CHAPTER 6

Compilers and Interpreters

6.1 ASPECTS OF COMPILATION

A compiler bridges the semantic gap between a PL domain and an execution domain. Two aspects of compilation are:

1. Generate code to implement meaning of a source program in the execution domain.
2. Provide diagnostics for violations of PL semantics in a source program.

To understand the issues involved in implementing these aspects, we briefly discuss PL features which contribute to the semantic gap between a PL domain and an execution domain. These are:

1. Data types
2. Data structures
3. Scope rules
4. Control structure.

Data types

Definition 6.1 (Data type) *A* data type *is the specification of* (*i*) *legal values for variables of the type, and* (*ii*) *legal operations on the legal values of the type.*

Legal operations of a type typically include an assignment operation and a set of data manipulation operations. Semantics of a data type require a compiler to ensure that variables of a type are assigned or manipulated only through legal operations. The following tasks are involved in ensuring this:

1. Checking legality of an operation for the types of its operands. This ensures that a variable is subjected only to legal operations of its type.
2. Use type conversion operations to convert values of one type into values of another type wherever necessary and permissible according to the rules of a PL.
3. Use appropriate instruction sequences of the target machine to implement the operations of a type.

Example 6.1 Consider the Pascal program segment

```
var
   x, y : real;
   i, j : integer;
begin
   y := 10;
   x := y + i;
```

While compiling the first assignment statement, the compiler must note that y is a real variable, hence every value stored in y must be a real number. Therefore it must generate code to convert the value '10' to the floating point representation. In the second assignment statement, the addition cannot be performed on the values of y and i straightaway as they belong to different types. Hence the compiler must first generate code to convert the value of i to the floating point representation and then generate code to perform the addition as a floating point operation.

Having checked the legality of each operation and determined the need for type conversion operations, the compiler must generate *type specific code* to implement an operation. In a type specific code the value of a variable of type $type_i$ is always manipulated through instructions which know how values of $type_i$ are represented.

Example 6.2 In example 1.12, we have seen how the instructions

```
CONV_R      AREG, I
ADD_R       AREG, B
MOVEM       AREG, A
```

are generated for the program segment

```
i : integer;
a,b : real;
a := b+i;
```

Each instruction is type specific, i.e. it expects the operand to be of a specific type.

Generation of type specific code achieves two important things. It implements the second half of a type's definition, viz. a value of $type_i$ is only manipulated through a legal operation of $type_i$. It also ensures execution efficiency since type related issues do not need explicit handling in the execution domain.

Data structures

A PL permits the declaration and use of data structures like arrays, stacks, records, lists, etc. To compile a reference to an element of a data structure, the compiler must develop a memory mapping to access the memory word(s) allocated to the element. A record, which is a heterogeneous data structure, leads to complex memory mappings. A user defined type requires mappings of a different kind—those that map the values of the type into their representations in a computer, and vice versa.

Example 6.3 Consider the Pascal program segment

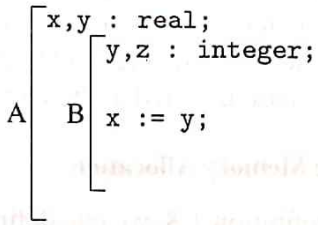
```
program example (input, output);
  type
    employee = record
      name : array [1..10] of character;
      sex : character;
      id : integer
    end;
    weekday = (mon, tue, wed, thu, fri);
  var
    info : array [1..500] of employee;
    today : weekday;
    i,j : integer;
  begin { Main program }
      today := mon;
      info[i].id := j;
      if today = tue then ..
end.
```

Here, `info` is an array of records. The reference `info[i].id` involves use of two different kinds of mappings. The first one is the homogeneous mapping of an array reference. This is used to access `info[i]`. The second mapping is used to access the field `id` within an element of `info`, which is a data item of type `employee`. `weekday` is a user defined data type. The compiler must first decide how to represent different values of the type, and then develop an appropriate mapping between the values `mon .. fri` and their representations. A popular technique is to map these values into a subrange of integers. For example, `mon .. fri` can be mapped into the subrange 1 .. 5. This does not lead to confusion between these values and integers because the legality check is applied to each operation. Thus `tue+10` is an illegal expression even if `tue` is represented by the value '2', because '+' is not a legal operation of type `weekday`.

Scope rules

Scope rules determine the accessibility of variables declared in different blocks of a program. The *scope* of a program entity (e.g. a data item) is that part of a program where the entity is accessible. In most languages the scope of a data item is restricted to the program block in which the data item is declared. It extends to an enclosed block unless the enclosed block declares a variable with an identical name.

Example 6.4



Variable x of block A is accessible in block A and in the enclosed block B. However, variable y of block A is not accessible in block B since y is redeclared in block B. Thus, the statement `x := y` uses y of block B.

The compiler performs operations called *scope analysis* and *name resolution* to determine the data item designated by the use of a name in the source program. The generated code simply implements the results of the analysis. Section 6.2.2 contains a detailed discussion of the implementation of scope rules.

Control structure

The *control structure* of a language is the collection of language features for altering the flow of control during the execution of a program. This includes conditional transfer of control, conditional execution, iteration control and procedure calls. The compiler must ensure that a source program does not violate the semantics of control structures.

Example 6.5 In the Pascal program segment

```
for i := 1 to 100 do
begin
   lab1: if i = 10 then ..
end;
```

a control transfer to the statement bearing label `lab1` from outside the loop is forbidden. Some languages (though not Pascal !) also forbid assignments to the control variable of a **for** loop within the body of the loop.

6.2 MEMORY ALLOCATION

Memory allocation involves three important tasks

1. Determine the amount of memory required to represent the value of a data item.
2. Use an appropriate memory allocation model to implement the lifetimes and scopes of data items.
3. Determine appropriate memory mappings to access the values in a non scalar data item, e.g. values in an array.

The first task is implemented during semantic analysis of data declaration statements. In this section, we discuss the static and dynamic memory allocation models, and memory mappings used in the case of arrays. Discussion of advanced memory mappings to handle data alignment requirements in vector data, e.g. records, can be found in the literature cited at the end of the chapter.

6.2.1 Static and Dynamic Memory Allocation

Following Definition 1.8, we can define memory binding as follows:

Definition 6.2 (Memory binding) *A* memory binding *is an association between the 'memory address' attribute of a data item and the address of a memory area.*

Memory allocation is the procedure used to perform memory binding. The binding ceases to exist when memory is deallocated. Memory bindings can be static or dynamic in nature (see Definitions 1.9 and 1.10), giving rise to the static and dynamic memory allocation models. In *static memory allocation*, memory is allocated to a variable *before* the execution of a program begins. Static memory allocation is typically performed during compilation. No memory allocation or deallocation actions are performed during the execution of a program. Thus, variables remain permanently allocated; allocation to a variable exists even if the program unit in which it is defined is not active. In *dynamic memory allocation*, memory bindings are established and destroyed *during* the execution of a program. Typical examples of the use of these memory allocation models are Fortran for static allocation and block structured languages like PL/I, Pascal, Ada, etc., for dynamic allocation.

Example 6.6 Figure 6.1 illustrates static and dynamic memory allocation to a program consisting of 3 program units—A, B and C. Part (a) shows static memory allocation. Part (b) shows dynamic allocation when only program unit A is active. Part (c) shows the situation after A calls B, while Part (d) shows the situation after B returns to A and A calls C. C has been allocated part of the memory deallocated from B. It is clear that static memory allocation allocates more memory than dynamic memory allocation except when all program units are active.

Dynamic memory allocation has two flavours—automatic allocation and program controlled allocation. According to the terminology of Section 1.4.2, the former implies memory binding performed at execution init time of a program unit, while the latter implies memory binding performed during the execution of a program unit. We describe the details of these bindings in the following.

In *automatic dynamic allocation*, memory is allocated to the variables declared in a program unit when the program unit is entered during execution and is deallocated when the program unit is exited. Thus the same memory area may be used for the variables of different program units (see Fig. 6.1). It is also possible that different memory areas may be allocated to the same variable in different activations of a program unit, e.g. when some procedure is invoked in different blocks of a program. In

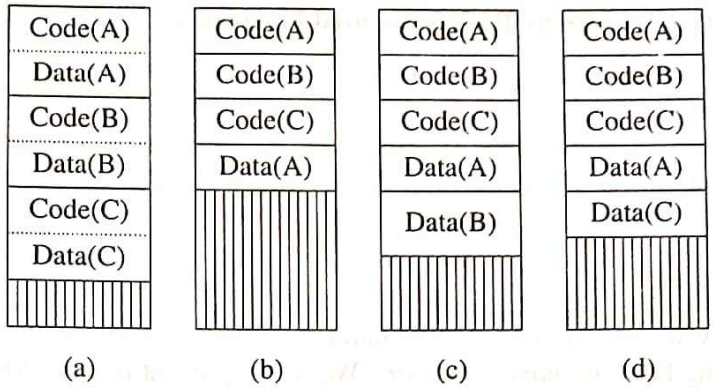


Fig. 6.1 Static and dynamic memory allocation

program controlled dynamic allocation, a program can allocate or deallocate memory at arbitrary points during its execution. It is obvious that in both automatic and program controlled allocation, address of the memory area allocated to a program unit cannot be determined at compilation time.

Dynamic memory allocation is implemented using stacks and heaps, thus necessitating pointer based access to variables. This tends to make it slower in execution than static memory allocation. Automatic dynamic allocation is implemented using a stack since entry and exit from program units is LIFO in nature (see Section 6.2.2 for more details). When a program unit is entered during the execution of a program, a record is created in the stack to contain its variables. A pointer is set to point to this record. Individual variables of the program unit are accessed using displacements from this pointer. Program controlled dynamic allocation is implemented using a heap. A pointer is now needed to point to each allocated memory area.

