16 SIMPLE BLOCK STRUCTURE

Our simple language has so far not provided for the *procedure* concept in any way. It is the aim of the next two chapters to show how Clang and its compiler can be extended to provide procedures and functions, using a model based on those found in block-structured languages like Modula-2 and Pascal, which allow the use of local variables, local procedures and recursion. This involves a much deeper treatment of the concepts of storage allocation and management than we have needed previously.

As in the last two chapters, we shall develop our arguments by a process of slow refinement. On the source diskette will be found Cocol grammars, hand-crafted parsers and code generators covering each stage of this refinement, and the reader is encouraged to study this code in detail as he or she reads the text.

16.1 Parameterless procedures

In this chapter we shall confine discussion to parameterless **regular procedures** (or *void functions* in C++ terminology), and discuss parameters and value-returning functions in the following chapter.

16.1.1 Source handling, lexical analysis and error reporting

The extensions to be discussed in this chapter require no changes to the source handler, scanner or error reporter classes that were not pre-empted in the discussion in Chapter 14.

16.1.2 Syntax

Regular procedure declaration is inspired by the way it is done in Modula-2, described in EBNF by

```
   Block            =   { ConstDeclarations | VarDeclarations | ProcDeclaration }
                        CompoundStatement .
   ProcDeclaration  =  "PROCEDURE" ProcIdentifier ";" Block ";" .
```

It might be thought that the same effect could be achieved with

```
   ProcDeclaration  =  "PROCEDURE" ProcIdentifier ";" CompoundStatement ";" .
```

but the syntax first suggested allows for nested procedures, and for named constants and variables to be declared local to procedures, in a manner familiar to all Modula-2 and Pascal programmers.

The declaration of a procedure is most easily understood as a process whereby a *CompoundStatement* is given a name. Quoting this name at later places in the program then implies execution of that *CompoundStatement*. By analogy with most modern languages we should like to extend our definition of *Statement* as follows:

```
   Statement        =   [  CompoundStatement | Assignment | ProcedureCall
                          | IfStatement | WhileStatement
                          | WriteStatement | ReadStatement ] .
   ProcedureCall    =  ProcIdentifier .
```

However, this introduces a non-LL(1) feature into the grammar, for now we have two alternatives for *Statement* (namely *Assignment* and *ProcedureCall*) that begin with lexically indistinguishable symbols. There are various ways of handling this problem:

- A purely syntactic solution for this simple language is possible if we re-factor the grammar as

```
Statement         =   [  CompoundStatement | AssignmentOrCall
                       | IfStatement | WhileStatement
                       | WriteStatement | ReadStatement ] .
AssignmentOrCall  =  Designator [ ":=" Expression ] .
```

  so that a *ProcedureCall* can be distinguished from an *Assignment* by simply looking at the first symbol after the *Designator*. This is, of course, the approach that has to be followed when using Coco/R.

- A simple solution would be to add the keyword `CALL` before a procedure identifier, as in Fortran, but this rather detracts from readability of the source program.

- Probably because the semantics of procedure calls and of assignments are so different, the solution usually adopted in a hand-crafted compiler is a static semantic one. To the list of allowed classes of identifier we add one that distinguishes procedure names uniquely. When the symbol starting a *Statement* is an identifier we can then determine from its symbol table attributes whether an *Assignment* or *ProcedureCall* is to be parsed (assuming all identifiers to have been declared before use, as we have been doing).

16.1.3 The scope and extent of identifiers

Allowing nested procedures - or even local variables on their own - introduces the concept of **scope**, which should be familiar to readers used to block-structured languages, although it often causes confusion to many beginners. In such languages, the "visibility" or "accessibility" of an identifier declared in a *Block* is limited to that block, and to blocks themselves declared local to that block, with the exception that when an identifier is *redeclared* in one or more nested blocks, the innermost accessible declaration applies to each particular use of that identifier.

Perhaps any confusion which arises in beginners’ minds is exacerbated by the fact that the rather fine distinction between compile-time and run-time aspects of scope is not always made clear. At compile-time, only those names that are currently "in scope" will be recognized when translating statements and expressions. At run-time, each variable has an **extent** or **lifetime**. Other than the "global variables" (declared within the main program in Modula-2 or Pascal, or outside of all functions in C++), the only variables that "exist" at any one instant (that is, have storage allocated to them, with associated values) are those that were declared local to the blocks that are "active" (that is, are associated with procedures that have been called, but which have not yet returned).

One consequence of this, which a few readers may have fallen foul of at some stage, is that variables declared local to a procedure cannot be expected to retain their values between calls on the procedure. This leads to a programming style where many variables are declared globally, when they should, ideally, be "out of scope" to many of the procedures in the program. (Of course, the use of modules (in languages like Modula-2) or classes (in C++) allows many of these to be hidden safely away.)

Exercises

16.1 Extend the grammar for Topsy so as to support a program model more like that in C and C++, in which routines may not be nested, although both global and local variables (and constants) may be declared.

16.1.4 Symbol table support for the scope rules

Scope rules like those suggested in the last section may be easily handled in a number of ways, all of which rely on some sort of stack structure. The simplest approach is to build the entire symbol table as a stack, pushing a node onto this stack each time an identifier is declared, and popping several nodes off again whenever we complete parsing a *Block*, thereby ensuring that the names declared local to that block then go out of scope. The stack structure also ensures that if two identifiers with the same names are declared in nested blocks, the first to be found when searching the table will be the most recently declared. The stack of identifier entries must be augmented in some way to keep track of the divisions between procedures, either by introducing an extra variant into the possibilities for the `TABLE_entries` structure, or by constructing an ancillary stack of special purpose nodes.

The discussion will be clarified by considering the shell of a simple program:

```
PROGRAM Main;
  VAR G1;                  (* global *)

  PROCEDURE One;
    VAR L1, L2;            (* local to One *)
    BEGIN
      (* body of One *)
    END;

  BEGIN
    (* body of Main *)
  END.
```

For this program, either of the approaches suggested by Figure 16.1(a) or (b) would appear to be suitable for constructing a symbol table. In these structures, an extra "sentinel" node has been inserted at the bottom of the stack. This allows a search of the table to be implemented as simply as possible, by inserting a copy of the identifier that is being sought into this node before the (linear) search begins.

