Notes on Code Generation using Standard ML

Mads Tofte
Department of Computer Science (DIKU)
Copenhagen University
Universitetsparken 1, DK-2100 Ø
Denmark

5 Feb 1991
### Contents

1 Introduction 1
   1.1 Using these notes to learn ML .......................... 1
   1.2 The PL/0 compiler .................................... 2
   1.3 Software requirements .................................. 2
   1.4 Books with which these notes can be used ............... 3

2 The Language PL/0 4

3 The SC Machine 7
   3.1 The stack .............................................. 7
   3.2 The instructions ........................................ 10
   3.3 An interpreter for the SC Machine ...................... 12

4 Elaboration 14
   4.1 Lexical indices .......................................... 14
   4.2 The static environment .................................. 16
   4.3 The elaborater .......................................... 17

5 Direct Code Generation 23
   5.1 The significance of lexical indices for the running system ............... 23
   5.2 Flow of control .......................................... 25
   5.3 Implementation .......................................... 25

6 Code Generation using Continuations 34
   6.1 Continuations .......................................... 34
   6.2 Generating code backwards ............................... 35
   6.3 Code containing holes ................................... 39
   6.4 Sequential composition .................................. 40
   6.5 Parallel composition .................................... 42
   6.6 Preserving the environment ................................ 43
   6.7 Lazy register preservation ............................... 48
   6.8 Tail recursion .......................................... 49

7 Register Allocation 53
   7.1 Introduction ............................................. 53
   7.2 Requirement maps ....................................... 53
   7.3 Register distribution .................................... 55
      7.3.1 Assignment .......................................... 59
      7.3.2 The sequential statement ......................... 60
      7.3.3 The if statement .................................... 60
      7.3.4 The while statement ................................. 61
      7.3.5 The call statement .................................. 63
7.3.6 Program variable expression ........................................ 63
7.3.7 Constant in expression ................................................ 65
7.3.8 Application of a binary numeric operator ........................... 65
7.3.9 Application of a binary boolean operator ........................... 66
7.4 Code generation ............................................................. 67

A Solutions ................................................................. 69
1 Introduction

These notes are the technical documentation of a collection of program modules which we shall refer to as the PL/0 compiler. The PL/0 compiler is written in Standard ML and exercises Standard ML extensively.

To understand the nature of these notes, a few historical comments are required. The work on these notes began when I had to teach a course on code generation for Computer Science undergraduate students at University of Nigeria. Most Nigerian students cannot afford good textbooks and they lack programming experience. I therefore wanted to develop both some notes about code generation and some nice programs to show what code generation looks like fully implemented. I must confess that for the purpose of writing and programming, it was not all bad to be physically separated from existing literature on the subject. Of course one wastes some time rediscovering things people first realised 30 years ago. On the other hand, if one tries writing a compiler from scratch, one has to think harder and one just might bump into things that have not been done before, especially because the developments in semantics and programming languages over the past 15 years or so can provide lots of inspiration when one tries to write compilers.

One small example illustrates this last point. Traditional code generators often generate code containing jumps to jumps. A separate optimisation phase then removes such superfluous jumps. This is of course perfectly alright, except that it is not very nice. It turns out that the reason such jumps occur (at least the context of the language we study) is that one tries to compile phrases under false assumptions about where the compiled code is to continue if it terminates. By introducing a very simple brand of continuations one obtains just the right level of parameterisation of the code generator functions and jumps to jumps simply never arise.

1.1 Using these notes to learn ML

If you do not know ML, but already are a programmer or have an interest in programming languages, you should be able to use these notes as a short cut to learning ML programming. The notes contain many examples of real ML programs and you are thrown in at the deep end with modules right from the beginning. However, most of the exercises are small programming exercises that gradually evolve from being very simple to being more demanding in terms of the ML proficiency assumed. Solutions to all the exercises are provided in Appendix A. In addition, the notes contain a number of projects which are intended as slightly larger programming exercises some of which might take a couple of days of programming.

Since these notes do not contain a general description of ML, you will need additional assistance when you use the language; see 1.4 below for suggested reading.

The focus of these notes is squarely on the problem at hand, code generation, so if one wants to use these notes to decide whether it is really so wonderful to use functors, for example, one has to take an interest in the code generation problems we are trying to solve. In my experience, most programmers agree that data abstraction is a good idea,
and yet they do not practise it. The reason might be that when taught in theory, data abstraction is so simple as to be almost trivial and until one actually has experienced its benefits in practical programming, one does not appreciate its importance. My own experience with writing these notes was that ML was indeed very suitable for solving the problems at hand. In particular, I am well pleased with the functorised code generation algorithm in Section 6; my first versions of this algorithm were much more explicit and less readable than the present functorised version, although they essentially work in the same way.

1.2 The PL/0 compiler

The language under study in these notes is called PL/0. It is due to N. Wirth, who in his book [9] defined the language and presented a compiler for it written in Pascal. PL/0 is a small subset of Pascal. It has block structure, static scoping and recursive procedures. The statements include assignment, if and while statements and procedure call. All values are integers. There is no input-output.

The PL/0 compiler documented in these notes is structured very differently from Wirth’s original PL/0 compiler. Wirth’s compiler did everything from lexical analysis to code generation in one pass. The object language of Wirth’s compiler was an abstract stack machine language. Our PL/0 compiler is divided into modules that do compilation in a number of passes in relative isolation from each other. The passes that are implemented so far are: elaboration, register distribution and code generation. We compile to three different object languages, starting from Wirth’s object language. One of the other object languages strongly resembles assembly language for a RISC machine. Translation into machine code has not been implemented as yet, but is planned for the future.

In writing the PL/0 compiler our overriding concern has been the clarity of the compiler and the code it produces. Efficiency of the compiler and the compiled code has been a major concern, but only indirectly in that inefficient code is often long and ugly. It is typical for the PL/0 compiler that it carefully avoids unnecessary symbolic labels and makes a concerted effort to do good register allocation. It is also typical that symbol tables (or environments, as they are called) deep down in the basic modules turn out to be implemented simply as linked lists. Such sources of potential inefficiency are not important at this stage, as the time one spends waiting for the ML system to compile or import modules far exceeds the time the PL/0 compiler takes to compile programs.

1.3 Software requirements

The PL/0 compiler is implemented using the “Standard ML of New Jersey” ML system, which mainly was written by David MacQueen of Bell Labs and Andrew Appel of Princeton University. The only nonstandard feature of the New Jersey implementation used in the PL/0 compiler is the separate compilation scheme. Thus it should not be difficult to port the PL/0 compiler to another ML implementation.

Besides an ML compiler, you will of course need the PL/0 compiler which is available
1.4 Books with which these notes can be used

In this section we mention some of the books that one might use together with these notes, if one is not already familiar with ML.

First I should acknowledge that the original PL/0 compiler appeared in [9]. Also, my PL/0 compiler was influenced greatly by [1], in particular I have borrowed their choice of continuations and their method of register preservation although my use of the continuations differs from theirs.

If you are new to functional programming, you can consult a textbook on functional programming. Two textbooks that rely on Standard ML for teaching functional programming are [6] and [8]. A book by Larry Paulson is expected Summer ’91 on Cambridge University Press.

There also exist shorter introductions to ML in the form of technical reports from Edinburgh University ([3], [2], [7]).

References


2 THE LANGUAGE PL/0

PL/0 is a small version of Pascal. Only integer values are allowed. Procedures can contain local declarations of constants, values and other procedures. Procedures have no parameters. There is no input or output. The lexical conventions are as in Pascal and will not be described here. The grammar of PL/0 is presented in Figure 1. We use braces ({ and }) for grouping in the grammar and angle brackets ⟨ and ⟩ to enclose optional parts; repetition 0 or more times is indicated with a superscripted star (*), while repetition 1 or more times is indicated with a superscripted plus (+). An example PL/0 program is shown in Figure 2.

Traditionally, a compiler for PL/0 would consist of separate phases: first lexical analysis, then parsing, then some kind of static analysis and type checking, then codegeneration.

Figure 1: Grammar for PL/0
Declaration:

```plaintext
var x;
procedure P;
x := x + 1;
procedure Q;
begin
  call P;
call P
end;
begin (* main *)
x := 0;
call Q
end.
```

Figure 2: An example PL/0 program

and perhaps optimisation. After parsing has been completed, the source program has been transformed into an abstract syntax tree. We shall not in these notes be concerned with lexical analysis and parsing. We therefore start out from abstract syntax trees. Below is an ML signature which specifies the data type of abstract syntax trees.

```plaintext
signature ABSTSYNTAX =
  sig
    datatype 'a option = NONE | SOME of 'a;

    type ident
    and number
    sharing type number = int
    and type ident = string

    datatype program = PROGRAM of block
    and constdec = CONST of (ident * number) list
    and vardec = VAR of ident list
    and procdec = PROCDEC of (ident * block) list
    and block = BLOCK of constdec option * vardec option *
    procdec option * statement
    and statement = ASSIGN of ident * expression
    | SEQ of statement * statement
    | IF of expression * statement
    | WHILE of expression * statement
    | CALL of ident
    | EMPTYSTAT
```
and expression = IDENT of ident
| NUM of number
| APPMONOP of monop * expression
| APPBINOP of expression * binop * expression
and monop = ODD | UPLUS | UMINUS
and binop = EQUALS | NOTEQUAL | LTH | GTH | LEQ | GEQ
| PLUS | MINUS | TIMES | INTDIV;
end;

Below is an ML term which represents part of the abstract syntax tree for the PL/0 program we gave in Figure 2:

\[
\text{BLOCK(NONE,NONE,NONE,}
\text{ASSIGN("x",APPBINOP(IDENT"x",PLUS,NUM 1))})
\]

**Exercise 2.1** What is the type of the above syntax tree?

**Exercise 2.2** Translate the following PL/0 phrase to an abstract syntax tree: \(3 * 4 + 5 / 8\). (Make sure to represent the precedence of the operators correctly.)

**Exercise 2.3** Write down an ML term which represents the abstract syntax for the entire program in Figure 2.

**Project 2.4** Start up an ML session in the directory `DirectCompiler`. Now type the following, line for line (the system will give you a response to every line):

```ml
import "AbstSyntax";
structure A = AbstSyntax();
open A;
```

Now you have data types as specified in the `ABSTSYNTAX` signature at your disposal. Try typing in your solution to Exercise #2.3. What is the type of the tree?
3 The SC Machine

The purpose of this section is to describe the so-call SC Machine. The SC Machine is an abstract machine, that is, it is a conceptual machine rather than a physical machine. It is, however, much closer to physical machines that PL/0 itself, and if we can compile PL/0 into instructions for the SC machine, we can compile such instructions into real native machine instructions in a separate phase.

3.1 The stack

The SC Machine has two stores, a stack, $S$, and a code store, $C$; hence its name. The code store holds instructions of the machine; once given a contents, it does not change during the execution of the program. There is a register $P$, the program pointer, which points to the next instruction to be executed. The stack is modified by the execution of the instructions. The stack is used for storing values, all of which are integers, and also for storing pointers into the stack and the code. We assume that the stack is make up of words. (Such a word might well be represented by a word in a physical machine.) Every word has a unique address. We shall assume that stack words have addresses $0, 1, 2, \ldots$, with 0 being the address of the bottom of the stack.

In drawing pictures of the stack, we choose to have the bottom of the stack to the left and the top of the stack to the right. The machine has a top register, $T$, which points to the topmost word of the stack.

Since PL/0 has procedures that can call each other an even themselves recursively, one cannot allocate a fixed memory location for each variable that is declared in the source program. Instead, when a procedure $Q$ is called, an activation record for that procedure is created as an extension to the stack at the top end. An activation record is a contiguous sequence of words; the address of the leftmost word of the record is called the base of the record; the word itself is called the base word of the record.

An activation record consists of the following components:

1. The Static Link, which is an addresss (“pointer”), namely the base of the topmost activation record for the procedure $R$, say, which textually surrounds $Q$. This is necessary in order to be able to access the variables that are free in $Q$ but declared in $R$. $R$ itself will have a static link pointing to the activation record of the procedure in which it itself was declared, and so on, till one reaches the activation record for the main block of the program. This static chain makes it possible to access any variable which is in scope within $R$ by descending the static chain.

2. The Dynamic Link, which is the base of the activation record for the procedure which called $R$. This is needed so that the activation record for this procedure can be reinstated, once the execution of $Q$ has finished.

3. The Return Address, i.e. the address of the first instruction to be executed, after the execution of $Q$ has finished. (This address is not in general a fixed property of $Q$, since $Q$ may be called from several places.)
var a, b;
procedure Q;
  var c, d;
procedure R;
begin
  c:= b + d;
  if c < a then
    begin b:= c; call Q end
end
begin (* Q *)
  d:= 1; call R
end;
begin (*main*)
  a:= 2; b:= 0; call Q
end.

Figure 3: A PL/0 program

4. **Local Variables**: After the three words needed for the above pointers comes the space reserved for the variables declared local to Q. There is one word for every variable. Hence, if Q has n local variables, the total size of the activation record is n + 3.

When a program is running, the activation record of the currently active procedure is the topmost activation record on the stack. The machine has a *base register*, B, which is used to hold the base of this activation record.

Immediately after Q has been called, and before the body of Q is executed, the base register B points at the activation record for Q and the top register T points at the top of the stack, which at this point is also the last word of the activation record. During the execution of Q, the stack may grow further because of operations that use the stack to hold temporary values, for example to compute arithmetical expressions. Also, if Q calls some other procedure, a new activation record will be allocated on the stack. However, when the time comes for Q to return, the T register will have come back to the rightmost word of the activation record for Q. The return itself the pops the activation record off the stack and reinstates the registers to the values they had before the call. Hence, if the entire program terminates, the stack will contain just one activation record, namely the one for the main program which contains the values of the global variables.

**Example 3.1** Consider the execution of the (compiled code of the) PL/0 program in Figure 3. At the point Q calls R for the first time, the stack looks as follows:
3.1 The stack

Here $m_1$ is the return address for $Q$ in the code of the main program. By convention, we draw the static link above the stack and the dynamic link below the stack. Notice that both the static and the dynamic link for this call of $Q$ point to the activation record for the main program.

Just after the call, the stack appears as follows:

\[
\langle \text{main} \rangle \langle Q \rangle \langle R \rangle \langle Q \rangle \langle R \rangle
\]

\[
\begin{array}{cccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{B} & \text{T} \\
2 & 0 & m_1 & 1 & q_1 & 0 & 1 & q_1 \\
\end{array}
\]

Next, $R$ calls $Q$; just as $Q$ starts executing, the stack is:

\[
\langle \text{main} \rangle \langle Q \rangle \langle R \rangle \langle Q \rangle \langle R \rangle \langle Q \rangle
\]

\[
\begin{array}{cccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{c} & \text{d} & \text{B} & \text{T} \\
2 & 1 & m_1 & 1 & q_1 & r_1 & 0 & q_1 \\
\end{array}
\]

Note that the static link for $Q$ points all the way back to the activation record for the main program; the dynamic link for $Q$ points to the activation record of its caller, $R$.