Example 6.7 Let program unit A contain a declaration for a simple variable `alpha`. When static memory allocation is used the compiler can use the absolute address of `alpha` to implement accesses to it. When dynamic allocation is used, let the compiler maintain the address of the memory record created for A in some register r. Address of variable `alpha` is now

$$< r > + d_{\texttt{alpha}}$$

where $d_{\texttt{alpha}}$ is the displacement of the memory area allocated to `alpha` from the start of the record for A. The constant $d_{\texttt{alpha}}$ is determined during compilation.

Dynamic allocation provides some significant advantages. Recursion can be implemented easily because memory is allocated when a program unit is entered during execution. This leads to allocation of a separate memory area for each recursive activation of a program unit. (We discuss this aspect later in the section.) Dynamic allocation can also support data structures whose sizes are determined dynamically, e.g. an array declaration `a [m,n]`, where `m` and `n` are variables.

6.2.2 Memory Allocation in Block Structured Languages

A *block* is a program unit which can contain data declarations. A program in a block structured language is a nested structure of blocks. A block structured language uses dynamic memory allocation. Algol-60 is the first widely used block structured language. The concept of block structure is also used in PL/I, Algol-68, Pascal, Ada and many other languages.

Scope rules

A data declaration using a name $name_i$ creates a variable var_i and establishes a binding between $name_i$ and var_i. We will represent this binding as ($name_i$, var_i), and call it the name–var binding. Variable var_i is *visible* at a place in the program if some binding ($name_l$, var_i) is effective at that place. A visible variable can be accessed using its name, $name_l$ in the above case. It is possible for data declarations in many blocks of a program to use a same name, say, $name_i$. This would establish many bindings of the form ($name_i$, var_k) for different values of k. Scope rules determine which of these bindings is effective at a specific place in the program.

Consider a block b which contains a data declaration using the name $name_i$. This establishes a binding ($name_i$, var_i) for a variable var_i. This binding only exists within block b. If a block b' nested inside block b also contains a declaration using $name_i$, a binding ($name_i$, var_j) exists in b' for some variable var_j. This suppresses the binding ($name_i$, var_i) over block b'. Thus variable var_i is not visible in b'. However, if b' did not contain a declaration using $name_i$, the binding ($name_i$, var_i) would have been effective over b' as well. Thus, var_i would have been visible in b'. We summarize the rules governing visibility of a variable in the following.

Scope of a variable

If a variable var_i is created with the name $name_i$ in a block b,

1. var_i can be accessed in any statement situated in block b.
2. var_i can be accessed in any statement situated a block b' which is enclosed in b, unless b' contains a declaration using the same name (i.e. using the name $name_i$).

A variable declared in block b is called a *local variable* of block b. A variable of an enclosing block, that is accessible within block b, is called a *nonlocal variable* of block b. The following notation is used to differentiate between variables created using the same name in different blocks:

$name_{block\_name}$: variable created by a data declaration using the name *name* in block *block_name*

Thus $\texttt{alpha}_{\texttt{A}}$, $\texttt{alpha}_{\texttt{B}}$ are variables created using the name `alpha` in blocks A and B.

Example 6.8 Consider the block-structured program in Fig. 6.2. The variables accessible within the various blocks are as follows:

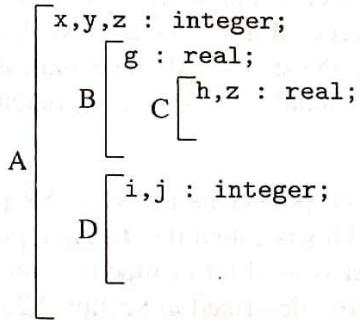


Fig. 6.2 Block structured program

Block	*Accessible variables*	
	local	*nonlocal*
A	x_A, y_A, z_A	–
B	g_B	$x_A, y_A\ z_A$
C	h_C, z_C	x_A, y_A, g_B
D	i_D, j_D	x_A, y_A, z_A

Variable z_A is not accessible inside block C since C contains a declaration using the name z. Thus z_A and z_C are two distinct variables. This would be true even if they had identical attributes, i.e. even if z of C was declared to be an integer.

Memory allocation and access

Automatic dynamic allocation is implemented using the extended stack model (see Section 2.2.1) with a minor variation—each record in the stack has *two* reserved pointers instead of one (see Fig. 6.3). Each stack record accommodates the variables for one activation of a block, hence we call it an *activation record* (AR). The following notation is used to refer to the activation record of a block:

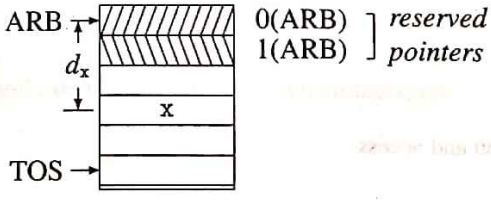


Fig. 6.3 Stack record format

AR_A^i : Activation record for the i^{th} activation of A

wherein we omit the superscript i unless multiple activations of a block exist. During the execution of a block structured program, a register called the *activation record base* (ARB) always points to the start address of the TOS record. This record belongs to the block which contains the statement being executed. A local variable x of this block is accessed using the address d_x(ARB), where d_x is the displacement of variable x from the start of AR. The address may also be written as <ARB> + d_x, where <ARB> stands for the words 'contents of ARB'.

Dynamic pointer

The first reserved pointer in a block's AR points to the activation record of its dynamic parent. This is called the *dynamic pointer* and has the address 0(ARB). The dynamic pointer is used for deallocating an AR. Actions at block entry and exit are analogous to those described in Section 2.2.1.

Example 6.9 Figure 6.4 depicts memory allocation for the following program:

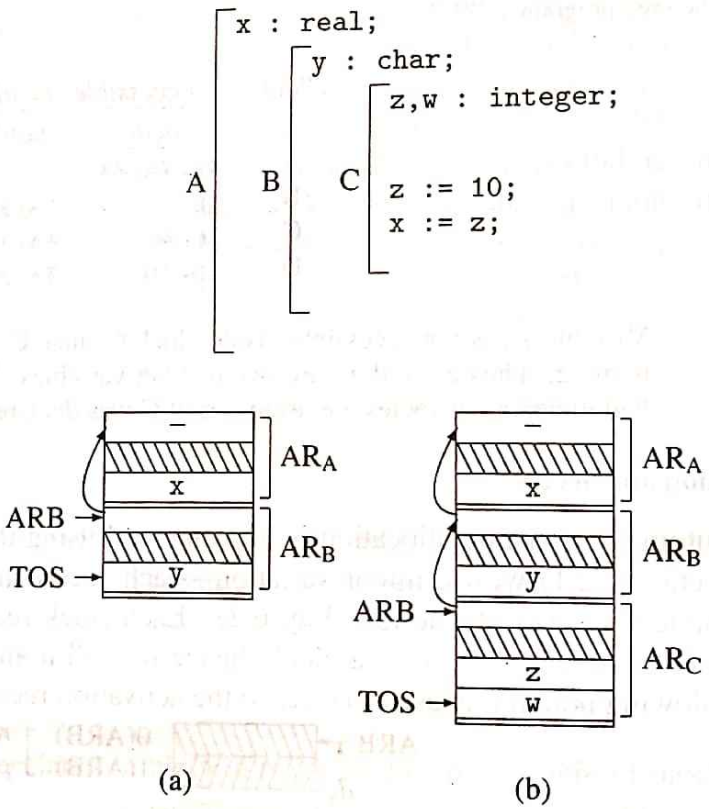


Fig. 6.4 Dynamic allocation and access

where each block could be a **begin** block in Algol, PL/I or a procedure without parameters in some block structured language. It is assumed that the blocks are entered in the sequence A, B, C during the execution of the program.

Figure 6.4(a) shows the situation when blocks A and B are active. Figure 6.4(b) shows the situation after entry to C. Situation after exit from C would again be as shown in

Fig. 6.4(a).

Actions at entry

The actions at entry of block C are described in Table 6.1.

Table 6.1 Actions at block entry

No.			*Statement*
1.	TOS	:=	TOS + 1;
2.	TOS*	:=	ARB; {Set the dynamic pointer}
3.	ARB	:=	TOS;
4.	TOS	:=	TOS + 1;
5.	TOS*	:=	...; {Set reserved pointer 2}
6.	TOS	:=	TOS + n;

where n is the number of memory words allocated to variables of block C, i.e. $n = 2$. After step 1, TOS points to the dynamic pointer of the AR being created. Step 2 sets the dynamic pointer to point to the previous activation record. Step 3 sets ARB to the start of the new AR. Step 6 is the memory allocation step. Note that positions of local variables z and w in AR_C are determined during compilation using their type and length information. d_z and d_w are 2 and 3 assuming an integer to occupy 1 word. Thus the address of variable z is <ARB> + 2.

Actions at exit

The actions at entry of block C are described in Table 6.2.

Table 6.2 Actions at block exit

No.			*Statement*
1.	TOS	:=	ARB − 1;
2.	ARB	:=	ARB*;

The first statement sets TOS to point to the last word of the previous activation record. The second statement sets ARB to point to the start of the previous activation record. These actions effectively deallocate the TOS AR in the stack.

Accessing nonlocal variables

A nonlocal variable *nl_var* of a block *b_use* is a local variable of some block *b_defn* enclosing *b_use* (trivially *b_defn* and *b_use* could be the same block). The following terminology is used for a block and its enclosing blocks. A *textual ancestor* or *static ancestor* of block *b_use* is a block which encloses block *b_use*. The block immediately enclosing *b_use* is called its Level 1 ancestor. A Level m ancestor is a block

which immediately encloses the Level $(m-1)$ ancestor. The *level difference* between *b_use* and its Level m ancestor is m. If $s\_nest_{b\_use}$ represents the static nesting level of block *b_use* in the program, *b_use* has a Level i ancestor, $\forall i < s\_nest_{b\_use}$.

