negative offset of

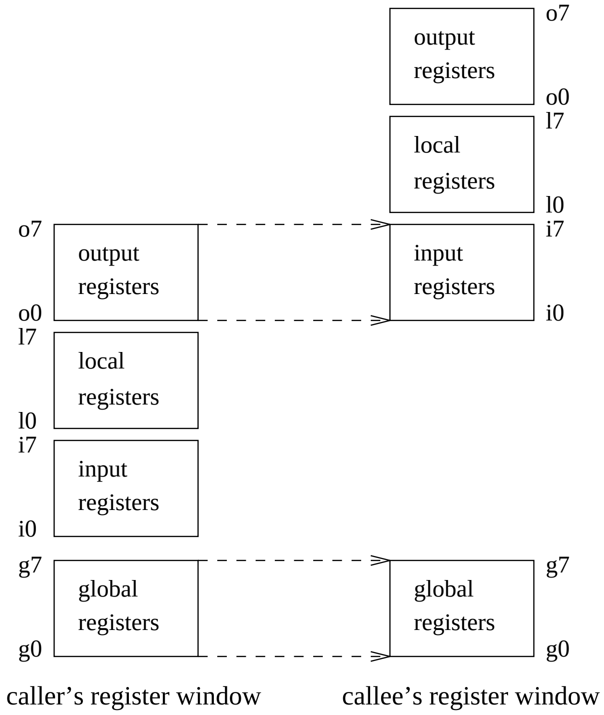


Figure 3.13: Overlapping register windows of caller and callee in *SPARC*

the first local variable, and the positive offset of the home location of the first parameter.

There are some special possibilities which arise in the case of leaf procedures. For leaf procedures that can compute their results without using other than the global and input registers it is not necessary to even save the previous register window. For small leaf procedures this may give a worthwhile performance gain.

Returning function results

It is almost universal that functions return scalar results in a machine register. On *Intel-386* it is the *eax* register, on *MIPS* it is r_2, while on *SPARC* it is $i0$ from the returning functions point of view, and $o0$ from the point of view of the caller.

The situation with floating point results is more diverse, although hardly more complex. Some machines have a specified floating point register which is used for function results, while in the *Intel* architecture it is usual to return the result in the top of the floating point coprocessor stack.

Rather more interesting is the task of returning results, such as structures, or even arrays, which are too large to fit in a machine register. At a certain level of abstraction, the solution is simple. The called function must be passed an additional argument which points to the location into which the result must be placed. There are really only two decisions to be made: where is the result location to be allocated, and how is the extra reference parameter to be passed?

Consider a statement which assigns a structured function result to some variable —

```
vector := VecFunc(···);
```

Conceptually, the returning of a function result is an indivisible operation in most languages. Yet, the function must obviously construct the result component by component in some fashion. The danger is that, for some pathalogical choice of input arguments, the partially constructed result might overwrite its own input data. This problem may be avoided, by sufficiently strict language semantics. For example, a language which permitted functions to take only value parameters, and forbade access to non-local data within function bodies, would prevent this problem. However, in conventional languages there are no such restrictions. There seems to be only two conservatively safe choices. One solution is to insist that functions which return structures must construct their results in a temporary location in their own stack frame, then return the value to the caller with a block copy. A second solution would be to allow such functions to construct their results in the indicated destination location, and have the caller determine when this destination must be a temporary location which is afterward copied to the real destination. The point of this second strategy is that when the caller is compiled it may be possible to determine that the destination is logically inaccessible to the function, and hence an additional block copy can be avoided. In the example given above, where the destination is a named location, a temporary location probably must be allocated by the caller. The caller then performs a block copy to the destination after the function returns.[6]

Some architectural conventions pass the result pointer as a hidden, additional first parameter. This fits well with stack parameter conventions, since after all of the explicit parameters have been pushed, the destination pointer is pushed. Within the called function, all parameters have their offsets increased by one pointer size. It is possible to adopt the same strategy even when parameters are passed in registers, and this is the standard convention on the *MIPS* architecture.

Other conventions are possible. In the case of *SPARC* a reserved location within the stack mark space is used for the destination pointer, while in other conventions an otherwise unused register is used to pass the pointer.

3.2.5 Addressing data from non-local scopes

The languages *Pascal*, *Modula-2* and *Ada* procedures may be declared nested inside other procedures. Language *C* does not permit such nesting.

It is semantically possible for nested procedures to refer to data which has been declared in enclosing scopes, and which exists in separate activation records. We refer to any such access as *uplevel addressing*. In this section, we shall explore two different mechanisms which provide such uplevel addressing. These two mechanisms are *the display vector*, and *the static chain*.

[6] The **gardens point** compilers adopt this second convention. Functions always construct results in the destination location, passed to them by their caller. The compiler of the caller must determine if the destination must be a temporary, or may be the ultimate destination.

```
program P0;
   var P0 globals;
   procedure P1(...);
      var P1 local variables;
      procedure P11(...);
         var P11 local variables;
      begin ··· end; (* P11 *)
      procedure P12(...);
         var P12 local variables;
      begin ··· end; (* P12 *)
   begin ··· end; (* P1 *)
   procedure P2(...);
      var P2 local variables;
      procedure P21(...);
         var P21 local variables;
      begin ··· end; (* P21 *)
   begin ··· end; (* P2 *)
begin ··· end. (* P0 *)
```

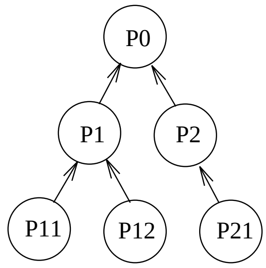


Figure 3.14: Program skeleton and corresponding scope tree

Nested scope rules

Since procedures must always be properly nested, each nested procedure has an unique enclosing procedure. We may therefore represent the nesting of procedures by means of a *scope tree.* The skeleton of a program, and the corresponding scope tree are shown in figure 3.14. This particular program is used as an example throughout this section.

The rule for identifier visibility in languages with nested scopes is that an applied occurrence of a particular identifier will be bound to the most local declaration of that identifier. In operational terms, the local scope name-table is searched first, and in the event that the name is not found, the ancestors in the scope tree are searched successively. If the root node is reached without a declaration being discovered, then the identifier is undeclared, and the occurrence is erroneous.

Note a curiosity with the scope tree in figure 3.14. Procedure names occur as labels of the scopes, and these names themselves denote declared entities. However, the declaration of a procedure occurs in the scope which is the *parent* of the node in the tree which is labelled with the procedure name. Thus it is possible, for example, to call procedure *P2* from procedure *P12* although it does not look from the figure as though *P2* is on the path from *P12* to the root of the scope tree. In fact, *P2* is declared inside the scope of *P0*, which *is* on the

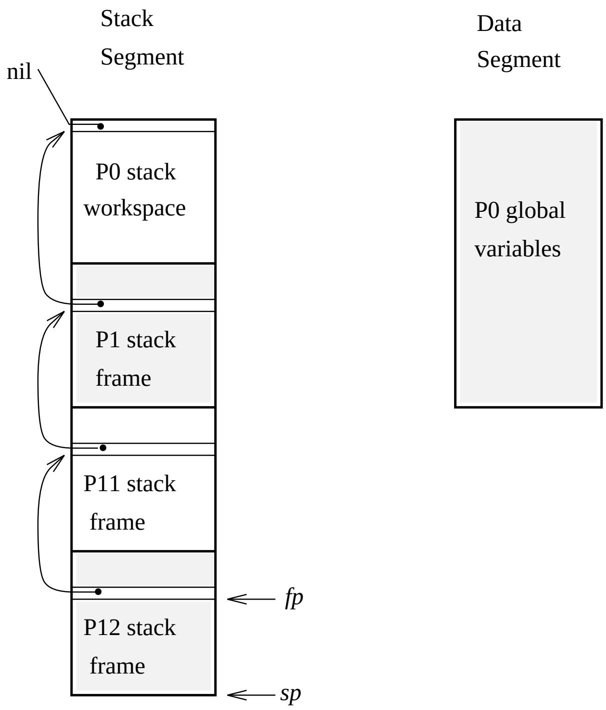


Figure 3.15: Stack frame showing dynamic chain. Accessible data areas are shaded

path.

The dynamic chain

Let us suppose, for the program shown in figure 3.14, that the following sequence of procedure calls takes place —

> program *P0* calls procedure *P1*
> procedure *P1* calls procedure *P11*
> procedure *P11* calls procedure *P12*

After these calls, the stack is as shown in figure 3.15, which also shows the dynamic chain linking the stack frames. In this figure, the accessible data areas are shown shaded, and the dynamic chain is the linked list shown down the left hand side of the stack.

The variables which are declared in the body of the program are placed in the data segment, as they have static extent. However, the program may still have a stack frame, if it needs some workspace. In this example, since the currently active procedure is nested, the code has access to the variables of procedure *P1*, as well as to its own locals and the program globals.

The display vector

One method of accessing data in the stack frames of enclosing procedures is to maintain a pointer stack, called a *display vector*, which points to each of the logically accessible stack frames. In the case of diagram 3.15, the vector would logically have three pointers, pointing to the local frame (*P12*), the enclosing frame (*P1*), and to the global data area. In practice only the intermediate levels of the vector need to be implemented, since the local pointer simply duplicates the frame pointer, and the global pointer points to the static data area. The intermediate levels of the display vector are usually kept in a fixed size array in the runtime support system. The fixed size of the array places a limit on the allowable depth of static nesting of procedure declarations, with limits of 10 to 16 being common. In the case of programming languages which support coroutines, the display vector forms part of the per-coroutine state.

Local data is accessed relative to the frame pointer, as always, while global data is also accessed in the normal way. Intermediate level data at level N, with stack frame offset k is accessed by the address expression —

$$\texttt{display}[N] + k$$

Maintainence of the display vector data is one of the tasks which must be performed by the procedure prolog and epilog. However, not every procedure needs to manipulate the display. A procedure has an attribute which indicates whether or not it has data which is uplevel addressed, and another which specifies the level of static nesting, “`lexLevel`”. The emission of display code in the prolog is controlled by the following logic —

```
IF procedure has uplevel addressed objects THEN
   emit code to save display[lexLevel] in workspace;
   emit code to copy fp to display[lexLevel]
END;
```

In the epilog, the corresponding logic is —

```
IF procedure has uplevel addressed objects THEN
   emit code to restore display[lexLevel] from workspace;
END;
```

It is seen that at most a single display location needs to be saved and restored on procedure entry and exit. That such saving and restoring is necessary at all may be understood by considering the situation where a very deeply nested procedure performs an “uplevel call” to a much less nested procedure. Consider figure 3.16 in which a procedure at level 4 calls one of its own predecessors at level 1. Before the call, the display vector has five active elements, and these elements must be unchanged when control returns to *P4* after *P1* returns. After *P1* has been called, the display has only two active elements.

It appears as though the whole display must be saved and restored when an uplevel call, such as that in figure 3.16 occurs. However, this is not so, since

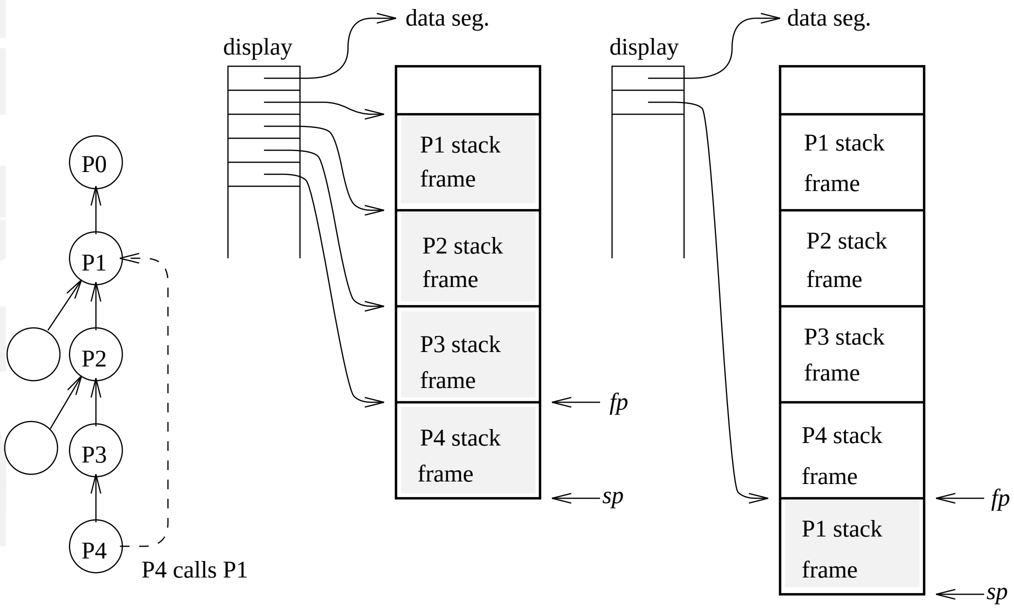


Figure 3.16: Scope tree, with before and after stack frames when *P4* calls *P1*

we may simply leave all of the display elements unchanged in the vector, and only save and restore the ones which are actually overwritten by the call. In the example given, only the element at index 1 is overwritten, so it is sufficient to save and restore that one element.

The static chain

The alternative to the display vector is the use of a second linked list of stack frames which we call the *static chain*, since it reflects the static structure of the program. With this organisation, each stack mark contains an additional datum, the *static link*, which points to the stack frame of the lexically enclosing scope.

Figure 3.17 is a modified version of figure 3.15, which shows the static chain (dashed lines) as well as the dynamic chain. Notice that all accessible data areas may be accessed by following the static links starting from the currently active procedure. The static links of other procedures, such as *P11* in figure 3.17, point to the stack frames of their statically enclosing procedure, but are of course inaccessible until control returns to the procedure with the chain-end.

In order to access uplevel data using static links, it is necessary to follow as many links in the chain as the lexical level difference between the currently active procedure and the procedure which declared the data. This is in contrast to the situation with a display vector, where access to any intermediate-level datum takes constant time, irrespective of the difference in lexical level. However, this distinction should not be weighted too heavily, since studies have shown that typical programs access data over only a very shallow range of nesting levels.

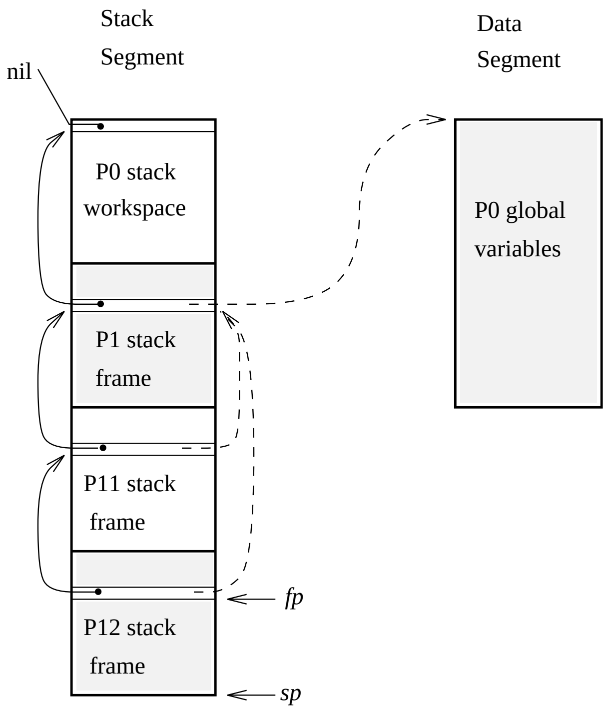


Figure 3.17: Stack frame showing static and dynamic chains. Static chain is dashed lines

In order to maintain the static chain, it is necessary for procedures to store the extra datum in the stack mark, as part of the entry prolog. This requires cooperation between the caller and callee. The caller must pass the extra datum, and the prolog must store it. It is common for the static link to be passed to the called procedure as a value in a machine register, but other possibilities exist.

Of course, not all procedures need to set up a static link. However, the conditions under which it is necessary to do so are different to the condition under which procedures need to manipulate the display vector. A procedure needs a static link to be passed to it when it is called, if either

- the procedure accesses uplevel data declared in a statically enclosing procedure, or

- the procedure encloses a procedure which accesses data declared in a statically enclosing procedure

The first rule corresponds to the rule for the display vector case. The second rule reflects the fact that the chain must pass from the frame of the procedure containing the data to the frame of the procedure which accesses the data. Along the way, the chain may pass through other frames which play no other part in the process.

The value of the static link which is passed to a called procedure varies according to the lexical relationship between the called and calling procedures. The link which is passed to the called procedure should point to the stack frame of the lowest common ancestor of the called and calling procedures.

Supposing that the link is to be passed in a register r_{link}, and "**caller**" and "**callee**" denote references to the respective procedure descriptors in the AST. We shall also assume that the frame pointer register *fp* points to the static link field in the stack mark. The relevant part of the code which emits the call would then be —

```
load up parameters to the call, if any;
IF callee needs a static link THEN
   emit "move fp,r_link"; (* initialize linkage register *)
   FOR ix := 0 TO caller^.lexLevel - callee^.lexLevel DO
      emit "lw (r_link),r_link"; (* dereference linkage register *)
   END;
END;
emit the call instruction;
```

In the case of the call of a child procedure, the static link is just the *fp* register of the caller, while for a call of a procedure at the same lexical level, the static link is a copy of the static link of the caller. In the general case, the static chain must be followed a number of steps equal to the difference in lexical levels between the caller and callee. The static link of that frame will point to the frame of the common ancestor of the called and calling procedure.

Figure 3.18 shows a variation on the uplevel call example in figure 3.16, but this time with static chains. It will be seen that the difference in lexical level between the caller (*P4*) and callee (*P2*) is two. The static link of *P4* points to the frame of *P3*, and the link of *P3* points to the frame of *P2*. The link of *P2* is thus passed as the static link to the call. As stated, the link points to the frame of *P1*, which is the lowest common ancestor of *P4* and *P2*.

3.2.6 Architectures with global data pointers

So far we have assumed that within any procedure, access to statically allocated data is uniform. That is to say, it has been assumed that access to data declared in the same compilation unit as the procedure (*local static data*) is performed in the same way as access to data declared in other compilation units (*external static data*). There are circumstances under which this assumption might not hold.

The final address of static data is unknown at compile time and becomes known, at the earliest, at link edit time. Indeed, in the case of dynamic linking, treated in section 3.2.8, the address of the data may be unknown until the module is loaded at runtime. Thus the output of a code generator can only refer to such data symbolically. In either case at runtime the address is known, typically as a 32-bit address constant. The instructions may refer to the data either using an address constant, or as an offset from a *global data pointer*.

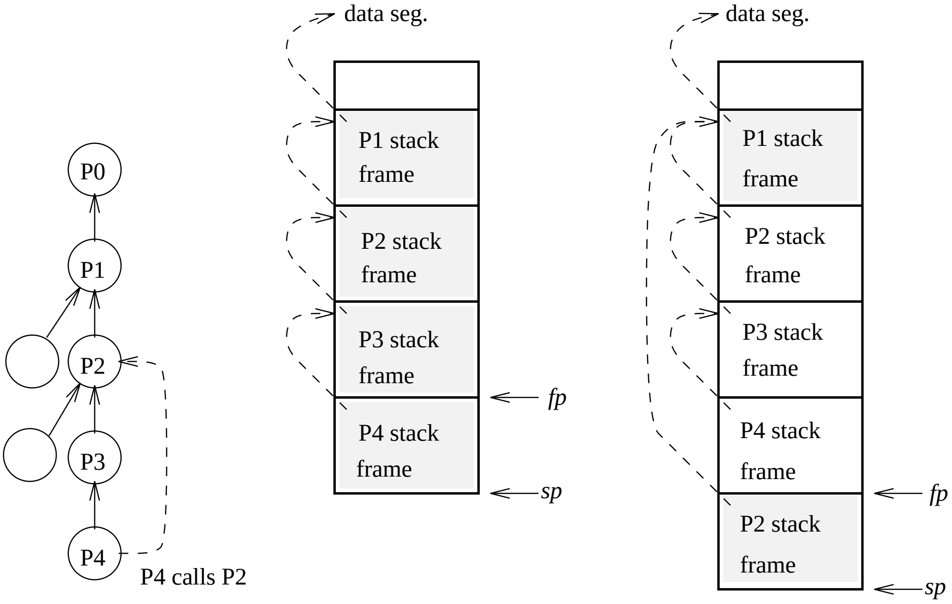


Figure 3.18: Scope tree and static chain, with stack frames when *P4* calls *P2*

For machines with variable length instructions, such as most *CISC* machines, there is no problem in using long immediate address constants as part of an address expression. For example, in the *i386* architecture address literals may be 32-bit immediate constants, and hence may point anywhere in the address space. In *RISC* machines this is impossible, since the 32-bit fixed-length instructions have only space for a short offset field. Typically literal addresses may be 16-bit offsets from some address register, or must be loaded into a register using two or more instructions.[7]

Many programs will require static data the total size of which easily exceeds the range of the allowed literal offset, posing a difficult design issue for the runtime conventions. The simplest, but least satisfactory solution, would be to accept that accessing statically allocated data will always take more that one instruction. Consider the example of loading the word at address *gName* into register `r7` say.

Logical asm code	Literal asm code	Code after linking
`lw` *gName*`,r7`	`lui hi(`*gName*`),$at`	`lui` *HHHH*`,$at`
	`lw lo(`*gName*`)($at),r7`	`lw` *LLLL*`($at),r7`

The logical assembler instruction just asks for the word at the named location to be loaded. The assembler must expand this out to two instructions by first

[7]Many *RISC* assemblers have a single opcode which apparently loads a 32-bit literal into a register. However, these instructions are invariably macros which expand out to two or three real instructions.

loading the high significance part of the address "`hi(`*gName*`)`" into a temporary register, using a *load upper immediate* "`lui`" instruction. The *assembler temporary* register, `$at`, is reserved for this role. A second instruction will take the low significance part of the address "`lo(`*gName*`)`" and use this as an offset from the assembler temporary register, finally loading the required word. After the linker has resolved the location of *gName*, the "`hi()`" and "`lo()`" expressions become short immediate operands which we have denoted *HHHH* and *LLLL*.

The only alternative to this laborious mechanism is to use one of the registers as a pointer into the static data area, and either address only some of the data this way, or change the register as different regions of the data are required.

Shared global pointers

The compromise which is conventionally used in systems based on the *MIPS* architecture is to reserve a single register for use as a global pointer throughout the whole of the program execution. This register, called `$gp` in the assembler format, is set up on program initialization, and is never changed.

In this convention, statically declared data is segregated, with small objects placed close together so that all may be addressed as a short offset from `$gp`. Larger objects are placed elsewhere in the address space, and must be addressed using the two instruction idiom.

If an object is within offset range of the global pointer, the code example above may be implemented by the much more efficient —

Logical asm code	Literal asm code	Code after linking
`lw` *gName*`,r7`	`lw ofst(`*gName*`)($gp),r7`	`lw` *SSSS*`($at),r7`

In this case, the linker resolves the offset of *gName* from the global pointer as a short immediate operand which we have denoted *SSSS*.

Of course, all of this can only work if the assembler knows those static names which are within offset range of the global pointer, and those which must be addressed using the two instruction form. In the *MIPS* case, the assembler must know the storage size of each object. If the object is not larger than the magic "gnum" then it is addressed relative to the global pointer with a short offset. If the object is larger than "gnum", or is of unknown size, then it must be addressed using the long form.

Since decisions based on the "gnum" are applied during the compilation of individual modules, it is possible that during linking the short data area will become too large, and overflow. If this happens, there is no alternative but to go back and recompile everything with a smaller "gnum". It is usually thought to not be practical to rewrite object code once it has been produced, since program relative jumps in the code might be invalidated by varying the length of instruction sequences.

Varying global pointers

An alternative scheme for addressing static data is based on the assumption that the data for each compilation unit will fall within the addressing range of the global pointer *for that compilation unit*. Thus each compilation unit will use a different content for the global pointer, making it point to its own area. There are two benefits to this scheme. Firstly, the chances of overflowing the area are sharply reduced, and secondly, even if the area does overflow the assembler will know this at compile time. However, the scheme has a disadvantage in that procedure call and return are complicated by the necessity of modifying and restoring the global pointer for every out-of-module call, using a *load address*, "`la`" instruction. Here are examples of in-module and out-of-module calls, for call conventions in which all the work is done on the caller side —

In-module call	**Out-of-module call**
load up parameters	*load up parameters*
`call` *ownProc*	`la` *farGP*`,$gp`
...	`call` *farProc*
	`la` *ownGP*`,$gp`
	...

Note that both of the loads of the global pointer in this example will need two instructions. Both *farGP* and *ownGP* are long symbolic literals. A module's own global pointer value could be stored in its own data area, but this would not help since after the call returns the module's own data is inaccessible since the global pointer is invalid. Some of the alternative methods to save an instruction or two are explored in the exercises.

A completely different approach is used in machines which try to share the global pointer between as many modules as possible. In such cases the linker must pack as much of the global data into one area as possible, and rewrite the object code to take account of the sharing of the pointer between modules.[8] In each case we shall assume that a dedicated location in each non-leaf stack frame is reserved for saving the global pointer. Every procedure call is now assembled optimistically, assuming that no modification of the global pointer will be necessary. However, the next instruction following the call is a "no-op". This instruction is the first instruction to be executed following the return of the called procedure, and is padding which reserves a location so that the linker has space to place the instruction which restores the global pointer if that should prove necessary. The sequence of events in the "lucky" case, where the global pointer is not modified, is shown in figure 3.19.

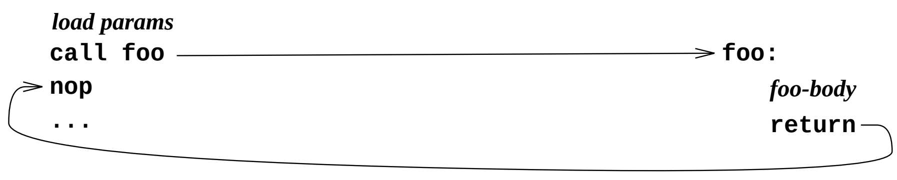


Figure 3.19: Procedure call not needing an adjustment to `$gp`

In the "unlucky" case, where an adjustment to the global pointer is required, the linker will create a *indirect calling stub*, and overwrite the call instruction

[8]The details of how the *IBM Power* architecture uses this idea are given in a later section.

in the object code so that the call goes to the stub. The "*no-op*" following the call instruction will be also overwritten, in this case with the instruction which restores the global pointer of the caller.

The code of the stub will save the current global pointer in the reserved stack frame location, load up the new global pointer, and *jump* to the real procedure entry point. Figure 3.20 shows the sequence of events in this case, where it is assumed that the reserved location in the stack frame is `($fp-8)`. Note carefully,

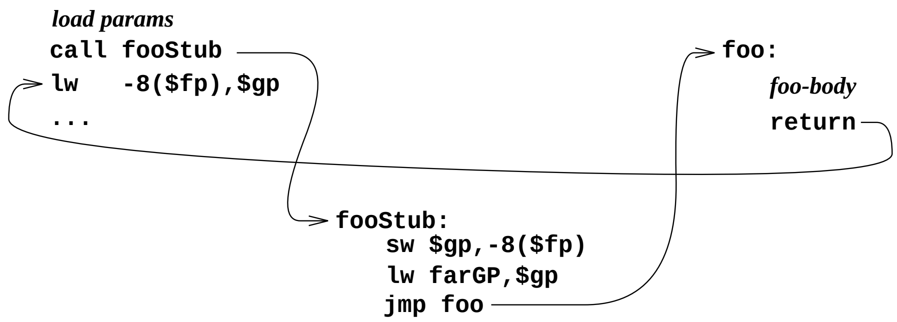


Figure 3.20: Procedure call using a stub to adjust `$gp`

that the stub does not *call* the real procedure, but jumps to it. This means that the return address from the first call is unmodified, and the real target procedure returns to its caller without passing through the stub again. Such stubs which are called as procedures, but *jump* rather than *return* are called *trampolines* since they "bounce" the flow of control on to another destination.

3.2.7 Procedure parameters and variables

Representation of procedure values

Many languages provide for procedures to be treated as values. In *Pascal* procedures may be passed as parameters to other procedures. *Modula-2* allows procedure types to be declared, and variables, parameters and function return values to be declared to be of such types. *C* allows types to be defined which are pointers to functions, and allows variables, parameters and function return values to be of such types.

The use of procedure values is deeply embedded in the style of programming which uses objects, and graphical user interfaces. In such environments procedure values are passed to procedures, and are entered in structures and tables so as to customize behaviour at runtime.

If we consider the case of simple procedures which are not nested within any other procedure, then there are essentially two ways of representing a procedure value at runtime. Most machine architectures provide a subroutine call instruction which can take either a literal constant label as a target operand, or can jump to an address held in a machine register. In the case of the *MIPS* architecture there are two instructions —

```
jal     proc-label        ; jump to label and link
jalr    r_dst             ; jump to reg target and link
```

This suggests that the address of the entry point of the procedure might be used as the value. Indeed this is the most common implementation of procedure values. When this is the case, creating a procedure value from a declared procedure consists of simply taking its entry point address. If some variable *foo* contains the entry point address, say of *InOut.WriteLn*, then the call `foo()` will generate the following code —

```
lw      foo,r11           ; any free reg will do
jalr    r11               ; jump to entry point and link
```

The alternative representation for procedure values uses a descriptor, sometimes in the data segment. In this case, the descriptor can contain an arbitrary amount of data, which must include the procedure entry address.

With such representations, the procedure value is the address of the procedure descriptor. When a procedure is called, the parameters of the indirectly called procedure are evaluated normally, but the descriptor address must be passed as an additional parameter to a dispatch procedure. In the case of *RISC* machines the additional parameter would normally be placed in a caller-saves register other than the standard parameter registers. Repeating the above example, in this case *foo* would have the descriptor address in it, and the call `foo()` would produce the following code —

```
lw     foo,r3             ; r3 is extra dispatch arg
jal    $$dispatch         ; call the dispatch procedure
```

The dispatch procedure, which is often a sytem routine, must perform whatever manipulations are required before jumping to the procedure entry point. Here is an outline of some typical code —

```
$$dispatch:                          ; dispatch procedure entry point
        initialize, using r3 to access descriptor
        move entry address from descriptor to r1
        jalr   r1                    ; call to real entry point
        restore original procedure state
        ret                          ; return to original caller
```

A difficult issue for the design of dispatch stubs of this kind is to decide where to save any state of the original caller, in cases where state must be saved. Since the final destination may be unaware of whether it is called directly or indirectly, the stub must rely on space surrendered by the caller to save any data. If the state is stored in the stack frame of the caller, and the state is restored by code in the caller, then the stub may be configured as a trampoline in the same way as outlined in figure3.20.

Procedure values for nested procedures are necessarily more complex, since they must not only have entry point information, but must have information regarding their environment as well. Such a combination of a procedure plus an execution environment is called a *closure*.

Nested procedures must have access to all the variables of all relatively global scopes in which they are declared. In the case of runtime organizations which use a display vector, a number of display vector elements equal to the static nesting depth of the procedure must be saved as part of the closure. In the case of runtime organizations which use a static chain, a single pointer to the head of the chain suffices.

Thus a closure consists of a minimum of two addresses, the entry point address, and a static link pointer. Such a procedure value may either be represented as an address pair, or may be a single pointer to a descriptor which contains the environment information.

The saving of procedure closures as variables is subject to some constraints. The environment part of the closure consists of references to activation records of enclosing procedures. In languages which allocate activation records on a runtime stack, the environment of a closure becomes invalid as soon as any of the stack frames of the environment are deallocated. Thus a closure can only be guaranteed to be valid if the extent of the environment is greater than the extent of the variable in which it is stored. This condition can be checked statically for assignment of literal procedures to procedure variables. However, it requires an explicit representation of the static depth, and a runtime check, in the case of assignment of one procedure variable to another.

Pascal procedure parameters

Pascal has only procedure parameters, and does not allow closures to be stored in variables. This choice ensures that procedure values are always valid.

The actual parameter passed to a formal parameter of procedure mode must necessarily be a procedure which is visible to the caller. The only nested procedures visible to a procedure are procedures which are declared in the caller or its ancestors. Hence the uplevel environment of the parameter must persist at least as long as the caller. Since the procedure value only persists as long as the callee, the closure must always be valid.

Consider the scope tree in figure 3.14 on page 63. Procedure *P12* may pass any one of *P1, P2, P11, P12* as actual parameters to a formal procedure. The first two have an environment which is static, and the latter two have an environment which is valid so long as *P1* has a valid stack frame. Since *P12* cannot be called unless *P1* is valid, the closure is always safe.

Modula-2 procedure variables

Modula-2 allows procedure values to be assigned to variables, passed as parameters and returned as function results. However it has a rule which prevents the extent problems mentioned in the last section. Procedure values in *Modula-2* can only be procedures declared at the outer level, that is, procedures which are not nested within other procedures. This is sometimes called the level-0 rule.

This rule removes all possibility of error, as procedures declared at the outer level have no environment other than the static environment. Therefore no environment needs to be stored as part of the closure, and procedure values may be represented as a single address value.

Language C pointer to functions

In language *C* the situation is even simpler than *Modula-2*. Nested procedures do not exist in *C*. This means that every procedure necessarily obeys the level-0 rule, so that procedure values do not require any environment information.

3.2.8 Dynamic linking

Operating systems such as *UNIX* provide for the sharing of the code of programs, if multiple instances of the same program are running at the same time. In this case, each instance of the program has its own private data segment, but all instances share the text segment.

In modern software systems, this sharing of the code of entire programs may still be inefficient, as several *different* programs running simultaneously may have large bodies of identical library code statically bound to each program's executable image. Dynamic linking is the mechanism which allows dissimilar programs to share common code. It is so called, because the code of libraries is loaded dynamically, *on demand.* In this way, code which is never used is never loaded. Furthermore requests to load modules which turn out to be already loaded are honored by allowing shared access to the already loaded code. There are a number of different ways in which dynamic linking is implemented. Here we shall only deal with one possibility in detail.

Dynamic loading places additional demands on code generation. In particular, the location of the dynamically loaded library cannot be known until the library has been loaded at runtime. Thus the addresses of the procedures in the libraries have to remain unresolved until runtime.

It is normal to allow for dynamic loading by making the calls to dynamically loaded procedures indirect. One way of arranging this is to create a global table which holds the addresses of all of the dynamically loaded procedure entry points. The table might hold (*name, address*) pairs with the addresses initially pointing to a system trap handler. When the trap handler is activated, it must arrange the loading of the library (if it is not already loaded), and the updating of the address field in the pair.

An alternative arrangement uses indirect calling stubs so that the call to the library routine goes first to the stub. The stub must invoke a trap handler on the first call, but may be updated to call the loaded procedure directly on subsequent calls. Note that this arrangement requires the stubs to be in a region of memory which is both executable and writable.

As will be seen directly, it is not only the code which calls dynamically loaded libraries which is different. Code capable of being dynamically loaded must able to run with the same result, no matter where in the memory space it is loaded. Such code is said to be *position independent*.

Position independent code

Position independent code is code which runs in the same way no matter where it is loaded in the address space. In order to be position independent, the code must contain no addresses which are bound before loading.

There are several issues involved, but let us deal first with a non-issue. Almost all contemporary machine architectures provide for conditional and unconditional branch instructions to be *program counter relative*. For example, an assembler processing the instruction —

```
br label2          ; unconditional branch to label2
```

will compute the address offset between the location "*label2*" and the *address of the next instruction*, symbolically "*here*" and will emit binary code corresponding to the following —

<*br opcode*><*offset*>

At runtime, the effect of this instruction is simply to add the immediate offset to the program counter, so that the successor instruction of the branch is the instruction at *label2*.

If all of the flow of control branching within a single procedure uses these relative branches, then the procedure will behave in exactly the same way no matter where in the memory it is loaded.

However, access to statically allocated data, and calls to other procedures traditionally use immediate addresses embedded in the code, and hence cannot be position independent. In fact, if the called code or data belong to a module which is loaded independently, then even program counter relative addressing does not help. In such cases, either the unresolved immediate addresses must be adjusted as the program is loaded, or access must be indirect.

The *IBM Power* architecture uses conventions in which all code is position independent. Each module contributes part of a table, called the *global offset table* which at runtime holds the addresses of all unbound locations. All accesses to locations which are not statically bound must be made by fetching the fixed-up address from the global offset table. This mechanism is conceptually simple, but requires some care with the details of implementation.

Access to the global offset table is very frequent, so it is important to ensure that these accesses are efficient. This is ensured by reserving a register R_{got} which points to the applicable table. For each separately linked program fragment the linker will attempt to construct a single table from the separate tables arising from each compilation. Provided the size of the single table does not exceed the offset addressing range, R_{got} will not need to be changed as procedures in the bound part call each other. If the size of the shared table exceeds the offset address range, or for calls between separately linked components the linker will introduce trampoline stubs and rewrite the object code as illustrated in figure 3.20.

Notice the rather subtle hierarchy of binding times involved here. At compile time, the assembler will know the *relative index* of a particular datum within the per-compilation-unit *table of contents*. At link time, these tables of contents will be merged into one or more global offset tables. The linker will be able to compute the absolute index into the global table corresponding to each relative index in the constituent modules. The linker is thus able to fix up the indexed accesses by overwriting the object code. Finally, the *contents* of the global table are determined by the loader, when the various parts of the program are loaded.

3.3 Object oriented languages, and method dispatch

3.3.1 Runtime polymorphism

The essence of the object oriented programming paradigm, is the creation of data objects which encapsulate both state (data values) and behaviour (methods). The encapsulation of behaviour is invariably implemented by associating procedure values with objects.

Most object oriented languages allow a restricted form of *polymorphism*, that is *variability of form*. There are two aspects of this polymorphism. Firstly, there is some kind of polymorphic assignment, in which objects belonging to any subtype of a particular type may be assigned to variables of that type. Thus for statically typed languages, variables have a type at runtime which may be some subtype of the statically declared type of the object. Such polymorphic assignment provides an alternative to the use of union types for the construction of heterogeneous data structures.

The second aspect of polymorphism is the runtime dispatch of methods. If a named method is invoked on a particular object, then the actual code which is executed is the method of that name which is associated with that particular object. Thus different subtypes of a particular type may have different behaviours associated with the same method name.

The implementation of method dispatch relies on the existence of tables of procedure values, together with some kind of indexing. Since each type may have a large number of methods defined on it, and there may be a large number of instances of each particular type, some sharing of method tables is possible. Where such shared tables exist at runtime, we shall call the data structure which contains them the *runtime descriptor*. The differences between various languages, in terms of runtime structure, come down to different methods of indexing into these method tables.

Classes and types

The use of the terms *class* and *type*, and consequently subclass and subtype, differ from language to language. However here we adopt the following definitions[9]

[9]For the sake of simplicity, we refer here to a class system in which every class contains all features of every superclass. We also make the assumption that all objects of a given class

- a **class** is an abstract object, which specifies a particular set of *features*, including data fields and methods which are bound to objects of that class
- a **subclass** is a class which is related to another class, the *superclass*, by extension. That is, every object of the subclass contains all of the features of the superclass including (possibly different) methods with the same names
- a **type** is a finite set of classes
- a **subtype** is any proper subset of a type
- a **object** is an instance of a class

Objects are thus data. Every object has a class, and may belong to one or more types which contain that class in their defining set.

The relationship between types and their subtypes, and classes and their subclasses are independent. However most object oriented languages use the same syntactic mechanisms to define both, or perhaps do not even distinguish between the two concepts.

Nevertheless, from the point of view of implementation of object oriented dispatch, it is helpful to think of a type as being a particular class, together with all of the subclasses of that class. In this case, from the above definition, we may be sure that every object of that class contains a subset of methods which exist for every class in the type.

3.3.2 Languages with single inheritance

Languages in which classes may be defined as subclasses of a single class are said to have *single inheritance*. In such languages the class structure may be represented by trees, with every class possessing a single parent class, which we call the superclass.

With single inheritance it follows that every object of a particular class has (at least) the same features as its single superclass. In particular, objects of a subclass may define new method names, but they will necessarily contain methods with the same names as the methods of the superclass.

The implementation of method dispatch in such cases is simple. The reason is that we may assign fixed offsets to particular method names at compile time. If a class has several subclasses, then any additional methods of the subclasses will have higher offsets in the descriptor then the method names which are common to all. Given this property, we may produce code to index into the tables at compile time. Figure 3.21 shows a possible runtime structure for objects with shared method tables.

In this figure, three objects are shown on the left, two of the same class T, and a third of class $T1$. All objects contain an unnamed field which references the runtime class descriptor, which in turn encapsulates the method dispatch table. It is assumed that $T1$ is a subclass of T, so that the two classes have common

share the same representation. We therefore refer to monotone, representational classes.

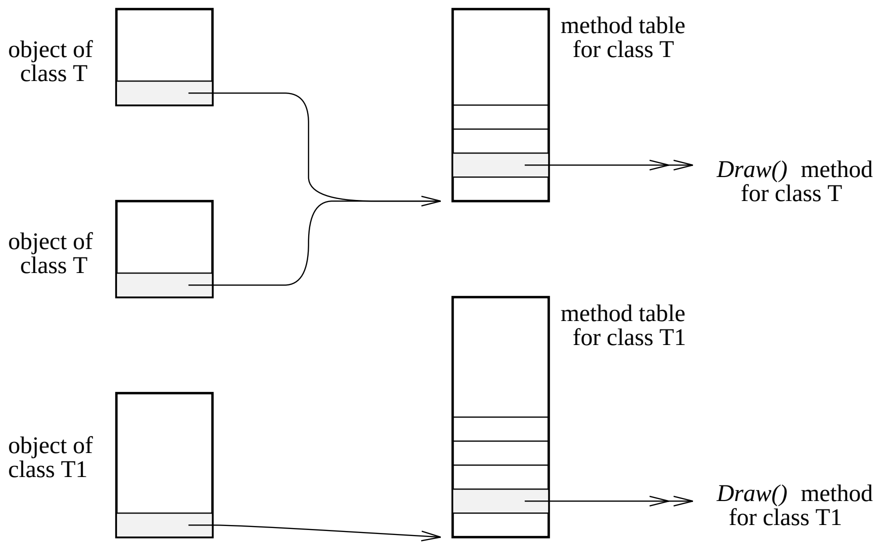


Figure 3.21: Method dispatch in language with single inheritance and shared method tables

method names, including a method called *Draw*. The two method tables place corresponding methods at the same table offsets. In order to access the *Draw* method of any object of a class belonging to the type, the steps are the same: follow the hidden pointer to the class descriptor, go to the known offset and call the procedure value found there.

3.3.3 Languages with multiple inheritance

Languages with multiple inheritance allow the definition of a class which is a subclass of more than one superclass. Different languages attach detailed semantics to such multiple-parent classes in several different ways. Here we are only concerned with the issue of object oriented dispatch in such languages.

The difficulty is that it is no longer possible to ensure that the offset of a particular named method is the same in a class and all of its descendants. Thus a more complicated way of navigating method tables must be used. For some languages this requires a second level of indirection.

An alternative implementation concept to that of using class descriptors also comes into contention in these cases. Viewed at the most basic level, method dispatch consists of querying the runtime class of an object, and accessing a table which maps from class and method name to procedure value.

$$\mathrm{DispatchTab} : \mathrm{Class} \times \mathrm{Name} \rightarrow \mathrm{ProcedureValue}$$

Of course, this mapping is very sparse, so that some kind of fast, sparse table lookup data structure is required. The method also requires that classes and names be identified by a globally unique denotation, a requirement which does not fit well with separate compilation of classes. Nevertheless, such schemes may be made quite efficient, particularly when they use a method cache which takes advantage of a sort of "locality of reference".

3.4 Error handling and exceptions

3.4.1 What is an exception

An exception is an abnormal event which occurs during the execution of a program, which prevents the program from continuing in its normal state. The act of signalling the detection of such an abnormal state is called *raising an exception*. Any attempt which the program or runtime system make to respond to the exception, by terminating the program in an orderly fashion for example, is called *handling the exception*.

Exceptions, in general have three different sources. There are exceptions which are detected by the hardware of the underlying machine. Most machines, for example, raise an exception if an attempt is made to perform a whole number division with a zero divisor. Exceptions may also be signalled by code which is implicitly produced by the compiler, as specified by the semantics of the source language. For example, if an attempt is made to assign an out-of-range integer to a subrange variable in *Pascal*-family languages, then an exception is raised. Finally, many languages provide for user code to explicitly raise exceptions, using syntactic constructs such as —

`IF` *Boolean-expression* `THEN Exceptions.RAISE(`*exception-id*`);`

Some systems may have hardware which detects an exception that other machines only detect using explicit tests in the object code. As an example, many *RISC* machines have instructions which trap integer overflows, but require software tests to detect unsigned overflows. Other machines are capable of trapping either signed or unsigned overflows. As an ideal, the possible responses of the runtime environment to exceptions raised from any one of these three sources should be identical.

Many exceptions are caused by errors, either in the code of the executing program, or by erroneous data. Thus much exception handling amounts to error handling of one kind or another. A full discussion of the possible responses to the raising of an exception is beyond our scope here, except for the impact which this has on runtime organization.

In general, when an exception is raised, the runtime system attempts to regain control of execution, and pass control to an *exception handler*. The exception handler is responsible for performing any corrections which are to be attempted before resuming normal execution, or to terminate the program in an orderly fashion.

3.4.2 Throwing and catching

In order to discuss the mechanisms of exception handling, it is necessary to settle on some kind of underlying primitive operations from which the detailed behaviour specified for different languages may be constructed. Here we shall assume just three common characteristics of the exception handling model with which we deal —

- **Static association of handlers with regions.** Every exception handling context is statically associated with a certain region of program execution, usually a compound statement or an entire procedure body. This "protected region" is called the *try-block*. Each such block has an *exception handler* associated with it

- **Automatic propagation to the caller.** In the event that the code which raises the exception is not statically within a try-block, control passes to the exception handling context of the caller of the code which raised the exception. In the event that there is no exception handler in the calling context, control passes to the exception handling context of the caller of the caller, and so on

- **Non-resumption semantics.** After a handler has completed, control does not continue in the raising code. Rather, control continues either from the end of the try-block, or at some explicitly specified retry point

The languages which fit this characteristic include *Ada*, *Modula-3*, `C++`, *Eiffel, Sather* and ISO-*Modula-2*. Of course, what happens inside the exception handlers for these various languages is quite different. What we are concerned with here is how control passes from code of the raiser to the entry point of the associated handler[10].

All of these languages have exception handling models which allow the definition of exception handling clauses in the syntax of the program source code. When an exception is raised, the exception is said to be *thrown*. Control then passes to the exception handling clause of the nearest dynamically enclosing procedure with a handler. The handler is said to *catch* the exception

Note that the association between the point of raising of the exception and the handler is *dynamic*, rather than depending on the static syntactic nesting structure of the code. This makes good sense, as it turns out. As a familiar example suppose that we wish to fold constant arithmetic in the static semantic analysis phase of a compiler. Immediately we are faced with a dilemma. We cannot allow the compiler to coredump with an overflow exception just because it is compiling code with an out of range expression. Instead we wish to produce an error message for the user, and continue with the semantic analysis. There are two approaches to this goal.

We might produce an arithmetic module which exports the arithmetic operations, and performs sufficient tests to determine from the operands whether or not an overflow will occur. Every procedure of the module would return a Boolean variable parameter, allowing an appropriate error message to be produced.

[10] Note that issues which have to do with concurrency, and asynchronous signals are orthogonal to the flow of control matters considered here.

Alternatively, we might simply go ahead and perform the arithmetic, but place an exception handler in the procedure which calls the expression evaluator. The handler can catch the exception thrown by the overflow, produce the error message, and then proceed with semantic analysis of the next expression. Note that it is not the procedure which raises the error which performs the exception handling and writes the error message. The raising procedure may be called from several different places in the compiler, and we might wish for the same exception to have different outcomes depending on the context. Thus we have a general abstraction principle: only the procedure raising the exception knows what the exception is, but only the caller of the procedure knows what an appropriate response is in any particular case.

It is usual for an exception handler to share the namespace of the procedure in which it is embedded. Thus the body of the procedure and the exception handler associated with it share the same activation record. It follows that between the throwing of the exception by the raiser, and the catching by the handler, zero or more activation records may need to be removed from the dynamic chain. In the case that the exception handling model does not provide for direct resumption of these contexts, these activation records may be discarded, and we speak of *unwinding the stack* to reach the stack frame with the handler.

3.4.3 Trap handlers

Given a uniform way of handling exceptions, it is necessary to transform hardware signalled exceptions, so that they follow the same control logic as exceptions thrown by software. It is usual to install *trap handlers* to catch such signals and to explicitly jump into the exception catching code. The details of such trap handling are very operating system specific.

3.4.4 Unwinding the stack

The process of throwing and catching an exception amounts, at the implementation level, to a non-local transfer of control.[11] The task of the runtime system is to restore the machine state to be consistent with that of the catching procedure. There are several aspects of this. Obviously, the state of the runtime stack must be such that the activation record of the catcher is on the top of the stack. However, we must also ensure that other aspects of machine state, such as the values of callee-saves registers have been restored. As an example, suppose we have the dynamic situation depicted in figure 3.22. Procedure A has called procedure B, which has an exception handler. Procedure B has called C, which has called procedure D. We assume that D has raised an exception.

The need to restore the callee-saves registers may be understood from this figure. Suppose, as an extreme example, that B needs few registers and saves no callee-saves registers in its prolog. Suppose however that C and D save some of these registers in their activation records. When the exception is caught by the exception handler of B, we must have already restored these values, otherwise

[11] Indeed, one of the most telling criticisms of exception handling as a programming language facility is that it provides a back-door way to place non-local **goto**s in programs.

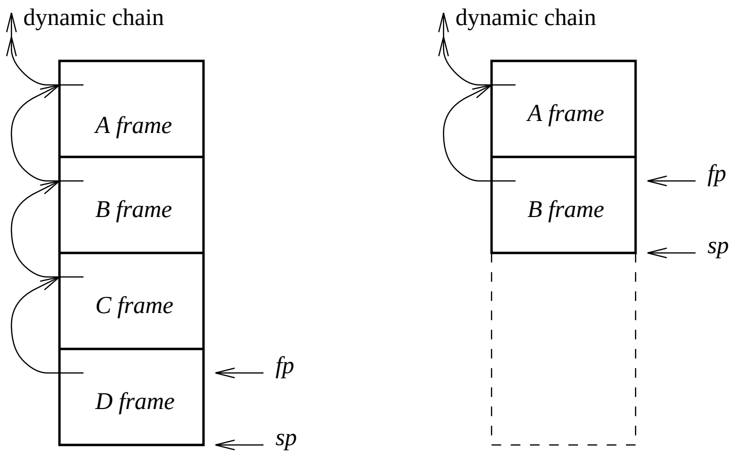


Figure 3.22: Exception handling example. Procedure D throws, procedure B catches

any subsequent return of B will fail to honor the obligation to return with callee-saves registers unchanged.

There are several ways of providing for such saving and restoring of procedure state. The most straightforward of these uses the equivalent of the primitive operations provided by the C-library functions "`setjmp`" and "`longjmp`".

Setjmp and longjmp

The functions "`setjmp`" and "`longjmp`" provide for non-local transfer of control in the standard C library. They do so by saving and restoring a substantial part of the stack and machine state in a machine dependent structure called a "`jmpbuf`". As used for non-local jumps, the use of these functions has the following structure —

```
jmpbuf state;      /* variable for state info */
...
if (setjmp(state) == 0) {
        ... control arrives here after initial call
} else {
        ... control arrives here after longjmp
}
```

The `setjmp` function is called with the allocated state buffer as parameter. The function saves all of the necessary state information in the jump buffer, and always returns from such explicit calls with a zero return value. Therefore, after every explicit call of the function control passes to the *then-part* of the if statement.

If at any subsequent stage of the program execution the call

`longjmp(state,`*number*`);`

is made, then the original call of `setjmp` "returns" again, with *number* as the return value. The explicit return value cannot ever be zero. The effect is that `longjmp` has "thrown" the value *number*, and the *else-part* of the if statement has "caught" it.

At one level, implementation is relatively straightforward. All callee-saves registers must be saved by the `setjmp`, as must the program counter, frame pointer, stack pointer, and possibly other data pointers. The call of `longjmp` simply restores the stack state and the registers from the jump buffer, and then jumps to the saved program counter address.[12]

Viewed at another level, the implementation and use of these facilities is problematic. Remember that the name "`setjmp`" is just another standard library identifier, and no special semantics are attached to the calling of such a function. In particular, it is semantically legal to use the name of such an integer function in an arbitrary expression. However, it is hazardous to do so. Indeed the *ANSI C* standard does not even guarantee that an assignment of the function return value will work correctly. Consider the following fragment of assembly code —

```
pushl myJmpBuf     ; push address of jmpbuf as param
save any caller-saves registers
call  setjmp       ; call the function, result in eax
restore any caller-saves registers
...
```

The problem is that after the `longjmp`, the code "restores" any caller-saves registers from their allocated temporaries in the caller's stack frame. But control has not necessarily stepped through the code as shown, indeed the same temporary locations may have been reused for the caller-saves registers of other calls. Thus after the long jump, the contents of caller-saves registers are unreliable. In particular, values corresponding to already evaluated subexpressions of the expression which contains the call of `setjmp` are suspect.

The problem may be avoided by adopting register allocation rules which simply do not permit values which are live across *any* function call to be allocated to caller-saves registers. This is an extreme but sufficient response.[13]

It is also important to remember that the jump buffer must be allocated such that the extent of the variable exceeds that of the code which contains the non-local jumps. The warning in the the standard *UNIX man* pages is worth repeating —

> If `longjmp(env)` is called even though `env` was never primed by a call to `setjmp(env)`, or when the last such call was in a function that has since returned, absolute chaos is guaranteed.

[12] This is yet another example of a *trampoline* procedure. It is called as a procedure, but "returns" by jumping to a location other than its real return address.

[13] In such a case, it is misleading to speak of caller-saves and callee-saves registers. Rather there are registers which are destroyed by calls, and others which are preserved over calls. The "no caller-saves" rule comes down to insisting that values which are live across any call must either be placed in preserved registers, or must be held in memory locations which are not reused for any other purpose within that procedure.

Building exception handling out of setjmp

In principle, it is possible to build exception handling using the facilities of `setjmp` and `longjmp`. In effect, every procedure with an exception handler allocates space for a jump buffer in its stack frame, and links this stack frame to a global handler list at entry. On exit, the handler is unlinked so the original head of the list once more becomes the *active exception handler*. The following code is inserted into the procedure, immediately following the prolog —

```
        ...
; end of standard prolog
        leal    NN(ebp),eax        ; load address of jmpbuf
        pushl   eax                ; push as param to setjmp
        call    setjmp             ; store the procedure state
        testl   eax,eax            ; is return value non-zero?
        jnz     handler            ; if so, jump to handler...
        link jmpbuf as head of handler list
        ... try-part body of procedure
        unlink jmpbuf from handler list
; start of standard epilog
        ...
        ret                        ; normal return
handler:                           ; handler body starts here...
        ...
```

Note that this use of `setjmp`, since it is completely controlled by the compiler, does not suffer from the problems which arise in the programmed use of the library function. Nevertheless, there are some disadvantages in organizing exception handling in this way. In particular, the overhead of setting and linking handlers may be quite considerable. This overhead is a penalty which is paid by every call of every procedure which contains an exception handler, even if the condition which the handler is intended to guard against is extremely rare.

As an ideal, we would wish the overhead of setting and removing handlers to be zero, or as close to zero as practicable. In this way a penalty would only be paid in cases where an exception actually did occur.

Low-overhead unwinding

The essence of low-overhead exception handling is the incremental reconstruction of the state information from each stack frame as the stack is unwound from the thrower to the catcher. In principle, the information is already present in the code and on the stack. After all, the normal chain of procedure return will restore the state in exactly this way.

When a procedure returns normally, state is restored by the cooperation of two mechanisms. In the code of the epilog, the offsets of saved data in the current stack frame are encoded. In the stack frame itself, another datum links the current stack frame to the frame of its caller.

Note that the process of incrementally unwinding the stack is necessarily inefficent. In figure 3.22 the frame of procedure D is discarded after performing the state restoration actions which are needed by D. Then the frame of C is discarded after the state restoration actions which C requires are performed. It is possible, even probable perhaps, that the state restoration actions involved for C will overwrite some of the state which has been restored for D. In effect, we may perform a significant amount of unnecessary work during stack unwinding. Nevertheless, it is our goal to make the exception-free execution run as efficiently as possible, even at the cost of making exception handling considerable less speedy.

In order to be able to unwind the stack incrementally, we must have access to information equivalent to the offsets encoded into each procedure's normal exit epilog, and we must be able to navigate the dynamic chain on the stack. As usual this information may be encoded either as data for interpretation, or procedurally. In the first case, a possible implementation would involve the creation of runtime procedure descriptors at compile time, and their loading into memory when the program is loaded. Each procedure descriptor needs to contain information on the number of registers which were saved, the locations in the stack frame at which they were saved, and the size of the stackframe. In this case, a routine in the runtime support system reads the data for the descriptor of the current procedure, and interpretively executes the state restoration operations as specified.

Alternatively, we might produce an additional sequence of object code for each procedure, which performs the state restoration increment when executed. We may think of these code segments as being *dummy epilogs*, since they perform the state restoration of a real epilog, but without the procedure return.

In the case of runtime descriptors, or in the case of dummy epilogs, note that *every* procedure must have one, and not just those procedures which have exception handlers.

The final problem which needs to be solved, is how to link together the procedure descriptors (or equivalently, the dummy epilogs) in the stack unwinding equivalent of the dynamic chain. An extremely simple solution would be to make every procedure place a pointer to its descriptor as part of the stack mark which it creates during the prolog. This method would have a space overhead equal to that of maintaining a static chain, that is, one extra pointer would need to be stored per stack frame. The execution overhead would be one extra instruction per procedure entry, where once again even procedures without exception handling would have to pay this price.