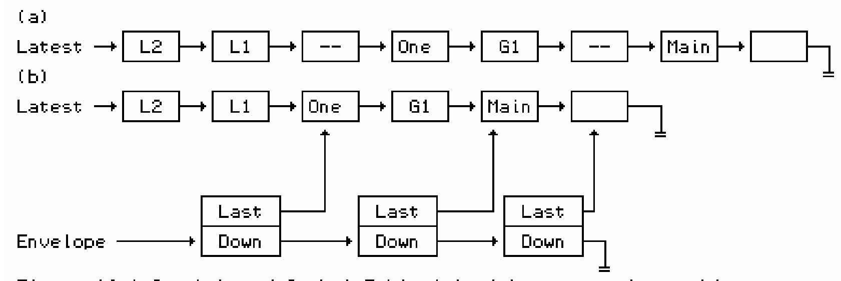


Figure 16.1 Stack based Symbol Table (a) with extra nodes marking scope boundaries (b) with ancillary stack marking scope boundaries

As it happens, this sort of structure becomes rather more difficult to adapt when one extends the language to allow procedures to handle parameters, and so we shall promote the idea of having a stack of *scope nodes,* each of which contains a pointer to the scope node corresponding to an outer scope, as well as a pointer to a structure of *identifier nodes* pertinent to its own scope. This latter structure could be held as a stack, queue, tree, or even hash table. Figure 16.2 shows a situation where queues have been used, each of which is terminated by a common sentinel node.

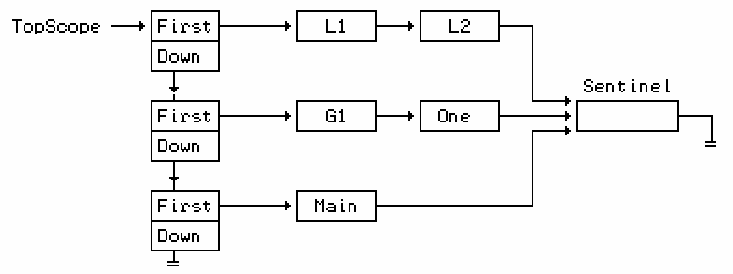


Figure 16.2 Symbol Table based on a set of queues, with ancillary stack marking scope boundaries

Although it may not immediately be seen as necessary, it turns out that to handle the addressing aspects needed for code generation we shall need to associate with each identifier the *static level* at which it was declared. The revised public interface to the symbol table class requires declarations like

```
  enum TABLE_idclasses { TABLE_consts, TABLE_vars, TABLE_progs, TABLE_procs };

  struct TABLE_entries {
    TABLE_alfa name;             // identifier
    int level;                   // static level
    TABLE_idclasses idclass;     // class
    union {
      struct {
        int value;
      } c;                       // constants
      struct {
        int size, offset;
        bool scalar;
      } v;                       // variables
      struct {
        CGEN_labels entrypoint;
      } p;                       // procedures
    };
  };

  class TABLE {
    public:
      void openscope(void);
      // Opens new scope before parsing a block

      void closescope(void);
      // Closes scope after parsing a block

      // rest as before (see section 14.6.3)
  };
```

On the source diskette can be found implementations of this symbol table handler, while a version extended to meet the requirements of Chapter 17 can be found in Appendix B. As usual, a few comments on implementation techniques may be helpful to the reader:

- The symbol table handler manages the computation of `level` internally.

- An `entry` is passed by reference to the `enter` routine, so that, when required, the caller is able to retrieve this value after an entry has been made.

- The outermost program block can be defined as level 1 (some authors take it as level 0, others reserve this level for standard "pervasive" identifiers - like `INTEGER, BOOLEAN, TRUE` and `FALSE`).

- It is possible to have more than one entry in the table with the same `name`, although not within a single scope. The routine for adding an entry to the table checks that this constraint is obeyed. However, a second occurrence of an identifier in a single scope will result in a further entry in the table.

- The routine for searching the symbol table works its way through the various scope levels from innermost to outermost, and is thus more complex than before. A call to `search` will, however, always return with a value for `entry` which matches the `name`, even if this had not been correctly declared previously. Such undeclared identifiers will seem to have an effective `idclass = TABLE_progs`, which will always be semantically unacceptable when the identifier is analysed further.

Exercises

16.2 Follow up the suggestion that the symbol table can be stored in a stack, using one or other of the methods suggested in Figure 16.1.

16.3 Rather than use a separate `SCOPE_nodes` structure, develop a version of the symbol table class that simply introduces another variant into the existing `TABLE_entries` structure, that is, extend the enumeration to

```
enum TABLE_idclasses { TABLE_consts, TABLE_vars, TABLE_progs, TABLE_procs,
TABLE_scopes };
```

16.4 How, if at all, does the symbol table interface require modification if you wish to develop the Topsy language to support `void` functions?

16.5 In our implementation of the table class, scope nodes are deleted by the `closescope` routine. Is it possible or advisable also to delete identifier nodes when identifiers go out of scope?

16.6 Some compilers make use of the idea of a forest of binary search trees. Develop a table handler to make use of this approach. How do you adapt the idea that a call to `search` will always return a well-defined `entry`?

For example, given source code like

```
PROGRAM Silly;
  VAR B, A, C;

  PROCEDURE One;
    VAR X, Y, Z;

    PROCEDURE Two;
      VAR Y, D;
```

the symbol table might look like that shown in Figure 16.3 immediately after declaring `D`.

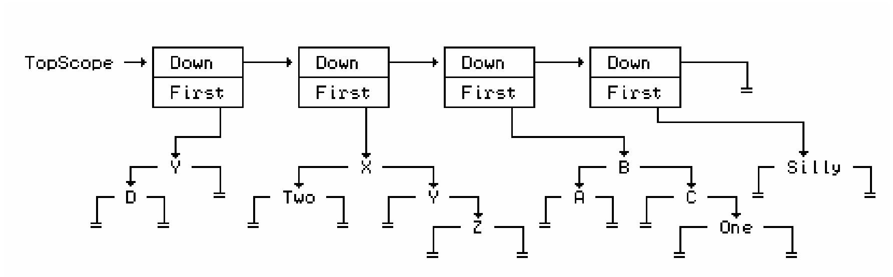


Figure 16.3 Symbol Table based on a forest of Binary Search Trees

Further reading

More sophisticated approach to symbol table construction are discussed in many texts on compiler construction. A very readable treatment may be found in the book by Elder (1994), who also discusses the problems of using a hash table approach for block-structured languages.