According to the rules of a block structured language, when *b_use* is in execution, *b_defn* must be active. Hence AR_{b_defn} exists in the stack, and *nl_var* is to be accessed as

$$\text{start address of } AR_{b\_defn} + d_{nl\_var}$$

where d_{nl_var} is the displacement of *nl_var* in AR_{b_defn}.

Static pointer

Access to nonlocal variables is implemented using the second reserved pointer in AR. This pointer, which has the address 1(ARB), is called the *static pointer*. When an AR is created for a block *b*, its static pointer is set to point to the AR of the static ancestor of *b*. The code to access a nonlocal variable *nl_var* declared in a Level m ancestor of *b_use*, $m \geq 1$, is as follows:

1. r := ARB; where r is some register.
2. Repeat step 3 m times.
3. $r := 1(r)$; i.e. load the static pointer into register r.
4. Access *nl_var* using the address $< r > + d_{nl\_var}$. (6.1)

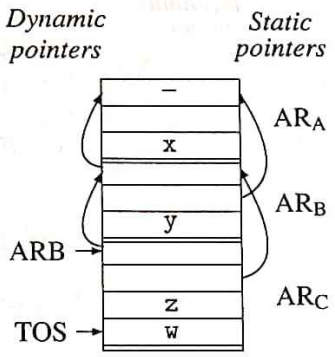
Thus a nonlocal variable defined in a Level m ancestor is accessed using m indirections through the static pointer. Note that the value of m is known during compilation from the static structure of the program.

Example 6.10 Figure 6.5 shows the status of memory allocation after executing statement `z := 10;` of block `C` in the program of Ex. 6.9. Steps 4 and 5 of the actions at block entry are:

No.			*Statement*
4.	TOS	:=	TOS + 1;
5.	TOS*	:=	address of AR of Level 1 ancestor.

When block `C` is entered, these actions set the static pointer in AR_C to point at the start of AR_B. When `B` was entered, the static pointer in AR_B would have been set to point at the start of AR_A. The code generated to access variable `x` in the statement `x := z;` is

1. r := ARB;
2. $r := 1(r)$;
3. $r := 1(r)$;
4. Access `x` using the address $< r > + d_x$.



currently active block → available in stack / which not exited.

Fig. 6.5 Accessing nonlocal variables with static pointers

Displays

For large values of level difference, it is expensive to access nonlocal variables using static pointers. *Display* is an array used to improve the efficiency of nonlocal accesses. When a block B is in execution, the entries of *Display* contain the following information:

$$\begin{aligned}
\text{Display}[1] &= \text{address of Level } (s_nest_b - 1) \text{ ancestor of B.} \\
\text{Display}[2] &= \text{address of Level } (s_nest_b - 2) \text{ ancestor of B.} \\
&\vdots \qquad\qquad \vdots \\
\text{Display}[s_nest_b - 1] &= \text{address of Level 1 ancestor of B.} \\
\text{Display}[s_nest_b] &= \text{address of } AR_B.
\end{aligned}$$

Let block B refer to some variable v_j defined in an ancestor block b_i. The address of v_j is calculated as

$$\text{Display}[s\_nest_{b_i}] + d_{v_j}$$

Hence the code generated for the access would be

1. r := Display[$s\_nest_{b_i}$];
2. Access v_j using the address $< r > + d_{v_j}$; (6.2)

which is more efficient compared with the steps in (6.1).

Example 6.11 Figure 6.6 illustrates the contents of *Display* while executing the program of Ex. 6.9. The reference to variable x in block C is implemented using the code

1. r := Display[1];
2. Access x using the address $< r > + d_x$.

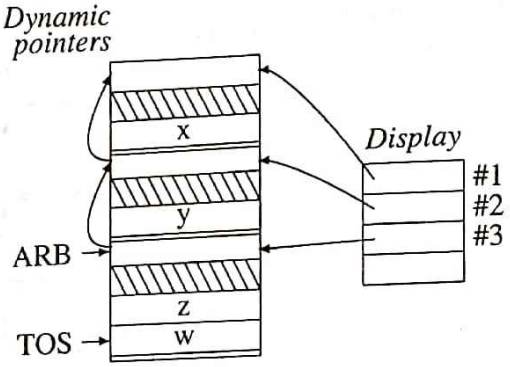


Fig. 6.6 Accessing nonlocal variables using display

Symbol table requirements

Imp

In order to implement dynamic allocation and access, a compiler should perform the following tasks while compiling the use of a name v in *b_current*, the block being compiled:

1. Determine the static nesting level of *b_current*.
2. Determine the variable designated by the name v. This should be done in accordance with the scope rules.
3. Determine the static nesting level of the block in which v is defined. Determine the value of d_v.
4. Generate the code indicated in (6.1) or (6.2).

We present a simple scheme to implement functions 1, 2 and 3. This scheme uses the extended stack model of Section 2.2.1 to organize the symbol table. When the start of block *b_current* is encountered during compilation, a new record is pushed on the stack. It contains

1. Nesting level of *b_current*
2. Symbol table for *b_current*.

The reserved pointer of the new record points to the previous record in the stack. This record contains the symbol table of the static ancestor of *b_current*. Each entry in the symbol table contains a variable's name, type, length and displacement in the AR.

Scope rules are implemented by searching the name v referenced in *b_current* in the symbol table as follows: Symbol table in the topmost record of the stack is searched first. Existence of name v there implies that v designates a local variable of *b_current*. If an entry for v does not exist there, the previous record in the stack is searched. It contains the symbol table for the Level 1 ancestor of *b_current*. Existence of v in it implies that v is a variable declared in the Level 1 ancestor block, and not redeclared in *b_current*. If v is not found there, it is searched in the previous

record of the stack, i.e. in the symbol table of the Level 2 ancestor, and so on. When v is found in a symbol table, its displacement d_v in the AR is obtained from its symbol table entry. The nesting level of the block defining v is obtained from the first field of the stack record containing the symbol table. Code can now be generated to implement the access to variable v as in (6.1) or (6.2).

Example 6.12 Figure 6.7 shows the symbol table for the program of Ex. 6.9. For simplicity, each symbol table entry only shows a symbol and its displacement in AR. The search for variable x accessed in the statement `x := z;` terminates on finding the entry of x in the symbol table for block A. The nesting level of A is 1. $d_x = 2$ and nesting level of C, the current block, is 3. This information is sufficient to generate code as in Ex. 6.10 or Ex. 6.11.

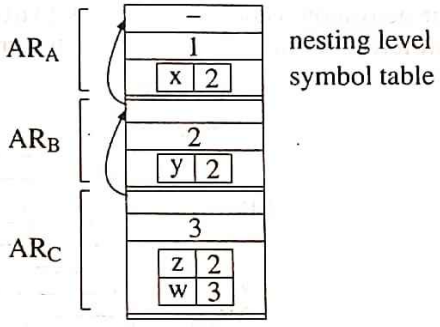


Fig. 6.7 Stack structured symbol table

Recursion

Recursive procedures (or functions) are characterized by the fact that many invocations of a procedure coexist during the execution of a program. A copy of the local variables of the procedure must be allocated for each invocation. This does not pose any problems when a stack model of memory allocation is used because an activation record is created for every invocation of a procedure or function.

Example 6.13 A Pascal program `sample2` contains a function `fib` which computes the n^{th} term of the Fibonacci series.

```
Program sample2 (input, output);
  var
    a,b : integer;
  function fib(n) : integer;
    var
      x : integer;
    begin
      if n > 2 then
        x := fib(n-1) + fib(n-2);
      else x := 1;
      return(x);
```

```
    end fib;
    begin
      fib(4);
end.
```

For the call `fib(4)`, the function makes recursive calls `fib(3)` and `fib(2)`, each of which makes further recursive calls, and so on. Assuming that a formal parameter is allocated memory analogous to a local variable, Fig. 6.8 shows the allocation status after the recursive call `fib(2)`. Creation of multiple AR's for function `fib` is handled naturally by the actions at block entry. Note that the static and dynamic pointers in the activation record for `fib(4)`, i.e. in AR^1_{fib} are identical because `sample2` is both a static and dynamic ancestor of this invocation. However static and dynamic pointers in the activation records for the calls `fib(3)` and `fib(2)`, i.e. in AR^2_{fib} and AR^3_{fib}, are different because their static and dynamic ancestors are different.

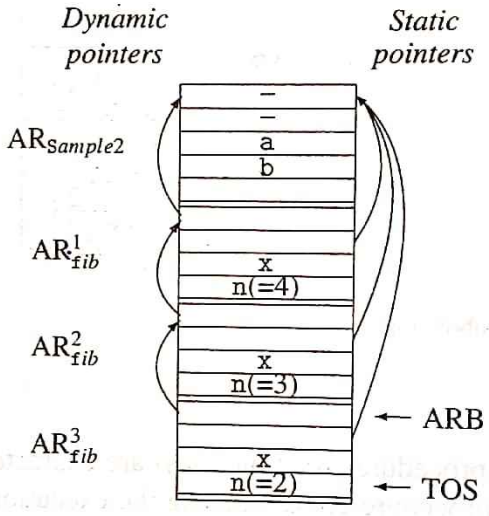


Fig. 6.8 Memory allocation in recursion

Limitations of stack based memory allocation

The stack based memory allocation model is not adequate for program controlled memory allocation, e.g. when the **new** or **dispose** statements of Pascal or `calloc` and `free` functions of C are used to allocate and free memory areas. The compiler must resort to the use of heaps for such allocation. Accesses to variables are now implemented through pointers associated with individual variables rather than with AR's. Free lists or garbage collection are used to implement the reuse of deallocated memory. The stack model is also inadequate for multi-activity programs, i.e. concurrent programs, because in such a program, program blocks may be entered in exited in a non-LIFO manner.

6.2.3 Array Allocation and Access

When an n dimensional array is organized in a computer's memory, the order in which the array elements are arranged in the memory is governed by the rules of the PL. In our discussion, we shall assume array elements to be ordered such that elements occupying adjoining memory locations always differ in the first subscript. For two dimensional arrays, this implies a column-wise arrangement of elements. Figure 6.9 shows the allocation for an array a [5, 10]. It is obvious that static memory allocation is feasible only if dimension bounds of an array are known at compilation time.