A second solution is in tune with the philosophy of making the overhead of setting handlers as low as possible, even at the cost of making the unwinding of the stack much slower. In this case, a runtime table maps instruction addresses to procedure descriptors. The compiler creates a symbolic version of part of this table as each module is compiled, and the linker resolves the symbols to their runtime addresses at link time. During unwinding of the stack, when the unwinding of one frame is complete, the return address stored in the frame is accessed. This address is mapped through the table, to find the descriptor of the calling procedure. The overhead of this lookup is quite high, since it is usually necessary to search hierarchically through a global, and then a per-module table to find the matching entry. The entries hold upper and lower bounds for the code addresses of each procedure. Figure 3.23 shows the address map structure.

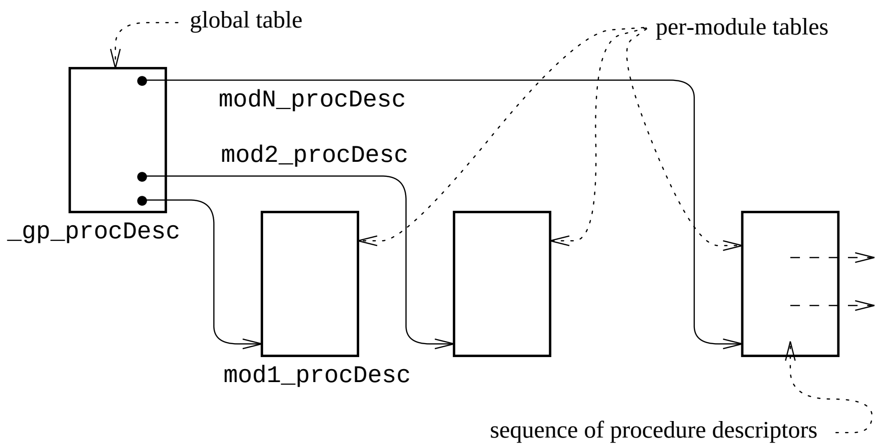


Figure 3.23: Two-level descriptor map, with global and per-module tables

The implementation described has a completely zero overhead, in the sense that there are no additional instructions which need to be executed in the normal entry or exit of any procedure. However, there is a subtle semantic point which needs to be treated correctly in such systems. Any local variables referenced in both the try-part and the handler part of a procedure must be stored in memory rather than in registers, since our compilers will be unable to uncover definition-use relationships which pass through the *secret control flow* of the exception mechanism. Thus all such variables must be treated as though they had the "`volatile`" attribute discussed on page 42. This semantic subtlety places a small but finite bound on the extent to which such exception handling may be said to be *zero overhead.*

3.5 Finding out more

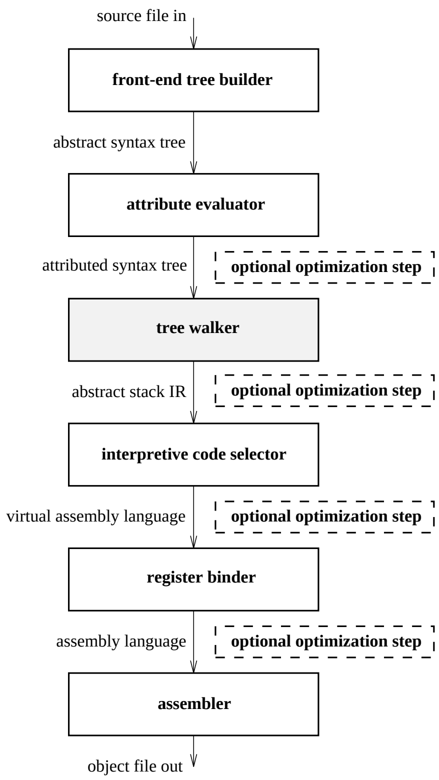
3.6 Exercises

3.1

3.2

Chapter 4

Generating Intermediate Code



4.1 Treewalking automata

The process of producing intermediate code from an abstract syntax tree (*AST*) consists of walking the tree, emitting the code as each node is visited. There are essentially two styles of intermediate code — those based on **tuples**, and those based on **abstract stack machines**. We shall be chiefly concerned with abstract stack machine forms, but it is as well to be aware that each style has both advantages and disadvantages. Most of the examples in this book uses the *DCode* representation detailed in the appendix.

It is important to discuss why it is useful to have an intermediate form at all. There are two distinct reasons why a compiler might omit this step. On the one hand simple compilers, particularly those which are targetted on a single architecture might omit this step in the interests of simplicity. The other circumstance occurs for those code generators which produce code by *bottom-up tree-rewriting*. For these code generators the trees themselves are treated as the intermediate form. Such code generators are considered in detail in chapter 6.

One of the aims of an intermediate representation (*IR*) is to provide a clean interface between the language dependent computations of the compiler front-end, and the machine dependent computations of the back-end. In principle, if a sufficiently capable intermediate representation can be found, a collection of n different front-ends (one for each language), and m different back-ends (one for each machine type), can be used to generate $n \times m$ compilers.

As it turns out, it is difficult to make a front-end completely independent of the target machine, at least for such fundamental matters as word-size. However, the changes required from one target to another are more a matter of parameterizing the front-end code rather than having a different front-end for each different machine. Similarly, it is difficult to make a back-end entirely language independent. For example, many optimizations require knowledge about which addresses could possibly *alias* other addresses. Such knowledge is very language dependent, since some languages permit addresses of operands to be taken in an almost unrestricted manner, while others specifically forbid such practices.

The goal of creating an intermediate representation which is suitable for *all* languages and machines is also somewhat problematical. It is not hard to use a common form for languages of similar type such as the *ALGOL*-based group, that is *Pascal*, *Modula-2*, *Ada* and *C*, and their object-oriented extensions. However, it appears to be difficult to find a single *IR* which spans languages as diverse as these and the functional languages such as Lisp, and the newer pattern matching functional languages. Even more specialized languages such as Prolog have their own specialized intermediate languages with instructions suited to the primitives of that language. It is interesting that the most widely used intermediate form for prolog is the *Warren Abstract Machine*, which is an abstract stack machine. However, the use of a stack is almost the only similarity with the abstract stack machines used for procedural languages.

Whichever form of intermediate representation is chosen, the production of intermediate code is performed by some set of recursive procedures. These procedures traverse the attributed syntax tree, emitting instructions in the intermediate representation. Such a set of procedures make up what we shall call a

tree-walking automaton.

4.1.1 Tuple intermediate forms

`<<still to come>>`

4.1.2 Abstract stack machine forms

Abstract stack machine intermediate code is the assembly language for some idealized stack-oriented machine. In such a machine the operands of instructions are implicitly on an evaluation stack. Binary operations are assumed to pop their two operands from the stack, compute the result of the specified operation, then push the result back onto the stack. Unary operations are assumed to pop their single operand from the stack, compute the result, then push the result back onto the stack. The only instructions which refer to memory explicitly are those which load a new value onto the stack, or which store the value currently onto the top-of-stack into memory. These load and store instructions are called **push** and **pop** instructions respectively.

As we shall see, expressions are evaluated by pushing values corresponding to the leaves of the expressions trees onto the stack, and computing the result as a value on the top of the stack. It is an entirely simple matter to generate the code to do this during a depth-first recursive traversal of the expression tree. In effect the traversal translates the tree into *postfix* form. It is the ease of this translation which is the chief advantage of abstract stack machine forms.

It is normal for the push and pop instructions to refer to memory in a way which is in some sense isomorphic to the actual runtime organization. Thus for the style of runtime organization which is conventionally used for Algol family languages we would expect to find instructions which push (that is, load) values from: statically allocated memory locations, known offsets within the local activation frame, and from uplevel activation frames.

Since it is usual for procedure activation frames to be allocated on a runtime stack, we must be scrupulous in distinguishing between the stack of the *abstract* machine, and the *runtime* stack of the target machine. The abstract stack appears only at compile time, it is an abstraction which has reality only as a data structure during code generation.

It is particularly important to distinguish between the abstract and runtime stacks when discussing procedure calling. As described in chapter 3, many machines pass parameters to sub-programs by pushing values, in order, onto the runtime stack. Since these parameter values will be evaluated onto the top of the abstract stack, we must have some explicit mechanism to emit an instruction to copy the top of the abstract stack onto the top of the runtime stack. Within *DCode* we have an abstraction which represents the transfer of parameter values to the required parameter locations, wherever they may be. There are two instructions which perform this action. `mkPar` transfers a single item from the abstract stack to the parameter location. It takes parameters which specify the size of the object and the offset in the assembly area. `blkPar` transfers an entire,

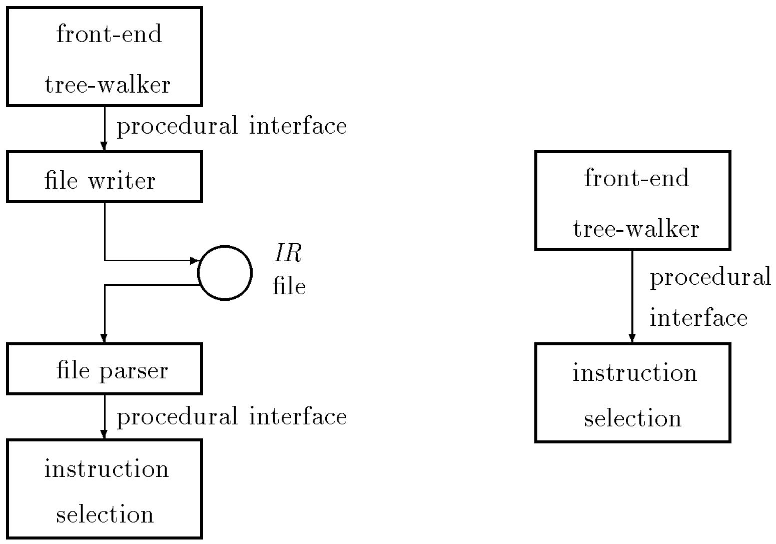


Figure 4.1: Structure with and without *IR* file

structured variable to the parameter location. It has parameters for size, offset and source alignment boundary.

File and procedural interfaces

The interface between the tree-walking automaton and the back end may be a file which contains some representation of the stream of *IR* instructions, or it may be simply a set of procedures in the back end which are called from the tree-walker. The difference between these two does not affect the way that the tree-walker operates. It thus makes little difference to the material of this chapter.

Let us suppose that the tree walking automaton in the front-end decides that some kind of *push* instruction is required. We will assume that this results in some *Push* procedure being invoked. In the case of a file based interface, the *Push* procedure will write a push instruction to the file, or buffer. In the back-end a file parser will recognize the push instruction, and call a *Push* procedure in the interpretive code generator. In the case of the procedural interface, the tree walker directly calls the *Push* procedure in the back-end. The process of writing to the file in the front-end and reading from the file in the back-end have simply disappeared. It is often convenient to generate a new compiler with a file interface, and to change to a procedural interface at some later stage in development (see figure 4.1).

In almost all compilers it is necessary to perform *some* kind of buffering of the instruction stream, at least on a per-procedure basis. We shall explore the consequences of this fact, and the options for resolving the problem. In order to motivate the discussion we shall, as an example, note that it is necessary to know the size of the procedure activation frame during the procedure prolog.

Now, the size of the stack frame cannot be computed in the front-end, since additional locations may need to be allocated for spilling registers. This need becomes apparent only in the final stages of code generation. We may attack this problem several different ways.

In the reference architecture used in this book, the virtual assembly language (*VAL*) is buffered for each procedure. After the *VAL* has been produced and the registers allocated, the size of the stack frame is finally known. At this stage the procedure prolog may be written to the output file, the instructions from the *VAL* buffer written to the output file, and the procedure epilog written. Since almost all compiler optimizations require whole procedure bodies to be buffered, this is the method of choice for optimizing compilers.

An alternative, used by many compilers which pass their output to fully-featured assemblers, is to leave the stack frame size as a symbolic value. In this alternative, the code generator emits code which looks as follows

```
.procentry _Example
        sub sp,FRAME,sp
;  procedure prolog
        ...
;  procedure epilog
.endp
#define   FRAME   28
```

The general facility for referring to macros which are defined later has been used to substitute the correct frame size into the resulting object code. In the case that the frame size is zero, the instruction may be transformed into a comment by the macro expansion.

Of course, this method simply postpones the need to buffer the code (or read the source twice) to the assembler.

All buffering may be avoided in very simple compilers which write their output directly to an object file, using two different methods. Firstly, it is possible for procedures to begin with a label, but for the procedure entry point and prolog to be written *after* the procedure body. In this case the prolog ends with a jump to the first label. This has some runtime cost, but is very simple for the compiler writer.

```
.procentry _Example
      jmp endLabel
realEntry:
      code body of procedure
;  procedure epilog
      restore saved callee-saves registers
      return
endLabel:
      sub sp,28,sp
      save required callee-saves registers
      jmp realEntry
```

It should be noted that when code is not buffered, the number of callee-saves registers which need to be freed is not known until after register allocation has been completed. Therefore this example defers the emission of code to save registers until after the code body has been emitted.

As a final alternative, an estimate of the frame size may be made, and this value immediately written to the output file. On those comparatively rare occasions that the size has been underestimated, a file position seek-and-overwrite is required.

In the most general structure, a tree walking automaton may produce one or more data streams as output during its traversal of the *AST*. The reference architecture used in this book has a single stream, but requires per-procedure buffering during the code generation process. The reference architecture also assumes that certain inter-procedural information is computed during semantic analysis of the *AST*.

Some compilers defer much of this computation to the back-end, by producing two output streams. Typically the first of these streams contains the *IR* for the body of each procedure, while the second has housekeeping information about each procedure, information on possible aliasing of addresses, and perhaps some interprocedural information. In such cases the second stream may need to be read to completion before the processing of the first may begin. In such a case the second stream may have a procedural interface, but the first must be buffered in a file.

Typed and unytped instructions

The operations of the intermediate code may be typed either explicitly or implicitly. For example the code generator may infer that an *add* instruction in an intermediate code stream should generate a floating point add operation if it is known that the top of the abstract stack currently contains a floating point operand. The advantage of using implicitly typed instructions in the intermediate representation is that the number of different operation codes is kept to an absolute minimum. However, it requires extra housekeeping operations in the code generator to track the type of each operand on the abstract stack. The intermediate form used for the examples in this book are explicitly typed. There are thus separate instructions for integer addition, and for single and double precision floating point addition.

4.2 Introduction to D-Code

The intermediate representation used in the examples in this book is called *DCode*, for somewhat obscure reasons. *DCode* is a very low level language as abstract stack machine languages go, since it does not even treat the *load* operations which push data onto the stack of the abstract stack machine as primitive operations. Instead the language has various *push address* operations, and various *dereference* operations. The justification for this rather unusual structure is the realization that with many of the new *RISC* machine architectures an address may be a valuable common subexpression. Thus an intermediate code which separates the addresses and the values which they point to provides additional opportunities for optimization.

The full *DCode* instruction set appears as the appendix A. Here we quickly overview a number of features of the *IR*.

4.2.1 Loads and stores

Load instructions

Data are pushed onto the stack of the abstract stack machine by instructions which push addresses of various kinds, and are dereferenced by various *derefX* instructions. There are three different kinds of addresses which occur. These are the addresses of statically allocated objects (*global addresses*), addresses of objects in the current stack frame (*local addresses*), and addresses of objects which are accessed via the display (*uplevel addresses*).

The instructions in *DCode* which push addresses are —

`pshAdr` *identifier* [±*offset*] – push the address of the statically allocated variable *identifier* onto the stack

`pshFP` ±*offset* – push the address at the given offset from the frame pointer fp ±*offset*

`pshDsp` *index,*± *offset* – push the address offset from the given display element by the given amount — *display*[*index*] ± *offset*.

Where particular operations require parameters, which all the pushes do, these follow the opcode. Optional parameters are shown in square brackets in these descriptions.

There is a dereference instruction for every different type of operand which the machine supports. These basic data types are *signed byte, unsigned byte, signed halfword, unsigned halfword, word, float, double.* All of the examples in this chapter use word-sized operands, in the interests of keeping the number of instructions small[1].

The instructions which dereference addresses on the abstract stack follow. In each case the dereferenced value replaces the address on the stack.

`derefSB` – dereferences the pointer to *signed byte* on the top-of-stack

`derefUB` – dereferences the pointer to *unsigned byte* on the top-of-stack

`derefS16` – dereferences the pointer to *signed 16-bit word* on the top-of-stack

`derefU16` – dereferences the pointer to *unsigned 16-bit word* on the top-of-stack

`derefW` – dereferences the pointer to *word* on the top-of-stack

[1] 64-bit extensions of *DCode* still use *word* for the "natural machine word size", but include an additional intermediate-sized pair of types – *unsigned mid-word, signed mid-word,* for 32-bit data

`derefF` – dereferences the pointer to *float* on the top-of-stack

`derefD` – dereferences the pointer to *double* on the top-of-stack

It is also necessary to push literals onto the stack: this is done by a push literal instruction, with just one special case —

`pshLit` *number* – push the literal number onto the stack

`pshZ` – push an integer zero onto the stack

Store instructions

The store instructions take the datum one below the top-of-stack, and assigns that value to the address on the top of the stack. Both data are popped. The instructions are —

`assignB` – assigns the least significant *byte* of the word second on the stack to the address specified on the top-of-stack

`assign16` – assigns the least significant *16-bit word* of the word second on the stack to the address specified on the top-of-stack

`assignW` – assigns the *word* on the second on the stack to the address specified on the top-of-stack

`assignF` – assigns the *float* second on the stack to the address specified on the top-of-stack

`assignD` – assigns the *double* second on the stack to the address specified on the top-of-stack

As well as these assign operations there is a block copy operation which provides for the assignment of entire variables. This instruction takes three arguments on the stack, and has an optional parameter which specifies any alignment constraint for the copying operation.

`blkCp` [*alignment*] – copies the number of bytes on the top of the stack from the address next to top, to the address below that on the stack. Pops all three arguments from the stack. The optional alignment will be 1, 2, 4, or 8, to indicate any, even, quad or octo-byte alignment respectively

4.2.2 Data operations

DCode contains the usual variety of arithmetic and bitwise logical operations. There is also a full set of arithmetic operations for single and double precision floating point values, and the expected format conversion instructions.

For all binary operations, the right operand is the one on the top of the stack, while the left operand is below that. In cases where it is convenient to evaluate the operands in a different order, it is possible to do so. For commutative operations no further consideration is required, while for non-commutative operations it is necessary to use the "`swap`" instruction to reverse the order before executing the operation.

Integer arithmetic

Integer arithmetic is always performed on word-sized quantities in *DCode*. It is assumed that memory variables of narrower storage size are widened to word-sized, either by zero extension or sign extension as appropriate, during the *deref*X operation which places the value on the abstract stack.

This corresponds with the common convention for *RISC* architectures, but ignores the capability of most *CISC* architectures, which provide arithmetic on byte and halfword quantities. The available operations are —

`abs` [*mode*] – takes the integer on the top of the abstract stack and replaces it by its absolute value. If *mode* is `intOver` then any attempt to find the absolute value of the most negative integer is trapped

`add` [*mode*] – adds the two words on the top of the stack with overflow trapping semantics as specified by mode

`div` *mode* – performs the div operation on the two words on the top of the stack with the signedness implied by the *mode*. The right operand is on top

`mod` *mode* – takes modulus of the two words on the top of the stack with the signedness implied by the *mode*. Right operand is on top

`mul` [*mode*] – multiplies the two words on the top of the stack with overflow trapping semantics as specified by *mode*

`negate` [*mode*] – negates the integer on the top-of-stack. If *mode* is `intOver` then any attempt to negate the most negative integer is trapped

`rem` *mode* – divides the two words on the top of the stack with the signedness implied by the *mode*, and returns the remainder. The right operand is on top

`slash` *mode* – divides the two words on the top of the stack with the signedness implied by the *mode*. The right operand is on top

`sub` [*mode*] – subtracts the two words on the top of the stack with overflow trapping semantics as specified by mode. The right operand is on top

Most of these arithmetic operations have a mode argument which is one of {`noTrap, crdOver, intOver`}. The mode indicates whether overflow checking is to be employed or not, and if so, whether the trapping semantics are based on signed or unsigned arithmetic. If the mode is optional, the absence of a mode has the same semantics as `noTrap`. It the mode is compulsory, it cannot be the `noTrap` value. The precise semantics of the division operations are defined in the appendix on page 288.

Bitwise logical operations

The bitwise logical operations form a functionally complete set.

`andWrd` – bitwise *AND* of two top-of-stack words

`bitNeg` – bitwise negate top-of-stack word

`orWrd` – bitwise *OR* of the two top-of-stack words

`xorWrd` – bitwise exclusive *OR* of the two top words

Boolean operations

DCode does not contain Boolean operations on logical values, since it is assumed that logical values will be evaluated by flow of control, as described later in this chapter. However, there is a Boolean negate operation. This is —

`boolNeg` – negates the Boolean value on the top-of-stack

Since the internal representation of Boolean values is $True = 1$ and $False = 0$, the same effect as the "`boolNeg`" instruction may be achieved by the instruction sequence —

```
pshLit 1
xorWrd
```

Some front-ends may produce the second form in the *IR*.

In cases where Boolean values are to be fully evaluated, the set "`andWrd, orWrd, boolNeg`" may be used to provide the Boolean operations.

4.2.3 Operations on addresses

There is a need to provide operations on addresses, to allow array indexing and record field selection. For many machines addresses may be manipulated using the ordinary operations on the unsigned integer type, particularly the ordinary "`add`" operation. However there are some advantages in distinguishing operations on addresses, since often the indexing can be "folded" into the addressing modes of the target machine. *DCode* provides the following operations.

`addAdr` – adds the word on the top-of-stack to the address below that

`addOff` $\pm$*offset* – adds the constant offset to the address on the top of the stack

In some cases it is necessary to perform more elaborate operations on addresses, for example comparisons of pointer variables for equality. For some machines the ordinary integer comparison operation is adequate. However, this is not the case for machines with segmented address spaces, such as *iapx86* in 16-bit mode. *DCode* provides primitives which convert between machine addresses and ordinal numbers so as to allow arbitrary address manipulations to be performed.

The first instruction abstracts the operation of turning an unsigned number into the internal machine address format. This instruction is called "`makeAdr`", it is the inverse of "`flatten`", which turns a machine address into an address ordinal.

As an example of the use of these instructions, the testing of pointer values for equality, in general, is as follows —

```
pshAdr   ptr1
derefW                  ; get pointer value 1
flatten                 ; make into address ordinal
pshAdr   ptr2
derefW                  ; get pointer value 2
flatten                 ; make into address ordinal
relEq                   ; relational test for equality
                        ; Boolean is now top-of-stack
```

The relational test pops the two top data, compares them, and pushes a Boolean result onto the top-of-stack.

The use of the flatten instructions is, of course, only necessary to make the *IR* portable to machines with segmented address architectures.

4.2.4 Comparison operations

DCode has a full repetoire of comparison operations, for both signed and unsigned types. These provide for tests of the predicates `>, >=, =, <=, <, <>`, and bitset membership. The set membership operation `setIn` applies to word-sized sets only, so that the indexing required for multi-word sets must appear explicitly in the *IR*. The *DCode*s are, respectively, `intGT, intGE, relEQ, intLE, intLT, relNE` for signed operands, and `crdGT, crdGE, relEQ, crdLE, crdLT, relNE` for unsigned operands. Set membership is tested by `setIn`, while `setLE` and `setGE` test non-strict set inclusion, $\subseteq, \supseteq$ respectively.

All of the binary predicates pop two operands from the stack, compute the Boolean result of the test, and push the result back onto the stack. The result of every comparison is thus an ordinal 0 or 1 on the top of the abstract stack.

4.2.5 Branch, jump and call instructions

Conditional branches

There are two conditional branch instructions —

- `brFalse` *target label* – branch if the top-of-stack word is zero, and pop the stack
- `brTrue` *target label* – branch if the top-of-stack word is non-zero, and pop the stack

Each of these take a label as operand.

Note that the interpretation of *False* and *True* implied by these instructions is slightly different to that used in the predicate instructions. This allows a useful optimization to be implemented in those frequent cases where comparisons with zero are required. For example, instead of translating "`IF x = 0 THEN`" as —

```
pshAdr   _x
derefW
pshZ
relEq
brFalse  elseLabel
```

we may use the test-against-zero directly implemented by the `brTrue` instruction.

```
pshAdr   _x
derefW
brFalse  elseLabel
```

Unconditional branches

There are three unconditional branches —

`branch` *target-label* – branch unconditionally

`exit` – unconditional jump to the procedure epilog

`switch` – indirectly jump to indexed address at jump table using top-of-stack as index. Performs a case statement jump (more details are given later)

Subprogram calls

There are three subprogram call instructions. These are —

`call` *identifier* – call the procedure

`popCall` – call the procedure whose address is on the top of the abstract stack and pop the abstract stack

`trap` *identifier* – same as `call` except no return is expected

The trap instruction is distinguished from the call only so that optimizers are able to construct more accurate flow of control information.

The parameter abstractions are handled by two instructions –

`mkPar` *size,offset* – pop the value on the top of the abstract stack, and pass as parameter of *size* bytes to the parameter area at the nominated offset. In case of machines which pass parameters on the runtime stack, the second value will be ignored

`blkPar` *size,offset,alignment* – pop the address on the top of the abstract stack, and copy the parameter block to which it points to the parameter area at the nominated offset. The *alignment* parameter, which is optional, indicates any alignment constraint of the source value

4.2.6 Stack housekeeping

A small number of housekeeping operations are necessary for the abstract stack. These allow values to be popped from the stack and discarded, duplicated, and swapped. These are —

- `dup1` – duplicates the top word on the stack
- `pop1` – pops the abstract stack by one word and discards the value
- `swap` – swaps the two top words on the stack

The `dup1` instruction is used to duplicate a value when a second copy is to be used later in the computation. This is needed, for example, if a value is to be computed, range checked, and then used. In general the range check will pop the tested value during the comparison, so a second copy is required.

The `pop1` instruction removes a value from the abstract stack, should the value not be required any further.

The `swap` instruction is used to reverse the order of the top two values on the abstract stack if for some reason it has been convenient to compute the two values in the reversed order. This might occur, for example, if the left hand side address of an assignment statement is computed before the right hand side value.

A common example occurs when producing code for the Modula statement *INC*(v,e), where v is some variable designator, and e is some expression. We know that we must produce two copies of the designator address, one to use as destination, and one to dereference to obtain the current designator value. However, we wish to compute the address only once. The following is an elegant solution, showing the abstract stack contents after each instruction.

```
push address of v        ;  &v
dup1                     ;  &v    &v
derefW                   ;  &v    v
push value of e          ;  &v    v     e
add                      ;  &v    v+e
swap                     ;  v+e   &v
assignW                  ;  empty stack
```

In the above, “&v” denotes the *address* of variable v, while “v” denotes the contents of the address.

Finally, there are a couple of instructions which allow for values to be moved from the abstract stack into temporary, off-stack locations. This is a convenience for values which are used multiple times. The instructions are —

- `mkTmp` *frmOffset* – move copy of top-of-stack to temporary at (*fp–frmOffset*)
- `pshTmp` *frmOffset* – push saved temporary at (*fp–frmOffset*) back onto stack

The make-temporary instruction non-destructively moves a copy of the top of stack value into a temporary. This temporary is pre-assigned a stack frame location (at *fp - frmOffset*) by the frontend. The advantage over using an explicit assign to the same location, is that even simple backends can take the hint and try to keep the copy in a machine register. The push-temporary instruction pushes the temporary from the nominated location.

Typical uses of this pair of instructions are discussed in section 4.3.4.

4.2.7 Procedure headers

DCode has a small number of constructs which specify the properties of each procedure. Procedure headers in the code specify the name, stack frame size, the parameter assembly area size (if required), and the lexical level (if required). The header is followed by a list of copy parameter records which specify the structured value parameters which need to be copied by the callee procedure. The exact form of the procedure header depends on the target machine conventions. Here are the headers for the same procedure for *iapx86*, configured for conventions which push parameters on the stack —

```
.PROC _StartHello(.SIZE=0,.NODISPLAY)
```

and for *MIPS* processors, which use fixed parameter assembly —

```
.PROC _StartHello(.SIZE=0,.NODISPLAY,.ASSEMBLY=8)
```

In each case the stack frame has zero size, as the procedure has no local variables. Since this is a procedure at the outer level there is no display manipulation required. In the case of the *MIPS* machine the outgoing parameters need to be assembled into a fixed area in the stack frame. In this case an assembly area of eight bytes is required, for the parameters of the *WriteString* procedure which this procedure calls.

The stack frame size which appears in the *DCode* header reflects the activation record size as it is known to the front-end. It is possible that the code generating back-end will enlarge this allocation, in order to make space for such things as register spilling.

4.3 Generating intermediate code for expressions

4.3.1 Ordinary expressions

Generating intermediate code for expressions is simplicity itself. A recursive procedure traverses the expression tree in the *AST*, emitting code for the subexpressions.

We shall make the following simplifying assumptions.

- the expression is represented by a tree
- the expression computes a word-sized value
- the expression is not of Boolean type (these are treated in section 4.4)

```
PROCEDURE PushValue(exp : ExprDesc);
BEGIN
  IF exp^.tag IN binops THEN
    PushValue(exp^.leftOp);
    PushValue(exp^.rightOp);
    EmitInstruction(tagToOpCodeMap[exp^.tag]); NewLn;
  ELSIF exp^.tag IN unarys THEN
    PushValue(exp^.childOp);
    EmitInstruction(tagToOpCodeMap[exp^.tag]); NewLn;
  ELSIF exp^.tag = literal THEN
    PushLit(exp^.litValue); NewLn;
  ELSE (* assert: tag = designator *)
    Designator(exp);
    EmitInstruction(derefW); NewLn;
  END;
END PushValue;
```

Figure 4.2: code of recursive expression traversal procedure

- there is a mapping from the expression node tags to *IR* op-codes which we shall assume is called `tagToOpCodeMap`

We assume that the expression is represented by a tree, rather than a *directed acyclic graph*, in the interests of simplicity. The more general case is treated later, in section 4.3.4.

The template of the traversal procedure is as given in figure 4.2. The procedure *Designator*() pushes the address corresponding to the designator. It is assumed that designators which are bound to literals have been transformed into literal nodes in the *AST* during tree-attribution. In figure 4.2 we have also ignored the need to output the *mode* marker for some instructions.

The model procedure as shown in figure 4.2 is able to handle relational operators using the tag-to-opcode map. In the case of the ordering operators such as *less than* this relies on the tree-attribution traversal having performed operator identification, that is, it must have been determined whether a signed or unsigned test should be performed. Not shown in the figure is the small amount of additional code which is required to deal with function calls.

4.3.2 Designators

In many languages, designators in expressions may represent several different kinds of object. In *Pascal*, for example, a designator may represent a variable, a manifest constant, or a function evaluation. We shall assume that during static semantic checking manifest constants have been bound to their compile-time values and transformed into literal nodes in the *AST*. We shall also assume that function evaluations have also been transformed into a distinct function-call node type. Thus at code generation time, designators in expressions represent variable accesses.

The process of generating *IR* code for designators has been abstracted away in the figure 4.2. Designators correspond to address expressions. We shall assume that designators are represented in the *AST* as sequences of terms, where each term is one of: a dereference operator, a constant offset, or an offset expression. The very first term in the sequence is a bound identifier which corresponds to some entire data object.

```
(* push base address of designator *)
PROCEDURE PushAddress(exp : ExprDesc);
BEGIN
  (* assert: exp^.tag is designator *)
  CASE exp^.mode OF
  | static :
      EmitInstruction(pshAdr);
      EmitName(exp^.desig.name); NewLn;
  | local  :
      EmitInstruction(pshFP);
      EmitInt(exp^.desig.varOffset); NewLn;
  | indirect :
      EmitInstruction(pshFP);
      EmitInt(exp^.desig.varOffset); NewLn;
      EmitInstruction(derefW);  NewLn;
  | uplevel :
      EmitInstruction(pshDsp);
      EmitInt(LexLevel(exp^.desig)); Comma;
      EmitInt(exp^.desig.varOffset); NewLn;
  | uplevIndirect :
      EmitInstruction(pshDsp);
      EmitInt(LexLevel(exp^.desig)); Comma;
      EmitInt(exp^.desig.varOffset); NewLn;
      EmitInstruction(derefW);  NewLn;
  END;
END PushAddress;
```

Figure 4.3: The *PushAddress* procedure

First, we shall describe the generation of address expressions for entire objects, that is, for simple designators without any selectors. We shall assume that during the tree-attribution traversal, each base object name has been decorated with an *accessMode* attribute. This attribute determines if the access is to a statically allocated object, an object in the local stack frame, an indirectly accessed object (such as a variable parameter), an object in an uplevel stack frame, or an indirect access to an object in an uplevel stack frame. The code is given in figure 4.3.

Note in figure 4.3 that it is necessary to distinguish the mode of access to particular entire variables. For example, the *IR* code used to generate the address of some local variable named x depends whether x is a variable parameter or an ordinary variable. The representation in the original program text does not distinguish these two case, but the *IR* code requires an additional dereference in one of the cases.

The process of forming designator addresses is best understood if the translation of individual selection operations is demonstrated first. Here are the simple cases, with the assumption in each case that the entire data object is some statically allocated variable —

```
; translation of a.b
    pshAdr   _a          ; push address of variable a
    addOff   NN          ; where NN is the offset of field b
```

Here is the code for a simple dereference —

```
; translation of a↑
    pshAdr   _a          ; push address of variable a
    derefW               ; replace by value of pointer a
```

Here is the code for a simple array index —

```
; translation of a[b]
    pshAdr   _a          ; push address of variable a
    pshAdr   _b          ; push address of variable b
    derefW               ; get contents of variable b
    pshLit   NN          ; where NN is size of array element
    mul      noTrap      ; compute offset of b element
    addAdr               ; compute address of a[b]
```

Of course, there are simplifications which apply when the element size *NN* is one, or is a power of two.

In the case that a designator may have several selectors, the tree-walker code requires a loop which traverses the sequence of selectors. Figure 4.4 has the details. Note carefully the code which tracks the type of the currently selected component, so that the element-size attribute may be accessed for indexing code.

4.3.3 Function calls

Function results are obtained by computing the parameters (if any) and emitting a call to the entry point name. Function calls are expected to leave their results on the abstract stack in the case of simple values. The previous chapter described the variations in the parameter evaluation process which are necessary for different machine architectures.

There are two cases to consider. In the *stack parameters* case, the parameters are evaluated by traversing the sequence of actual parameter expression trees, in the required left-to-right or right-to-left order. Each parameter must

```
PROCEDURE Designator(exp : ExprDesc);
  VAR typ : TypeDesc; (* current type *)
BEGIN
  PushAddress(exp); (* base address *)
  typ := exp^.desig.varType;
  FOR  every selector DO
    CASE selector OF
    | dereference :
        typ := typ^.targetType;
        EmitInstruction(derefW); NewLn;
    | fieldSelect :
        typ := selector^.field^.fieldType;
        EmitInstruction(addOff);
        EmitInt(selector^.field^.fieldOffset); NewLn;
    | indexSelect :
        typ := typ^.elementType;
        IF selector^.expression^.tag = literal THEN
          EmitInstruction(addOff);
          EmitInt(selector^.expression^.value * typ^.size); NewLn;
        ELSE (* variable index *)
          PushValue(selector^.expression); (* recurse *)
          PushLit(typ^.size); NewLn;
          EmitInstruction(mul); NewLn; (* no trap *)
          EmitInstruction(addAdr); NewLn;
        END;
    END;
  END;
END Designator;
```

Figure 4.4: Code of designator procedure.

be evaluated on the stack of the abstract machine, and then tranferred to the parameter location by a "`mkPar`" instruction.

In the case of *fixed assembly*, parameters evaluations which involve function calls must be evaluated first, before any values are moved to the parameter locations. Note that this restriction applies to some non-obvious cases, as well as to explicit function evaluations. Consider the case of a machine without a multiply instruction, in which multiplication is performed by a function call. In such a case, any parameter evaluation which involves a multiplication must be performed before any parameters are moved, just as for any other function evaluation.

Apart from the constraint that evalutions involving function calls must be performed first, parameters may be evaluated in any convenient order. A straightforward implementation strategy is to split the argument expression list into two. The list of arguments which involve function calls are evaluated, in turn, onto the stack of the abstract machine. When this first list is exhausted, all of the values are popped and moved to their specified offsets in the parameter assembly area, by a sequence of parameterized "**mkPar**" instructions. The expressions of the second list may be evaluated and moved one by one.

Just as *DCode* abstracts away the details of the mechanism by which parameters are moved from the abstract stack to the location required by the target architecture, so it is necessary to have some abstract way of specifying that the returned value is moved from the actual return location and pushed onto the abstract stack. The `pshRet`*NN* family of instructions perform this task. Here is the *DCode* arising from the source language expression $RealMath.sin(x)$ —

```
pshAdr    _x              ; address of actual parameter
derefD                    ; get fp double value
mkPar     8,0             ; move to parameter location
call      _RealMath_sin
pshRetD                   ; push return fp double value
```

It has been assumed in this example that some kind of *name munging*[2] has been used to convert the qualified name in the program into a unique name in the flat name space of the system linker.

4.3.4 Evaluating DAG expressions

The emission of expression code in situations where the expression is represented by a *directed acyclic graph (DAG)* rather than a tree, introduces some new issues. One rather simple case of this has been considered already, that of the *INC* statement in Modula. In that case the use of the `dup1` instruction provided a simple solution. However, in general, the existence of shared nodes in the expression representation requires a more flexible approach.

The idea behind the method presented here is to mark shared nodes prior to the generation of intermediate code. Such nodes are allocated a location in the stack frame which will be used in the corresponding `mkTmp` and `pshTmp` instructions. During code emission, when the node is visited for the first time, the value computed is copied to the temporary. Subsequent visits to the node do not recurse to the shared node's successors, but cause emission of a `pshTmp`.

As an example, consider generation of *DCode* from language *C*'s multiple assignment construct. In *C* the statement

```
v1 = v2 = v3 = exp;      /* assign exp to 3 int variables */
```

assigns the value of the expression to all three variables. This statement is not the same as

```
v1 = exp; v2 = exp; v3 = exp;      /* not the same */
```

either in terms of efficiency, or semantically, if the evaluation of the expression *exp* has side-effects. The *DAG forest* for this statement is shown in figure 4.5. If we suppose that the shared node is assigned stack location *fp*–16 the *DCode* becomes —

[2] Also known as *name mangling*.

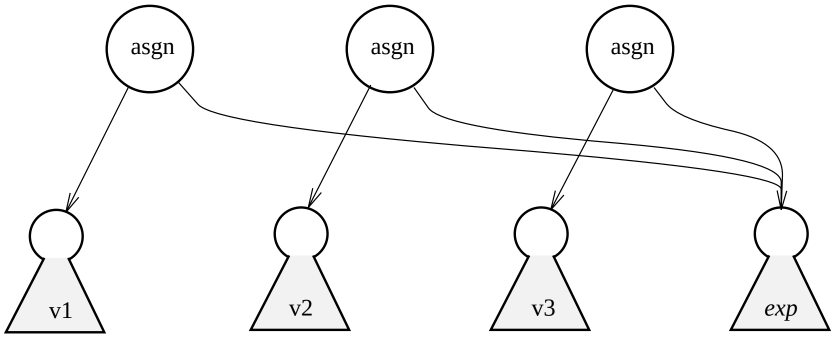


Figure 4.5: *DAG forest* arising from multiple assignment

push exp value	;	*compute exp once only*
`mkTmp 16`	;	*make a copy to tmp*
push v3 address	;	
`assignW`	;	*assign to v3*
`pshTmp 16`	;	*use the tmp*
push v2 address	;	
`assignW`	;	*assign to v2*
`pshTmp 16`	;	*use the tmp again*
push v1 address	;	
`assignW`	;	*assign to v1*

4.4 Generating code for Boolean expressions

Boolean expressions occur in two distinct contexts in programming languages. The most common case is that of expressions which are evaluated in order to affect flow-of-control. All of the Boolean expressions which occur in conditional statements fall into this category. The other possibility is that of evaluation of an expression so as to store the value, or pass it as a parameter to a subprogram. Most contemporary languages specify **short circuit** (lazy) evaluation of Boolean expressions[3], so that correct semantics are always obtained by evaluating Boolean expressions using what we shall call **jumping code**.

Jumping code evaluates expressions incrementally, and exits from the code to evaluate terms and factors just as soon as a result is known. The main advantage of this feature is not so much efficiency (it is not always the most efficient, as shown in an exercise at the end of this chapter) but the possiblity of writing guarded code such as —

```
IF  (ptr <> NIL) AND (ptr^.size <= 8) THEN ...
```

In the case of a language with full evaluation this would have to be written as the alternative —

[3] *Ada* is an exception, as it allows the programmer to specify either full evaluation (the default) or short circuit evaluation using the special operators '**or else**' and '**and then**'.

```
IF  ptr <> NIL THEN
  IF ptr^.size <= 8 THEN ...
```

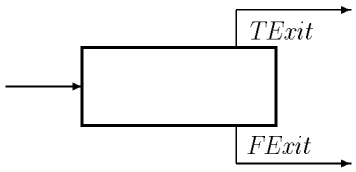
4.4.1 Simple jumping code

Jumping code is generated by a recursive traversal of the *AST* expression, being careful to evaluate strictly left-to-right as demanded by the desired semantics. The recursive procedure which generates the code should be passed the expression descriptor and two labels. These labels are the targets to which the control must jump in the event that a *false* or *true* result is determined. The expression signature is

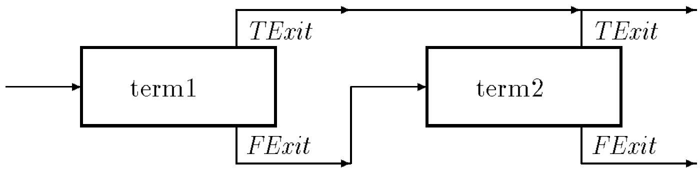
```
PROCEDURE BoolExpr(exp : ExprDesc; fLab, tLab : LabelType);
```

For example, for a simple *if* statement, the procedure would be called with the conditional expression as *exp*, and with the *elsepart label* and the *thenpart label* as the other two parameters. The interesting part of the code arises from the way in which this procedure calls itself recursively.

We shall introduce a graphical representation which makes the understanding of the code generation process simple. We represent the code which evaluates each subexpression as a black box with one entry and two exits. The exits are the *true exit*, which is taken when the expression evaluates to true, and the *false exit* which is taken otherwise.



These blocks are connected together in various ways in order to construct code for more complex expressions. Here is the diagram for an *OR* expression.



In this diagram the expression is comprised of two terms. In the case that the first term evaluates to *true* then control passes to the true label of the whole expression. That is, the evaluation is “short circuited” when a term evaluates to *true*. However, if the first term evaluates to *false* control passes to the input of the second evaluation. In order to achieve this effect we must allocate a label for the position which corresponds to the input of the second term-box. Suppose that this label is called `temp`. Now code for the expression is generated by a recursive call of —

```
AllocateLabel(temp);
BoolExpr(exp^.leftOp,temp,tLab);
EmitLabel(temp);
BoolExpr(exp^.rightOp,fLab,tLab);
```

Note that the call of *BoolExpr* which generates code for the second term uses the same labels as the entire expression. This is in accordance with the diagram shown above.

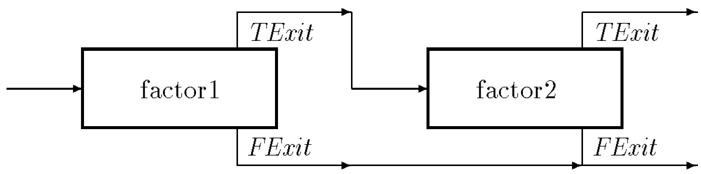
Suppose we have the expression `(a OR b)`, where a and b are variables of Boolean type. We wish to jump to `thenLab` if the expression is true, and `elseLab` otherwise.

The *DCode* generated by the above calls should be —

```
            pshAdr    _a
            derefUB
            brTrue    elseLab
            branch    lab1
    lab1:                           ; entry of second box
            pshAdr    _b
            derefUB
            brTrue    elseLab
            branch    thenLab
```

This code can clearly be optimized by eliminating redundant jumps and labels. This topic is the subject of the next section.

In the case of a term comprised of two factors with an *AND* connective the situation is dual. Here is the black-box diagram —



In this diagram the term is comprised of two factors. In the case that the first factor evaluates to *false* then control passes to the false label of the whole term. That is, the evaluation is "short circuited" when a factor evaluates to *false*. However, if the first factor evaluates to *true* control passes to the input of the second evaluation. In order to achieve this effect we must allocate a label for the position which corresponds to the input of the second factor-box. Suppose that this label is called `temp`. Now code for the term is generated by execution of —

```
AllocateLabel(temp);
BoolExpr(exp^.leftOp,fLab,temp);
EmitLabel(temp);
BoolExpr(exp^.rightOp,fLab,tLab);
```

Note that the call of *BoolExpr* which generates code for the second factor uses the same labels as the entire term. This is in accordance with the diagram shown above.

Suppose we have the expression `(a AND b)`, where *a* and *b* are variables of Boolean type. We wish to jump to **`thenLab`** if the expression is true, and **`elseLab`** otherwise.

The *DCode* generated by the above calls should be —

```
            pshAdr    _a
            derefUB
            brTrue    lab1
            branch    elseLab
    lab1:                                ; entry of second box
            pshAdr    _b
            derefUB
            brTrue    thenLab
            branch    elseLab
```

This code can clearly be optimized by eliminating redundant jumps and labels. This topic is the subject of the next section.

Finally, we may note that the generation of code for a *NOT* node is almost trivial —

```
BoolExpr(exp^.notOp,tLab,fLab); (* just swap labels *)
```

Since *NOT* complements logical values, the generation of code requires a single call to *BoolExpr* with the true and false labels simply swapped in parameter position!

4.4.2 Optimized jumping code

Jumping code may be optimized so as to generate the minimal number of labels. This is done by the introduction of a distinguished value of the label type named *fallThrough*. The use of this value as a parameter to *BoolExpr* indicates that control may pass to this location by simply falling through. For example, the code for an *if* statement may be generated so that a false value in the condition test causes a jump to the *elsepart*, while a true value simply falls into the start of the code for the *thenpart*.

Before we look at the changes to the generation of code for the binary operators for fall-through labels, we discuss the generation of code for the leaves of our expression trees.

Boolean expression tree leaves

The evaluation of Boolean leaves is performed by pushing a Boolean value onto the top of the abstract stack. In the case of Boolean variables, these are pushed onto the abstract stack in the same way as any other variable.

Relational tests are performed by the normal *PushValue* procedure. The two operands of the relational test are pushed onto the stack and the appropriate relational test code is generated. All these codes leave their Boolean result on the top of the abstract stack.

Once the value is pushed, we have two ways of generating the required jumps, using either `brTrue` or `brFalse`. In the case that both labels are ordinary labels it does not matter which of the following we choose.

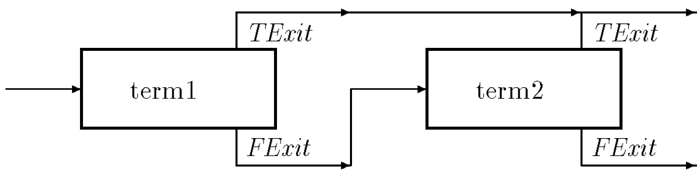
```
; first version
      brFalse   falseLabel
      branch    trueLabel

; second version
      brTrue    trueLabel
      branch    falseLabel
```

However, if one of the labels is the distinguished value *fallThrough* we may produce nicer code. Here is the fragment of the *BoolExpr* code which generates the code.

```
(* Bool is on top-of-stack, so ... *)
 IF tLab = fallThrough THEN
   EmitBrFalseTo(fLab);
 ELSIF fLab = fallThrough THEN
   EmitBrTrueTo(tLab);
 ELSE (* two real labels *)
   EmitBrFalseTo(fLab);
   EmitBranchTo(tLab);
 END;
```

Optimal code for binary operands

Let us look once again at the block diagram for an *OR* expression.



Since we generate code sequentially for each evaluation, it is clear that the false exit of the first evaluation should fall through into the entry of the second evaluation. We would thus expect to be able to omit the allocation of the label named `temp` in the previous section, and perform the following calls —

```
BoolExpr(exp^.leftOp,fallThrough,tLab);
BoolExpr(exp^.rightOp,fLab,tLab);
```

However, this is not quite all there is to it. It is possible that the true label which was passed as an actual parameter to the outer call of *BoolExpr* may have been the distinguished value *fallThrough*. A rather obvious observation is that we cannot use *fallThrough* as actual parameter of *both* formals in the first call of *BoolExpr*. Indeed we cannot *jump* to *fallThrough* out of the first term box. In this case we must allocate a new label *xLab* to jump to. The code for *OR* then becomes more complex —

```
IF tLab # fallThrough THEN xLab := tLab ELSE AllocateLabel(xLab) END;
BoolExpr(exp^.leftOp,fallThrough,xLab);
BoolExpr(exp^.rightOp,fLab,tLab);
IF tLab = fallThrough THEN EmitLabel(xLab) END;
```

The treewalking code is more complex, but the generated code is optimal. Suppose, as before, that we have the expression `(a OR b)`, where *a* and *b* are variables of Boolean type. We wish to fall through into the *thenpart* if the expression is true, and jump to the label `elseLab` otherwise. The initial call to *BoolExpr* will have actual false and true labels (`elseLab, fallThrough`) respectively.

The *DCode* generated by the above calls should be —

```
        pshAdr _a
        derefUB
        brTrue   lab1          ; lab1 is xLab
; else fall into second test on false
        pshAdr _b
        derefUB
        brFalse  elseLab
; else fall into thenpart label on true
lab1:                          ; start of thenpart
```

BoolExpr and PushBool procedures

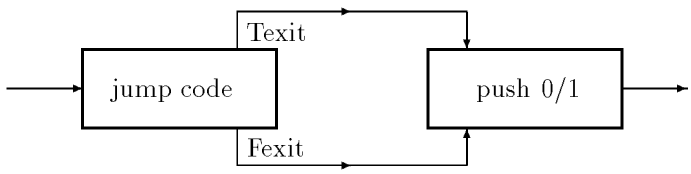
Figure 4.6 has the complete code for the *BoolExpr* procedure. This procedure is called whenever the transition has been made to jumping code. Within this procedure the generated code may revert to ordinary expression evaluation, if a simple Boolean value is required from a variable, function, or from a relational operator. This is seen in the *desigNode* case and the *elsepart* of the case statement.

We have not yet shown how the transition is made in the forward direction, from *PushValue* to the jumping code. Remember that figure 4.2 is a simplified version which glosses over this point. Occurrences of Boolean expressions as controls in conditional statements do not cause this transition to be made, since from the start the result of the evaluation is known to be a flow of control action. However, wherever Boolean expressions occur as right-hand-sides of assignments, or as actual value parameters in procedure calls, the *PushValue* procedure must decide how to compute the value onto the top of the abstract stack while respecting the short-circuit semantics.

The existing *BoolExpr* procedure is perfectly adequate for relational operators, Boolean valued designators and function calls, and even for the unary *NOT* operators applied to any one of these. However if the expression requires jumping code we must arrange for this flow of control to be turned back into a Boolean value on the top of the abstract stack. We may think of this diagramatically thus —

```
PROCEDURE BoolExpr(exp : ExprDesc;
                   fLab,tLab : LabelType);
  VAR local : LabelType;
BEGIN
  CASE exp^.exprClass OF
  | orNode :
      IF tLab # fallThrough
          THEN local := tLab ELSE AllocateLabel(local) END;
      BoolExpr(exp^.leftOp,fallThrough,local);
      BoolExpr(exp^.rightOp,fLab,tLab);
      IF tLab = fallThrough THEN EmitLabel(local) END;
  | andNode :
      IF fLab # fallThrough
          THEN local := fLab ELSE AllocateLabel(local) END;
      BoolExpr(exp^.leftOp,local,fallThrough);
      BoolExpr(exp^.rightOp,fLab,tLab);
      IF fLab = fallThrough THEN EmitLabel(local) END;
  | notNode : BoolJump(exp^.notOp,tLab,fLab);
  | desigNode : (* Boolean valued variables *)
      Designator(exp);                   (* address... *)
      EmitInstruction(derefUB); NewLn; (* now value  *)
      IF fLab = fallThrough THEN EmitBrTrueTo(tLab);
      ELSIF tLab = fallThrough THEN EmitBrFalseTo(fLab);
      ELSE EmitBrFalseTo(fLab); EmitBranchTo(tLab);
      END;
  | literalNode : (* unconditional branch !! *)!
      IF exp^.constValue = 0 THEN EmitBranchTo(fLab);
      ELSE (* ==> "true" *) EmitBranchTo(tLab);
      END;
  ELSE (* function calls, relational ops *)
    PushValue(exp);
    IF fLab = fallThrough THEN EmitBrTrueTo(tLab);
    ELSIF tLab = fallThrough THEN EmitBrFalseTo(fLab);
    ELSE EmitBrFalseTo(fLab); EmitBranchTo(tLab);
    END;
  END; (* case *)
END BoolExpr;
```

Figure 4.6: The complete *BoolExpr* procedure



For the sake of simplicity we shall assume that the code which joins together the two flows of control to produce a single Boolean value on the top of the stack is produced by a procedure *PushBool*. This procedure takes the *false* and *true* labels as parameters. For the moment, we shall assume that the two result values are merged on the top of the evaluation stack. The procedure is shown in figure 4.7. In the case that the *false* label is the special *fallThrough* value,

```
PROCEDURE PushBool(fLab, tLab : LabelType);
  VAR xLab : LabelType;
BEGIN
  AllocateLabel(xLab);
  IF fLab = fallThrough THEN
    EmitInstruction(pshZ); NewLn;    (* push a "false" value. *)
    << pop into off-stack location -- see note >>
    EmitBranchTo(xLab);
    EmitLabel(tLab);
    PushLit(1); NewLn;               (* push a "true" value.  *)
    << pop into off-stack location -- see note >>
    EmitLabel(xLab);                 (* Bool is now stack-top *)
    << push off-stack location -- see note >>
  ELSIF ...
  END;
END PushBool;
```

Figure 4.7: part of the *PushBool* procedure

the code produced is of the form —

```
; jump-code from BoolExpr (exp,fallthrough,tLab)
; beware, this is non-conformant DCode!
        pshZ                      ;   push FALSE result
        branch xLab               ;
tLab:
        pshLit 1                  ;   push TRUE result
xLab:
```

Other cases are explored in the exercises.

Interpretive code generators are simplified by the guarantee that the stack contents are identical along all paths reaching a label. This is not true in the case shown. Along each path the stack has a literal pushed, either zero or one, but at the label where the control flow merges the value magically must be treated as a variable! The simplest way to ensure that the values are correctly merged is to move them off-stack along each path, and reload the value below the merge. *DCode* provides a pair or primitive instructions for doing this, but the recommended solution is to create a local variable in the runtime stack frame, and merge the values in this location. In effect the Boolean expression has become a conditional assignment to the anonymous variable.

Using this method, the conformant version of the example would become —

```
; jump-code from BoolExpr (exp,fallthrough,tLab)
; this is the conformant version
           pshZ                    ;  push FALSE result
           pshFP NN                ;  assign to anon variable
           assignW                 ;
           branch xLab             ;
tLab:
           pshLit 1                ;  push TRUE result
           pshFP NN                ;  assign to anon variable
           assignW                 ;
xLab:
           pshFP NN                ;  fetch merged value
           derefW                  ;
```

In this example, the stack is empty at the label *xLab*, and the Boolean value is at (*fp*+*NN*) As will be seen later, most code generators will treat the extra *DCode*s as housekeeping and generate no extra instructions. The position of the extra code to produce this output is marked in figure 4.7.

4.4.3 Conditional expressions

As described in the last section, Boolean expressions may be treated as assignments of a conditional value. *Modula-2* has such expressions as the only form of conditional expression. However, conditional expressions occur in explicit form in certain other *Algol*-family languages. Language *C*'s conditional expressions are a familiar instance. Figure 4.8 is an example which shows the correct translation of the expression —

$$(exp_1 \, ? \; exp_2 : \; exp_3)$$

The frontend must allocate a temporary (in the example at (*fp*–24) in the local stack frame), and assign to this location along each path. The resulting value is pushed on the stack after the flow-of-control merge point.

4.5 Generating code for statements

4.5.1 Assignments and procedure calls

Code for assignment statements is produced in a straightforward fashion. The procedures required have already been met in this chapter. The expression on the right hand side of the assignment is evaluated by a call to *PushValue*, the address of the left hand side variable is pushed on the stack by a call to *Designator*, and an appropriate sized member of the `assign`*NN* family is emitted. Here is the code for the assignment `a := b + 1`, for word-sized operands.

```
        compute exp1 onto the stack
        brFalse  fLabel   ;  if zero, jump to else label
        compute exp2 onto the stack
        pshFP    -24      ;  fp-24 is an anonymous variable
        assignW           ;  assign exp2 to the anonymous variable
        branch   xLabel
fLabel:
        compute exp3 onto the stack
        pshFP    -24      ;  fp-24 is an anonymous variable
        assignW           ;  assign exp3 to anon variable
xLabel:
        pshFP    -24
        derefW            ;  value is on top-of-stack
```

Figure 4.8: Push the value `IF exp1 <> 0 THEN exp2 ELSE exp3 END.`

```
pshAdr    _b
derefW
pshLit    1
add                       ; right hand side is top-of-stack
pshAdr    _a              ; left hand side address
assignW                   ; do the assignment
```

Some front-ends try to minimize the height of the abstract stack, by using a *most complex first* strategy. In the case of very complex left hand side designators this will require that the assignment destination address be computed before the right hand side expression. Most languages allow such a choice of evaluation order. The produced *DCode* for the same example as before, but evaluated in reverse order would then be —

```
pshAdr    _a              ; left hand side address
pshAdr    _b
derefW
pshLit    1
add                       ; right hand side is top-of-stack
swap                      ; destination is now top-of-stack
assignW                   ; do the assignment
```

The final case of assignment code is assignment of entire structured variables. This is performed by emitting a "`blkCp`" operation code. This instruction has three operands on the stack: on the top-of-stack is the number of bytes of the block, below that is the source address, below that the destination address. All three words are popped. Here is an example, arising from the source statement `arr := "Hello world"` —

```
pshAdr   _arr         ; destination address
pshAdr   _cData+8     ; address of string
pshLit   12           ; number of bytes in string
blkCp                 ; do the block copy
```

Procedure calls

Procedure calls require the evaluation of parameters, if any, in some order appropriate for the target architecture, and the call of the procedure. The call conventions of some target machine and language combinations may require the caller to remove the parameters from the runtime stack following the return. Here is an example, for the *iapx86* architecture, arising from the procedure call `WriteString("abc")` using the language *C* calling conventions —

```
pshLit    3                     ; HIGH value for parameter
mkPar     4,0
pshAdr    _cData+12             ; address of parameter
mkPar     4,0
call      _InOut_WriteString
cutPars   8                     ; remove 8 bytes from stack
```

Note in this case that we may actually *evaluate* the parameter expressions in any convenient order, but the values must be moved to the runtime stack in exactly the order expected by the called procedure.