Next, $Q$ calls $R$, where the test $c < a$ now becomes false. At this point, the stack is:

\[
\langle \text{main} \rangle \langle Q \rangle \langle R \rangle \langle Q \rangle \langle R \rangle \langle Q \rangle \langle R \rangle
\]

\[
\begin{array}{cccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{c} & \text{d} & \text{B} & \text{T} \\
2 & 1 & m_1 & 1 & q_1 & r_1 & 2 & q_1 \\
\end{array}
\]

Thus $R$ will have to return to $Q$ at address $q_1$. After the return, the stack is
But nothing remains to be done in Q, which therefore returns to R at code address \( r_1 \):

\[
\langle \text{main} \rangle \langle Q \rangle \langle R \rangle \langle Q \rangle
\]

\[
\begin{array}{cccccc}
2 & 1 & m_1 & 1 & 1 & q_1 & r_1 & 2 & 1 \\
\end{array}
\]

Then R returns to Q and Q returns to the main program, resulting in the final stack:

\[
\langle \text{main} \rangle
\]

\[
\begin{array}{cccc}
2 & 1 \\
\end{array}
\]

Exercise 3.1  Consider the execution of the PL/0 program in Figure 2. Draw a picture of the stack just after the code for \( x := x + 1 \) has been executed (a) for the first time, and (b) for the second time.

3.2  The instructions

The instructions of the machine and their types can be summarised by the following ML signature, which you can find in the file DirectCompiler/INST.sml:

```ml
signature INST =
sig
  type label
  datatype inst =
    LIT of int | OPR of int | LOD of int * int | STO of int * int |
    CAL of int * label | INT of int | JMP of label | JPC of label |
    LAB of label;
  val showinst: inst -> string
  val show_indent_inst: inst -> string (* put blanks in front of instructions that are not
```

```ml
```
The meaning of these instructions is as follows. LIT\(i\) pushes the integer \(i\) onto the top of the stack. OPR\(i\), where \(1 \leq i \leq 11\) represent the built-in basic operations, for example OPR\(2\) is addition. The built-in operations fall into two categories. A unary operation, for example odd and integer negation, replaces the value on the top of the stack by the result of applying the operation to that value. A binary operator, such as addition and equality, replaces the two topmost values \(x\) and \(y\) of the stack by a single integer, namely the result of applying the operation to \(x\) and \(y\).

LOD\((i, j)\) is used for loading the contents of a variable onto the top of the stack. Let \(b\) be the contents of \(B\). (Recall that \(b\) is the base of the topmost activation record.) Also, for any address \(a\), let \(c(a)\) denote the contents of the word at address \(a\). If \(i\) is 0, the address of the variable is \(b + j\), i.e. the \((j + 1)\)st word in the topmost activation record. This occurs when the variable is local to the currently active procedure. If \(i\) is 1, the address of the variable is \(c(b) + j\); in general, i.e. for any \(i \geq 0\) the address is \(c^i(b) + j\), i.e. one starts from the base word in the topmost activation record, follows \(i\) levels of indirections through the stack and then adds \(j\) to get the address of the desired variable.

STO\((i, j)\) is exactly the inverse operation of LOD\((i, j)\), i.e. it removes the value on the top of the stack and stores it in the location found by following the static link as described above.

INT\((i)\), which stands for “increment \(T\)”, increments the value of \(T\) by \(i\). It is used to allocate stack space for a new activation record.

JMP\((l)\), where \(l\) is a label, jumps to the instruction a label \(l\). JPC\((l)\) jumps to \(l\) if the top of the stack is 0 (signifying false); the value on the top of the stack is removed.

Also there is an instruction LAB\((l)\) which does nothing, but is labelled \(l\), so that it can be the target of a jump.

CAL\((i, l)\) is used for calling the procedure, whose code starts at label \(l\). Again, let \(b\) be the contents of \(B\). The number \(i\) is used to find the static link \(b'\) which is to be put in the activation record of the called procedure. If \(i\) is 0, we take \(b'\) to be \(b\). This occurs when the called procedure is declared as a procedure local to the calling procedure. If \(i\) is 1, we take \(b'\) to be \(c(b)\), i.e. the static link of the called procedure will be the static link in the calling procedure; this occurs when the called and the calling procedures are declared at the same level, for example when a procedure calls itself. In general, i.e. for any \(i \geq 0\), \(b'\) is taken to be \(c^i(b)\), signifying that there exist procedures \(P_0, P_1, \ldots P_i\) such that the calling procedure is \(P_i\), the called procedure is declared local to \(P_0\) and \(P_{j+1}\) is declared locally to \(P_j\), for \(j\) satisfying \(0 \leq j \leq i - 1\).

Also as part of executing the CAL instruction, the return address is stored on the stack and the dynamic link of the called procedure is made to point to the base word of the activation record for the calling procedure. Then \(B\) is made to point to the beginning of the new activation record, and a jump to \(l\), the code of the called procedure, is performed.

The inverse of the CAL instruction is the return instruction, which is actually encoded as OPR\(0\) as in Wirth’s original PL/0 compiler. Following the dynamic link down the
stack, it resets the \( T \) and \( B \) registers to the values they had before the call and jumps to the return address which was kept on the stack.

**Example 3.2** The code for the statement \( c := b + d \) in Figure 3 is \( \text{LOD}(2,4) \) \( \text{LOD}(1,4) \) \( \text{OPR} \ 2 \) \( \text{STO}(1,3) \). By looking at the pictures in Example #3.1, one sees that these load and store instructions really do access the right variables.

**Example 3.3** The code for the call \( Q \) in Figure 3 is \( \text{CAL}(2, \text{entrypoint3}) \), where \( \text{entrypoint3} \) is the label at the beginning of the code for \( Q \).

**Exercise 3.2** Why is the first argument to this \( \text{CAL} \) instruction 2?

**Exercise 3.3** What is the first argument to the \( \text{CAL} \) instruction which is the result of compiling \( \text{call Q} \) in the main program?

### 3.3 An interpreter for the SC Machine

The programs that are documented in these notes include an interpreter for the SC Machine. The signature is

```ml
signature SC_MACHINE =
  sig
    structure I : INST
    sharing type I.label = string
    type SC_state
    val load: I.inst list -> SC_state
    val step: SC_state -> SC_state (* execute one instruction *)
    val Step: int -> SC_state -> SC_state (* Step n : do up to n steps *)
    val finish: SC_state -> SC_state (* repeat executing one instruction
till halt instruction is encountered *)
  end;
```

To get access to the interpreter, start up an ML session in the directory `DirectCompiler` and type

```
use "buildSC.sml";
```

As a result, you will get access to the components of structures that match the signatures \( \text{INST} \) and \( \text{SC_MACHINE} \) above. At the end of the file `buildSC.sml` is an example of how one can construct a little SC Machine program by hand; it is called `prog` and it is actually the compiled code corresponding to Figure 3. You can use the function \( \text{load} \) to convert handwritten SC programs to executable form; then you can use the functions \( \text{step} \) and \( \text{Step} \) to step through the execution slowly; do **not** try \( \text{finish} \) yet (for reasons described below). Try for example typing:
3.3 An interpreter for the SC Machine

load prog;
step it;
step it;
step it;
Step 5 it;

It so happens that the SC Machine interpreter is not fully implemented; also, it contains a nasty mistake which results in an infinite loop, if you try typing finish it. The following projects concern completing the implementation and finding and correcting the mistake.

The SC Machine is found in the file SCMachine.

Exercise 3.4 You will see that multiplication and unary plus have not been implemented. Implement them.

Exercise 3.5 Step through the execution of prog using step and Step till you discover which SC Machine instruction is not correctly interpreted. What is the problem and how is it solved?
4 Elaboration

One can easily write a compiler directly from PL/0 abstract syntax to SC Machine code. However, certain aspects of the translation do not depend on the target machine. By splitting the translation into smaller phases we can obtain a separation of the part of the translation that is the same for a range of target machines from the machine specific part. We shall translate in two phases: elaboration and code generation. Elaboration is concerned with checking that all variables have been declared before use and that all identifiers are being used according to their type; for example, PL/0 does not allow procedure identifiers in expressions. Moreover, elaboration has the effect of decorating all occurrences of identifiers in the abstract syntax tree with so-called lexical indices that describe for each applied occurrence of an identifier which binding occurrence it refers to. If successful, elaboration results in a decorated abstract syntax tree.

The second phase, code generation, then transforms the annotated syntax tree to code for the machine in question. For every identifier occurrence, the lexical indices are converted to stack indices without the need for looking up the identifier in an environment.

The rest of this chapter is devoted to elaboration alone; in the subsequent chapters we shall present a number of different methods of code generation for a range of target machines.

4.1 Lexical indices

In PL/0, as in many other programming languages, occurrences of identifiers can be divided into two, namely binding vs. applied occurrences. In a statically scoped language, such as PL/0, every applied occurrence of an identifier refers to precisely one binding occurrence of that identifier. The scope rules of the language define how applied occurrences refer to binding occurrences. Elaboration annotates every applied occurrence of an identifier with a so-called lexical index; the purpose of this index is to refer explicitly to the right binding occurrence of that identifier. This is of course not entirely trivial, since a program may contain many different binding occurrences of the same identifier. We shall now explain what lexical indices are and how they are computed.

We start by making the distinction between applied and binding occurrences precise in the case of PL/0. Any occurrence of an identifier within a variable or constant declaration is binding. Moreover, in a procedure declaration procedure ident ; block ; the occurrence of ident is binding. All other occurrences are applied. (This includes all occurrences of identifiers in statements and expressions.)

Next, we associate with each phrase that occurs in the program a current level. The current level of the entire program is 0. Moreover, with one exception, whenever a phrase phrase is made up of immediate constituent sub-phrases, all of these inherit the current level of phrase. The exception is when phrase is of the form procedure ident ; block ;, in which case the current level of ident is the same as the current level of phrase but the current level of block is 1 plus the current level of phrase.

We shall not define the scope rules of PL/0, as we assume the reader is already
4.1 Lexical indices

![Program code]

Figure 4: An elaborated PL/0 program

familiar with block structured languages. However, we shall describe what lexical indices are in PL/0; the idea is general and applies to other statically scoped languages as well. Every identifier occurrence denotes either a constant, a variable or a procedure. Constant identifiers can simply be replaced by their value and we shall not annotate constant identifiers with lexical indices. We now treat variable and procedure identifiers in turn:

- **Variables:** The lexical level of a variable identifier is a pair \((\text{distance}, \text{offset})\) of natural numbers. Let \(o\) be an occurrence of an identifier \(\text{ident}_j\) in a variable declaration at current level \(i\) which declares distinct variables \(\text{ident}_0, \ldots, \text{ident}_n\) in that order. Then the lexical index is defined to be the pair \((i, j)\). Here \(i\) is often referred to as the *declaration level* of the identifier. Next, let \(o\) be an applied occurrence of an identifier at current level \(i\). Then the lexical index of \(o\) is \((i - i_0, j_0)\), where \((i_0, j_0)\) is the lexical index of the binding occurrence of the identifier. Intuitively, the \(i - i_0\) is the distance from the applied occurrence to the binding occurrence counted as the number of nested procedure declarations one has to cross.

- **Procedure identifiers:** The lexical index of a procedure identifier is a pair \((i, l)\), where \(i\) is a natural number and \(l\) is a *unique label*. Here \(i\) is determined as in the case of variable identifiers. In general, there may be many different procedures with the
same name in a program. The unique label can be thought of a a procedure identifier which uniquely names the procedure within the entire program. The unique label is the same in the lexical index of a binding occurrence and in any applied occurrence that refers to it.

**Example 4.1** The result of elaborating the program in Figure 3 is shown in Figure 4, where we have shown all the lexical indices.

Below, we present the signature of annotated abstract syntax trees.

```plaintext
signature ANN_SYNTAX =
  sig
  type ident and number and distance and offset and label
  sharing type number = distance = offset = int
      and type ident = string
  (* Any occurrence of the type ident in the following is to allow pretty printing only *)
  datatype program = PROGRAM of block
    and vardec = VAR of int (* number of variables declared *)
    and procdec = PROCDEC of (ident * label * block)list
    and block = BLOCK of vardec * procdec * statement
    and statement = ASSIGN of ident * distance * offset * expression
      | SEQ of statement * statement
      | IF of expression * statement
      | WHILE of expression * statement
      | CALL of ident * distance * label
      | EMPTYSTAT
    and expression = IDENT of ident * distance * offset
      | NUM of number
      | APPMONOP of monop * expression
      | APPBINOP of expression * binop * expression
    and monop = ODD | UPLUS | UMINUS
    and binop = EQUALS | NOTEQUAL | LTH | GTH | LEQ | GEQ
      | PLUS | MINUS | TIMES | INTDIV;
  end;
```

### 4.2 The static environment

Elaboration of most PL/0 phrases is relative to a *static environment* which records information about identifiers. In the literature, the term *symbol table* is often used to mean
4.3 The elaborator

more or less the same thing. The kind of information kept for each identifier can be summarised as follows:

\[
\text{datatype statinfo} = \begin{cases} 
\text{ISCONST of int} \\
\text{ISVAR of \{declev: int, offset : int\}} \\
\text{ISPROC of \{declev: int, codeadr: label\}} 
\end{cases}
\]

An identifier can either denote a constant (in which case the value of the constant is recorded), or a variable or a procedure (in which case the lexical index of the binding occurrence is recorded).

In the implementation the empty environment is called \texttt{emptyenv}. The function \texttt{mkenv} takes an identifier and a value of type \texttt{statinfo} and produces a static environment containing a single binding. The operation \texttt{++} is a binary operation on environments: \(E_1 \mathbin{++} E_2\) is the environment so that \((E_1 \mathbin{++} E_2)(\texttt{ident})\) means \(E_2(\texttt{ident})\) if this is defined, and otherwise \(E_1(\texttt{ident})\), i.e. the bindings in \(E_2\) shadow over the bindings of \(E_1\). The complete signature for static environments appears below.

signature STATENV =

sig
    type ident and label (* inherited *)
    type statenv
    datatype statinfo = ISCONST of int
                       | ISVAR of \{declev: int, offset : int\}
                       | ISPROC of \{declev: int, codeadr: label\};

    val emptyenv : statenv
    val \mathbin{++} : statenv * statenv \rightarrow statenv
    val mkenv: ident * statinfo \rightarrow statenv
    datatype 'a lookupResult = NOTFOUND | FOUND of 'a
    val lookup: statenv * ident \rightarrow statinfo lookupResult
    val enter: (ident * statinfo) * statenv \rightarrow statenv
end;

4.3 The elaborator

The signature of the elaborator is as follows:

signature ELAB =

sig
    structure A: ABSTSYNTAX
    and B: ANN_SYNTAX
type statenv (* inherited *)  
val Eprogram: A.program -> B.program  
and Econstdec: A.constdec -> statenv  
and Evardec: A.vardec * level -> statenv * B.vardec  
and Eprocdec: A.procdec * statenv * level -> statenv * B.procdec  
and Eblock: A.block * statenv * level -> B.block  
and Estatement: A.statement * statenv * level -> B.statement  
and Eexp: A.expression * statenv * level -> B.expression  
and Emonop: A.monop -> B.monop  
and Ebinop: A.binop -> B.binop  
end;

Note that most of the elaborator functions take a level as a parameter, namely the current level of the phrase they elaborate.