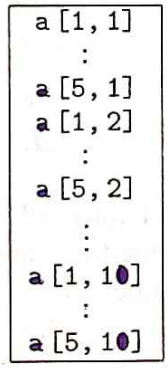


Fig. 6.9 Memory allocation for a 2 dimensional array

The arrangement of array elements in the memory determines the memory mapping to be used in computing the address of an array element. In Fig. 6.9, it has been assumed that the lower bound of each dimension to be 1. Under this assumption, address of an array element a $[s_1, s_2]$ can be determined as follows:

$$Ad.\ \mathrm{a}[s_1, s_2] \quad = \quad Ad.\ \mathrm{a}[1,1] + \{(s_2 - 1) \times n + (s_1 - 1)\} \times k$$

where n is the number of rows in the array and k is the number of memory locations (words or bytes) occupied by each element of the array. For a general two dimensional array

$$\mathrm{a}\,[\,l_1 : u_1, l_2 : u_2\,]$$

where l_i and u_i are the lower and upper bounds of the i^{th} subscript, respectively; address of a $[s_1, s_2]$ is given by the formula

$$Ad.\ \mathrm{a}[s_1, s_2] \quad = \quad Ad.\ \mathrm{a}[l_1, l_2] + \{(s_2 - l_2) \times (u_1 - l_1 + 1) + (s_1 - l_1)\} \times k$$

Defining $range_i$ to be the range of the i^{th} subscript, we have

$$range_1 \quad = \quad u_1 - l_1 + 1,$$
$$range_2 \quad = \quad u_2 - l_2 + 1, \text{ and}$$

$$\begin{aligned} Ad.\,a[s_1,s_2] &= Ad.\,a[l_1,l_2] + \{(s_2-l_2)\times range_1 + (s_1-l_1)\}\times k \\ &= Ad.\,a[l_1,l_2] - (l_2\times range_1 + l_1)\times k \\ &\quad + (s_2\times range_1 + s_1)\times k \\ &= Ad.\,a[0,0] + (s_2\times range_1 + s_1)\times k \end{aligned} \tag{6.3}$$

Note that a [0, 0]—which we will call the *base element* of array a—does not exist unless $l_i \leq 0\ \forall i$. However, this fact is irrelevant to the address calculation formula. Generalizing for an m dimensional array

$$\begin{aligned} Ad.a[s_1,..,s_m] &= Ad.a[0,..,0] \\ &\quad +\{((\ldots(s_m\times range_{m-1} + s_{m-1})\times range_{m-2} \\ &\quad +s_{m-2})\times range_{m-3} + ..)\times range_1 + s_1\}\times k \qquad (6.4) \\ &= Ad.a[0,..,0] + \Sigma_{i=1}^{m}\, s_i \times (\Pi_{j=1}^{i-1} range_j)\times k \qquad (6.5) \end{aligned}$$

where $\Pi_{j=1}^{i-1} range_j = 1$ for $i = 1$. Values of l_i, u_i and $range_i$ for all dimentions of the array can be computed at compilation time (if subscript bounds are constants), or at array allocation time. These values can be stored in an array descriptor called a *dope vector* (DV). Figure 6.10 shows the dope vector format for an m dimensional array. If dimension bounds are known at compilation time, the dope vector needs to exist only during program compilation; it is accommodated in the symbol table entry for the array and its contents are used while generating code for an array reference (see Ex. 6.14). In some machine architectures, (6.5) can be computed more efficiently than (6.4) if $\forall j\ \Pi_{j=1}^{i-1} range_j \times k$ is a precomputed constant. In such cases $\Pi_{j=1}^{i-1} range_j \times k$ can be computed and stored in the dope vector instead of $range_j$.

Ad. a [0,...,0]		
No. of dimensions (m)		
l_1	u_1	$range_1$
l_2	u_2	$range_2$
⋮	⋮	⋮
l_m	u_m	$range_m$

Fig. 6.10 Dope vector

Example 6.14 The code generated for an array reference a [i, j] of a two dimensional array a [1:5, 1:10] is shown in Fig. 6.11. Note that the various subscript bounds and subscript ranges used here become constants of the generated program. Provision is made to call an error routine if any subscript value falls outside the corresponding range.

If array dimension bounds are not known during compilation, the dope vector has to exist during program execution. The number of dimensions of an array determines the format and size of its DV. The dope vector is allocated in the AR of a block and

```
MOVER     AREG, I
COMP      AREG, ='5'
BC        GT, ERROR_RTN      Error if i>5
COMP      AREG, ='1'
BC        LT, ERROR_RTN      Error if i<1
MOVER     AREG, J
COMP      AREG, ='10'
BC        GT, ERROR_RTN      Error if j>10
COMP      AREG, ='1'
BC        LT, ERROR_RTN      Error if j<1
MULT      AREG, ='5'         Multiply by range1
ADD       AREG, I
MULT      AREG, ='2'         Multiply by element size
ADD       AREG, ADDR_ABASE   Add Address of a[0, 0]
```

Fig. 6.11 Code for an array reference

d_{DV}, its displacement in AR, is noted in the symbol table entry of the array. The array is allocated dynamically by repeating step 6 of Table 6.1 for the array. Its start address, values of l_j, u_j and $range_j$ are entered in the DV. The generated code uses d_{DV} to access the contents of DV to check the validity of subscripts and compute the address of an array element.

Example 6.15 Figure 6.12 shows the memory allocation for the following program segment:

```
var
   x,y : real;
   alpha : array [l_1:u_1, l_2:u_2] of integer;
   i,j : integer;
begin
   alpha[i,j] := ...;
```

where l_1, l_2, u_1 and u_2 are nonlocal variables. Since the dimension bounds are not known during compilation, the dope vector must exist during execution. Hence DV is allocated in the AR. d_{alpha_DV}, its displacement in AR, is thus known at compilation time. Array alpha is allocated when the program segment is activated. The address of its base element is recorded in its DV. The generated code uses knowledge of d_{alpha_DV} to access the contents of DV. For example, 4 (ARB) is the *address* field in DV, 6 (ARB) gives information about its first dimension, etc. Note that the field *no. of dimensions* may be omitted from DV since this information is implicit in the generated code.

EXERCISE 6.2

1. Develop an algorithm to build the stack structured symbol table shown in Fig. 6.7.
2. In this section we have not described how memory mapping is performed for records of Pascal, Cobol or structures of C. Refer to Dhamdhere (1997) for a discussion of the same.

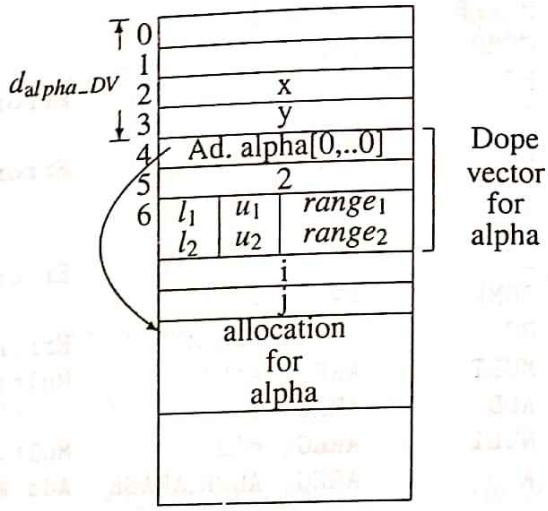


Fig. 6.12 Dope vector for array with dynamic dimension bounds

3. Many languages support bit and character strings with dynamically varying lengths. Would these language features interfere with the use of a stack model for memory allocation?
4. For the recursive procedure `fib`, show contents of the *Display* before and after each recursive call.

6.3 COMPILATION OF EXPRESSIONS

6.3.1 A Toy Code Generator for Expressions

The major issues in code generation for expressions are as follows:

1. Determination of an evaluation order for the operators in an expression.
2. Selection of instructions to be used in the target code.
3. Use of registers and handling of partial results.

The evaluation order of operators depends on operator precedences in an obvious way—an operator which precedes its left and right neighbours must be evaluated before either of them. Hence, a feasible evaluation order is the order in which operators are reduced during a bottom-up parse, or the reverse of the order in which operators are produced during a top down parse. (Other feasible evaluation orders are discussed in the next section.) In the following discussion the use of a bottom up parser to determine the evaluation order has been assumed.

The choice of an instruction to be used in the target code depends on the following:

1. The *type* and *length* of each operand
2. The *addressability* of each operand, i.e. *where* the operand is located and *how* it can be accessed.

The addressability of an operand indicates *where* an operand is located and *how* it can be accessed.

We introduce the notion of an *operand descriptor* to maintain the type, length and addressability information for each operand. Thus the choice of an instruction can be made by analysing an operator and the descriptors of its operands.

A *partial result* is the value of some subexpression computed while evaluating an expression. In the interests of efficiency, partial results are maintained in CPU registers as far as possible. However, some of them have to be moved to memory if the number of results exceeds the number of available CPU registers. An important issue in code generation is when and how to move partial results between memory and CPU registers, and how to know which partial result is contained in a register. We use a *register descriptor* to maintain information for the latter purpose.

We develop a toy code generator for expressions using a YACC like bottom-up parser generator. For simplicity we present a code generator which uses a single CPU register. However, it can be easily extended to handle multiple registers.

Operand descriptors

An operand descriptor has the following fields:

1. *Attributes:* Contains the subfields *type*, *length* and *miscellaneous information*.
2. *Addressability:* Specifies *where* the operand is located, and *how* it can be accessed. It has two subfields
   (a) *Addressability code:* Takes the values 'M' (operand is in memory), and 'R' (operand is in register). Other addressability codes, e.g. address in register ('AR') and address in memory ('AM'), are also possible, however we do not consider them in this section.
   (b) *Address:* Address of a CPU register or memory word.