In the case of the *MIPS* architecture, the parameters are moved to a parameter assembly area, at specified offsets. Here is the code —

```
pshAdr   _cData+12             ; address of parameter
mkPar    4,0                   ; first parameter at offset 0
pshLit   3                     ; HIGH value for parameter
mkPar    4,4                   ; next parameter at offset 4
call     _InOut_WriteString    ; no cutback is needed
```

Note that in this case, since we specify the location of each parameter in the assembly area, we may evaluate and move the parameters to the assembly area in any convenient order (provided they do not involve function calls).

4.5.2 Conditional statements

The production of code for conditional statements fits very naturally with the techniques described for evaluation of Boolean expressions by jumping code. As an example, we consider the *if-then-elsif-else-end* structure. We shall assume that the *AST* has a representation of these structures corresponding to the following *IDL* fragment —

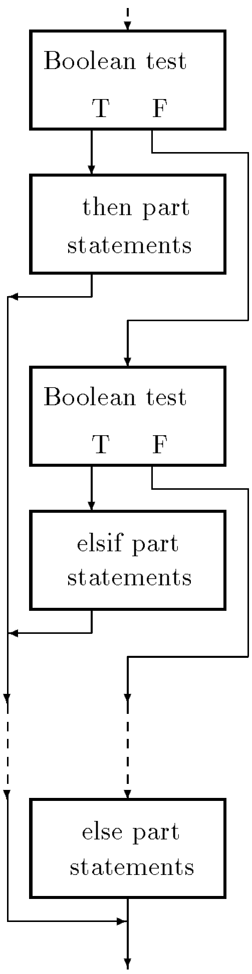


Figure 4.9: Block structure of if statement

```
STATEMENT   ::= ... | ifStatement | ... ;
ifStatement => branches  : seqOf CONDSTAT;
...
CONDSTAT    ::= ... | guardedSs | ... ;
guardedSs   => condition  : EXPR,
               statements : seqOf STATEMENT;
```

Each guarded statement sequence consists of a Boolean guard expression, and the statement sequence which is executed if the guard evaluates to true. Thus a single branch corresponds to each of the *then, elsif* and *else*parts of the original statement. For the *else*part we may adopt the convention that the condition expression is empty. We also assume that every such statement has at least one guarded statement with a non-empty guard.

The code structure which we wish to produce mimics the order of the source code with the conditional expressions interleaved with the statement sequences.

Figure 4.9 shows the block diagram of the code as it is laid out in the object

```
PROCEDURE EncodeIf(stat : StatDesc);
  VAR this : IfBranchDesc;
      exitLab, nextLab : LabelType;
BEGIN
  AllocateLabel(exitLab);
  get the first branch, assign it to this;
  WHILE this is not the last branch DO
    (* assert: this.condition <> NIL *)
    AllocateLabel(nextLab); (* next test start *)
    BoolExpr(this.condition,nextLab,fallThrough);
    StatementSeq(this.statements);
    EmitBranchTo(exitLab);
    EmitLabel(nextLab);
    get the next branch and assign to this;
  END;
  (* now the last -- maybe only -- branch *)
  IF this.condition <> NIL THEN
    BoolExpr(this.condition,exitLab,fallThrough);
  END;
  StatementSeq(this.statements);
  EmitLabel(exitLab);
END EncodeIf;
```

Figure 4.10: Outline of *IF* statement encoding

file. Figure 4.10 has an almost fully elaborated version of the algorithm. Note the special treatment of the last branch in this procedure. It is only the final branch which may have an empty condition. In this branch the control falls through into the final statement sequence if the condition is true, or empty, and the control falls out of the final statement sequence into the exit label, instead of taking an unconditional jump as for all other branches.

Case statements

Case statements are usually implemented by means of jump tables. It would be possible for a back-end to optimize low-density jump tables by transforming them into sparse table structures. However, an optimization of this kind is probably better done in the front-end. The number of entries in the jump table will be equal to the range of non-default ordinal values in the case labels. We shall assume that the range of values has been computed during the static semantic processing, as discussed on page 25. The generated code must index into the table at runtime, to find the address of the target code block, and jump to that location. Generating the code is simple. Generating the jump table is the more difficult task.

Here is a simple example —

```
    CASE value OF
    | 3,7 : x := 0;
    | 5   : x := 1;
    ELSE ABORT;
    END;
  END Proc;
```

The jump table will have entries for values from 3 to 7. The required intermediate code is as follows —

```
pshAdr   _value        ; get the selector
derefW
pshLit   3             ; subtract lower bound
sub
mkTmp    16            ; save the selector value
pshLit   5             ; entries index [0 ..  4]
crdGE
brTrue   defaultLab    ; jump to default label
pshTmp   16            ; a copy of selector value
switch   jmpTabLab     ; jump via jump table
```

For compilers which produce *AST*s, it is simplest to allow control to fall into the code which indexes into the jump table, and follow this code with the code for each separate case. The end of each case (for *Pascal*-like languages) is marked by an unconditional branch to the statement exit label.

Creating the jump table

The creation of the jump table requires a traversal of the sequence structure which represents the case list in the *AST*. This traversal can be merged with the traversal which emits the code for the various cases. During the traversal a *selector-value to label* mapping is created. Since we know the bounds of the domain, an array is a convenient data structure to use. At the start of the traversal an array is dynamically allocated which is large enough to hold the known number of jump table entries. All entries are initialized to the default label value.

As the statement sequence of each case is emitted a label is allocated as the entry point of each case's statement sequence. These labels are written into the jump table mapping at all locations which correspond to values in the case selector list for that case. At the conclusion of the traversal all map entries which do not appear in any case list will still select the default (else) case. The map may then be written to the output file as soon as all cases have been treated.

Sparse tables

`<< still to come >>`

Single pass restrictions

The creation of jump tables is rather more problematical for compilers which do not buffer code in an *AST*. In these cases the size of the jump table is not known until all cases have been processed. For such compilers it is usual to begin the case statement code with an unconditional forward jump to some label which follows all the case code.

The code for the various cases is directly emitted, and the *selector-value to label* mapping is built concurrently. Note that an array cannot easily be used for this purpose since the bounds are not yet known. Some other data structure must be used to store the mapping as it is created. Note that for such compilers the check that no case selector value is duplicated must be merged with code emission and table building.

After the code for all cases has been emitted the size of the jump table is known, and the table may be emitted. After this, the code which indexes into the jump table can be emitted. All cases thus require three jumps — one forward jump to the index code, the vectored, backward jump to the selected case, and the final, unconditional jump to the statement exit.

4.5.3 Various loops

In general terms, code generation for loops consists of emitting a *loop* label, then the code for the body of the loop, then a conditional or unconditional branch back to the loop label.

Loops and exits

Simplest of all is the structure for the simple endless loop of *Modula-2*'s *LOOP* statement, or C's `for(;;)`. The overall form of the code is shown in figure 4.11(a). In this case, the branch at the end of the loop is unconditional. We allocate two labels at the start — one for the *loop*-label, the other for the *exit*-label. Note that the exit label must be passed as a value parameter to the procedure which emits the statement seqence, so that any *EXIT* statements will have the correct target label. This is necessary, since *LOOP* statements may be nested, and an *EXIT* only exits from the innermost one.

The repeat statement

The *REPEAT* statement requires the post-tested repeat loop shown in figure 4.11(b). The test is evaluated with the true outcome causing control to fall through out of the loop. The procedure which emits this code is simplicity itself, see figure 4.12.

The while statement

The most straightforward translation of the *WHILE* statement would lead to the structure shown in figure 4.11(c). This may be achieved by a procedure

(a)	(b)	(c)	(d)

`loop:`
  loop body
  `branch loop`
`exit:`
	`loop:`
	  loop body
	  Bool eval
	  `brFalse loop`
		`loop:`
		  Bool eval
		  `brFalse exit`
		  loop body
		  `branch loop`
		`exit:`
			Bool eval
			  `brFalse exit`
			`loop:`
			  loop body
			  Bool eval
			  `brTrue loop`
			`exit`

LOOP	*REPEAT*	*WHILE*	*WHILE*

Figure 4.11: Various loop structures

```
PROCEDURE EncodeRepeat(stat : StatDesc);
  VAR loop : LabelType;
 BEGIN
   AllocateLabel(loop);
   EmitLabel(loop);                            (* loopLab:  *)
   StatementSeq(stat^.body);                   (* the body  *)
   BoolExpr(stat^.conExp,loop,fallthrough);    (* post-test *)
 END EncodeRepeat;
```

Figure 4.12: Writing out code for *REPEAT* loop

analagous to the post-tested loop. However there are advantages in using the alternative structure shown in figure 4.11(d). Note that the loop does not terminate with a double jump, thus saving a costly instruction execution. In addition, as we shall see in chapter 9, the testing of the condition prior to entry to the loop creates a safe position to which loop-invariant code may be moved. The preferred version does duplicate the code of the test, but this is usually not a concern. The code is produced as shown in figure 4.13.

```
PROCEDURE EncodeWhile(stat : StatDesc);
  VAR loop, exit : LabelType;
BEGIN
  AllocateLabel(loop); AllocateLabel(exit);
  BoolExpr(stat^.conExp,exit,fallThrough);  (* pre-test  *)
  EmitLabel(loop);                          (* loopLab:  *)
  StatementSeq(stat^.body);                 (* the body  *)
  BoolExpr(stat^.conExp,fallThrough,loop);  (* post-test *)
  EmitLabel(exit);                          (* exitLab:  *)
END EncodeWhile;
```

Figure 4.13: Writing out code for modified *WHILE* loop

(a) — "test, then increment"	(b) — "increment, then test"
`  ix := lower-bound`	`  ix := lower-bound`
`  t1 := upper-bound`	`  t1 := upper-bound`
`  IF ix > t1 THEN goto exit END`	`  IF ix > t1 THEN goto exit END`
`  t2 := t1 - step`	`loop:`
`loop:`	*statement sequence*
statement sequence	`  INC(ix,step)`
`  IF ix > t2 THEN goto exit END`	`  IF ix <= t1 THEN goto loop END`
`  INC(ix,step)`	`exit:`
`  goto loop`	
`exit:`	

Figure 4.14: Two ways of encoding a for loop, both wrong

The for statement

The for statement in languages such as *Pascal* and *Modula-2* is rather more difficult to encode than the other loops. This difficulty arises because of the precise semantics of the statement. There are two obvious ways in which a for loop might be encoded, shown in figure 4.14. Since the step size is a constant, the two ought to be equivalent, provided a suitable value is chosen for the limit value. Where the two actually are the equivalent, we should prefer the (b) version, since it uses a single branch statement in each loop.

In this section we will consider only loops which count upwards. The same reasoning applies to loops which count down, and we usually need separate case code for producing the output for the two directions.

The problem with for loops is that the *loop-control variable* must have some finite representation at runtime, and the representation may overflow when the increment is performed. Consider the loop

```
FOR ix := i TO j BY 10 DO ... END;
```

in the case that lower limit expression i has the value $MAX(INTEGER)$ - 15, and j is $MAX(INTEGER)$. On the third traversal of the loop, the increment operation in version (b) of figure 4.14 will overflow the control variable. This overflow will either cause the program to abort with a fatal error, or, if N is the number of bits in the word, will *wrap-around* the variable modulo-2^N. This will be a negative value, causing the loop to erroneously continue. Clearly both behaviours are incorrect.

Figure 4.14 version (a) is no better. If the lower bound i is $MIN(INTEGER)$, and the upper bound has value $MIN(INTEGER) + 5$, the initial computation of $j - 10$ will overflow, causing either a trap or a modulo-2^N *wrap-around*. Once again, both behaviours are incorrect.

There are many special cases in which one of the simple forms in the diagram is correct. This occurs, for example, when one or other of the limits is a constant, or belongs to some subrange. Equally well, of course, there are cases in which it is possible to show that the pre-test code is unnecessary.

(a)	(b)
ix := lower-bound	ix := lower-bound
t1 := upper-bound	t1 := upper-bound
IF ix > t1 THEN goto exit END	IF ix > t1 THEN goto exit END
t2 := t1 - step	loop:
loop:	*statement sequence*
statement sequence	INC(ix,step)
IF ix > t2 THEN goto exit END	IF *overflow* THEN goto exit END
INC(ix,step)	IF ix <= t1 THEN goto loop END
IF ix <= t1 THEN goto loop END	exit:
exit:	

Figure 4.15: Two ways of safely encoding the for loop

In the most general case both limit expressions are variable, and the range of possible values spans the whole range of the representation. In such cases, two tests are necessary for correctness. Figure 4.15 shows two separate ways of performing two tests. The choice between the two forms might depend on the availability of an overflow test instruction on the target machine.

4.6 Finding out more

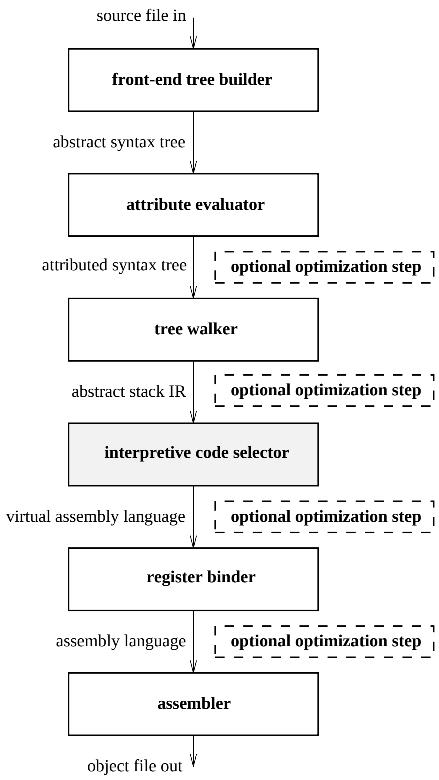
4.7 Exercises

4.1 <<showing cases where jumping code is worse>>

4.2 <<all the other cases of turning jumps into booleans>>

Chapter 5

Interpretive Code Selection



Interpretive code selectors are push-down automata. They take as input the strings of the intermediate representation (IR), and output strings in the target language. In our case, the target language is the virtual assembly language (VAL).

In the general case it is necessary for the automaton to have the language recognition power of a push down automatom (PDA), since the input language is not regular. The code selector has thus a potentially unlimited number of internal states.

The task of the code selector is to take sequences of instructions in the *IR* and to emit some sequence of instructions in the *VAL* which have the same meaning. In order for the code selector to produce good quality code it is necessary to accumulate as much state information as possible *before* it issues each instruction. The normal method of doing this is to maintain at compile time a data structure which we shall call a **shadow stack**. The shadow stack at compile time emulates the stack of the abstract stack machine. An example will help to make the process clear.

5.1 Code selection for a load-store architecture

The architecture which we shall use as the target for our first example has characteristics shown in figure 5.1. We defer all consideration of such things as non-scalar (and even floating point) variables until later.

We assume that the primitive data types of the target machine are declared as an enumeration —

```
TYPE Object = (byteInt,  byteCard,  (* 8-bit types  *)
               shortInt, shortCard, (* 16-bit types *)
               midInt,   midCard,   (* 32-bit types *)
               word,                (* machine word *)
               float, double)       (* ieee types   *)
```

We also ssume that the *DCode* instruction set, and the target machine instruction set are likewise declared as enumerations —

```
TYPE Instruction = ( ... ); (* DCode enumeration *)
     OpCode      = ( ... ); (* Target VAL enum.  *)
```

All operations on data are performed entirely on registers. Thus to perform operations on data in memory it is necessary to first load the operands into registers, operate on the registers, then store the result back into memory. For this reason such machine architectures are called **load-store**.

Binary operations such as "`add`" or "`sub`" take exactly three arguments, which are either all registers, or two registers and a literal. Unary operations such as "`move`" take two arguments which are both registers. In the first instance we will assume that comparison operations (*set instructions*), take three arguments, two register operands and a destination register. The destination register has a zero or one placed into it as the Boolean result of the specified comparison. Such a machine can then encode all possible flow of control on the basis of conditional branches on zero or non-zero register values.

Other *RISC* architectures have *condition codes* which are set as a side-effect of instruction execution, and have conditional branch instructions which depend on these condition codes. In principle, such an organization allows for execution time to be saved, since explicit tests can often be replaced by tests on side-effects of instructions which are already present. However, it is suprizingly difficult to achieve these savings, so we shall deal with the simpler case first.

The load instructions either load a datum from memory into a specified register, or load a literal into a register. The store instruction stores the value in the specified register into the specified memory location. We shall assume that the load and store instructions have exactly one address mode — *indexed indirect*. In this mode the instruction `lw` *offset*`(`*reg*`),`*dst* loads into the destination register *dst* the datum found in memory at the location given by the contents of register *reg* plus the constant offset *offset*. The only slight trick is that one of the registers, register *zero* or r_z as we shall call it, always contains the value zero[1]. Thus by using an offset of zero, ordinary indirect addressing is obtained. By using an index register of r_z absolute addressing is obtained.

At this stage in the process of code selection, the offset of a memory operand may be denoted as the sum of a symbolic name and a constant. At runtime, the offset will not be symbolic, but will be a constant. During compilation it may be necesary to refer to the operand as, let us say, "`extVar+4`". The linker program will work out exactly what constant corresponds to *extVar* at link time, and will adjust the executable file accordingly.

This machine may seem unduly simple, but is representative of the current selection of reduced instruction set (*RISC*) computers. The most widely available of these machines are the *MIPS Corporation's R3000* architecture and *SUN Microsystem's SPARC*. The described machine is almost exactly the *MIPS* processor. *SPARC* is only slightly more complex with *two* address modes, rather than the one which we consider here, and with condition codes instead of the *set* instructions.

5.1.1 The shadow stack

The key to the design of the shadow stack automaton is to have the elements of the shadow stack represent, in some sense, the possible operands of the instructions of the target machine. The stack elements thus belong to some union type.

In our case, we shall allow the elements of the shadow stack to be one of three variants — registers, literals, or symbolic addresses. Register elements on the virtual stack simply contain the tag, indicating that this element does denote a register, and an ordinal value which denotes the virtual register. It is assumed that there is an unlimited number of these **virtual registers**. It is convenient to represent the real machine registers as the first few, reserved values of the *VRegister* type. Literal elements on the stack simply contain the tag, and a word with the value of the literal. The structure of the symbolic address has three components. These are an identifier ordinal *ident*, an offset *off*, and an index register *reg*. The structure of the shadow stack elements is defined as follows —

[1]If r_z is used as destination, the result is discarded; when used as source, a zero is obtained.

opcode	description	format	example
load and store instruction examples			
li	load immediate	$I \rightarrow R$	li 255,r_{dst}
lw	load word	$M \rightarrow R$	lw var+4(r_{ix}),r_{dst}
la	load address	$M \rightarrow R$	la var+4(r_{ix}),r_{dst}
sw	store word	$R \rightarrow M$	sw r_{src},var+4(r_{ix})
binary operation instruction examples			
nor	logical not-or	$RR \rightarrow R$	nor r_{src1},r_{src2},r_{dst}
nori		$RI \rightarrow R$	nori r_{src1},255,r_{dst}
add	addition **mod** 2^{32}	$RR \rightarrow R$	add r_{src1},r_{src2},r_{dst}
addi		$RI \rightarrow R$	add r_{src1},427,r_{dst}
sle	set $r_{dst} = r_{src1} \leq$ op2	$RR \rightarrow R$	sle r_{src1},r_{src2},r_{dst}
slei		$RI \rightarrow R$	slei r_{src1},213,r_{dst}
miscellaneous operation instruction examples			
move	reg-to-reg move	$R \rightarrow R$	move r_{src},r_{dst}
beq	branch if equal	$RRL \rightarrow$ **void**	beq r_{src1},r_{src2},label7
jal	subroutine call	$M \rightarrow$ **void**	jal procXYZ
jalr	subroutine call	$R \rightarrow$ **void**	jalr r_{src}

Notes

- Format descriptors are in the form *inputs*$\rightarrow$*outputs*. In these formats, R denotes a register, I an immediate (literal) operand, L a (local) label, and M a (possibly symbolic) memory address. Instructions which do not have an explicit destination have the form *inputs*$\rightarrow$**void**
- In the examples, r_{dst} denotes the destination register, r_{srcN} denotes the N-th source register, r_{ix} a source register used as an index
- Many other instructions exist, and many more are constructed using special cases of the example instructions. For example arithmetic negation and branch on zero are instructions which are produced as follows —
  neg r_{src},r_{dst} is produced by sub r_z,r_{src},r_{dst}, and
  bez r_{src},*label* is produced by beq r_z,r_{src},*label*

Figure 5.1: Opcode formats

```
TYPE TagType = (literal,register,address);
     SymbolicValueDescriptor = RECORD
         CASE tag : TagType OF
         | literal  : litW : WORD;
         | register : vReg : VRegister;
         | address  : idnt : HashBucketType;
                      ofst : INTEGER;
                      iReg : VRegister;
         END; (* case *)
       END;
```

We also define an array of elements of the descriptor type which will act as the shadow stack, and will be the main data structure of the automaton.

```
VAR  shadow : ARRAY ArrayIndex OF SymbolicValueDescriptor;
     tos    : ArrayIndex;
```

It is assumed that there is a special value of the *HashBucketType* type which denotes no identifier. We will call this distinguished value *idNil*. Each of the other fields of the symbolic address belong to a type which already has a distinguished value. The constant zero denotes no offset, while the register r_z denotes no indexing[2]

We will assume that there are three special dedicated registers which simplify variable addressing. The frame pointer register r_{fp} points to the current frame of the runtime stack. Thus the variable at offset 16 in the current runtime stack frame would be represented as a symbolic address by the ordered triple (*idNil*,16,r_{fp}).

The second special register is the stack pointer register r_{sp}. This register points to the top of the runtime stack, and is used for indexing into the parameter assembly area (and possibly also into the rest of the stack frame if the frame size is fixed).

The third special register is the global pointer register r_{gp}. This register points to the global area of memory. In general, the offsets of named variables relative from this pointer are known only at link time. As an example, the third element of a character array named *str* would be denoted symbolically by the triple (*str*,2,r_{gp}).

It should be noted that in typical *RISC* architectures the instruction format places a restriction on the possible offsets which may be encoded. It may thus be necessary to change the value of the global pointer when executing different parts of the program, so that local static data may be accessed. For the purposes of the examples treated here, we shall assume that there is no limit on the allowed offsets, and that it is not necessary to indicate indexing by the global pointer explicitly. The appearance of a symbolic name will indicate to the assembler that the global pointer value must be included in the final object code.

5.1.2 Using the shadow stack

As a first example, we consider the following simple procedure body.

```
PROCEDURE DoAssign(current : CARDINAL);
BEGIN
  total := total + current;
END DoAssign;
```

In this example *total* is some statically allocated variable known to the linker. The parameter *current* is in the current stack frame at offset 8.

The resulting *DCode* sequence is —

[2]Note that the use of a *physical, machine* register with zero contents is a convenience only. In the case of machines such as the *iapx86*, which does not implement such a register, we can simply invent a special value of the virtual register type meaning *no register*.

```
pshAdr total        ; push address of total
derefW              ; get word pointed to by TOS
pshFP 8             ; offset 8 in current frame
derefW              ; get word pointed to by TOS
add                 ; do the add
pshAdr total        ; address of total again
assignW             ; do assignment
exit                ; do return
endp                ; end marker
```

We will follow the progress of the shadow stack, step by step, as each *DCode* instruction is issued.

At first, the stack is empty. After the first instruction the stack has exactly one element. No output is produced.

adr: $(total,0,r_0)$	← top-of-stack

The second *DCode* asks for the top-of-stack to be dereferenced. The result of this operation is to replace the address. In load-store architectures there is no point in delaying output emission once a dereference is encountered. The reason is that these machines have no instructions which operate on memory. Thus *every possible* instruction which uses this value will require the value to be in a register. Therefore it may as well be fetched immediately. In the case of machines such as the *iapx86*, in contrast, we would delay producing any output, hoping that the load operation could be folded into an address mode of the referencing instruction.

At this point it is necessary to decide on a destination register for the load instruction which is to be produced. In the case of code selectors which produce real assembly language (rather than *VAL*) it would be necessary to request a register allocator module to supply a register. Such an approach is called *on-the-fly* register allocation. The limitations of this approach are explained in chapter 10. In any case, we shall assume the existence of a register allocator which supplies a unique virtual register drawn from the effectively infinite set.

Suppose the newly allocated virtual register is called v_1. A *VAL* instruction (lw) to load the word at address *total* to virtual register 1, `lw total,`v_1, is now emitted. and the shadow stack is —

reg: v_1	← top-of-stack

The next two *DCode* instructions repeat the process except that the symbolic address refers to the frame pointer register. The instruction `lw 8(`r_{fp}`),`v_2 is emitted, and the shadow stack is —

reg: v_2	← top-of-stack
reg: v_1	

The `add` *DCode* is next. The operands for the instrucion are the two top elements on the stack, registers v_1,v_2. A further new register is obtained, and an add *VAL* instruction emitted. The instruction is `add` v_1`,`v_2`,`v_3, and the shadow stack is then —

reg: v_3	← top-of-stack

The next instruction pushes the symbolic address of *total* again, so that the shadow stack becomes —

adr: $(total, 0, r_0)$	← top-of-stack
reg: v_3	

The assign instruction causes the emission of a store word (sw) *VAL* instruction. This instruction, `sw v3,total`, stores the word in virtual register 3 into the memory at address *total*. The assign instruction pops the two top elements from the stack leaving the stack empty once again.

The final instruction, causes control to proceed to the procedure exit preamble, which unlinks the runtime stack frame and executes a return instruction.

The complete output of the body of this procedure is thus —

```
lw  total,v1
lw  8(fp),v2
add v1,v2,v3            ; destination is v3
sw  v3,total            ; source is on left
```

5.1.3 Internal structure of the shadow stack automaton

The shadow stack automaton consists of a set of procedures, and a small number of utilities. The utilities do such things as allocating virtual registers, and initializing descriptors. Each of the procedures *Init?Element*, where ? is *Reg,Lit* or *Adr*, takes as a first parameter the stack index at which the value is to be constructed. The remaining parameters are assigned to the fields of the value, in order. We shall also assume the existence of procedures *EmitRRtR, EmitRItR, EmitRtR, EmitRtM*, and so on. These procedures are responsible for emitting an instruction in the specified format *RR→R*, *RI→R*, *R→R*, *R→M*, etcetera.

There is one procedure for each instruction of the *IR*. Each procedure is responsible for updating the state of the automaton, and for determining if an instruction needs to be emitted. We shall consider four or five typical cases, still dealing only with our simple load-store machine.

The push instructions

The push instructions are relatively simple, and never cause an instruction to be emitted. They all increment the top of the shadow stack by one, and create a new symbolic value descriptor for the new top-of-stack element. For example, the procedure for pushing a literal value might be as follows.

```
PROCEDURE PshLit(val : WORD);
BEGIN
  INC(tos);
  InitLiteralElem(tos,val);
END PshLit;
```

The deref instructions

For these instructions and this architecture, as explained earlier, an instruction is always produced. The machine instructions, and the *DCode* instructions are assumed to belong to separate enumerated types. Assuming the existence of the obvious procedures, the code could be as follows.

```
PROCEDURE Deref(elem : Object); (* byte,word,etc. *)
  VAR ins : OpCode;
      new : VirtualReg;
BEGIN
  ins := LoadMap[elem]; (* maps element type to instr. *)
  NewVReg(new);         (* gets a newly allocated vreg *)
  MakeAddress(tos);     (* make tos into adr if needed *)
  WITH shadow[tos] DO EmitMtR(ins,idnt,ofst,vReg,new) END;
  InitRegisterElem(tos,new);
END Deref;
```

In this procedure, *EmitMtR* is a procedure which emits an instruction in the $M \rightarrow R$ format.

Up until now we have assumed the existence of a single data type, *words*. In the case of the load-store machines, the complexity added by the existence of other data types in memory is very slight. All objects in registers are of 32-bit word size, and all operations on registers operate on words. Thus smaller objects are widened to word size when they are loaded, and narrowed to their target size when they are stored. In the above procedure this additional factor is apparent. The lookup table *LoadMap*

$$LoadMap : Object \rightarrow OpCode$$

maps the primitive element types to machine instructions so that the value type *unsigned byte* maps into a *load byte unsigned* "`lbu`" instruction, while the type value *word* maps into "`lw`", and so on.

The procedure *MakeAddress* requires some discussion, as its operation is at the heart of the method. At the time that *Deref* is called, the top-of-stack element is not always a symbolic address. It might be a register value, as a result of loading a pointer value, or as a result of pointer arithmetic. In some source languages it might even be a literal value. The task of *MakeAddress* it to transform this element into a symbolic address, as required by the load instruction. The code performs a simple case analysis, and is shown in figure 5.2.

The assign instructions

The *Assign* instructions are similar in general terms to the *Deref* group. For our machine, the top-of-stack element must be a memory address, while the element below that must be loaded in a register.

At the time that *Assign* is called, the tos is not necessarily a symbolic address, and the stack element which denotes the right hand side of the assignment is not necessarily loaded into a register. The right-hand value might, for example, be the address of some object which is to be assigned to some reference datum. The code, by analogy with *Deref* becomes —

```
PROCEDURE MakeAddress(lev : StackIndex);
  (* post : shadow[lev] is a symbolic address *)
BEGIN
  IF shadow[lev].tag <> address THEN
    WITH shadow[lev] DO
      IF tag = register THEN (* overwrite as adr *)
        InitAddressElem(lev,idNil,0,regVal);
      ELSE (* tag = literal; so overwrite as adr *)
        InitAddressElem(lev,idNil,litW,noReg);
      END;
    END; (* with *)
  END; (* if *)
END MakeAddress;
```

Figure 5.2: The *RISC* version of *MakeAddress*

```
PROCEDURE Assign(elem : ValueType); (* byte,word,etc. *)
  VAR ins : OpCode;
BEGIN
  ins := StoreMap[elem]; (* maps element type to instr. *)
  MakeAddress(tos);      (* make tos into adr if needed *)
  MakeRegister(tos - 1); (* load next element into reg. *)
  WITH shadow[tos] DO
    EmitRtM(ins,shadow[tos - 1].vReg,idnt,ofst,iReg);
  END;
  DEC(tos,2);            (* remove two top stack elems  *)
END Assign;
```

The code for loading stack values into a register, *MakeRegister*, is similar to that for *MakeAddress*, with one important exception. *MakeAddress* only updates the state of the shadow stack, *MakeRegister* may additionally emit an output instruction. Figure 5.3 has the procedure. We shall assume that there is a similar procedure *MakeRegOrLit* which loads the stack argument if it is neither a literal nor a register.

Assign is thus an unusual case, since it is possible that a single *IR* instruction may cause *two* machine instructions to be emitted. This exceptional case arises, for example, whenever a literal is assigned.

The addAdr instruction

For load-store architectures with a flat 32-bit virtual address space it is not necessary to distinguish between address arithmetic and data arithmetic, since the semantics of each are the same. This is not the case for machines with segmented addressing, where address arithmetic has different semantics.

Nevertheless, if our *IR* distinguishes the case of adding addresses and adding data elements, we may use this to treat some special cases. The *addAdr* instruction adds a numeric value on the top-of-stack to an address below that. The address, as usual, may or may not be symbolic. The procedure in figure 5.4

```
PROCEDURE MakeRegister(lev : StackIndex);
  (* post : shadow[lev] is loaded *)
  VAR new : VRegister;
BEGIN
  IF shadow[lev].tag <> register THEN
    WITH shadow[lev] DO
      NewVReg(new);         (* get a new reg *)
      IF tag = address THEN (* emit load adr *)
        EmitMtR(la,idnt,ofst,iReg,new);
      ELSE (* tag = literal so emit load lit *)
        EmitItR(li,litW,new);
      END;
      InitRegisterElem(lev,new);
    END; (* with *)
  END; (* if *)
END MakeRegister;
```

Figure 5.3: The no-condition-codes version of *MakeRegister*

shows how this is done. Note that in several of the special cases it is possible

```
PROCEDURE AddAdr();
  VAR new, old : VRegister;
BEGIN
  MakeAddress(tos - 1);
  (* now try to fold the add into an address mode *)
  WITH shadow[tos - 1] DO
    CASE shadow[tos].tag OF
    | literal :  (* just add to offset *)
        ofst := ofst + CAST(INTEGER,shadow[tos].litW);
    | register : (* fold into index, if possible *)
        old := shadow[tos].vReg;
        IF iReg = noReg THEN iReg := old;
        ELSE (* emit an add instruction *)
          NewVReg(new);
          EmitRRtR(add,iReg,old,new);
          iReg := new;
        END;
    | address : Assert(FALSE,"bad tos element");
    END; (* case *)
  END; (* with *)
  DEC(tos);
END AddAdr;
```

Figure 5.4: The *RISC* version of *AddAdr*

to avoid the emission of even a single instruction. Even in the case where an instruction is emitted, that instruction simply adds the two index registers, but leaves any offset unchanged.

If the new top-of-stack address value subsequenctly dereferenced by a *load* instruction, then the symbolic addition of the offset will have been performed at zero cost. However, if it should happen that the new top-of-stack address is loaded into a register, rather than dereferenced, then the deferred add instruction will be emitted at that time.

By contrast a simple add, where it is not known that the result is an address, could take two instructions. One to load the address operand, and then another to add the top-of-stack register to the loaded address.

Finally, we look at the selection of instructions for binary operations.

The add instruction

The "`add`" *DCode* calls for the two top elements of the shadow stack to be added together. For our target machine, the left operand must be in a register, while the right operand may be either a register or a literal. There is a special case which benefits from special treatment. If the *left operand* is a literal, then we will save an instruction if we use the commutative property of addition, and swap the two top-of-stack elements. It is assumed that the front-end will already have eliminated the case of *both* operands being literals. Thus we have the following code in figure 5.5. This case is a model for all the commutative

```
PROCEDURE Add();
  VAR new : VRegister;
BEGIN
  NewVReg(new);
  IF shadow[tos - 1].tag = literal THEN SwapTopElements() END;
  MakeRegOrLit(tos);
  MakeRegister(tos - 1);
 (* --------- possible stack top elems ---------- *
  *    literal <-- tos        register <-- tos    *
  *    register               register            *
  *    ...                    ...                 *
  * -------------------------------------------- *)
  IF shadow[tos].tag = literal THEN
    EmitRItR(addi,shadow[tos - 1].vReg,shadow[tos].litW,new);
  ELSE (* do register to register add *)
    EmitRRtR(add,shadow[tos - 1].vReg,shadow[tos].vReg,new);
  END;
  DEC(tos);
  shadow[tos].vReg := new;
END Add;
```

Figure 5.5: The *RISC* version of *Add*

binary operations. The non-commutative ones simply omit the test and swap at the procedure entry.

This is almost all there is to it, at least for load-store machines. However, before we proceed it is as well to consider a more complex example.

Using the shadow stack: complex address expressions

Consider the following statement, which is the first statement of the inner loop of a matrix transpose procedure. The variable *array* is assumed to be a *VAR* parameter, which points to the actual array to be transposed. For simplicity, it is assumed the size of the array is a constant known to the compiler, so that access to *HIGH* values are not required.

```
temp := array[i,j];
```

The resulting *DCode* sequence is —

```
pshFP 8          ; 1   push address of param
derefW           ; 2   get value of the param
pshFP -4         ; 3   address of local var i
derefW           ; 4   get value of i
pshLit 64        ; 5   cardinality * element size
mul              ; 6   multiply by array size
addAdr           ; 7   add offset to address
pshFP -8         ; 8   address of local var j
derefW           ; 9   get value of j
pshLit 4         ; 10  element size
mul              ; 11  multiply by elem size
addAdr           ; 12  add offset to address
derefW           ; 13  get array element
pshFP -12        ; 14  address of local var temp
assignW          ; 15  do the assign
```

For this example the instructions have been numbered for ease of reference. The output instructions are as follows, and are emitted at *DCode* indices as indicated —

```
lw 6(fp),v1      ; (2)
lw -4(fp),v2     ; (4)
muli v2,64,v3    ; (6) or: shl v2,6,v3
add v1,v3,v4     ; (7)
lw -8(fp),v5     ; (9)
muli v5,4,v6     ; (11) or: shl v5,2,v6
add v4,v6,v7     ; (12)
lw (v7),v8       ; (13)
sw v8,-12(fp)    ; (15)
```

Comments in the *VAL* show where it would be possible to perform a *strength reduction* by converting a relatively expensive operation (a multiplication "`muli`") into a relatively cheaper one (a left shift "`shl`"). It is possible to recognize the special case of multiplication by powers of two during the tree walk, in the shadow stack automaton, or during the final code output process.

Clearly, for these load-store architectures the simplicity of the machine model leads to a shadow stack automaton which is simple and elegant. As we shall see later, this constrasts to the difficulties which more complex machine models can place in the way of good quality code selection.

5.1.4 Flow of control: the branch instructions

In our abstract machine model Boolean expressions are evaluated by instructions which leave either a zero or one value on the top-of-stack. The flow of control instructions use this Boolean value for branching, using the "`brTrue`" and "`brFalse`" instructions of the *IR*.

Some *RISC* architecture machines provide machine instructions which match this model very well. For machines which provide a complete set of test instructions and a limited range of branch instructions, a direct translation of the *IR* instructions is sensible. Figure 5.6 is an example procedure. Note that there

```
PROCEDURE IntRelLE(); (* signed <= *)
  VAR code : OpCode;
      newV : VRegister;
BEGIN
  NewVReg(newV); (* allocate dst register *)
  IF shadow[tos - 1].tag <> literal THEN
    code := sle; (* set if <= *)
  ELSE (* do swap *)
    SwapTopElements();
    code := sgt; (* set if > *)
  END;
  MakeRegOrLit(tos);
  MakeRegister(tos - 1);
  IF shadow[tos].tag = literal THEN
    EmitRItR(code,shadow[tos - 1].vReg,shadow[tos].litW,newV);
  ELSE
    EmitRRtR(code,shadow[tos - 1].vReg,shadow[tos].vReg,newV);
  END;
  DEC(tos);                   (* pop one operand *)
  InitRegisterElem(tos,newV); (* overwrite other *)
END IntRelLE;
```

Figure 5.6: A typical *RISC* machine relop procedure

is a little cleverness here. In our target machine there are *set* instructions with *RR*→*R* and *RI*→*R* formats, so that we will save an instruction by swapping operands in the case that the left operand is a literal. Of course, the actual instruction then must be adjusted so as to still produce the correct Boolean result.

The translation of *brTrue* and *brFalse* is just an instruction which branches on the tos register being not-zero or zero respectively. This does not merit further discussion.

The traditional case of machines with *condition code* or *flag* registers is somewhat more involved. This case is treated in section 5.2.

5.1.5 Flow of control: the subprogram calls

The selection of code for the calling of subprograms is dependent on the machine conventions for the passing of parameters and returning function results. As a first, simple example we shall treat a somewhat mixed model. We shall assume that parameters are passed on the stack, with the *last* parameter being pushed *first*. We shall further assume that the *calling* procedure is responsible for removing these parameters from the stack after the return. These are the normal language *C* conventions, for machine conventions which pass parameters on the runtime stack. We shall assume that the machine returns scalar function results in a known register r_{ret}[3].

Some care is needed in trying to understand how code is generated for parameter passing. For the first time we need to keep *two* stacks in our minds — without mixing them up! There is the stack of the abstract machine which our shadow stack emulates, and then there is the real, runtime stack of the target machine. We shall be pushing and popping both.

The standard *DCode* sequence for procedure calling consists of a sequence of expression evaluations, which leave their results on the top of the abstract stack. After each expression evaluation a special "`mkPar`" instruction is placed in the *DCode*. This instruction calls for the top element of the *abstract* stack to be pushed on the *concrete* or runtime stack. There is a parameter which indicates the size of the parameter on the concrete stack. This is necessary since we may evaluate an expression into a 32-bit register, for instance, but need to pass it as a 16-bit quantity to the called procedure.

The code is relatively straightforward —

```
PROCEDURE MkParam(size : CARDINAL);
BEGIN (* assert : size is either 2 or 4 *)
  MakeRegister(tos);
  IF size = 2 THEN                     (* 16-bit word *)
    EmitRt_(pushw,shadow[tos].vReg);   (* short word  *)
  ELSE                                 (* 32-bit word *)
    EmitRt_(pushl,shadow[tos].vReg);   (* long word   *)
  END;
  DEC(tos); (* remove from abstract stack *)
END MkParam;
```

Generating the actual procedure call itself is quite trivial. There are two cases. Calling of the named procedure *name* causes a `jal` *name* instruction to be emitted. In the case of the call of procedure variables, the procedure *value* is evaluated onto the top of the abstract stack, which is then popped and called. This means that the top-of-stack must be loaded into a register and a "`jalr reg`" instruction emitted.

[3] We have departed here from the usual conventions for load-store machines. These invariably pass their first few parameters in registers. We show how to deal with this variation in a later section.

After the generation of the call instruction an instruction must be generated to pop the concrete stack. For this reason, the `cutPars` *DCode* has a machine dependent parameter — the amount by which the concrete stack pointer must be adjusted.

The returned value of function procedures is somewhere off-stack in our abstract machine model. In the *DCode*, function calls are followed by a `pushRet` instruction which pushes the returned value back onto the abstract stack. In our shadow stack a book-keeping operation is required to note that the top-of-stack is in a register.

With the *VAL* of our reference model, it is simplest to insist that only *virtual* registers are allowed on the shadow stack. Thus we generate an instruction which immediately copies the value of the real register to a virtual register.

```
PROCEDURE PshRet(); (* move return reg to a vReg *)
  VAR newV : VRegister;
BEGIN
  NewVReg(newV);
  EmitRtR(move,retReg,newV);
  INC(tos);
  InitRegisterElem(tos,newV);
END PshRet;
```

The benefit of this choice is that it is not necessary to be concerned about nested function calls. During the register binding process, if there is no conflicting use, the virtual register will be assigned to the return register and the move instruction will disappear.

Conversely, the return statement of function procedures becomes an expression evaluation followed by a *popRet* instruction in the *IR*. This is translated as follows —

```
PROCEDURE PopRet(); (* move top-of-stack to return reg *)
BEGIN
  MakeRegister(tos);
  EmitRtR(move,shadow[tos].vReg,retReg);
  DEC(tos);
END PopRet;
```

5.2 Condition codes, more address modes

In this section we consider a number of topics which extend the simple code selection procedures which we have considered up until now. We shall consider three extensions — machines with *condition codes* rather than Boolean set instructions, load-store machines with more address modes, and the passing of parameters in registers.

These extensions come closer to the reality of load-store architectures such as *SPARC* and *MIPS*. We shall defer consideration of *CISC* instruction sets to section 5.3.1.

5.2.1 Condition codes

Conventional machine architectures have **condition code** or **flag** registers which store the outcome of compare and test instructions. On many machines these bits may be set as a side-effect of other instructions, as well as the explicit tests and compares. The concept is plausible, since it allows sometimes for a test to be performed at no additional cost. Unfortunately, the use of condition codes in this way is difficult. In machines such as *SPARC* there are two versions of many instructions — one which modifies the condition codes, and one which does not. In this way it is easier to control the use of the condition codes when the instruction which *sets* the flags becomes separated from the instruction which *uses* the flags. Even so, the use of such side-effects makes several optimizations much more difficult. Here we shall always set the condition codes explicitly by use of a test instruction. Later, after all other optimizations have been done, we may try to remove the tests by setting the flags as a side-effect of some other instruction.

Use of condition codes falls into two cases. The condition code may immediately be used for a conditional branch, or the Boolean result which the code represents may be used as an ordinal Boolean value. In one case we may use the condition codes directly in a conditional branch instruction, in the other, we need to convert the codes to an ordinal value.

It is relatively expensive to convert condition codes into an ordinal value if the machine does not support such a conversion. Here is some typical code for the *SPARC* processor —

```
    move %g0,%i0        ;  move zero to i0
    bnCC label          ;  branch on CC false
    inc %i0             ;  increment ordinal
label:                  ;  i0 now has Boolean
```

Let us see how a shadow stack can convert the primitives of our *IR* into instructions which set and use condition codes. We do this by introducing an extremely powerful concept — that of using the shadow stack to hold symbolic *values* as well as the symbolic addresses which we have used so far.

When a relational test instruction is encountered in the *IR* no instruction is produced. Instead the shadow stack is updated to show that the value on the top-of-stack is the value which would be generated *if the specified test was carried out.* We defer actually generating any instruction until it is known whether the value is to be used for branching or as an ordinal.

For load-store machines we will still need to load the operands of the test, since we assume that the values will need to be loaded for the real test. We therefore introduce a new symbolic Boolean element of the *SymbolicValueDescriptor* union, corresponding to this symbolic value.

```
TYPE TagType = (symBool,literal,register,address);
     SymbolicValueDescriptor =
       RECORD
         CASE tag : TagType OF
         | symBool  : test  : DCodeType;
                      lhReg : VRegister;
                      rhReg : VRegister;
         | literal  : litW  : WORD;
         ...
```

The codes for all of the comparison procedures of the shadow stack automaton now become extremely simple, and can all share the same procedure body —

```
PROCEDURE Relop(dcode : DCodeType);
BEGIN
  MakeRegister(tos - 1);
  MakeRegister(tos);
  InitBoolElement(tos - 1,dcode, (* test  *)
         shadow[tos - 1].vReg,   (* left  *)
         shadow[tos].vReg);      (* right *)
  DEC(tos);
END Relop;
```

The newly produced symbolic Boolean value will be used in one of two ways. It may be on the top-of-stack when a `brTrue` or `brFalse` is found. In this case the shadow stack automaton will emit a compare instruction followed by the appropriate conditional branch.

For the other case of value use, the comparison result is used as an ordinal value. This value might be used in an assignment, or as an operand in another comparison. In this case the code of the shadow stack automaton will reach a state in which it is asked to load the stack element, and will find the symbolic value. The procedure to load abstract stack elements into registers must take into account this new case, by emitting code to actually perform the test and load the value into a register. The *MakeRegister* procedure gets some extra lines, as compared to figure 5.3 —

```
PROCEDURE MakeRegister(lev : StackIndex);
  (* post : shadow[lev] is loaded *)
  VAR new : VRegister;
BEGIN
  IF shadow[lev].tag <> register THEN
    WITH shadow[lev] DO
      NewVReg(new);         (* get a new register *)
      IF tag = symBool THEN (* compare and load   *)
        EmitRRt_(cmp,lhReg,rhReg);
        EmitRt_(setMap[test],new);
      ELSIF tag = address THEN (* emit load adr *)
      ...
```

Note that we require a mapping table to convert the various values of the test field of the record into the appropriate *set* function. In this table, for example, `setMap[intRelLE] = sle` or whatever the instruction is for producing a

Boolean value from a signed comparison. In the case of machines such as the earlier models of the *iapx86* family it would require several instructions to produce the value, as described earlier.

The advantage of deferring the emission of the comparison operation should now be clear. In the case of complex Boolean expressions the result of a comparison can temporarily be pushed down below the top of the shadow stack. It does no harm to push symbolic values down the stack, and once a value is loaded into a virtual register it is safe also. However, condition code values cannot be pushed down the stack, since their values are volatile, and will be overwritten by later instructions. Of course, the ability to push a symbolic value below the stack top without changing its meaning depends on the single assignment property of these registers. If there is only one definition of each of the virtual registers which are involved in the test, the values will be the same no matter how much later the test is performed.

Figure 5.7 shows step by step the selection of code for a Boolean evaluation in which symbolic values are pushed below the top of the stack. The *VAL* in this figure uses the *Intel-386* compare "`cmp`", and set "`sle`", "`sne`" instructions. This method of using a symbolic value to defer code emission is quite simple, and has advantages even in the case of machines such as *MIPS* which do not have condition codes. In such a case, the use of the symbolic value allows the shadow stack automaton to wait until the use of the value is known. In some cases it is then possible to avoid the emission of the set instruction entirely, by using a compare and branch instruction directly.

The *IBM Power* architecture goes even further, in that multiple sets of condition codes exist. In this case it is sensible to treat the condition code registers as just another resource modelled by the set of virtual registers. The task of register allocation can then be combined with the allocation of absolute condition code registers.

5.2.2 More address modes

In order to show the way to still more complex cases, we shall consider the selection of code for an architecture which has just one additional address mode. Using *SUN Microsystem's SPARC* as a model we shall assume that the second address mode is a two-register, indexed form. A load or store instruction may use a memory address which is one of the two forms

$$\mathit{symbol}{+}\mathit{offset}\ (r_{index})$$

and

$$(r_{src1}{+}r_{src2})$$

In this second address mode, the effective address is the sum of the contents of the two nominated registers.

Of course the first address mode still has all the usual special cases which arise when one of the symbols has one of the distinguished values *idNil*, 0, or *noReg*.

It is suggested that the best way of coping with this extra mode is to enrich the structure of the symbolic addresses on the shadow stack. Suppose that we modify the *SymbolicValueDescriptor* type so that it may represent either address

Code for – `IF (var1 < var2) <> (var3 < var4) THEN ...`

```
    pshAdr var1
    derefW              ; emit "lw var1,v1"
    pshAdr var2
    derefW              ; emit "lw var2,v2"
    intRelLS
```

The shadow stack is now —

bool: (intLE, v_1, v_2)	← top-of-stack

```
    pshAdr var3
    derefW              ; emit "lw var3,v3"
    pshAdr var4
    derefW              ; emit "lw var4,v4"
    intRelLS
```

The shadow stack is now —

bool: (intLE, v_3, v_4)	← top-of-stack
bool: (intLE, v_1, v_2)	

Note that we have two deferred Boolean evaluations on the shadow stack, and now wish to compare them. Recall that relational operations on the shadow stack must always load both operands into registers, so —

```
    relNE               ; emit "cmp v1,v2"
                        ; emit "sle v5"
                        ; emit "cmp v3,v4"
                        ; emit "sle v6"
```

The shadow stack is now —

bool: (relNE, v_5, v_6)	← top-of-stack

```
    brFalse             ; emit "cmp v5,v6"
                        ; emit "bne elseLabel"
```

Figure 5.7: Code selection for complex Boolean test

type.

```
TYPE TagType = (symBool,literal,register,address);
     SymbolicValueDescriptor = RECORD
        CASE tag : TagType OF
        | literal  : litW  : WORD;
        | register : vReg  : VRegister;
        | symBool  : test  : DCodeType;
                     lhReg : VRegister;
                     rhReg : VRegister;
        | address  : idnt  : HashBucketType;
                     ofst  : INTEGER;
                     reg1  : VRegister;
                     reg2  : VRegister;
        END; (* case *)
      END;
```

The values of the symbolic address type will be initialized in the same way as previously. The major changes caused by this greater generality are shown in the procedure which implements the **addAdr** instruction. When a value in a register is to be added to an address, it may go into either of the index register positions, without generating any code.

```
PROCEDURE AddAdr();
  VAR new, old : VRegister;
BEGIN
  MakeAddress(tos - 1);
  (* now try to fold add into address modes *)
  WITH shadow[tos - 1] DO
    CASE shadow[tos].tag OF
    | literal :  (* add to offset *)
        off := off + CAST(INTEGER,shadow[tos].litW);
    | register : (* fold into index *)
        old := shadow[tos].vReg;
        IF reg1 = noReg THEN reg1 := old;
        ELSIF reg2 = noReg THEN reg2 := old;
        ELSE (* emit an add instruction *)
          NewVReg(new);
          EmitRRtR(add,reg2,old,new);
          reg2 := new;
        END;
    | address : Assert(FALSE,"bad tos element");
    END; (* case *)
  END; (* with *)
  DEC(tos);
END AddAdr;
```

Figure 5.8: The *SPARC* version of *AddAdr*

A typical implementation is shown in figure 5.8. Compared to the equivalent procedure in the previous section, figure 5.4 it will be seen that even fewer cases lead to the emission of an instruction.

As an example of the use of this address mode, consider the selection of code for indexing into a array $array[i, j]$, where the array $array$ is a variable parameter. In section 5.1.3 we considered an assignment statement which included this expression evaluation, for the simple load-store architecture. The *DCode* sequence is —

```
pshFP 8        ; 1   push address of param
derefW         ; 2   get value of the param
pshFP -4       ; 3   address of local var i
derefW         ; 4   get value of i
pshLit 64      ; 5   cardinality * element size
mul            ; 6   multiply by array size
addAdr         ; 7   add offset to address
pshFP -8       ; 8   address of local var j
derefW         ; 9   get value of j
pshLit 4       ; 10  element size
mul            ; 11  multiply by elem size
addAdr         ; 12  add offset to address
derefW         ; 13  get array element
```

After the processing of instruction 6 by the shadow stack automaton, the following instructions will have been emitted —

```
lw 8(fp),v1        ; (2)
lw -4(fp),v2       ; (4)
muli v2,64,v3      ; (6) or: shl v2,6,v3
```

and the shadow stack will be —

reg: v_3	← top-of-stack
reg: v_1	

In this case the `addAdr` code does not cause an instruction to be emitted. Instead the stack is updated so as to have a symbolic address, in the new mode, on top.

adr: $(idNil, 0, v_1, v_3)$	← top-of-stack

The final instructions emitted are, with their indices into the *DCode* —

```
lw -8(fp),v4       ; (9)
muli v4,4,v5       ; (11) or: shl v4,2,v5
add v3,v5,v6       ; (12)
lw (v1+v6),v7      ; (13)
```

At *DCode* index 12 we need a third index register, so it becomes necessary to emit an instruction which adds two index registers. Nevertheless, a whole instruction has been saved by the use of this new address mode.

We have created a symbolic address representation which is the union of the two actual modes available on the machine. It may be noted that this allows the shadow stack automaton to accumulate symbolic addresses which cannot be emitted in a single instruction. For example we might end up with an address expression which has both two index registers *and* an offset.

In the case that an impossible address mode is created, we have to surrender the benefit of the saved instruction by emitting two instructions. The *Deref* and *Assign* procedures must check for such impossible address modes. In the event that it exists, a preliminary instruction must add the two index registers. The result register is then used as the index register for a single-index-with-offset instruction.

5.2.3 Passing parameters in registers

Almost all *RISC* machines use procedure call conventions which pass parameters in registers. There are advantages at runtime from using such a convention even for conventional machines, but the price for doing so may be incompatability with the conventions of other language processors for the same machine.

It is normal for the first few word-sized parameters to be passed in registers, and the remainder to be passed in a parameter assembly area, at the top of the current stack frame. For example, the convention for *MIPS* processor systems is to pass the first four parameters in registers r_4–r_7. In *SPARC* the first six parameters go in registers $o0 - o5$. In both cases space is reserved on the stack for the parameters, so that the *called procedure* has a pre-allocated place to save the registers, should that be necessary.

The conventions for passing structured value-mode parameters are more variable, and were explored in the chapter on runtime organization.

This convention sets several problems for our code-selection process. Firstly, the calling of functions during parameter evaluation is immediately made more complex. Recall that when passing parameters on the stack, the stack semantics automatically caters for such nested calls. Suppose we wish to make the procedure call —

```
WriteCard(GCD(a,b),0);
```

When parameters are passed on the stack, the sequence of events is as follows. First, the zero value is pushed on the stack. Finding that the next parameter is a function call, we push the values of b then a, then call the function *GCD*. After the function returns, we cut the two parameters off the stack, and push the value returned by the function. The parameters of *WriteCard* are now ready!

Now contrast the situation with the *MIPS* architecture. For the call of *WriteCard* the parameters will be placed in registers r_4 and r_5. These same registers are used by parameters of the *GCD* function. The conclusion is simple, we cannot transfer the parameters into their final positions in an arbitrary order.

Note that this problem is not actually caused by the fact that the parameters are passed in registers. It is caused by the fact that parameters are passed in data structures which are shared. Exactly the same problem arises if we pass parameters in fixed locations at the top of the caller's stack frame. In practice *RISC* machines invariably pass the non-register parameters in a fixed *parameter assembly area* at the top of the caller's stack frame. Thus the reasoning of this section applies to all parameters, and not just those which are finally placed in registers.

The correct way to pass parameters into fixed locations is to serialize the procedure calls. This necessitates first evaluating all those parameters which require function calls. These expressions are evaluated into virtual registers. When all such values are complete, the values are copied into their destination positions. The remainder of the parameters are then evaluated and placed in position, in any order. Note that there may be non-obvious cases, such as variable designators which require function evaluation in array index calculations.

For our example procedure call, the sequence of events might be as follows. *WriteCard* has two parameters, one of which involves a function call, so we must evaluate this parameter first. The function call of *GCD* involves only simple parameters so that we fetch the value of a and place it in register r_4, and the value of b in r_5. The function *GCD* is now called. The function returns its result in register r_2 in the *MIPS* conventions, so we copy the return register to r_4. Finally, we copy the zero register r_z into r_5, in order to place a zero in the second position. We are now ready!

In practice it is sensible to evaluate all parameters in an order which evaluates those that use the most registers first. This places the least **register pressure** on the register binder. The reason is simple. As each parameter is transferred into its parameter register, the number of registers available for expression evaluation is decreased by one.

We shall assume that the front-end tree walker has determined the optimal parameter evaluation order. It is also necessary for the front-end to indicate the position of the parameter, so that it is known whether to put it into a register, or into the parameter assembly area in memory. The semantics of the `mkPar` instruction must thus be extended to carry this information.