16.1.5 Parsing declarations and procedure calls

The extensions needed to the attributed grammar to describe the process of parsing procedure declarations and calls can be understood with reference to the Cocol extract below, where, for temporary clarity, we have omitted the actions needed for code generation, while retaining those needed for constraint analysis:

```
PRODUCTIONS
  Clang
  =                          (. TABLE_entries entry; .)
     "PROGRAM"
     Ident<entry.name>       (. entry.idclass = TABLE_progs;
                                Table->enter(entry); Table->openscope(); .)
     WEAK ";" Block "." .

  Block
  =  SYNC
     { ( ConstDeclarations | VarDeclarations | ProcDeclaration ) SYNC }
     CompoundStatement       (. Table->closescope(); .) .

  ProcDeclaration
  =                          (. TABLE_entries entry; .)
     "PROCEDURE"
     Ident<entry.name>       (. entry.idclass = TABLE_procs;
                                Table->enter(entry); Table->openscope(); .)
     WEAK ";"  Block ";" .

  Statement
  =  SYNC [  CompoundStatement | AssignmentOrCall | IfStatement
          | WhileStatement | ReadStatement | WriteStatement ] .

  AssignmentOrCall
  =                          (. TABLE_entries entry; .)
     Designator<classset(TABLE_vars, TABLE_procs), entry>
     (                       (. if (entry.idclass != TABLE_vars) SemError(210); .)
        ":=" Expression SYNC
       |                     (. if (entry.idclass != TABLE_procs) SemError(210); .)
     ) .
```

The reader should note that:

- Variables declared local to a *Block* will be associated with a level one higher than the block identifier itself.
- In a hand-crafted parser we can resolve the LL(1) conflict between the assignments and procedure calls within the parser for *Statement*, on the lines of

```
void Statement(symset followers)
{ TABLE_entries entry; bool found;
  if (FirstStatement.memb(SYM.sym))             // allow for empty statements
  { switch (SYM.sym)
    { case SCAN_identifier:                     // must resolve LL(1) conflict
        Table->search(SYM.name, entry, found);  // look it up
        if (!found) Report->error(202);         // undeclared identifier
        if (entry.idclass == TABLE_procs) ProcedureCall(followers, entry);
        else Assignment(followers, entry);
        break;
      case SCAN_ifsym:                          // other statement forms
        IfStatement(followers); break;          // as needed
  }
  test(followers, EmptySet, 32);                // synchronize if necessary
}
```

Exercises

16.7 In Exercise 14.50 we suggested that undeclared identifiers might be entered into the symbol table (and assumed to be variables) at the point where they were first encountered. Investigate whether one can do better than this by examining the symbol which appears after the offending identifier.

16.8 In a hand-crafted parser, when calling *Block* from within *ProcDeclaration* the semicolon symbol has to be added to *Followers*, as it becomes the legal follower of *Block*. Is it a good idea to do this, since the semicolon (a widely used and abused token) will then be an element of all *Followers* used in parsing parts of that block? If not, what does one do about it?

16.2 Storage management

If we wish procedures to be able to call one another recursively, we shall have to think carefully about code generation and storage management. At run-time there may at some stage be several *instances* of a recursive procedure in existence, pending completion. For each of these the corresponding instances of any local variables must be distinct. This has a rather complicating effect at compile-time, for a compiler can no longer associate a simple address with each variable as it is declared (except, perhaps, for the global variables in the main program block). Other aspects of code generation are not quite such a problem, although we must be on our guard always to generate so-called *re-entrant code*, which executes without ever modifying itself.

16.2.1 The stack frame concept

Just as the stack concept turns out to be useful for dealing with the compile-time accessibility aspects of *scope* in block-structured languages, so too do stack structures provide a solution for dealing with the run-time aspects of *extent* or *existence*. Each time a procedure is called, it acquires a region of free store for its local variables - an area which can later be freed when control returns to the caller. On a stack machine this becomes almost trivially easy to arrange, although it may be more obtuse on other architectures. Since procedure activations strictly obey a first-in-last-out scheme, the areas needed for their local working store can be carved out of a single large stack. Such areas are usually called **activation records** or **stack frames**, and do not themselves contain any code. In each of them is usually stored some standard information. This includes the **return address** through which control will eventually pass back to the calling procedure, as well as information that can later be used to reclaim the frame storage when it is no longer required. This housekeeping section of the frame is called the **frame header** or **linkage area**. Besides the storage needed for the frame header, space must be also be allocated for local variables (and, possibly, parameters, as we shall see in a later section).

This may be made clearer by a simple example. Suppose we come up with the following variation on code for satisfying the irresistible urge to read a list of numbers and write it down in reverse order:

```
PROGRAM Backwards;
  VAR Terminator;

  PROCEDURE Start;
    VAR Local1, Local2;

    PROCEDURE Reverse;
      VAR Number;
      BEGIN
        Read(Number);
        IF Terminator <> Number THEN Start;  10: Write(Number)
      END;

    BEGIN (* Start *)
      Reverse;  20:
    END;

  BEGIN (* Backwards *)
    Terminator := 9;
    Start; 30:
  END (* Backwards *).
```

(Our language does not provide for labels; these have simply been added to make the descriptions easier.)

We note that a stack is also the obvious structure to use in a non-recursive solution to the problem, so the example also highlights the connection between the use of stacks to implement recursive ideas in non-recursive languages.

If this program were to be given exciting data like `56 65 9`, then its dynamic execution would result in a stack frame history something like the following, where each line represents the relative layout of the stack frames as the procedures are entered and left.

```
                          Stack grows ---->
start main program        Backwards
call Start                Backwards   Start
call Reverse              Backwards   Start   Reverse
read 56 and recurse       Backwards   Start   Reverse   Start
   and again              Backwards   Start   Reverse   Start   Reverse
read 65 and recurse       Backwards   Start   Reverse   Start   Reverse   Start
   and again              Backwards   Start   Reverse   Start   Reverse   Start   Reverse
read 9, write 9, return   Backwards   Start   Reverse   Start   Reverse   Start
   and again              Backwards   Start   Reverse   Start   Reverse
write 65 and return       Backwards   Start   Reverse   Start
   and again              Backwards   Start   Reverse
write 56 and return       Backwards   Start
   and again              Backwards
```

At *run-time* the actual address of a variable somewhere in memory will have to be found by subtracting an *offset* (which, fortunately, *can* be determined at *compile-time*) from the address of the appropriate stack frame (a value which, naturally but unfortunately, *cannot* be predicted at compile-time). The code generated at compile-time must contain enough information for the run-time system to be able to find (or calculate) the base of the appropriate stack frame when it is needed. This calls for considerable thought.

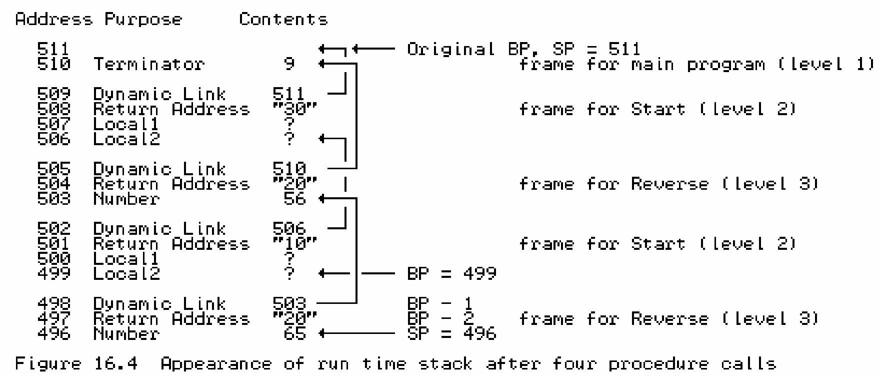
The run-time stack frames are conveniently maintained as a linked list. As a procedure is called, it can set up (in its frame header) a pointer to the base of the stack frame of the procedure that called it. This pointer is usually called the **dynamic link**. A pointer to the top of this linked structure - that is, to the base of the most recently activated stack frame - is usually given special status, and is called the **base pointer**. Many modern architectures provide a special machine register especially suited for use in this role; we shall assume that our target machine has such a register (`BP`), and that on procedure entry it will be set to the current value of the stack pointer `SP`, while on procedure exit it will be reset to assume the value of the dynamic link emanating from the frame header.