An operand descriptor is built for every operand participating in an expression, i.e. for *id*'s, constants and partial results. A descriptor is built for an *id* when the *id* is reduced during parsing. A partial result pr_i is the result of evaluating some operator op_j. A descriptor is built for pr_i immediately after code is generated for operator op_j. For simplicity we assume that all operand descriptors are stored in an array called *Operand_descriptor*. This enables us to designate a descriptor by its index in *Operand_descriptor*, e.g. descriptor #*n* is the descriptor in *Operand_descriptor*[*n*].

Example 6.16 The code generated for the expression `a*b` is as follows:

```
MOVER     AREG, A
MULT      AREG, B
```

Three operand descriptors are used during code generation. Assuming `a`, `b` to be integers occupying 1 memory word, these are:

1	(int,1)	M, addr(a)	Descriptor for a
2	(int,1)	M, addr(b)	Descriptor for b
3	(int,1)	R, addr(AREG)	Descriptor for a*b

A skeleton of the code generator is shown in Fig. 6.13. Operand descriptor numbers are used as attributes of NT's. The routine `build_descriptor` called from the semantic action for the production <*F*> ::= <*id*> builds a descriptor for <*id*> and returns its entry number in *Operand_descriptor*. The code generation routine generates code for an operator and builds an operand descriptor for the partial result. Semantic actions of other rules merely copy the operand descriptors as attributes of NT's.

```
%%
E   :  E + T   {$$ = codegen('+', $1, $3)}
    |  T       {$$ = $1}
    ;
T   :  T * F   {$$ = codegen('*', $1, $3)}
    |  F       {$$ = $1}
    ;
F   :  id      {$$ = build_descriptor($1)}
    ;
%%
 build_descriptor(operand)
 {
    i = i + 1;
    operand_descr[i] = ((type), (addressability_code, address))
                                    of operand;
    return i;
 }
```

Fig. 6.13 Skeleton of the code generator

Register descriptors

A register descriptor has two fields

1. *Status:* Contains the code *free* or *occupied* to indicate register status.
2. *Operand descriptor #:* If *status* = *occupied*, this field contains the descriptor # for the operand contained in the register.

Register descriptors are stored in an array called *Register_descriptor*. One register descriptor exists for each CPU register.

Example 6.17 The register descriptor for AREG after generating code for `a*b` as in Ex. 6.16 would be

Occupied	#3

This indicates that register AREG contains the operand described by descriptor #3.

Generating an instruction

When an operator op_i is reduced by the parser, the function codegen is called with op_i and descriptors of its operands as parameters. A single instruction can be generated to evaluate op_i if the descriptors indicate that one operand is in a register and the other is in memory, If both operands are in memory, an instruction is generated to move one of them into a register. This is followed by an instruction to evaluate op_i.

Saving partial results

If all registers are occupied (i.e. they contain partial results) when operator op_i is to be evaluated, a register r is freed by copying its contents into a *temporary location* in the memory. r is now used to evaluate operator op_i. For simplicity we assume that an array `temp` is declared in the target program (i.e. in the generated code) to hold partial results. A partial result is always stored in the next free entry of `temp`. Note that when a partial result is moved to a temporary location, the descriptor of the partial result must change. The *operand descriptor #* field of the register descriptor is used to achieve this.

Example 6.18 Consider the expression `a*b+c*d`. After generating code for `a*b`, the operand and register descriptors would be as shown in Ex. 6.16 and 6.17. After the partial result `a*b` is moved to a temporary location, say `temp[1]`, the operand descriptors must become

1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	M, addr(temp[1])

to indicate that the value of the operand described by operand descriptor # 3 (viz. `a*b`) has been moved to memory location `temp[1]`.

Figure 6.14 shows the complete code generation routine. It first checks the addressabilities of its operands to decide whether one of them exists in a register. If so, it generates a single instruction to perform the operation. If none of the operands is in a register, it needs to move one operand into the register before performing the operation. For this purpose, it first frees the register if it is occupied and changes the operand descriptor of the result it contained previously. (Note how it uses the register descriptor for this purpose.) It now moves one operand into the register and generates an instruction to perform the operation. At the end of code generation, it builds a descriptor for the partial result.

Example 6.19 Code generation steps for the expression `a*b+c*d` are shown in Tab. 6.3, where the superscript of an NT shows the operand descriptor # used as its attribute, and the notation $<id>_v$ represents the token $<id>$ constructed for symbol v.

Figure 6.15(a) shows the parse tree and operand and register descriptors after step 8. Figure 6.15(b) shows the parse tree and descriptors after step 9. Note that operand descriptor #3 has been changed to indicate that the partial result `a*b` has been moved

```
codegen (operator, opd1, opd2)
{
   if opd1.addressability_code = 'R'
        /* Code generation -- case 1 */
      if operator = '+' generate 'ADD AREG, opd2';
        /* Analogous code for other operators */
   else if opd2.addressability_code = 'R'
        /* Code generation -- case 2 */
      if operator = '+' generate 'ADD AREG, opd1';
        /* Analogous code for other operators */
   else
        /* Code generation -- case 3 */
      if Register_descr.status = 'Occupied'
        /* Save partial result */
        generate  ('MOVEM AREG, Temp[j]');
        j = j + 1;
        Operand_descr[Register_descr.Operand_descriptor#]
                    = (<type>, (M, Addr(Temp[j])));
      /* Generate code */
      generate 'MOVER AREG, opd1';
      if operator = '+' generate 'ADD AREG, opd2';
        /* Analogous code for other operators */
   /* Common part -- Create a new descriptor
   Saying operand value is in register AREG */
   i = i + 1;
   operand_descr[i] = (<type>,('R', Addr(AREG)));
   Register_descr = ('Occupied', i);
   return i;
}
```

Fig. 6.14 Code generation routine

to the temporary location `temp[1]`. The register descriptor now points to operand descriptor #6 which describes the partial result `c*d`.

To see how temporary locations are used to hold partial results, consider the source string `a*b+c*d*(e+f)+c*d`. Figure 6.16 shows the expression tree for the string wherein operator numbers indicate the bottom up evaluation order. Figure 6.17(a) shows the code generated for this string using the code generation routine of Fig. 6.14. The contents of the temporary locations and their usage are summarized in Table 6.4.

The general rule concerning the use of partial results is that the partial result saved last is always used before the results saved earlier. For example, the value of `c*d` is used before the value of `a*b`. (It is easy to prove the rule by contradiction—if this fact does not hold, we must have evaluated the operators in some order other than the bottom up evaluation order!) It is therefore possible to use a LIFO data structure,

Table 6.3 Code generation actions for `a*b+c*d`

Step no.	*Parsing action*	*Code generation action*
1.	$<id>_a \rightarrow F^1$	Build descriptor # 1
2.	$F^1 \rightarrow T^1$	–
3.	$<id>_b \rightarrow F^2$	Build descriptor # 2
4.	$T^1 * F^2 \rightarrow T^3$	Generate `MOVER AREG, A` `MULT AREG, B` Build descriptor # 3
5.	$T^3 \rightarrow E^3$	–
6.	$<id>_c \rightarrow F^4$	Build descriptor # 4
7.	$F^4 \rightarrow T^4$	–
8.	$<id>_d \rightarrow F^5$	Build descriptor # 5

9.	$T^4 * F^5 \rightarrow T^6$	Generate `MOVEM AREG, TEMP_1`
		`MOVER AREG, C`
		`MULT AREG, D`
		Build descriptor # 6

10.	$E^3 + T^6 \rightarrow E^7$	Generate `ADD AREG, TEMP_1`

Table 6.4

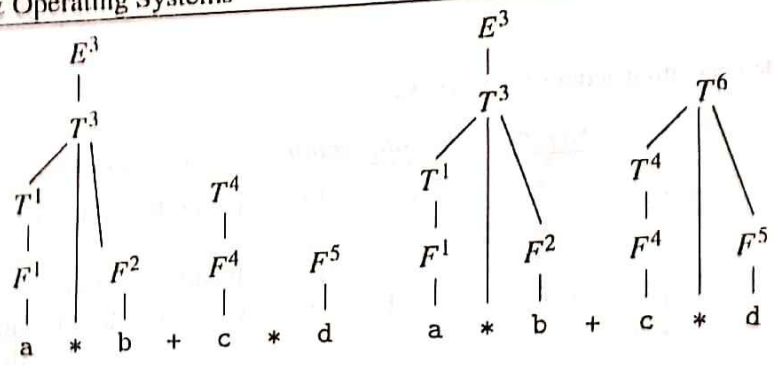
Temporary	*Contents*	*Used in*
temp [1]	Value of node 1 (i.e. value of `a*b`)	evaluating node 5
temp [2]	Value of node 2	evaluating node 4
temp [3]	Value of node 5	evaluating node 7

i.e. a stack, to organize the temporary locations. Use of a stack permits the reuse of temporary locations during the evaluation of an expression, thus reducing the number of temporary locations required. Figure 6.17(b) shows the code for the expression resulting from the reuse of *temp* [1]. The contents of the temporary locations and their usage are summarized in Table 6.5.

Note that *temp* [1] is reused to store the value of node 5. This is realized as follows: When the value contained in *temp* [2] is used in evaluating node 4, *temp* [2] is marked free (see the 9^{th} instruction in Fig. 6.17(b)). Similarly *temp* [1] is marked free

Table 6.5

Temporary	*Contents*	*Used in*
temp [1]	Value of node 1	evaluating node 5
temp [2]	Value of node 2	evaluating node 4
temp [1]	Value of node 5	evaluating node 7



Operand descriptors

1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	R, addr(AREG)
4	(int,1)	M, addr(c)
5	(int,1)	M, addr(d)

1	(int,1)	M, addr(a)
2	(int,1)	M, addr(b)
3	(int,1)	M, addr(temp[1])
4	(int,1)	M, addr(c)
5	(int,1)	M, addr(d)
6	(int,1)	R, addr(AREG)

Register descriptor

Occ.	3

Occ.	6

(a) (b)

Fig. 6.15 Code generation actions

when the value contained in it is used in evaluating node 5 (see the 10^{th} instruction). While generating the code for node 6, the partial result contained in the register is saved in the first available location. This happens to be *temp* [1].