5.3 Machines with complex address modes

The main attribute which distinguishes the programming model of *CISC* from *RISC* machines is the existence of instructions which take their operands from memory. Since the structure of the *SymbolicValueDescriptor* type in some sense mirrors the capabilities of the machine address modes, we must expect the symbolic address values of the union to be more complex.

There are two sources for this additional complexity. Firstly, the symbolic address values must have sufficient structure to represent all of the address modes which the code selector is to use. There is a further difficulty however. Most machines in this traditional model have restrictions on the purposes for which particular machine registers may be used. The real registers are semantically quite different from the nicely uniform and interchangable virtual registers that we have considered so far. With our reference architecture we may separate these two sources of complexity. The shadow stack automaton must model the complexity of the address modes, while the register binder must determine the uses of each virtual register so that an appropriate actual register may be allocated.

5.3.1 A CISC machine model

For our examples, we shall use a machine model based on the instruction set and address modes of the *Intel-386* and *486*[4]. This model illustrates most of

[4] The following deals with the flat address model of the *Intel* processors. This address model is used in OS/2, Windows NT and the various *UNIX* implementations. When the *Intel-x86* is used in "16-bit segmented" address mode, as it is by MS-DOS, the following is rather more complex.

the difficulties that such machines may place in the way of interpretive code selection. Earlier models of the *Intel-x86* family are more restrictive in their use of registers than the *386* and later models.

Compared to the load-store architectures, the most striking difference is that most instructions are able to take one of their operands from memory. We will try therefore to use operands directly from memory whenever possible, in order to avoid having to load operands with separate instructions[5].

The *VAL* which we shall use is similar to the actual *iap386* machine except for the following aspects. We shall separate the uses of the "`mov`" instruction for loads, stores, and register-to-register transfers by using different virtual instructions for the three cases. We shall call these instructions "`lw, sw, move`" respectively. During the register binding process we will convert all of these into "`mov,movzbl,movzwl,movsbl`" or "`movswl`" instructions, as required.

A rather more striking change is made to all the operate instructions. In the real machine, these all overwrite one of their operands as the destination register. In the *VAL* these all get an extra operand, so that the source and destination operands are always separate. For example, the "`add`" instruction in the *VAL* has three operands rather than two. In this way we can ensure that every virtual register has a single definition point. The register binder will attempt to allocate registers so that the source and destination are the same, as is the case for the real "`add`" instruction. If it is not possible to allocate the same actual register to both virtual registers then the instruction must be preceeded by a register to register copy. In this way we avoid the destruction of values in registers until we know that the values are no longer needed. In any case, this modification makes the *VAL* much more similar to the load-store model described so far.

The machine model has a large number of memory address modes, but the most general one considered here is *based, indexed, scaled*. In this address mode an address consists of the sum of a base register, an index register which is optionally scaled by either two or four, and an offset which may involve a symbolic name. Either or both of the registers may be omitted, the offset expression and the scale factor may also be omitted, obtaining all other address modes as special cases of this mode.

In our symbolic representation, an address expression consists of six fields — base register, index register, index scale, symbolic name, offset, and type. The type denotes whether the address is known to be that of a signed or unsigned byte, and so on. Address modes which may be thought of as special cases of this general mode will be permitted.

The most important aspect of the design of the shadow stack for this *CISC* machine model is that we introduce a *symbolic value* member of the union, with a structure matching that of the symbolic addresses.

[5]Such a strategy uses fewer instructions, and uses fewer registers, but exposes fewer optimization opportunities.

```
TYPE TagType  = (literal,register,ccResult,symAdr,symVal);
     ScaleTag = (noScale,scalX2,scalX4);
     SymbolicValueDescriptor = RECORD
          CASE tag : TagType OF
          | register : vReg  : VRegister;
          | ccResult : test  : Instruction; (* dcode *)
          | symAdr, symVal :
                       type  : Object;
                       bReg  : VRegister;
                       iReg  : VRegister;
                       scale : ScaleTag;
                       ofst  : INTEGER;
                       idnt  : HashBucketType;
          | literal :  litW  : WORD;
          END; (* case *)
        END; (* record *)
```

The new value type allows us to perform **symbolic dereferences** on addresses. When our *IR* calls for an address value to be dereferenced we do not emit any instruction. Instead, we change the tag value from *symAdr* to the *symVal* value, and simply mark the stack element with the datum type. This represents the fact that the top of the shadow stack is the value which would be obtained *if the given address was to be dereferenced.* We wish to delay the actual emission of an instruction which performs the dereference because we may be able to incorporate the load into an address mode of an operate instruction with one operand in memory.

Of course the correctness of this technique must be critically examined. In effect, the *IR* has called for a value to be fetched from memory and we wish to delay this step in the hope of saving an instruction. We must be completely sure that when we finally fetch the operand, it is still the same value. Consider evaluation of the expression (*var*+*fx*()). Our code selector will delay dereference of the value of *var* hoping to fold it into the add instruction. Indeed we will be successful in this quest, since the function result will be in a register and we may use the commutative law to generate the virtual instructions —

```
call fx                    ;    return value is in eax
add eax,var,v1
```

This sequence is preferred to the alternative —

```
lw var,v1
call fx                    ;    return value is in eax
add v1,eax,v2
```

The result will be the same, provided that the function *fx* does not have side-effects which modify the value of *var*.

Programming languages such as *Modula-2*, *C* and *Ada* allow the commutative law to be used to rearrange such expressions, and permit similar freedom in the order of evaluation of parameters, array indices and the like. Certainly in all of these languages a program which gave different results depending on which of the above instruction sequences was used would be considered to be incorrect. However, for languages which demand strict order of evaluation of expression components, it may be necessary to be more careful. What is required, in such a strict case, is the guarantee that whenever a function is about to be called the shadow stack will be checked. All symbolic values must then be *realized* by being loaded into registers.

From the descriptor type declaration it will be seen that there is another new tag-type value, *ccResult*. For the machine model of this section an alternative handling of the condition codes is suggested. In this case the comparison operation is carried out immediately, with the stack element simply noting the intended semantics. This lowers the register pressure, in the event that either or both of the register operands have their final use at the comparison. However, it is necessary for all the *push* procedures to check if the top-of-stack is one of these values before any instruction is emitted which could overwrite the condition codes. If that is the case, the value must be loaded into a virtual register by a set operation.

5.3.2 Generating code for the model

We shall consider just a few key steps in the operation of the shadow stack automaton, to illustrate the general method. We shall consider the procedures *Deref, Assign, PshAdr, Add* and *Relop.*

The dereference instructions

In this case, the code is straightforward. We make sure that the top of stack is a symbolic address, and modify the tag to make it a symbolic value.

```
PROCEDURE Deref(elem : ValueType); (* word, halfword etc *)
BEGIN
  MakeAddress(tos);     (* make tos into adr if needed *)
  WITH shadow[tos] DO
    tag := symVal;
    type := typeMap[elem];
  END; (* leave other fields unchanged *)
END Deref;
```

The magic is all in the *MakeAddress* procedure. This procedure must take the stack value, and convert it into an address if it is not already an address.

The *Emit* procedures are modified so that for the case of instructions with memory operands all six fields of the symbolic address are passed as parameters. Figure 5.9 is the *CISC* version of the make address procedure. Note that if the value is already a symbolic value, then that value must be loaded into a register, and that becomes the base register of a new symbolic address.

The assign instructions

The assignment operation nicely illustrates the difficulty of the architecture. The value to be assigned could initially be of any tag type. We might insist

```
PROCEDURE MakeAddress(lev : StackIndex);
  (* post : shadow[lev] is a sym adr *)
  VAR idx : VRegister;
BEGIN
  IF shadow[lev].tag <> symAdr THEN (* load it *)
    WITH shadow[lev] DO
      IF tag = symVal THEN
        NewVReg(idx); (* load into new register *)
        EmitMtR(lw,bReg,iReg,scale,ofst,idnt,idx);
      ELSE
        Assert(tag = register,"bad address conversion");
        idx := vReg;
      END;
      InitAddressElem(lev,idx,noReg,scale1,idNil,0);
      (* make register the base reg of new symAdr *)
    END; (* with *)
  END; (* if *)
END MakeAddress;
```

Figure 5.9: The *CISC* version of *MakeAddress*

that the value be loaded into a register, as we were obliged to do in the case of the load-store architecture, and then assigned. However it is wasteful to do so. The kinds of values which are able to be directly assigned are — values in registers, literal values, and symbolic addresses without address registers. We thus need a utility procedure *MakeAssignOp* which guarantees that the shadow stack operand is in one of these forms. We then emit instructions in one of the four forms —

```
sw    r_src,mem-dst

sw    literal,mem-dst

sw    address,mem-dst    ; no regs in address

lea   address,v_n        ; other symbolic addresses use the
sw    v_n,mem-dst        ; load effective address instruction
```

The *MakeAssignOp* procedure will load *ccResults* and *symVals* into registers. If the stack element is a symbolic address which is not in one of the required forms already, it will be loaded as follows —

```
CASE tag OF
| literal, register : (* do nothing *)
| symAdr :
    IF (bReg <> noReg) OR (iReg <> noReg) THEN
      MakeRegister(lev);
   (* ELSE do nothing *)
    END;
ELSE MakeRegister(lev);
END; (* case *)
```

As with *Deref*, the magic is all in the *MakeRegister* procedure, which now has to cope with a larger number of possible tags, as shown in figure 5.10

```
PROCEDURE MakeRegister(lev : StackIndex);
  (* post : shadow[lev] is a register *)
  VAR new : VRegister;
BEGIN
  IF shadow[lev].tag <> register THEN
    WITH shadow[lev] DO
      CASE tag OF
      | symVal : (* just load it *)
          EmitMtR(LoadMap[type],bReg,iReg,scale,ofst,idnt,new);
      | symAdr : (* load effective address *)
          EmitMtR(leal,bReg,iReg,scale,ofst,idnt,new);
      | literal : (* load literal *)
          EmitItR(li,litW,new);
      | ccResult : (* load condition *)
          Emit_tR(CondMap[test],new);
      END;
      InitRegisterElem(lev,new);
    END; (* with *)
  END; (* if *)
END MakeRegister;
```

Figure 5.10: The *CISC* version of *MakeRegister*

In this procedure, the map *CondMap*

$$CondMap : Instruction \rightarrow OpCode$$

maps (a subrange of) *DCode* instructions to the *OpCode* type.

The push operations

The procedures to push values on the shadow stack are relatively straightforward. However, for correctness it is necessary to ensure that we do not push a value with a *ccResult* tag below the top-of-stack, or the condition codes may be overwritten. All the *Psh–* procedures must check this.

As an example, the *PshAdr* procedure has two tasks. It must check the current top-of-stack, and load it if it is a *ccResult*, then a new address element must be constructed.

```
PROCEDURE PshAdr(nam : HashBucketType, off : INTEGER);
BEGIN
  IF shadow[tos].tag = ccResult THEN MakeRegister(tos) END;
  INC(tos);
  InitAddressElement(tos,noReg,noReg,noScale,nam,off);
END PshAdr;
```

The add instruction

The add instruction is a model for all of the operations in this machine model. The left hand operand of the instruction must be loaded into a register. The right hand operand may be a register, a literal, or a symbolic value. This set of permitted tag values is not quite the same as for an assignment, so that we need a different *Make...* procedure. Figure 5.11 has the modified *Add* procedure. Compared to figure 5.5, you should note the significantly more complex conditions under which it is productive to swap the top-of-stack elements.

As before, this procedure is a model for all of the commutative operators and, without the element swap, for the non-commutative ones as well.

The relational operators

As described earlier, for this machine model we shall emit a compare instruction immediately, and save a marker on the shadow stack so that the intended semantics of the test are remembered. For the compare instruction the permitted address modes do not exactly correspond to any of the *Make...* procedures which we have described so far. Since the combination of modes is unique, we do the conversions inline as shown in figure 5.12. It may be noted that it would also be possible to swap the operands and use a modified test in order to save a load under some circumstances, but this detail has not been shown here.

There are a number of other possibilities which have been glossed over here, such as the possibility of using the *Intel-x86* chip's short compare instructions "`cmpb`" or "`cmpw`" if the memory operand is shorter than 32-bits, and the literal is also representable in a short range.

Using the shadow stack: complex address expressions

Consider the following assignment statement, which is the same one used as an example at the end of the section 5.1.3. The variable *array* is assumed to be a *VAR* parameter, which points to the actual array.

```
temp := array[i,j];
```

The resulting *DCode* sequence is —

```
PROCEDURE Add();
  VAR new : VRegister;
BEGIN
  NewVReg(new);
  IF (shadow[tos - 1].tag = literal) OR
     ((shadow[tos - 1].tag <> register) AND
      (shadow[tos].tag = register)) THEN
    SwapTopElements();
  END;
  MakeRegister(tos - 1);
  IF shadow[tos].tag = literal THEN
    EmitRItR(add,shadow[tos - 1].vReg,shadow[tos].litW,new);
  ELSIF shadow[tos].tag = register THEN
    EmitRRtR(add,shadow[tos - 1].vReg,shadow[tos].vReg,new);
  ELSIF (shadow[tos].tag = symVal) AND
        (shadow[tos].type = word) THEN
    WITH shadow[tos] DO
      EmitRMtR(add,shadow[tos - 1].vReg,
                      bReg,iReg,scale,idnt,ofst,new);
    END; (* with *)
  ELSE
    MakeRegister(tos);
    EmitRRtR(add,shadow[tos - 1].vReg,shadow[tos].vReg,new);
  END;
  DEC(tos);
  shadow[tos].vReg := new;
END Add;
```

Figure 5.11: The *CISC* version of *Add*

`pshFP 8`	; 1	*push address of param*
`derefW`	; 2	*get value of the param*
`pshFP -4`	; 3	*address of local var i*
`derefW`	; 4	*get value of i*
`pshLit 64`	; 5	*cardinality * element size*
`mul`	; 6	*multiply by array size*
`addAdr`	; 7	*add offset to address*
`pshFP -8`	; 8	*address of local var j*
`derefW`	; 9	*get value of j*
`pshLit 4`	; 10	*element size*
`mul`	; 11	*multiply by elem size*
`addAdr`	; 12	*add offset to address*
`derefW`	; 13	*get array element*
`pshFP -12`	; 14	*address of local var temp*
`assignW`	; 15	*do the assign*

For this example the instructions have been numbered for ease of reference. The output instructions are as follows, and are emitted at *DCode* indices as indicated —

```
PROCEDURE Relop(dcode : DCodeType);
  VAR leftT  : TagType; (* tag of left element  *)
      rightT : TagType; (* tag of right element *)
BEGIN
  leftT  := stack[tos - 1].tag;
  rightT := stack[tos].tag;
  IF (leftT <> register) AND
     ((leftT <> symVal) OR (stack[tos - 1].type <> word)) THEN
    MakeRegister(tos - 1);
    leftT := register;
  END;
  IF (rightT <> literal) AND
     (rightT <> register) THEN
    MakeRegister(tos);
    rightT := register;
  END;
 (* assert: rhs is reg or lit, lhs is reg or word-sized symVal *)
  WITH stack[tos - 1] DO
    IF leftT = symVal THEN
      IF rightT = literal THEN
        EmitMIt_(cmpl,bReg,iReg,scale,idnt,ofst,shadow[tos].litW);
      ELSE
        EmitMRt_(cmpl,bReg,iReg,scale,idnt,ofst,shadow[tos].vReg);
      END;
    ELSIF rightT = literal THEN
      EmitRIt_(cmpl,vReg,shadow[tos].litW);
    ELSE
      EmitRR(cmp,vReg,shadow[tos].vReg);
    END;
  END; (* with *)
  DEC(tos);
  InitCcResultElement(tos - 1,dcode);
END Relop;
```

Figure 5.12: The *CISC* version of *Relop*

```
; ebp is the frame pointer register
lw -4(ebp),v1            ; (6)
mul v1,64,v2             ; (6) or: shl v1,6,v2
lw 8(ebp),v3             ; (7)
lw -8(ebp),v4            ; (11)
mul v4,4,v5              ; (11) or: shl v4,2,v5
; addAdr has overflowed the symAdr format, must emit an add ...
add v2,v3,v6             ; (12)
lw (v6,v5),v7            ; (15)
sw v7,-12(ebp)           ; (15)
```

In this particular case, the symbolic dereferences delay the emission of the first instruction until six whole *DCode*s have been read.

It may be noted that we have still wasted one instruction in this code. The multiplication of v_4 by four could have been folded into the address mode of the last mode, by using the *based, indexed, scaled* address mode. But how are we to know *not* to perform the multiplication by four at *DCode* index 11? The answer should by now be familiar. We must defer the emission of such instructions until we know the context in which the value is to be used. We do this by introducing yet another extension of the shadow stack element type. Suppose that we call these new tags *regX2* and *regX4*. Whenever such a value is used, if it is used for any purpose other than as the index field in a symbolic address, the deferred multiplication must be performed after all. However, in the example above, at *DCode* index 11 we emit the `lw` only, and change the top-of-stack tag to *regX4*. The address addition still folds the overflowed format. The final code produced is —

```
lw -4(ebp)v1            ; (6)
mul v1,64,v2            ; (6) or: shl v1,6,v2
lw 8(ebp),v3            ; (7)
lw -8(ebp),v4           ; (11)
; addAdr has overflowed the symAdr format, must emit an add ...
add v2,v3,v6            ; (12)
lw (v6,v4,4),v7         ; (15)
sw v7,-12(ebp)          ; (15)
```

5.4 Exercises

5.1 `<<showing cases where jumping code is worse>>`

5.2 Many machines have "`set`*CC*" instructions to convert the results of comparisons into Boolean values. In general, those machines which do not have such instructions must jump on the comparison result, and use code similar to figure 4.7 to create a Boolean value. It is best to avoid this construction whenever possible.

Machines with "add with carry" or "subtract with borrow" instructions can generate a Boolean without jumping in some special cases. On *SPARC* these instructions are called "`addx`" and "`subx`". The carry flag is loaded into register r_d by adding the carry to two zeros –

`addx g0,0,`r_d ; *result is 0 or 1*

The negation of the carry flag is loaded into r_d by —

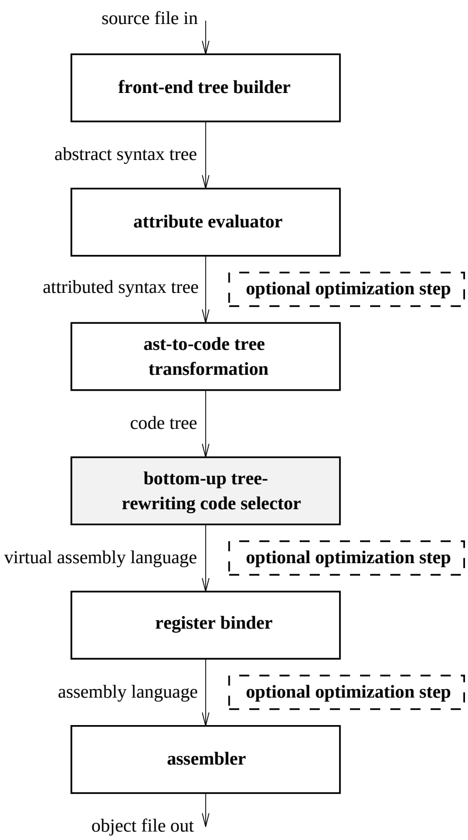
`subx g0,-1,`r_d ; *result is 1 or 0*

The unsigned less-than comparison generates a carry flag as its result, and all the asymmetric unsigned comparisons can be turned into this form by some combinaton of swapping of operands, incrementing one operand by one, or loading the negation of the flag. Work out how to transform $(r_l \leq r_r)$, $(r_l > r_r)$, and $(r_l \geq r_r)$ to use this technique, where r_l and r_r are unsigned operands in registers.

5.3 Symmetric comparisons (equal and unequal) are also able to be performed using the technique of the previous question. In this case we need to rely on the property that (a `xor` b) is zero if and only if $a = b$. See if you can work out the remaining details, so as to compute $(r_l = r_r)$, and $(r_l \neq r_r)$ into a register without having to use branching code.

Chapter 6

Bottom-up Tree Rewriting



6.1 Introduction to the theory

The theory of bottom-up tree rewriting systems is important both for elegantly solving the *operator identification problem* referred to in section 2.5.1, and also for providing the theoretical basis for the most powerful method of instruction selection.

The problem is modelled by a situation in which every node of an abstract syntax tree has to be labelled with a *typeform* attribute. These typeforms must be consistent according to a finite set of rules which constrain the relationship between the typeform of a particular parent node and the typeforms of its descendants. Furthermore, we do not require that a consistent typeform-labelling be unique. Instead we allocate a non-negative *cost* to each rule which is used, and seek the consistent labelling with the lowest total cost.

It is important not to confuse the concept of *typeform* with the concept of *datatype* as determined by static semantic analysis. The set of values which datatypes take are values such as "pointer to record..." and so on. The *typeform* of a node specifies the location and representation of a value at runtime. The set of values which typeform takes includes values such as "register", "symbolic address" and "immediate constant". These correspond to the values of the *TagType* type-enumerations used for *SymbolicValueDescriptor* records throughout the previous chapter.

Since we wish to model tree pattern-matching and tree-rewriting transformations we borrow a formalism familiar from the world of phrase grammars and string transformations. We define a set of node *typeforms* which will be the non-terminal symbols of the grammar. The operator tags of the tree nodes will be the terminal symbols of the grammar. A finite set of productions each have a single non-terminal symbol on the left, which specifies the result typeform of the permitted transformation. The right-hand-side of each production specifies the tree pattern to which it applies. These tree patterns may specify the node-*tag* to which they apply (a terminal symbol of the grammar), and the typeforms of that node and its decendants (non-terminal symbols). In general such tree-grammars are ambiguous, with many self-consistent typeform attributions. We use the costs associated with each production to guide the choice of an attribution which has the minimal cost.

6.1.1 A simple example

As a simple example, suppose that we wish to find the minimal cost labelling of the assignment statement —

$$V_{int} := V_{flt} + V_{int} + V_{dbl}$$

for the *Intel-x86* instruction set. The variables have the types indicated by their names, so that *Vint* is an integer operand, *Vflt* is a single precision "float", while *Vdbl* is a double precision value.

We assume that the language in which the statement is written provides automatic coercions from integer to floating point types, and that an implicit downward coercion across an assignment statement is also permitted. The left tree in figure 6.1 is the *AST* of the statement. On the right of the figure is the code-tree which results from static semantic analysis of the *AST*. In the *Intel-x86* coprocessor architecture all floating point computations are done in

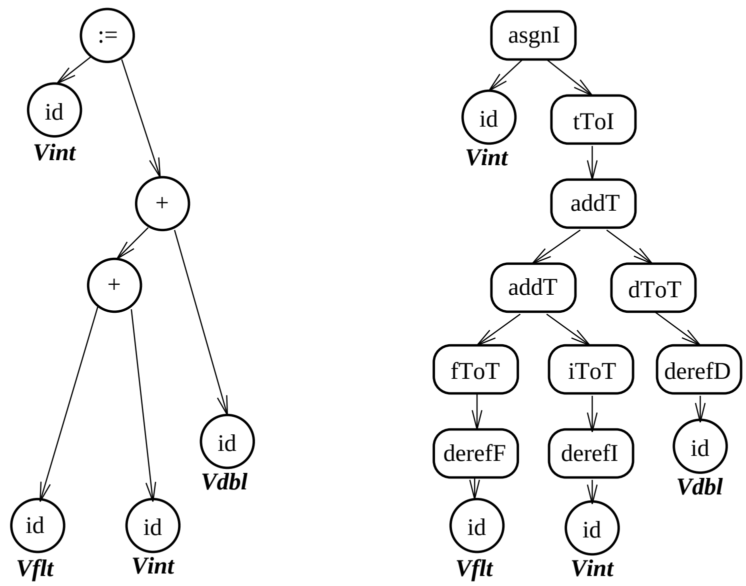


Figure 6.1: *AST* and code-tree of example expression

an extended precision *temporary format*. This is denoted T in the names of the coercions in the diagram. In essence the variable addresses must be dereferenced and the resulting value converted into the internal temporary format. When the floating point result is computed, the value must be converted back down to an integer value, using the *tToI* conversion.

Each available instruction in the machine instruction set matches a pattern of adjacent nodes in a tree. Two example patterns are shown in figure 6.2 along with the fully parenthesized formulas which define them. For both the root of

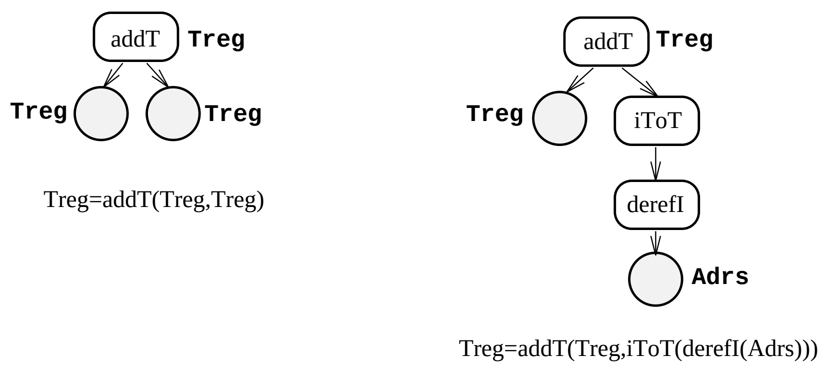


Figure 6.2: tree patterns, and corresponding prefix expressions

each tree, and the leaves of the trees we annotate the node with a *typeform*, which indicates the form in which the expression rooted at that node must have. In the examples, the two forms are *Treg*, the internal floating point temporary format, and *Adrs*, a generalised memory address form. Of course, the leaves of the patterns, shown shaded in the figure, will actually be the roots of subtrees in the subject trees.

On the left, an instruction, denoted "`fadd`" in most assemblers for the *Intel-x86* architecture, adds together two values in the *Treg* form to produce a result in the *Treg* form. The production which specifies this pattern is —

```
Treg = addT(Treg,Treg)
```

The pattern on the right, which in most assemblers for *Intel-x86* has the mnemonic "`fiadd`", takes a *Treg* on the left, and a memory address on the right. The instruction dereferences the memory address, converts the resulting integer into the internal floating point form, and adds the two values to create a result in the *Treg* form. The defining production pattern is —

```
Treg = addT(Treg,iToT(derefI(Adrs)))
```

Notice carefully, that these patterns do not specify anything about the leaves of the patterns, except for the form in which the value must be. In other words, we do not care at all about the *structure* of the subtrees which are rooted at the leaves of the pattern, provided that the value may be placed in the required *typeform*.

The available coprocessor operations are shown in table 6.3, with the costs shown in the figure.[1] Notice that each of the add-from-memory tree patterns

	production	**cost**	**comment**
1	`Stmt = asgnI(Adrs,tToI(Treg))`	79	convert Treg to int and store
2	`Treg = iToT(derefI(Adrs))`	45	load int and convert to Treg
3	`Treg = fToT(derefF(Adrs))`	20	load float and convert to Treg
4	`Treg = dToT(derefD(Adrs))`	25	load double and convert to Treg
5	`Treg = addT(Treg,Treg)`	23	add two Tregs together
6	`Treg = addT(Treg,iToT(derefI(Adrs)))`	57	load int, convert and add
7	`Treg = addT(iToT(derefI(Adrs)),Treg)`	57	load int, convert and add
8	`Treg = addT(Treg,fToT(derefF(Adrs)))`	24	load float, convert and add
9	`Treg = addT(fToT(derefF(Adrs)),Treg)`	24	load float, convert and add
10	`Treg = addT(Treg,dToT(derefD(Adrs)))`	29	load double, convert and add
11	`Treg = addT(dToT(derefD(Adrs)),Treg)`	29	load double, convert and add
12	`Adrs = ident`	0	a variable name is an address

Figure 6.3: productions for example

occurs in two forms, with the memory operand on the left, or on the right. This is because the add operation is commutative. The pattern on the left in figure 6.2 is production 5, while the pattern on the right is production 6. The final production states that a variable name is a legal address, and that the use of this production has no runtime cost.

[1]The costs, for this example, are the minimum number of cycles taken to execute the instructions on an *Intel-386* and *387* coprocessor.

How are we to encode the tree of figure 6.1? Take for example the addition which is the lower of the two in the tree, that of the float and integer operands. There are three possible ways in which this single operation may be encoded. We might load and convert each of the operands, using productions 3 and 2 respectively, and then add using production 5. Alternatively we might load just one of the operands and convert and add the other in a single instruction. These three rewritings of the subtree are shown in figure 6.4. Looking at the total costs

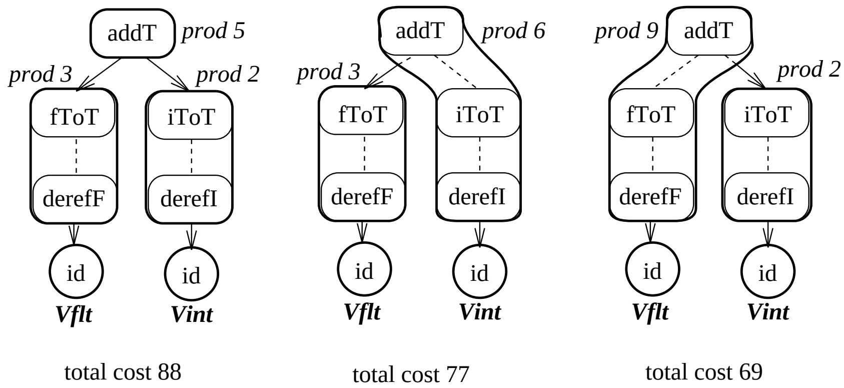


Figure 6.4: three encodings of the lower add operation in figure 6.1

of the tree-fragments shown in the figure shows that it is actually best to load and convert the integer operand, and then convert and add the float operand in a single instruction. Note in this figure that the least costly cover makes use of production 9, which is the commutative version of production 8, but with the memory operand on the left, rather than the right.

Applying similar reasoning to the rest of the tree shows that the whole statement can be encoded for a cost of 177 cycles,[2] by the following instruction sequence.

`fild Vint`	; *prod 2*	load and convert the integer – cost 45
`fadd Vflt`	; *prod 9*	convert and add the float – cost 24
`fadd Vdbl`	; *prod 10*	convert and add the double – cost 29
`fistp Vint`	; *prod 1*	convert result and store integer – cost 79

6.1.2 Dynamic programming

In the preceding discussion, it was implicitly assumed that finding the best rewriting of the first add instruction would necessarily lead to the best rewriting for the whole tree. In effect we assumed that an optimal choice for every subtree would lead to an optimal rewriting for the whole tree.

[2] Actually, there are a number of other instructions which have been ignored here. These save and restore the rounding mode of the coprocessor to provide the correct semantics for the downward coercion.

The assumption that every subtree of an optimal tree is separately optimal, allows us to use a classic divide and conquer optimization technique called *dynamic programming*. It works in this case because the cost of the rewriting is a simple sum of the costs of the subtrees plus the cost of the production, and because of a sort of context free property of the production costs. In effect, the cost of a particular production depends only on the typeforms of the operands, and not at all on the encoding of the subtrees which delivered operands of those typeforms. In the specific example in the last section, the cost of the second `fadd` does not depend on the way in which the left subtree computes its *Treg* result. Thus we may choose the fastest way of producing an operand of this typeform, secure in the knowledge that the particular choice will not affect the costs of decisions further up the tree.

In general, we do not have a single possible typeform for an operand, and we must work out the optimal rewriting of a given subtree for every possible typeform of the subtree's root node. The details of the way in which such trees are annotated is given in a later section, however the underlying idea is quite simple. We make one pass over the tree, bottom-up, computing *all* of the possible result typeforms, and their associated minimal costs. A second, top-down pass over the tree propagates a *required typeform* attribute down the tree, and determines which production is to be used.

6.1.3 A second example

A second motivating example explores how to optimally compute the following expression into a register —

$$d + (a + b + c \times 4) \times 4$$

Here, all variables are 32-bit "long" integers, and the encoding costs are the number of cycles to perform each instruction in the *Intel-386* architecture. Table 6.5 is a set of applicable productions and their costs[3]. The *AST* for the expression

production	code	cost	comment
`Reg = add(Reg,Reg)`	addl r,r,r	2	add two registers
`Reg = add(Reg,deref(Adrs))`	addl r,m,r	6	add memory var to register
`Reg = deref(Adrs)`	movl m,r	4	load memory var to register
`Reg = mul(Reg,Lit4)`	shll r,2,r	3	shift register left by two
`Reg = mul(Reg,Lit4)`	leal (r,4),r	2	scaled mode can multiply by four
`Reg = add(Reg,Reg)`	leal (r,r),r	2	leal can also add two registers
`Reg = add(Reg,mul(Reg,Lit4))`	leal (r,r,4),r	2	... or a reg and a 4× scaled reg

Figure 6.5: productions for second example

is shown in figure 6.6. The optimal encoding of this expression, based on these costs, is shown as annotations to the figure 6.7. Since the typeform of the root node must be the register form *Reg*, the minimal cost is 22 cycles.

[3] In this table a special typeform *Lit4* has been introduced to denote a literal node which just happens to have the value four. An alternative approach is explained later.

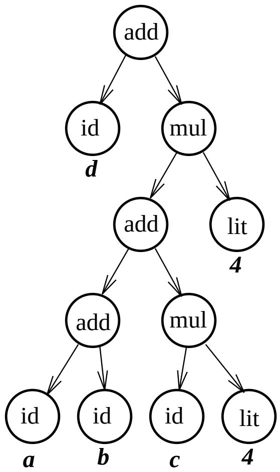


Figure 6.6: initial *AST* for the second example

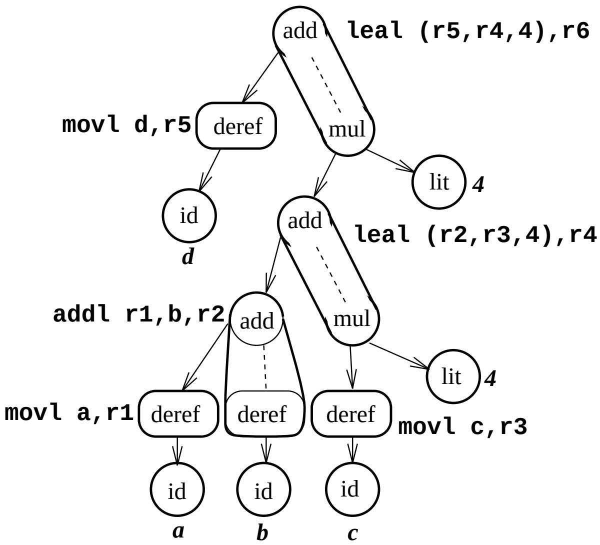


Figure 6.7: least cost rewritten code-tree for the second example

This particular example is quite difficult, because of the possibility of using the *load effective address long* (`leal`) instruction in unconventional ways. This instruction contributes other patterns also, such as the addition of two registers, with or without scale factors, and an immediate literal operand. Recognizing all of the possible patterns due to this instruction, taking into account all possible ordering of operands in the tree, adds considerable complexity to an interpretive code generator of the kind discussed in the previous chapter.

6.2 Labelling and reducing

The optimal rewriting of expression trees is based on two passes over the tree. A bottom-up pass labels the tree, and a top-down pass then chooses the rewriting productions. We call these two passes *labelling* and *reducing*.

6.2.1 Structure of the attributes

Firstly, we assume that there is some finite set of productions, which are indexed by an ordinal type *ProdIndex*. We also assume that there is some finite set of typeforms *FormEnum* which are the possible result-typeforms of nodes. These will usually include typeforms such as *Register, Immediate* and various kinds of symbolic addresses.

Productions are specified by the pattern which they match, the result typeform which they produce, and a strictly non-negative cost function. Usually the cost function will be constant, but allowing more general functions is sometimes useful. The patterns are specified by a parenthesized prefix notation. Thus "`op`(*lFrm,rFrm*)" specifies a pattern matched by a node "`op`" with children of typeforms *lFrm* and *rFrm* respectively. Using the parenthesized prefix form, it is possible to define patterns in which the typeforms of more distant offspring are specified. The examples in figure 6.2 illustrate the possibilities.

In terms of the usual attribute grammar notation, each production is a rule for computing a single element of a set-valued synthesized attribute. Patterns which do not match the typeforms of the pattern-leaf nodes are conventionally treated as though they returned an unbounded, distinguished cost value *maxC*.

It is usual to also have productions which consist of a single typeform name on each side of the production symbol. Such rules are called *chain productions*, and correspond to operations which are able to coerce the typeform of nodes.

We shall assume that every node of the tree is annotated with a *state* structure, which holds an array of pairs, indexed on the ordinal typeform type. The abstract syntax is as follows.

```
FormAnnotation ==> costArray  : ARRAY TypeForm OF FormInfo;
FormInfo       ==> ruleIndex  : ProdIndex,
                   costValue  : CARDINAL;
```

During the bottom-up labelling traversal, the arrays will be evaluated, with information corresponding to all possible result typeforms, and their associated minimal costs. During the top-down reduction traversal, the inherited *required typeform* attribute will select just one of the array elements as the chosen production index of that particular visit.

6.2.2 The labelling pass

Nodes of the tree are labelled in bottom-up order, during a depth-first recursive traversal. Each node is visited exactly once. Thus when each node is labelled, any descendant nodes will have already been labelled.

We shall assume that the array's elements are initialized with the empty production, and the maximum possible cost. At each node the following computation is performed —

```
FOR every matching pattern DO
  compute the pattern cost C;
  IF C is less than current cost for that form THEN
    replace current element with new (index,cost) pair
  END;
END;
compute chain-closure of result typeform set ;
```

The pattern cost is found by summing the minimum costs of the result typeforms which are matched in the pattern leaf nodes, and adding the cost of the matched production. With this particular algorithm, if several patterns result in the same cost, the first minimal-cost production encountered will be stored.

We must also take into account the possibility of chain rules being applied to the result typeforms of matching patterns. In fact, we must take into account all typeforms which are reachable by means of all possible sequences of one or more chain rules. We call the set of typeforms which can be reached by chain rules the *chain-closure* of the result typeform set. The final node labelling is computed as follows —

```
(* compute the Chain-Closure *)
FOR every form R in result set DO
  FOR every T in chain-closure of R DO
    compute the cost of producing T;
    IF C is less than current cost for that form THEN
      replace current element with new (index,cost) pair
    END;
  END;
END;
```

Let us revisit the second example from section 6.1.3. We must now be a little more formal about the possible result typeforms of nodes, taking into account all the possible typeforms of symbolic addresses. In particular, we allow the load-effective-address instruction to load any general address expression into a register. We also allow for the folding of literal components of address expressions at compile time. Figure 6.8 has the detailed productions and costs for this example. The scaled-register form *SReg* is the product of a register and a literal value, provided that the literal has one of the allowed values, 2 or 4. We may think of these node typeforms as arising from the use of production 13 with a (conditional) cost expression —

$$\ddagger = \textbf{if } \textit{lit-val} \in \{2,4\} \textbf{ then } 0 \textbf{ else } maxC$$

Several of the productions simply accumulate parts of symbolic addresses, and hence cause no instruction to be emitted, and have zero cost.

	production	code	cost
1	`Reg = add(Reg,Reg)`	addl r,r,r	2
2	`Reg = add(Reg,deref(Adrs))`	addl r,m,r	6
3	`Reg = deref(Adrs)`	movl m,r	2
4	`Reg = Adrs`	lea m,r	2
5	`Adrs = Imm`		0
6	`Adrs = Reg`		0
7	`Adrs = SReg`		0
8	`Adrs = add(Reg,Reg)`		0
9	`Adrs = add(Reg,SReg)`		0
10	`Adrs = add(Adrs,Imm)`		0
11	`Imm = literal`		0
12	`Imm = ident`		0
13	`SReg = mul(Reg,literal)`		‡

Figure 6.8: detailed productions for second example

Figure 6.9 has the code-tree for the *AST* in figure 6.6. Each node is labelled with its node number at the upper left. Starting at node 8, the lowest `add` node in the figure, we note that each of the subtrees may be placed in the *Reg* form for a cost of 4, using production 3. Node 8 may take the register form at a cost of 10, using production 2. In fact, there are other choices with the same cost.

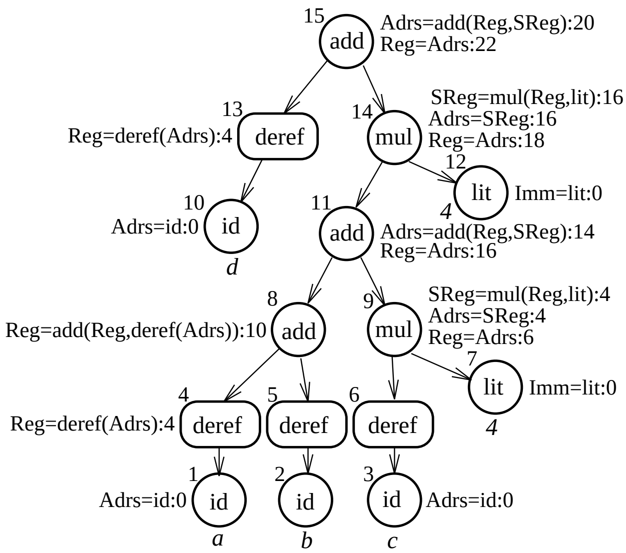


Figure 6.9: *AST* for the second example, labelled with (production:cost) pairs

Node 6 may take the register form at a cost of 4, and the literal at node 7 is 4. Hence at node 9 production 13 matches, producing the *SReg* form at a cost of 4. We may successively apply the chain rules 7 and 4, to produce a *Reg* form at node 9 at a cost of 6.

At node 11 we may add the two *Reg* forms at nodes 8 and 9, for a total cost of 18 (10 plus 6 plus two for the production), but we may match production 9 to produce an *Adrs* form for a cost of 14, and then use the chain rule production 4 to obtain a *Reg* form at a cost of just 16.

Continuing similarly, we finally arrive at the labelling shown in figure 6.9.

6.2.3 The reduction pass

The reduction pass over the labelled tree is also a recursive traversal. In this case, the recursion carries with it the *required typeform* for each visited node as an inherited attribute. The overall strategy is as follows —

```
PROCEDURE VisitNode(this : NodeDesc; form : FormEnum);
  VAR cFrm : FormEnum;
      pIdx : ProdIndex;
BEGIN
  select array element indexed by form;
  pIdx := prodIndex of matching element;
  FOR each pattern-leaf-node in production pIdx DO
    cFrm := pattern-leaf-form in selected production;
    VisitNode(pattern-leaf-node,cFrm);
  END;
  do semantic action for production pIdx;
END VisitNode;
```

It should be noted that the recursion in this traversal is not *structural recursion* controlled by the structure of the tree, but is rather a *semantic recursion* which is controlled by the visit order demanded in the production. In particular, a chain production will cause the same node to be revisited, while other recursions may even skip directly to distant decendants.

Consider emission of code for the expression $a - (b \times c)$ where a, b, c are floating point double variables, and where, as usual, we assume that the three variables are in memory. The applicable productions are shown in figure 6.10, and

	production	**code**	**cost**
1	`Treg = dToT(derefD(Adrs))`	`fld m`	25
2	`Treg = subT(Treg,Treg)`	`fsub`	26
3	`Treg = subT(Treg,dToT(derefD(Adrs)))`	`fsub m`	28
4	`Treg = subT(dToT(derefD(Adrs)),Treg)`	`fsubr m`	28
5	`Treg = mulT(Treg,Treg)`	`fmul`	29
6	`Treg = mulT(Treg,dToT(derefD(Adrs)))`	`fmul m`	32
7	`Treg = mulT(dToT(derefD(Adrs)),Treg)`	`fmul m`	32

Figure 6.10: detailed productions for third example

the labelled *AST* is shown in figure 6.11.

We expect to produce the following *VAL* code from the traversal —

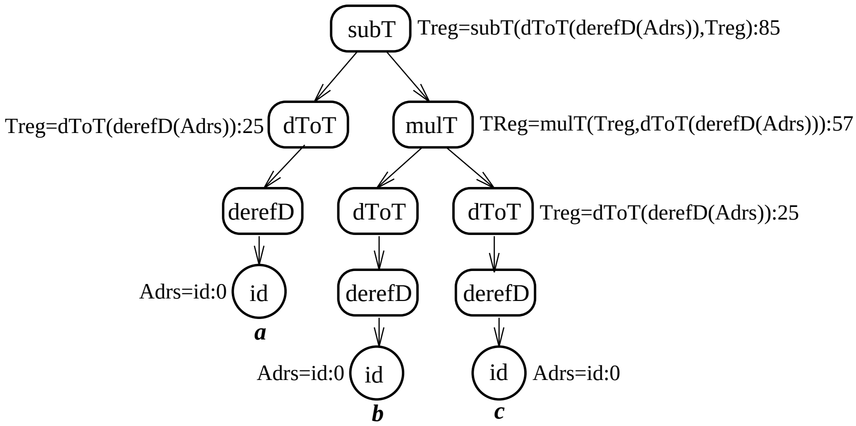


Figure 6.11: labelled *AST* showing reversal of node visit order

```
fld b            ; push operand b onto fp-stack
fmul c           ; multiply c to top-of-stack
fsubr a          ; reverse subtract top from a
```

This code makes use of production 4, which has the memory operand on the left, and uses the reversed subtraction instruction `fsubr`. Without this instruction, we would have needed to load the left operand and use production 2 at an additional cost of 25 cycles. There is no need for a reversed multiplication instruction, since multiplication is commutative. If we wished to multiply a memory operand on the left by an expression result in a *Treg* then we would simply use production 7. This emits the same `fmul` *mem* instruction as production 6, even though the memory address attribute used in the semantic action comes from the left rather than the right subtree.

6.3 Implementing bottom-up rewriting

In this section we look at some further details for the implementation of bottom-up tree rewriting. We shall assume the type definitions in figure 6.12.

The main data structure of the *AST* comprises a tree of nodes of *NodeInfo* type, accessed from a statically declared root. Each node contains a state array of costs and rule indices, indexed by the *FormEnum* enumeration. When the tree is built, the costs are all set to the limit value *maxC*. The rules are all initialized to the zero production index value, which is reserved to denote an invalid production.

For the moment we shall assume that there is a global array of *ProdInfo* records which contain information on each production.

Throughout this section we shall treat the case of the *SPARC* processor, the addressing modes of which were discussed in section 5.2.2. We shall process the full grammar of the address computations, and some typical binary operations. In particular, we shall distinguish between those cases where addition,

```
TYPE  NodeDesc  = POINTER TO NodeInfo;
      OperEnum  = (errTag, other enum values...);
      FormEnum  = (errFrm, other enum values...);

      NodeInfo  = RECORD
                    lOp, rOp  : NodeDesc;
                    operator : OperEnum;
                    state    : StateType;
                  END;

      StateType = RECORD
                    costs    : ARRAY FormEnum OF SHORTINT;
                    rules    : ARRAY FormEnum OF ProdIndex;
                    other data fields, as needed
                  END;
```

Figure 6.12: type definitions for implementation

for example, must be tested for overflow, and those cases in which it may be folded into an address computation. The table in figure 6.13 shows the subset of productions which we consider. In this example, instead of allowing conditional

	production	**condition**	**code**	**cost**
1	`Stmt = assign(Adr,Reg)`		`st r,adr`	1
2	`Stmt = assign(add(Reg,Reg),Reg)`		`st r,(r,r)`	1
3	`Reg = deref(Adr)`		`ld adr,r`	1
4	`Reg = deref(add(Reg,Reg))`		`ld (r,r),r`	1
5	`Reg = add(Reg,Reg)`		`add r,r,r`	1
6	`Reg = add(Reg,Imm)`	&(lit<13bits)	`add r,i,r`	1
7	`Reg = addV(Reg,Reg)`		`addCC r,r,r`	1
8	`Reg = addV(Reg,Imm)`	&(lit<13bits)	`addCC r,i,r`	1
9	`Reg = Con`		`set c,r`	1
10	`Con = static`			0
11	`Con = Imm`			0
12	`Con = add(Con,Imm)`			0
13	`Imm = literal`			0
14	`Imm = add(Imm,Imm)`			0
15	`Adr = Reg`			0
16	`Adr = Con`			0
17	`Adr = add(Con,Reg)`			0
18	`Adr = add(Adr,Imm)`			0
19	`Adr = local`			0
20	`Reg = mul(Reg,Imm)`	&(lit=2^n)	`shl r,i,r`	1
21	`Reg = mul(Reg,Reg)`		`mul r,r,r`	8

Figure 6.13: productions for *SPARC* processor

cost expressions, we only permit the cost of each production to be a constant, but allow an optional Boolean predicate to be applied during pattern matching. Productions 6 and 8 in the figure each have a condition which limits the literal to fit into a 13-bit signed integer field. Production 20 has a condition which allows the replacement of a multiplication by a left shift only if the immediate literal is a power of two.

The production patterns are specified in a fully parenthesized prefix notation. The table shows only the *Word* version of the many dereference and assign instructions which apply to different datatypes.

For this production set, the typeform enumeration is —

```
TYPE TypeForm = (errFrm,Stmt,Reg,Adrs,Con,Imm);
```

where *Imm* is an immediate literal value, and *Con* is any compile-time constant, including symbolic location names. Production 10 is a nullary production which matches the name of a statically allocated memory location as a *Con* form, while production 19 matches the address of a local variable as an *Adr* form.

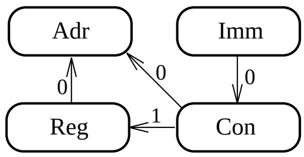


Figure 6.14: chain productions for sparc address modes

The coercion graph for the chain productions is shown as figure 6.14. Each edge in the graph is labelled with the cost of the chain production which is represents. From this graph we may conclude that a production to load an immediate value into a register would be redundant, since we may transform an *Imm* into a *Con* at zero cost, and then load the *Con* into a register by production 9. On the other hand, production 16 is not redundant, since it allows the use of a *Con* value as an *Adr* at zero cost. Taking the transitive path from *Con* to *Adr* through productions 9 and 15 would add one to the total cost.

In this particular case the chain-rule graph is acyclic. In principle there is no problem with chain-rules forming cycles, since we only continue along the chain so long as the corresponding element of *state.costs*[...] is reduced at each step.

6.3.1 Implementing labelling

We suggest a style of implementation which is "object oriented" in its reliance on method dispatch based on the operator and production enumerations. The main labelling recursion is launched by a single procedure. This procedure dispatches a labelling procedure selected from an array indexed on the root-node operator value.

```
PROCEDURE LabelNode(self : NodeDesc);
BEGIN
  labelProc[self^.operator](self);
END LabelNode;
```

The work is all done in the *labelProc* procedures, of which there is one for each operator. These procedures are hard-coded to check just those production right-hand-sides which apply to that operator, and encode the bottom-up recursion pattern according to the arity of the operator.

As an example, suppose we wish to match the productions which apply to assignment nodes in figure 6.13. The *labelProc* array will be initialized so that *labelProc*[*assign*] selects the procedure *assignMatch*, shown in figure 6.15. The

```
PROCEDURE assignMatch(self : NodeDesc);
  VAR rCost, lCost : INTEGER;
BEGIN
 (* first recurse down the tree *)
  labelProc[self^.lOp^.operator](self^.lOp);
  labelProc[self^.rOp^.operator](self^.rOp);
 (* now check for pattern matches *)
  IF (self^.lOp^.rules[Adr] <> 0) AND
     (self^.rOp^.rules[Reg] <> 0) THEN
   (* prod 1: Stmt = assign(Adr,Reg) *)
    rCost := self^.rOp^.costs[Adr]; lCost := self^.lOp^.costs[Reg];
    InsertStmt(self,1,rCost + lCost + 1);
  END;
  IF (self^.lOp^.operator = add) AND
     (self^.lOp^.lOp^.rules[Reg] <> 0) AND
     (self^.lOp^.rOp^.rules[Reg] <> 0) AND
     (self^.rOp^.rules[Reg] <> 0) THEN
   (* prod 2: Stmt = assign(add(Reg,Reg),Reg) *)
    rCost := self^.rOp^.costs[Reg];
    lCost := self^.lOp^.lOp^.costs[Reg]+self^.lOp^.rOp^.costs[Reg];
    InsertStmt(self,2,rCost + lCost + 1);
  END;
END assignMatch;
```

Figure 6.15: match procedure for assign nodes

procedure begins by recursing on its two subtrees. It then tries to match each of the two patterns which begin with a *assign*. Note that the first of these is a simple pattern, while the second is a more complex, nested pattern. The *Insert** procedures update one element of the state arrays of the *self* node, as detailed later.

The corresponding procedure for a unary operator is illustrated by figure 6.16 which is the procedure *derefMatch*.

In these procedures we test whether a node can possibly have a particular typeform T by checking if the rule $node\uparrow.rules[T]$ is non-zero. Here we make

```
PROCEDURE derefMatch(self : NodeDesc);
  VAR lCost : INTEGER;
BEGIN
 (* first recurse down the tree *)
  labelProc[self^.lOp^.operator](self^.lOp);
 (* now check for pattern matches *)
  IF self^.lOp^.rules[Adr] <> 0 THEN
   (* prod 3: Reg = deref(Adr) *)
    lCost := self^.lOp^.costs[Adr];
    InsertReg(self,3,lCost + 1);
  END;
  IF (self^.lOp^.operator = add) AND
     (self^.lOp^.lOp^.rules[Reg] <> 0) AND
     (self^.lOp^.rOp^.rules[Reg] <> 0) THEN
   (* prod 4: Reg = deref(add(Reg,Reg)) *)
    lCost := self^.lOp^.lOp^.costs[Reg]+self^.lOp^.rOp^.costs[Reg];
    InsertReg(self,4,lCost + 1);
  END;
END derefMatch;
```

Figure 6.16: match procedure for deref nodes

use of the fact that the rules are numbered from one, and the rules array is initialized to all zeros.

The procedures *Insert** update the cost and rule information, and invoke the chain-closure operation. In the case of *InsertReg*, figure 6.17, there is a single chain-closure transition.

```
PROCEDURE InsertReg(self : NodeDesc;
                    prod : ProdIndex;
                    cost : INTEGER);
BEGIN
  IF self^.costs[Reg] > cost THEN
    self^.costs[Reg] := cost;
    self^.rules[Reg] := prod;
   (* Reg has a single chain rule *)
    InsertAdr(self,15,cost);
  END;
END InsertReg;
```

Figure 6.17: example cost update procedure

Notice that since in general there will be multiple matches on each node, it might be worthwhile to not invoke chain-closure after each insertion, but rather do it once after all costs have been entered. The performance issues involved are unclear. In this case we suggest an insert procedure for each typeform, which calls the *Insert** procedure(s) which are adjacent on the chain-closure graph, figure 6.14. Notice particularly that these chain calls should be protected by the cost test, since costs can only decrease monotonically. Thus if an insertion fails to lower a cost, it is not possible for the chain-closure insertions to lower the costs of any dependent typeform.

Automatic production of labellers

The labelling procedures which are required for the labelling traversal may be automatically produced from specification. The *LabelNode* procedure is the same for all labellers, while the *Insert** and **Match* procedures may be derived from the productions.

The relationship between the productions and the **Match* procedures is as direct and predictable as it is between a context free production and a recursive descent recognizer. We simply need to collect up all those patterns which are specified with a particular node-tag as root, and enumerate the patterns.

For example, in the grammar fragment figure 6.13 there are six productions which match *add* nodes. Of these six, two start with a *reg* typeform. We are led directly to code along the lines of figure 6.18 In the *addMatch* procedure, note especially the natural way in which the optional predicate merges with the pattern matching for production 6. It is assumed that a function is used to test the range of the literal value, but it might equally well be tested by an inline Boolean expression.

An important special case of matching applies to leaf nodes. In such cases there are no patterns, and we have only chain rules to implement. Provided that the costs of chain rules are constant, and do not have predicates, we may optimize the matching of leaf nodes by simply copying precomputed fields into the *StateType* record, and avoid calls to the *Insert** procedures entirely. Figure 6.19 has a simple example.

The *Insert** procedures all have essentially the same body, with a tail call to the appropriate procedure for every adjacent typeform in the chain-closure graph. Thus once we have computed figure 6.14, we may immediately write figure 6.17.

6.3.2 Implementing reduction

In general, a reduction traversal over a labelled tree, entails the choosing of a typeform at each node visit, and the execution of some specified semantic action. We may visit a node repeatedly in the case of chain rules, or leave nodes out in the case of non-local patterns. Thus we must let the selected production determine the visit order of the traversal, and the semantic action performed at each node.

In order to make clear the issues involved, we shall consider the case of a reduction which emits *VAL* into a code buffer, using the same set of procedures used in the previous chapter.