Evardec and Eprocdec both produce static environments, which is to be expected since they elaborate declarations. In these cases, the current level is bound to the identifiers that are being declared.

All elaborator functions that work on phrases which can contain applied occurrences of identifiers (Eprocdec, Eblock, Estatement and Eexp) take a static environment which they can use to look up identifiers to find their declaration levels. The distance to an identifier is computed as the difference between the current level and the declaration level of the identifier.

The code for the elaborator appears below. The subsequent exercises bring out the main points of interest.

functor Elab
  (structure L: LIST
   structure A: ABSTSYNTAX
   structure B: ANN_SYNTAX
   structure Lab: LABEL
   structure E : STATENV
     sharing type E.ident = A.ident = B.ident
     and type E.label = Lab.label = B.label
   structure Crash: CRASH): ELAB =
struct
  structure A = A
  structure B = B
  type level = int; (* declaration level of variables *)
  type statenv = E.statenv

infix ++
val op ++ = E.++
open A

fun Eprogram(PROGRAM block)= B.PROGRAM(Eblock(block,E.emptyenv,0))
and Econstdec(CONST conbinds)=
  L.fold (fn ((ident,num),E) => E.enter((ident,E.ISCONST num),E))
  conbinds
  E.emptyenv
and Evardec(VAR idents,level) =
  let val n = length idents
    fun Eidentdec([],count) = E.emptyenv
    | Eidentdec(ident::rest,count) =
      E.enter((ident,E.ISVAR{declev = level, offset = count}),
               Eidentdec(rest,count+1))
  in
    (Eidentdec(idents,0), B.VAR n)
  end
and Eprocdec(PROCDEC procdecs, E, level)=
  let
    fun Eprocdec'((ident,block),(E,procdecs')) =
      let val lab = Lab.newlabel "entrypoint";
      val E1 =
        E.mkenv(ident,
                 E.ISPROC{declev = level, codeadr = lab})
      val block' = Eblock(block,E ++ E1, level+1)
      in
        (E ++ E1, (ident,lab,block') :: procdecs')
    end
    val (E',procdecs') = L.foldL Eprocdec' procdecs (E, [])
  in
    (E', B.PROCDEC procdecs')
  end

and Eblock(BLOCK(constdec_opt, vardec_opt, procdec_opt, statement),
           E,level)=
  let
    val E1 = case constdec_opt of
      NONE => E.emptyenv
      | SOME constdec => Econstdec(constdec)
    val (E2, vardec') =
      case vardec_opt of
        NONE => (E.emptyenv, B.VAR 0)
val (E3, procdec') =
  case procdec_opt of
    NONE => (E.emptyenv, B.PROCDEC [])
  | SOME procdec =>
    Eprocdec(procdec, E ++ E1 ++ E2, level)
val statement' = Estatement(statement, E ++ E1 ++ E2 ++ E3, level)
in
B.BLOCK(vardec', procdec', statement')
end

and Estatement(statement, E, level)=
case statement of
  ASSIGN(id, exp) =>
    let val exp' = Eexp(exp, E, level)
    val (declevel, offset) =
      case E.lookup(E, id) of
        E.FOUND(E.ISVAR{declev=d, offset=offs}) => (d, offs)
      | _ => Crash.impossible("Variable not declared: " ^ id)
    in
      B.ASSIGN(id, level-declevel, offset, exp')
    end
  | SEQ(stmt1, stmt2) =>
    B.SEQ(Estatement(stmt1, E, level), Estatement(stmt2, E, level))
  | IF(exp, stmt) =>
    let val exp' = Eexp(exp, E, level)
    val stmt' = Estatement(stmt, E, level)
    in
      B.IF(exp', stmt')
    end
  | WHILE(exp, stmt) =>
    let val exp' = Eexp(exp, E, level)
    val stmt' = Estatement(stmt, E, level)
    in
      B.WHILE(exp', stmt')
    end
  | CALL id => (case E.lookup(E, id) of
      E.FOUNDE.ISPROC{declev=declev, codeadr=codeadr}) =>
        B.CALL(id, level-declev, codeadr)
    | _ => Crash.impossible(id ^ " is not a procedure")
  | EMPTYSTAT => B.EMPTYSTAT
and Eexp(exp, E, level)=
4.3 The elaborator

case exp of
  IDENT(id) => (case E.lookup(E, id) of
      E.FOUNDED(E.ISVAR{declev=declev,offset=offset}) =>
        B.IDENT(id, level-declev, offset)
      | E.FOUNDED(E.ISCONST n) =>
        B.NUM n
      | _ => Crash.impossible(id ^ "is a procedure")
    |
    | NUM(n) => B.NUM n
    | APPMONOP(monop,exp1) =>
      B.APPMONOP(Emonop monop, Eexp(exp1,E,level))
    | APPBINOP(exp1,binop,exp2) =>
      B.APPBINOP(Eexp(exp1,E,level), Ebinop binop, Eexp(exp2,E,level))

and Emonop monop =
  case monop of
    ODD => B.ODD
    | UPLUS => B.UPLUS
    | UMINUS => B.UMINUS

and Ebinop binop =
  case binop of
    EQUALS => B.EQUALS
    | NOTEQUAL => B.NOTEQUAL
    | LTH => B.LTH
    | GTH => B.GTH
    | LEQ => B.LEQ
    | GEQ => B.GEQ
    | PLUS => B.PLUS
    | MINUS => B.MINUS
    | TIMES => B.TIMES
    | INTDIV => B.INTDIV

end;

Exercise 4.1 What happens if the elaborater encounters an undeclared identifier in an expression?

Exercise 4.2 How is an identifier which is declared as a constant elaborated when occurring in an expression?

Exercise 4.3 Consider the elaboration of a block consisting of constant, variable and procedure declarations followed by a statement in a static environment $E$. Suppose the combined effect of the declarations is to produce a static environment $E'$. What is the static environment in which the statement should be elaborated?

Exercise 4.4 In which procedure is the current level is increased? Is it ever decreased?
Exercise 4.5  What happens if you try to elaborate an assignment \( P := 7 \), where \( P \) is a procedure? Improve the error message given.

Exercise 4.6  In order to elaborate a recursive function declaration, we need to elaborate its body. However, in order to elaborate the body of the function, the static environment must already have static information about the recursive procedure. How is that done in the elaborator?

Project 4.7  As you can discover by looking at Eprocdec, procedures that are declared in the same block are not considered mutually recursive (although each of them are recursive and can call any previously declared procedure in that block). Of course one could introduce a forward declaration to be used with mutually recursive procedures, but that would be a bit clumsy. Change the elaboration of procedure declarations such that procedures declared at the same level automatically are considered mutually recursive.
5 Direct Code Generation

We shall now describe the translation from annotated syntax trees to SC Machine code. The crucial observation is that there is an intimate relationship between the lexical indices calculated during elaboration and the addressing mechanism in the running system. This point is discussed in Section 5.1. The only other non-trivial question in the translation is how to handle flow of control in the code. One simple way of doing this is described in Section 5.2. Finally in Section 5.3 we present part of the ML implementation with examples of compiled code.

5.1 The significance of lexical indices for the running system

The crucial significance of lexical indices is that they provide all the information which is required to access variables and call procedures in the running system. Consider an applied occurrence of an identifier in the annotated syntax tree decorated with lexical index \((i, j)\). Then \(i\) indicates the number of indirections one needs to follow along the static chain in order to find the most recent activation record for the block in which the identifier is declared. This is true regardless of whether the identifier denotes a variable or a procedure. In the case of procedures, no value representing the procedure is actually kept in the stack; yet, the two things that are needed to call a procedure (namely the address of the code of the procedure and a pointer to the environment in which the procedure is to run) are available when the procedure is called: the label of the code is part of the call instruction itself, and the environment pointer is obtained by following \(i\) directions through the static link.

At this point it is appropriate to consider to which extent the idea of lexical indices is tied exactly to the SC Machine. Certain aspects of the SC Machine are orthogonal to the environment management described above. For example, the fact that all data is kept in the stack, including temporary values that arise during expression evaluation, is not essential at all. One could have a linked list of environment records that contain just the static link plus local variables in each record, and essentially the same addressing scheme would work. Also, the addressing mechanism does not really require that the static chain be traversed each time an identifier is accessed. A famous arrangement, due to Dijkstra, represents the static chain by an array, DISPLAY, which is such that DISPLAY\([i]\) is the base of the most recent activation record of the textually surrounding block which has current level \(i\). This makes identifier access fast, at the expense of setting up the display each time a new environment is installed. It is clear, therefore, that the notion of lexical indices is useful for a range of target machine architectures. On the other hand, it is intimately tied to the idea of the environment being essentially a linked list (perhaps indexed by a display or similar arrangement).

There are perfectly sensible alternative ways of organising the environment in the running system which do not correspond to the way we elaborate programs. In particular, upon entering a block, one could build values that represent the locally declared procedures as follows. For every procedure we build a value which contains pointers to the
values of all the identifiers that occur free in the procedure in a particular order which is assumed inside the body of the local procedure. This gives faster access to variables but makes block activation more expensive. Let us refer to this handling of free variables as shallow pointer binding. It is similar to creating a display which is stored with the procedure value.

If PL/0 did not allow assignment to variables (in ML one cannot change the value of a variable once the value has been bound to the variable) an even simpler possibility exists, namely to copy the value of the variable into the procedure value. This results in an object which is completely independent of any global environment. This strategy is called shallow binding. The value representing the procedure is called a closure, although this term is used generally to denote a value consisting of an environment and the address of the code of the procedure, whether or not the environment is represented by a pointer to a global environment (as in the SC Machine) or as a list of values. The arrangement used in the SC Machine is commonly called deep binding. Of course, if one used shallow (pointer) binding, one could still compute lexical indices that refer to this handling of the environment. The following chapters will present a variety of code generation methods, but they will all be based on deep binding. Shallow binding and shallow pointer binding will not be covered further.

Returning to the explanation of the lexical index \((i, j)\) of applied occurrences of identifiers, \(j\) is the relative address of the variable within the activation record but for the constant amount of space taken up by pointers at the beginning of the activation record. In the SC Machine this constant amount is 3 (one word for the static link, one for the dynamic link and one for the return address).

It is appropriate to note at this point that the environment handling of the SC Machine relies on properties of PL/0 which fail to hold of more powerful languages. One example is the way the environment of a called procedure is obtained from the environment of the calling procedure. This mechanism relies on the fact that the environment of a called procedure is always represented by an initial segment of the static chain of the calling procedure. This would not be the case if it was possible for a procedure which was declared at level 5, say, to call a procedure which was declared at level 10, say. If PL/0 procedures had parameters and procedures were allowed as parameters to procedures, this situation could arise. Also consider what would happen if PL/0 had a more general call statement:

\[
\text{call } \text{exp}
\]

where \(\text{exp}\) could be an expression which was complex enough that it was not decidable at compile time which procedure would be called, e.g. \(\text{call (if cond then P else Q)}\). Then the values of the distance and the label would not be apparent a compile time and would have to be produced as a result of evaluating \(\text{cond}\) at run time.

We shall address the problem of procedures as values later in these notes.
5.2 Flow of control

This section describes a very simple, in fact almost naïve, handling of program points and flow of control, based on the simple observation that the unique labels with which procedure identifiers were labelled during elaboration can be used as symbolic labels when representing SC Machine code in a form that resembles symbolic machine language. Moreover, we shall mark other program points to which jumps may be required by symbolic labels.

The code for a compound phrase is simply obtained by filling in the code fragments of the constituent phrases in a fixed schema of SC code characteristic for the compound phrase in question. For example, the schema for a variable in an expression is simply LOAD(i, j + 3) if the variable is decorated by lexical index (i, j). The schema for an application of a binary operator $exp_1$ binop $exp_2$ is $code_1 & code_2 & inst$, where $code_i$ is the code for $exp_i$, $(i = 1, 2)$, & is concatenation of code and $inst$ is the instruction in the machine corresponding to $binop$. (Notice that the SC Machine conveniently has one instruction for each of the operators in PL/0.) As a final example of code schemas, the code for if $exp$ then $stmt$ is

$$code_1 & JPC \ lab \ & code_2 & LAB \ lab$$

where $code_1$ is the code for $exp$, $code_2$ is the code for $stmt$ and $lab$ is a fresh symbolic label.

This approach, loosely characterised by the fact that an abstract syntax tree is mapped to the same result regardless of its context, we shall refer to as direct code generation. The approach is clearly somewhat simplistic. For example, an identifier in an expression is translated into code for fetching it from the stack and putting it on the top of the stack, irrespective of the fact that the variable might have fetched recently and is in fact available somewhere in a more accessible place. Also, the indiscriminate generation of symbolic labels gives clumsy looking symbolic machine code and a rather crude handling of flow of control, as subsequent examples will illustrate. Therefore, real compilers hardly ever use direct code generation. One of the interesting challenges of code generation is exactly to find nice ways of representing useful information about the computation of which the phrase is a part and generate code which is appropriate in the given context. However, direct code generation deserves to be spelled out, mainly because it is extremely simple.

5.3 Implementation

The direct code generator needs to know about the instructions of the machine, the annotated syntax and the following types and operations concerning the code:

```latex
signature CODE =
  sig
    type inst
    type code
```
The important thing is that the code generator does not need to know how the type `code` is implemented; nor does it need to know how the concatenation operator `&` and the other operations on `code` are implemented. If you check the implementation which matches the above signature (look in PL0/DirectCompiler/Code.sml) you will find that `code` is just implemented as lists of instructions and that the concatenation is list append, but the code generator does not need to know this and it is much better that is doesn’t. A `code environment` (see the signature) is intended to be a map from `headers` (which are strings used for printing purposes) to complete streams of code, one stream for each block in the program. Again, the actual implementation is not important.

We can now specify the signature of the direct code generator:

```ml
signature CODEGEN =
  sig
    structure A: ANN_SYNTAX
    type code and codenv
    val Cprogram: A.program -> codenv
    val Cvardec: A.vardec -> code
    val Cprocdec: A.procdec * codenv -> codenv
    val Cblock: A.block * codenv -> code * codenv
    val Cstatement: A.statement -> code
    val Cexp: A.expression -> code
    val Cmonop: A.monop -> code
    val Cbinop: A.binop -> code
  end;
```

The types of these code generators leave us in no doubt that they really are very direct: two of the code generators take a code environment as an argument, but the intention is that this is just in order to be able to contribute new code streams to it due to locally declared procedures.

The code generator functions are presented below. The appropriate way of expressing the data abstraction of the code and code environment types in ML is by forming a functor.
which has a code structure as a formal parameter. In that way, the ML type checker can ensure that the code generators do not rely on any particular structure; instead they are expressed in terms of a hypothetical structure $C$ which is specified in the formal argument signature $\text{CODE}$. All that is known about $C$ is what the structure specifications in the parameter list of the functor. For example, without having to change one line of the functor, one could change the representation of $\text{code}$ and the operations that operate on it.