If a variable is local to the procedure currently being executed, its run-time address will then be given by `BP - Offset`, where `Offset` can be predicted at compile-time. The run-time address of a non-local variable must be obtained by subtracting its `Offset` from an address found by descending a chain of stack frame links. The problem is to know how far to traverse this chain, and at first seems easily solved, since at declaration time we have already made provision to associate a static declaration level with each entry in the symbol table. When faced with the need to generate code to address an identifier, we can surely generate code (at compile-time) which will use this information to determine (at run-time) how far down the chain to go. This distance at first appears to be easily predictable - nothing other than the difference in levels between the level we have reached in compilation, and the level at which the identifier (to which we want to refer) was declared.

This is nearly true, but in fact we cannot simply traverse the dynamic link chain by that number of steps. This chain, as its name suggests, reflects the *dynamic* way in which procedures are *called* and their frames stacked, while the level information in the symbol table is related to the *static* depth of nesting of procedures as they were *declared*. Consider the case when the program above has just read the second data number `65`. At that stage the stack memory would have the appearance depicted in Figure 16.4, where the following should be noted:

- The number (`511`) used as the highest address is simply for illustrative purposes.

- Since we are assuming that the stack pointer `SP` is decremented *before* an item is pushed onto the stack, the base register `BP` will actually point to an address just above the current top stack frame. Similarly, immediately after control has been transferred to a procedure the stack pointer `SP` will actually point to the last local variable.



```
Address Purpose          Contents

  511                              <-- Original BP, SP = 511
  510   Terminator          9          frame for main program (level 1)

  509   Dynamic Link      511
  508   Return Address   "30"          frame for Start (level 2)
  507   Local1              ?
  506   Local2              ?

  505   Dynamic Link      510
  504   Return Address   "20"          frame for Reverse (level 3)
  503   Number             56

  502   Dynamic Link      506
  501   Return Address   "10"          frame for Start (level 2)
  500   Local1              ?
  499   Local2              ?  <-- BP = 499

  498   Dynamic Link      503      BP - 1
  497   Return Address   "20"      BP - 2    frame for Reverse (level 3)
  496   Number             65  <-- SP = 496
```

Figure 16.4 Appearance of run time stack after four procedure calls

The compiler would know (at compile-time) that `Terminator` was declared at static level 1, and could have allocated it an offset address of 1 (relative to the base pointer that is used for the main program). Similarly, when parsing the reference to `Terminator` within `Reverse`, the compiler would be aware that it was currently compiling at a static level 3 - a level difference of 2. However, generation of code for descending two steps along the dynamic link chain would result (at run-time) in a memory access to a dynamic link masquerading as a "variable" at location 505, rather than to the variable `Terminator` at location 510.

16.2.2 The static link chain

One way of handling the problem just raised is to provide a second chain for linking data segments, one which will be maintained at run-time using only information that has been embedded in the code generated at compile- time. This second chain is called the **static link chain**, and is set up when a procedure is invoked. By now it should not take much imagination to see that calling a procedure is not handled by simply executing a machine level `JSR` instruction, but rather by the execution of a complex activation and calling sequence.

Procedure activation is that part of the sequence that reserves storage for the frame header and evaluates the actual parameters needed for the call. Parameter handling is to be discussed later, but in anticipation we shall postulate that the calling routine initiates activation by executing code that

- saves the current stack pointer `SP` in a special register known as the **mark stack pointer** `MP`, and then

- decrements `SP` so as to reserve storage for the frame header, before

- dealing with the arrangements for transferring any actual parameters.

When a procedure is called, code is first executed that stores in the first three elements of its activation record

- a *static link* - a pointer to the base of the stack frame for the most recently active instance of the procedure within which its source code was nested;

- a *dynamic link* - a pointer to the base of the stack frame of the calling routine;

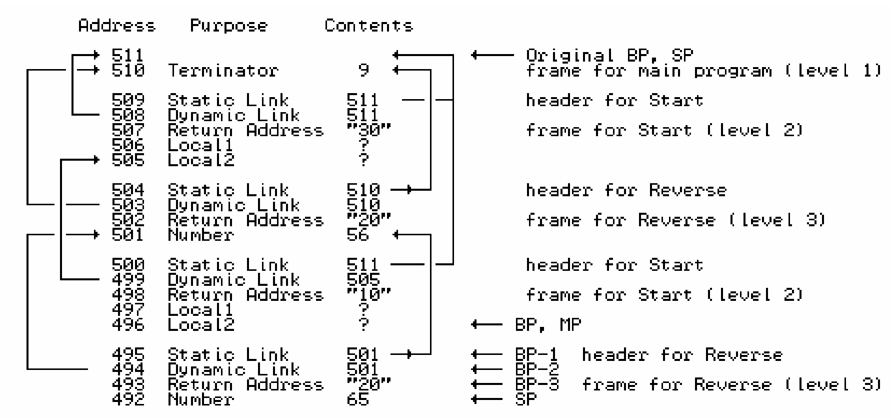
- a *return address* - the code address to which control must finally return in the calling routine;

whereafter the `BP` register can be reset to value that was previously saved in `MP`, and control transferred to the main body of the procedure code.

This calling sequence can, in principle, be associated with either the caller or the called routine. Since a routine is defined once, but possibly called from many places, it is usual to associate most of the actions with the called routine. When this code is *generated*, it incorporates (a) the (known) level difference between the static level from which the procedure is to be called and the static level at which it was declared, and (b) the (known) starting address of the executable code. We emphasize that the static link is only set up at run-time, when code is *executed* that follows the extant static chain from the stack frame of the *calling* routine for as many steps as the level difference between the calling and called routine dictates.

Activating and calling procedures is one part of the story. We also need to make provision for accessing variables. To achieve this, the compiler embeds address information into the generated code. This takes the form of pairs of numbers indicating (a) the (known) level difference between the static level from which the variable is being accessed and the static level where it was declared, and (b) the (known) offset displacement that is to be subtracted from the base pointer of the run-time stack frame. When this code is later executed, the level difference information is used to traverse the static link chain when variable address computations are required. In sharp contrast, the dynamic link chain is used, as suggested earlier, only to discard a stack frame at procedure exit.

With this idea, and for the same program as before, the stack would take on the appearance shown in Figure 16.5.