As an example, consider the array access —

```
k := arr[i,j];
```

```
PROCEDURE addMatch(self : NodeDesc);
  VAR lCost : INTEGER;
BEGIN
 (* first recurse down the tree *)
  labelProc[self^.lOp^.operator](self^.lOp);
  labelProc[self^.rOp^.operator](self^.rOp);
 (* now check for pattern matches *)
  IF self^.lOp^.rules[Reg] <> 0 THEN (* prod 5 or 6 *)
    lCost := self^.lOp^.costs[Reg];
    IF self^.rOp^.rules[Reg] <> 0 THEN
     (* prod 5: Reg = add(Reg,Reg) *)
      InsertReg(self,5,lCost + self^.rOp^.costs[Reg] + 1);
    END;
    IF (self^.rOp^.rules[Imm] <> 0) AND
        FitsIn13(self^.rOp^.litVal) THEN (* Note the predicate here *)
     (* prod 6: Reg = add(Reg,Imm) *)
      InsertReg(self,6,lCost + self^.rOp^.costs[Imm] + 1);
    END;
  END;
  IF self^.lOp^.rules[Con] <> 0 THEN (* prod 12 or 17 *)
    lCost := self^.lOp^.costs[Con];
    IF self^.rOp^.rules[Reg] <> 0 THEN
     (* prod 17: Adr = add(Con,Reg) *)
      InsertAdr(self,17,lCost + self^.rOp^.costs[Reg]);
    END;
    IF (self^.rOp^.rules[Imm] <> 0) THEN
     (* prod 12: Con = add(Con,Imm) *)
      InsertCon(self,12,lCost + self^.rOp^.costs[Imm] + 1);
    END;
  END;
  IF (self^.lOp^.rules[Imm] <> 0) AND
     (self^.rOp^.rules[Imm] <> 0) THEN
   (* prod 14: Imm = add(Imm,Imm) *)
    InsertImm(self,14,self^.lOp^.costs[Imm] + self^.rOp^.costs[Imm]);
  END;
  IF (self^.lOp^.rules[Adr] <> 0) AND
     (self^.rOp^.rules[Imm] <> 0) THEN
   (* prod 18: Adr = add(Adr,Imm) *)
    InsertAdr(self,18,self^.lOp^.costs[Adr] + self^.rOp^.costs[Imm]);
  END;
END addMatch;
```

Figure 6.18: match procedure for add nodes

where *arr* is a *VAR* parameter array of 16×16 integers. The integer local variables i, j, k are at offsets –4, –8, and –12 respectively, and the parameter is at offset +68 in the stack frame. The expression on the right-hand-side has been considered in detail in the previous chapter.

```
PROCEDURE literalMatch(self : NodeDesc);
BEGIN
  self^.lOp^.costs := CostArray{maxC,maxC,  1,  0,  0,  0};
  self^.lOp^.rules := RuleArray{   0,   0,  9, 16, 11, 13};
END literalMatch;          (* errFrm,Stmt,Reg,Adr,Con,Imm *)
```

Figure 6.19: “match” procedure for literal nodes

With the productions in the figure 6.13, the labelling of the tree for this example gives us figure 6.20. Alongside each node are one or more labels, showing

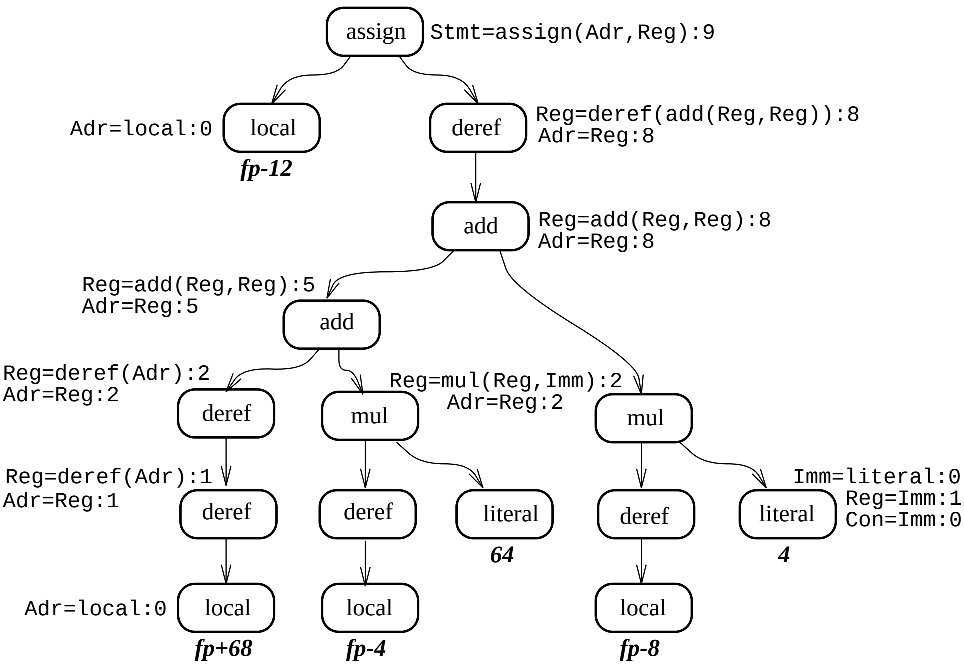


Figure 6.20: code-tree for array access example, labelled for sparc

the applicable productions, with the minimal cost after the colon. To save space on the diagram, the nodes have been arranged so that the production and cost labels apply to all the nodes with the same operator at the same level. For example, the three production and cost labels at the bottom left apply to all of the local-name nodes *fp* $\pm$ *offset*.

In order to implement the reduction traversal, we require a recursive framework which is able to implement the dependent node visit-sequence demanded by the particular production which is selected by the *required typeform* attribute. The figure 6.21 suggested a kind of interpreted implementation, where the recursion is determined by production information available in a table. However, an attractive alternative is to encode the visit order for each production into a separate procedure, and to dispatch these procedures from an indexed table. The recursion could be launched by the procedure shown in figure 6.21. As usual,

```
PROCEDURE Reduce(this : NodeDesc; form : FormEnum);
  VAR prodOrd : ProdIndex;
BEGIN
  prodOrd := this^.rules[form];
  (* now dispatch procedure recHelp[prodOrd] *)
  recHelp[prodOrd](this);
END Reduce;
```

Figure 6.21: implementation of reduction framework

the work is all done in the dispatched *recursion helper* procedures. These procedures encode the required visit order, and the user-specified semantic actions. A typical procedure, for production 1 is shown in figure 6.22. This reduction

```
PROCEDURE Reduce1(this : NodeDesc);
  VAR leaf1, leaf2 : NodeDesc;
BEGIN (*  Prod 1 Stmt = assign(Adr,Reg) *)
  leaf1 := this^.lOp;
  recHelp[leaf1^.rules[Adr]](leaf1);
  leaf2 := this^.rOp;
  recHelp[leaf2.rules[Reg]](leaf2);
  EmitRtoAdr(sw,leaf2^.dstReg, adr from left tree);
END Reduce1;
```

Figure 6.22: reduction helper for production 1

helper procedure is called whenever production 1 has been selected. In order to find out which helper procedures to call recursively, the state of the nodes which are the leaves of the pattern in production 1 are examined.

Production 1 demands that its left node be in the *Adr* form. The element $rules[Adr]$ of the left child holds the index of the production which gives the lowest cost *Adr* form for the left subtree. This value thus determines the reduction helper procedure to recursively call on the left child.

Similarly, production 1 demands that its left node be in the *Reg* form. The element $rules[Reg]$ of the right child holds the index of the production which gives the lowest cost *Reg* form for the right subtree. This value thus determines the reduction helper procedure to recursively call on the right child.

After the recursion returns, an appropriate store word *VAL* instruction is emitted into the code buffer. Notice that the parameters to the call which emits the *VAL* depend on synthesized attributes such as the register ordinal of the right-hand expression. Because the semantic action is placed *after* the recursion returns, the procedure can access these values in the decendant nodes.

A more complex case arises with production 2, which has a nested pattern. The reduction helper for this production is shown in figure 6.23. In this case,

```
PROCEDURE Reduce2(this : NodeDesc);
  VAR leaf1, leaf2, leaf3 : NodeDesc;
BEGIN (*  Prod 2 Stmt = assign(add(Reg,Reg),Reg) *)
  leaf1 := this^.lOp^.lOp;
  recHelp[leaf1^.rules[Reg]](leaf1);
  leaf2 := this^.lOp^.rOp;
  recHelp[leaf2^.rules[Reg]](leaf2);
  leaf3 := this^.rOp;
  recHelp[leaf3^.rules[Reg]](leaf3);
  EmitRtoA2r(sw,leaf3^.dstReg,
              leaf1^.dstReg, leaf2^.dstReg);
END Reduce2;
```

Figure 6.23: reduction helper for production 2

the matched pattern has *three* leaf nodes, so that we recurse three times. First the *left-left* grandchild node, then the *left-right* grandchild node, then finally the *right* child. As in the simpler case, the reduction helper is chosen by indexing into an array of procedure variables, using the production index held in the rules array of the pattern-leaf nodes.

In this case, the semantic action deposits a *VAL store word* "`sw`" instruction in the buffer with an address specified in the `A2r` (*address with two index registers*) form.

Some productions do not cause the emission of any instructions. Instead, such instructions compute synthesized attributes which may be used by productions further up the tree. Many productions emit instructions, but also need to compute some attributes for other productions. For example, in order to emit *VAL*, every production which has a left-hand side in the *Reg* form will allocate a virtual register as a destination register, and will enter this attribute in a *dstReg* field.

As a final reminder of the effect of chain-rules, consider the reduction helper procedure for the chain-rule production 15, shown in figure 6.24. The object of this reduction is to create an address value out of a register form. Thus the procedure revisits the same node to invoke the reduction which produces the register value. This reduction will allocate a destination register, and emit code to compute the value into the register. The semantic action of production 15 takes the destination register from the recursion, and creates an address in the (name, index-register, offset) form.

Automatic production of reducers

The actions which are performed during the reduction traversal of a labelled tree are dependent on the purpose of the rewriting. In the case considered in the last

```
PROCEDURE Reduce15(this : NodeDesc);
  VAR leaf1 : NodeDesc;
BEGIN (*  Prod 15 Adr = Reg *)
  leaf1 := this;
  recHelp[leaf1^.rules[Reg]](leaf1); (* revisit same node! *)
  this^.adrNam := noName;
  this^.adrReg := this^.dstReg;
  this^.adrOff := 0;
END Reduce15;
```

Figure 6.24: reduction helper for production 15

section this was the emission of calls to procedures which place *VAL* instructions in a buffer.

The recursion pattern of the helper procedures may be derived from the form of the production right-hand-sides, so at least part of the code of these procedures may be automatically produced. Any semantic actions required as part of the recursion must either be specified as an extension to the grammar, or editted by hand into the automatically produced templates.

The general rule for the creation of the recursion helper is straightforward. For the selected production, the dependent nodes which must be visited, and the typeform parameter which they inherit is entirely determined by the production right-hand-side.

Unary productions visit the left child "`this^.lOp`", while binary productions first visit the left child "`this^.lOp`" then the right child "`this^.rOp`". Chain productions revisit the same node "`this`".

Taking into account the possibility of non-local patterns is also straightforward. For example, the production —

$$Reg = \texttt{add(}Reg,\texttt{deref(}Adr\texttt{))}$$

has the visit order — "`this^.lOp`" with the production selected by the *Reg* index in the rules array, followed by a visit to "`this^.rOp^.lOp`" with the production selected by the *Adr* index in the rules array.

It is usual that the semantic actions of the reductions depend on synthesized attributes from further down the tree. Thus it is normal to recurse down the tree passing the inherited typeform attribute *before* performing the semantic action on the node itself. It is possible to choose a syntax for the reducer specification which allows both *pre-recursion actions* and *post-recursion actions* to be specified.

There are circumstances for which the order of emission of code for the subtrees of a node needs to be controlled by attributes other than the typeform. For example, the order of code emission which uses the least number of physical registers relies on the computation of the *Sethi number* attributes, as explained in section 7.1.5. These numbers are synthesized attributes, and cannot be exactly calculated until after the reduction traversal has been performed. In this case, it is probably easiest to use the semantic actions of the reduction to compute the Sethi numbers, and perform a separate pass to emit the code.

6.4 Putting it all together

Bottom up tree rewriters can work by directly traversing a code-tree created from an abstract syntax tree, or they may traverse a tree which is reconstructed from an intermediate form such as *DCode*.

In terms of efficiency, it is possibly preferable to traverse a tree produced by a compiler frontend, and take the opportunity to do without a conventional intermediate representation. This is the structure which is hinted at in the modified architecture on page 161.

6.4.1 Traversing code-trees

If a tree rewriter traverses the tree produced by the compiler frontend, then the frontend must transform the *AST* into a *code-tree* either as part of static semantic attribution, or following attribution.

Consider the three trees in figure 6.25. In this figure the leftmost tree is an

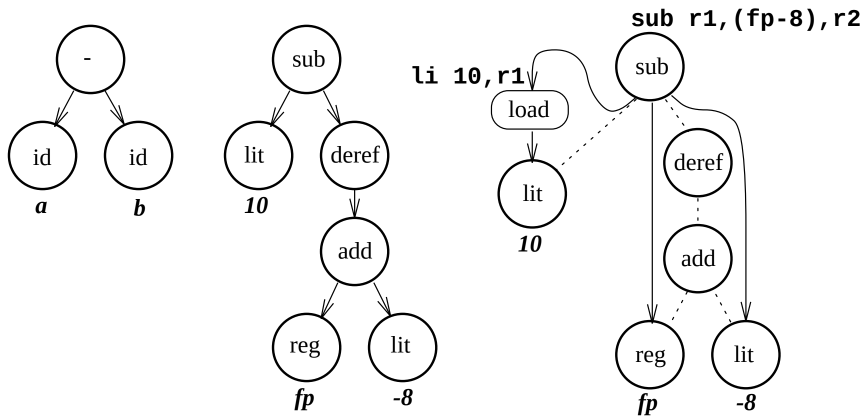


Figure 6.25: abstract syntax tree, code-tree, instruction tree

abstract syntax tree arising from a source language expression such as “`(a-b)`”. In the middle, is the result of static analysis. The names have been bound, and it has been discovered that the name a denotes a constant of value 10, while the name b denotes a local variable at an offset -8 from the frame-pointer fp. The middle tree is the code-tree which expresses the actual computation required to evaluate the expression.

The tree at the right shows a covering which implements the expression evaluation for a *CISC*-style machine which can take one of its operands from memory, and has an indexed address mode. Note that in the rewriting of the tree an extra node hase been introduced to signify the loading of the literal into a register, using the *load immediate*, "li" instruction. All four nodes in the right subtree have been subsumed into the address modes of the other instruction which subtracts the value in memory, at location (*fp–8*), from the value in the register r1. The result is placed in the register r2.[4]

The chief argument in favor of a concrete intermediate form is that the form partitions the design of frontend and backend, so that the first may be target independent, while the latter may be language independent. In the case of code selection by bottom-up tree-rewriting of a code-tree, the tree itself is the intermediate form. If the same tree rewriter specification is to be reused for multiple languages, then special care must be taken to ensure that the code-trees make explicit all of the details of the language semantics. As an example, consider the *MOD* operation in the two related languages *Modula-2* and *Modula-3*. The semantics of the operation in the two languages are *almost* identical. Unfortunately, in the earlier language an exception must be raised if the right operand is less than zero. In the later language, the result is defined for all non-zero values of the divisor. In *DCode* the expression "a MOD b" has the translations —

Modula-2

```
pshAdr _a
derefW
pshAdr _b
derefW
dup1
test _modTrap,1,2147483647
mod intOver
```

Modula-3

```
pshAdr _a
derefW
pshAdr _b
derefW
mod intOver
```

In the case of a common intermediate form, the two separate tree-walkers are part of separate frontends, and each understands the semantics of the language for which it emits *DCode*. However, if the *same* tree-rewriter is to be used for the two languages, then the necessity for the test-and-trap must already be explicit in the code-tree.

Note that if a rewriter is only to be used on a single language, then knowledge as to whether *MOD* needs a test or not can be easily incorporated into the semantic actions of the rewriter. Such a choice would certainly make the *AST* to code-tree transformation much simpler. However, in such a case the rewriter would not be correct for other languages, and the specification could not be reused.

6.4.2 Traversing reconstructed trees

Although it maybe somewhat less efficient to do so, it is entirely possible to construct code-trees from a stack-based intermediate form. The interface between the *AST* tree-walker and the *code-tree* constructor might be either textual, or procedural, as shown in figure 4.1 on page 92.

[4] Of course, the rewritten tree on the right is usually not physically created. The solid edges in this tree denote the virtual tree which corresponds to the visit order in the reduction traversal.

Code trees are created bottom-up from abstract stack machine code. A shadow stack holds pointers to a forest of subtrees, which are connected together and pushed back onto the shadow stack. For example, when a binary operation is recognised, a new binary tree node is allocated, and the two subtrees popped from the top-of-stack are attached as right and left subtrees of the new node. A pointer to the new node is pushed as the new top-of-stack.

Because the trees are constructed bottom up, it is possible to label the trees as they are constructed. Such an *incremental labelling* removes the need for a separate labelling traversal. Seen in this light, the transitory appearance of the abstract stack machine code may be thought of as no more than a systematic aid to transforming from *AST* to *code-tree*. From such a point of view, the removal of the separate labelling traversal of the tree is a welcome bonus.

6.5 Summary

Bottom up tree rewriting is the most favoured method of code selection at present. The rewriters are almost always automatically produced from specifications. Using this technology, code selectors may be easily maintained and are easily modified.

Bottom-up rewriting code selectors which perform their dynamic programming at compile time tend to run somewhat slower than a well hand-coded shadow stack automaton. However, a tree rewriter based on dynamic programming at compiler-compile time would almost certainly be faster than any hand written selector. Nevertheless, the time differences involved are not a significant proportion of the total compile time, particularly for those compilers which perform significant levels of code optimization.

The relationship between bottom up tree rewriting and interpretive code selection should now be clear. The non-terminal symbols of the tree grammar correspond directly to the various symbolic value variants in the shadow stack formalism. The cases where dynamic programming brings a benefit are exactly those cases where it is necessary to hold a value as a symbolic quantity on the shadow stack, until more information is known about the uses of the value.

For extremely simple instruction sets architectures, selection of instructions is correspondingly simple. In such cases tree rewriting seems to bring no performance benefit, although it may be a good choice for other reasons. However, for architectures with complex addressing modes, or for complex instruction sets, tree rewriting can bring measurable benefits. Over time, as chip designers get a larger ration of silicon, instruction set architectures seem to be becoming more complex, even for load/store architectures. Thus, is seems certain that the benefits of tree rewriting over interpretive code selection will become more apparent in future.

6.6 Finding out more

A number of bottom-up rewriting tools are freely available, with slightly different behaviour and specification requirements.

"`Iburg`"[?] is a tool which produces bottom up tree rewriters which perform dynamic programming at compile time. The generated tree-rewriter is coded in *C*. `Iburg` uses a `yacc`-like specification style, and allows production costs to be arbitrary expressions. For this tool, semantic constraints are enforced by the use of conditional cost-functions which return *max-cost* for disallowed cases. Fraser and Hanson's book [?] on the compiler `lcc` has detailed examples of the use of a rewriter based on `iburg`.

"`Mburg`"[?] is a tool which produces bottom up tree rewriters which, like those created by `iburg`, perform dynamic programming at compile time. In this case, the generated rewriter is written in *Modula-2*. The specification style is similar to `coco/r`, rather than `yacc`. Productions are specified in the usual nested prefix form, and each production may optionally be guarded by an arbitrary predicate, as described in section 6.3. Each production may also specify a semantic action, which is automatically placed after the recursion in the reduction helper procedures. For rewritings which can be performed in a single reduction pass, `mburg` can produce a complete rewriter with no hand-coding at all. *Mburg* also supports the incremental labelling referred to in the previous section.

"`Burg`"[?] is a tool which performs dynamic programming at compiler-compile time, and generates a a table-driven, finite-state labelling automaton. The specification style is identical to `iburg`, except that costs may only be constants. In practice some transformation of typical `iburg` grammars is usually required. `Burg` rewriters are faster than those produced by any other of the tools mentioned here, typically by a factor of about 5.

Performing the dynamic programming at compiler-compile time, requires the summarization of an infinite number of possible cost-states into a finite (and hopefully small) number of equivalence classes. These equivalence classes become the states of the labelling automaton. In theory, there are some tree-grammars for which the process does not converge, but these do not appear to arise in practice. The most comprehensive account of `burg`, and its underlying theory, is given by by Proebsting[?].

6.7 Exercises

6.1 IBM's Power architecture, and the latest MIPS instruction set both provide floating point instructions which take three floating point registers as input, denoted f_1, f_2, f_3 here, and compute one of the following expressions into a fourth register f_4 —

$$\begin{aligned} f_4 &= (f_1 \times f_2 + f_3) \\ f_4 &= (f_1 \times f_2 - f_3) \\ f_4 &= -(f_1 \times f_2 + f_3) \\ f_4 &= -(f_1 \times f_2 - f_3) \end{aligned}$$

Using the usual prefix notation, write down productions which match these patterns in a code-tree.

6.2 In the following "*DReg, IReg, Adrs*" are non-terminals of a tree grammar, denoting values in floating point double registers, integer registers, and symbolic addresses, respectively. Using this notation, a standard word-assignment production is —

$$\mathit{Stmt} = \texttt{assignW(}\mathit{Adrs},\ \mathit{IReg}\texttt{)}$$

Consider conversions between integer and floating point double values on those machines which do not have direct data paths between the floating point registers and the integer registers. In such a case, a production such as —

$$\mathit{IReg} = \texttt{dToI(}\mathit{DReg}\texttt{)}$$

requires the floating point value to be transformed into an integer, still in a coprocessor register, then written to memory, and then reloaded into an integer register. In the case that the integer register is immediately assigned to some memory location, this is very inefficient. A similar inefficiency applies to the conversion —

$$\mathit{DReg} = \texttt{iToD(}\mathit{IReg}\texttt{)}$$

when the integer value is already available in memory.

Bypass these inefficiencies, either by adding new conversion productions, or new assign productions, or both.

6.3 It is tempting to suggest that another round of constant folding might be built in to a bottom up tree rewriter. For example, the production —

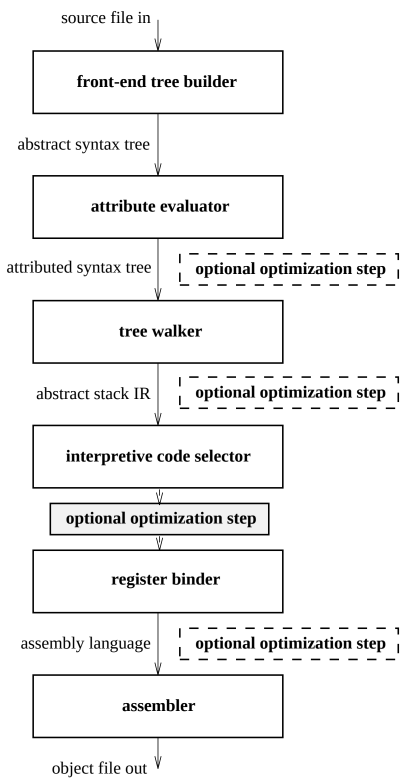
$$\mathit{Imm} = \texttt{add(}\mathit{Imm},\ \mathit{Imm}\texttt{)}$$

might be used with a reduction action which sets the literal value to the sum of the two leaf values. However, this is incorrect if other productions rely on testing the value of literals either using conditional expressions or pattern predicates. Why is this?

6.4 Sometimes it is intuitively clear that a particular terminal symbol in a tree grammar can only result in a single non-terminal *type-form*. In such cases, it would appear that the subtree rooted at such a node could usefully be reduced immediately. Determine sufficient conditions on the productions of a tree grammar, to ensure that such an immediate reduction can safely be performed.

Chapter 7

Introduction to Optimization



7.1 Introduction

In this introductory section a brief overview of local optimization is given. The implementation of three techniques: local common subexpression elmination, local value tracking, and peephole optimization, are treated in detail in later sections of this chapter. A further local optimization, instruction scheduling, has the whole of a later chapter devoted to it.

In section 1.7 the notions of safety and profitability were briefly introduced. The working definitions which we shall use are —

> **Definition:** An optimization is said to be **safe** if for every correct program the optimized code has equivalent input-output behaviour as the same program compiled without the optimization. In particular, the two compiled program versions must fail for exactly the same input data sets.

> **Definition:** An optimization is said to be **profitable** if the transformed program is an improvement over the original with respect to the objective of the optimization.

Basic blocks and extended basic blocks

A **basic block** is a segment of code which has a single entry and a single exit point. The control flow in a basic block thus consists of sequential execution with no branching. A basic block has the fundamental property that if any given instruction of the block is executed, then all previous instructions must already have been executed, and all following instructions will be executed.

In *VAL*, basic blocks begin at the entry point to procedures, at labels, and immediately following branch instructions. Basic blocks end at branches, so that the last instruction of a block is often a conditional or unconditional branch (including return instructions). They also end immediately before a label. For these purposes, it is usual to treat procedure calls as complex instructions with possible side-effects, rather than as jumps or branches.

Local optimizations are those which are performed on the basis of information which is accumulated for a single basic block at a time. This introduction outlines the ideas behind the two important local optimizations: common subexpression elimination, and value tracking. Both of these optimizations have global counterparts which are treated in a later chapter.

As it turns out, for the two local optimizations which we shall be discussing, the restriction to a basic block as defined above may be relaxed. We define an **extended basic block** as a segment of code which has a single entry point and one or more exit points. An extended basic block has the fundamental property that if any given instruction is executed then all previous instructions in the block must have already been executed.

In *VAL*, extended basic blocks begin at the entry point to procedures, and at labels. Extended basic blocks end at return instructions, unconditional branches, and immediately before labels.

7.1.1 Strength reduction and constant folding

There are a number of optimizations which involve modification of a local instruction sequence so as to replace the sequence with a less expensive sequence. Such a replacement is called *strength reduction*.

An example of strength reduction which we have already noted, is the replacement of multiplication by shifting, in cases where the multiplicand is a literal power of two, and overflow trapping is not required. On almost all machines, such a shift would be faster than a multiplication. Indeed on most machines the cost of a multiplication is equivalent to perhaps between 5 and 50 simple instructions. If the factor is towards the high end of this range, as occurs for machines without a hardware multiplication instruction, then multiplication by many constants may be strength reduced. For typical *RISC* instruction sets, multiplication by any number in the range $[-7..7]$ can be performed in three instructions or less.

It is possible to deal with larger constants by means of a recursive algorithm. The typical such algorithm would look similar to the schema in figure 7.1.

```
PROCEDURE InlineMul(add :  BOOLEAN; mult :  INTEGER);
  VAR M,L : INTEGER;
BEGIN (* pre :  mult is positive, left-op name is mCnd *)
  IF mult ∈ [0..7] THEN
    emit case code to add/sub mCnd*mult to total
  ELSE
    find smallest M, largest L such that 2^M ≥ mult > 2^L
    IF most significant non-zero bits of mult are 111 THEN
      emit code to add/sub SHIFT(mCnd,M) to total
      InlineMul(NOT add, M**2 - mult);
    ELSE (* do just one bit *)
      emit code to add/sub SHIFT(mCnd,L) to total
      InlineMul(add, mult - L**2);
    END;
  END;
END InlineMul;
```

Figure 7.1: Recursive strength reduction for multiply

This is a variant of "multiplication with shifting over ones". The algorithm has been simplified[1] for purposes of clarity, and assumes that the initial multiplier is positive. In effect the initial call is

`InlineMul(mult>= 0, ABS(mult))`

As an example of the use of the algorithm in figure 7.1, if the multiplicand is in some register v_m, the result is accumulated in register v_t, and the multiplier is 47 (that is, 101111 binary), then the steps are as follows —

[1]Indeed a more complex algorithm due to Bernstein[?], produces lower cost evaluations for many multiples.

Compile-time computation	*Runtime operations*	*Emitted instructions*
InlineMul(add=true,mult=47),$M = 6, L = 5$	$v_t \leftarrow v_m \times 32$	`shl` $v_m, 5, v_t$
InlineMul(add=true,mult=15),$M = 4, L = 3$	$v_t \leftarrow v_t + v_m \times 16$	`shl` $v_m, 4, v_1$
		`add` v_t, v_1, v_t
InlineMul(add=false,mult=1)	$v_t \leftarrow v_t - v_m$	`sub` v_t, v_m, v_t

The multiplication by 47 has been strength reduced to four integer instructions. For most machine architectures this would be a useful improvement. It is normal to apply some instruction sequence length limit in such an algorithm to ensure profitability of the transformation. In the example above the length was four. For machines without hardware multiply instructions recursion depths much greater than this would still be profitable.

There are other cases of strength reduction which apply to loops. Suppose, for example, that each time around a loop an expression $\&a + c + i \times 8$ is computed, where $\&a$ is some fixed address, i is the loop counter, and c is some constant. At the end of each iteration, it is assumed that the loop counter i is incremented by one. This code might arise in the case of non-zero based indexing into an array of double precision floating point values.

An expression like the one in the example is said to be an *induction variable*. There is a simple relationship between the value of the expression between one iteration of the loop and the next. We can exploit this fact and avoid computing the expression each time we traverse the loop. Instead, we introduce a new variable and assign the initial value of the expression to this value in the loop header. At the end of each iteration, the variable is incremented, in this example by 8. We refer to this transformation as strength reduction, because the more expensive operation of computing the index expression has been replaced by the cheaper operation of incrementing the induction variable. If the induction variable is used several times within the loop, then the gain is correspondingly greater.

Finding induction variables in loops requires global analysis, so we defer further consideration of this topic until later in the book.

Constant folding is the process of replacing runtime computations by values which are computed at compile time. In general, computations involving constants can be evaluated without executing the program. Such constant expressions occur in numerous contexts, and it is usual for every phase of the compilation to try to recognise constant expressions, and replace each computation by its value.

In source programs symbolic expressions are often used to make the nature of a constant clear —

```
CONST twoPi = pi * 2.0;
```

or to parameterize a piece of code —

```
FOR index := 0 TO maxIndex DIV bytesInBitset DO
```

It goes without saying that such expressions will be evaluated by the front-end, and replaced by their result in the generated intermediate code.

However, constant expressions occur at later stages as well, sometimes as a result of other optimizations. Consider accessing an element of an array at some constant offset. The address computation will involve multiplying the constant index by some element size, and possibly adding some offset. Clearly this is a constant expression which was not visible to the front-end, and so must be folded by the code generator.

Finally, it turns out that one of the optimizations which we discuss later often tells us that the value of some particular variable is actually a constant at some point in the program. This *constant propagation* will introduce yet more opportunities for constant folding. Thus it is necessary to perform constant folding again *after* value tracking and many other optimizations.

7.1.2 Keeping information in registers

The retention of data in registers for multiple uses is very important, so as to avoid unnecessary data loads from memory. In the case of *RISC* machines, even as simple an expression as a loaded literal value, or the address of some entire variable may be a valuable common sub-expression.

A simple example hints at why this might be the case. Consider the simple task of reloading a statically allocated datum into a register in a *RISC* architecture machine. The symbolic assembly language might look as follows —

> `lw` *varName*,r_{dst}
> *use* r_{dst} *as operand*

where *varName* is a symbolic name for a variable. The location of the variable will be known at link-edit time. At that time the reference to the location is "fixed up" in the object file by means of a linker relocation. In general this fix-up will be a 32-bit literal.

Unfortunately, most of these machines only allow short literals in their opcode formats, and thus require two instructions to load a datum from such a symbolic address —

`lui`	`hibits'varName,`v_n	; *load upper half*
`lx`	`lobits'varName(`v_n`),`r_{dst}	; *load at offset*
		; *load delay here?*
use r_{dst} *as operand*		

where "`lui`" is a *load immediate upper* instruction, and where *hibits'* and *lobits'* are attribute functions which extract the relevant bits from the 32-bit address at link time.

As detailed in the later section on instruction scheduling, there is a delay after the issuing of a load instruction, before the operand becomes available for use by later instructions. For many *RISC* machines this delay is a single cycle. In some cases the compiler is able to find a useful instruction to schedule in this delay slot, but this is not always possible. In this typical case keeping the value in a register, rather than redundantly reloading it, will save as much as three machine cycles.

7.1.3 Common sub-expression elimination

Common sub-expression elimination (CSE) is the optimization which finds parts of computations which are repeated, and retains the value from the first computation for subsequent reuse. There are two variants on this transformation. Local-CSE elimination attempts to find reuses of the same sub-expression within the same (extended) basic block, while global-CSE elimination attempts to find reuses of sub-expressions within the same procedure.

In either case, there are two issues to be considered. Is a previous computation of the same expression *available*, and is the previous value still *valid*. By being available, we mean that some other occurrence of the same computation lies along *every* control path which leads to the new occurrence. This is easy to ensure for local CSEs. Recall the defining property of an extended basis block is that all preceding instructions in a block are executed before control reaches any particular instruction.

Validity is the property that guarantees that if the computation were to be repeated then the same value would be produced. Suppose that we have some computation, say $(a + b)$. If we have some later occurrence of $(a + b)$ the CSE is valid only if we can be absolutely certain that neither the value of a nor the value of b could have been modified by the intervening instructions. This may be quite difficult to certify if, for example, either a or b is a global variable, and there is a procedure call between the two computations.

7.1.4 Value tracking

Value tracking is another optimization which is often done together with CSE-elimination. As for CSE-elimination, there are both local and global versions of this optmization.

Value tracking retains expressions which have been assigned to variables, and substitutes the retained values for references to the destination variable in subsequent code.

Note the important difference between these two cases —

$\cdots(a+b)\cdots$	`v :=` *expression*
intervening statements...	*intervening statements...*
$\cdots(a+b)\cdots$	$\cdots$`v`$\cdots$

in the left hand (CSE) case, the subexpression $(a+b)$ is repeated, and if the value is still valid the result of the first computation may be reused. In the right hand (value tracking) case, the *expression* is not repeated, but the variable to which the expression was assigned appears later. In this second case we retain the value of the assigned expression, so that it may be substituted for occurrences of the variable.

With both value tracking and common sub-expression elimination the question of *liveness* of the saved values is an issue. Note that any assignment (or even threat of possible assignment) to a or b in the intervening instructions will kill the common sub-expression in the example above. By constrast, assignment to one of the operands of the expression in the value tracking example does not kill the value. Only assignment (or threatening) of the destination variable v kills the saved value in the value tracking example.

7.1.5 Lowering register pressure

An important aspect of many optimizations is that redundant computation, and redundant references to memory are removed. In order to do this, the lifetimes of various values held in machine registers is lengthened. All such algorithms *raise register pressure*, by requiring a greater supply of registers at critical points in the program. In many cases, the advantage which can be obtained by the use of optimization is limited by the supply of registers which are available to hold useful values. Thus it is important to prevent unnecessary register usage.

As will be noted in a later chapter, optimal register allocation is a computationally infeasible problem. Nevertheless, there are a number of important techniques which lower register pressure, at moderate computational cost. The first of these is the Sethi-Ullman algorithm.

The Sethi-Ullman algorithm

The Sethi-Ullman algorithm demonstrates an optimal ordering for expression tree evaluation, so as to use the minimal number of registers. It is important in its own right, but even more so for the **most complicated first** principle which it epitomizes.

During expression evaluation subexpressions are evaluated into registers, which are then used in the computation of other subexpressions. The important metric which we wish to minimize is the "peak" number of registers which are used during the evaluation of a particular expression. As we shall see, the problem has an elegant recursive solution.

Let us presume, as a first example, that a particular machine has a load/store architecture. Thus values must be loaded into registers before being used as operands in any computational instructions. The peak number of registers required to compute the value of a tree consisting of a single leaf is one.

Now let us consider the general case. A binary operand has two sub-trees, and we shall presume without loss of generality that the peak number of registers required to compute the left and right subtrees is n and m respectively (figure 7.2).

Let us suppose that we compute the left subexpression first. During this evaluation we use a maximum of n registers. When we have completed the evaluation, the result is held in a single register, and occupies that register while the right subexpression is evaluated. Computation of the right subexpression, by assumption, requires an additional m registers. It follows that the peak number of registers required for this evaluation is $max(n, m + 1)$ where the max function returns the larger of its two arguments. By symmetry, if the other possible order of evaluation is chosen the peak number of registers required is $max(n + 1, m)$. Thus, we conclude the following: if the number of registers required for the two subexpressions is the same, say n, then the number of registers required for the complete expression is $n + 1$. On the other hand if n and m are different, and

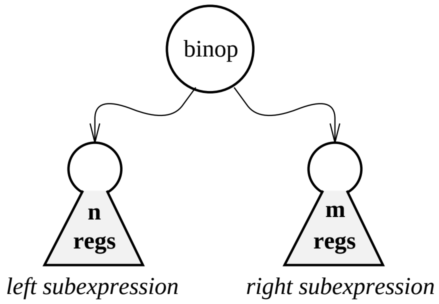


Figure 7.2: A binary expression tree

the more demanding of the two expressions is evaluated first, then the number required for the complete expression is just $max(n, m)$.

```
PROCEDURE Number(node : ExprDesc; VAR num : CARDINAL);
  VAR lNum,rNum : CARDINAL;
BEGIN
  IF exp^.tag IN binops THEN
    Number(exp^.leftOp,lNum);
    Number(exp^.rightOp,rNum);
    IF      lNum > rNum THEN num := lNum;
    ELSIF   rNum > lNum THEN num := rNum;
    ELSE (* lNum = rNum *)   num := lNum + 1;
    END;
  ELSIF exp^.tag IN unarys THEN
    Number(exp^.childOp,num);
  ELSIF exp^.tag = literal THEN
    num := 0;
  ELSE (* assert: tag = designator *)
    NumberDesignator(exp,num);
  END;
  exp^.sethiNum := num;
END Number;
```

Figure 7.3: Calculating Sethi numbers for a load/store architecture

We may thus evaluate the minimal number of registers required for evaluation of a particular expression tree during a recursive traversal. The traversal annotates each expression with the minimal number of registers required for its evaluation. This is called the *Sethi number* of the node. Figure 7.3 gives the code of a numbering procedure. The calculation of the numbers for designators, which may involve recursive numbering of index expressions, is abstracted away and left as an exercise.

Machine architectures where one of the operands of a binary operation may be taken from memory need a slightly modified algorithm to compute the Sethi numbers of leaf nodes. In the given code, these details would be hidden in the *NumberDesignator* call. It may also be noted that we are assuming, at line 14, that literals do not need to be loaded into registers. For most *RISC* machines this is only correct for certain uses, and for sufficiently small literals.

The code emission traversal of a numbered tree is a modfied version of figure 4.2, and is outlined in figure 7.4.

```
PROCEDURE PushValue(exp : ExprDesc);
BEGIN
  IF exp^.tag IN binops THEN
    IF exp^.rightOp^.sethiNum > exp^.leftOp^.sethiNum THEN
      PushValue(exp^.rightOp);
      PushValue(exp^.leftOp);
      EmitInstruction(swap2);
    ELSE
      PushValue(exp^.leftOp);
      PushValue(exp^.rightOp);
    END;
    EmitInstruction(tagToOpCodeMap[exp^.tag]);
  ELSIF exp^.tag IN unarys THEN
    ...
```

Figure 7.4: traversal procedure for numbered expressions

7.1.6 Instruction scheduling

Modern computer architectures rely on instruction level parallelism to produce additional speed. This overlapping of instruction execution arises from two separate mechanisms. Firstly, the use of instruction execution pipelines is universal, with pipeline depths of four or five being common in entry level implementations of *RISC* architectures. The use of an execution pipeline allows the time between the issue of adjacent instructions to be much less than the time taken to actually complete each instruction. Secondly, modern superscalar[2] microprocessors rely on the existence of multiple execution units to allow *multiple instruction issue*. Issue widths of two and three are now common, and greater issue widths are likely within a few years.

The existence of instruction pipelines requires some reordering of the logical instruction stream, if advantage of the potential for execution overlap is to be

[2]Machines which are theoretically capable of executing more than one instruction per clock cycle from a single instruction and data stream are called *superscalar* machines. This is to distinguish such machines from *vector* machines which obtain instruction level parallelism by operating on parallel "vector" data.

realized. The problems which have to be solved are the dependencies between instructions, and resource conflicts.

If one instruction is issued, and the next instruction has to wait before it can be issued, then we say that we have an *instruction isssue stall*, or a *bubble* in the pipeline. Such a bubble causes a direct loss of performance,since it represents a machine cycle during which no new work is started. The aim of instruction scheduling is to reorder instructions so that the machine cycle wastage is minimized, but the semantics of the program are unaltered.

The need for such instruction scheduling arose first in the context of *RISC* machines, but is now universal. Even apparently *CISC* machines such as the *Intel-486* have internal mechanisms which are pipelined. In the case of the Intel machine this exhibits itself in such phenomena as a one cycle delay in the execution if the result of an instruction is used on the very next instruction as an index or base address register.

Instruction dependencies

There are three kinds of logical dependencies which two instructions may exhibit. These are *read-after-write, write-after-write* and *write-after-read* dependencies.

Suppose a *VAL* sequence adds two registers and compares the result with a literal. The instructions might be —

`add`	v_1,v_2,v_{dst}
`cmp`	v_{dst},7

The problem is that the compare instruction cannot start its comparison, until the result is available from the previous instruction. This is a *read-after-write* dependency, since the result must apparently be written to the register file before the next instruction can read it out and use it. In fact, due to a clever hardware design trick called *result forwarding*, such delays are almost eliminated in many machines. However, the *read-after-write* delay after a load from memory is typically one or two cycles.

If two instructions both write to the same register, then the instructions must be issued in such an order that their results are written to the register in the correct order. Otherwise, the final register content would be incorrect. This is an example of a *write-after-write* dependency, it is a constraint on the way in which instructions may be reordered.

If one instruction reads from a register, and another instruction writes to the same register, then the second instruction cannot be issued if the result would be written to the register before the first instruction has finished reading the value. This is an example of a *write-after-read* dependency, it is a constraint on the way in which instructions may be reordered.

Delayed branching

Many *RISC* machines have instruction pipelines in which the execution of branch instructions takes so long that the execution of the folowing instruction is already irrevocably started. Such a following instruction is said to be in the *delay slot* of the branch. It is always safe to place a nul operation "`nop`" instruction in the branch delay slot, but it is possible to use instruction scheduling to try to eliminate this waste of a cycle.

If an instruction can be found which is always useful, and may be moved to the delay slot without conflict, then the waste of the delay slot may be eliminated. Some machines provide a facility called *annulled instructions* or *branch squashing* in which a conditional branch may be tagged so that the result of the following instruction is discarded if the branch does not proceed in the predicted direction. Such a facility means that a delay slot can always be made "probabalistically useful" by moving the first instruction of the predicted successor block into the delay slot, even if there is no instruction from the current block which may be safely moved to the delay slot.

Local scheduling

Contemporary instruction scheduling is usually performed on a single basic block. However, some interesting experimental work is being done on global scheduling. Such approaches will become universal as issue widths increase.

A later section deals with the implementation details of a simple scheduler, but it is important to first discuss a fundamental problem. The question is whether scheduling is best carried out on the *VAL*, before register binding, or after register binding when the actual registers are known.

Neither answer to this question is entirely satisfactory. Scheduling of *VAL* maintains maximum flexibility, but has at least two problems. Firstly, in general, scheduling increases the demand for registers, and hence the register pressure. If the increased pressure leads to a register being spilled, the time penalty will almost certainly be greater than the time from the original pipeline bubble. Consider the ordering of the computation of $(a + b + c)$ in a load/store machine, where v_n are virtual registers. Suppose the machine has a single cycle load delay, so that the result of a load is not available until the instruction after the following one.

Sethi-Ullman order	Best scheduled order
`lw` a,v_1	`lw` a,v_1
`lw` b,v_2	`lw` b,v_2
wait for v_2 to be available	`lw` c,v_4
`add` v_1,v_2,v_3	`add` v_1,v_2,v_3
`lw` c,v_4	`add` v_3,v_4,v_5
wait for v_4 to be available	
`add` v_3,v_4,v_5	

The optimal, Sethi-Ullman ordering is shown on the left. It requires only two registers,[3] but has two wasted cycles. The order on the right requires an additional register but has no wasted cycles.

[3]For example, v_2, v_3 and v_5 could all share the same register, and v_1 and v_4 could share the other.

The second problem with scheduling on the *VAL* is register to register copy instructions may be carefully scheduled, and then eliminated by the register allocator if it manages to allocate the same register to both virtual registers. The problem with scheduling after register allocation is that during register binding false dependencies are introduced. Two virtual registers may be logically independent, and the instructions which create them might be able to be executed in either order. However, if the register allocator places both virtual registers in the same physical register, then a *write-after-write* dependency will have been created. Consider the expression evaluation on the left in the previous example. A register allocator might place v_4 in the same physical register as either v_2 or v_1. Once that has been done, it will be impossible to move the load of v_4 above the first add, so as to fill the load delay slot of the load of $v2$.

Situations such as this, in which either order of performing two operations has disadvantages, are said constitute a **phase order problem**. There are two strategies to lessen the problem. One is to use a register allocation method which is aware of scheduling. The other possiblity is to schedule on the *VAL*, but do a scheduling correction after register allocation.

The implementation of instruction scheduling is considered in Chapter 11.

7.1.7 Peephole optimization

One of the noticable aspects of code produced by the methods described in this book is that code selection is purely *local*. This is an inevitable consequence of the fact that code is produced algorithmically during a traversal of some *AST*. Thus we may produce quite good code for each statement, but have poor code in the transitions between statements. Peephole optimization attempts to recognise certain short patterns in the code, and replace these by equivalent, but better sequences.

The name is chosen because often these methods look at a sequence of instructions of some fixed finite length. To take a particular, simple example, consider the code which is produced when a value is made into a temporary in *DCode*. The instruction `mkTmp` makes a copy of the value on the top of the abstract stack, leaving the original value unchanged. Suppose however that we do not need the copy which is left on the stack. In this case we simply emit a `pop1` to clean up the stack. On most machines, for most operands, this is purely a housekeeping adjustment at compile time. But for a value in the floating point coprocessor stack, each of these instructions corresponds to a real runtime operation. The following shows the *DCode*, *VAL* and resulting assembly language in the case that the top-of-stack is a floating point coprocessor stack value.

DCode	*VAL*	*Final Assembly Language Instructions*	
`mkTmp 24`	`fdup`	`ld st(0)`	; *duplicate tos*
	`fstpt fp-24`	`fstpt -24(ebp)`	; *save to temp*
`pop1`	`fpop`	`ffree st(0)`	; *set free and*
		`fincstp`	; *pop fp stack*

The code which translates each *DCode* is accurate, but yet the resulting assembly language is truly dreadful. If the three instruction pattern were to be recognized in the *VAL*, then we could replace the three instructions by a single instruction.

DCode	*VAL*	*Final Assembly Language Instructions*
`mkTmp 24`	`fdup`	
	`fstpt fp-24`	
`pop1`	`fpop`	`fstpt -24(ebp)` ; *save and pop fp stack*

In general the number of patterns which can be translated is quite large, so that it useful to implement peephole optimizers which are automata automatically generated from specifications. The implementation of such automata is considered in some detail later.

7.2 Implementing CSE and value tracking

7.2.1 Converting trees to dags

The essence of common subexpression elimination is to recognize when a computation of some subexpression is redundant because the result is already available. All uses of the redundant subexpression then become references to the previously computed value. The method thus transforms a forest of trees into a forest of directed acyclic graphs (*DAGs*).

It is possible to place CSE elimination in several different places in the compilation process. The tree to *DAG* transformation might, quite literally, be performed on the abstract syntax tree representation of the program, and indeed this is sometimes done. Equally, it is possible to perform the optimization as an *IR* to *IR* transformation. Unfortunately, the decision as to whether a particular common subexpression is worth saving is rather dependent on the machine architecture[4], which argues for performing the elimination as late as possible in the code-generator. Furthermore, the algorithms are probably easiest to understand when applied to a *tuple* representation, such as our typical *VAL* forms.

The transformation is done relatively easily one basic block at a time, but the same elimination may be done globally. The removal of global common subexpressions is treated in a later section, in a manner compatible with the treatment of this section.

We focus therefore on elimination of redundant instructions in the *VAL*.

7.2.2 Value numbering

The value numbering method of Cocke and Schwartz provides a basis for local CSE-elimination in a tuple-based context. We consider a variant of this method which extends naturally to the global case considered later.

[4]For example, architectures with complex address modes can fold quite complex address expressions into the memory address of a single instruction. In such a case, the common address subexpression may not be very valuable. For a *RISC* machine, reusing the same address subexpression might save several instructions.

For each extended basic block, we allocate a *value number* to each logically distinct value mentioned in the block. Whenever an entity is updated explicitly, or its value is threatened by some side-effect, we shall assume that the value has been changed, and allocate a new value number. This is a conservative assumption, and ensures that if two data have the same value number that they will certainly hold the same value at runtime.

In order to determine if the operation represented by a particular binary instruction is redundant we do not ask whether we have performed the same operation on the same two *data*. Rather we must ask whether we have performed the same operation on the same two *values*.

Two types of data with values occur in our *VAL*. These are the virtual registers, and values fetched from memory. If we have generated the *VAL* in such a way that each virtual register is defined in a single place only, then the data structures which we shall need become extremely simple. The extra computations which are required if registers are *mutable* are considered later. For the moment we shall also make the simplifiying assumption that all of memory is a single, structured variable, and that loads from memory simply fetch some component of this single variable. This is an extremely safe and conservative assumption, since it means that initially we shall conclude that whenever *any* write to memory is performed that the whole of the memory "variable" has a new value number.

Value numbering may be thought of as a *symbolic execution* of the basic block. We step through the instructions of the extended block in the forward direction, modifying instructions and updating a *value number table* as we go. There are two associated data structures. The *virtual register equivalence table* maps value numbers to the unique virtual register which first computed the same value in the current block. The *tuple lookup table* determines whether an equivalent instruction has occurred before in the block.

Let N be the domain of value numbers. There is a distinguished value of the domain **nil**, denoting the undefined value. Let V be the domain of virtual registers. There is a distinguished value of the domain, also denoted **nil**, denoting no register. The overloading of the **nil** symbol should cause no difficulty.

Let the set of data in the block be denoted E. The set $E = V \cup \{m\}$, where m is the memory variable, and V is the set of virtual registers. The value number table *numTab*, is a mapping from data to value numbers.

$$numTab : E \to N$$

Let P be the set of operators. Instructions, in general, have an operator $op \in P$, one or more input data (operands) $e_i \in E$, and a result entity $r \in E$. Two instructions are equivalent if they have the same operator, and input operands with the same value numbers. If this is the case, they compute the same value.

The lookup table *hshTab*, determines if a particular instruction is redundant. The table is a mapping from tuples to value numbers.

$$hshTab : P \times N^{+} \to N$$

In the event that a tuple is previously unknown, the *hshTab* function returns **nil**, and we know that the instruction is useful. The result value is issued with a new value number n, the instruction is entered in *hshTab*, and the pair $(r \mapsto n)$ is entered in *numTab* overwriting any previous mapping of r.

On the other hand, if the tuple is already present in the table, we know that the same value m has been computed before. In this case the instruction may be deleted, and the pair $(r \mapsto m)$ is entered in *numTab*. This indicates that future uses of r will be uses of the value number m.

We shall assume that memory has an initial value number 0 on entry to the block, but all virtual registers have value number **nil**.

At the abstract level, the algorithm may be represented as follows —

```
FOR every instruction I in the block DO
   m := hshTab (op,numTab(e1),···);
   IF m = nil THEN
      allocate a new value number n,
      and insert ((op,numTab(e1),···) ↦ n) in hshTab;
   ELSE
      delete I, and insert (r ↦ m) in numTab
   END;
   account for side effects in numTab;
END;
```

The final line, accounting for side-effects, is simple in this case, but is important since it gives the clue as to how the method is extended to less conservative assumptions about memory. In effect, after every write to memory the memory entity is allocated a new value number. Similarly, every procedure and function call allocates a new value number to memory. If the domain E includes any physical (as opposed to virtual) registers, then the effect of instructions which modify physical registers must be taken into account similarly. For most machines this includes caller-saves registers potentially modified by call instructions, and physical registers modified by divide instructions, and so on.

Finally the instructions which remain are rewritten. The defining occurrences of virtual registers which held redundant values have been removed, so we must map used occurrences of these missing registers to uses of the equivalent virtual register. Since we have assumed that virtual registers have a unique definition, we may choose the register number of the first definition of each value as the value number. In this way the value number table becomes a map from virtual registers to equivalent virtual registers.

We shall work through an example, the inner block of a bubble sort program. The source code is —

```
IF a[i] < a[i+1] THEN
  t := a[i+1];
  a[i+1] := a[i];
  a[i] := t;
END;
```

The *VAL*, and the filled in value number table is shown in figure 7.5.

In this figure, the table *numTab* is shown to the right of each instruction. A number in the particular position in the table indicates that the instruction is a

code	**entity name**												**req**
	mem	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	10	v_{11}	
`; line 1`	0												
`lw` i, v_1	x	1											√
`lw` $a(v_1), v_2$	x	x	2										√
`lw` i, v_3	x			1									
`lw` $a+1(v_3), v_4$	x			x	4								√
`bgt` v_2, v_4,lab			x		x								√
`; line 2`													
`lw` i, v_5	x					1	x						
`lw` $a+1(v_5), v_6$	x					x	4						
`sw` v_6,t	12						x						√
`; line 3`													
`lw` i, v_7	x							7					√
`lw` $a(v_7), v_8$	x							x	8				√
`lw` i, v_9	x									7			
`sw` $v_8, a+1(v_9)$	13								x	x			√
`; line 4`													
`lw` t,v_{10}											10		√
`lw` i, v_{11}												11	√
`sw` $v_{10}, a(v_{11})$	14										x	x	√

Figure 7.5: *VAL* and value table for bubble block (#1)

defining occurrence for that value number. The numbers remain valid until they are overwritten by another definition for the same entity. For example, second and fourth instructions both define the same value number by loading the same variable i. The second of these loads is clearly redundant.

An 'x' in a particular column of the value table indicates that the instruction has a *used occurrence* of the current value for the entity of that column. A "tick" $\surd$ in the final column indicates that the instruction is useful and thus is **req**uired. In this case exactly 11 of the original 15 instructions turn out to be useful. Three of the four instructions which are eliminated are redundant reloads of the variable i.

In this example, the effect of the simplifying assumptions on memory are all too apparent. After the first assignment to t the previous values for both a and i are killed, and must be reloaded for their next use. If we are able to distinguish between these three global variables a, i, t, then we obtain the figure 7.6.

After remapping the remaining instructions, in this more careful case we obtain the following result, with only 8 of the original 14 instructions remaining —

```
; line 1
    lw  _i,v1              ;  old i1
    lw  _a(v1),v2          ;  old i2
    lw  _a+1(v1),v4        ;  old i4
    bgt v2,v4,xLab         ;  old i5
; line 2
    sw  v4,_t              ;  old i8
; line 3
    sw  v3,_a+1(v1)        ;  old i12
; line 4
    lw  _t,v10             ;  old i14
    sw  v10,_a(v1)         ;  old i15
xLab:
```

code	**entity name**														**req.**
	a	i	t	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	
`; line 1`	0	12	13												
`lw` i, v_1		x		1											√
`lw` $a(v_1), v_2$	x			x	2										√
`lw` i, v_3		x				1									
`lw` $a + 1(v_3), v_4$	x					x	4								√
`bgt` v_2, v_4,lab					x		x								√
`; line 2`															
`lw` i, v_5		x						1	x						
`lw` $a + 1(v_5), v_6$	x							x	4						
`sw` v_6,t			14						x						√
`; line 3`															
`lw` i, v_7		x								1					
`lw` $a(v_7), v_8$	x									x	2				
`lw` i, v_9		x										1			
`sw` $v_8, a + 1(v_9)$	15										x	x			√
`; line 4`															
`lw` t,v_{10}			x										10		√
`lw` i, v_{11}		x												1	
`sw` $v_{10}, a(v_{11})$	16												x	x	√

Figure 7.6: *VAL* and value table for bubble block (#2)

It may be noticed that even this version still has a redundancy. The reloading of the temporary value t at old instruction i14 is surely unneeded, since the value is still in register v_4. To be able to recognize this fact requires the implementation of value tracking, as well as common subexpression elimination.

Value tracking may be added almost trivially, while retaining the same data structures. Whenever a store instruction is symbolically executed, we simply add a dummy tuple to the *hshTab* map. The dummy tuple is a load operation on the same selected element of memory to the source virtual register. Then whenever a load from that location is met, it will be found to be redundant by the normal mechanism. In the example above, the dummy tuple would be a load of t to location v_4. Thus instruction i14 would become redundant, and the store at instruction i15 would be of v_4.

Mutable register values

Some modification needs to be made to the scheme of the previous section, if register values are allowed to have mutable values. It was a guiding principle of *VAL*-creation that any virtual register had a single definition only.[5] Mutable register values occur for two reasons. Firstly, physical registers may occur explicitly in code, as parameters, return values, or as operands of instructions which take fixed registers. Also, if particular variables are allocated to registers, then these registers may have multiple definition points.

Having mutable registers invalidates one of the key assumptions of the algorithm given earlier. For example, consider the following *VAL* fragment, in which p_1 is a register variable.

[5] Note, by the way, that we do not actually demand that a virtual register have a single value. Rather we demand that a virtual register has a single point at which it is defined. A virtual register which has its single definition inside a loop will be assigned a new value at runtime each time the loop is traversed. Nevertheless, the register has a single definition point, and has a single value number in this analysis.

```
; line 1
     div v1,v2,p1          ;  result in regvar
     ...
     lw _i,p1              ;  new value for p1
     ...
     div v1,v2,v3          ;  redundant ???
     ...
     v3 is used here ...
```

We assume that v_1 and v_2 are conventional virtual registers, with a single point of definition, and hence are not modified between the first and last instructions of this fragment. Our data structures will correctly tell us that the second division will result in a previously computed value. However we cannot simply replace later uses of v_3 by p_1, since p_1 no longer has the same value number.

In response we might simply accept that the instruction is not redundant and recompute the value. This would be rather unfortunate, since the divide instruction is expensive. Alternatively, with some care we can improve on this situation. Suppose we adopt the strict convention that no computation directly deposits a result in a mutable register. Instead we always compute the value into a single assignment register, then copy *that* register to the mutable register. Our example then becomes —

```
; line 1
     div  v1,v2,v4         ;  move result to regvar
     move v4,p1            ;  but indirectly !
     ...
     lw _i,v5              ;  new value for p1
     move v5,p1            ;  but indirectly !
     ...
     div v1,v2,v3          ;  redundant ???
     ...
     v3 is used here ...
```

Our algorithm will now tell us that the second division is redundant, and later uses of v_3 can safely be replaced by v_4. The register allocator will correctly determine that v_4 and p_1 must be allocated different registers, since p_1 is redefined while v_4 is still live. However the second copy will be eliminated by allocating v_5 and p_1 to the same register. The final result is that we have eliminated the redundant division and replaced it by a value-preserving register-to-register copy. We may even have lowered the register pressure locally by shortening the live ranges of v_1 and v_2.

If we can guarantee that the *VAL* has this property, we can leave the previous algorithm almost unchanged. We must take account of the fact that mutable registers, like memory, have an initial value on entry to the block. Further, we must never eliminate a copy to a mutable register, since the value may be needed outside of the current block. Instead we leave the register allocator to eliminate the redundant copies later.