To implement a stack of temporaries, variable `i` of routine `codegen` is used as a stack pointer. Now, `i` needs to be decremented whenever an operand of an instruction is a temporary location. If dynamic memory allocation is used, it is best to locate the stack *after* the activation record for a block. Now we can increment/decrement TOS instead of variable `i`. Figure 6.18 illustrates this arrangement.

EXERCISE 6.3.1

1. Extend the toy code generator to incorporate the following:
   (a) Parentheses in an expression,
   (b) Non-commutative operators like '–' and '/'.
2. Extend the toy code generator to handle multiple registers in the CPU. Show various steps in the code generation for the expression `(a+b)/(c+d)` using 2 CPU registers.

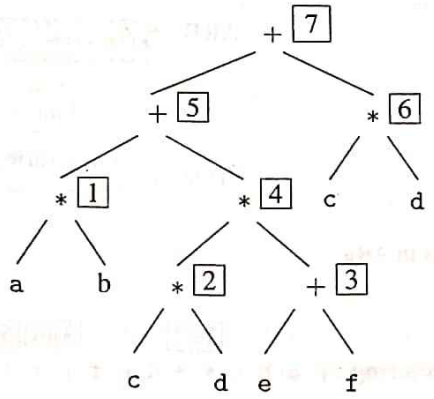


Fig. 6.16 Expression tree

```
MOVER   AREG, A          MOVER   AREG, A
MULT    AREG, B          MULT    AREG, B
MOVEM   AREG, TEMP_1     MOVEM   AREG, TEMP_1
MOVER   AREG, C          MOVER   AREG, C
MULT    AREG, D          MULT    AREG, D
MOVEM   AREG, TEMP_2     MOVEM   AREG, TEMP_2
MOVER   AREG, E          MOVER   AREG, E
ADD     AREG, F          ADD     AREG, F
MULT    AREG, TEMP_2     MULT    AREG, TEMP_2
ADD     AREG, TEMP_1     ADD     AREG, TEMP_1
MOVEM   AREG, TEMP_3     MOVEM   AREG, TEMP_1
MOVER   AREG, C          MOVER   AREG, C
MULT    AREG, D          MULT    AREG, D
ADD     AREG, TEMP_3     ADD     AREG, TEMP_1

        (a)                      (b)
```

Fig. 6.17 Illustration for temporary location usage

6.3.2 Intermediate Codes for Expressions

Postfix strings

In the postfix notation, each operator appears immediately after its last operand. Thus, a binary operator op_i appears after its second operand. If any of its operands is itself an expression involving an operator op_j, op_j must appear before op_i. Thus operators can be evaluated in the order in which they appear in the string.

Example 6.20 Consider the following strings:

$$\begin{array}{l} \boxed{2}\ \boxed{1}\ \boxed{5}\ \boxed{4}\ \boxed{3} \\ \text{Source string : } \vdash \texttt{a + b * c + d * e : f} \dashv \end{array} \tag{6.6}$$

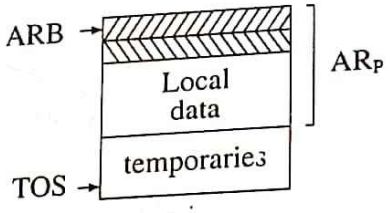


Fig. 6.18 Stack of temporaries in AR_P

Postfix string : ⊢ a b c * + d e f ↑ * + ⊣

The numbers appearing over the operators indicate their evaluation order. The second operand of the operator '+' marked [2] in the source string is the expression b*c. Hence its operands b, c and its operator '*' appear before '+' in the postfix string.

The postfix string is a popular intermediate code in non-optimizing compilers due to ease of generation and use. We perform code generation from the postfix string using a stack of operand descriptors. Operand descriptors are pushed on the stack as operands appear in the string. When an operator with arity k appears in the string, k descriptors are popped off the stack. A descriptor for the partial result generated by the operator is now pushed on the stack. Thus in Ex. 6.20, the operand stack would contain descriptors for a , b and c when '*' is encountered. The stack contains the descriptors for a and the partial result b*c when '+' is encountered.

Conversion from infix to postfix is performed using a slight modification of Algorithm 3.4. Each stack entry simply contains an operator. Operands appearing in the source string are copied into the postfix string straightaway. The TOS operator is popped into the postfix string if TOS operator $>$ current operator.

Triples and quadruples

A *triple* is a representation of an elementary operation in the form of a pseudo-machine instruction

Operator	*Operand 1*	*Operand 2*

Triples are numbered in some convenient manner. Each operand of a triple is either a variable/constant or the result of some evaluation represented by another triple. In the latter case, the operand field contains that triple's number. Conversion of an infix string into triples can be achieved by a slight modification of Algorithm 3.4.

Example 6.21 Figure 6.19 contains the triples for the expression string (6.6). [1] in the operand field of triple 2 indicates that the operand is the value of b*c represented by triple number 1.

	operator	operand 1	operand 2
1	*	b	c
2	+	[1]	a
3	↑	e	f
4	*	d	[3]
5	+	[2]	[4]

Fig. 6.19 Triples for string 6.6

A program representation called *indirect triples* is useful in optimizing compilers. In this representation, a table is built to contain all distinct triples in the program. A program statement is represented as a list of triple numbers. This arrangement is useful to detect the occurrences of identical expressions in a program. For efficiency reasons, a hash organization can be used for the table of triples. The indirect triples representation provides memory economy. It also aids in certain forms of optimization, viz. common subexpression elimination.

Example 6.22 Figure 6.20 shows the indirect triples' representation for the program segment

```
z := a+b*c+d*e↑f;
y := x+b*c;
```

Use of `b*c` in both statements is reflected by the fact that triple number 1 appears in the list of triples for both statements. As we shall see in Section 6.5.2, this fact is used to advantage in local common subexpression elimination.

	operator	operand 1	operand 2
1	*	b	c
2	+	[1]	a
3	↑	e	f
4	*	d	[3]
5	+	[2]	[4]
6	+	x	[1]

triples' table

stmt no.	triple nos.
1	1,2,3,4,5
2	1,6

statement table

Fig. 6.20 Indirect triples

A *quadruple* represents an elementary evaluation in the following format:

Operator	*Operand 1*	*Operand 2*	*Result name*

Here, *result name* designates the result of the evaluation. It can be used as the operand of another quadruple. This is more convenient than using a number (as in the case of triples) to designate a subexpression.

	operator	*operand 1*	*operand 2*	*result name*
1	*	b	c	t_1
2	+	t_1	a	t_2
3	↑	e	f	t_3
4	*	d	t_3	t_4
5	+	t_2	t_4	t_5

Fig. 6.21 Quadruples

Example 6.23 Quadruples for the expression string (6.6) are shown in Fig. 6.21.

Note that $t_1, t_2, ...t_5$ in Fig. 6.21 are not temporary locations for holding partial results. They are result names. Some of these become temporary locations when common subexpression elimination is implemented. For example, if an expression `x+b*c` were also present in the program, then `b*c` may be a common subexpression. t_1 would then become a compiler generated temporary location. We discuss details of optimization by elimination of common subexpressions in Section 6.5.2.

Expression trees

We have so far assumed that operators are evaluated in the order determined by a bottom up parser. This evaluation order may not lead to the most efficient code for an expression. Hence a compiler back end must analyse an expression to find the best evaluation order for its operators. An *expression tree* is an abstract syntax tree (see Section 3.2) which depicts the structure of an expression. This representation simplifies the analysis of an expression to determine the best evaluation order.

Example 6.24 Figure 6.22 shows two alternative codes to evaluate the expression `(a+b)/(c+d)`. The code in part (b) uses fewer MOVER/MOVEM instructions. It is obtained by deviating from the evaluation order determined by a bottom up parser.

```
MOVER   AREG, A
ADD     AREG, B
MOVEM   AREG, TEMP_1
MOVER   AREG, C
ADD     AREG, D
MOVEM   AREG, TEMP_2
MOVER   AREG, TEMP_1
DIV     AREG, TEMP_2
```

(a)

```
MOVER   AREG, C
ADD     AREG, D
MOVEM   AREG, TEMP_1
MOVER   AREG, A
ADD     AREG, B
DIV     AREG, TEMP_1
```

(b)

Fig. 6.22 Alternative codes for `(a+b)/(c+d)`

A two step procedure is used to determine the best evaluation order for the operations in an expression. The first step associates a *register requirement label* (RR

label) with each node in the expression. It indicates the number of CPU registers required to evaluate the subtree rooted at the node without moving a partial result to memory. Labelling is performed in a bottom up pass of the expression tree. The second step, which consists of a top down pass, analyses the RR labels of the child nodes of a node to determine the order in which they should be evaluated.