Address	Purpose	Contents	
511			← Original BP, SP
510	Terminator	9	frame for main program (level 1)
509	Static Link	511	header for Start
508	Dynamic Link	511	
507	Return Address	"30"	frame for Start (level 2)
506	Local1	?	
505	Local2	?	
504	Static Link	510	header for Reverse
503	Dynamic Link	510	
502	Return Address	"20"	frame for Reverse (level 3)
501	Number	56	
500	Static Link	511	header for Start
499	Dynamic Link	505	
498	Return Address	"10"	frame for Start (level 2)
497	Local1	?	
496	Local2	?	← BP, MP
495	Static Link	501	← BP-1 header for Reverse
494	Dynamic Link	501	← BP-2
493	Return Address	"20"	← BP-3 frame for Reverse (level 3)
492	Number	65	← SP

Figure 16.5 Appearance of run time stack after four procedure calls, using static links for non-local variable access

16.2.3 Hypothetical machine support for the static link model

We postulate some extensions to the instruction set of the hypothetical stack machine introduced in section 4.4 so as to support the execution of programs that have simple procedures. We assume the existence of another machine register, the 16-bit `MP`, that points to the frame header at the base of the activation record of the procedure that is about to be called.

One instruction is redefined, and three more are introduced:

`ADR L A`	Push a run-time address onto the stack, for a variable stored at an offset `A` within the stack frame that is `L` steps down the static link chain that begins at the current base register `BP`.
`MST`	Prepare to activate a procedure, saving stack pointer `SP` in `MP`, and then reserving storage for frame header.
`CAL L A`	Call and enter a procedure whose code commences at address `A`, and which was declared at a static difference `L` from the procedure making the call.
`RET`	Return from procedure, resetting `SP`, `BP` and `PC`.

The extensions to the interpreter of section 4.4 show the detailed operational semantics of these instructions:

```
case STKMC_adr:                       // push run time address
  cpu.sp--;                           // decrement stack pointer
  if (inbounds(cpu.sp))
  { mem[cpu.sp] = base(mem[cpu.pc])    // chain down static links
                + mem[cpu.pc + 1];    // and then add offset
    cpu.pc += 2; }                    // bump program count
  break;
case STKMC_mst:                       // procedure activation
  cpu.mp = cpu.sp;                    // set mark stack pointer
  cpu.sp -= STKMC_headersize;         // bump stack pointer
  inbounds(cpu.sp);                   // check space available
  break;
case STKMC_cal:                       // procedure entry
  mem[cpu.mp - 1] = base(mem[cpu.pc]); // set up static link
  mem[cpu.mp - 2] = cpu.bp;           // save dynamic link
  mem[cpu.mp - 3] = cpu.pc + 2;       // save return address
  cpu.bp = cpu.mp;                    // reset base pointer
  cpu.pc = mem[cpu.pc + 1];           // jump to start of procedure
  break;
case STKMC_ret:                       // procedure exit
  cpu.sp = cpu.bp;                    // discard stack frame
```

```
    cpu.pc = mem[cpu.bp - 3];            // get return address
    cpu.bp = mem[cpu.bp - 2];            // reset base pointer
    break;
```

The routines for calling a procedure and computing the run-time address of a variable make use of the small auxiliary routine `base`:

```
  int STKMC::base(int l)
  // Returns base of l-th stack frame down the static link chain
  { int current = cpu.bp;                // start from base pointer
    while (l > 0) { current = mem[current - 1]; l--; }
    return (current);
  }
```

16.2.4 Code generation for the static link model

The discussion in the last sections will be made clearer if we examine the refinements to the compiler in more detail.

The routines for parsing the main program and for parsing nested procedures make appropriate entries into the symbol table, and then call upon *Block* to handle the rest of the source code for the routine.

```
  Clang
  =                            (. TABLE_entries entry; .)
     "PROGRAM"
     Ident<entry.name>         (. entry.idclass = TABLE_progs;
                                  Table->enter(entry); Table->openscope(); .)
     WEAK ";"
     Block<entry.level+1, TABLE_progs, 0>
     "." .

  ProcDeclaration
  =                            (. TABLE_entries entry; .)
     "PROCEDURE"
     Ident<entry.name>         (. entry.idclass = TABLE_procs;
                                  CGen->storelabel(entry.p.entrypoint);
                                  Table->enter(entry); Table->openscope(); .)
     WEAK ";"
     Block<entry.level+1, entry.idclass, CGEN_headersize>
     ";" .
```

We note that:

- The address of the first instruction in any procedure will be stored in the symbol table in the `entrypoint` field of the entry for the procedure name, and retrieved from there whenever the procedure is to be called.

- The parser for a *Block* is passed a parameter denoting its static level, a parameter denoting its class, and a parameter denoting the offset to be assigned to its first local variable. Offset addresses for variables in the stack frame for a procedure start at 4 (allowing for the size of the frame header), as opposed to 1 (for the main program).

Parsing a *Block* involves several extensions over what was needed when there was only a single main program, and can be understood with reference to the attributed production:

```
  Block<int blklevel, TABLE_idclasses blkclass, int initialframesize>
  =                            (. int framesize = initialframesize;
                                  CGEN_labels entrypoint;
                                  CGen->jump(entrypoint, CGen->undefined); .)
     SYNC
     { (   ConstDeclarations
         | VarDeclarations<framesize>
         | ProcDeclaration
       ) SYNC }                (. blockclass = blkclass; blocklevel = blklevel;
                                  // global for efficiency
                                  CGen->backpatch(entrypoint);
```

```
                               CGen->openstackframe(framesize - initialframesize); .)
   CompoundStatement        (. switch (blockclass)
                               { case TABLE_progs :
                                   CGen->leaveprogram(); break;
                                 case TABLE_procs :
                                   CGen->leaveprocedure(); break;
                               }
                               Table->closescope(); .) .
```

in which the following points are worthy of comment:

- Since blocks can be nested, the compiler cannot predict, when a procedure name is *declared*, exactly when the code for that procedure will be *defined*, still less where it will be located in memory. To save a great deal of trouble such as might arise from apparent forward references, we can arrange that the code for each procedure starts with an instruction which may have to branch (over the code for any nested blocks) to the actual code for the procedure body. This initial forward branch is generated by a call to the code generating routine `jump`, and is backpatched when we finally come to generate the code for the procedure body. With a little thought we can see that a simple optimization will allow for the elimination of the forward jump in the common situation where a procedure has no further procedure nested within it. Of course, calls to procedures within which other procedures *are* nested will immediately result in the execution of a further branch instruction, but the loss in efficiency will usually be very small.

- The call to the `openstackframe` routine takes into account the fact that storage will have been allocated for the frame header when a procedure is activated just before it is called.

- The formal parameters `blkclass` and `blklevel` are copied into global variables in the parser to cut down on the number of attributes needed for every other production, and thus improve on parsing efficiency. This rather nasty approach is not needed in Modula-2 and Pascal hand-crafted parsers, where the various routines of the parser can themselves be nested.