The direct code generators are as presented below. Notice that one must make explicit the identification of instruction labels and unique procedure labels in order to get the functor to type check.

```ml
defunctor CodeGen
  (structure L: LIST
   structure A: ANN_SYNTAX
   structure Lab: LABEL
   structure I: INST
   structure C: CODE
   sharing type C.inst = I.inst
   and type Lab.label = I.label = A.label
   structure Crash: CRASH): CODEGEN =
struct
  structure A = A
  type code = C.code
  type codenv = C.codenv
  infix &
  val op & = C. &
  val n0 = 3 (* every stack frame contains at least 3 pointers *)
  open A I C

  fun Cprogram(PROGRAM block) =
    let val (code, codeE) = Cblock(block, emptyenv)
      in enter("main program", code), codeE)
    end
  and Cvardec(VAR n) = mkcode(INT(n + n0))
  and Cprocdec(PROCDEC procdecs, codeE) =
    let fun Cprocdec'((ident, label, block), codeE) =
      let
        val (code, codeE') = Cblock(block, codeE)
        val code' = mkcode(LAB label) &
          code &
          mkcode(OPR 0) (* return instruction *)
      in
        enter("Code for " ^ ident, code'), codeE')
end

\[ \text{L.foldL} \text{Cprocdec'} \text{procdecs codeE} \]

end

and Cblock(BLOCK(vardec, procdec, statement), codeE)=
  let
    val code1 = Cvardec(vardec)
    val codeE2 = Cprocdec(procdec, codeE)
    val code3 = Cstatement(statement)
  in
    (code1 & code3, codeE2)
  end

and Cstatement(statement)=
  case statement of
    ASSIGN(id,distance,offset,exp) =>
      Cexp(exp) & mkcode(STO(distance,offset+n0))
    | SEQ(stmt1,stmt2)=>
      Cstatement(stmt1) & Cstatement(stmt2)
    | IF(exp,stmt) =>
      let val code1 = Cexp(exp)
          val lab = Lab.newlabel "falsebranch";
          val code2 = Cstatement(stmt)
      in
        code1 & mkcode(JPC lab) & code2 & mkcode(LAB lab)
      end
    | WHILE(exp,stmt) =>
      let val lab1 = Lab.newlabel "whilentry"
          val code1 = Cexp(exp)
          val code2 = Cstatement(stmt)
          val lab2 = Lab.newlabel "whileexit"
      in
        mkcode(LAB lab1) & code1 & mkcode(JPC lab2) & code2 &
        mkcode(JMP lab1) & mkcode(LAB lab2)
      end
    | CALL(id,distance,label) =>
      mkcode(CAL(distance, label))
    | EMPTYSTAT => emptycode
  end

and Cexp(exp) =
  case exp of
    IDENT(id,distance,offset) =>
5.3 Implementation

```
mkcode(LOD(distance, offset+n0))
| NUM(n) => mkcode(LIT n)
| APPMONOP(monop,exp1) => Cexp(exp1) & Cmonop monop
| APPBINOP(exp1,binop,exp2)=>
    Cexp(exp1) & Cexp(exp2) & Cbinop binop

and Cmonop monop =
  case monop of
    ODD => mkcode(OPR 6)
| UPLUS => emptycode
| UMINUS => mkcode(OPR 1)

and Cbinop binop =
  case binop of
    EQUALS => mkcode(OPR 8)
| NOTEQUAL => mkcode(OPR 9)
| LTH => mkcode(OPR 10)
| GTH => mkcode(OPR 12)
| LEQ => mkcode(OPR 13)
| GEQ => mkcode(OPR 11)
| PLUS => mkcode(OPR 2)
| MINUS => mkcode(OPR 3)
| TIMES => mkcode(OPR 4)
| INTDIV => mkcode(OPR 5)
end;
```

Example 5.1 The result of translating the annotated program from Figure 4 (page 15) is shown in Figure 6.

Exercise 5.1 What is the result of compiling the program in Figure 2 (page 5)?

Project 5.2 Now try to load the entire compiler and compile some programs. The components of the direct compiler (i.e. the combination of the elaborator and the direct code generator) are found in the directory PL0/DirectCompiler and the testprograms (there are about 10 of them) are found in PL0/testprogs. In order to be able to load the test programs correctly from the DirectCompiler directory, make sure the variable called path in the file Compiler.sml leads to testprogs. From the DirectCompiler directory, start an ML session and type:

```
use "build.sml"; (* this loads and builds the compiler *)
readall(); (* loads all testprograms *)
compileall "myfile"; (* compile all testprogs and put result on myfile *)
```

Now you can use an editor to see the test programs and what they were compiled to.
The operations the compiler provides are summarised in the following signature:

```plaintext
signature COMPILER =
  sig
    structure A : ABSTSYNTAX
    val compile: A.program -> string
    val path : string ref (* path of test programs *)
    val enter: string * A.program -> unit (* load a test program *)
    val readall: unit -> unit (* load all test programs *)
    val compileall: string (* output file name*) -> unit (* compile loaded test programs *)
    val clearall: unit -> unit (* forget any loaded test program *)
  end;
```

**Project 5.3** Modify the compiler so that the generated code automatically is commented. For each load and store instruction, the identifier should be used as a comment. Also, the opcodes to OPR instructions should be commented by saying which operation it codes for. Think carefully about where the printing is best done and which signatures and functors to alter.

The code produced by the direct code generator often has superfluous labels and jumps. For example, compiling the program in Figure 5 results in the code in Figure 7. Notice that there are two labels at the same program point (`whileexit17` and `falsebranch14`) and that there is a jump (`JPC falsebranch16`) to a jump (`JMP whileentry15`).

It can be argued that the excess of labels is not too serious, since they take up no space in the final machine language program. Also, one can argue that the extra jumps are not too serious, since they can be removed by a code optimizer at a later stage or even

```plaintext
var a;
a:= 10;
if a=10 then
  while a>0 do begin
    a:= a-1;
    if odd a then a:=a-1
  end;
a:= 5.
```

Figure 5: A PL/0 program with non-trivial flow of control
left in the code without serious loss of efficiency. However, these extra labels and jumps are simply not pleasing to the eye, and that should be enough reason for thinking that something is not quite right about the current code generator. There are other and more serious deficiencies in the current code generator; for example it sometimes results in many unnecessary return operations, a fact which can make recursive procedures unnecessarily expensive to use.

In the next section we explore an alternative to direct code generation which is more general and has the added benefit that it easily avoids unnecessary labels, jumps and returns.
Code for R
entrypoint4

    INT 3
    LOD(2,4)
    LOD(1,4)
    OPR 2
    STO(1,3)
    LOD(1,3)
    LOD(2,3)
    OPR 10
    JPC falsebranch5
    LOD(1,3)
    STO(2,4)
    CAL(2,entrypoint3)

falsebranch5

    OPR 0

Code for Q
entrypoint3

    INT 5
    LIT 1
    STO(0,4)
    CAL(0,entrypoint4)
    OPR 0

main program

    INT 5
    LIT 3
    STO(0,3)
    LIT 0
    STO(0,4)
    CAL(0,entrypoint3)

Figure 6: The SC Machine code for the program in Figure 4 (page 15)
5.3 Implementation

main program

INT 4
LIT 10
STO(0,3)
LOD(0,3)
LIT 10
OPR 8
JPC falsebranch14

whileentry15

LOD(0,3)
LIT 0
OPR 12
JPC whileexit17
LOD(0,3)
LIT 1
OPR 3
STO(0,3)
LOD(0,3)
OPR 6
JPC falsebranch16
LOD(0,3)
LIT 1
OPR 3
STO(0,3)

falsebranch16

JMP whileentry15

whileexit17

falsebranch14

LIT 5
STO(0,3)

Figure 7: The result of compiling the program in Figure 5
6 Code Generation using Continuations

In this section we present the idea of using continuations in code generation. This has the effect of eliminating unnecessary labels and jumps. Also, unnecessary returns are avoided in the case of tail recursive procedures.

The stack of the SC Machine is split into two data structures, namely an environment and a stack. The resulting machine is called the SEC Machine. The $T$ and $B$ registers are explicitly manipulated by SEC Machine instructions. Whereas the call and return instructions of the SC Machine implicitly did a number of operations on these two registers, such operations are done explicitly in the SEC Machine. Thus it is possible to omit operations on these registers, when they are not necessary. The stack of the SEC Machine is used for evaluating expressions and conditions as in the SC Machine and for saving the $B$ and $T$ registers, when necessary. The environment holds the values of program variables in frames that are linked together with static links as in the SC Machine.

6.1 Continuations

The somewhat clumsy handling of jumps and labels in the direct code generator presented in the previous section is due to the fact that generation of code for a phrase happens without knowledge about what happens at run-time after the code for that phrase has come to its end. In compiling and if statement, for example, we generate a jump to the end of the code for the then branch without knowing whether that is where execution is really going to continue.

In denotational programming language semantics, a continuation is function which, if activated at some point in a computation, carries out the entire remainder of the computation leading to the final answer. A phrase in a source program may be executed more than once when the program is executed, each time using a different continuation (because different parts of the computation remains). For instance, a procedure does not in general have a unique continuation associated with it, for the procedure can be called from different places so that it has to continue at (i.e. return to) more than one place. This is why the return address is put on the stack, as a kind of parameter to the called procedure, so that it can know where to continue. However, for the purpose of compilation, we can distinguish between a few different kinds of continuations. In the case of PL/0, for example, we shall use the following definition of continuations:

\[
\text{datatype cont} = \text{GOTO of label} \\
| \text{NEXT} \\
| \text{RETURN}
\]

The idea is this: at any point during execution, we want to continue either at a known label (hence the \text{GOTO of label}), or at the physically next instruction (hence \text{NEXT}) or we want to return to a calling procedure (hence \text{RETURN}).
6.2 Generating code backwards

The code generated for a phrase is now dependent on a continuation. For example, consider the previously mentioned problem of generating a conditional jump instruction when compiling an if statement. If the continuation is GOTO \( l \), for some label \( l \), we just generate JPC(\( l \)); if the continuation is RETURN we will have to generate a jump to a return instruction. If the continuation is NEXT, however, we do not know where to continue; we could of course generate a label as in the direct code generator and generate a jump to it, but that would lead to the ugly multiple labels we saw in the previous section. The proper thing to do in this case is to generate a JPC instruction with a “hole” in it which can be filled out later, when it is established with certainty what label we should jump to. This label may be provided by a following statement (for example, the code for a while statement always starts with a label) or it may have to be invented, if the code is followed by a stream of instructions that does not already have a label. Thus it is the concatenation of code fragments that generates labels, not the compilation of the individual subphrases.

6.2 Generating code backwards

But where is this continuation going to come from? The answer is as baffling as it is simple: by generating the code backwards. The only continuation we know from the beginning is the finish continuation, namely that the program should halt (or do a return, in case we regard the program itself as a procedure). Suppose the program is a sequence of statements \( S_1; S_2 \). The idea now is to generate code for \( S_2 \) using the finish continuation as a parameter; the result is going to be some code and a new continuation, which we then use to generate code for \( S_1 \).

One way of implementing this would be to change the types of the code generators to involve continuations, e.g.

\[
\text{val Cstatement: statement * cont -> code * cont}
\]

so that \text{Cstatement} would be implemented by something like

```ml
and Cstatement(statement, cont)=
  case statement of
    ...
    | SEQ(stmt1,stmt2)=>
      let val (code2, cont2) = Cstatement(stmt2, cont)
      val (code1, cont1) = Cstatement(stmt1, cont2)
      in
        (code1 & code2, cont1)
      end
```

Notice how the continuation is propagated backwards: the continuation cont which obtains after statement is propagated to stmt2 resulting in a new continuation cont2 which is used in generating code for stmt1.

While we could proceed as outlined above, there is a much more elegant solution. We can change the meaning of the type code without changing the type of the code generator functions themselves. From now on, let us by fixedcode refer to the kind of code which was generated by the direct code generators and let us by code refer to the thing that the continuation style code generators produce. Then the basic idea is to define:

\[
\text{type code} = \text{cont} \rightarrow \text{fixedcode} \times \text{cont}
\]

In other words, a piece of code is a function \( f \) which produces some fixed code and a new continuation if applied to a given continuation. With the above definition of code, if the code generator Cstatement has type statement \( \rightarrow \) code it really has type

\[
\text{val Cstatement: statement} \rightarrow \text{cont} \rightarrow \text{fixedcode} \times \text{cont}
\]

which is just the curried version of the type we first suggested.

Because code now takes a continuation as parameter and produces a (perhaps transformed) continuation as output, we say that any object of type code is a continuation transformer.

The types of code and the operations on them are specified by the following signature:

```signature CODE =
  sig
  type inst and code and fixedcode and cont (*NB*)
  val emptycode: code
  val mkcode: inst \rightarrow code
  val & : code * code \rightarrow code (* sequential combination*)
  val && : code * code \rightarrow code (* parallel combination*) (*NB*)
  val fix: code \rightarrow cont \rightarrow fixedcode (*NB*)
  val fixcont: code \rightarrow cont \rightarrow code (*NB*)
  type header sharing type header = string
  type codenv
  val emptyenv: codenv
  val enter: (header * fixedcode) * codenv \rightarrow codenv (*NB*)
  val showcodenv: codenv \rightarrow string
  end;
```

There are two combinators on code, namely \& which is used for sequential composition of code and \&\& which is used for combining pieces of code in parallel; the latter operator
6.2 Generating code backwards

is used when generating code of branching phrases, i.e. `while` and `if` statements. The function `fix` is used to specialise code to a known continuation yielding fixed code. The function `fixcont` is supposed to work as follows: `fixcont(code)(cont)` is the code which always has continuation `cont`, i.e. for all `cont'`, `fix(fixcont(code)(cont)) cont' = fix(code)(cont)`. This is used when the continuation of a phrase is known, for example when compiling the body of a `while` statement.

The operations `&` and `&&` are operations on functions rather than on fixed code. Of course, the code generator is a functor which does not know the actual declaration of type `code` and the operations on it, so we can reuse the code generator functor from the direct code generator with a minimum of changes. The signature of the code generator functor is unchanged (see Section 5.3). The entire code generator functor follows below; only parts labelled with `(*NB*)` are different from in the direct code generator. Notice that the code generator itself now is actually smaller and simpler than the previous version (compare the treatment of while loops, for example). All the pasting together of code, including generation of jumps, returns and labels now takes place in the structures on which `CodeGen` is parameterised.