Implementing the data structures

The data structures which we require in order to perform local common subexpression elimination and value tracking are the value number table *numTab*, and the tuple mapping table *hshTab*. We do not need a table for mapping value numbers to equivalent virtual registers, provided we number the first definition of each register value so that it is equal to the virtual register index.

The value number table may be conveniently implemented as a one-dimensional array, using some unique denotation of the entity as index. The tuple lookup table is more difficult however, since it maps to the value number from a tuple, the arity of which varies for instructions with different format. For example, a unary operation has an operator plus one input operand, while a binary operation has an operator plus two input operands. It is normal to implement this table as a *hash function* as the name already hinted. For each different instruction format we must have a hash function which maps the used fields of the instruction format into a hash table index.

This structure may be elegantly implemented using an array of procedure variables, indexed on instruction format-type.

7.3 Implementing peephole optimization

`<still to come>`

7.4 Finding out more

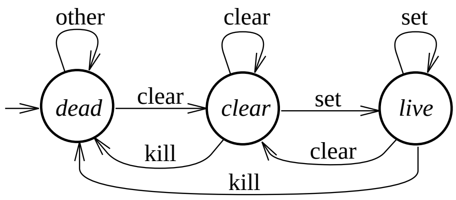
`<< various pointers to the literature >>`

7.5 Exercises

7.1

7.2

7.3 The IBM *Power* architecture has a "sticky" overflow flag which once set remains set until it is tested, or some instruction is executed which destroys the contents of the flag register. For our purposes, there are four kinds of instructions. There are those that leave the contents of the flag unchanged. There are those that conditionally *set* the flag. There are those that *kill* the flag, destroying its contents. Finally there are those that *clear* the flag. At the individual instruction level the following *FSA* applies —



Within a basic block, we may try to economize the number of overflow tests by only testing when it is necessary to do so. Work out how to do this, using the convention that at the start of each basic block the state is *dead*. Assume the existence of an instruction "`clear`" which clears the flag, and another "`test`" which traps if the flag is set, and clears the flag as a side-effect.

Chapter 8

Dataflow Analysis

8.1 Creating the control flow graph

By dataflow analysis we mean all of the computations which are performed on the **control flow graph** (*CFG*) of a procedure or function. A control flow graph represents the control flow in a procedure, it is a directed graph, and in general is cyclic. Each **node** of a *CFG* denotes a single basic block, as defined in the previous chapter. There is an directed **edge** in the *CFG* between two nodes, if the second represents a basic block which is a possible control flow successor of the basic block which the first represents. There is a distinguished *start* node at which all control flows enter the graph, and we shall also arrange that all control flows which leave the graph pass through a single *end* node. The existence of these two distinguished nodes says nothing at all about whether or not the actual code has one or many entry and exit points.

8.1.1 Definitions

Formally then, a *CFG* is a quad (N, E, s, e), where N is the set of nodes, E is the set of edges, and $s \in N$ and $e \in N$ are the start and end nodes respectively. The edges are pairs of nodes which are adjacent in the graph, so we may treat E as a relation on pairs of nodes. We say $(a, b) \in E$ if and only if there is an edge (directly) connecting node a to node b. We also denote this relation $a \to b$, meaning that there is an edge (directly) from node a to node b.

We define two sets for each node in N.

$$\forall a \in N, pred(a) = \{b : b \to a\}$$

$$\forall a \in N, succ(a) = \{b : a \to b\}$$

These two sets are the control flow *predecessors* and *successors* of the given node. Node s (probably alone) has an empty *pred* set, while node e has an empty *succ* set. The cardinality of the two sets are the *in-degree* and *out-degree* of a particular node.

$$\forall a \in N, \textit{in-degree}(a) = |pred(a)|$$

$$\forall a \in N, \textit{out-degree}(a) = |succ(a)|$$

For any two nodes a, b we say that b is *reachable* from a if there is a directed path from a to b. Reachability is the transitive closure of the $\rightarrow$ relation. Formally, we write

$$a \rightarrow^{+} b \equiv a \rightarrow b \vee \exists c \in N : a \rightarrow c \wedge c \rightarrow^{+} b$$

$$a \rightarrow^{*} b \equiv a = b \vee \exists c \in N : a \rightarrow c \wedge c \rightarrow^{*} b$$

The last is the reflexive, transitive closure of the $\rightarrow$ relation.

A stronger notion than being a predecessor of a particular node is that of *dominators*. The dominators of a particular node are all those nodes which occur on *every* path from s to the chosen node. This is an important set, since we know that the code in any such basic block must have been executed at least once before control reaches our node of interest.

Such definitions, stated in terms of the *non-existence* of paths with certain properties, are problematical. In this case we have demanded that there be no path from *start* to the node n which does not pass through every $k \in dom(n)$. We need to find an inductive definition which is equivalent.

As an insight, consider the relation between the dominators of nodes b and a if $b \rightarrow a$. If b is the only predecessor of a, then b is a dominator of a, and every node which dominates b also dominates a. Suppose now that b and c are the only two predecessors of node a. The paths which reach a through b have passed, by assumption, through all dominators of b along the path, and similarly all paths through c have passed through all the dominators of c. Clearly, the dominators of a must be the intersection of the dominators of a and b. Generalizing to any number of predecessors, we obtain the recursive definition —

$$\forall a \in N, dom(a) = \bigcap_{b \in pred(a)} (dom(b) \cup b)$$

This is an interesting equation, being a set of simultaneous, recursive equations on set-valued quantities. Several questions immediately arise: how is one to solve such a set of equations; **is** there always a solution; is there **more than one** solution?

As it turns out, almost all the equations in this chapter are of this kind. The answer to the questions are: there are at least two ways to compute the solution to such equations, there always is a solution, and yes, unfortunately there are many solutions. In the next section we consider one of the methods of solution, but first let us explore why there is more than one solution.

Firstly, note that if we substitute $\forall b, dom(b) = N$ into the equations, where N is the set of all nodes, then they are trivially satisfied. However, this implies that every node dominates every other node, which does not satisfy our specification, even if it does satisfy the equations. In this particular case, what we need is the *least* solution in the sense of having the sets with the smallest possible number of elements which satisfy the equations. This is the *least fixed point* of the equations, in terms of the theory of functions.

Since these equations are written with the values at each node depending on information from the predecessor nodes, we call this a **forward flow** problem. We shall meet other dataflow problems later, in which the values at nodes depend on information from the successor nodes. We shall call this second kind **backward flow** problems.

Iterative solutions to set equations

One method of solution for equations of the kind given above, is iteration. In this method, we make an initial "guess" at the solution and iterate by substitution of the approximation on the right hand side of the equations in order to converge toward a solution. Since we want a *least* solution we may safely start with empty sets as a first approximation.[1]

Since we deal with finite sets, and can only *add* extra elements to the approximations at each iteration, it is plausible to argue that the process will converge in a finite number of steps. In fact, for certain *CFG*s we can see that the solution can be found in just one iteration.

Suppose that when we wish to calculate $dom(a)$ we already have the correct dom set of every predecessor of a. In that case we can simply substitute the correct values on the right, and have the answer in one step. So the question then is, can we find a way of visiting the nodes so that when we reach a node we have already reached all of its predecessors?

For acyclic graphs, there is an order of traversal of the *CFG* which has this property. It is called **reverse post-order**. As it turns out this is a good order of traversal, for all forward flow problems, and not just for the case of acyclic graphs.

Reverse post-order traversal

The property that we require of reverse post-order traversal, is that every predecessor of a node is processed before the node is processed.

There are two kind of recursive traversal familiar to every computer scientist. These are pre-order, which has the property that a node is visited before any successor is visited, and depth-first post-order in which every successor is recursively processed before the node is processed.

This depth first post-order is exactly the opposite of what is needed, so we may use a depth-first post-order traversal to number the nodes, or more generally to place the nodes in a list. Figure 8.2 is an algorithm which places the nodes of a graph on a list in reverse post-order. The procedure is called initially with the list empty, and with the actual parameter being the start node s.

Immediate dominators and the dominator tree

Of all the nodes in the dominator set of a particular node there is an unique node which is *nearest* to the node. We call this nearest dominator node the

[1] Ok, let us play safe. Go and check that all sets empty is not a trivial solution to the set of equations!

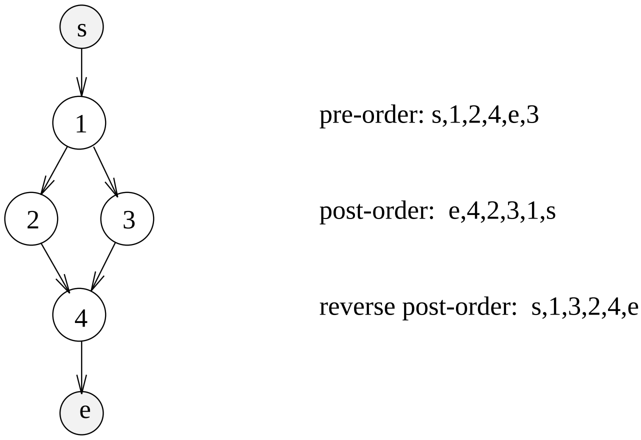


Figure 8.1: Example *CFG*, showing pre- post- and reverse-post-order

```
VAR revList : BlockSequence;

PROCEDURE ReversePostOrder(node : BBlock);
  VAR successor : BBlock;
BEGIN
  mark this node as visited;
  FOR every unvisited successor of node DO
    ReversePostOrder(successor);
  END;
  link this node to left hand of revList;
END ReversePostOrder;
```

Figure 8.2: Building a reverse post-order node list

immediate dominator of the particular node. All the other elements in the dominator set of the original node will be in the dominator set of the immediate dominator. The immediate dominator of a node a is denoted $idom(a)$.

We may arrange for the nodes of any graph to be arranged in a tree, with the start node as the root, such that the parent of each node in the tree is the immediate dominator of that node. This structure is called the **dominator tree** of the graph.

As it turns out, we need the dominator tree more often than the dominator sets, and we may construct the tree directly without first forming the sets. The algorithm depends on having the nodes in a list (as one might have expected) in reverse post order. Figure 8.3 is a typical worklist algorithm.[2] The first *while* loop in the algorithm places each new node in the graph in the correct postion *with respect to those predecessors which have already been placed*. If the

[2] The algorithm in figure 8.3 has complexity $O(N \log N)$. There is a more involved algorithm which is of almost linear complexity.

```
PROCEDURE BuildDominatorTree();
  VAR fixList :  BlockSequence;
  VAR element :  BBlock
BEGIN
  fixList := empty;
  remove first revList node and make root of tree;
  WHILE revList is not empty DO
    remove leftmost element from revList;
    IF not all predecessors in tree THEN
      place element in fixList;
    END;
    add element to tree as child of lowest common ancestor
            of those predecessors which are in the tree already;
    END;
  END;
  (* now we must do corrections *)
  WHILE fixList is not empty DO
    remove leftmost element from fixList;
    recompute position of element;
    IF node was moved THEN
      add all children of element to fixList;
    END;
  END;
END BuildDominatorTree;
```

Figure 8.3: Building a reverse post-order node list

CFG is acyclic, then the reverse post order processing will ensure that all the predecessors are placed, and the result will be immediately correct.

When nodes are processed for which not all the predecessors have been placed in the graph, the possibility arises that the initial placement will be incorrect. These possibly incorrect nodes are inserted in a fix-up list, so that a later check-and-correct step may be performed. The fix-up list is the initial worklist for the correction loop.

The correction loop at the end of this algorithm recomputes the postion of nodes on the fix-up list. Nodes may need to be moved higher up the tree as a result of the recomputation. When a node is moved, the nodes which are dominated by the moved node are placed on the fix-up worklist, so that they may also have their postions checked. However, the only cases where nodes are moved by the correction arise from so-called *irreducible CFG*s. In short, such graphs arise as a result of loops which have multiple entry points, and cannot occur in the case of the graphs which arise from languages without `goto` statements.

Figure 8.4 shows the dominator graph for the same *CFG* as shown in figure 8.1. Note that in this example, when we come to insert node 1 in the graph, of the two predecessor nodes, only only one of the predecessors (node s, but not node 4) is in the tree. Therefore node 1 must be placed on the fixup worklist.

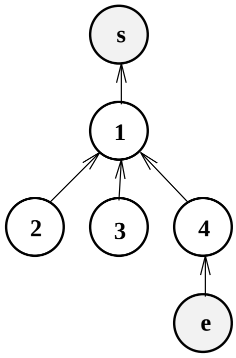


Figure 8.4: Dominator tree for *CFG* in previous figure

When all nodes have been placed in the tree, we check the position of nodes on the worklist. In this case, node 1 is on the list, and the closest shared ancestor of the predecessor nodes, s and 4, is node s. Thus the node does not need to be moved, and we are done.

With the cyclic graphs which arise from structured loops we will always get this same pattern. The *header* node will be inserted into the tree before any other nodes of the loop. Since one predecessor of the loop header is the *latching node* that has the *back-edge* of the loop, not all predecessors are present in the tree. However, since the loop header node of a structured loop dominates all nodes in the loop, including the latching node, the *idom* node computed without taking into account the back-edge is always correct.

If a loop has two or more entry points, then no “header node” dominates all the nodes of the loop, and the recomputation of *idom* for the header node may move the node to a new position in the tree. In any case, if a code generator is to be language independent then it must be able to cope with the irreducible control flow graphs which arise from ill-advised use of the `goto` statement.[3]

There is another reason to have the capability to deal with irreducible graphs. For some purposes we may wish to compute the **immediate post-dominators** of a *CFG*. A post-dominator of a given node is a node with the property that all paths through the given node pass through the post-dominator to reach e. This is exactly the immediate dominators of the *CFG* with the direction of flow reversed. Since reducibility is not invariant under flow direction reversal, we may expect even the graphs from structured languages to have irreducible reverse-flow *CFG*s. Indeed, every loop with a multiple exit in the *CFG* will have multiple entries in the reverse-flow *CFG* and hence make the graph irreducible.

Finding loops

For a number of applications we would like to find all those blocks of a *CFG* which belong in a particular loop in a program. This seems like a reasonably

[3]It should be noted that the restriction on loop structure required for reducibility is on the number of *entry points* to the loop, and does not prevent loop header nodes from having multiple non-loop predecessors.

trivial task, but turns out to be surprisingly complicated.

We shall assume, in this section that the *CFG* is reducible, so that each loop has an unique header node. These header nodes may be found during the construction of the reverse post-order list, since they are the nodes which are the target of **back-edges**. A back-edge is detected by the fact that it leads back to a node which is already visited, but not yet listed.

Suppose that h is known to be a header node. As a first guess, we might assume the nodes in the loop headed by h are all those nodes in the set —

$$\{i \in N : h \rightarrow^* i \wedge i \rightarrow^* h\} \qquad \texttt{(* beware, incorrect! *)}$$

That is, the set of all those nodes which may be reached from the loop header node, and which have a path which reaches back to the loop header node.

The problem with this set is that it includes too many nodes. For example, in the figure 8.5 *all* the numbered nodes can be reached from node 2, and all can reach node 2.[4] A method of separating the nodes of a graph into subsets

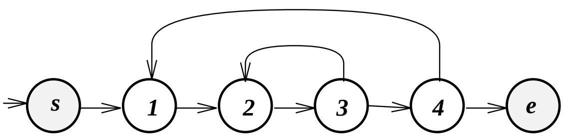


Figure 8.5: Nested loops in *CFG*

which include at most one loop is *interval analysis*. For technical reasons, we begin by adding a dummy edge directly from s to e. Interval sets are constructed iteratively as follows —

```
begin with an interval consisting of the start node;
WHILE nodes of the current interval have unvisited successors DO
   choose an unvisited successor;
   IF successor has another predecessor not in the interval THEN
      start a new interval at successor;
   ELSE
      add successor to the interval;
   END;
END;
```

If we apply this process to the *CFG* in figure 8.5 we obtain the following intervals — $\{s, e\}, \{1\}, \{2, 3, 4\}$. Note that the interval starting at the header node 2 contains only the inner loop.

We now "collapse" all of the nodes in each interval into a single, compound node, which has all of the in-edges and out-edges of its component nodes. In the example, we shall create a new node from the third interval $\{2, 3, 4\}$, and call the new node "$(2, 3, 4)$". The reduced *CFG* will have new new intervals, which in

[4] Indeed all numbered nodes can be reached from every other numbered node. All these nodes form a single **strongly connected component**.

this case will be — $\{s, e\}, \{1, (2, 3, 4)\}$. The second interval now contains both loops, and after a further reduction step we are left with a single interval — $\{s, e, (1, (2, 3, 4))\}$.

The possibility of carrying out this process until the whole graph collapses into a single interval is the origin of the concept of "reducibility".

Interval theory is important as the basis of the second, non-iterative method of solving dataflow equations, but we shall not pursue that use here. Instead we note that if we work from the most nested loop out we may use intervals rooted at loop header nodes to find exactly those nodes in the loop.

Having found an interval containing an innermost loop, we discard nodes for which *all* successors are outside the interval. Note that discarding one node may lead to the later discarding of another node in the interval. When we are through, the remaining nodes will all have at least one edge which leads (either through a forward or a backward-edge) to another node in the interval. These are exactly the nodes within the loop. It is left to the exercises to show that this method works even for strange loops with multiple back-edges, provided the loop has an unique header node.

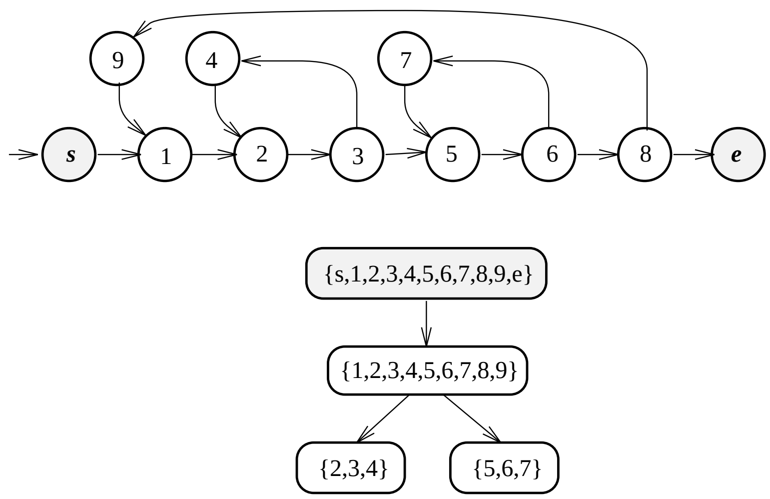


Figure 8.6: Control flow graph and corresponding loop tree

If a graph is reducible, then loops must be properly nested. If this is the case, we may construct a *loop tree*. In this structure, each node is the set of nodes belonging to a particular loop, except for the root node, which contains all nodes in the graph. The parent of any particular loop node is the node which contains all non-loop predecessors of the loop. Figure 8.6 is an example. In this case, the nodes have been numbered in reverse post order, so that the lowest numbered node in each set is necessarily the header node.

8.1.2 Building the *CFG*

We shall suppose that the data structure which represents the *CFG* is constructed either during the filling of the *VAL* buffer, or thereafter. Of course it is possible to build the graph at other phases of the compilation process. For structured languages, the abstract syntax tree of the procedure itself has information which is equivalent to the *CFG*.

Nevertheless, since most of the manipulations for which the graph is required are performed on the *VAL*, we shall perservere with our supposition. We now have a reasonably complete view of the abstract syntax of *CFG*s. Each node represents a basic block, and requires information sufficient to allow predecessors, successors, dominator tree parent and perhaps post-dominator tree parent blocks to be found. As well, we will need some information indicating the extent of the basic block in the *VAL* buffer.

In practice the *CFG* representation will have fields other than the structural information listed here. This additional information corresponds to the *per-block* information which is needed for the analyses which we wish to perform. As an indicative example, the information required in order to perform the best methods of register allocation is, for each block, three sets of virtual registers, these are the *liveIn, liveOut* and the *defined* sets.

The essential subsidiary data structure, used in the construction of the graph, is a lookup table for label names. We may load up the table during creation of the *VAL*, and make an extra pass over the buffer filling in the edges of the graph. Alternatively, we may build the graph on the fly as the buffer is filled, doing a moderate amount of back-patching as the destination labels of forward jumps are discovered. In either case it is advantageous to know which branch targets are trap code entry points, since we would like to ignore such branches in the structure of the graph.

8.2 Forward and backward flow

Dataflow equations occur in four different flavors. They are all sets of simultaneous, equations on set-valued variables, which include a union or intersection operation which iterates over either the predecessor or successor nodes of the given node.

Thus we have forward or reverse information flow on the *CFG*, depending as information flows into the right hand side of the equations from predecessors or successors respectively. Similarly, information merges into the node along *all-paths* in the case of iterated intersections, and *any-path* in the case of iterated unions. The equation for the dominators of a node, repeated here from the previous section

$$\forall a \in N, dom_a = \bigcap_{b \in pred(a)} (dom_b \cup b)$$

is therefore classified as a *forward flow, all paths* problem.

An example of a backward flow, any path problem is given in the chapter on register allocation, where it is used to compute the set of virtual registers which are live at the entry and exit of each basic block.

8.2.1 Downsafe expressions, a backward-flow all paths problem

Here we shall consider the problem of computing the set of expressions which are computed along *every* path which leads from some particular node to the *end* node. We say that such an expression is *down-safe* at that node. The name is chosen because we may be sure that if the evaluation of an expression is moved to a *down-safe* location, then we shall never compute an expression which would not have been computed anyway, perhaps further down the control flow. This is a backward flow, all paths dataflow problem.

An expression is *killed* in a particular block if any operand of the expression has a value assigned to it in that block. An expression is *locally anticipatible* (*locAnt*) in a block if it is computed in that block before it is *killed*.[5] An expression is down-safe at the exit of a block if it is down-safe at the entry of *every* successor block. Finally, an expression is down-safe at the entry of a block either if it is locally anticipatible in the block, or it is down-safe at the output of the block, and is not killed inside the block. Formalizing these observations, we have the following equations —

$$\begin{aligned} downSafeIn_e &= \Phi \\ downSafeIn_i &= locAnt_i \cup (downSafeOut_i - killed_i) \\ downSafeOut_i &= \bigcap_{j \in succ(i)} downSafeIn_j \end{aligned}$$

We may substitute the last equation in the previous one to give the equation in its directly recursive form —

$$\forall i \in N, downSafeIn_i = locAnt_i \cup \bigcap_{j \in succ(i)} downSafeIn_j - killed_i$$

together with the boundary condition that the down-safe set for the end node is the empty set.

Note that the universal set is a trivial solution of these equations so, as in the previous case, the solution which we require is a *least* solution. In this example however, the *downSafeIn* variables are sets of expressions, rather than sets of blocks as in the case of the dominators.

8.2.2 Solving the dataflow equations

The iterative method of solution to dataflow equations works by computing successive approximations to the required solution, starting with some appropriatedly initialized sets.

[5] This is what we might call an *upwardly exposed definition* of the expression. The name comes from the observation that any such expression could just as safely be computed at the entry to the block, rather than at the position at which the computation is originally placed.

It is usual to represent the sets by bit-vectors, so as to gain the speed-up of performing the unions and intersections by word-wide logical *OR* and *AND* instructions. In Modula we would probably use a data structure which is a dynamically allocated array of *BITSET* type. We usually need to allocate the sets dynamically, because the cardinality of the base type is not known in advance.

If we take the backward flow, all paths example from the previous section to illustrate the technique, we will assume that the each basic block descriptor has two variable sized sets declared, the *downSafeIn* and *downSafeOut* sets. During earlier processing, the *locAnt* set is created into the *downSafeIn* set, the *killed* set is computed, and the *downSafeOut* sets are all initialized to empty.

In order to compute the sets we traverse the graph, computing a new value for the *downSafeOut* set from the *downSafeIn* sets of the successor nodes. From the *downSafeOut* set, we compute a new approximation to the *downSafeIn* set. Clearly we wish to visit the nodes in such an order that as far as possible the successor nodes are processed before the the preceding node. Thus we choose to visit the nodes in *post-order*.

```
    VAR done :  BOOLEAN;
        node :  BBlock;
        succ :  BBlock;
  BEGIN
    REPEAT
      done := TRUE;
      FOR every node in post-order DO
        FOR every successor of node DO
          node.downSafeOut :=
                  Intersection(node.downSafeOut,succ.downSafeIn);
        END;
        node.downSafeOut := SetMinus(node.downSafeOut,node.killed);
        IF NOT Subset(node.downSafeIn,node.downSafeOut) THEN
          done := FALSE;
          downSafeIn := SetUnion(node.downSafeIn,node.downSafeOut);
        END;
      END;
    UNTIL done;
  END;
```

This is a simple algorithm which repeatedly traverses the *CFG* in post-order, until a complete traversal is performed without any set being changed. The fact that a set will change as a result of the set union operation is precomputed from the fact that the right hand set is not a subset of the left hand set. This avoids a costly dynamic set copy. The algorithm relies on the existence of a module which can perform the set operations on the dynamically sized sets. Such a module is central to all such dataflow solvers. In fact, a slightly better algorithm than the one shown here can be devised, based on using a worklist to propagate the information from the changed nodes.

8.2.3 Available expressions, a forward flow all paths problem

In the classical method for finding global common subexpressions, an important subproblem is that of finding *available expressions*. An expression is said to be available at a particular node if it is computed along *every* path which leads to that node. This is another example of a forward flow, all paths dataflow problem.

As before, an expression is *killed* in a block if any operand of the expression has a value assigned to it in the block. An expression is *locally deferable* (*locDef*) in a block if it is computed in that block and is not subsequently *killed* in the same block.[6] An expression is available at the entry to a block if it is available at the exit of *every* predecessor block. Finally, an expression is available at the exit of a block either if it is locally deferable in the block, or it is available at the entry to the block, and is not killed inside the block. Formalizing these observations, we have the following equations —

$$\begin{aligned} availableIn_s &= \Phi \\ availableOut_i &= locDef_i \cup (availableIn_i - killed_i) \\ availableIn_i &= \bigcap_{j \in pred(i)} availableOut_j \end{aligned}$$

We may substitute the last equation in the previous one to give the equation in its directly recursive form —

$$\forall i \in N, availableOut_i = locDef_i \cup \bigcap_{j \in pred(i)} availableOut_j - killed_i$$

together with the boundary condition that the available set for the start node is the empty set.

These equations may be solved in ways which are analogous to the method used in the previous section.

Global CSEs, the classical method

We may use the *availableIn* sets to find global common subexpressions. The method is only sketched here, as an alternative method is given in some detail in the following chapter.

The computation of an expression in a block is redundant, and may be replaced by a use of a previously computed value provided two conditions are met. Firstly, the expression must have been computed earlier in the control flow, and secondly, the recomputation of the expression must yield the same value as the previous computation. In section 7.1.3 we referred to these properties as *availability* and *validity*.

[6] This is what we might call an *downwardly exposed definition* of the expression. The name comes from the observation that any such expression could just as safely be computed at the exit to the block, rather than at the position at which the computation is originally placed.

In the previous case, within a single basic block, the problem was straightforward. In the global case, for some expression e in block i, we need to ensure that the e is *available* at the input of block i, that is, $e \in availableIn_i$, and to ensure that e is not killed in block i prior its new definition.

In terms of the attributes defined in the previous sections, an expression evaluation e is redundant in block i if the following holds —

$$e \in availableIn_i \;\wedge\; e \in locAnt_i$$

In the next chapter we shall see a more convenient method of finding global common subexpressions. However, this later method is actually slightly less powerful than the definition given here.

8.3 Finding out more

There are an extremely large number of different optimizations which have been expressed using the dataflow formalism treated here. Many of these are treated in Aho and Ullman[?], and Aho, Sethi and Ullman[?].

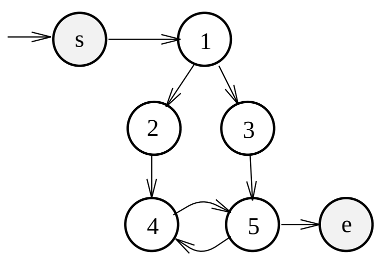
An example of a dataflow problem which does not fall into the pattern described here is *partial redundancy elimination*. The equations involved in solving this problem are *bidirectional*, that is they involve both predecessors and successors. The problem was first formulated by Morel and Renvoise[?]. Recently Knoop, Steffen and Ruthing[?] have published an alternative treatment, which avoids the bidirectional dataflow, but requires computation of four separate dataflow attributes.

The interval method of solving dataflow equations, which is the alternative to the iterative method given here, is due to Francis Allen[?]. An asymptotically more efficient way of constructing the dominator tree is given in Targan and Lengauer[?].

8.4 Exercises

8.1 Execute the algorithm in figure 8.3 by hand, so as to produce the dominator tree of the *CFG* in figure 8.5.

8.2 Here is an example *CFG*. This graph is irreducible.

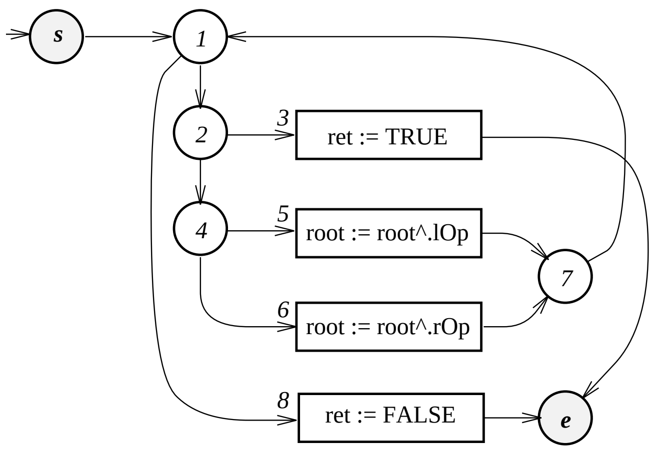


Execute the algorithm in figure 8.3 by hand, so as to produce the dominator tree of the *CFG*.

8.3 Consder the following iterative binary tree procedure —

```
PROCEDURE Search(val : INTEGER; root : Tree) : BOOLEAN;
BEGIN
   WHILE root <> NIL DO
     IF    val = root^.key THEN RETURN TRUE;
     ELSIF val > root^.key THEN root := root^.lOp;
     ELSE (* val < key *) root := root^.rOp;
     END; (* if-stmt *)
   END; (* while *)
   RETURN FALSE;
END Search;
```

A possible *CFG* for this procedure is —



Find the dominator tree and loop tree for this *CFG*.

8.4 An interesting variation on the problem of computing available expressions is that of computing *live* expressions. An expression is live at the entry of some particular *CFG* block, if it is used in that block before it is killed. An expression is also live at entry to some particular block if the expression is not killed in that block, and it is live at the entry of *any* successor block.

Derive the dataflow equations for live expressions.

8.5 Another attribute of expressions is the property of being *busy*. An expression is busy[7] if it is live in at the entry of *every* successor block. The intuition behind computing busy expressions is seen by noting that an expression which is not busy is not used along every path. Therefore it might be better to push the computation down the *CFG* so that it is only computed when it is certain that it is needed.

Derive the dataflow equations for busy expressions.

[7] Also called "very busy" in some accounts.

8.6 The local optimization of overflow tests on the IBM *Power* architecture, suggested on page 207, has a disadvantage in that it may redundantly clear the flag at the start of some basic blocks. Dealing with the propagation of the state values between blocks in a general way leads to a non-standard, bi-directional dataflow problem. However, we obtain a conventional forward-flow problem if we adopt the convention that the state at block entry is only permitted to be *dead* or *clear*.

Derive the equations for this convention. [Note: in this case the propagated value is a single Boolean, rather than a *bit-vector* of Booleans. Thus the equations will have Boolean $\vee$ and $\wedge$ operators, rather than the usual set operations $\cup$ and $\cap$.]

Chapter 9

Static single assignment form

Static single assignment form (*SSA*) is a modern way to solve many of the classical problems of global dataflow analysis. It forms an elegant basis for the implementation of many of the classical optimizations of the past.

The central idea of *SSA* analysis is to create a modified intermediate form for programs, so that each value in each procedure has exactly one position at which it is defined. These single definitions are the global analog of the *value numbers* which guided the computation of local common subexpressions in chapter 7. The key property of *SSA* form, which makes the transformation so powerful, flows from the fact that every (renamed) variable of the transformed program has a single definition point. Thus if two expressions perform the same operation on variables of the same names, then the value computed will be the same. Recall that in trying to determine if a computation was redundant we had to ask two questions — has this computation been already done, and — if the computation was repeated would the same value be produced. *SSA* form makes the second question redundant, since the values of variables are never modified. In this case value-equivalence reduces to a simple test of lexical identity.

The transformation of a procedure into *SSA* form involves some relatively complex computations. However, once the transformation has been performed, many different optimizations can be performed on the modified form. Finally, the procedure must be transformed back into standard form, so that ordinary code emission can take place.

9.1 Transforming into SSA form

Suppose that we have a procedure in which some variable, a say, has more than one definition. Figure 9.1 is a fragment of such a program. We want to modify this program so that each variable has only a single point of definition. We do this by renaming variables so that each definition of the variable a is given a different name. We might call these new, separated variables a_1, a_2 and so

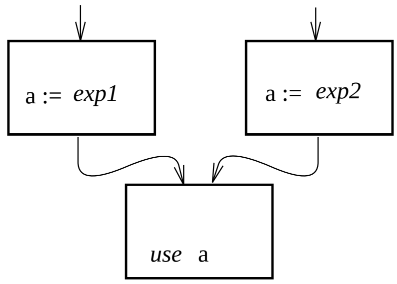


Figure 9.1: Fragment of program in conventional form

on. We will replace defining occurrences of a by a new variable with an unique, indexed name. Used occurrences of a will be modified to refer to the unique value which reaches that particular use.

However, figure 9.1 demonstrates an immediate problem with this concept. Where the control flow merges there is a use of a which two separate definitions reach. Thus we cannot decide which of the two indexed variables to enter as the indexed name of the use in the merge block.

The solution to this conundrum, and the key to the whole *SSA* technique, is to introduce a synthetic assignment at the entry of each such block. These Φ functions[1] assign to a new, unique version of the underlying variable. The arity of the function is exactly the in-degree of the block in the *CFG*, and the arguments of the function are the instances of the value which reach the block along each of the incoming paths. Figure 9.2 shows the transformed version of the same program fragment. The semantics of the Φ function are that the

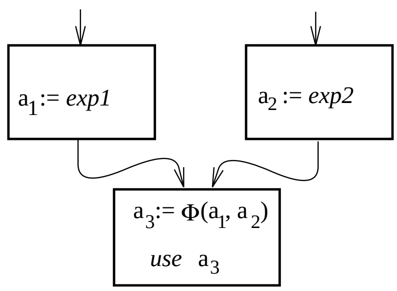


Figure 9.2: Fragment of program in SSA form showing Φ function

value assigned to the variable (a_3 in our example) depends on the path by which control enters the merging node. If control enters through the n-th predecessor, then the n-th argument is assigned to the variable. In our example, if control enters from the left hand predecessor then a_1 is assigned to a_3, while if control enters from the right, then a_2 is assigned to a_3.

[1]Variously spelled in full as "phi" or "fi", and confusingly enough pronounced "fee" in American English, and "fy" in British English.

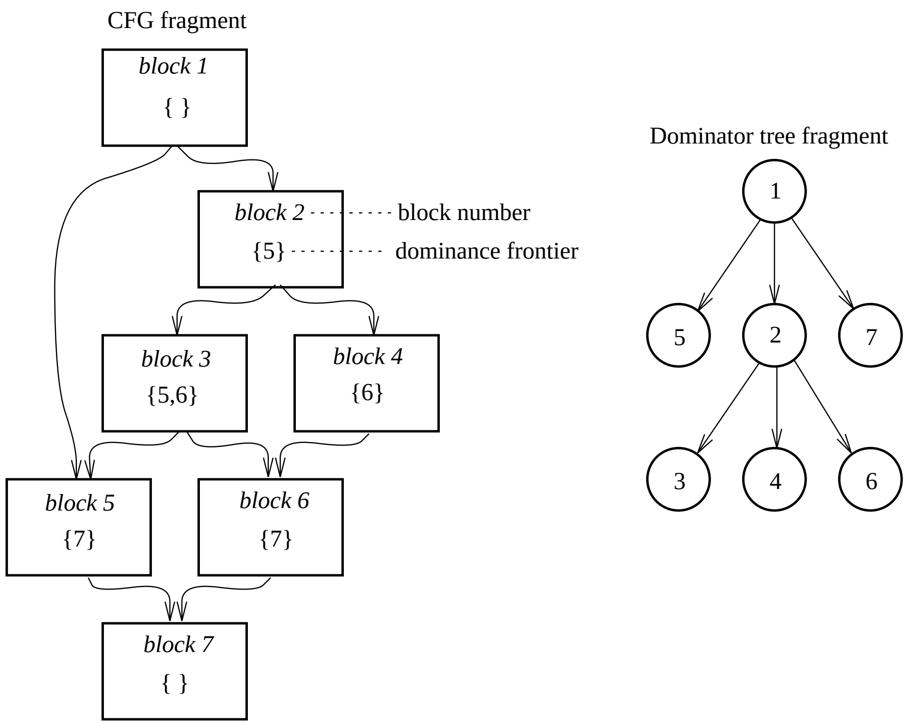


Figure 9.3: *CFG* fragment with dominance frontier sets, and corresponding dominator tree fragment

9.1.1 Placing the Φ functions

The key computational problem to be solved in transforming to *SSA* form is the placement of the Φ functions. If a function is placed for every variable in the procedure which has more than one definition, at every point of merging control, then the expansion in the size of the program can be undesirably great. An algorithm for efficiently computing the positions of a minimal set of Φ functions is known, based on the concept of dominance frontiers.

The dominance frontier of a particular node in a graph, is the set of all those nodes which have a predecessor which is dominated by the given node, but are not themselved dominated by the given node. Consider figure 9.3, which shows a *CFG* fragment and the corresponding fragment of the dominator tree.

In the figure, the dominance frontier sets are shown in braces in each block of the *CFG*. As an example, block 3 is dominated by block 2, but does not dominate block 5. Therefore from our definition, block 5 is on the dominance frontier of block 2. Similarly block 3 dominates itself, but does not dominate either blocks 5 or 6. Thus both blocks are in the dominance frontier of block 3.

Once the *CFG* and dominance tree have been computed, the dominance frontiers may be found by an efficient algorithm referenced at the end of this chapter.

The dominance frontiers tell us exactly where to put the Φ functions in the *CFG*. For every variable which has more than one definition, a Φ function must be placed at the entry of every block which is in the dominance frontier of a block with a definition for that variable.

It is assumed that during the creation of the original intermediate form a list is created which gives for each variable a list of the *CFG* nodes in which it is defined, either directly, or by possible side-effects. This information selects the dominance frontier sets which place Φ functions for each variable.

9.1.2 Renaming definitions

Once the positions of the Φ functions are known, we are able to rename all of the defining occurrences of variables, according to the *SSA* property.

We traverse each block, marking each explicit definition of each variable, including the Φ function definitions, with an unique index. We do not need at this stage to allocate unique indices to the implicit definitions which threaten variables by means of side effects.

9.1.3 Renaming applied occurrences

The index values for the used occurrences of variables in the form must now be filled in. This is done by symbolically executing the code of each block, after initializing the indices for each variable at the start of the block. As well as renaming the applied occurrences, we wish to compute the final value of the index at the end of each block, for each indexed variable.

If we process the blocks top down in the dominator tree, then we can take the initial, incoming indices to be equal to the outgoing indices for the dominator tree parent block. In the case of a control flow merge, this index value will be incorrect for any variable which has been modified along any of the paths leading from the dominating node to the merge node. However in this case the value is not used, since it is precisely these circumstances where the Φ functions will redefine the index at the top of the block.

After the outgoing variable indices have been computed for all blocks, we are able to return and rename the variables which are the incoming arguments to the Φ functions. After this step, the *SSA* form is complete.

9.2 Dealing with memory variables

Static single assignment form deals with *entire variables*. That is to say that individual, named variables are treated as single objects, even though they may have some structure.

Whenever a memory variable is written to, the whole variable is treated as having a new value number. Accesses to components of structured variables are thought of as applying suitable selection functions to the entire variables. Thus the algorithms described here must be able to determine which logical program variable a particular load or store applies to.

In the absence of some kind of annotation from the frontend, extremely conservative assumptions must be made. Furthermore, when dealing with memory, some account must be taken of possible aliassing of locations. Consider the assignment of a value to some location in memory, where the location is pointed to by some computed address value. All of the program variables to which the address might possibly point must be allocated a new value number.

There will always be cases for which the frontend is unable to exactly specify which variables might be the target of the assignment. Similarly, calls to unknown procedures must be assumed to modify the values of all statically allocated variables, and perhaps even local variables which have had their addresses taken and stored.

To some extent the *may alias* information depends on the semantics of the source language. Certainly languages which allow unrestricted taking of addresses require more conservative assumptions than those which prevent such practices. Nevertheless, for all languages certain general approaches seem possible.

The key notion is to separate every logical variable into an identity which is unique to that variable, and an *alias equivalence class*. The complexity of the computations required, depend on the granularity of the chosen classes.

A simple equivalence class set

We shall assume that every memory access belongs to one of the alias equivalence classes: *static, local* and *unknown*. Static variables are all those which are statically allocated, while local variables and parameters are allocated positions in the stack frame. The *unknown* denotation is used for all memory accesses the target of which is unknown at compile time. There may be many variables of the first three equivalence classes, but there is only one variable of the unknown class, *others*.

We may now formulate rules which govern which variables require new value numbers when an assignment is made to a variable of a particular class. As an example consider an assignment to a static variable *foo*. We may conclude, for most languages, that an assignment to *foo* will not cause a new value to appear in another static variable *bar*. What however are we to make of two consecutive accesses to the same unknown variable which are separated by an assignment to the variable *foo*? Consider —

```
lw (v99),v100    ; load value pointed to by vReg 99
sw v101,foo      ; store word to variable foo
lw (v99),v102    ; the same value as vReg 100?
```

Unless we can be sure that `v99` could not possibly point to *foo*, we must ensure that the store gives the *others* variable a new value number.

We thus have a rule: stores to static locations do not affect other static locations, but do assign a new value to variables of the unknown class. Similarly, we conclude that procedure calls assign new values to all static, local and unknown variables. Continuing with such reasoning, we are lead to the table in figure 9.4. It is left as an exercise to determine what the modified table would be, in cases

Action	**Effect**
store to static	that static, and unknown class is modified
store to local	that local, and unknown class is modified
store to unknown	all classes are modified
procedure call	all classes are modified

Figure 9.4: Alias rules for simple alias equivalence class model

where it is known which local variables and parameters have their addresses taken. Similarly, knowing which statically allocated variables are exported from the module being compiled may limit the number of variables which are thought to be threatened by out-of-module procedure calls.

9.3 Transforming back to ordinary code

In order to transform back into ordinary code we must remove the Φ functions, which do not correspond to normal instructions in the instructions sets of our target machines. What must be done is to break the functions into separate value-to-value copies which are propagated back to each of the predecessor nodes of the merge node. We must then allocate memory or register locations for each of the variables.

It may be noted that we cannot simply assume that every instance of (say) variable a may be simply allocated the location of the original variable a. This is because the optimizations performed on the *SSA* may have overlapped the lifetimes of what were originally distinct versions of the same variable. Thus we must perform a proper "graph coloring" allocation of locations or registers for the variable instances. The techniques for doing this are the same as those used for modern techniques of register allocation, and are described in detail in chapter 10.

9.4 Simple optimizations based on SSA

9.4.1 Constant propagation

If a variable in *SSA* form is defined by a constant expression, then the value is constant everywhere, and all uses of the value can be replaced by uses of the constant. This may lead to simply better code, or perhaps to the recognition of the possibility for constant expression folding. This in turn may lead either to further constant propagation, or even to dead code elimination if the result of conditional branches can be computed at compile time.

The data necessary to support simple constant propagation involves the addition of a two new fields to the per-value data structure. These fields are a Boolean *isConst* and the constant value itself.

We perform a symbolic execution of the code starting from the start node, and working our way down to the end node. Blocks need to be processed in top-down order in the dominator tree (or equivalently, in reverse post-order), so that if a value has been assigned a constant value further up the control flow then it will be guaranteed to be already noted as such in the value table. During the symbolic execution, we consider each operation. If it is a computation entirely on constants we can compute the value at compile time, and enter the result in the value table. If a computation involves a constant, and it is advantageous to do so, we change the instruction to use the literal constant rather than the virtual register.

In general we do not remove instructions which load constant values into virtual registers, since there may be later instructions for which we decide that the direct use of the literal is not justified, or is not possible. This might happen, for example, if a machine has an instruction set which does not allow literal operands for some instructions. It does no harm to leave the instruction which loads a literal into a virtual register, since if all uses of the register manage to use the literal instead, then the load instruction becomes dead code and will be detected and removed during live register analysis.

9.4.2 Dead code elimination

Since the *SSA* form has provided accurate information relating definitions and uses, we may accurately determine those definitions which are unused and remove them. The removal of unused definitions of virtual registers is one of the side-issues in register allocation.

The standard algorithm for placing Φ functions can sometimes place these in merge blocks where the left-hand-side variable is not used. If such functions are not removed, they will place an extra constraint on the register allocation process. Consider figure 9.2 back on page 226. After this fragment is transformed back into standard form, the register allocator will try to place a_1 and a_2 in the same register. This constraint makes register allocation more difficult, but saves a register to register copy on at least one of the merging control paths. If there is no use for a_3, then we should like to remove the Φ function either before or during register allocation. Once the function is removed, the register allocator has the freedom to place a_1 and a_2 in different registers.

As a result of constant propagation, it is sometimes possible to compute the direction of conditional branches at compile time. In such cases, whole blocks of code may be found to be dead. If this is the case, there may be further consequential changes, including further constant propagation.

In fact, the possibility of removing whole blocks of code suggests that we might beneficially adopt a more exact approach to constant propagation. For the naive algorithm suggested above, a single pass over the code marks all those values which are known to be constants. A value is known to be a constant if it has a constant expression assigned to it.

A better algorithm would allow the possibility that an expression computed by a Φ function might still be a constant, either because the value computed along every incoming path is the same, or because all paths except one are dead code. Instead of a Boolean attribute *isConst*, we define a mathematical lattice with three possible value markers $\top$, *const*, and $\bot$. The value $\top$, pronounced "top", means *unknown*, and $\bot$, pronounced "bottom", means *not-constant*. Initially, every value's attribute is set to $\top$, and the attribute values get lowered by a classic worklist algorithm.

The semantics of ordinary expressions for values from this value domain are defined by the following "multiplication table", where *op* is any infix operator —

a op b	$\top$	*const*	$\bot$
$\top$	$\top$	*const*	$\bot$
const	*const*	*const*	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

For example, when a $\top$ value is added to a *const* value the result is (for the time being) still *const*. As the worklist algorithm iterates, if either of the input values gets lowered, a new result is obtained.

The semantics of the Φ functions applied to this value domain are slightly different, since these functions do not combine their operands in the same way as the arithmetic functions. The table corresponding to a binary Φ function is defined as follows —

$\Phi(a, b)$	$\top$	*const*	$\bot$
$\top$	$\top$	*const*	$\bot$
const	*const*	see note	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

The special case arises if both incoming values are constant. In that case we have the result —

$$\Phi(const, const) = \begin{cases} const & \text{if consts are equal} \\ \bot & \text{otherwise} \end{cases}$$

These are optimistic definitions. We do not accept that a value is non-constant until we actually know that two different values merge at that point, or one of the incoming arguments is definitely non-constant.

`<How much detail do we want to give of the worklist algorithm?>`

9.4.3 Global common subexpression elimination

The techniques for global common subexpression elimination, become essentially identical to those used for local *CSE* elimination, given the *SSA* property.

In particular, we may convert *VAL* into *SSA*, and then perform global elimination using the same techniques as used in the local case. Even the data structures are essentially the same as those used in the local case. A value number table is required, and a hash table to lookup the tuples of the instructions, so as to determine if the computation has been performed before.

We perform a symbolic execution of the code starting from the start node, and working our way down to the end node. Blocks need to be processed in top-down order in the dominator tree (or equivalently in reverse post-order),

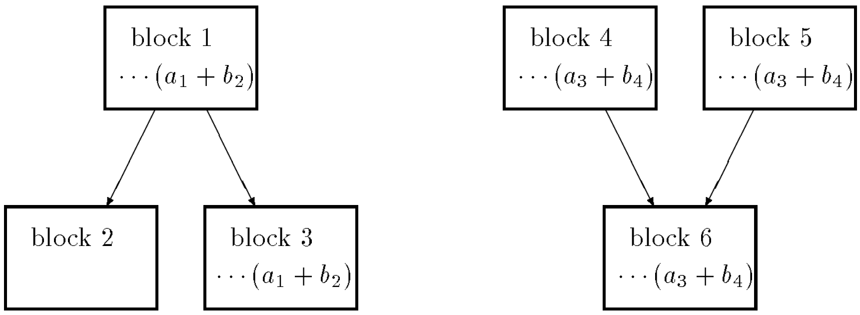


Figure 9.5: Global common subexpression examples

so that if a value has been computed further up the control flow then it will be guaranteed to be already in the table. Note a small difference however, in that if an instruction is found to be present already in the table then we cannot immediately conclude that the computation is redundant. It will be redundant if the previous computation was performed in a block which dominates the current block. However, it is possible that in the order that the blocks are processed a computation will be repeated in two nodes neither of which dominates the other. In such a case neither computation is redundant.

Consider the left hand side *CFG* fragment in figure 9.5. In this example the value of $a_1 + b_2$ computed in block 1 need not be recomputed in block 3. We see that this is the case, since the expression uses the same renamed *SSA* variables (a_1 and b_2), and block 1 dominates block 3.

Now consider the right hand side of figure 9.5, and suppose that block 4 has been processed first in the reverse post-order traversal. In this case when we find the expression $a_3 + b_4$ in block 5 we will find the tuple already entered into the value table. However, we will find that the definition of the previous computation is in block 4, which does not dominate the current block. Therefore we conclude that the computation in block 5 is not redundant.

Finally, we note that *SSA* method of computing global common subexpressions is weaker than the classical method described in section 8.2.3. If the expression $a_3 + b_4$ was computed in block 6 in figure 9.5 then our *SSA*-based algorithm would conclude that the computation in block 6 was not redundant. This is because we would find that although the computation had been previously entered into the value table, it had not been computed in a block which dominates the current block. In the method of section 8.2.3, we would have determined that $a + b$ was *available* at the entry to block 6, that the computation in block 6 was *locally anticipatible*, and hence that the computation in block 6 was redundant.

9.5 Loop-invariant code motion

Throughout this section and the next we refer to the example of the nested loops which arise in matrix multiplication. Figure 9.6 is the conventional coding of the algorithm, which multiplies an $(l \times m)$ matrix a by the $(m \times n)$ matrix b to produce a $(l \times n)$ product matrix c. We shall assume that the bounds l, m, n

```
FOR i := 0 TO l DO
  FOR j := 0 TO n DO
    t := 0.0;
    FOR k := 0 TO m DO
      t := t + a[i.k] * b[k,j];
    END;
    c[i,j] := t;
  END;
END;
```

Figure 9.6: Matrix multiplication algorithm

are not known at compile time.

When this algorithm is converted into *SSA*-form, the innermost loop, which contains two basic blocks, might appear as in figure 9.7. Note that in this figure, in the interests of simplicity, the address expressions have not been fully expanded into *VAL*. In this figure, v_{xN} denotes the virtual register holding the

`lab1:`		; *loop label*
	$v_{k2} := v_{k1} + 1$	; *increment loop counter*
`lab2:`		; *loop entry label*
	$v_{k1} := \Phi(v_{k0}, v_{k2})$	; Φ *function on* k
	$v_{t1} := \Phi(v_{t0}, v_{t2})$	; Φ *function on* t
	$v_1 := \&a + v_i \times v_m \times 8 + v_{k1} \times 8$	; *address of* a[i,k]
	$v_2 := \&b + v_{k1} \times v_n \times 8 + v_j \times 8$	; *address of* b[k,j]
	$v_3 := v_1 \uparrow_D$	; *value of* a[i,k]
	$v_4 := v_2 \uparrow_D$	; *value of* b[k,j]
	$v_{t2} := v_{t1} +_D v_3 \times_D v_4$	; *update the total* t
	`cmp` v_{k1}, v_m	; *test for loop end*
	`bne lab1`	; *branch if not equal*

Figure 9.7: *SSA*-form of inner loop of matrix multiply

SSA variable x_N. The numbered virtual registers are unique temporaries. The initial values of k and t entering the loop at label `lab2` are denoted v_{k0} and v_{t0} respectively. The operators with subscript D are double precision floating point operations. As usual, $\&a$ denotes the base address of variable a, and so on.

9.5.1 Finding loop-invariant expressions

One of the major opportunities for optimization in loops arises when an expression within a loop is evaluated, but yields the same value on each traversal. This is an interesting example of optimization, since it raises issues of safety and profitability.

Expressions which yield the same value on each traversal of a loop are said to be **loop-invariant**. Such expressions arise very frequently, and do not indicate careless coding on the part of the programmer. As an example, consider the code for accessing the (i, j) element of an array a. The address expression is $(\&a + i \times c_1 + j \times c_2)$, where $\&a$ is the array base address, and where c_1 and c_2 are the constant row size and element size respectively. In an inner loop in which only one array index varies, a substantial subexpression of the computation is loop-invariant.

Recognising loop-invariant subexpressions in *SSA* form is very simple. Assuming for the moment that we have identified the loop header node, and know which instructions are inside the loop and which outside, we have the following recursive definitions —

> **Definition:** An *SSA variable* used within a loop is loop-invariant if its (single) definition is outside the loop, or if it is defined by a loop-invariant expression.
>
> **Definition:** An *expression* within a loop is loop-invariant if all operands of the expression are loop-invariant variables or subexpressions.

These definitions allow loop-invariant expressions to be recognized by the following strategy. We describe the process for the case of a single loop, but the extension to nested loops is straightforward. We extend the attribute used to mark values for constant propagation, in section 9.4.1, so that it now has four values: $\top$,*const, invariant*, and $\perp$. If we assume that constant propagation has been already carried out, some values are already marked as constant, and we initially mark all others as invariant.

The basic blocks of the loop are now visited in reverse post-order, and within each block each value-definition is examined. If the value arises from a Φ function, it is not loop-invariant and is lowered to $\perp$. If the value is computed from operands at least one of which is $\perp$, then the value is also marked $\perp$. After this traversal all the non-invariant values will have been so marked. Note that a single reverse post-order pass over the blocks is sufficient, since we conservatively assume that the Φ functions always merge different values.

The treatment of memory accesses in this algorithm is instructive. If a particular memory variable is defined within a loop, then the algorithm in section 9.1.1 will place a Φ function for that variable in the loop header node. Thus any values loaded from the variable will be marked $\perp$. Consider however the case of an array which does not have a definition inside the loop. In that case the variable will not have a Φ function, and hence the value of the array, treated as an *entire variable*, will be loop-invariant. However, any indexed access to the array will only be invariant if the *index* is invariant. In essence the load instruction —

```
lw  arr(v99),v100   ; load word at the index given by vReg 99
```

is only loop-invariant if the array **and** the index are both invariant.

Looking at the data accesses of the matrix multiplication of figure 9.7, it is seen that the variables a and b are both loop-invariant. In the inner loop, major parts of the address expressions are loop-invariant. In the expression —

$$\&a + v_i \times v_m \times 8 + v_{k1} \times 8$$

$(\&a + v_i \times v_m \times 8)$ is loop invariant. If this value is loaded into a virtual register, say v_5, outside the loop, then the address evaluation becomes —

$$v_1 := v_5 + v_{k1} \times 8$$

Similarly, the second address expression —

$$\&b + v_{k1} \times v_n \times 8 + v_j \times 8$$

$\&b$ is loop invariant, as is v_n and $(v_j \times 8)$. If the first and last of these expressions are loaded into v_6 and v_7 respectively, then the address evaluation becomes —

$$v_2 := v_6 + v_{k1} \times v_n \times 8 + v_7$$

In this particular case, if we had been able to detect a potential use of the commutative and associative laws, we would have been able to add v_6 and v_7 outside the loop, and also scale v_n by 8, reducing the computation inside the loop still further.

9.5.2 Moving loop-invariant code

Ideally we would like to simply move loop-invariant expressions outside the loop, and compute them once only, but some caution is required. As a general rule, whenever code is moved within a *CFG* if must be moved to a position where it dominates the original code location. This requirement ensures that uses of the moved values will find the values available. However, for safety we must also ensure that we do not compute values which might otherwise not have been computed. Consider a loop with an *IF* statement nested within it. Figure 9.9 on page 238 has such a loop.

If one branch of the loop has a loop-invariant expression within it, then moving the expression outside the loop will ensure that the expression is always computed, even in cases where the original code may have not done so. The property which is required to ensure that no computation is performed which would not otherwise have been done is precisely the *downSafeIn* propery discussed in section ??. If an expression is loop-invariant and is *downSafeIn* at the loop header node, then it may safely be moved outside the loop.

A final question arises as to where to move such loop-invariant computations. In *REPEAT* loops the situation is simple, since placing the invariant code immediately above the loop header is safe. In *WHILE* loops the problem is more difficult. Consider the two alternative encodings of the *WHILE* loop shown in figure 4.11c and 4.11d on page 123. In the first case the position immediately above the loop is unsafe, since the code is unprotected by the loop pre-test condition, and would be executed even in cases where the loop should have been skipped over. The second encoding is safe, since there is a position between the first conditional test and the loop header in which the invariant code may be placed.