- After the *CompoundStatement* has been parsed, code is generated either to halt the program (in the case of a program block), or to effect a procedure return (by calling on `leaveprocedure` to emit a `RET` instruction).

Code for parsing assignments and procedure calls is generated after the LL(1) conflict has been resolved by the call to `Designator`:

```
  AssignmentOrCall
  =                             (. TABLE_entries entry; .)
     Designator<classset(TABLE_vars, TABLE_procs), entry>
     (   /* assignment */       (. if (entry.idclass != TABLE_vars) SemError(210); .)
         ":=" Expression SYNC (. CGen->assign(); .)
       | /* procedure call */ (. if (entry.idclass == TABLE_procs)
                                   { CGen->markstack();
                                     CGen->call(blocklevel - entry.level, entry.p.entrypoint);
                                   }
                                   else SemError(210); .)
     ) .
```

This makes use of two routines, `markstack` and `call` that are responsible for generating code for initiating the activation and calling sequences (for our interpretive system these routines simply emit the `MST` and `CAL` instructions). The routine for processing a *Designator* is much as before, save that it must call upon an extended version of the `stackaddress` code generation routine to emit the new form of the `ADR` instruction:

```
  Designator<classset allowed, TABLE_entries &entry>
  =                             (. TABLE_alfa name;
                                   bool found; .)
     Ident<name>                (. Table->search(name, entry, found);
                                   if (!found) SemError(202);
                                   if (!allowed.memb(entry.idclass)) SemError(206);
                                   if (entry.idclass != TABLE_vars) return;
                                   CGen->stackaddress(blocklevel - entry.level,
                                                      entry.v.offset); .)
     (   "["               (. if (entry.v.scalar) SemError(204); .)
         Expression        (. /* determine size for bounds check */
                                   CGen->stackconstant(entry.v.size);
                                   CGen->subscript(); .)
         "]"
       |                   (. if (!entry.v.scalar) SemError(205); .)
     ) .
```

We observe that an improved code generator could be written to make use of a tree representation for expressions and conditions, similar to the one discussed in section 15.3.2. A detailed Cocol grammar and hand-crafted parsers using this can be found on the source diskette; it suffices to note that virtually no changes have to be made to those parts of the grammar that we have discussed in this section, other than for those responsible for assignment statements.

16.2.5 The use of a "Display"

Another widely used method for handling variable addressing involves the use of a so-called **display**. Since at most one instance of a procedure can be active at one time, only the latest instance of each local variable can be accessible. The tedious business of following a static chain for each variable access at execution time can be eliminated by storing the base pointers for the most recently activated stack frames at each level - the addresses we would otherwise have found after following the static chain - in a small set of dedicated registers. These conceptually form the elements of an array indexed by static level values. Run-time addressing is then performed by subtracting the predicted stack frame offset from the appropriate entry in this array.

When code for a procedure call is required, the interface takes into account the (known) absolute level at which the called procedure was declared, and also the (known) starting address of the executable code. The code to be executed is, however, rather different from that used by the static link method. When a procedure is called it still needs to store the dynamic link, and the return address (in its frame header). In place of setting up the start of the static link chain, the calling sequence updates the display. This turns out to be very easy, as only one element is involved, which can be predicted at compile-time to be the one that corresponds to a static level one higher than that at which the name of the called procedure was declared. (Recall that a *ProcIdentifier* is attributed with the level of the *Block* in which it is declared, and not with the level of the *Block* which defines its local variables and code.)

Similarly, when we leave a procedure, we must not only reset the program counter and base pointer, we may also need to restore a single element of the display. This is strictly only necessary if we have called the procedure from one declared statically at a higher level, but it is simplest to update one element on all returns.

Consequently, when a procedure is called, we arrange for it to store in its frame header:

- a *display copy* - a copy of the current value of the display element for the level one higher than the level of the called routine. This will allow the display to be reset later if necessary.

- a *dynamic link* - a pointer to the base of the stack frame of the calling routine.

- a *return address* - the code address to which control must finally return in the calling routine.

When a procedure relinquishes control, the base pointer is reset from the dynamic link, the program counter is reset from the return address, and one element of the display is restored from the display copy. The element in question is that for a level one higher than the level at which the name of the called routine was declared, that is, the level at which the block for the routine was compiled, and this level information must be incorporated in the code generated to handle a procedure exit.

Information pertinent to variable addresses is still passed to the code generator by the analyser as pairs of numbers, the first giving the (known) level at which the identifier was declared (an absolute level, not the difference between two levels), and the second giving the (known) offset from the run-time base of the stack frame. This involves only minor changes to the code generation interface so far developed.

This should be clarified by tracing the sequence of procedure calls for the same program as before. When only the main program is active, the situation is as depicted in Figure 16.6(a).

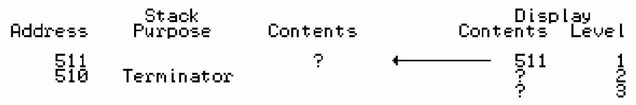


Figure 16.6(a) Stack frames extant immediately after starting execution

After `Start` is activated and called, the situation changes to that depicted in Figure 16.6(b).

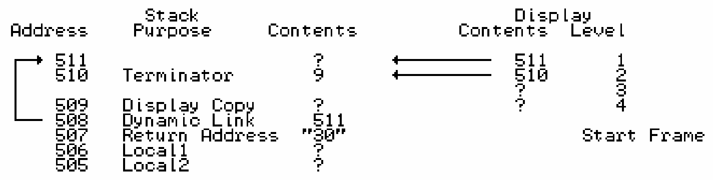


Figure 16.6(b) Stack frames extant immediately after first procedure call

After `Reverse` is called for the first time it changes again to that depicted in Figure 16.6(c).

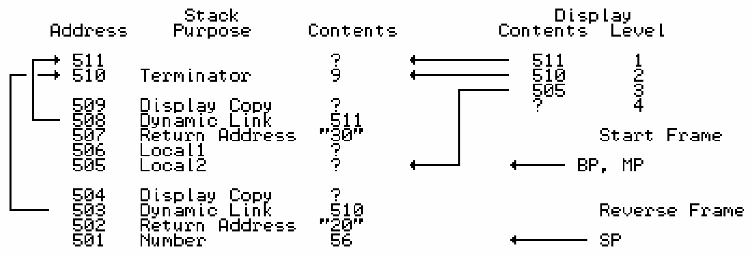


Figure 16.6(c) Stack frames extant immediately after first call to Reverse

After the next (recursive) call to `Start` the changes become rather more significant, as the display copy is now relevant for the first time (Figure 16.6(d)).