```plaintext
functor CodeGen
  (structure L: LIST
   structure A: ANN_SYNTAX
   structure Lab: LABEL
   structure I: INST
   structure C: CODE
   structure Crash: CRASH
   structure Cont: CONT (*NB*)
     sharing type C.inst = I.inst
     and type Lab.label = I.label = A.label = Cont.label
     and type Cont.cont = C.cont): CODEGEN =
struct
  structure A = A
  type code = C.code
  type fixedcode = C.fixedcode
  type codenv = C.codenv
  infix & &
  val op & = C. &
  val op && = C.&&
  val n0 = 1 (* every stack frame contains 1 pointer *) (*NB*)
open A I C Cont
fun Cprogram(PROGRAM block)=
  let val (code,codeE) = Cblock(block,emptyenv)
in enter(("main program", fix code RETURN), codeE) (*NB*)
```

and Cvardec(VAR n) = mkcode(INT(n + n0))
and Cprocdec(PROCDEC procdec, codeE) =
  let fun Cprocdec’((ident, label, block), codeE) =
    let
      val (code, codeE’) = Cblock(block, codeE)
      val code’ = mkcode(LAB label) & code
      (* no return instruction *) (*NB*)
    in (*NB*)
      enter(("Code for " ^ ident, fix code’ RETURN), codeE’)
    end
  in 
    L.foldL Cprocdec’ procdec codeE 
  end

and Cblock(BLOCK(vardec, procdec, statement), codeE) =
  let
    val code1 = Cvardec(vardec)
    val codeE2 = Cprocdec(procdec, codeE)
    val code3 = Cstatement(statement)
  in
    (code1 & code3, codeE2)
  end

and Cstatement(statement) =
  case statement of
    ASSIGN(id, distance, offset, exp) => (*NB*)
      Cexp exp & mkcode(STO(distance, offset + n0))
    | SEQ(stmt1, stmt2) => Cstatement(stmt1) & Cstatement(stmt2)
    | IF(exp, stmt) => Cexp exp && Cstatement stmt (*NB*)
    | WHILE(exp, stmt) => (*NB*)
      let val lab = Lab.newlabel "whilentry"
        val code1 = Cexp exp
        val code2 = fixcont (Cstatement stmt) (GOTO lab)
      in
        (mkcode(LAB lab) & code1) && code2
      end
    | CALL(id, distance, label) => mkcode(CAL(distance, label))
    | EMPTYSTAT => emptycode
  end

and Cexp(exp) =
  case exp of
    IDENT(id, distance, offset) =>
6.3 Code containing holes

When generating fixed code, one cannot always know which labels to put in branching instructions such as JPC. Intuitively, the natural fixed code for

\[
\text{if } a = 0 \text{ then}
\begin{align*}
\text{if } a = 1 \text{ then } b &:= 0
\end{align*}
\]

given continuation NEXT is the following fixed code, where • is a “hole”, i.e. a place holder for an unknown label

\[
\begin{align*}
\text{LOD}(0,3) &\quad a \\
\text{LIT} &\quad 0 \\
\text{OPR} &\quad 8 \\
\text{JPC} &\quad \bullet \\
\text{LOD}(0,3) &\quad a \\
\text{LIT} &\quad 1 \\
\text{OPR} &\quad 8
\end{align*}
\]
The bullet can then be replaced by a label when the label becomes known. This operation is called backpatching. In a piece of fixed code, there can be any number of holes, but they all stand for the same unknown label. Thus, another way of viewing such an incomplete piece of fixed code is as a function of one variable, the variable being the hole. In other words, a hole is represented by a bound label variable, while an absolute label is a free label. For example, the above fixed code can be represented by the function

\[
\text{fn lab =>}
\]

\[
\begin{align*}
&\text{LOD(0,3) a} \\
&\text{LIT 0 0} \\
&\text{OPR 8 = JPC lab} \\
&\text{LOD(0,3) a} \\
&\text{LIT 1 1} \\
&\text{OPR 8 = JPC lab} \\
&\text{LIT 0 0} \\
&\text{STO(0,4) b:=}
\end{align*}
\]

We shall represent fixed code in our compiler as outlined above. More precisely, the type code is defined as follows:

```plaintext
datatype 'a option = NONE | SOME of 'a;
datatype stream = COMPLETE of inst list
| INCOMPLETE of label -> inst list

type fixedcode = label option * stream
  * reg list (* needs *)
  * reg list (* modifies *)

type code = cont -> fixedcode * cont
```

A piece of fixed code is a quadruple \((lopt, s, n, m)\); if present, \(lopt\) is the label with which the code starts; \(s\) is a (perhaps incomplete) stream of instructions; \(n\) is a list of registers, namely the registers needed by the stream and \(m\) is the list of registers modified by the stream. The rôle of \(m\) and \(n\) will be described later.

### 6.4 Sequential composition

The operation \& relies on a slightly more basic operation \% which composes pieces of fixed code sequentially, performing backpatching automatically, when required.
(* Backpatching *)

fun backpatch(lab,(lopt,COMPLETE _, n,m)): fixedcode = 
  Crash.impossible "backpatching of complete code"
| backpatch(lab,(lopt,INCOMPLETE f, n,m)) = 
  (lopt, COMPLETE(f lab), n, m)

fun prefix(lab, (_, COMPLETE insts, n, m)): fixedcode = 
  (SOME lab, COMPLETE(LAB lab :: insts), n, m)
| prefix(lab, (_, INCOMPLETE f, n, m)) = 
  (SOME lab, INCOMPLETE(fn lab' => LAB lab :: f lab'), n, m)

(* Appending two fragments of code in sequence *)

infix %

(* fixedcode1 % fixedcode2 concatenates two pieces of fixed code, *)
(* backpatching automatically, if required. *)

fun comb1(i1, COMPLETE i2) = COMPLETE(i1 @ i2)
| comb1(i1, INCOMPLETE f2) = INCOMPLETE(fn lab => i1 @ f2 lab)

fun isempty((_,COMPLETE[],_,_): fixedcode) = true
| isempty _ = false

fun (q1: fixedcode) % (q2: fixedcode) : fixedcode = 
  if isempty q1 then q2
  else if isempty q2 then q1
  else (* both q1 and q2 are non-empty *)
    case (q1,q2) of 
      ((l1_opt, COMPLETE i1,n1,m1), (_, s2, n2, m2)) => 
        (l1_opt, comb1(i1,s2),
         union(n1, setminus(n2,m1)), union(m1,m2))
      | ((_,INCOMPLETE f1, _,_), (SOME(12), _, _, _)) =>
          backpatch(12,q1) % q2
      | ((_, INCOMPLETE f1, _,_), (NONE, _, _, _)) =>
          let val l2 = newlabel "seq"
          in backpatch(12, q1) % prefix(12, q2)
          end
The rest of this subsection is devoted to a detailed study of this code. In particular we shall prove that if fixed code is composed using \( \% \) then there will never be two labels immediately following each other.

In what follows, definitions of properties of lists of instructions are extended to apply to streams and fixed code in the natural way; in particular, a stream of the form \( \text{INCOMPLETE}(f) \) is said to have some property if \( f(lab) \) has that property, for all labels \( lab \).

A list of instructions is nice if it does not end with a label and nowhere contains two labels right after each other.

A label \( lab \) begins a list \( l \) of instructions if \( l \) is non-empty and the first instruction in \( l \) is \( \text{LAB} \ lab \).

A function \( f: \text{label} \to \text{inst list} \) is consistent if for all labels \( lab, lab1 \) and \( lab2 \), \( f(lab) \) is not empty and \( lab \) begins \( f(lab1) \) if and only if \( lab \) begins \( f(lab2) \). A stream is consistent if it is of the form \( \text{COMPLETE} \ 1 \) or is of the form \( \text{INCOMPLETE} \ f \) and \( f \) is consistent. A fixed code \( \text{fcode} = (\text{lopt},s,m,n) \) is consistent if \( s \) is consistent and \( s \) begins with label \( lab \) if and only if \( \text{lopt} \) is \( \text{SOME}(lab) \).

**Exercise 6.1** Prove the following fact. Let \( \text{fcode} \) be a consistent piece of fixed code. Then \( \text{isempty}(\text{fcode}) \) evaluates to true iff \( \text{fcode} \) is empty.

**Exercise 6.2** Prove the following fact. Let \( i1 \) be a list of instructions and let \( s2 \) be a stream. If \( i1 \) and \( s2 \) are nice then \( \text{comb1}(i1, s2) \) is nice. If \( i1 \) and \( s2 \) are consistent then \( \text{comb1}(i1, s2) \) is consistent.

**Exercise 6.3** Prove the following fact. Let \( q1 \) and \( q2 \) be consistent and nice pieces of fixed code. Then \( q1 \% q2 \) is consistent and nice.

### 6.5 Parallel composition

When combining pieces of code in sequence, any hole in the first piece is backpatched to beginning of the second piece, if the second piece is nonempty. When combining pieces of code in parallel the holes of the combined code are the union of the holes for each of the constituent pieces, without any backpatching. The basic operation on fixed code that makes this combination is called \( \%\% \).

```
infix \%\% 

fun comb2(COMPLETE inst1, COMPLETE inst2) = COMPLETE (inst1 @ inst2) | comb2(COMPLETE inst1, INCOMPLETE f2) = INCOMPLETE (fn lab => inst1 @ f2 lab) | comb2(INCOMPLETE f1, COMPLETE inst2) = INCOMPLETE (fn lab => f1 lab @ inst2) | comb2(INCOMPLETE f1, INCOMPLETE f2) = INCOMPLETE (fn lab => f1 lab @ f2 lab)
```
6.6 Preserving the environment

The base register $B$ of the SEC Machine points to the base of the topmost environment frame. The top register $T$ points to the topmost word of the topmost environment frame. An environment frame of the SEC Machine is the same as an activation record for in the SC machine except that the return address and the dynamic link have been removed. Thus an environment frame consists of one pointer (the static link) plus one word for each variable in the block.

The stack can contain labels and values. Expressions are evaluated on the stack as before. The stack can be pushed and popped using the two instructions $\texttt{SAV(register)}$ (read “save” or “push”) and $\texttt{RST(register)}$ (read “restore” or “op”). Also, there is an instruction $\texttt{PLB(label)}$ (read “push label”) for pushing a label onto the stack.

The call and return instructions now do much less work. We assume a register $\texttt{CONT}$ (read “continue” or “continuation”) which can hold one label. The convention is that a returning procedure always finds its return label on the top of the stack. (As we shall see later, a calling procedure does not always push a return label.) Using the notation from Section 3, the call and return instructions are informally defined by

\[
\text{CAL}(n, \text{lab}) \equiv \text{store}(T + 1) := c^n(B) \quad \text{JMP lab}
\]

\[
\text{OPR 0 } \equiv \text{RST CONT} \quad \text{JMP CONT}
\]

Notice that these instructions do not push or change the value of the $T$ and $B$ registers. The idea is that we will be more reluctant to save and restore these registers. Suppose for example we want to combine two pieces of fixed code $\text{fcode1}$ and $\text{fcode2}$ in sequence. If $\text{fcode1}$ modifies $T$ and $B$ and $\text{fcode2}$ needs their values to be unchanged, then we wrap save and restore instructions around $\text{fcode1}$:

\begin{verbatim}
SAV B
SAV T
fcode1
RST T
RST B
fcode2
\end{verbatim}

For example, the result of compiling a sequential statement $\texttt{call P; x:= 5}$ relative to continuation $\texttt{NEXT}$ will be something like
because the code for the statement \( x := 5 \) needs the environment. On the other hand, the code for
\[
\text{call } P; \\
\text{while } 3 > 5 \text{ do begin end}
\]
will start
\[
\text{PLB seq} \\
\text{CAL(0, entrypont3)}
\]
\[
\text{seq} \\
\text{RST } T \\
\text{RST } B \\
\text{LIT } 5 \\
\text{STO } (1, 3)
\]
Note that we do not save and restore \( B \) and \( T \), since \text{while } 3 > 5 \text{ do begin end} does not need the environment. Of course, if the sequence occurs in front of something that does need the environment, the \( B \) and \( T \) registers will be saved and restored further out. The most important instance where this lazy approach is useful is when nothing follows a procedure call. This will be discussed further later.

The preservation of the environment is done by a function \text{preservenv} defined as follows: \text{preservenv(fcode1, fcode2)} is \text{SAV B SAV T fcode1 RST T RST B fcode2} if \text{fcode1} modifies the environment and \text{fcode2} needs the environment to be unchanged; otherwise, \text{preservenv(fcode1, fcode2)} is just \text{fcode1 fcode2}. Given this function, we can present the declaration of \text{&}. Notice that the fixed code is generated backwards.

\[
\text{infix } \&; \\
\text{fun (code1 } \& \text{ code2) cont =} \\
\text{let} \\
\text{val (fixedcode2, cont2) = code2 cont} \\
\text{val (fixedcode1, cont1) = code1 cont2} \\
\text{in} \\
\text{(preservenv(fixedcode1, fixedcode2), cont1)} \\
\text{end;}
\]
The other code operator, &&, appears below. Recall that \textit{code1} is the code of an expression which is the condition of an \textit{if} or \textit{while} statement and \textit{code2} is the code of a statement which is the body of a \textit{if} or \textit{while} statement. Assuming that the continuation \textit{cont} is NEXT and that the body of the \textit{if} or \textit{while} is non-trivial (i.e. that \textit{cont2} is NEXT), the fixed code we wish to generate is as follows

\begin{verbatim}
fcode1
JPC ●
fcode2
\end{verbatim}

With other combinations of continuations, the fixed code varies slightly from the above. The details appear in the definition of \textit{mkJPCcode}.

\begin{verbatim}
fun mkJPCcode(jump,no_jump) : fixedcode * fixedcode * fixedcode=
let
  val (jmpcode,returncode) =
    case jump of
      NEXT => (theJPCcode, emptyfixedcode)
    | GOTO lab => (backpatch(lab,theJPCcode), emptyfixedcode)
    | RETURN => let val lab = newlabel "ret"
            in (backpatch(lab,theJPCcode), prefix(lab,theRETcode))
            end
  val nojumpcode =
    case no_jump of
      NEXT => emptyfixedcode
    | GOTO lab => mkJMPlab lab
    | RETURN => theRETcode
  in
    (jmpcode, nojumpcode, returncode)
  end;

infix &&

fun (code1 && code2) cont =
let
  val (fcode2, cont2) = code2 cont
  val (fcode1, cont1) = code1 NEXT
  val (jmpcode,nojumpcode,perhaps_return)= mkJPCcode(cont,cont2)
  in
    ((fcode1 % jmpcode) %% (nojumpcode % fcode2 % perhaps_return), cont1)
  end;
\end{verbatim}
To take an example, consider the program in Figure 8. The result of compiling it appears in Figure 9. Notice that there are just two labels in the compiled program and multiple jumps to both of them: if the condition in the inner if is false, one jumps straight to the condition of the while statement and when the while condition becomes false, one jumps straight to the final assignment.

**Exercise 6.4** What is the continuation to which the code for the outer if is ultimately applied?

**Exercise 6.5** When the code for the outer if is composed with the code for the last assignment, how many holes are backpatched?

**Exercise 6.6** What is the continuation to which the code for the inner while is applied?

**Exercise 6.7** How is it achieved that the third JPC jumps straight to whilentry14?

**Exercise 6.8** What code is produced for the following program?