Restructuring the CFG

The implementation of loop-invariant code motion is made much simpler by restructuring the control flow graph so that it has certain standard characteristics. In particular every loop in the transformed form has three distinguished nodes, which we shall call the *pre-header*, the *header*, and the *latching node*. Figure 9.8 shows a loop in the canonical form. If the *CFG* is reducible then every loop

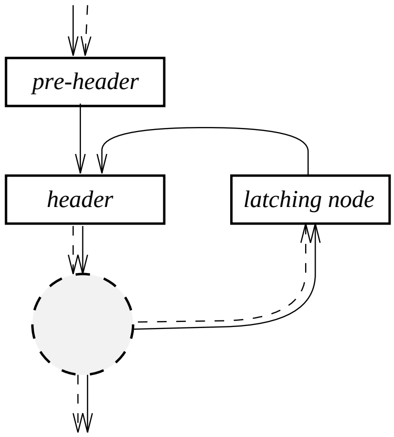


Figure 9.8: A loop in the canonical form

has an unique header node. The purpose of the latching node is to ensure that the header node has a single backedge incident on it. If there are multiple back edges in the original *CFG*, arising (say) from use of the `continue` statement in language *C*, then an empty node is introduced as the target of the original backedges, and a single edge leads from this new node to the header node. Note that the header node is the only successor of the latching node.

The purpose of the pre-header is to ensure that all control flow reaching the header node from outside the loop passes through a single node. This node dominates the header, and is postdominated by the header node. In the case of pretested loops, this may require some significant restructuring of the loop, as described in section 4.5.3 and figure 4.11. Loop-invariant code will be moved to this node. Note that the header node is the only successor of the pre-header node.

Once the loop has been restructured into this form, we may use it to perform a number of tests on the *CFG*, so as to categorize the blocks of the loop. We use the following definitions —

- a block is *semi-safe* if it is executed on every traversal of the loop which reaches the latching node
- a block is *down-safe* if it is executed on every traversal of the loop, including the last

In figure 9.9 the semi-safe blocks do not include either the thenpart or elsepart

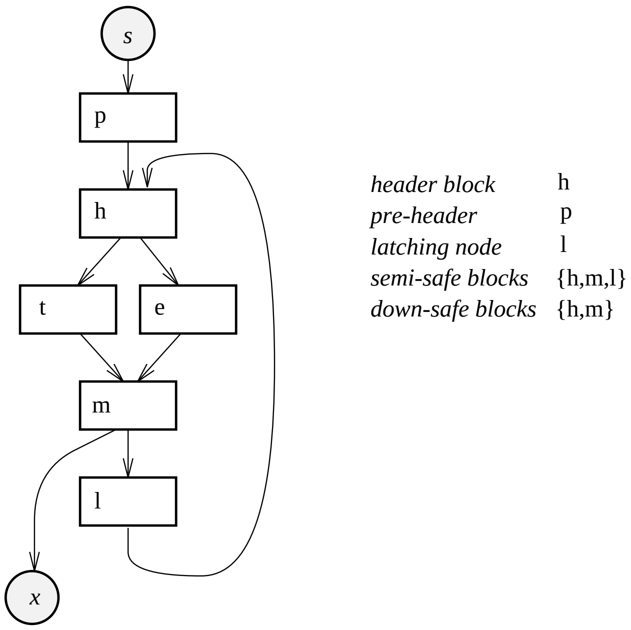


Figure 9.9: Example loop, with safe block-sets

blocks of the if-statement in the loop. The latching node is not in the down-safe set, since on the last traversal of the loop this will not be executed. Indeed, if the loop is traversed only once, then the code of the latching node will never be executed at all.

Taking a loop in canonical form, we may immediately see that all the blocks in the loop which dominate the latching node are semi-safe, since these are the blocks that *must* be executed before the latching node is reached. The blocks of the loop which post-dominate the header are exactly those that *will* be executed, once the loop is entered.

In general, we may find the semi-safe blocks by traversing the dominator tree starting from the latching node, and ending with the header node. It is left as an exercise to show that a loop is reducible if and only if the header dominates the latching node.

Down-safe blocks may be found by listing all the ancestors of the header node in the post-dominator tree, excluding nodes not in the loop. There does not seem to be any simpler way than finding each ancestor in turn, and testing to see if it is within the loop. The asymmetry in behaviour of the dominator

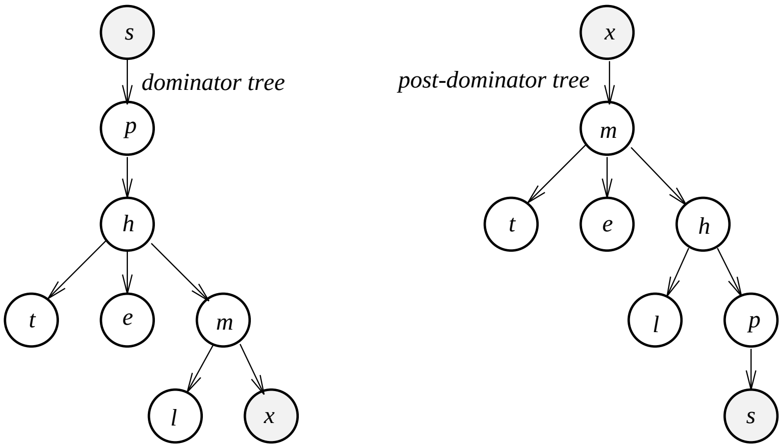


Figure 9.10: Dominator and post-dominator trees, for figure 9.9

and postdominator trees arises because the control flow graph with all edges reversed may not necessarily be reducible. Indeed in cases where loops have multiple exits this is very common.

The dominator and post-dominator trees for the graph in figure 9.9 are shown in figure 9.10. Note that the down-safe blocks lie from h up to m in the post-dominator tree, while the semi-safe blocks lie from l up to h in the dominator tree.

Moving invariant expressions

Once the down-safe blocks of a loop have been identified, it becomes possible to find the loop-invariant expressions which are computed in those blocks. This may be done using the algorithm described earlier, but restricted to the down-safe blocks, visited as usual in reverse post-order.

Note that any values computed within the loop which are in down-safe blocks are always safe to move to the pre-header, since the pre-header, by construction, is post-dominated by every down-safe block in the loop.

Values computed in other blocks of the loop are more problematical, both in terms of safety and profitability. Safety is a consideration because moving computations from non-down-safe blocks might cause runtime errors which would not occur in the unoptimized program. Therefore it is necessary to check that any expressions which are *speculatively moved* in this way are not able to cause runtime errors. Profitability is likewise a consideration, since the speculative moving of even safe expressions may lead to the computation of values which are never used. Consider an expensive computation in figure 9.9, in the elsepart block labelled “e” in the figure. In the absence of any other evidence it may be thought that the else path would be followed half of the time. If this estimate is grossly in error, then the optimization may actually make the code run more slowly. Some kind of compromise is clearly needed, and a reasonable choice is to speculatively move non-trapping computations out of semi-safe blocks, but to leave other computations where they are. Since it is normal to perform a second pass of global common subexpression elimination after any code motion, it is possible that code in non-safe blocks may be able to use the values in the moved code in any case.

Some practical considerations

There are a number of practical issues which need to be considered whenever code motion algorithms, including loop-invariant code motion, are implemented. One of these is the effect of such code motion on the number of registers which will be used by the program.

Consider the case of an access to a scalar, statically allocated variable inside a loop. We shall assume that the variable is not loop invariant.

```
lw  statVar,v100    ; load word statVar into vReg 100
```

Although many *RISC* machine assemblers support syntax as in this example, for all such machines the load will actually take at least two machine instructions. This is so, because address values are too large to fit into the immediate field of the instruction format. Now, note that although the *value* to be fetched is not invariant, the *address* of the value certainly is. Thus we might aggressively preload the address of the variable outside of the loop, leaving ourselves a single instruction load inside the loop.

```
                    ; outside the loop
la   statVar,v101   ; load address of statVar
...
                    ; inside the loop
lw  (v101),v100     ; load word statVar into vReg 100
```

Similar reasoning applies in the case of non-invariant indexing into arrays.

The problem is that a large loop might easily load more registers with addresses than it is possible to contain in the register file of the machine. If these address values are spilled, then the code will almost certainly run slower than without the “optimization”.

A practical solution is to aggressively move such address expressions out of loops, but to mark the values so that they may be given priority for spilling if machine registers become scarce. Furthermore, if such constant values fail to be allocated to machine registers, then the values should not be saved to memory and restored. Instead the known values should be *reincarnated* as required within the loop. In effect, each virtual register holding a constant expression should be marked, and the evaluation required to regenerate the value should be remembered[2].

Similarly, practical considerations usually forbid the moving of memory store instructions from their original blocks. Moving values in registers causes no problem, since if several instances of the same program variable overlap, then

[2]Note well that this technique only works in the presence of *SSA*. It is of no use to remember that a particular register was assigned the value 0, if somewhere else in the code the same register was assigned the value 1.

the graph coloring involved in converting back into standard code will ensure that each instance is allocated to a different machine register. However, if memory values are moved, then we must be prepared to reassign locations in memory if it happens that instances have overlapping lifetimes.

9.6 Induction variables

There is a different kind of loop optimization which is often very valuable. This arises when variables inside a loop have values which increase or decrease by the same amount on each traversal of the loop. Taking the example of matrix multiplication again, figure 9.11 shows the *SSA* code after the removal of loop-invariant subexpressions.

`lab1:`		; *loop label*
	$v_{k2} := v_{k1} + 1$	; *increment loop counter*
`lab2:`		; *loop entry label*
	$v_{k1} := \Phi(v_{k0}, v_{k2})$	; Φ *function on* k
	$v_{t1} := \Phi(v_{t0}, v_{t2})$	; Φ *function on* t
	$v_1 := v_5 + v_{k1} \times 8$	; *address of* a[i,k]
	$v_2 := v_6 + v_{k1} \times v_n \times 8 + v_7$	; *address of* b[k,j]
	$v_3 := v_1 \uparrow_D$	; *value of* a[i,k]
	$v_4 := v_2 \uparrow_D$	; *value of* b[k,j]
	$v_{t2} := v_{t1} +_D v_3 \times_D v_4$	; *update the total* t
	`cmp` v_{k1}, v_m	; *test for loop end*
	`bne lab1`	; *branch if not equal*

Figure 9.11: Inner loop of matrix multiplication, after removal of loop-invariant subexpressions

In this figure, some analysis will show that each time through the loop the value of v_1 is greater than the previous value by exactly 8. Similarly, each time through the loop the value of v_2 is greater by exactly $(v_n \times 8)$. Thus it is possible to initialize the two address values outside the loop, and increment each by the appropriate amount in the loop latching node. The total effect of this transformation is to replace the multiplication and additions on each pass with two simple additions.

In order to perform such transformations, we must be able to recognize those expressions which have these simple stepwise value sequences. Such values are called *simple induction variables*. The name refers to the classical *proof by mathematical induction* technique that would be used to prove that the recurrence equation —

$$\forall m \geq 0 : x_{m+1} = x_m + c$$

has the solution —

$$\forall n \geq 0 : x_n = x_0 + c \times n$$

9.6.1 Recognizing induction variables

The most general form of induction variable which we consider here are those that step in value by some loop-invariant amount on each traversal of the loop. We shall assume that on the n-th iteration of the loop the expression has the value —

$$v_n = E_{init} + E_{step} \times tripCount$$

where E_{init} is a loop-invariant initialization value, and E_{step} is a loop-invariant increment value, and where *tripCount* is the number of times that the backedge of the loop has been traversed.

In order to determine which expressions computed within a loop are induction variables in the required form, we first observe that all such expressions must depend directly or indirectly on variables appearing in Φ functions in the loop header block. We call all such variables, which appear as left-hand-sides of loop header Φ functions *loop carried variables.*

Not all loop carried variables are useful for forming induction expressions. In order to be useful, the loop variable must step by some loop-invariant amount. We shall call such variables *induction base variables.*

In order to find the induction base variables, each Φ function in the loop header block must be examined in turn. The argument which reaches the function along the backedge from the latching node is traced backward along the edges of its dependence graph until its behaviour is known. If it depends only on its own loop carried variable, with additive invariant expressions, then it is a base variable[3]. All the other loop-caried variables are marked as bottom ($\perp$). The data structures used for global common subexpression elimination contain sufficient information to allow such tracing of the dependences to be performed.

Once the induction base variables are known, it is possible to find all induction variables in a single reverse post order pass over the blocks of the loop. We extend our value lattice so that it distinguishes induction (*step*) variables, as well as the usual constant, loop-invariant and $\perp$ values. In this framework, we rank constants (*const*) higher than loop invariants (*invariant*), which in turn are higher than induction variables of either kind (*step*). All of these are higher than $\perp$.

With this formulation, all of the following are induction variables —

- an induction base variable
- the sum of two values each $\geq$ *step*
- the product of a *step* and any value $\geq$ *invariant*

All other computations are invariant, or are marked as $\perp$.

Note that we are not required to build any additional data structure to represent the initial and step expressions of an induction variable. In the *SSA*-form, the value dependence graph contains sufficient information to allow the

[3]Here we have actually wimped out slightly. In truth, some dependences on other loop carried variables should also be permitted. The choice made here captures the vast majority of cases, and simplifies the algorithm significantly.

initial part and the step expression to be determined from separate traversals of the graph. We may treat the dependence graph of an induction expression as having an embedded expression tree, the leaves of which are constants, loop-invariants and induction base variables.

9.6.2 Transforming the loop

Suppose that an induction variable has been discovered within a loop in our canonical form. Figure 9.12 shows such a canonical loop before and after the loop is transformed. Once an induction variable has been identified, the initial-

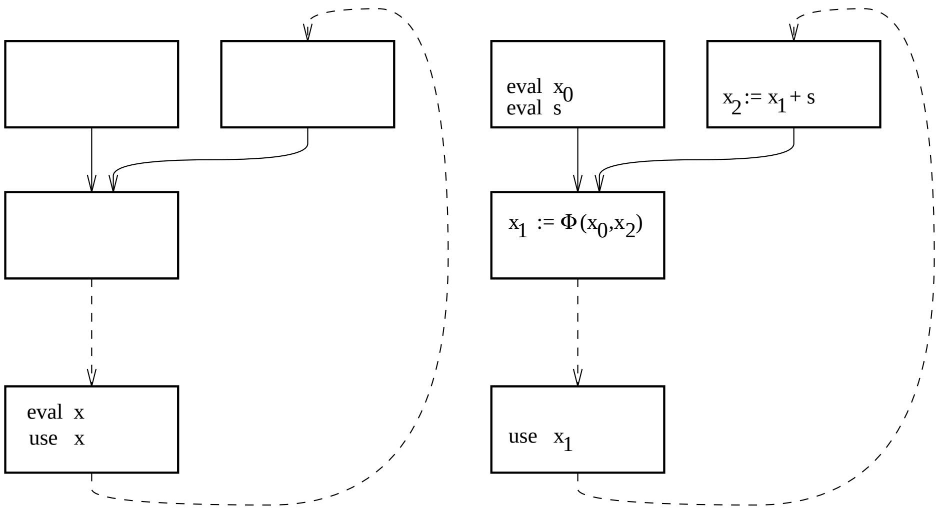


Figure 9.12: Loop with induction variable $x := x_0 + s \times n$, where n is the trip count, before and after transformation

ization and step expressions must be determined. A new loop carried variable is introduced which will be used instead of the induction variable. Of course, this variable will require a new Φ function in the loop header. The evaluations of the initialization and step expressions are then placed in the loop pre-header, and an increment of the new variable by the step amount placed in the loop latching node.

On machine architectures which have a two-register address mode, it is possible to determine all those memory accesses with induction variable addresses which share the same increment expression. All of the memory accesses within such a group may then share the same loop carried variable. In this case each address would use its own initialization as one address register, and the loop carried variable shared with all other members of the group as the second address register.

When induction variable analysis is applied to our matrix multiplication example we find two induction variables, which we shall call x and y. The first,

`lab1:`		; *loop label*
	$v_{k2} := v_{k1} + 1$	; *increment loop counter*
	$v_{x2} := v_{x1} + 8$	; *increment ind-variable* x
	$v_{y2} := v_{y1} + v_8$	; *increment ind-variable* y
`lab2:`		; *loop entry label*
	$v_{k1} := \Phi(v_{k0}, v_{k2})$	; Φ *function on* k
	$v_{t1} := \Phi(v_{t0}, v_{t2})$	; Φ *function on* t
	$v_{x1} := \Phi(v_{x0}, v_{x2})$	; Φ *function on* x
	$v_{y1} := \Phi(v_{y0}, v_{y2})$	; Φ *function on* y
	$v_3 := v_{x1} \uparrow_D$	; *value of* a[i,k]
	$v_4 := v_{y1} \uparrow_D$	; *value of* b[k,j]
	$v_{t2} := v_{t1} +_D v_3 \times_D v_4$	; *update the total* t
	`cmp` v_{k1}, v_m	; *test for loop end*
	`bne lab1`	; *branch if not equal*

Figure 9.13: Matrix multiply inner loop, with transformed induction variables

x, steps by a constant 8, and is initialized to —

$$x_0 := \&a + v_i \times v_m \times 8$$

The second induction variable, y, steps by the loop invariant amount $v_n \times 8$. This variable is initialized to —

$$y_0 := \&b + v_j \times 8$$

With these definitions, the inner loop of the matrix multiply becomes as shown in figure 9.13. Note that the address expression computations have been completely eliminated, and replaced by two simple additions instructions in the loop latching node. The appropriate comparison is with figure 9.11, which on each pass of the loop requires three additions, two shifts and one multiply. Of course, without loop invariant code motion the comparison is even more favorable to our final version.

As usual, some practical considerations apply. Induction variable analysis will generate additional register values, which are required to be live throughout the body of the loop. This may add considerably to the register pressure in some cases, causing additional register spilling.

As well, questions of safety and profitability suggest that the transformation of induction expressions which are computed in non-down-safe blocks should be done with the usual cautions.

9.7 Extended forms of SSA

The simple approach to static single assignment form outlined here may be extended in a number of ways. *SSA* form, in common with the underlying dataflow analysis method, deals with *all possible* paths in the *CFG*. A more exact analysis is possible if we take account of the conditional expressions which lead to particular paths being taken. Gated *SSA*[?] is such a form, in which Φ functions are decorated with the conditional expressions. The *program dependence graph* is another representation, which carries equivalent information.

`<Maybe something on an application of gated-SSA? ... >`

There are sevaral different extensions to *SSA*, in which the constant propagation idea is extended to rather different attributes on the values. Corney and Gough[?] propagated upper bounds on the polymorphic types of values in programs in *Oberon-2*, in order to eliminate runtime type tests. Patterson[?] used a similar idea in a much more complex setting, to propagate value-range density functions, in an attempt to derive extremely sharp branch probability estimates.

Gough and Klaeren[?] propagated value-range information in order to eliminate redundant index, range and overflow tests. In ordinary *SSA*, a new value is created whenever an assignment is made to a program variable. In this extended case, a new value is also created whenever the bounds on a value-range are changed by conditional branches or by runtime tests on that value.

9.8 Finding out more

The definitive paper on transforming to *SSA* form is Cytron, Ferrante, Rosen, Wegman and Zadeck[?]. There are now some known improvements on the algorithm given in this paper, but the original algorithm works well in practice. In the case of structured languages, it is possible to find the placement for Φ functions directly from the *AST*, as shown by Brandis and Mössenböck[?].

The key paper on optimistic constant propagation is also by Wegman and Zadeck[?]. Gated *SSA* is introduced in [?].

Click has shown how to use *SSA* form as a basis for performing very aggressive code motion[?]. In this work *SSA* is used to construct a global value dependence graph. This graph is then used to schedule all operations, with almost no regard to the original placement of operations in basic blocks. In theory it appears that such code motion may sometimes unnecessarily lengthen the lifetime of temporary values in registers, and hence increase register pressure. Nevertheless, the measured results achieved by the algorithm are very impressive.

9.9 Exercises

9.1

9.2

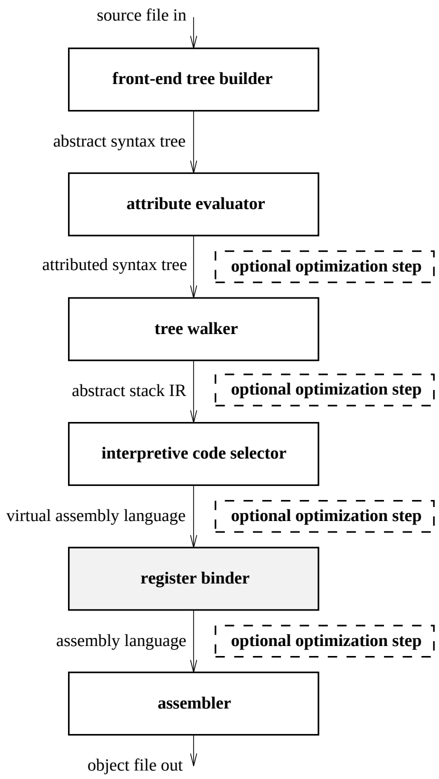
9.3 Consider the situation shown in right-hand side of figure 9.5. In this case neither computation of $(a_3 + b_1)$ dominates the other. Nevertheless, the definitions of a_3 and b_1 must necessarily be in *CFG* blocks which dominate both block 4 and block 5. Can you see why this must be the case?

Why is it not always safe to simply *hoist* the computation up to the nearest node which dominates all of the uses of $(a_3 + b_1)$?

Can you think of a case where hoisting the code might be unprofitable, even if it is safe?

Chapter 10

Register Allocation



10.1 Introduction

The allocation of registers for temporary values and variables is one of the most important factors in the production of good quality code. It has historically been an important factor for those machines with a small number of registers, even for the production of code to evaluate expressions. Furthermore, older machine architectures typically placed restrictions on the use of registers by implementing instructions which required operands to be in particular registers, or in particular register subsets.

Recent machine architectures appear to have alleviated this problem by offering a relatively large number of general purpose registers, and *orthogonal* instruction sets which allow any register to be used as an operand for any instruction. Almost all contemporary machine architectures provide at least 32 registers. However, all is not so simple. The so-called *RISC* architectures have provided this increased resource at the cost of simplifying the instruction set and adopting a load-store processor model. In order to achieve good results with such architectures it is essential to use the register resource efficiently. The paradox is that modern machines provide more registers, and greater ease of use, but demand proportionately greater attention to register allocation.

The importance of register management may be expected to increase even further in the future. This is so, since the speed of processors is increasing at a greater rate than the speed of main memory. If follows that the time taken for fetching an operand from memory will increase relative to the instruction cycle time. Thus the penalty for failing to place a value in a register will be correspondingly greater.

The efficient use of the register resource requires two separate problems to be solved. We expect our compilers to work very hard to find many useful values to hold in registers. In some cases we may even find too many. When we have found too many candidates, we must decide *which values* will be allocated to registers, and for which parts of the program. Further, we must decide *which register* each of these values will be assigned to. These two problems may be called the **allocation problem**, and the **assignment problem**. These two problems may be tackled separately, or together.

There is one further dimension to the register allocation problem. We may try to allocate registers *locally*, that is, one basic block at a time, or we may try to allocate registers *globally*, that is, a whole procedure at a time[1]. Here is a quick overview of the different approaches to global register allocation. A later section of this chapter looks at the simple heuristics of local allocation.

The born-in-register approach

In this approach to register allocation it is assumed that *every* value is held in a register throughout its useful lifetime. This assumption may be an overallocation, so that the attempt to assign registers to values fails because the supply of registers becomes exhausted. In this case one or more values must be *spilled* to memory, and a new allocation attempted. This process continues until an assignment attempt completes successfully.

The main difficulty with this approach is that the most powerful algorithms for assignment require restarting after each spill decision. There does not seem to be any way to update the data structures incrementally after values have been

[1] Even more radical approaches have attempted to allocate registers across procedure boundaries. However, these truly global approaches are highly experimental, and do not fit well with systems which rely on separate compilation of program modules.

spilled. These algorithms thus tend to be very laborious when register pressure is high, and use quite large amounts of memory during compilation.

The born-in-memory approach

In this approach to register allocation it is assumed that every value is either a variable, or is a temporary value which has been allocated a location in the local stack frame. The benefits of keeping each value in register are estimated, and these values are used to decide which values to *promote* into registers.

Once again, the main difficulty with this approach is the high level of algorithmic complexity which is involved. Algorithms based on this approach also use very substantial amounts of memory during compilation.

The local-incremental approach

In this approach to register allocation an initial assignment is performed using a local allocation algorithm. Having determined how many registers remain free, the algorithm attempts to incrementally move additional values into registers until all registers are exhausted.

For languages for which the programmer is allowed to express a preference for which local variables are placed in registers, this approach can allow a simple implementation. For example, if there are registers left over after local allocation, the remaining registers may be allocated to local variables in the priority order given by the programmer, or the front-end. The main difficulty with this approach is the rather subjective nature of the decision process which decides which values to promote into registers during the second phase of the process.

10.2 Graph coloring

10.2.1 Live ranges

The problem of assigning registers to values is modelled by a well understood but difficult mathematical problem, that of **graph coloring**.

At any particular point during the excecution of a program, certain values may be held in registers. Each of these values is *defined,* or brought into being, by one or more instructions. Along every subsequent path of control each value will have some final use. All of the locations which lie between the point(s) of definition and any final use are called the **live range** of that value.

The live range of a value is always a connected region on the control flow graph but is not necessarily a contiguous range of locations in the *VAL* buffer. Consider a value defined prior to an *if* statement, and used in each of the branches, as in figure 10.1.

In this case, a value is live across a *fork* in the control flow. Since the flow of control may only *fall through* into one of the branches, it is clear that the single (forked) live-range will appear as two distinct intervals in the object code buffer.

In the case of straight-line control flow, every live range has a single definition, and a single last use. Thus the live range of a value is simply a contiguous range

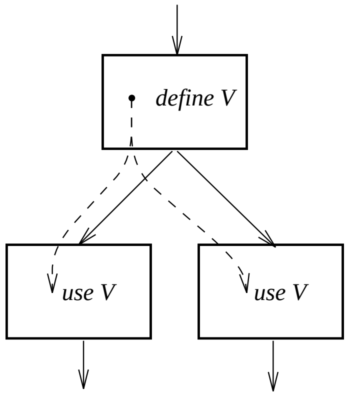


Figure 10.1: A forked live range

of instruction locations. This range extends from the definition point, to the final use. Such live ranges in the contiguous case, are called **intervals**.

For the code generation methods outlined in previous chapters virtual registers possess a single definition point. However the values which we consider here have a richer, more complex structure. Consider the code fragment —

```
IF condition THEN
  x := expr1;
  ...
ELSE
  x := expr2;
  ...
END;
...
v := ... x ...;
...
```

The value of x is a candidate for holding in a register, between its two definition points and its subsequent use following the *if*. We would thus like to target the same register as the register destination for the evaluation of both *expr1* and *expr2*. In this case the "value" has two different values.

Contrast this with the following case where a temporary variable is used for two different purposes —

```
tmp := v1;     (* swap v1 and v2  *)
v1  := v2;
v2  := tmp;    (* tmp is now dead *)
...
tmp := x1;     (* now swap x1, x2 *)
x1  := x2;
x2  := tmp;    (* tmp is now dead *)
```

It is advantageous to hold the value of *tmp* in a register, rather than actually storing the value into memory after each assignment and reloading it later. The value-tracking optimization discussed in the previous chapter allows us to do this quite simply. However, there is no advantage in targeting the two assignments of the variable *tmp* to the *same* register.

Recognizing the difference between these two cases is the key to understanding the nature of live ranges. In the first example the two definitions of x belong to the same live range because (at least one) use of the value is reached by both of the definitions. In the second case the two assignments to variable *tmp* are to different live ranges, since the live ranges are not connected in the control flow graph.

The idea of *live range* in this sense is subtle. A live range is defined by some smallest connected region of definition(s) and use(s). We may have a single definition which is used in several locations even after control flow divergences. Equally we may have multiple definitions which are connected to the same value below control flow merge points.

Live ranges arising from *SSA*

In the *VAL* which arises from the *SSA* techniques discussed in the last chapter, a live range with multiple definitions will occur as separate virtual registers which are joined together by a Φ function. In this case, we shall initially treat every separate virtual register as a separate live range, and then try to merge these connected live ranges before register allocation. If we are successful, then the whole of the live range will have a single register allocated to it. If we are unable to merge the ranges into a single register, then the Φ function will give rise to one or more register-to-register copies in a predecessor of the merging block.

Consider the *VAL* which arises from a statement which returns a Boolean value, in figure 10.2. In this figure, the live ranges for v_x, v_y and v_z are initially

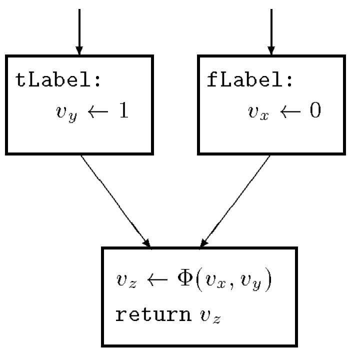


Figure 10.2: Live-range arising from “`return` *BoolExp*”

separate. However, we shall try to merge these ranges during allocation, so that the two literals are directly allocated to the same register. Of course, our register allocator will also try to ensure that the merged live range is allocated the function return register, so that the return statement requires no instructions.

10.2.2 The interference graph

Values which have overlapping live ranges must be assigned to different registers. In fact, at any point in the execution of the program each of the values which are live at that point must be assigned to a different register.

We model this rule by a classical problem from the theory of (undirected) graphs. Each live range corresponds to a node in the **interference graph**. An edge is drawn between each pair of nodes which have overlapping live ranges, and which hence interfere with each other. The problem is to find a way of coloring the nodes of the graph so that no two adjacent edges have the same color.

For such undirected graphs the **chromatic number** is the minimal number of colors which are required to color the graph. A graph is k-colorable if it has a chromatic number less than or equal to k.

A subset of nodes for which every node is adjacent to every other node in the subset is called a **clique**[2]. The **clique number** of a graph is the size of the maximum clique. Clearly all members of a clique must be allocated different colors, and hence the chromatic number of the graph cannot be smaller than the clique number. The chromatic number may exceed the clique number, as is shown by figure 10.3, where the chromatic number is three, although no clique is larger than 2.

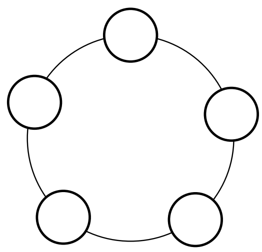


Figure 10.3: Graph with chromatic number 3, clique number 2

In the register allocation context each color corresponds to the allocation of a particular live range to a particular register. The chromatic number of the graph is the number of registers required to complete an allocation. All of the registers which are alive at any particular point in a program form part of a clique, and hence must be in different registers. However, a clique may be larger than the number of registers alive at any one point. Consider the program control flow graph in figure 10.4, where the clique number (and the chromatic number) is 3, but no more than two values are live at any point in the execution of this program.

The problem of computing the chromatic number of an arbitrary graph is known to be *NP*complete, which means that an exact solution will be computationaly infeasible except for the most simple cases. The apparently simpler

[2] Clique is a word used to describe an intimate and exclusive social group. In effect, every person is "connected" to every other of the group. The word is pronounced "cleek" and "click" in different English dialects.

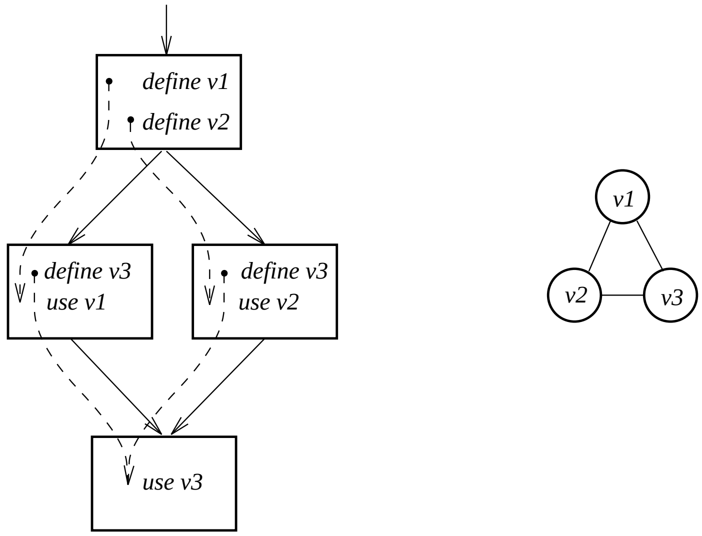


Figure 10.4: *CFG* and interference graph with chromatic number 3

problem of answering the question "is graph G k-colorable" is also NPcomplete.

Perfect graphs

Graphs for which the clique number of the graph is equal to the chromatic number are said to be **perfect**. There are a variety of special topologies for which all graphs are known to be perfect. For example, graphs which model the overlap of intervals on the number line are known to be perfect. In this model, intervals on the number line correspond to nodes of the graph, with edges between those nodes which correspond to intervals which overlap on the line. These **interval graphs** exactly model the behaviour of register allocation in straight-line control flow.

An important property of perfect graphs is that they may be colored in polynomial time. In the case of interval graphs, these may be colored in almost linear time. We use this propery in later sections.

The graphs induced by the interference of intervals on a circle are also perfect, suggesting that special, locally optimal register allocation algorithms may be computationaly feasible for program fragments with simple loop structures.

Similarly, the interference between subtrees of a tree induce perfect graphs. This is an interesting result since values arising from a single definition point have live ranges which are trees on the control flow graph. Perhaps a restriction to such values might lead to a simple coloring algorithm.

10.3 Heuristics for local allocation

10.3.1 On-the-fly allocation

The method of register allocation used in the very simplest compilers is to allocate registers as required, as code is generated. This method is called on-the-fly allocation. In effect, the code generator has a module which manages the registers, and maintains the data structures which represent the register state. This module typically exports procedures with the following signatures —

```
PROCEDURE AllocRegister() : RegIndex;
PROCEDURE FreeRegister(reg : RegIndex);
```

In the case of machines with different register classes there may be a need to parameterize the allocation procedure.

The simple strategy of reallocating an already freed register whenever possible is an effective heuristic for the allocation procedure. This strategy is actually optimal for straight line code, although there are some serious practical limitations which we discuss later.

Consider the problem of coloring an interval graph, where the nodes are colored in order of left-most end of the intervals. We shall suppose that there is a data structure which lists the colors used and then freed. When a color is required for a new interval, and the free list is not empty, an arbitrary choice is made from the "freed list". If the list is empty, as it is initially, then a new color is chosen. When the end of each interval is found, the color allocated to that interval is placed on the free list.

In effect this algorithm scans the number line from left to right allocating and deallocating colors as the beginning and end of each interval is found. At any point on the line, the number of colors used is exactly equal to the number of intervals which cover that point. Since the colors are reused, it follows that the maximum number of colors used is exactly the clique number of the induced graph. Since interval graphs are perfect, it follows that the algorithm allocates the minimal number of registers.

Discussion

On-the-fly register allocation, in spite of its local optimality, has a number of serious problems. In order to see why this is, we need to consider to what extent the assumptions inherent in the method are justified.

The major difficulty with the method in practice, is that insufficient is known about the uses of the value at the time that the allocation must be made. Even on machines with large numbers of general purpose registers there are conventions which govern the use of particular registers. Typically registers are divided into caller-saves and callee-saves subsets, as detailed in the chapter on runtime organization. Furthermore, register use conventions may require specific registers to be used for parameter passing or the return of function values.

If follows that on-the-fly allocators often place values in registers which later on turn out to be needed for some dedicated purpose, or need to be saved and restored around a procedure call. In such cases the register allocation needs to be corrected, or additional code generated to copy or save the value.

The allocation may be done much more effectively if full information is available on the uses of the value. However, as we shall see presently, to accumulate this usage information requires some buffering of code, in which case alternatives to on-the-fly allocation become possible.

Spilling registers on-the-fly

If it should happen that the number of registers needed exceeds the available number, then at least one value previously in a register must be discarded. The choice of which value to discard first is important.

Given sufficient information, we may classify values in registers on the basis of two attributes —

- a value is **locally-live** if it is used again within the same extended basic block, otherwise it is **locally-dead**
- a value is either **pending, saved** or **volatile**. Newly computed values of variables are **pending** until they are written to their allocated memory locations, at which point they become **saved**. Loaded variable values are also already **saved**. Temporary values in registers which do not have memory locations allocated to them are **volatile**

Notice that we distinguish between a value in a register being locally dead, (that is, not used again in the extended basic block) and the corresponding variable itself being dead. To determine the liveness or otherwise of variables requires non-local analysis. However, if we simply agree to *always* save pending values, we shall always be safe. We shall occasionally save a value to memory which is never reused, but we do not need to compute variable liveness. This is equivalent to assuming that all variables are live at the end of each basic block.

We may now deduce the cost of spilling for each category of value, for our target architecture. This cost is computed as follows —

- a locally-live, pending value must be saved to memory and reloaded prior to the next use. The incremental cost of freeing such a register is just the cost of the reload, as the value needed to be saved to memory anyway
- a locally-live, saved value does not require a memory write, but must be reloaded prior to the next use. The cost is the same as for the locally-live, pending case above
- a locally-live, volatile value must have a memory location allocated to it, be written to that location, and be reloaded prior to the next use. This is always the most expensive category of value to spill
- a locally-dead, pending value must be saved to memory. This frees the register at zero incremental cost, since the value had to be saved in any case, and no reload is required

- a locally-dead, saved value requires no action, since the value is already saved, and frees its register at zero cost
- a locally-dead, volatile value requires no action, and frees its register at zero cost

If information was available on the liveness of *variables* rather than liveness of register values, we could improve these cost estimates in the two pending value cases.

The spill cost information may be used to decide the order in which register values should be spilled to memory. Of course, the values with the lowest incremental cost of spilling should be spilled first.

Determining local liveness

We have so far avoided the question of the determination of local liveness properties. Since the context is on-the-fly allocation, we shall assume that instructions are being emitted immediately, and no buffering of instructions is performed. If this is the case, the only source of information on liveness of values is the shadow stack of the interpretive code generator. If the value is duplicated on the stack, then it is locally live, otherwise it is locally dead.

In our reference architecture, where we buffer the virtual assembly language at least one basic block at a time, we can do much better. In this case, the local common sub-expression analysis of the previous chapter may have found further uses for values which would have been discarded in analysis of shadow stack contents. We may therefore perform a backward scan through the buffer to determine local liveness. The next section shows that when this is the case, it is advantageous to allocate registers during the same backward scan.

10.3.2 Backwards allocation in VAL

Excellent register allocation may be obtained by assigning registers during a backward pass through the *VAL* buffer. In a later section we consider the extension of this method to global allocation.

After the *VAL* code for an extended basic block has been placed into a buffer it is possible to perform exactly optimal register assignment for the values in the block. The optimality result depends on a result derived from considering the coloring of the induced interval graph. No graph is actually constructed, the graph is only necessary to prove the optimality of the rule used in the assignment.

The algorithm uses a data structure which maps virtual to physical registers. For each virtual register encountered, the following information is required —

```
TYPE VRegInfo = RECORD
                  physReg : PhysReg; (* initially noReg *)
                  lastUse : BuffIdx; (* initially zero  *)
                  ...       (* other fields as required *)
                END;
```

The information shown is all that is required for typical *RISC* machines. For machines with different classes of register, it may be necessary to define and include a set of *value-uses* so that when the allocation is done it is known which class of register must be used. This information might include predicates such as *isUsedAsIndexRegister*, and so on.

The other required data structure is an array indexed on physical register ordinal, which records the next (that is next highest) index at which the physical register has a new value defined.

```
VAR  nextDef = ARRAY PhysReg OF BuffIdx; (* initially maxIx *)
```

The data in both of these structures is computed during the backward scan of the buffer, and provide all of the information needed for an optimal assignment.

The algorithm works as follows —

1 starting from the final instruction in the block, each instruction is examined in turn

2 if the instruction is the defining occurrence for a particular virtual register (and by design the *only* defining occurrence) then a physical register is assigned to the *VReg* using the following rules, in order —

> **if** the defining instruction is a copy *from* some physical register, and that register is free throughout the live range, then assign that register to the *VReg*
>
> **else** find the physical register which has least *nextDef* value which is greater than *lastUse*

If the allocation succeeds, then the previous (that is, next higher) buffer index becomes *nextDef* for the assigned register. If the final step fails then no registers remain, and the value must be allocated a memory temporary, and accessed out of memory

3 if the instruction includes a first observation of a used occurrence of some virtual register then this is the final use of that *VReg*, and the *lastUse* field of the corresponding element of the map is set accordingly

4 if the instruction defines a new value for some physical register then mark this register as having a new value defined at the previous (that is, next higher) index

5 if the instruction is a procedure call, all physical registers which are threatened by the call are marked as having a new value defined at the previous index

Several points need to be made about this algorithm. Firstly, if it is the first *if-then*-step of the assignment algorithm which succeeds, then the copy operation is made redundant, and will be eliminated during the buffer writing operation.

The algorithm automatically caters for the case where a virtual register is copied *to* a physical register, as happens, for example, when a parameter value is copied to the designated parameter register. Since the destination register is defined at the index immediately following the copy, the destination register will automatically be selected by the second rule, unless the copy is not the final used occurrence of the *VReg*. If that is the case, then the copy is necessary, and must remain.

Finally, the consequences of failure of the allocation step must be considered. With typical *RISC* machines it is essentially impossible to run out of registers for expression evaluation alone. Because this algorithm allocates registers in backward order of definition point, short lived values whose live ranges are entirely overlapped by longer lived values are assigned registers first. Thus the longer lived values which arise from common sub-expression elimination and value tracking are the ones which fail to have registers assigned to them. Since all such values have had memory locations assigned to them, the values may be accessed out of memory. The process is as follows, for a load-store machine. Any occurrence of an unmapped virtual register, v_z say, as a destination operand of some instruction in the *VAL* —

`add` v_x,v_y,v_z ; *dst reg* v_z *is unmapped*

is translated into the two instruction sequence in the assembly language —

`add` r_2,r_3,r_{at} ; *copy to scratch reg at*
`sw` r_{at},-24(*fp*) ; *store to spill location*

where we have assumed that v_x was allocated r_2, v_y was allocated r_3, and the unmapped register v_z was allocated the spill location (*fp*-24).

The case of virtual source registers which do not have an assigned physical register is similar, so that if v_x in unmapped in —

`add` v_x, v_y, v_z ; v_x is unmapped

then the assembly language result will be —

`lw` -28(*fp*),r_{at} ; *load from spill location*
`add` r_{at},r_4,r_5 ; *into scratch register at*

Notice that even if all three registers of a binary operation are spilled, we shall only need two scratch registers, since the destination and one of the operands can use the same scratch register.

In the case of the *MIPS* architecture, the register *at*, also known to the assembler as r_1, is reserved as a scratch register. It is always safe to use this register to carry a temporary value between adjacent instructions. In other architectures it may be necessary to explicitly reserve a scratch register by removing it from the allocation pool.

Optimality of the method

`< still to come >`

10.4 Global register allocation

In our reference architecture, register allocation is performed on the *VAL*. We shall, in fact, assume that each virtual register in the *VAL* corresponds to a separate live range. Thus, although we refer below to live ranges, we may also view the process of register allocation as an n-to-k mapping of the n virtual registers to the k physical registers available on the real processor.

We shall assume that the code for the whole procedure is held in *VAL* form in some buffer, and that some explicit representation of the control flow graph is available. We further assume that definition and use information is precomputed for each virtual register, and that some estimate of the cost of spilling each virtual register to memory is precomputed.

10.4.1 Building the interference graph

Contemporary methods of global register allocation, make explicit use of the theory of graph coloring. In all variations of these methods an interference graph is constructed for all of the live ranges in a procedure. This is an undirected graph, which for simplicity we shall assume is represented as a two dimensional bit-matrix. In this matrix the element $I[i,j] = 1$ if the relation $\mathit{Interfere}(i,j)$ is true. We note that the interference graph, as an undirected graph, is necessarily symmetric. That is, for all i, j we have that $I[i,j] = I[j,i]$.

The interference graph must be computed from the live register information for each basic block. First however, we must formulate a precise definition of the informal concept of "live ranges overlap" which we have used so far.

Suppose the set of registers which are live at any point p in a control flow graph is denoted L_p. Thus —

$$r_i \in L_p \quad = \quad p \in \mathit{LiveRange}(r_i)$$

Definition: A register r_i which has a definition at point p in a control flow graph interferes with every register which is in L_p.

Note that this last definition does not give us *all* the interferences of r_i. Two registers r_i and r_j interfere if *either* at some point of definition of r_i we have $r_j \in L_p$ or at some point of definition of r_j we have $r_i \in L_p$. However, we need only use the above definition to insert edges into our graph provided that we maintain symmetry in the data structure by always including `I[i,j]` and `I[j,i]` at the same time.

The problem of computing the interference graph thus reduces to computing the live register set L_p for every point p in the control flow where a register is defined. The two edges corresponding to each definition are then added to the interference graph at each such definition point p.

The liveIn and liveOut register sets

We assume that for each basic block i we have generated two local register sets. The set $\mathit{localLive}_i$ is the set of all registers which are either used in block i but not defined in i, or are used in block i before they are defined. Similarly, the set $localDef_i$ is the set of registers which are defined in block i. Using these definitions we have the following equations —

$$\begin{aligned} LiveOut_{exit} &= \Phi \\ LiveIn_i &= localLive_i \cup (LiveOut_i - localDef_i) \\ LiveOut_i &= \bigcup_{j \in \mathrm{succ}(i)} LiveIn_j \end{aligned}$$

where the set of blocks *succ*(i) contains the successors of the control flow graph block i.

We must solve these equations to find the least fixed point. The standard, backward flow, all path solution method is sufficient.

Computing the live register set

Once the *LiveIn* and *LiveOut* sets are known for any block, we are able to compute the live register set for each location within that block.

We want to initialize the live register set to *LiveIn* for the block, and go sequentially through the block inserting edges in the interference graph. At each definition we insert interference edges between the defined register and every register in the current live set. We then add the newly defined register to the live register set.

If we wish to be truly accurate, we should also remove registers from the live register set at the end of live ranges which fall within a basic block. Finding the end of live ranges requires some further computation, and for this reason many implementations simply ignore this factor. Such a choice is always safe, but inserts *additional* edges in the interference graph.

Finding the points at which registers become dead poses different problems to finding the points at which registers become live. The points at which a register becomes live is immediately visible in the code of the block, since the newly live register appears as the destination operand of some *VAL* instruction. The point at which register becomes dead is not so apparent, since when we meet a register use during a forward scan through the block we do not know if it is the last use. We must thus annotate the block with information which indicates where registers become dead. This information may be computed as follows. The set of registers which become dead somewhere inside the block i is given by the *doomed* set —

$$doomed = (LiveIn_i \cup localDef_i) - LiveOut_i$$

During a backwards scan through the block we find the last reference to each of these registers on this control path (that is, the first use encountered during the backwards scan). As these last uses are found, the location in the basic block is marked, and the register is deleted from the *doomed* set. In order to account for the possibility that the same register might become live and then dead more than once in a single block, we add new registers to the *doomed* set as we meet their definitions in the backward scan. An outline of the algorithm is given

```
(* localLive, localDef, LiveIn and LiveOut are known *)
FOR each block i DO
  doomed := (LiveIn[i] + localDef[i]) - LiveOut[i];
  FOR every point p in block i, backwards DO
    IF destination reg at p is noReg OR
       destination reg d is not doomed THEN
      INCL(doomed,d);
      FOR every source reg s used at p with s ∈ doomed DO
        mark s as killed at p;
        EXCL(doomed,s);
      END;
    ELSE (* destination reg is doomed !  *)
      delete instruction at index p;
    END;
  END;
END;
```

Figure 10.5: Marking the end of live ranges

in figure 10.5. Note that in this algorithm instructions which compute doomed values are removed. This is just one more round of dead code removal.

The mechanism for computing the interference graph is given in figure 10.6. A later section discusses some of the implementation issues in more detail.

```
FOR each block i DO
  liveRegSet := LiveIn[i];
  FOR every buffer index p in block i, forwards DO
    liveRegSet := liveRegSet - regsKilledAt(p);
    FOR every reg r in liveRegSet DO
      insert edges between destination reg d and r;
    END;
    INCL(liveRegSet,d);
  END;
END;
```

Figure 10.6: Computing the interference graph

10.4.2 Chaitin's method

In an historically important paper Chaitin in 1983 described an efficient algorithm for coloring an interference graph. The algorithm attempts to perform a k-coloring, where k is the number of available registers. If the algorithm does not succeed, registers must be spilled, a new intererence graph constructed, and an attempt made to color the new graph.

In the interference graph, each node represents a live range. The *degree* of a node is the number of edges incident on the node. The central idea behind Chaitin's method is to remove nodes from the graph one by one. A node is *available* for removal as soon as it has less than k edges in the graph. As each node is removed all of its edges are removed, and the node is placed on a list. The removal of edges lowers the degree of adjacent nodes, perhaps making other nodes available for removal.

The process of node removal terminates in one of two ways. Either the graph becomes empty, or the algorithm becomes blocked with every remaining node having at least k-edges. If the algorithm becomes blocked then live ranges must be spilled, otherwise the graph may be k-colored.

When the graph becomes empty, the nodes are put back into the graph in the reverse order to removal. That is, nodes are put back into the graph in last out, first in order. As they are replaced, a register is allocated to the live range which the node denotes. Since nodes are only removed if they have less than k edges, it follows that when they are replaced they will again have less than k edges. Even if the adjacent nodes have all been allocated different registers they will have used up less than k registers, and so there must always be at least one free register for the newly inserted node.

In fact Chaitin's algorithm is conservative, since even if a node has exactly $(k-1)$ edges when it is removed from the graph, there may still be a choice of register when the node is replaced. This will happen, for example, if two or more of the $(k-1)$ adjacent nodes are colored with the same register.

```
PROCEDURE TryColoring(VAR success :  BOOLEAN);
  VAR n :  VRegister;
BEGIN
  MakeEmptyList(regList);
  WHILE interference graph is non empty DO
     IF any node has less than k edges THEN
       let n be any such node;
       AddToFront(regList,n);
       remove all graph edges leading to n;
     ELSE
        success := TRUE; RETURN;
      END;
   END;
   success := TRUE;
 END TryColoring;
```

Figure 10.7: Removing the nodes, one by one

Spilling registers in Chaitin's method

If Chaitin's algorithm becomes blocked then we are left with a graph in which each node has at least k edges. We must choose at least one live range to spill. If the live range corresponds to some program variable it may already have had some memory location allocated to it. Otherwise, the spilled range must be allocated some temporary location in the local stack frame. At this point it is necessary to rely on some kind of heuristic to determine which registers are the most useful to spill. We should take into account the degree to which each live range blocks other ranges from removal from the graph, and the cost of spilling each remaining range.

The residual graph in the blocked state gives useful information to help with the spilling decision. The number of edges left in the graph for any particular node tells us how many other nodes will have their degree lowered by the spilling of that register. We should also have some pre-computed estimate of the spill cost for each live range. Chaitin's original paper suggested the heuristic of spilling the range with the least quotient of spill cost to number of edges. This still appears to be an excellent choice.

We should continue to spill ranges from the graph until some remaining node has its degree lowered to less than k. The removal process then resumes until the process either becomes blocked once again, or the graph becomes empty.

When the graph has become empty after one or more ranges being spilled, the code buffer needs to be rewritten to take into account the spilled ranges. All definitions of the live range must be written to the allocated spill location, and every used occurrence of the live range must be loaded from the spill location. It should be noted that the spilling of a live range often *increases* the number of virtual registers in the *VAL*. This is because (at least for *RISC* machines) every mention of the spilled virtual register will be replaced by a definition or use of some new, temporary register which is either stored or loaded from the spill location. Each of these new virtual registers will interfere with all of the other registers in the live register set at the point of their brief lifetime. Because of this factor, there seems to be no good way of incrementally computing the new interference graph from the previous, unspilled version. We therefore recompute the live register sets, and the new interference graph, and then make another attempt at coloring the new graph. It is possible but unusual for a second round of spilling to take place if there the additional temporary registers cannot be colored in the new graph.

Variations on Chaitin's method

Since the original paper, a number of small improvements on Chaitin's method have been proposed. The most important of these is Briggs' variation.

It has already been noted that the basic method may be overly pessimistic in that it assumes that if a live range has $(k-1)$ neighbours in the interference graph then all neighbours will have different colors. It turns out the "airline booking method" of choosing k larger than the actual number of registers is too dangerous. However, Briggs came up with a clever algorithm which finds just

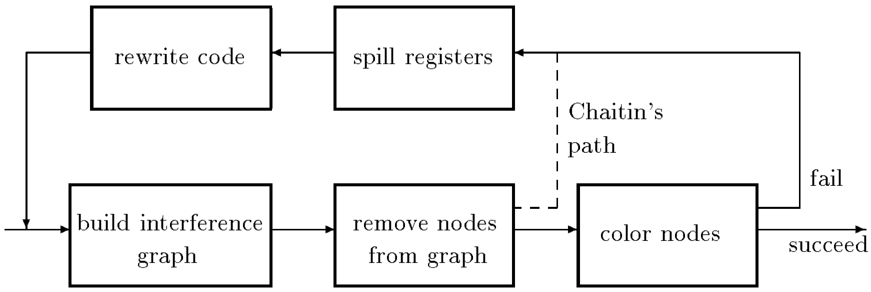


Figure 10.8: Figurative representation of Briggs' and Chaitin's method

those cases for which the original method is pessimistic.

The key element in Briggs' method is that registers are spilled during coloring, and not during node removal from the graph. Recall that in Chaitin's method, when every remaining node has degree k or more the algorithm blocks and one or more nodes must be spilled. In Briggs' method the removal of nodes continues anyway, until the graph is empty. When the coloring step is performed, nodes with edges to too many other nodes may fail to get a color allocated to them *and are spilled at that stage*. As always, if nodes have been spilled, the code must be rewritten and the process repeated with the new interference graph. This process is represented by the figure 10.8. In this figure, note the dotted line, which is the path which is followed in Chaitin's method if the node removal fails. In Briggs' method, the solid path is followed in the case that the *coloring* step fails.

It is important to order the removal of nodes in Briggs' method so that some control is exercised over the likely registers that will be spilled. Briggs suggests ordering the removal of nodes which have degree larger than k using the same heuristic as Chaitin used for choosing nodes to spill.

Another variation on Chaitin's method which is of importance is "heirarchical graph coloring". In this method an attempt is made to color the interference graph arising from a subgraph of the control flow graph, which corresponds to the most frequently exercised paths in the procedure. This coloring is extended by successively adding parts of the control flow graph corresponding to less exercised paths, until finally the interference graph of the whole control flow graph has been colored.

10.5 Implementing graph coloring

10.5.1 Choosing data structures

The main data structures required for graph coloring allocators, are the interference graph, and various register sets and lists. The operations which must be performed on the register sets are insertion, deletion, and the usual set operations. The interference graph is relatively sparse, and requires edge insertion and deletion operations, and the counting of edges of particular nodes.

The number of live ranges in a procedure is not known until compile time, and may become quite large. One of the subroutines in the *SPEC-92* benchmark suite has almost 5000 live ranges in it, for some machine architectures.

Bit-vector implementations

One of the possible choices of data structure is bit-vectors, and bit-matrices. In this case, register sets must be dynamically allocated, and the size can be frozen once the number of live ranges is known for each particular procedure.

Such a choice has the advantage of simplicity, but leads to some operations having non-linear complexity, as a function of N the number of live-ranges. This somewhat suboptimal behaviour may still be tolerable, provided the average case of small procedures is handled with low overhead.

In effect, the runtime type of the register sets is —

$$\textbf{type } \mathit{VRegSet} = \textbf{pointer to array } [0 \mathinner{..} \mathit{sSiz}] \textbf{ of } \mathit{BITSET};$$

where *sSiz* is the first integer equal to or larger than N divided by *bitsInWord*.

The inclusion and exclusion of elements in such bit-vectors is obviously order $O(1)$. Set union and intersection are performed by looping over the words in the set representation and using the word-wide, bitwise Boolean operations. The complexity is order $O(N)$. Note that this complexity depends on the cardinality of the universal set N, and not on the cardinality of the actual operand sets. Although the sets may be quite sparse, the efficiency of performing the operations 32 or 64 bits at a time is compelling.

Using a bit-matrix adjacency representation for the interference graph can lead to very large data structures. A procedure with 5000 live ranges will require a dynamically allocated block of memory with 25×10^6 bits, or about 3 Mega-Bytes. Furthermore, in order to allow quick indexing it is necessary to create a *dope-vector* of pointers into the matrix, pointing to the beginning of each row. Thus the matrix may be treated as an **array** $[0 \mathinner{..} N-1]$ **of** *VRegSet*. As well as facilitating indexing, such a representation allows quick extraction of the interferences of any given register.

With this representation, we again consider the construction of the interference graph, refining the algorithm in figure 10.6. Firstly, the set difference at line 4,

```
4      liveRegSet := liveRegSet - regsKilledAt(p);
```

is easily performed by one or two set exclusion operations. The insertions in the for loop on lines 5–7, is not so straightfowrard however.

```
5    FOR every reg r in liveRegSet DO
6       insert edges between destination reg d and r;
7    END;
```

Given that the bitvector *liveRegSet* is a dense representation of a sparse set, it makes sense to quickly deal with zero words in the set —

```
5a    FOR wIx := 0 TO sSiz DO
5b      IF liveRegSet^[wIx] <> BITSET{} THEN
5c        FOR bIx := 0 TO bitsInWord-1 DO
5d          IF bIx IN liveRegSet^[wIx] THEN
6a            r := wIx * bitsInWord + bIx;
6b            add the edge (d, r);
7d          END;
7c        END;
7b      END;
7a    END;
```

where the add-edge operation sets *two* bits in the symmetric bit matrix.

When the bit manipulations are expanded out, the nested loops in this code may be very heavily optimized. However, the execution of this nested loop for every single instruction in the buffer is rather time consuming.

An alternative approach, which may be more efficient on some architechtures, is to insert just one of the bits for each interference, and restore the symmetry of the bit-matrix in a single scan over the elements after all insertions have been made. In this case the code simply replaces the d-th row of the bit-matrix by the union of the d-th row and the live register set.

```
5a    FOR wIx := 0 TO sSiz DO
6a      row[d]^[wIx] := row[d]^[wIx] + liveRegSet^[wIx];
7a    END;
```

Restoring symmetry to the graph requires a nested loop over all matrix elements.