Algorithm 6.1 (Evaluation order for operators)

1. Visit all nodes in an expression tree in post order (i.e., such that a node is visited *after* all its children).

   For each node n_i

   (a) If n_i is a leaf node then
   if n_i is the left operand of its parent then RR(n_i) := 1;
   else RR(n_i) := 0;

   (b) If n_i is not a leaf node then
   If RR($l\_child_{n_i}$) $\neq$ RR($r\_child_{n_i}$) then
   RR(n_i) := max (RR($r\_child_{n_i}$), RR($l\_child_{n_i}$));
   else RR(n_i) := RR($l\_child_{n_i}$) + 1;

2. Perform the procedure call *evaluation_order* (*root*) (See Fig. 6.23), which prints a postfix form of the source string in which operators appear in the desired evaluation order.

```
procedure evaluation_order (node);
  if node is not a leaf node then
    if RR(l_child_node) ≤ RR(r_child_node) then
      evaluation_order (r_child_node);
      evaluation_order (l_child_node);
    else
      evaluation_order (l_child_node);
      evaluation_order (r_child_node);
  print node;
end evaluation_order;
```

Fig. 6.23 Procedure *evaluation_order*

Let RR=q for the root node. This implies that the evaluation order can evaluate the expression without moving any partial result(s) to memory if q CPU registers are available. It thus provides the most efficient way to evaluate the expression. When the number of available registers $< q$, some partial results have to be saved in memory. However, the evaluation order still leads to the most efficient code. The code generation algorithm is described in Dhamdhere (1997) and Aho, Sethi, Ullman (1986).

Example 6.25 Figure 6.24 shows the expression tree for the string `f+(x+y)*((a+b)/(c-d))`. The boxed numbers indicate operator positions in

the source string. The RR label of each node is shown as the superscript of the operand or operator at that node. The evaluation order according to algorithm 6.1 is [6], [4], [5], [2], [3], [1]. This evaluation order is determined as follows: When *evaluation_order* is called with the root node, i.e. operator [1], as the parameter, it decides that the right child of the node should be evaluated before its left child since RR([3]) > RR(node for `f`). This leads to a recursive call with node [3] as the parameter. Here RR([5]) > RR([2]) leads to a recursive call with node [5] as the parameter. At node [5] also decision is made to visit its right child before its left child. This leads to the postfix string `cd-ab+/xy+*f+` in which operators appear in the evaluation order mentioned above.

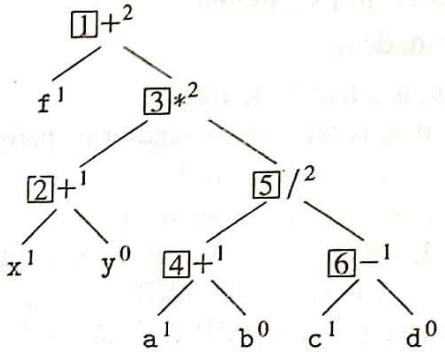


Fig. 6.24 Register requirement labels

EXERCISE 6.3.2

1. Modify the code generator of Fig. 6.13 to
   (a) Generate a postfix string as an IR,
   (b) Generate an expression tree as an IR.

6.4 COMPILATION OF CONTROL STRUCTURES

Definition 6.3 (Control structure) *The* control structure *of a programming language is the collection of language features which govern the sequencing of control through a program.*

The control structure of a PL consists of constructs for control transfer, conditional execution, iteration control and procedure calls. In this section we outline the fundamentals of control structure compilation and describe the compilation of procedure calls in some detail.

Control transfer, conditional execution and iterative constructs

Control transfers implemented through conditional and unconditional **goto**'s are the most primitive control structure. When the target language of a compiler is a machine language, compilation of control transfers is analogous to the assembly of forward or backward **goto**'s in an assembly language program. Hence similar techniques

based around the use of a label table can be used. When the target language is an assembly language (as is mostly the case in practice), a statement **goto lab** can be simply compiled as the assembly statement `BC ANY, LAB`. In the following, the target language of a compiler is assumed to be an assembly language.

Control structures like **if**, **for** or **while** cause significant semantic gap between the PL domain and the execution domain because the control transfers are implicit rather than explicit. This semantic gap is bridged in two steps. In the first step a control structure is mapped into an equivalent program containing explicit **goto**'s. Since the destination of a **goto** may not have a label in the source program, the compiler generates its own labels and puts them against the appropriate statements. Figure 6.25 illustrates programs equivalent to the **if** and **while** statements wherein the labels int_1, int_2 are introduced by the compiler for its own purposes. In the second step, these programs are translated into assembly programs. Note that the first step need not be carried out explicitly. It can be implied in the compilation action, as seen in Ex. 6.26.

```
if (e1) then                    if (¬e1) then goto int1;
   S1;                             S1;
else                      ⇒     goto int2;
   S2;                       int1: S2;
S3;                          int2: S3;
```

(a)

```
while (e2) do                int3: if (¬e2) then goto int4;
   S1;                             S1;
   S2;                       ⇒     S2;
   ⋮                               ⋮
   Sn;                             Sn;
endwhile;                          goto int3;
<next statement>             int4: <next statement>
```

(b)

Fig. 6.25 Control structure compilation

Example 6.26 Compilation of an **if** statement proceeds as follows (see Fig. 6.25): The compiler selects a pair of labels, viz. int_1 and int_2 in Fig. 6.25, to be associated with the **else** statement and the end of the **if** statement, respectively. Code is generated to evaluate the expression $\overline{e}_1$. This is followed by an instruction `BC ..,` int_1 which transfers control to int_1 if $\overline{e}_1$ is *true*. Code is now generated for the statements in the **then** part, followed by an instruction `BC ANY,` int_2. int_1 is now associated with the first statement of the **else** clause. int_2 is similarly associated with the statement following the **if** statement. The resulting code is:

```
if (e1) then                { instructions for ē1 }
                            BC  ..., int1  {Branch if true}
   S1;                      { instructions for S1 }
                    ⇒       BC  ANY, int2
else
   S2;                  int1: { instructions for S2 }
S3;                     int2: { instructions for S3 }
```

6.4.1 Function and Procedure Calls

A function call, viz. the call on fn_1 in the statement

```
x := fn_1(y,z)+b*c;
```

executes the body of `fn_1`, and returns its value to the calling program. In addition, the function call may also result in some side effects.

Definition 6.4 (Side effect) *A* side effect *of a function (procedure) call is a change in the value of a variable which is not local to the called function (procedure).*

A procedure call only achieves side effects, it does not return a value. Since all considerations excepting the return of a value are analogous for function and procedure calls, in the following only compilation of function calls are discussed.

Example 6.27 A side effect of the call `fn_1(y,z)` could be a change in the value of an actual parameter, viz. `y` or `z`, or of a nonlocal variable of `fn_1`.

While implementing a function call, the compiler must ensure the following:

1. Actual parameters are accessible in the called function.
2. The called function is able to produce side effects according to the rules of the PL.
3. Control is transferred to, and is returned from, the called function.
4. The function value is returned to the calling program.
5. All other aspects of execution of the calling programs are unaffected by the function call.

The compiler uses a set of features to implement function calls. These are described below.

1. *Parameter list:* The parameter list contains a *descriptor* for each actual parameter of the function call. The notation D_p is used to represent the descriptor corresponding to formal parameter p.
2. *Save area:* The called function saves the contents of CPU registers in this area before beginning its execution. The register contents are restored from this area before returning from the function.

3. *Calling conventions:* These are execution time assumptions shared by the called function and its caller(s). The conventions include the following:
   (a) How the parameter list is accessed
   (b) How the save area is accessed
   (c) How the transfers of control at call and return are implemented
   (d) How the function value is returned to the calling program.

   Most machine architectures provide special instructions to implement items (c) and (d).

Calling conventions

In a static memory allocation environment, the parameter list and the save area are allocated in the calling program. The calling conventions require the addresses of the function, the parameter list and the save area to be contained in specific CPU registers at the time of call. Let r_{par_list} denote the register containing the address of the parameter list and let $(d_{D_p})_{par\_list}$ denote the displacement of D_p in the parameter list. $(d_{D_p})_{par\_list}$ is computed while processing the formal parameter list of the function and is stored in the symbol table entry of p. During execution, D_p has the address $< r_{par\_list} > + (d_{D_p})_{par\_list}$. Hence every reference to p in the function body is compiled using this address. At return, the function value may be returned in a CPU register or in a memory location.

In a dynamic memory allocation environment, the calling program constructs the parameter list and the save area on the stack. These become a part of the called function's AR when its execution is initiated. Start of the parameter list has a known displacement d_{par_list} in the AR. For every formal parameter p, the displacement of D_p in the AR, denoted as $(d_{D_p})_{AR}$, is computed and stored in the symbol table entry of p. During execution, D_p has the address <ARB> + $(d_{D_p})_{AR}$. Ex. 6.28 illustrates calling conventions in the static and dynamic allocation environments.

Example 6.28 Figure 6.26 illustrates the calling conventions used in IBM mainframe systems to implement a call on function `fn_1` with the formal parameters `a` and `b`. The use of registers can be summarized as follows:

Register no.	Purpose
0	Function value
1	Address of parameter list
13	Address of save area
14	Return address
15	Address of function

Figure 6.27 illustrates a simplified version of the calling conventions used in a dynamic memory allocation environment. The calling program constructs the parameter list on the stack before invoking the function. At call, ARB is set to point at a location prior to the parameter list. Thus the parameter list is accessible as part of the function's AR. Register contents are now saved in the save area. Before returning, the function

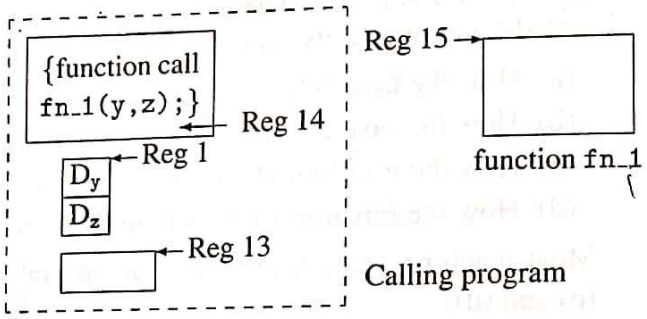


Fig. 6.26 Calling conventions in static memory allocation

value is stored in the AR. At return, TOS is set to <ARB>+2. Thus, the function value is available at the top of the run-time stack. (The stack locations allocated to the static and dynamic pointers would be wasted on return from the function. To avoid this, a practical compiler would put these pointers *after* the function value in the AR format.)