```pascal
var x;
while x = 0 do
  if x = 0 then begin end.
```

**Exercise 6.9** Mr B. L. Under, being keen to simplify the && operation, changed the last four lines to

```pascal
in
  (fcode1 % jmpcode % nojumpcode % fcode2 % perhaps_return, cont1)
end;
```

Why was that a blunder?

**Project 6.10** Extend the PL/0 code generator to have an if then else statement. Do it in such a way that statements of the form
6.6 Preserving the environment

Figure 9: The result of compiling the program in Figure 8
48

6 CODE GENERATION USING CONTINUATIONS

\[
\begin{align*}
\text{if } \text{cond1} \\
\text{then stmt1} \\
\text{else if } \text{cond2} \\
\text{then stmt2} \\
\vdots \\
\text{else stmtn}
\end{align*}
\]

are compiled without generating superfluous jumps or labels. \text{(Hint: You might have to introduce a fourth kind of continuation.)}

6.7 Lazy register preservation

It is possible to compute for each piece \textit{fcode} of fixed code

1. The set of registers \textit{needed} by \textit{fcode}

2. The set of registers \textit{modified} by \textit{fcode}

To give a complete definition of what these sets are, it suffices to define them for the basic instructions and for the operations by which pieces of fixed code are combined, such as \%, and \%%.

The needs and modifies registers of the individual instructions is given by the following table (\textit{N} stands for \textit{needs, M} for \textit{modifies} and \textit{c} ranges over pieces of fixed code):

<table>
<thead>
<tr>
<th>(c)</th>
<th>(N(c))</th>
<th>(M(c))</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIT((n))</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>OPR((0))</td>
<td>(\emptyset)</td>
<td>{CONT}</td>
</tr>
<tr>
<td>OPR((i), i &gt; 0)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>LOD((i, j))</td>
<td>{(T, B)}</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>STO((i, j))</td>
<td>{(T, B)}</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>INT((i))</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>LAB((lab))</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>JMP((lab))</td>
<td>{(T, B)}</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>JPC((lab))</td>
<td>{(T, B)}</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>SAV((\text{reg}))</td>
<td>{\text{reg}}</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>RST((\text{reg}))</td>
<td>(\emptyset)</td>
<td>{\text{reg}}</td>
</tr>
<tr>
<td>PLB((lab))</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>CAL((n, lab))</td>
<td>{(T, B)}</td>
<td>all registers</td>
</tr>
<tr>
<td>(c_1%c_2)</td>
<td>(N(c_1) \cup (N(c_2) \setminus M(c_1)))</td>
<td>(M(c_1) \cup M(c_2))</td>
</tr>
<tr>
<td>(c_1%%c_2)</td>
<td>(N(c_1) \cup N(c_2))</td>
<td>(M(c_1) \cup M(c_2))</td>
</tr>
<tr>
<td>preserve((\text{reg}, c_1, c_2))</td>
<td>(N(c_1%c_2))</td>
<td>((M(c_1) \setminus {\text{reg}}) \cup M(c_2))</td>
</tr>
<tr>
<td>preservenv((c_1, c_2))</td>
<td>(N(c_1%%c_2))</td>
<td>((M(c_1) \setminus {T, B}) \cup M(c_2))</td>
</tr>
</tbody>
</table>

Some of these entries deserve explanation. First note that \textit{CAL} needs the values of the \textit{B} and \textit{T} to perform the call; in the worst case, the called function may modify all the
6.8 Tail recursion

In the code produced by the direct code generator, a called procedure always returns to its caller. This is no longer the case. We adopt the convention that whenever a procedure returns, it finds its return address on the top of the stack. Some procedures never return, however, because the last thing they do is to call another procedure. Such a procedure is said to be tail recursive.

Consider the compilation of a PL/0 statement CALL Q, occurring in the body of procedure P, say. The compilation is relative to a continuation. If the continuation is not RETURN, instructions for storing the return address on the stack are generated.

However, if the continuation is RETURN then there is no need to return to the calling procedure, P. Since the next thing P would do, were we to return to it after the call, would be to return, we can assume that a return address r is already on the top of the stack. We therefore just generate a CAL instruction without saving any new return address on the stack, so that Q, if it needs to return, will return to r. The following declarations give the details.

```ml
fun mkreturningCAL(i,label) =
  (NONE, INCOMPLETE(fn lab => [PUSHLAB lab, CAL(i,label)]), [T,B], allregs)

fun mktailCAL(i,label) =
  (NONE, COMPLETE [CAL(i, label)], [T,B], allregs)

fun mkfcode inst cont : fixedcode =
  case inst of
    ...
```
| CAL(i,label) =>
  (case cont of
    NEXT =>
      (* Push a return address and jump *)
      mkreturningCAL(i,label)
    | GOTO lab =>
      (* Here lab is a local jump. *)
      (* Push a return address and jump to procedure. *)
      (* The code at the target of the local jump might *)
      (* need the environment, so we preserve it. *)
      preservenv(mkreturningCAL(i,label),mkJMPlab(lab))
    | RETURN =>
      (* Tail recursion: no need to push a new return address; *)
      (* simply jump to procedure *)
      mktailCAL(i,label))
  | _ => Crash.impossible ("mkfcode: illegal instruction" ^
    showinst inst)

Below are two examples of tail recursion:

```plaintext
var x;
procedure P;
  x := x + 1;
procedure Q;
  begin call P; call P end;
begin (* main *)
  x := 0; call Q
end.
```

Code for Q
```
  entrypoint2
    INT 1
    SAV B
    SAV T
    PLB seq3
    CAL(1,entrypoint1)
  seq3
    RST T
    RST B
    CAL(1,entrypoint1)
```
6.8 Tail recursion

Code for P

entrypoint1

```
INT 1
LOD(1,1)
LIT 1
OPR 2
STO(1,1)
RST CONT
OPR 0
```

main program

```
INT 2
LIT 0
STO(0,1)
CAL(0,entrypoint2)
```

```
var a, b;
procedure Q;
var c, d;
procedure R;
begin
  c := b + d;
  if c < a then
    begin b := c; call Q end
end
begin (* Q *)
d := 1; call R
end;
begin (*main*)
a := 2; b := 0; call Q
end.
```

Code for R

entrypoint5

```
INT 1
LOD(2,2)
LOD(1,2)
OPR 2
STO(1,1)
LOD(1,1)
LOD(2,1)
```
OPR 10
JPC ret6
LOD(1,1)
STO(2,2)
CAL(2,entrypoint4)

ret6
RST CONT
OPR 0

Code for Q
entrypoint4
INT 3
LIT 1
STO(0,2)
CAL(0,entrypoint5)

main program
INT 3
LIT 2
STO(0,1)
LIT 0
STO(0,2)
CAL(0,entrypoint4)
7 Register Allocation

This section describes a method of using registers for holding the values of variables and expressions. Registers are allocated backwards, using an analogy between Hoare triples and so-called requirement triples.

7.1 Introduction

The SEC Machine uses the stack for all expression evaluation. Also, it has one access to the environment for each free variable occurrence. In register machines it is desirable (in fact sometimes necessary) to perform arithmetic operations in registers; also, since accessing the environment can be considerably more expensive than accessing registers, it is desirable to keep the values of variables in registers, when possible.

If one has a very large supply of registers, one can simply allocate one register for each variable and allocate extra registers for holding the values of expressions. An upper bound on the number of temporary values that have to exist simultaneously can be determined on the basis of the depth of the expressions. However, the number of registers cannot in general be assumed to be large enough to make this strategy practical.

Good register allocation is difficult. A number of different approaches exist; generally, there is a tension between how efficient and small one wants the compiled code to be and how much effort one is willing to put into the register allocation.

The method we shall present in this section is designed to fit with the code generation style presented in the previous section. In particular, register allocation is done backwards. When the value of a register is to be preserved across a computation, it will be kept on the stack, if necessary. The machinery for deciding when it is necessary to preserve a register on the stack was already developed in the continuation style code generator and involves keeping with each piece of fixed code a set of modified and needed registers.

We shall first describe the idea on which the register allocation algorithm is based and then discuss register allocation in the case of selected PL/0 statement and expression forms.

7.2 Requirement maps

In the continuation style code generator, the continuation expresses where evaluation is going to continue, once the phrase in question has been evaluated completely. Similarly, we shall now use a requirement map to express where the rest of the computation expects variables to be found at run-time. Formally, a requirement map is a map from dynamic program variables to places. By a dynamic variable, we understand a pair (distance, offset). A place can be one of the following: in the environment; in a particular register; both in the environment and in a particular register; anywhere, meaning that there is no requirement on where the variable is to be placed. A requirement map is well-formed, if it does not associate two different dynamic variables with the same register. We are only interested in well-formed requirement maps.
We use \( \text{req} \) to range over requirement maps. When writing requirement maps, we shall often identify a dynamic variable with the variable for which it stands. To take an example of the notation that will be used,

\[
\text{req} = \{ x : \text{R0}, y : \text{ENV}, z : (\text{ENV}, \text{R1}), v : ? \}
\]

is a well-formed requirement map, which states that \( x \) is required to be in register \( \text{R0} \), that \( y \) must be in the environment, that \( z \) must be in the environment and in register \( \text{R1} \), and that there is no requirement on where \( v \) is placed. Whenever we write a requirement map \( \text{req} \) in the form \( \{ \text{dynvar}_1 : \text{place}_1, \ldots, \text{dynvar}_n : \text{place}_n \} \) the convention is that \( \text{req}(\text{dynvar}) = \text{ENV} \), for all \( \text{dynvar} \notin \{ \text{dynvar}_1, \ldots, \text{dynvar}_n \} \).

The place \( ? \) is only used for variables whose current value is definitely not necessary to the rest of the computation; such variables are called \textit{dead}, as opposed to \textit{live} variables, which are variables whose current values may be required by the subsequent computation.

A \textit{requirement triple} is a triple \( (\text{req}', \text{fcode}, \text{req}) \), where \( \text{fcode} \) is a piece of fixed code. We shall write it as follows:

\[
\{ \text{req}' \} \text{fcode} \{ \text{req} \}
\]

read as follows: if \( \text{fcode} \) is evaluated in a state which satisfies \( \text{req}' \) and the evaluation terminates then the resulting state satisfies \( \text{req} \). Here \( \text{req}' \) is called the \textit{pre-requirement (map)} and \( \text{req} \) the \textit{post-requirement (map)} of the triple. A requirement triple for the code which results from compiling the statement \( y := 5 \); \( x := (y+z) \times 3 \) is shown below.

The machine instructions shown in this example are all of the form \( \text{OP} \langle \text{dest} \rangle \langle \text{source} \rangle \), where \( \text{dest} \) is a register whose contents may be changed by the operation. Also, we drop the outer \( \{ \} \) around requirement maps in requirement triples.

\[
\{ x : ?, y : ?, z : \text{R2} \}
\]

\begin{align*}
\text{MOV} & \text{ R0, 5} \\
\text{STO} & \text{ y, R0} \\
\text{ADD} & \text{ R0, R2} \\
\text{MOV} & \text{ R1, R2} \\
\text{MOV} & \text{ R2, 3} \\
\text{MUL} & \text{ R0, R2} \\
\{ x : \text{R0}, y : \text{ENV}, z : \text{R1} \}
\end{align*}

Note that there are no pre-requirements on \( x \) or \( y \) as the statement kills \( x \) and \( y \). Also note that \( y \) is stored into the environment because the post-requirement map requires it to be there, whereas \( x \) is not stored in the environment.

The reader will have noticed a strong resemblance between requirement triples and the well-known triples which Hoare has devised for proving programs correct. This resemblance is deliberate. Requirement maps are created backwards and they can be composed by structural induction on the PL/0 phrases. In the above example we first compile the
last assignment \( x := (y+z) \times 3 \) yielding the requirement triple

\[
\{ x :?, y : (\text{ENV}, R0), z : R2 \}
\]

\[
\begin{align*}
\text{ADD} & \ R0, \ R2 \\
\text{MOV} & \ R1, \ R2 \\
\text{MOV} & \ R2, \ 3 \\
\text{MUL} & \ R0, \ R2 \\
\{ x : R0, y : \text{ENV}, z : R1 \}
\end{align*}
\]

Then the first requirement of this triple is used to generate a triple for the statement \( y := 5 \):

\[
\{ x :?, y :?, z : R2 \}
\]

\[
\begin{align*}
\text{MOV} & \ R0, \ 5 \\
\text{STO} & \ y, \ R0 \\
\{ x :?, y : (\text{ENV}, R0), z : R2 \}
\end{align*}
\]

and finally the triples are composed using the rule:

\[
\{ \text{req}_1 \} \text{ fcode}_1 \{ \text{req}_2 \} \quad \{ \text{req}_2 \} \text{ fcode}_2 \{ \text{req}_3 \} \\
\{ \text{req}_1 \} \text{ fcode}_1 \% \text{ fcode}_2 \{ \text{req}_3 \}
\]

It is possible to combine code generation and register allocation in a single backwards phase. However, greater clarity is obtained by separating the two tasks, so that register allocation is done in a separate phase prior to code generation. We refer to this phase as register distribution; it computes requirement maps and produces phrases that are annotated with register information. Code generation then produces the actual code from these register distributed phrases, without having to know anything about requirement maps. We now describe these two phases in turn, the main phase of interest of course being register distribution.

### 7.3 Register distribution

Register distribution transforms annotated syntax trees to what we shall call register distributed syntax trees. The signature of these appears below. The phrase classes \textit{program}, \textit{vardec}, \textit{procdec} and \textit{block} are not changed. The important change is that the phrase classes \textit{statement} and \textit{expression} have been collapsed into one, also called \textit{statement}. Expressions produce values, but in the register distributed form, all values are manipulated explicitly by moves between registers and between the environment and registers. In the signature below, note that \texttt{ASSIGN} assigns to a register, not a dynamic variable. The operator \texttt{SEQ} combines two statements that are to be evaluated in sequence in the same environment; \texttt{GLUE} simply concatenates two statements. As for the \texttt{IF} and \texttt{WHILE}, notice that those expressions that really are conditions have been singled out in the type \texttt{cond}; the reason for this is that on many machines boolean operations are slightly different from arithmetic operations. (Often the result of a boolean operation is not put in a register.) There are new statements for explicit moves between registers (\texttt{MOV}), from the environment to a
56

7 REGISTER ALLOCATION

register (LOAD) and back (STORE). For numbers (NUM), application of monadic numeric operators (APPMONOP) and application of binary numeric operators (APPBINOP) the registers where the values are to go are part of the statements.

signature ALLOC_SYNTAX =
  sig
    type ident and number and distance and offset and label and reg
    sharing type number = distance = offset = int
    and type ident = string

    (* Any occurrence of the type ident in the following is
     to allow pretty printing only *)

datatype program = PROGRAM of block
  and vardec = VAR of int (* number of variables declared *)
  and procdec = PROCDEC of (ident * label * block)list
  and block = BLOCK of vardec * procdec * statement
  and statement = ASSIGN of ident * reg *
    statement * statement * statement
    | SEQ of statement * statement
    | GLUE of statement * statement
    | IF of cond * statement
    | WHILE of cond * statement
    | CALL of ident * distance * label
    | STORE of (distance * offset) * reg
    | LOAD of reg * (distance * offset)
    | MOVE of reg * reg
    | EMPTYSTAT
    | NUM of reg * number
    | APPMONOP of monop * reg * statement
    | APPBINOP of binop * reg * reg * statement * statement
    and cond =
    | APPMONCOND of moncond * reg * statement * statement
    | APPBINCOND of bincond * reg * reg *
    | statement * statement * statement

    and moncond = ODD
    and bincond = EQUALS | NOTEQUAL | LTH | GTH | LEQ | GEQ
    and monop = UPLUS | UMINUS
    and binop = PLUS | MINUS | TIMES | INTDIV;
  end;
The signature of the requirement map structure is presented below. ANYWHERE is what was written in the earlier examples, where we also abbreviated INENV to ENV, INREG(r) to r and INENVandREG(r) to (ENV, r).

The initial requirement map, req₀ is the requirement map that maps every dynamic variable to INENV.

An application \text{add}(\langle d, \text{place} \rangle, \text{req}) adds \text{place} to the requirement req(d), where \text{d} is a dynamic variable (distance, offset).

An application \text{update}(\langle d, \text{place} \rangle, \text{req}) replaces the requirements on the dynamic variable \text{d} already in \text{req} by the requirement \text{place}.

An application \text{connect}(\text{req}_1, \text{req}_2) produces a statement. If this statement is evaluated in a state satisfying \text{req}_1, the resulting state will satisfy \text{req}_2 (although the resulting state might in fact satisfy more that required by \text{req}_2). The operation \text{connect}(\text{req}_1, \text{req}_2) only makes sense if for no dynamic variable \text{d}, \text{req}_1(d) = \text{ANYWHERE} and \text{req}_2(d) \neq \text{ANYWHERE}. In other words, there is no sensible way of connecting from a state where \text{d} is dead to where it is alive. (The converse situation where \text{req}_2(d) = \text{ANYWHERE} and \text{req}_1(d) \neq \text{ANYWHERE} is perfectly normal and these two requirements are connected by the empty statement.)

An application \text{select_target}(\text{req}, \text{dynvar}) selects a register which is to hold the value of \text{dynvar}.

An application \text{release}(\text{reg}, \text{req}) produces a requirement map by replacing any requirement on \text{reg} in \text{req} by INENV.

An application \text{assign}(\text{req}) returns a free register if one exists and otherwise an arbitrary used register. A register is free (with respect to \text{req}) if it does not occur in any requirement in the range of \text{req}.

An application \text{assignreg}(\text{req}, \text{reg}) is equivalent to \text{assign}(\text{req}) except that the selected register is always chosen to be different from \text{reg}.