Using this representation leads to very low overhead for small values of N, but is asymptotically rather slow. For job mixes which involve many large procedures with very many live ranges the sparcity of the sets suggests that a *dual* representation for the live-register-set, using a bit-vector *and* a list is preferable.

One of the other performance issues for graph coloring involves tracking the number of remaining graph edges in particular rows of the graph. There are at least two approaches to this, here is one of them.

Firstly, the number of edges for each node n is determined by counting e, the number of "1" bits in the corresponding row of the bit-matrix. Those nodes which have less than k edges are placed on an array of lists indexed on e. These nodes are removed in order of increasing e, and a set of removed nodes "*removed*" is incrementally updated. We then loop over the remaining nodes re-evaluating the edge counts, and removing nodes for which the edge count has fallen below k, as shown here —

```
(* remove remaining nodes *)
WHILE (removed <> allRegs) AND not-blocked DO
   FOR every remaining node n DO
     IF Edges(row[n] - removed) < k THEN
       put n on list, and include in removed;
     END;
   END;
END;
```

This algorithm looks as though it might evaluate the number of edges many times, but in practice the number of evaluations for moderate N is typically $(1.5 \times N)$ and almost never more that $(3.0 \times N)$. The reason is that once the low edge nodes have been removed, many of the remaining nodes have already dropped below k.

The alternative strategy, counting edges once and adjusting the counts as adjacent nodes are removed, is much less favorable. Over the same test suite of procedures the edgecount of typical nodes is decremented about 10 times, and the operations are more costly.

Counting the edges

Counting the edges in a bit-vector consists of adding up the number of "1" bits in all the word-sized bitsets in the vector. The efficiency depends critically on the algorithm which is used to count "1" bits in each word. Since no contemporary machines offer a single instruction to do this, it is worth noting that there are two separate, efficient ways of doing this. Neither of these is exactly obvious, and both of them are much superior to a mindless iteration over the bits of the word.

The more straightforward way of performing the count is to construct a byte lookup table `map`[0..255], which is initialized so that `map[`n`]` contains the number of "1" bits in the binary representation of n. Then at runtime we compute the number of "1" bits in a word by —

```
(* pattern is a word-sized union *)
IF pattern.set <> emptySet THEN
   INC(count,map[pattern.byt[0]]); (* bits in byte-0 *)
   INC(count,map[pattern.byt[1]]); (* bits in byte-1 *)
   INC(count,map[pattern.byt[2]]); (* bits in byte-2 *)
   INC(count,map[pattern.byt[3]]); (* bits in byte-3 *)
  (* more for eight-byte words... *)
END;
```

The four inner lines compile down to just 17 instructions on a typical load-store architecture, but include 8 memory references. On some machines fetching the whole word once, and shifting and masking to obtain the four index bytes would be better. In this case we would have three extra instructions, but only four memory references.

The second method is rather more obscure. Figure 10.9 is an assembly language version in *SPARC* assembler. The method works by first separating the odd and even bits of the word by shifting and masking. The two words are then added. Because of the shift, there is no carry propagation, and we are left with an array of 16 bit-pairs, each of which can only be in the range 0 to 2.

The algorithm now shifts and masks the new word so as to separate the odd and even bit-pairs, which are then added. Once again there is no carry propagation, and we are left with eight bit-quads, each of which can only be in the range 0 to 4.

```
; word is in register i0 to start
      set   0x55555555,%g1     ; load up mask1
      shr   %i0,1,%i1          ; shift by one bit
      and   %i0,%g1,%i0        ; odd numbered bits
      and   %i1,%g1,%i1        ; even numbered bits
      add   %i0,%i1,%i0        ; array of 16 bit-pairs
; each bit-pair has value 0,1 or 2
      set   0x33333333,%g2     ; load up mask2
      shr   %i0,2,%i1          ; shift by two bits
      and   %i0,%g2,%i0        ; two bits at a time
      and   %i1,%g2,%i1        ; two bits at a time
      add   %i0,%i1,%i0        ; array of 8 bit-quads
; each bit-quad is in range 0..4
      set   0x0f0f0f0f,%g3     ; load up mask3
      shr   %i0,4,%i1          ; shift by four bits
      and   %i0,%g3,%i0        ; four bits at a time
      and   %i1,%g3,%i1        ; four bits at a time
      add   %i0,%i1,%i0        ; array of 4 octets
; each octet is in range 0..8
      set   0x00ff00ff,%g4     ; load up mask4
      shr   %i0,8,%i1          ; shift by eight bits
      and   %i0,%g4,%i0        ; eight bits at a time
      and   %i1,%g4,%i1        ; eight bits at a time
      add   %i0,%i1,%i0        ; array of 2 half-words
; each half-word is in range 0..16
      shr   %i0,16,%i1         ; shift by sixteen bits
      and   %i0,31,%i0         ; mask lower count
      add   %i0,%i1,%i0        ; final count!
; %i0 is in the range 0..32
```

Figure 10.9: Counting the bits in a word, hierarchical method

Continuing hierarchically, we finally obtain the bit count after four and a half steps. In the version given here, there are 23 instructions[3], but no memory accesses. On machines which allow 32-bit literals, this instruction count would come down to just 19. In any case, if these instructions are the kernel of a loop which iterates over all the words in a large set, the mask words are loop invariant, and the number of instructions per bitset is 19.

There is little to choose between these two methods for 32-bit architectures, but note that for 64-bit architectures the first method uses twice as many instructions, while the second uses just one extra step of four instructions.

Range merging and copy elimination

As well as the obvious purpose, register allocation also has the task of redundant copy elimination. Particularly in the case of *VAL* arising from *SSA* there are

[3] Well, to be strict, the `sets` should count as two each.

many register-to-register copies, and we would like to eliminate as many of them as possible.

The classical method of doing this is to make a *range merging* pass over the code buffer *after* the interference graph has been constructed. For every register-to-register copy, if the source and destination registers do not interfere, all uses of the destination are replaced by uses of the source. As well, the source register node has all interferences of the destination added to its edge set. The now reduced number of nodes are colored in the normal way.

An alternative is to color the original graph, but build an heuristic into the allocator which attempts to choose colors which eliminate copies whenever possible. This may involve keeping for each register a list of registers which are connected by copy instructions. Such an algorithm may sometimes fail to eliminate all copies. Nevertheless, it has the desirable property that when registers do run out, live ranges are automatically split at the joins, and a *chameleon coloring*[4] is attempted, before a joined-up live range is spilled. This strategy seems to behave particularly well in the case of machines with very few physical registers.

Coloring the list

Once all of the nodes have been removed from the graph, the graph must be reconstructed node by node, and colors (physical registers) chosen for each node. The constraint is that the register chosen for the latest node be separate to all of the registers which have been allocated to the other nodes which have been colored already, and which interfere with the latest node. Essentially, a mapping from the interfering virtual register set to the unavailable physical register set must be constructed. With our bit-vector representation an $O(N)$ scan over the bit-vector seems unavoidable. Suppose that *replaced* is the set of virtual registers already allocated a register and replaced in the graph.

```
(* calculate set of infeasible physical registers pRegClash *)
(* given the set of vRegs already assigned to physical registers *)
pRegClash := PhysRegSet{};
vRegClash := Intersection(row[n],replaced);
 FOR wIx := 0 TO sSiz DO
   IF vRegClash^[wIx] <> BITSET{} THEN
     FOR bIx := 0 TO bitsInWord-1 DO
       IF bIx IN vRegClash^[wIx] THEN
         r := wIx * bitsInWord + bIx;
         INCL(pRegClash,PhysRegOf(r));
       END;
     END;
   END;
 END;
```

Here, we compute the set of clashing virtual registers as the intersection of the replaced set with the set of *all* interferences of register n, $row[n]$. We assume that

[4] A chameleon coloring changes color within the same live range. For almost all machines it is better to perform a register-to-register copy, rather than to spill and restore a register.

the function *PhysRegOf* returns the physical register allocated to a particular virtual register in the replaced set.

Estimating spill cost

If graph coloring methods do not find a spill-free allocation, then one or more live ranges must be chosen for spilling. In Chaitin-style allocators this means spilling the whole of the chosen ranges. In order to make a good choice it is necessary to estimate of the cost and benefit of spilling every candidate range. The choice should add as little as possible to the runtime cost, by way of extra execution cycles, and should reduce the edge count on as many other nodes as possible.

The runtime spill cost consists of two components: the actual cost of the extra instruction cycles which will need to be executed, and the relative frequency of those instructions. It is almost certainly much more costly to spill a register which is used inside a loop than a register which is not used in a loop. Thus some kind of heuristic for estimating execution frequency is required. The simplest methods just count the loop nesting level, and presume that each loop is executed some moderate number of times.

Estimating the runtime cost of spilling is quite straightforward for load-store architectures, since the number of extra instructions is known. In particular, the cost of spilling a particular register can be calculated from the *VAL*, and is independent of any other spilling decisions.

The situation is much more difficult for machines which allow operands to be used from memory. As an example, consider the cost of spilling a register used in a compare "`cmp`" instruction in the *Intel-x86* architecture. Either one of the operands can be taken from memory without any extra instructions, so the additional overhead comes only from the slightly decreased code density, the the poorer cache performance which may result. However, if *both* of the operand registers become spilled, then the cost amounts to a whole extra instruction per comparison. Thus the cost of spilling a particular register depends on which other registers have been spilled already. In spite of this, the compile-time effort required to recompute the spill costs after each spill decision seems prohibitive, so some inaccuracy must probably be tolerated for such architectures.

10.6 Finding out more

This chapter has concentrated on just one of the methods of graph coloring allocation, Chaitin's method and its extensions. It has also concentrated on just one of the simple implementation possibilities, using bit vectors.

`<< cite Chaitin, and Briggs papers >>`

`<< cite refs for priority coloring (Chow and Hennessy), plus a selection of hierarchical coloring papers>>`

`<< give a starting point for interprocedural allocation refs >>`

`<< pointer to Patterson's work on estimating execution density >>`

10.7 Exercises

10.1

10.2

Chapter 11

Instruction Scheduling

Chapter 12

Writing Assembly Language

Appendix A

DCode Reference Manual

A.1 Overview

DCode is a stack based language for an abstract stack machine — the *D-Machine*. This machine is similar, in general terms to the well known P-Code[?] and U-Code[?] abstract machines. The main differences are due to the lower level at which address expressions are treated.

DCode has three components: interface information, data declarations, and procedure headers and code. The first part declares imports and exports of global symbols. The data declarations define the statically allocated data. Then for each procedure there is a header, with local declarations, and the code of that procedure's body. The instructions act on an abstract stack, pushing new values, popping values, and operating on the value(s) on the top of the stack. The abstract machine is a so-called "zero address machine". The only instructions which take memory addresses as arguments are those which push addresses onto the abstract stack, and those that call procedures at particular symbolic locations.

Values on the abstract stack are typed, but have only three types: *words, floats* and *doubles*. Objects of any primitive machine type may be pushed or popped. Operations such as `add` exist in different forms for each of these types. In the case of whole number arithmetic, and conversions between numeric formats, the overflow trapping mode, if any, is explicitly specified.

The *DCode* form described in this appendix is conventionally read and written to an ordinary *ASCII* file, and is designed to be readable by humans, as well as by language processors. However, it may be also used as the definition for procedural interface, as shown in figure 4.1.

A.2 Lexical issues

In the *ASCII* representation of *DCode* line breaks are significant, and mark the end of various structures. Comments, and other white space has no syntactic significance, and may appear between any tokens. The other elements of the lexis

$$
\begin{array}{rcl}
\textit{alpha} & = & \{\text{`a' .. `z', `A' .. `Z', `\$', `\_'}\} \\
\textit{term1} & = & \{\text{`"'}, \textit{eoln}\} \\
\textit{term2} & = & \{\text{`''}, \textit{eoln}\} \\
\textit{digit} & = & \{\text{`0' .. `9'}\} \\
\textit{octDigit} & = & \{\text{`0' .. `7'}\} \\
\textit{hexDigit} & = & \textit{digit} \cup \{\text{`A' .. `F'}\} \\
\textit{alphanum} & = & \textit{alpha} \cup \textit{digit} \\
 & & \\
\text{ident} & \rightarrow & \textit{alpha}\ \textit{alphanum}^{*}\ . \\
\text{litstring} & \rightarrow & \text{`"'}\ \textit{any} - \textit{term1}\ \text{`"'} \mid \text{`''}\ \textit{any} - \textit{term2}\ \text{`''}\ . \\
\text{relop} & \rightarrow & \text{`}\texttt{<}\text{'} \mid \text{`}\texttt{<=}\text{'} \mid \text{`}\texttt{>}\text{'} \mid \text{`}\texttt{>=}\text{'} \mid \text{`}\texttt{=}\text{'} \mid \text{`}\texttt{<>}\text{'} \mid \text{`}\texttt{\#}\text{'}\ . \\
\text{number} & \rightarrow & [\text{`}-\text{'} \mid \text{`+'}]\ \textit{digit}^{+} \mid [\text{`}-\text{'} \mid \text{`+'}]\ \textit{octDigit}^{+}\ \text{`B'} \\
 & & \quad \mid [\text{`}-\text{'} \mid \text{`+'}]\ \textit{digit}\ \textit{hexDigit}^{*}\ \text{`H'}\ .
\end{array}
$$

Table A.1: Character sets, and regular expressions for lexical categories.

are the keywords, identifiers, numbers and various other markers. Comments start with a semicolon, and end at the end of a line. As in Modula, character case is always significant. The lexical categories are defined in the table A.1

By convention, comments which start with the exclamation character `!` are treated by code generators as pragmas. On backends which support this feature, the remainder of the line is copied to the output file in a backend-defined manner.

Identifiers obey the Modula conventions, except that dollar signs are allowed as valid characters, and lowline (underscore) characters may be freely used. By convention, frontends will prepend an underscore to all user-defined identifiers from the source code. Internally generated identifier names which do not have an underscore can therefore not collide with any user-defined external name. For systems in which the linker conventions do not use the prepended underscore the backend must strip this character from identifiers, before emitting the identifiers to the assembler.

Labels are lexically the same as identifiers, and defining occurrences are followed by a colon character. By convention, loop header labels are marked with the directive *.LOOP* in front of the label. Similarly, the ends of loops are marked with an *.ENDLOOP* directive. Simple backends may rely on this convention, see section A.6.

Literal strings have the same format as in Modula, and cannot extend beyond a line end. They are enclosed in matching quote characters, which may be either single quotes '`'`' or double quotes '`"`'.

Some instructions have *relop* or *mode* arguments. As in Modula, the relops are "`<, <=, >=, >, =, <>, #`". For whole number arithmetic operations, and for numeric conversions, the mode may be *noTrap, crdOver, intOver* designating no overflow trap, unsigned, or signed trapping respectively. When the mode argument is optional, the default is always *noTrap*. In the case of division and remainder operations, the trapping mode is mandatory, as the signedness of the operators must be known in order to define the semantics of the operation.

Numbers are optionally signed, and obey Modula-2 lexical conventions. Numbers must begin with a decimal digit after the optional sign. Hexadecimal numbers end with the character "H", while octal numbers end with a "B". Decimal numbers require no suffix.

Starting with version 2.0 it is possible to include files verbatim in the *DCode* file. The format is —

`#include` *litstring*

The hash character must be the first character in the line, and the filename must appear as a literal string, that is, in single or double quotes. Only one level of inclusion is permitted.

A.2.1 Reserved words and predeclared identifiers

The keywords of *DCode* all begin with a period '.' and use all upper case characters. The keywords are—

.ADRS	.ALIGN	.ASCII	.ASCIIZ	.ASSEMBLY	.BITS16
.BITS32	.BYTE	.CHECK	.CONST	.COPY	.DATA
.DISPLAY	.DOUBLE	.ENDLOOP	.ENDP	.ENTRY	.EXCEPT
.EXPAND	.EXPORT	.FILE	.IMPORT	.JUMPTAB	.LOCAL
.LOOP	.NODISPLAY	.NOCHECK	.OPENCOPY	.PROC	.RETCUT
.RETRY	.SIZE	.TITLE	.TRAP	.VAR	.WORD
crdOver	fpParam	intOver	noTrap		

The identifiers in the last line in the table are predeclared identifiers. These are context sensitive marks rather than reserved words. However, because of the underscore convention for user-defined identifiers they should never clash with other identifiers.

A.3 File syntax and semantics

The conventional filename extension used for *ASCII DCode* files is "`.dcf`". The format of these files is described by the extended *bnf* in the table A.2.

A.3.1 Primitive data types

DCode knows of nine different primitive data types[1]. Some of these are absolutely sized, so that it is the responsibility of every frontend to map the logical types of its source language into the types supported by *DCode*.

The types are *unsigned byte*, *signed byte*, *unsigned 16-bit word*, *signed 16-bit word*, *unsigned 32-bit word*, *signed 32-bit word*, *word*, *float*, and *double*. The *word* type is the "natural" word size of the machine, while other fixed types may be shorter.

[1]Later revisions of the *DCode* form extend the instruction set to include 64-bit integer operands. The latest revision is available by ftp[?].

<table>
<tr><td>file</td><td>→</td><td>Title Declarations eofSy.</td></tr>
<tr><td>Title</td><td>→</td><td>.TITLE ident eolnSy .FILE litstring eolnSy.</td></tr>
<tr><td>Declarations</td><td>→</td><td>{Exports | Imports} {DataOrProc}.</td></tr>
<tr><td>Exports</td><td>→</td><td>.EXPORT NameList eolnSy.</td></tr>
<tr><td>Imports</td><td>→</td><td>.IMPORT NameList eolnSy.</td></tr>
<tr><td>NameList</td><td>→</td><td>ident [':' number] {',' [eolnSy] ident [':' number]}.</td></tr>
<tr><td>DataOrProc</td><td>→</td><td>.CONST [number] eolnSy {Label {ConstDec}}</td></tr>
<tr><td></td><td>|</td><td>.DATA [number] eolnSy {Label {ConstDec}}</td></tr>
<tr><td></td><td>|</td><td>.VAR eolnSy {Label VarDec}</td></tr>
<tr><td></td><td>|</td><td>[.LOCAL] .PROC HeaderInfo .ENTRY eolnSy
{Statement} {JumpTable} .ENDP eolnSy.</td></tr>
<tr><td>VarDec</td><td>→</td><td>.BYTE number [.ENTRY number] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.BITS16 number [.ENTRY number] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.BITS32 number [.ENTRY number] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.WORD number [.ENTRY number] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.DOUBLE number [.ENTRY number] eolnSy.</td></tr>
<tr><td>ConstDec</td><td>→</td><td>.BYTE number {',' number} eolnSy</td></tr>
<tr><td></td><td>|</td><td>.BITS16 number {',' number} eolnSy</td></tr>
<tr><td></td><td>|</td><td>.BITS32 number {',' number} eolnSy</td></tr>
<tr><td></td><td>|</td><td>.WORD number {',' number} eolnSy</td></tr>
<tr><td></td><td>|</td><td>.ASCII litstring eolnSy</td></tr>
<tr><td></td><td>|</td><td>.ASCIIZ litstring eolnSy</td></tr>
<tr><td></td><td>|</td><td>.ADRS ident [number] eolnSy.</td></tr>
<tr><td>HeaderInfo</td><td>→</td><td>ident "(" Arg {',' Arg} ")"eolnSy {CopyInfo} {LocalInfo}.</td></tr>
<tr><td>Arg</td><td>→</td><td>.NODISPLAY | .DISPLAY : number | .NOCHECK | .CHECK</td></tr>
<tr><td></td><td>|</td><td>.SIZE = number | .ASSEMBLY = number | .RETCUT = number.</td></tr>
<tr><td>LocalInfo</td><td>→</td><td>.LOCAL ident number ',' number
"(" number ',' number ',' number ")" ["fpParam"] [litstring] eolnSy.</td></tr>
<tr><td>CopyInfo</td><td>→</td><td>.COPY number ',' number .SIZE number [Align] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.EXPAND ['('number')'] number ',' number .SIZE number [Align] eolnSy</td></tr>
<tr><td></td><td>|</td><td>.OPENCOPY ['('number')'] number .SIZE number [Align] eolnSy</td></tr>
<tr><td>JumpTable</td><td>→</td><td>.JUMPTAB Label eolnSy {IdList eolnSy}.</td></tr>
<tr><td>IdList</td><td>→</td><td>ident {',' [eolnSy] ident}.</td></tr>
<tr><td>Statement</td><td>→</td><td>[LabelTag] Label [OpCode [OpArg]] eolnSy</td></tr>
<tr><td></td><td>|</td><td>OpCode [OpArg] eolnSy</td></tr>
<tr><td></td><td>|</td><td>Directive eolnSy.</td></tr>
<tr><td>LabelTag</td><td>→</td><td>.EXCEPT | .LOOP | .RETRY .</td></tr>
<tr><td>Directive</td><td>→</td><td>.TRAP ident',' ident {',' OpArg}</td></tr>
<tr><td></td><td>|</td><td>.ENDLOOP [number] .</td></tr>
<tr><td>Label</td><td>→</td><td>ident ':' .</td></tr>
<tr><td>OpArg</td><td>→</td><td>number {',' number} ["fpParam"]</td></tr>
<tr><td></td><td>|</td><td>mode | relop</td></tr>
<tr><td></td><td>|</td><td>ident [number].</td></tr>
<tr><td>Align</td><td>→</td><td>.ALIGN number .</td></tr>
</table>

Table A.2: The extended bnf for the input file.

A.3.2 The title

The title and file declarations specify the name of the module and the source file name from which it is derived. The title declaration has no particular function, except for identification purposes for the human reader. In contrast, the *.FILE* declaration is assumed to be the name of the source file, and should be written to the object file by all backends, for the benefit of debugger tools.

A.3.3 Declarations

Imports and exports

The declarations part of the file begins with imports and exports. Exported names are visible to the linker, while imported names are expected to be resolved to locations external to the module.

$$\begin{array}{rcl}
\text{Declarations} & \rightarrow & \{\text{Exports} \mid \text{Imports}\}\ \{\text{DataOrProc}\}. \\
\text{Exports} & \rightarrow & \text{.EXPORT NameList eolnSy.} \\
\text{Imports} & \rightarrow & \text{.IMPORT NameList eolnSy.} \\
\text{NameList} & \rightarrow & \textit{ident}\ [\text{':'}\ \textit{number}]\ \{\text{','}\ [\text{eolnSy}]\ \textit{ident}\ [\text{':'}\ \textit{number}]\}.
\end{array}$$

The elements of name lists are comma separated, and may contain embedded end of lines. The optional colon and number give the storage size of the imported or exported object. A zero number has the default semantics. Knowing the size of imported objects is important for machines with conventions which place small objects in a separate data segment.

Data declarations

Variables and constant declarations appear after imports and exports, and may alternate freely with procedure declarations.

Data may be declared in three ways. Constant declarations declare values which, where possible, will be placed by backends in a readonly memory segment. Data declarations using the *.DATA* keyword are initialized variables, and will be placed in a read-write memory segment. Finally variable declarations request uninitialized space, conventionally in the *BSS* memory segment.

$$\begin{array}{rcl}
\text{DataOrProc} & \rightarrow & \text{.CONST}\ [\textit{number}]\ \text{eolnSy}\ \{\text{Label}\ \{\text{ConstDec}\}\} \\
 & \mid & \text{.DATA}\ [\textit{number}]\ \text{eolnSy}\ \{\text{Label}\ \{\text{ConstDec}\}\} \\
 & \mid & \text{.VAR eolnSy}\ \{\text{Label VarDec}\}
\end{array}$$

Constant and initialized data declarations may be specified in any convenient way. Backends usually ensure that labels are word-aligned, so that the data may be specified byte-by-byte, or in larger units. There is no provision for handling floating point representations, so that frontends must determine the representation of floating point constants.

Note that multiple labels may be interspersed within declarations of constant or initialized data blocks. The optional number, if present, gives the total size of the following block.

ConstDec	$\rightarrow$	.BYTE *number* {',' *number*} eolnSy
	|	.BITS16 *number* {',' *number*} eolnSy
	|	.BITS32 *number* {',' *number*} eolnSy
	|	.WORD *number* {',' *number*} eolnSy
	|	.ASCII *litstring* eolnSy
	|	.ASCIIZ *litstring* eolnSy
	|	.ADRS *ident* [*number*] eolnSy.

The *.ADRS* declaration declares an symbolic address datum. The actual value will be filled in after compilation, probably by the linker. The optional *number* field allows symbolic addresses which are at a fixed offset from some labelled location.

The *.ASCII* declaration declares an initialized string, while the *.ASCIIZ* declaration appends a nul (zero) character to the literal string which is specified.

Variable declarations do not define the values of the bytes or words which are specified, but rather specify the *number* of units which are required. Note that a label is mandatory with each variable declarations, but multiple variables may be declared following the *.VAR* keyword.

VarDec	$\rightarrow$	.BYTE *number* [.ENTRY *number*] eolnSy
	|	.BITS16 *number* [.ENTRY*number*] eolnSy
	|	.BITS32 *number* [.ENTRY*number*] eolnSy
	|	.WORD *number* [.ENTRY *number*] eolnSy
	|	.DOUBLE *number* [.ENTRY *number*] eolnSy.

Variable declarations guarantee that a contiguous region of memory of the specified size will be allocated in the one memory segment. However, backends need not guarantee that variables will be set out in memory in the order in which they are declared in the *DCode*. Thus, if it is required that the label be placed in the interior of a region of memory, then the optional form with an *.ENTRY* declaration must be used. The entry declarations specifies the distance in bytes from the start of the contiguous region at which the label will be placed.

A.4 Procedure declarations

Procedure declarations consist of the procedure header, local declarations, the procedure body, followed perhaps by jump tables for case or switch statements.

DataOrProc	$\rightarrow$	... | [.LOCAL] .PROC HeaderInfo .ENTRY eolnSy
		{Statement} {JumpTable} .ENDP eolnSy.

Procedure declarations begin with the marker keyword *.PROC*, and end with the keyword *.ENDP*. The header has the declared name of the procedure (which may or may not correspond to an exported name).

The *.PROC* keyword may be preceded by the optional keyword *.LOCAL*. If present, this marker states that the procedure is neither exported or assigned as a procedure variable. For some machine architectures faster calls may be made to such procedures.

A.4.1 Procedure headers

HeaderInfo → ident "(" Arg {',' Arg} ")" eolnSy {CopyInfo} {LocalInfo} .

Procedure headers declare various attributes of the procedure which have been evaluated by the frontend. There are three parts, the header proper, copy information for any value parameters passed by reference, and local variable information.

The header information

The arguments of the procedure header specify properties of the procedure prolog and epilog.

Arg → .NODISPLAY | .DISPLAY : *number* | .NOCHECK | .CHECK
| .SIZE = *number* | .ASSEMBLY = *number* | .RETCUT = *number*.

The conventional method used by *DCode* compilers to access uplevel data uses a display vector. The keywords *.DISPLAY, .NODISPLAY* indicate whether or not a display location needs to be updated. In the case that the display is needed the level is indicated by the notation—

`.DISPLAY` : *lexLevel*

Compilation unit bodies are at lexical level 0, and require no display. The default is no display, which in turn is the same as display at level zero. If a procedure has no objects which are accessed from nested scopes, then no display updating is required, and it is permissible to declare *.NODISPLAY*.

The keywords *.CHECK, .NOCHECK* in the procedure header indicate whether or not a stack overflow check is to be performed. Checking is the default, so the check keyword is usually included simply for clarity.

The *.SIZE = number* phrase declares the size of the (fixed part of the) stack-frame. The number following the equality is the size in bytes. The default size is zero. The size includes any callee copied structures of fixed size, but does not include conformant arrays.

The *.ASSEMBLY = number* phrase declares the size of the parameter assembly area in the stack frame. This is not needed for architectures where parameters are pushed onto a runtime stack. The number following the equality is the size in bytes. The default size is zero.

The *.RETCUT = number* phrase declares that this procedure uses "Pascal call conventions". Such procedures are expected to remove *number* bytes from the runtime stack during the exit epilog.

Local variable information

Local variables are declared by specifying their identifier, stack frame offset, size, and other attributes. The first number following the identifier is the stack frame offset, while the second is the size of the variable in bytes.

LocalInfo → .LOCAL *ident number* ',' *number*
"(" *number* ',' *number* ',' *number* ")" ["fpParam"] [*litstring*] eolnSy.

The attribute information is a triple of numbers each of which may be only a single zero or one digit. In the first position a one specifies that the corresponding variable is accessed by a nested procedure. In the second position a one specifies that the corresponding variable has its value threatened[2] by a nested procedure. The final attribute in the parentheses is one to specify that the variable has its address taken within this procedure. Conventionally, structures and unions are given a one attribute in the final position to ensure that the backend will not attempt to place such a variable in a register.

The optional *fpParam* marker specifies that the variable is a floating point datum. This marker should be generated by any frontends which are intended to be portable, since some architectures will be unable to locate floating point parameters without this hint. It is permissible, and perhaps helpful to use this marker for non-parameter local variables.

The optional literal string at the end of the declaration allows type information to be inserted so that the code generator can create debugger type information for the local variable.

Parameter copy information

CopyInfo	→	.COPY *number* ',' *number* .SIZE *number* [Align] eolnSy
	|	.EXPAND ['('*number*')'] *number* ',' *number* .SIZE *number* [Align] eolnSy
	|	.OPENCOPY ['('*number*')'] *number* .SIZE *number* [Align] eolnSy
Align	→	.ALIGN *number* .

Parameter copy information is required in two circumstances. Firstly, if value parameters are passed by reference and are copied by the called procedure then this must be specified. Secondly, if conformant array parameters are either copied by the called procedure or are assembled by the caller at runtime, then information must be given to allow the array to be copied or the correct amount of space allocated respectively.

The *.COPY* declaration specifies that a value parameter has been passed by reference, and must be copied by the called procedure. The arguments to the declaration are —

`.COPY` *paramOffset* , *destOffset* `.SIZE` *objectSize* `.ALIGN` *alignNum*

The first argument is the offset in the stack frame of the parameter which is the reference to the actual parameter. The second is the destination offset in the fixed part of the declared stack frame. The declared size is the number of bytes in the object to be copied.

The alignment specification may be omitted if the target machine has no alignment constraints, or if the object is byte aligned. Otherwise the alignment number would be 1, 2, 4 or 8 for contemporary machines.

The copying of value mode, variable size, open arrays parameters is indicated as follows —

`.OPENCOPY`(*dimensions*) *paramOffset* `.SIZE` *elementSize* `.ALIGN` *alignNum*

[2] A variable is threatened by a procedure if the procedure directly assigns to the variable, or if takes the address of the variable or passes the address as a parameter to another procedure.

The meaning of the alignment attribute is as for fixed size array copies. However, in this case, the destination address must be determined at runtime. The size of the copy is determined by the computation —

$$size = elSize \times \prod_{i=1}^{dim} (high_i + 1)$$

where *elSize* is the specified element size, *dim* is the number of dimensions and $high_i$ is the i-th high value. The actual parameter reference is at stack frame offset *paramOffset* and it is assumed that the high value(s) are word-sized and at consecutive postions in the stack frame.

After the array has been copied, the backend must overwrite the reference so as to point to the copy. The allocated space need not be in the stack frame, but in any case the procedure epilog is responsible for deallocating the space at exit.

The *.EXPAND* declaration specifies that the procedure prolog will allocate and deallocate space for an open array of a shape specified by a designated actual parameter reference, the *template array*. The declaration is —

`.EXPAND`(*dimensions*) *paramOfst, dstPtrOfst* `.SIZE` *elemSize* `.ALIGN` *alignNum*

As for open array copies, the size of the allocated space is given by —

$$size = elSize \times \prod_{i=1}^{dim} (high_i + 1)$$

where *elSize* is the specified element size (which is not necessarily the same as the element size of the template array), *dim* is the number of dimensions and $high_i$ is the i-th high value. The template array reference is at stack frame offset *paramOffset* and it is assumed that the various *high* values are contiguous with that reference.

In this case, after the space has been allocated, the address of the allocated space will be placed in the stack frame at the offset given by *dstPtrOfst*. The allocation of an offset to hold this pointer is the responsibility of the frontend.

A.4.2 Procedure bodies

Procedure bodies begin with the keyword *.ENTRY* and end either with jump table declarations, or with the *.ENDP* marker. The form of the statements which make up the body of the procedures is treated in section A.5.

A.4.3 Jump tables

Case and switch statements are implemented by a vectored jump through a jump table. Jump tables appear at the end of procedures, immediately preceding the *.ENDP* marker.

JumpTable	→	.JUMPTAB Label eolnSy {IdList eolnSy}.
IdList	→	*ident* {‘,’ [eolnSy] *ident*}.

Jump tables consist of a label, by which the jump table is referenced, and an identifier list corresponding to all of the labels which the table references.

All of the identifiers in the list correspond to labels local to the current procedure body. The list is ordered so that consecutive identifiers in the list correspond to the labels which will be selected by consecutive values of the select expression index.

A.5 Statements

The body of procedures consists of statements and directives, one per line. Statements have optional labels, and labels may appear alone on lines.

Statements are the instructions of the *DCode* form, together with a small number of directives. The statements operate on the stack of the abstract machine, taking any non-literal operands from the stack, and returning results to the stack.

Statement	→	[LabelTag] Label [OpCode [OpArg]] eolnSy
	|	OpCode [OpArg] eolnSy
	|	Directive eolnSy.
LabelTag	→	.EXCEPT | .LOOP | .RETRY .
Directive	→	.TRAP *ident*',' *ident* {',' OpArg}
	|	.ENDLOOP [*number*] .
OpArg	→	*number* {',' *number*} ["fpParam"]
	|	*ident* [*number*]
	|	*relop*
	|	*mode* .
Label	→	*ident* ':' .

DCode knows of nine different primitive data types, as detailed in section A.3.1. However, the abstract stack itself knows only three: *words, floats* and *doubles*. For existing implementations, the first corresponds to either 32– or 64–bit words, and the other two to single and double precision *IEEE* floating point formats respectively. Whenever a datum is pushed on the abstract stack, the type of the source datum is specified and any format conversion, such as zero extension or sign extension, is implicit.

The semantics of statements are for the most part self explanatory, but some more detailed discussion is required for the division operators, testing and trapping, the handling of the parameter abstraction, and the handling of off-stack temporaries.

A.5.1 Directives

Marker directives are placed in the code to pass information to the backends. The directives are as follows —

`.ENDLOOP` this directive marks the end of any loop. The number following the directive, if present, has a meaning which is defined by the frontend, for the benefit of the human reader. For example, it might indicate the loop label with which it pairs.

`.EXCEPT` This label tag declares that the tagged label is the entry point to which control is transfered if an exception is raised in the preceding procedure body.[3]

`.LOOP` this label tag marks the start of a loop. The directive is attached to the label to which the back-edge(s) of the loop jump(s).

`.RETRY` this label tag marks the point to which control is transferred following the execution of a retrial attempt in an exception handler.

`.TRAP` this directive is used to indicate to backends that an out-of-line trap call is required, at the end of the current procedure.

Semantics of the trap directive

The trap directive is a declaration, and may logically occur anywhere within the body of the procedure to which is refers. It has two mandatory arguments, both of which are identifiers, and as many as four optional arguments. The form allows backends to create low overhead, out-of-line trap calls to runtime system entry points which are unknown to the backend.

The mandatory arguments of the directive are the trap identifier, and the identifier associated with the out-of-line label. Following the mandatory arguments as many as four optional arguments may occur. These are of the form of identifiers (with optional numeric offsets), or numbers.

At runtime, when a branch is taken to the label specified by the second argument to a trap directive, control passes to an out-of-line sequence of machine instructions. In this sequence the specified, optional arguments (third, ... arguments to the directive) will be passed (as first, ... parameters to a call) to the runtime trap entry point specified by the first argument to the directive. It is assumed that control will never return from these traps.

A.5.2 Instructions

Arguments to the instructions

The majority of abstract stack machine instructions have no explicitly specified operands. Exceptions include the *push address* and *call procedure* instructions, these take a symbolic address as operand. In all such cases the symbolic address argument has as its most general form *ident* $\pm$ *number*.

Some other instructions, such as *push literal* and *add offset* take numeric arguments. As noted in section A.2 such numeric tokens take an optional sign.

Most arithmetic instructions take an optional or mandatory mode argument. These arguments are one of *noTrap, intOver, crdOver* denoting no overflow trapping, integer overflow detection or cardinal (unsigned) overflow detection. The default is no trapping.

[3] This implies that exception handling must be on a per-procedure basis. More capable mechanisms are currently under consideration for later revisions of this intermediate form.

For the division instructions, the trapping mode is mandatory, and must be either *intOver* or *crdOver*, in order to completely specify the semantics of the operation.

Finally, the floating point comparison instructions have an argument which specifies the relational operator to be tested. As usual, the relops are “`<, <=, >=, >, =, <>, #`”.

Division and remainder instructions

There are a wide variety of division and remainder instructions, sufficient to cater for all the common high-level languages. The definitions of these instructions is as follows —

$$\begin{aligned}
a\,\text{`}\mathtt{div}\text{'}\,b &= \lfloor a/b \rfloor \\
a\,\text{`}\mathtt{mod}\text{'}\,b &= a - (a\,\text{`}\mathtt{div}\text{'}\,b) \times b \\
a\,\text{`}\mathtt{slash}\text{'}\,b &= \mathcal{R}_0(a/b) \\
a\,\text{`}\mathtt{rem}\text{'}\,b &= a - (a\,\text{`}\mathtt{slash}\text{'}\,b) \times b
\end{aligned}$$

where the / operators on the right denote exact division, and where $\mathcal{R}_0 : \mathcal{R} \rightarrow \mathcal{I}$ is the *round toward zero* function.

Here is a table, which gives an example of the results of applying the various division and remainder operations to some typical operands on the stack. In each case the top of stack is the right operand for the operation, while the left operand is below that.

stack top	10	–10	10	–10
next elem.	31	31	–31	–31
`slash`	3	–3	–3	3
`rem`	1	1	–1	–1
`div`	3	–4	–4	3
`mod`	1	–9	9	–1

The `test` instruction

The test instruction allows frontends to encode such things as index and range tests without inline expansion in the intermediate code. This has some advantages with target architectures which possess test-and-trap instructions. In some cases, it is also advantageous to keep tests as indivisible primitives in order to prevent the unnecessary fragmentation of basic blocks. The arguments to the instruction are the name of the trap to call if the test fails, and the low and high bounds against which to test.

Handling the parameter abstraction

The passing of parameters to procedures is achieved by two instructions `mkPar` and `blkPar`.

`mkPar` *par-size, asm-offset* [`fpParam`]

The effect of the `mkPar` instruction is to pop the abstract stack and move the popped datum to the parameter location. Typically this location will be the top of the runtime stack, or some specified location in the parameter assembly area, or perhaps some machine register. The first argument to the instruction is the size of the datum,[4] while the second argument, *par-offset* is the offset in the parameter assembly area. In the case of backends which pass parameters on a runtime stack, it is permissible for the second argument to be an arbitrary number. The optional `fpParam` marker indicates that the datum is a floating point value.

`blkPar` *par-size, asm-offset, opt-alignment*

The effect of the `blkPar` instruction is to copy *par-size* bytes from the location pointed to by the top of the abstract stack, and to move this datum to the parameter location. The address is popped from the abstract stack. Typically the parameter location will be the top of the runtime stack, or some postion in the parameter assembly area starting at offset *par-offset*. In the case of backends which pass parameters on a runtime stack, it is permissible for the second argument to be an arbitrary number. An optional third argument specifies the source alignment.

In any case, the details of which parts of the parameter assembly area (if any) are moved into registers is transparent to the *DCode* form. However, the method of parameter assembly cannot be hidden from the frontend, since this constrains the order of parameter evaluation.

Returning function values

Function values are communicated by means of the `popRet`X`,` `pshRet`X abstraction. The first of these pops the top-of-stack *word, float* or *double* to the return location. This is typically a machine register. The `pshRet`X instructions take the value in the return location and push it onto the abstract stack. In the case of short values, the exact sign or zero extension is specified.

DCode provides some choice in the way that structured function values are returned. The default is to require the frontend to allocate space for the return value, and to pass a reference to this location as an implicit, additional, first parameter to the function. Using this method does not require any special instructions. However, for machines for which it is conventional to pass such function return-value pointers in a dedicated location in the runtime stack frame an optional abstraction has been introduced. The `mkDstP` instruction pops the top of the abstract stack into the dedicated function return-value pointer location. Within the called procedure the function return-value pointer is accessed by means of the `pshDstP` instruction.

[4]Of course, the size of the datum could have been computed by an interpretive code generator in the backend, but is specified explicitly for simplicity.

Off-stack temporaries

DCode provides for values to be moved from off the stack of the abstract machine into off-stack locations. There are two separate mechanisms for doing this.

The `mkTmp, pshTmp` instruction pair store and retrieve temporary values off the stack. The *make-temporary* instruction non-destructively copies the datum on the top-of-stack into an off-stack location. The instruction specifies the absolute value of the offset in the stack frame which the frontend has temporarily reserved for this datum. Corresponding *push-temporary* instructions will push the value onto the abstract stack. Several points need to be made about this mechanism. Code generating backends will attempt to store such temporaries in machine registers, and are not obliged to use the same register for each defining occurrence of a temporary at a particular offset. The mechanism should therefore not be used to attempt to merge values computed along different control paths in a program. Furthermore, interpretive code generators may treat certain temporaries as symbolic quantities, so that the temporary has no concrete existence at runtime, either at the stack frame location which was specified, or in a machine register.

The `popFi, pshFi` instruction pair store and retrieve values in off-stack locations which are determined by the backend. The *pop*-Φ instruction pops the top of the abstract stack into an off-stack location which is allocated by the backend. The argument to the instruction is simply an index which uniquely selects a particular location. Every such instruction with the same index within any single procedure, will move the top-of-stack to the same location. The *psh*-Φ instruction pushes the saved value back onto the abstract stack.

This instruction pair is suitable for merging values which are computed along different control paths in a program, as is required by language *C*'s conditional expressions. However, it is preferable in all such cases for the frontend to allocate a dummy local variable, with '(0,0,0)' attributes (see section A.4.1), and merge values by assigning to this variable. This attribute assignment will ensure that the variable is considered for allocation to a machine register.

Alphabetical instruction list

In the following table parameters in square brackets mean that the argument is optional. Instructions marked with a dagger sign † are instructions which have to do with the specifics of segmented architectures such as the iapx86, and might be ignored in other versions.

opcode	params	description
`abs`	[mode]	Takes the integer on the top of the stack and replaces it by its absolute value. If mode is *intOver* then an attempt to negate the minimum integer is trapped
`absDbl`	. . . .	Takes the double precision floating point number on the top of the stack and replaces it by its absolute value
`absFlt`	. . . .	Takes the single precision floating point number on the top of the stack and replaces it by its absolute value
`add`	[mode]	Adds the two values on the top of the stack with overflow trapping semantics as specified by mode. The stack is popped by two, and the result pushed
`addAdr`	. . . .	Adds the value on the top-of-stack to the address below that. The addend may be either positive or negative, and the computation is performed modulo the size of the address space. The stack is popped by two, and the result pushed
`addDbl`	. . . .	Adds the two *double*s on the top-of-stack. The stack is popped by two, and the result pushed
`addFlt`	. . . .	Adds the two *float*s on the top-of-stack. The stack is popped by two, and the result pushed
`addOff`	offset	Adds the constant offset to the address on the top of the stack. The offset may be either positive or negative, and the computation is performed modulo the size of the address space. The new address value replaces the old on the top-of-stack
`andWrd`	. . . .	Bitwise *AND* of two top-of-stack values. The stack is popped by two, and the result pushed
`assign16`	. . . .	Assigns the least significant 16-bits of the value second on the stack to the address specified on the top-of-stack. The stack is popped by two
`assign32`	. . . .	Assigns the least significant 32-bits of the value second on the stack to the address specified on the top-of-stack. The stack is popped by two
`assignB`	. . . .	Assigns the least significant *byte* of the value second on the stack to the address specified on the top-of-stack. The stack is popped by two
`assignD`	. . . .	Assigns the *double* second on the stack to the address specified on the top-of-stack. The stack is popped by two
`assignF`	. . . .	Assigns the *float* second on the stack to the address specified on the top-of-stack. The stack is popped by two

opcode	**params**	**description**
`assignW`	....	Assigns the *word* second on the stack to the address specified on the top-of-stack. The stack is popped by two
`bitNeg`	....	Bitwise negate the top-of-stack value. The result replaces the top of stack value
`blkCp`	[number] ...	Block copy. This instruction has three parameters on the stack: on the top-of-stack is the number of bytes of the block, below that is the source address, below that the destination address. All three values are popped. The optional number is 1, 2, 4 or 8 and gives the block alignment. The default alignment is 1 (byte aligned)
`blkPar`	number, number [,number] ..	This instruction is performs a parameter block copy. The top of the stack is a pointer to the actual parameter. The first two arguments to this directive are the parameter size, and the destination location in the parameter assembly area. An optional third parameter gives the *source* alignment boundary, and is always 1, 2, 4 or 8. In the case of machines which pass their parameters on the stack, the number of bytes specified will be copied to the runtime stack. In the case of fixed parameters the block will be copied to the specified offset from the present stack pointer. In either case, the stack is popped by one
`boolNeg`	....	Negates the Boolean value on the top-of-stack. The result replaces the top-of-stack value
`brFalse`	label	Branch if the top-of-stack value is zero, and pop the stack
`brTrue`	label	Branch if the top-of-stack value is non-zero, and pop the stack
`branch`	label	Branch unconditionally
`call`	identifier, number.....	Call the procedure designated by the identifier, with the specified number of parameters
`crdGE`	....	Compares two top values as unsigned words. Returns *TRUE* if the second from the top is greater than or equal to the top. The stack is popped by two
`crdGT`	....	Compares two top values as unsigned words. Returns *TRUE* if the second from the top is greater than the top. The stack is popped by two
`crdLE`	....	Compares two top values as unsigned words. Returns *TRUE* if the second from the top is less than or equal to the top. The stack is popped by two
`crdLS`	....	Compares two top values as unsigned words. Returns *TRUE* if the second from the top is less than the top. The stack of the abstract machine is popped by two
`cutPars`	number.....	Remove parameter bytes (only for machines which pass parameters by pushing them onto the stack of the actual machine – the *concrete stack*)
`dblRel`	relop	Pop and compare *double*s on top of stack, and push the Boolean result. Right operand is initially top-of-stack
`derefF`	....	Dereferences the pointer to *float* on the top-of-stack. The top-of-stack value is replaced by the result

opcode	params	description
`derefD`	....	Dereferences the pointer to *double* on the top of stack. The top-of-stack value is replaced by the result
`derefSB`	....	Dereferences the pointer to *signed byte* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefS16`	....	Dereferences the pointer to *signed 16-bit word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefS32`	....	Dereferences the pointer to *signed 32-bit word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefUB`	....	Dereferences the pointer to *unsigned byte* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefU16`	....	Dereferences the pointer to *unsigned 16-bit word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefU32`	....	Dereferences the pointer to *unsigned 32-bit word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`derefW`	....	Dereferences the pointer to *word* on the top of stack. The value on the top of the stack is replaced by the dereferenced value
`div`	mode.......	Performs the div operation on the two values on the top of the stack with semantics as specified by mode. The right operand is on top. The stack is popped by two, and the result pushed
`divDbl`	....	Divides the *doubles* on the top-of-stack. The right operand is on top. The stack is popped by two, and the result pushed
`divFlt`	....	Divides the *floats* on the top-of-stack. The right operand is on top. The stack is popped by two, and the result pushed
`dRound`	[mode]	Convert *double* to word rounding to nearest. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top-of-stack value is replaced by the result
`dFloor`	[mode]	Convert *double* to word rounding to minus infinity. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top-of-stack value is replaced by the result
`dToFlt`	....	Shorten *double* on top-of-stack to *float*. The top-of-stack value is replaced by the result
`dTrunc`	[mode]	Convert *double* to word rounding to zero. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top-of-stack value is replaced by the result

opcode	params	description
`dup1`	....	Duplicates the top value on the stack
`endF`	....	Marks the end of the *DCodes*
`endP`	....	Marks the end of the current procedure
`exit`	....	Unconditional jump to the procedure epilog
†`flatten`	....	Transforms the segmented address on the top of stack into a unsigned (*nop* except for iapx86 in 16-bit segmented mode)
`fltRel`	relop	Pop and compare *floats* on top-of-stack, and push the Boolean result. Right operand is initially top-of-stack
`fRound`	[mode]	Convert *float* to word rounding to nearest. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top-of-stack value is replaced by the result
`fFloor`	[mode]	Convert *float* to word rounding to minus infinity. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top-of-stack value is replaced by the result
`fToDbl`	....	Widens a *float* to a *double*. The top-of-stack is replaced by the result
`fTrunc`	[mode]	Convert *float* to word rounding to zero. If mode is *intOver* or *crdOver* then values outside the (respectively) signed or unsigned range are trapped. If trapping is not specified, then backends should produce the correct value if possible. The top-of-stack value is replaced by the result
`intGE`	....	Compares two top values as signed words. Returns *TRUE* if the second from the top is greater than or equal to the top. The stack of the abstract machine is popped by two
`intGT`	....	Compares two top values as signed words. Returns *TRUE* if the second from the top is greater than the top. The stack of the abstract machine is popped by two
`intLE`	....	Compares two top values as signed words. Returns *TRUE* if the second from the top is less than or equal to the top. The stack of the abstract machine is popped by two
`intLS`	....	Compares two top values as signed words. Returns *TRUE* if the second from the top is less than the top. The stack of the abstract machine is popped by two
`iToDbl`	....	Floats the top-of-stack from integer to *double*. The top-of-stack is replaced by the result
`iToFlt`	....	Floats the top-of-stack from integer to *float*. The top-of-stack is replaced by the result
`jump`	....	Branch to the address on the top of the stack. Performs a computed goto jump, by jumping to the address on the top of the stack, which is popped
`lineNum`	number.....	Marks the line number for diagnostics only. Typically generated by compilers in response to the –**g** flag

opcode	params	description
†`makeAdr`	....	Transform the top-of-stack unsigned into an address, this is the inverse of `flatten`
`mkDstP`	number.....	This instruction pops the value from the top of the stack, and moves it to the dedicated destination pointer location. This is only used on architectures such as *SPARC* which do not pass destination pointers as an additional ordinary parameter. The argument is the destination size, which is used in some conventions as an attribute in a trap instruction
`mkTmp`	number.....	Associates the top-of-stack value with the current stack frame location \$fp − *number*. In most backends symbolic values on the stack will be left as such. Even for non-symbolic values backends should take the frame location as a hint, and attempt to keep the value in a machine register
`mkPar`	number, number, [`fpParam`]...	This instruction causes the value on the top of the stack is to be popped and transferred to the parameter location. It takes two arguments. The first of these is the size of the parameter, rounded to the smallest parameter size allowed in the machine conventions. The second is the parameter offset in the assembly area, or zero. The optional marker indicates that this is a floating point parameter
`mod`	mode.......	Takes the modulus of the two values on the top of the stack with semantics as specified by mode. Right operand is on top. The stack is popped by two, and the result pushed
`mul`	[mode]	Multiplies the two values on the top of the stack with semantics as specified by mode. The stack is popped by two, and the result pushed
`mulDbl`	....	Multiplies the two *double*s on the top-of-stack. The stack is popped by two, and the result pushed
`mulFlt`	....	Multiplies the two *float*s on the top-of-stack. The stack is popped by two, and the result pushed
`negate`	[mode]	Negates the integer on the top-of-stack. If mode is *intOver* then an attempt to negate the minimum integer is trapped. The result replaces the top-of-stack
`negDbl`	....	Negates the *double* on the top-of-stack. The top-of-stack is replaced by the result
`negFlt`	....	Negates the *float* on the top-of-stack. The top-of-stack is replaced by the result
`orWrd`	....	Bitwise *OR* of the two top-of-stack words. The stack is popped by two, and the result pushed
`pop1`	....	Pop the stack by one word
`popCall`	number.....	Call the procedure whose address is on the top-of-stack with the specified number of parameters. The address is popped from the stack
`popFi`	number.....	Pop the top-of-stack value into a location uniquely but arbitrarily associated with the ordinal number
`popRetD`	....	Pop the top-of-stack *double* into the double precision *double* return location

opcode	params	description
`popRetF`	....	Pop the top-of-stack *float* into the single precision *float* return location (see note 2)
`popRetW`	....	Pop the top-of-stack into the word return location
`pshAdr`	identifier ...	Push the address of the statically allocated variable *identifier* onto the stack
`pshDsp`	index, offset	Push the address pointed to by the display vector element $display[index] + offset$.
`pshDstP`	....	This instruction pushes the contents of the dedicated location which holds the address of the return pointer. This is only used on architectures such as *SPARC* which do not pass destination pointers as an additional ordinary parameter
`pshFi`	number.....	Push contents of the location uniquely but arbitrarily associated with the ordinal number onto the stack
`pshFP`	offset.......	Push the address at the given offset from the frame pointer \$fp + *offset*
`pshLit`	number.....	Push the literal onto the stack (see note 1)
`pshRetD`	....	Copy the *double* in the return location to the top-of-stack
`pshRetF`	....	Copy the *float* in the return location to the top-of-stack
`pshRetSB`	....	Copy the *signed byte* in the return location and sign-extend
`pshRetS16`	....	Copy the *signed 16-bit word* in the return location and sign-extend
`pshRetS32`	....	Copy the *signed 32-bit word* in the return location and sign-extend
`pshRetUB`	....	Copy the *unsigned byte* in the return location and zero-extend
`pshRetU16`	....	Copy the *unsigned 16-bit word* in the return location and zero-extend
`pshRetU32`	....	Copy the *unsigned 32-bit word* in the return location and zero-extend
`pshRetW`	....	Pushes the word return register to the top of stack
`pshTmp`	number.....	Pushes the value associated with location \$fp − *number* onto the stack. Matches mkTmp
`pshZ`	....	Pushes the literal 0 (zero)
`relEQ`	....	Compares the two top word-sized values. Returns *TRUE* if the second from the top is equal to the top. The stack is popped by two
`relNE`	....	Compares two top word-sized values. Returns *TRUE* if the second from the top not equal to the top. The stack is popped by two
`rem`	mode.......	Takes the remainder of the two values on the top of the stack with semantics as specified by mode. The stack is popped by two, and the result is pushed
`rotate`	....	Rotates the *word* second on stack left or right by the number of places specified by the top-of-stack value. Positive rotates are leftward. The stack is popped by two, and the result is pushed

opcode	**params**	**description**
`setExcl`	....	Excludes the bit specified by the value on the top of the stack from the word-sized bitset second on the stack. If a range test is required on the bit-specifying ordinal, this should be done explicitly. Implementations are free to perform the operation modulo-*bits-in-word*. The stack is popped by two, and the result pushed
`setIn`	....	Tests if the bit specified by the value on the top of the stack is set in the word-sized bitset second on the stack. Returns a Boolean on the top-of-stack. Top-of-stack word is taken modulo-*bits-in-word*. The stack is popped by two, and the result pushed
`setIncl`	....	Includes the bit specified by the value on the top of the stack from the word-sized bitset second on the stack. If a range test is required on the bit-specifying ordinal, this should be done explicitly. Implementations are free to perform the operation modulo-*bits-in-word*. The stack is popped by two, and the result pushed
`setGE`	....	Compares the two top values on the stack. Returns *TRUE* if the set second from the top is a superset of the top. The stack is popped by two, and the result pushed
`setLE`	....	Compares the two top values on the stack. Returns *TRUE* if the set second from the top is a subset of the top. The stack is popped by two, and the result pushed
`shLeft`	....	Shifts the value second on the stack left by the number of places specified by the unsigned value on the top-of-stack. The stack is popped by two, and the result pushed
`shiftV`	....	Shifts the value second on the stack left or right by the number of places specified by the integer value on the top-of-stack. Positive shifts are leftward. The stack is popped by two, and the result pushed
`shRightS`	....	Shifts the signed value second on the stack right by the number of places specified by the unsigned value on the top-of-stack. This is an arithmetic shift. The stack is popped by two, and the result pushed
`shRightU`	....	Shifts the unsigned value second on the stack right by the number of places specified by the unsigned value on the top-of-stack. This is a logical shift. The stack is popped by two, and the result pushed
`slash`	mode.......	Divides the two values on the top of the stack with semantics as specified by mode. The stack is popped by two, and the result pushed
`sub`	[mode]	Subtracts the two values on the top of the stack with overflow trapping semantics as specified by mode. The right operand is on top. The stack is popped by two, and the result pushed
`subDbl`	....	Subtract the two *doubles* on the top-of-stack. The stack is popped by two, and the result pushed
`subFlt`	....	Subtract the two *floats* on the top-of-stack. The stack is popped by two, and the result pushed
`swap`	....	Swaps the two top values on the stack

opcode	params	description
`switch`	identifier ...	Perform a case statement jump using the top of the stack as an index into a jump table at label *identifier*
`test`	identifier, number, number.....	Perform the specified test-and-trap operation on the top-of-stack datum, then pop the stack
`trap`	identifier, number.....	Same as `call` except no return is expected
`uToDbl`	....	Floats the top-of-stack from unsigned to *double* (*LFLOAT*). The top-of-stack is replaced by the result
`uToFlt`	....	Floats the unsigned value on the top-of-stack to *float* (*SFLOAT*) The top-of-stack is replaced by the result
`xorWrd`	....	Bitwise exclusive *OR* of the two top values. The stack is popped by two, and the result pushed

Note 1

Current backends assume that floating point instructions never have operands that have been pushed on the stack as literals.

Note 2

For machines which use an integer register for returning single precision floating point values *popRetF* and *popRetW* will be treated as synonyms.

A.6 Limitations on the control flow

DCode places the following restrictions on the control flow permitted in program representations.

- all control flow paths leading to any label in the code must have the same stack height along all edges leading to the label
- all control transfers, whether conditional or unconditional, must jump forward, except for back-edges of loops. All the back-edges of any particular loop must be enclosed in a *.LOOP, .ENDLOOP* directive pair
- the loop marker directives of nested loops must be properly nested

Backends which perform global dataflow analysis may very well not need this information.