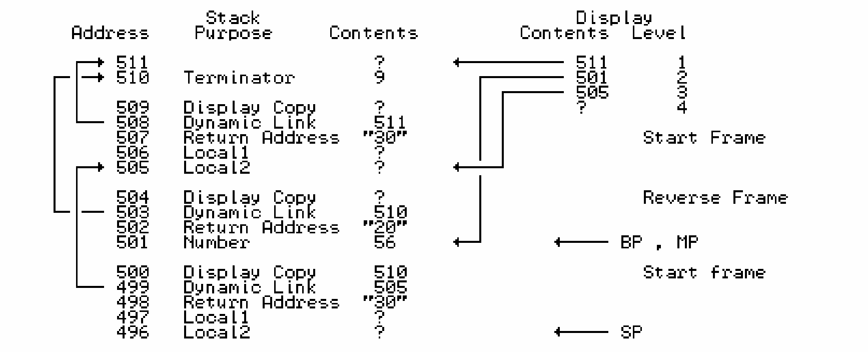


Figure 16.6(d) Stack frames extant immediately after second call to Start

After the next (recursive) call to `Reverse` we get the situation in Figure 16.6(e).

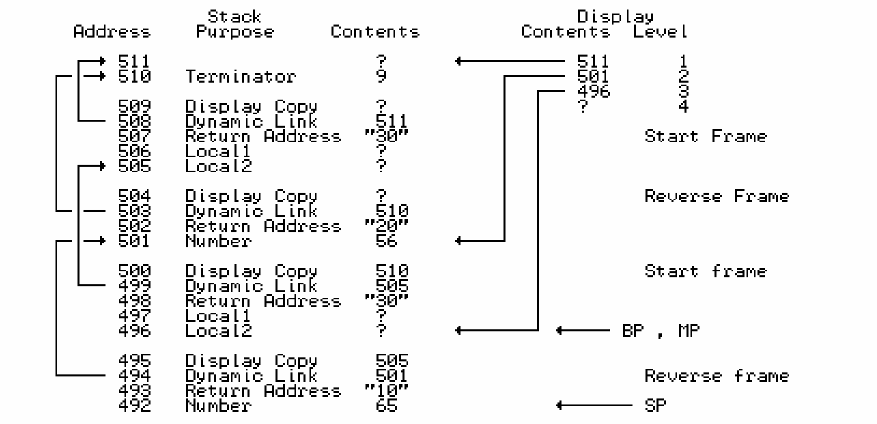


Figure 16.6(e) Stack frames extant immediately after second call to Reverse

When the recursion unwinds, `Reverse` relinquishes control, the stack frame in `495-492` is discarded, and `Display[3]` is reset to `505`. When `Reverse` relinquishes control yet again, the frame in `504-501` is discarded and there is actually no need to alter `Display[3]`, as it is no longer needed. Similarly, after leaving `Start` and discarding the frame in `509-505` there is no need to alter `Display[2]`. However, it is easiest simply to reset the display every time control returns from a procedure.

16.2.6 Hypothetical machine support for the display model

Besides the mark stack register `MP`, our machine is assumed to have a set of display registers, which we can model in an interpreter as a small array, `display`. Conceptually this is indexed from 1, which calls for care in a C++ implementation where arrays are indexed from 0. The `MST` instruction provides support for procedure activation as before, but the `ADR`, `CAL` and `RET` instructions are subtly different:

`ADR L A`	Push a run-time address onto the stack for a variable that was declared at static level `L` and predicted to be stored at an offset `A` from the base of a stack frame.
`CAL L A`	Call and enter a procedure whose *ProcIdentifier* was declared at static level `L`, and whose code commences at address `A`.
`RET L`	Return from a procedure whose *Block* was compiled at level `L`.

The extensions to the interpreter of section 4.4 show the detailed operational semantics of these instructions:

```
 case STKMC_adr:                              // push run time address
   cpu.sp--;                                  // decrement stack pointer
   if (inbounds(cpu.sp))
   { mem[cpu.sp] = display[mem[cpu.pc] - 1]    // extract display element
                   + mem[cpu.pc + 1];          // and then add offset
     cpu.pc += 2; }                           // bump program count
   break;
 case STKMC_cal:                              // procedure entry
   mem[cpu.mp - 1] = display[mem[cpu.pc]];     // save display element
   mem[cpu.mp - 2] = cpu.bp;                  // save dynamic link
   mem[cpu.mp - 3] = cpu.pc + 2;              // save return address
   display[mem[cpu.pc]] = cpu.mp;             // update display
   cpu.bp = cpu.mp;                           // reset base pointer
   cpu.pc = mem[cpu.pc + 1];                  // enter procedure
   break;
 case STKMC_ret:                              // procedure exit
   display[mem[cpu.pc] - 1] = mem[cpu.bp - 1]; // restore display
   cpu.sp = cpu.bp;                           // discard stack frame
   cpu.pc = mem[cpu.bp - 3];                  // get return address
   cpu.bp = mem[cpu.bp - 2];                  // reset base pointer
   break;
```

16.2.7 Code generation for the display model

The attributed productions in a Cocol description of our compiler are very similar to those used in a static link model. The production for *Block* takes into account the new form of the `RET` instruction, and also checks that the limit on the depth of nesting imposed by a finite `display` will not be exceeded:

```
Block<int blklevel, TABLE_idclasses blkclass, int initialframesize>
=                           (. int framesize = initialframesize;
                               CGEN_labels entrypoint;
                               CGen->jump(entrypoint, CGen->undefined);
                               if (blklevel > CGEN_levmax) SemError(213); .)
   SYNC
   { (   ConstDeclarations
       | VarDeclarations<framesize>
       | ProcDeclaration
     ) SYNC }               (. blockclass = blkclass; blocklevel = blklevel;
                               CGen->backpatch(entrypoint);
                               CGen->openstackframe(framesize
                                                    - initialframesize); .)
   CompoundStatement        (. switch (blockclass)
                               { case TABLE_progs :
                                   CGen->leaveprogram(); break;
                                 case TABLE_procs :
                                   CGen->leaveprocedure(blocklevel); break;
                               }
                               Table->closescope(); .) .
```

The productions for *AssignmentOrCall* and for *Designator* require trivial alteration to allow for the fact that the code generator is passed absolute static levels, and not level differences:

```
AssignmentOrCall
=                           (. TABLE_entries entry; .)
   Designator<classset(TABLE_vars, TABLE_procs), entry>
   (  /* assignment */      (. if (entry.idclass != TABLE_vars) SemError(210); .)
      ":=" Expression SYNC  (. CGen->assign(); .)
    | /* procedure call */  (. if (entry.idclass == TABLE_procs)
                               { CGen->markstack();
                                 CGen->call(entry.level, entry.p.entrypoint);
                               }
                               else SemError(210); .)
   ) .

Designator<classset allowed, TABLE_entries &entry>
=                           (. TABLE_alfa name;
                               bool found; .)
   Ident<name>              (. Table->search(name, entry, found);
                               if (!found) SemError(202);
                               if (!allowed.memb(entry.idclass)) SemError(206);
                               if (entry.idclass != TABLE_vars) return;
                               CGen->stackaddress(entry.level, entry.v.offset); .)
   (   "["                  (. if (entry.v.scalar) SemError(204); .)
       Expression           (. /* determine size for bounds check */
                               CGen->stackconstant(entry.v.size);
```

```
                             CGen->subscript(); .)
        "]"
      |                   (. if (!entry.v.scalar) SemError(205); .)
    ) .
```