```signature
signature REQMAP =
  sig
    type req (* requirement maps *)
      and reg and distance and offset and stmt
    sharing type distance = offset = int
    datatype place = INENV | INREG of reg
      | INENVandREG of reg | ANYWHERE
  val req0: req
  val add: ((distance * offset) * place) * req -> req
  val update:((distance * offset) * place) * req -> req
  val connect: req * req -> stmt
  val select_target: req * (distance * offset) -> reg
  val release: reg * req -> req
  val assign: req -> reg
  val assignreg: req * reg -> reg
  val assignregpair: req -> reg * reg
```
An application `assignpair(req)` returns a pair of different registers, free if possible.

The signature of the register distributor structure is presented below. The types of `Astatement` and `Aexp` deserve comment.

An application `Astatement statement req` produces a register distributed statement `stmt'` and a requirement map `req'` such that if `stmt'` is evaluated in a state which satisfies `req'` then the resulting state satisfies `req`. Notice that the requirement map is inferred backwards.

We adopt the convention that the value of an expression always is put in a register. The register distribution function for expressions, `Aexp` takes a parameter `target` which is the register in which the value of the expression is to be placed. An application `Aexp exp target req` produces a triple `(stmt', req', stmt'')` such that if `stmt'` is evaluated in a state which satisfies `req'` then the value of `exp` is placed in the register `target`; at this intermediate stage, the state does not necessarily satisfy `req`, but if `stmt''` is evaluated starting from the intermediate state, one will achieve a state which satisfies `req`. The reason register allocation is slightly more complicated for expressions than for statements is this. Expressions produce values and we always put these values in registers. If `req` states that all registers are used for other purposes by the subsequent program, it is impossible to produce code for the expression which puts its result in a register without upsetting the final requirement `req`. The solution to this problem is to split the code into a fragment `stmt'` which produces the value in the designated target and a fragment `stmt''` which does the “repair work” required to obtain a state which satisfies the post-requirement. By keeping these fragments separate in the register allocated statements, we can leave their combination to the code generation phase, which has all the machinery necessary to ensure, for example, that the value of `target` is preserved till it is used, by use of the stack, if necessary.

We now proceed with a description of how registers are distributed over statements and expressions. This takes place in the functor `RegAlloc`, from which the code fragments shown below stem. Throughout the following subections, `req` is the post-requirement map given as argument to `Astatement` or `Aexp`.

```
signature REGALLOC =
  sig
    structure A: ANN_SYNTAX
    structure B: ALLOC_SYNTAX
    type req (* requirement maps *)
      and reg (* registers *)
    val Aprogram: A.program -> B.program
    val Avardec: A.vardec -> B.vardec
    val Aprocdec: A.procdec -> B.procdec
```
7.3 Register distribution

val Ablock: A.block → B.block
val Astatement: A.statement → req → B.statement * req
val Aexp: A.expression → reg → req → B.statement * req * B.statement
val Acond: A.expression → B.cond * req
val Amonop: A.monop → B.monop
val Abinop: A.binop → B.binop
end;

7.3.1 Assignment

The case for assignment appears below. We first select a register target which is to hold the value of the variable. We release this register from req, yielding req₂. We then distribute registers over the expression relative to req₂ updated so as to record that the variable dv is dead at the end of the expression (because we are going to assign to dv immediately after). The resulting pre-requirement map req₁ is propagated to be the pre-requirement of the entire statement. Furthermore, we know that stmt₁ puts the value of the expression in target. We therefore strengthen req₂ with the information that dv is now in target (only), and use connect to produce a statement that will take us from the strengthened requirement map to req, the post-requirement. All the statement fragments and the target register are assembled to form the resulting statement.

A.ASSIGN(id,distance,offset,exp) =>
let
  val dv = (distance,offset+n0)
  val target = select_target(req,dv)
  val req2 = release(target,req)
  val (stmt1,req1,stmt1') =
    Aexp exp target (R.update((dv,ANYWHERE), req2))
  val req' = R.update((dv, INREG target),req2)
  val stmt2 = connect(req',req)
in
  (B.ASSIGN(id, target, stmt1, stmt1', stmt2), req1)
end
7.3.2 The sequential statement

Here we simply propagate the requirement maps backwards:

```plaintext
| A.SEQ(stmt1,stmt2)->
  let
    val (stmt2',req2) = A.statement stmt2 req
    val (stmt1',req1) = A.statement stmt1 req2
  in
    (B.SEQ(stmt1', stmt2'), req1)
end
```

7.3.3 The if statement

When we try to express the general idea of propagating requirement maps backwards through an if statement

```plaintext
if exp then stmt
```

we are faced with an interesting difficulty. We want req to hold after the entire if statement. We could sensibly propagate req backwards through stmt yielding some requirement req₁, since the termination of stmt also marks the termination of the if statement. However, we cannot in general tell whether evaluation after exp will continue with stmt or not. Thus the relevant requirement map after exp might be req or req₁.

It is therefore natural to look for a requirement map req' from which one can connect to both req and req₁. The register allocation for exp now uses req' as post-requirement. We then need to generate (1) a statement to connect from req' to req₁ at the beginning of the translation of stmt and (2) a statement to connect from req' to req to be evaluated in the case that exp evaluates to false.

If one has a partial ordering on requirement maps with least upper bounds, one possible choice of req' is the least upper bound of req and req₁, meaning the least demanding requirement map that demands no less than req and req₁. Then neither of the two connecting statements mentioned in the previous paragraph are required; however, the price for this gain is that the requirements of stmt now have to be met further out, even in cases where stmt is not evaluated. It is not necessarily the case, therefore, that this choice of req' is the best.

Alternatively, one can simply let req' be the initial requirement map req₀. This makes sense since one can connect from req₀ to any requirement map. The price that is paid for this simplicity is superfluous load and store operations.

With this simple choice of requirement map, the register distribution can be implemented as follows:
7.3 Register distribution

| A.IF(exp,stmt) =>
  let
  val stmt_end = connect(req0, req)
  val (stmt2, req2) = Astatement stmt req0
  val stmt_then = connect(req0,req2)
  val (stmt1, req1) = Acond(exp)
  in
  (B.SEQ(B.IF(stmt1, B.GLUE(stmt_then, stmt2)),
     stmt_end),
   req1)
  end

Notice that Acond does not take a requirement map as a parameter. It always allocates registers relative to the post-requirement map req0.

Exercise 7.1 At first, one might think that there is a simpler treatment of conditionals, namely

| A.IF(exp,stmt) =>
  let
  val (stmt2, req2) = Astatement stmt req
  val stmt_then = connect(req,req2)
  val (stmt1, req1) = Acond(exp)req
  in
  (B.IF(stmt1, B.GLUE(stmt_then, stmt2))
   req1)
  end

where we have assumed that Acond has been changed to take a requirement map parameter. Unfortunately, that does not work. Why?

7.3.4 The while statement

Consider the case of a while statement

while exp do stmt

We know that the last thing that is evaluated is exp, no matter whether the loop is executed zero or more times. Therefore, we start by propagating req0 back through exp, obtaining a pre-requirement req1. Also, whenever stmt terminates, control will continue at the beginning of exp, so we can propagate req1 back through stmt. Finally we have
connection to $req_0$ at the beginning of the body of the loop and after the entire statement:

| A.WHILE(exp,stmt) =>
| let
| val stmt_end = connect(req0, req)
| val (stmt1, req1) = Acond(exp)
| val (stmt2, req2) = Astatement stmt req1
| val stmt_do = connect(req0,req2)
| in
| (B.SEQ(B.WHILE(stmt1,B.GLUE(stmt_do, stmt2)),
| stmt_end),
| req1)
| end

Exercise 7.2  One might try to be smart and “simplify” the above to

| A.WHILE(exp,stmt) =>
| let
| val (stmt1, req1) = Acond(exp)req
| val (stmt2, req2) = Astatement stmt req1
| val stmt_do = connect(req,req2)
| in
| (B.WHILE(stmt1,B.GLUE(stmt_do, stmt2))
| req1)
| end

Unfortunately this does not work. Why?

Exercise 7.3  Consider the PL/0 program in Figure 10. Register distribution over the while loop is done relative to the initial requirement map $req_0$. You may assume that there are four data registers $R0$ to $R3$ and that the register with the lowest number is chosen, when there is a choice between equally good registers. What is the post-requirement map which is used for register distribution over the body of the while loop?

Exercise 7.4  Compare the PL/0 program in Figure 10 with Figure 11, the corresponding compiled code. Explain the presence of the STO $(0,2), R0$ instruction two lines above the label ret10.

Exercise 7.5  Explain the presence of the two load instructions just after the label seq9.
7.3 Register distribution

var x,y,z,q,r;
procedure multiply;
  var a,b;
  begin a:= x; b:= y; z:= 0;
    while b > 0 do
      begin
        if odd b then z := z + a;
        a := 2 * a;
        b:= b / 2
      end
    end;
begin (* main *)
  x:= m; y:= n; call multiply
end.

Figure 10: Multiplication procedure in PL/0

7.3.5 The call statement

In our simple code generator all procedure bodies start executing assuming that all variables are in the environment only. (In more advance code generators, one might associate with each procedure a requirement map which the caller would have to satisfy and a requirement map that will hold after the execution, but we have not done so.) Thus, in the case of the call statement, we simply connect to req\(_0\) and return req\(_0\):

<table>
<thead>
<tr>
<th>A.CALL(id,distance,label) =&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>let val stmt2 = connect(req0, req)</td>
</tr>
<tr>
<td>val stmt1 = B.CALL(id,distance, label)</td>
</tr>
<tr>
<td>in (B.SEQ(stmt1, stmt2), req0)</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

7.3.6 Program variable expression

We now consider the problem distributing registers over an expression relative to a target target and a post-requirement map req.

In the case of the expression being just a (dynamic) variable \(d\), we produce \((stmt, req_1, \ldots)\),
Code for multiply

entrypoint7

    INT 3
    LOD R1,(1,1)
    LOD R0,(1,2)
    STO (0,1),R1
    STO (0,2),R0
    MOV R1,0
    STO (1,3),R1

whilentry8

    MOV R1,0
    CMP R0,R1
    JLE ret10
    LOD R0,(0,2)
    TOD R0
    JEV seq9
    LOD R1,(0,1)
    LOD R0,(1,3)
    ADD R0,R1
    STO (1,3),R0

seq9

    LOD R2,(0,1)
    LOD R0,(0,2)
    MOV R1,2
    MUL R1,R2
    STO (0,1),R1
    MOV R1,2
    DIV R0,R1
    STO (0,2),R0
    JMP whilentry8

ret10

    RST CONT
    OPR 0

main program

    INT 6
    MOV R0,7
    STO (0,1),R0
    MOV R0,85
    STO (0,2),R0
    CAL(0,entrypoint7)

Figure 11: Compiled multiplication procedure
stmt$1$), where the first component is the empty statement, $req_1$ requires $d$ to be in $target$ and $stmt_1$ is the required reparation statement.

\begin{verbatim}
A.IDENT(id,distance,offset) =>
  let
    val dv = (distance, offset+n0)
    val req1 = R.add((dv, INREG target), release(target,req))
    val stmt1 = connect(req1,req)
  in
    (B.EMPTYSTAT, req1, stmt1)
end
\end{verbatim}

### 7.3.7 Constant in expression

In the case of a numeric constant, we release the target register and keep it with the constant together with any reparation statement necessary.

\begin{verbatim}
| A.NUM(n) =>
  let
    val req2 = release(target, req)
    val stmt2 = connect(req2,req)
    val stmt1 = B.NUM(target, n)
  in
    (stmt1, req2, stmt2)
end
\end{verbatim}

### 7.3.8 Application of a binary numeric operator

Consider an expression

\[ exp_1 \, \text{binop} \, exp_2 \]

We assume that every binary numeric operation operates on two registers (the source and the destination), returning the result in destination. We therefore pick a (preferably free) target $target_2$ for $exp_2$ and propagate $target$ to the left-hand operand expression $exp_1$.

\begin{verbatim}
| A.APPBINOP(exp1,binop,exp2) =>
  let
    val target2 = assignreg(req,target)
end
\end{verbatim}
val (stmt2, req2, stmt2') = Aexp(exp2)(target2)(req)
val (stmt1, req1, stmt1') = Aexp(exp1)(target)(req2)
in
  (B.APPBINOP(Abinop binop, target, target2, stmt1,
    B.GLUE(stmt1', stmt2)),
    req1,
    stmt2')
end

To understand this, note that stmt_1 will bring the value of exp_1 into target, although it might not achieve req_2. Therefore, to evaluate exp_2 we must first evaluate the reparation code for exp_1 so that we achieve req_2; then we evaluate stmt_2. That is why we glue stmt_1' and stmt_2 together. The preservation of target_1 across the evaluation of stmt_1' and stmt_2 is taken care of by the code generator.

7.3.9 Application of a binary boolean operator

Consider a condition of the form

\[ \text{exp}_1 \text{ binrel } \text{exp}_2 \]

where binrel is a relational operator. On some machines, built-in boolean operations, such as test for equality of integers, do not produce results in data registers but affect suitable status flags. Thus the target of a condition is not necessarily a register. Therefore, we have to choose targets for both exp_1 and exp_2. In other respects, the distribution of registers over conditions resembles the one for expressions:

\[
\text{and Acond (A.APPBINOP(exp1,binrel,exp2)) : B.cond * req =}
\]
\[
\text{let}
\]
\[
\text{val (target1,target2) = assignregpair req0}
\]
\[
\text{val (stmt2,req2,stmt2')} = \text{Aexp exp2 target2 req0}
\]
\[
\text{val (stmt1,req1,stmt1')} = \text{Aexp exp1 target1 req2}
\]
\[
in
\]
\[
(B.APPBINCOND(Abincond binrel, target1, target2, stmt1,
  B.GLUE(stmt1', stmt2), stmt2''),
  req1)
\]
\end

Project 7.6 Clearly, the handling of if and while statements leads to superfluous load and store instructions. Devise and implement an improvement which rests on a
better choice of the unifying requirement map \( req' \) than \( req_0 \). (Hint: Consider the solutions to the exercises in Sections 7.3.3 and 7.3.4. Also take a good look at the definition of \texttt{connect} in the \texttt{ReqMap} functor and extend its range of application.)

**Project 7.7** Register distribution over a block is done relative to \( req_0 \). Thus, variables that are local to that block will ultimately be flushed back into the environment, even though they are dead when the block terminates. Implement an improvement of the register distribution algorithm which avoids these unnecessary flushes. (This can actually give significantly improved code since one no longer propagates any environment requirement about local variables backwards into while loops.)

### 7.4 Code generation

The basic principles of the code generator are as in the continuation style code generator. Of course code generation now happens from register distributed syntax trees. The full details can be found in the functor \texttt{CodeGen} (in the file \texttt{PL0/RegCompiler/CodeGen.sml}). Below we include just the part concerning statements.

The type of \texttt{Cstatement} is simply

\[
\text{val Cstatement: statement \to code}
\]

In the case of assignment, notice how we preserve the value of the target register over the reparation code that stems from the expression. The function \texttt{preservereg} is specified in the signature \texttt{CODE} with the following type:

\[
\text{val preservereg: reg \times code \times code \to code}
\]

The result of an application \texttt{preservereg(reg, code\textsubscript{1}, code\textsubscript{2})} is \texttt{code\textsubscript{1}&code\textsubscript{2}} if \texttt{code\textsubscript{1}} does not modify \texttt{reg} or \texttt{code\textsubscript{2}} does not need \texttt{reg}; otherwise the result is

\[
\begin{align*}
\text{SAV} \ & \texttt{reg} \\
\text{code\textsubscript{1}} \\
\text{RST} \ & \texttt{reg} \\
\text{code\textsubscript{2}}
\end{align*}
\]

In the case of \texttt{if} and \texttt{while} statements, notice that \texttt{Acond} produces both some code and some fixed code (\texttt{jumpcode}), which is the jump instruction that corresponds to the boolean operator in the condition.

Finally note the translation of an application of a binary operator to two expressions (\texttt{APPBINOP}). Here we preserve the value of \texttt{target\textsubscript{1}}, i.e. the value of the first subexpression, across any computation (by \texttt{code\textsubscript{2}}) which takes place before the value of the first expression is needed, which happens precisely at \texttt{code\textsubscript{3}}. Notice that there is a big advantage to have kept the structure of the source program so that the lifetime of temporary values is apparent.
and Cstatement(statement) =
    case statement of
    ASSIGN(id,target,stmt1,stmt2,stmt3) =>
        let
            val code1 = Cstatement stmt1
            val code2 = Cstatement stmt2
            val code3 = Cstatement stmt3
        in
            code1 & preservereg(target, code2, code3)
        end
    | SEQ(stmt1,stmt2) => Cstatement(stmt1) & Cstatement(stmt2)
    | GLUE(stmt1,stmt2) => Cstatement(stmt1) glue Cstatement(stmt2)
    | IF(cond1,stmt2) =>
        let
            val code2 = Cstatement(stmt2)
            val (code1,jumpcode) = Ccond cond1
        in
            (code1 && code2)jumpcode
        end
    | WHILE(cond1,stmt2) =>
        let
            val lab = Lab.newlabel "whilentry"
            val (code1, jumpcode) = Ccond cond1
            val code2 = fixcont (Cstatement stmt2) (GOTO lab)
        in
            (mkcode(LAB lab) & code1 && code2)jumpcode
        end
    | CALL(id,distance,label) => mkcode(CAL(distance,label))
    | EMPTYSTAT => emptycode
    | STORE(dynvar,reg) => mkcode(STO(dynvar,reg))
    | LOAD(reg,dynvar) => mkcode(LOD(reg,dynvar))
    | MOVE(reg,reg') => mkcode(MOV(reg,REG reg'))
    | NUM(reg,n) => mkcode(MOV(reg,CON n))
    | APPMONOP(monop,reg,stmt) => Cstatement stmt & Cmonop monop reg
    | APPBINOP(binop,target1,target2,stmt1,stmt2) =>
        let
            val code1 = Cstatement stmt1
            val code2 = Cstatement stmt2
            val code3 = Cbinop binop (target1, target2)
        in
            code1 & preservereg(target1, code2, code3)
        end
A Solutions

2.1 block

2.2

\[
\text{APPBINOP(APPBINOP(NUM 3, TIMES, NUM 4), PLUS, APPBINOP(NUM 5, INTDIV, NUM 8))}
\]

2.3 One possible way of expressing it is:

\[
\begin{align*}
\text{let} & \quad \text{val Pblock=} \\
& \quad \quad \quad \text{BLOCK} \\
& \quad \quad \quad \quad \text{(NONE,NONE,NONE,ASSIGN("x",APPBINOP(IDENT "x", PLUS, NUM 1)))} \\
\text{val Qblock=} & \quad \text{BLOCK} \\
& \quad \quad \quad \text{(NONE,NONE,NONE,SEQ(CALL"P",CALL"P"))} \\
\text{in} & \quad \text{PROGRAM} \\
& \quad \text{(BLOCK} \\
& \quad \quad \quad \text{(NONE, (* constants *))} \\
& \quad \quad \quad \text{SOME} \\
& \quad \quad \quad \quad \text{(VAR ["x"])} \\
& \quad \quad \quad \text{SOME} \\
& \quad \quad \quad \quad \text{(PROCDEC} \\
& \quad \quad \quad \quad \quad ([("P", Pblock),("Q",Qblock)]) \\
& \quad \quad \quad \),} \\
& \quad \quad \quad \text{SEQ} \\
& \quad \quad \quad \quad \text{(ASSIGN("x",NUM 0),} \\
& \quad \quad \quad \quad \quad \text{CALL "Q"} \\
& \quad \quad \quad \quad \quad \text{)} \\
& \quad \quad \quad \quad \quad \text{)} \\
& \quad \text{end}
\end{align*}
\]

3.1 At (a), the stack is

\[
\begin{array}{c}
\langle\text{main}\rangle\langle\text{Q}\rangle\langle\text{P}\rangle \\
\hline \\
\text{x} \\
\text{B} \\
\text{T}
\end{array}
\]

After the first call of \( P \) was completed, the topmost activation record was removed; then another activation record was pushed for the second call, so that we have the following stack at (b):

\[
\langle \text{main} \rangle \langle \ Q \rangle \langle \ P \rangle
\]

\[
\begin{array}{l}
\text{x} \\
\text{B} \\
\text{T}
\end{array}
\]

3.2 Because with \( P_0 = \text{main} \), \( P_1 = Q \) and \( P_2 = R \), we have that the called procedure is local to \( P_0 \), \( P_1 \) is local to \( P_0 \), \( P_2 \) is local to \( P_1 \) and \( P_2 \) is the calling procedure.

3.3 The argument is 0

3.4 Change

\[
| \text{OPR 7} \Rightarrow \text{Crash.unimplemented "unary plus"} \ (\text{* unary plus *})
\]

to

\[
| \text{OPR 7} \Rightarrow \{P=P+1, T=T, B=B\} \ (\text{* unary plus *})
\]

and change

\[
| \text{OPR 4} \Rightarrow \text{Crash.unimplemented "multiplication"}
\]

\[
(\text{* multiplication *})
\]

to

\[
| \text{OPR 4} \Rightarrow (\text{update}(S,T-1,S \sub (T-1) \ast S \sub T);
\{P=P+1, T=T-1, B=B\}) \ (\text{* multiplication *})
\]

3.5 The problem that the return instruction OPR 0 reinstates the \( B \) register to the value of the static link, not the dynamic link. (The interpretation of \text{prog} goes wrong in the 37th step.) The fix is easy; change

\[
| \text{OPR 0} \Rightarrow \{P= S \sub (B+2), T=B-1, B= S \sub B\} \ (\text{* return *})
\]

to

\[
| \text{OPR 0} \Rightarrow \{P= S \sub (B+2), T=B-1, B= S \sub (B+1)\} \ (\text{* return *})
\]

4.1 It tries to look up the identifier in the static environment and is told that no information is recorded for that identifier. It then issues an error message. See the code for \text{Eexp}, the case where \text{exp} matches \text{IDENT(id)}. 
4.2 Identifiers that are declared as constants are replaced by their value during elaboration. Notice that Econstdec does not produce any constant declaration but a static environment.

4.3 The statement should be elaborated in $E++E'$, so that the declarations become visible inside the statement. See the code in Eblock.

4.4 In Eprocdec (in fact in Eprocdec'). No, the current level is not explicitly decreased anywhere: this would have been necessary were it represented by an updatable variable, but not necessary when it is used as a parameter to the elaborator functions. The same is true of the static environment: one does not need an operation for taking anything away from the environment.

4.5 One gets the somewhat misleading error message: Variable not declared: P. One possibility of improving the error handling slightly is to replace the lookup by:

```plaintext
case E.lookup(E,id) of
    E.FOUND(E.ISVAR{declev=d,offset=offs}) =>(d,offs)
| E.FOUND(_) => Crash.impossible ("Cannot assign value to\n    \ identifier which was declared as\n    \ constant or procedure: " ^ id)
| E.NOTFOUND => Crash.impossible("Variable not declared: " ^ id)
```

4.6 The static information that is required is just the declaration level and a unique label. Both of these can be determined before elaborating the body. The crucial point is that the body of the procedure is elaborated in the environment $E++E_1$, where $E_1$ is a small environment mapping the procedure identifier to its static information. See the code for Eprocdec'. To disallow recursion, one would simply replace $E++E_1$ by just $E$.

5.1

Code for P

```plaintext
entrypoint1
    INT 3
    LOD(1,3)
    LIT 1
    OPR 2
    STO(1,3)
    OPR 0
```

Code for Q

```plaintext
entrypoint2
    INT 3
    CAL(1,entrypoint1)
    CAL(1,entrypoint1)
    OPR 0
```
main program

```
INT 4
LIT 0
STO(0,3)
CAL(0,entrypoint2)
```

6.1 If `isempty(fcode)` evaluates to `true` then `fcode` is of the form `(_, COMPLETE [], _, _)` and therefore empty. Conversely, if `fcode` is empty it cannot be of the form `(_, INCOMPLETE f, _, _)`, for `f` would have to be consistently non-empty. Therefore `fcode` must be of the form `(_, COMPLETE [], _, _)`, so `isempty(fcode)` evaluates to true.

6.2 We first prove that `comb1(i1,s2)` is nice. There are two cases according to the definition of `comb1`. **Case 1:** Assume `s2` is of the form `COMPLETE i2`. Then `i2` is nice. Then `i1 @ i2` is nice (`i1` does not end with a label, so even if `i2` begins with a label, `i1 @ i2` does not contain two adjacent label instructions). **Case 2:** Assume `s2` is of the form `INCOMPLETE f2`. For every label `lab`, `f2(lab)` is nice, so again `i1 @ i2` is nice. We then prove that `comb1(i1,s2)` is consistent. If both `i1` and `s2` are empty then first clause in the definition of `comb1` applies with `i2=[]` (because `s2` is consistent) in which case `comb1(i1,s2)` is `COMPLETE []`, which is consistent. Otherwise at least one of `i1` and `s2` are non-empty, in which case the second clause applies and `comb1(i1,s2)` is consistent.

6.3 By definition of `%`, the claim is obvious in the case where `q1` or `q2` is empty. Thus there remains three cases corresponding to the three cases of the `case` expression in the definition of `%`. **Case 1:** Here `i1` and `s2` are nice and consistent, so `comb1(i1,s2)` is nice and consistent by a previous exercise. Also, `q1` is non-empty, so `comb(i1,s2)` begins with a label `lab` if and only if `i1` begin with `lab`. Therefore `q1 % q2` is consistent in this case. **Case 2:** Here `backpatch(l2,q1)` is nice and consistent and has the form covered by Case 1. **Case 3:** Since `q2` is consistent, the stream in `s2` does not begin with a label. Thus `prefix(l2,q2)` is nice and consistent. Also, `backpatch(l2,q1)` is consistent and nice, so Case 3 reduces to Case 1.

6.4 NEXT

6.5 2 holes

6.6 NEXT (inherited from the outer if)

6.7 The code for the inner if is ultimately applied to the continuation `GOTO whilentry14`. Then `mkJCPCode(GOTO whilentry14, NEXT)` evaluates to `(JPC whilentry14, emptyfixedcode, emptyfixedcode)`.

6.8

main program

```
INT 2
```
whilentry34
  LOD(0,1)
  LIT 0
  OPR 8
  JPC ret35
  LOD(0,1)
  LIT 0
  OPR 8
  JPC whilentry34
  JMP whilentry34

ret35
  RST CONT
  OPR 0

6.9 Because the jmpcode will be backpatched to the beginning of nojumpcode % fcode2 % perhaps_return. The effect would be as if all conditions were always evaluated to true.

7.1 The connect(req, req2) might not make sense. Assume $d$ is a variable which is specified as being dead in req. There is no guarantee that $d$ will also be dead in req2; then connect(req, req2) makes no sense.

7.2 The requirement map req is not in general the requirements of the computation that follows after the computation of exp, for the computation after exp might involve several computations of stmt and exp itself. It therefore makes no sense to try to propagate req back through exp. If one does, the application connect(req, req2) might not make sense, because exp or stmt might contain live variables that are specified as dead in req.

7.3 The requirement map req1 = \{b : (ENV, R0)\}, i.e. the result of distributing registers over b > 0 with post-requirement req0.

7.4 The register allocation for b := b/2 takes place relative to the requirement map req1 defined in the previous solution. After the elaboration of b/2, we record that b is in R0 only. The store instruction serves to connect from this requirement map to req1.

7.5 The pre-requirement map which is obtained by distributing registers over a := 2 * a; b := b/2 using req1 as post-requirement is the requirement map req2 = \{a : R2, b : R0\}. This distribution in itself does not generate any load instructions. However, at the end of the preceding if statement, we have to connect from req0 to req2, and this is where the load instructions come from.