It may be of interest to show the code generated for the program given earlier. The correct Clang source

```
            PROGRAM Debug;
              VAR Terminator;

              PROCEDURE Start;
                VAR Local1, Local2;

                PROCEDURE Reverse;
                  VAR Number;
                  BEGIN
                    READ(Number);
                    IF Terminator <> Number THEN Start;
                    WRITE(Number)
                  END;

                BEGIN
                  Reverse
                END;

              BEGIN
                Terminator := 9;
                Start
              END.
```

produces the following stack machine code, where for comparison we have shown both models:

```
    Static link       Display

     0 BRN     39        0 BRN     41   jump to start of main program
     2 BRN     32        2 BRN     33   jump to start of Start
     4 DSP      1        4 DSP      1   start of code for Reverse (declared at level 3)
     6 ADR  0  -4        6 ADR  3  -4   address of Number (declared at level 3)
     9 INN               9 INN          read (Number)
    10 ADR  2  -1       10 ADR  1  -1   address of Terminator is two levels down
    13 VAL              13 VAL          dereference - value of Terminator on stack
    14 ADR  0  -4       14 ADR  3  -4   address of Number is on this level
    17 VAL              17 VAL          dereference - value of Number now on stack
    18 NEQ              18 NEQ          compare for inequality
    19 BZE     25       19 BZE     25
    21 MST              21 MST          prepare to activate Start
    22 CAL  2   2       22 CAL  1   2   recursive call to Start
    25 ADR  0  -4       25 ADR  3  -4   address of Number
    28 VAL              28 VAL
    29 PRN              29 PRN          write(Number)
    30 NLN              30 NLN
    31 RET              31 RET      3   exit Reverse
    32 DSP      2       33 DSP      2   start of code for Start (declared at level 2)
    34 MST              35 MST          prepare to activate Reverse
    35 CAL  0   4       36 CAL  2   4   call on Reverse, which is declared at this level
    38 RET              39 RET      2   exit Start
    39 DSP      1       41 DSP      1   start of code for main program (level now 1)
    41 ADR  0  -1       43 ADR  1  -1   address of Terminator on stack
    44 LIT      9       46 LIT      9   push constant 9 onto stack
    46 STO              48 STO          Terminator := 9
    47 MST              49 MST          prepare to activate Start
    48 CAL  0   2       50 CAL  1   2   call Start, which is declared at this level
    51 HLT              53 HLT          stop execution
```

16.2.8 Relative merits of the static link and display models

The display method is potentially more efficient at run-time than the static link method. In some real machines special purpose fast CPU registers may be used to store the display, leading to even greater efficiency. It suffers from the drawback that it seems necessary to place an arbitrary limit on the depth to which procedures may be statically nested. The limit on the size of the display is the same as the maximum static depth of nesting allowed by the compiler at compile-time. Murphy’s Law will ensure that this depth will be inadequate for the program you were going to write to ensure you a niche in the Halls of Fame! Ingenious methods can be found to overcome these problems, but we leave investigation of these to the exercises that follow.

Exercises

16.9 Since Topsy allows only a non-nested program structure for routines like that found in C and C++, its run-time support system need not be nearly as complex as the one described in this section, although use will still need to be made of the stack frame concept. Discuss the implementation of void functions in Topsy in some detail, paying particular attention to the information that would be needed in the frame header of each routine, and extend your Topsy compiler and the hypothetical machine interpreter to allow you to handle multi-function programs.

16.10 Follow up the suggestion that the display does not have to be restored after every return from a procedure. When should the compiler generate code to handle this operation, and what form should the code take? Are the savings worth worrying about? (The Pascal-S system takes this approach (Wirth, 1981; Rees and Robson, 1987).)

16.11 If you use the display method, is there any real need to use the base register `BP` as well?

16.12 If one studies block-structured programs, one finds that many of the references to variables in a block are either to the local variables of that block, or to the global variables of the main program block. Study the source code for the Modula-2 and Pascal implementation of the hand-crafted parsers and satisfy yourself of the truth of this. If this is indeed so, perhaps special forms of addressing should be used for these variables, so as to avoid the inefficient use of the static link search or display reference at run-time. Explore this idea for the simple compiler-interpreter system we are developing.

16.13 In our emulated machine the computation of every run-time address by invoking a function call to traverse the static link chain might prove to be excessively slow if the idea were extended to a native- code generator. Since references to "intermediate" variables are likely to be less frequent than references to "local" or "global" variables, some compilers (for example, Turbo Pascal) generate code that unrolls the loop implicit in the `base` function for such accesses - that is, they generate an explicit sequence of *N* assignments, rather than a loop that is performed *N* times - thereby sacrificing a marginal amount of space to obtain speed. Explore the implications and implementation of this idea.

16.14 One possible approach to the problem of running out of display elements is to store as large a display as will be needed in the frame header for the procedure itself. Explore the implementation of this idea, and comment on its advantages and disadvantages.

16.15 Are there any dangers lurking behind the peephole optimization suggested earlier for eliminating redundant branch instructions? Consider carefully the code that needs to be generated for an `IF ... THEN ... ELSE` statement.

16.16 Can you think of a way of avoiding the unconditional branch instructions with which nearly every enveloping procedure starts, without using all the machinery of a separate forward reference table?

16.17 Single-pass compilers have difficulty in handling some combinations of mutually recursive procedures. It is not always possible to nest such procedures in such a way that they are always "declared" before they are "invoked" in the source code - indeed, in C++ it is not possible to nest procedures (functions) at all. The solution usually adopted is to support the *forward* declaration of procedures. In Pascal, and in some Modula-2 compilers this is done by substituting the keyword `FORWARD` for the body of the procedure when it is first declared. In C++ the same effect is achieved through the use of *function prototypes*.

Extend the Clang and Topsy compilers as so far developed so as to allow mutually recursive routines to be declared and elaborated properly. Bear in mind that all procedures declared `FORWARD` *must* later be defined in full, and at the same level as that where the forward declaration was originally made.

16.18 The poor old `GOTO` statement is not only hated by protagonists of structured programming. It is also surprisingly awkward to compile. If you wish to add it to Clang, why should you prevent users from jumping into procedure or function blocks, and if you let them jump out of them, what special action must be taken to maintain the integrity of the stack frame structures?

Further reading

Most texts on compiling block-structured languages give a treatment of the material discussed here, but this will make more sense after the reader has studied the next chapter.

The problems with handling the `GOTO` statement are discussed in the books by Aho, Sethi and Ullman (1986) and Fischer and LeBlanc (1988, 1991).