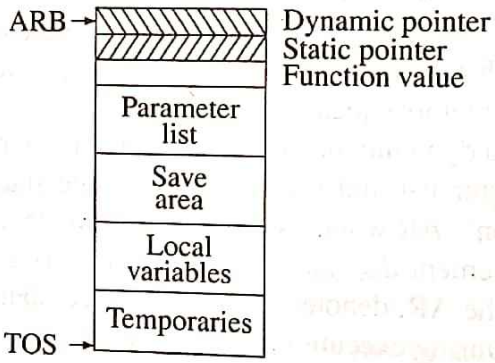


Fig. 6.27 Calling conventions in dynamic memory allocation

Parameter passing mechanisms

Language rules for parameter passing define the semantics of parameter usage inside a function, thereby defining the kind of side effects a function can produce on its actual parameters. The side effect characteristics and the execution efficiency of common parameter passing mechanisms are briefly described.

Call by value

In this mechanism, values of actual parameters are passed to the called function. These values are assigned to the corresponding formal parameters. Note that the passing of values only takes place in one direction—from the calling program to the called function. If a function changes the value of a formal parameter, the change is not reflected on the corresponding actual parameter. Thus a function cannot produce

any side effects on its parameters.

Call by value is commonly used for built-in functions of the language. Its main advantage is its simplicity. A called function may allocate memory to a formal parameter and copy the value of the actual parameter into this location at every call. Thus a formal parameter appears like an initialized local variable, hence the compiler can treat it as if it were a local variable. This simplifies compilation considerably. This mechanism is very efficient if parameters are scalar variables.

Call by value-result

This mechanism extends the capabilities of the call by value mechanism by copying the values of formal parameters back into corresponding actual parameters at return. Thus, side effects are realized *at return*. This mechanism inherits the simplicity of the call by value mechanism but incurs higher overheads.

Call by reference

In this mechanism, the address of an actual parameter is passed to the called function. If the parameter is an expression, its value is computed and stored in a temporary location and the address of the temporary location is passed to the called function. If the parameter is an array element, its address is similarly computed at the time of call.

The parameter list is thus a list of addresses of actual parameters. At every access of a formal parameter in the function, the address of the corresponding actual parameter is obtained from the parameter list. This is used as the address of the formal parameter. The code generated to access the value of formal parameter p is

1. $r \leftarrow$ <ARB> + $(d_{D_p})_{AR}$ or $r \leftarrow < r_{par\_list} > + (d_{D_p})_{par\_list}$.
2. Access the value using the address contained in register r.

Thus each access to a parameter in the function body incurs the overheads of step 1. Step 2 produces instantaneous side effects if the address in r is used on the left hand side of an assignment. This mechanism is very popular because it has 'cleaner' semantics than call by value-result.

Example 6.29 Consider a function

```
function alpha (a,b : integer) : integer;
   z := a;
   i := i+1;
   b := a+5;
return;
end alpha;
```

where `z`, `i` are nonlocal variables of `alpha`. Let the parameters be passed by reference and let `alpha` be called as `alpha(d[i],x)`. The address of `d[i]` is evaluated at the time of call and put into the parameter list. This address is used as the address of `a` in `alpha`. Note that a change in the value of `i` in the body of `alpha` does not

change this address. The assignment `b := ..` changes the value of `x` instantaneously. This is important in case a nested function call accesses `x` as a nonlocal variable. (If parameters are passed by value-result, the side effect on `x` would be realized at return. Thus a nested function call would access the old value of `x` rather than its new value assigned in function `alpha`.)

Call by name

This parameter transmission mechanism has the same effect as if every occurrence of a formal parameter in the body of the called function is replaced by the name of the corresponding actual parameter.

Example 6.30 In the program of Ex. 6.29, name substitution implies replacement of `a` by `d[i]` in the body of `alpha` during the execution of `alpha`. Thus the call `alpha (d[i],x)` has the same effect as execution of the statements

```
z := d[i];
i := i+1;
x := d[i]+5;
```

Apart from achieving instantaneous side effects, call by name has the implication of changing its actual parameter corresponding to a formal parameter *during* the execution of a function, e.g. formal parameter `a` in the above code corresponds to two different elements of `d` at different times in the execution of `alpha`.

Call by name is implemented as follows: Each parameter descriptor in the parameter list is the address of a routine which computes the address of the corresponding actual parameter. The code generated for every access of formal parameter `p` in the function body is

1. $r \leftarrow <\text{ARB}> + (d_{D_p})_{AR}$ or $r \leftarrow < r_{par\_list} > + (d_{D_p})_{par\_list}$.
2. Call the function whose address is contained in r.
3. Use the address returned by the function to access `p`.

As seen in Ex. 6.30, the actual parameter corresponding to a formal parameter can change dynamically during the execution of a function. This makes the call by name mechanism immensely powerful. However, the high overheads of step 2 in the code make it less attractive in practice.

Most languages use call by value for built-in functions. C also uses it for programmer defined functions. Pascal permits a programmer to choose between call by value and call by reference. Fortran and PL/I use call by reference only. Ada leaves the choice between call by value-result and call by reference to the compiler writer. Algol-60 used call by value and call by name.

EXERCISE 6.4

1. Many block structured languages permit nonlocal `goto`'s. Comment on the compilation of such **goto**'s. (Hint: Think of memory deallocation.)

2. Most PLs forbid control transfers into the iteration control constructs, e.g. goto's to statements situated in **for** loops. Devise a scheme to detect such control structure violations.
3. Compare and contrast the following parameter passing mechanisms in terms of execution efficiency and power to produce side effects.
   (a) call by value-result
   (b) call by reference
   (c) call by name.
4. What are the results produced by the following program for the different parameter passing mechanisms discussed in this section?

```
a : array[1..20] of real;
k := 7;
i := 11;
a[11] := 0.0;
a[12] := -0.5;
sub_1(i,k,a[i],i+k);
print k,i,a[i],a[k];
```

```
procedure sub_1(x,y,z,w);
x, y : integer;
y := x;
x := x + 1;
z := w;
return;
end;
```

5. A compiler uses a stack to compile nested control structures in a program. Each entry in the stack indicates information about one control structure. This information is used for error detection and code generation. An entry for a control structure is pushed/popped when the start/end of the control structure is encountered during compilation.
   Design a scheme to compile **if** and **for** statements using this approach. (Hint: see Ex. 6.26.)
6. Study the **case** control structure of Pascal and its generated code. Comment on how the code differs from nested **if** statements. Design a code generation model for the **case** statement.

6.5 CODE OPTIMIZATION

Code optimization aims at improving the execution efficiency of a program. This is achieved in two ways:

1. Redundancies in a program are eliminated.
2. Computations in a program are rearranged or rewritten to make it execute efficiently.

It is axiomatic that code optimization must not change the meaning of a program. Two points concerning the scope of optimization should also be noted. First, optimization seeks to improve a program rather than the algorithm used in a program. Thus replacement of an algorithm by a more efficient algorithm is beyond the scope of optimization. Second, efficient code generation for a specific target machine (e.g. by fully exploiting its instruction set) is also beyond its scope; it belongs in the back end of a compiler. The optimization techniques are thus independent of both the PL and the target machine.

Figure 6.28 contains a schematic of an optimizing compiler. It differs from the schematic in Fig. 1.12 in the presence of the optimization phase. The front end generates an IR which could consist of triples, quadruples or ASTs. The optimization phase transforms this to achieve optimization. The transformed IR is input to the back end.

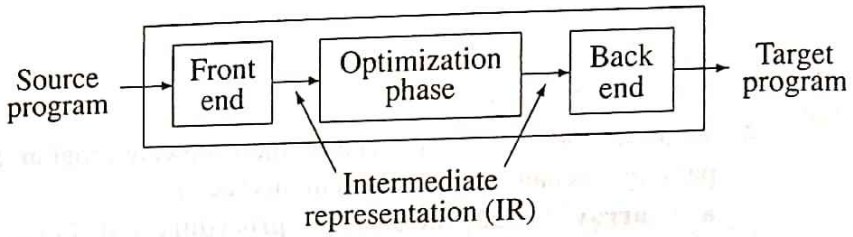


Fig. 6.28 Schematic of an optimising compiler

The structure of a program and the manner in which it manipulates its data provide vital clues for optimization. The compiler analyses a program to collect information concerning these aspects. The cost and benefits of optimization depend on how exhaustively a program is analysed. Experience with the Fortran 'H' optimizing compiler for the IBM/360 system (Lowry, Medlock, 1969) gives a quantitative feel for the cost and benefits of code optimization. The compiler was found to consume 40 percent extra compilation time due to optimization. The optimized program occupied 25 percent less storage and executed three times as fast as the unoptimized program.

6.5.1 Optimizing Transformations

An *optimizing transformation* is a rule for rewriting a segment of a program to improve its execution efficiency without affecting its meaning. Optimizing transformations are classified into *local* and *global* transformations depending on whether they are applied over small segments of a program consisting of a few source statements, or over larger segments consisting of loops or function bodies. The reason for this distinction is the difference in the costs and benefits of the optimizing transformations. A few optimizing transformations commonly used in compilers are discussed below.

Compile time evaluation

Execution efficiency can be improved by performing certain actions specified in a program during compilation itself. This eliminates the need to perform them during execution of the program, thereby reducing the execution time of the program. *Constant folding* is the main optimization of this kind. When all operands in an operation are constants, the operation can be performed at compilation time. The result of the operation, also a constant, can replace the original evaluation in the program. Thus, an assignment `a := 3.14157/2` can be replaced by `a := 1.570785`, thereby eliminating

a division operation. An instance of compile time evaluation can be found in array address arithmetic. The products $(\Pi_{j=1}^{i-1} range_j) \times k$ can be evaluated at compilation time (see Section 6.2.3) if $range_j \; \forall j$ and k are constants. This avoids evaluation of these products while executing every array reference.

Elimination of common subexpressions

Common subexpressions are occurrences of expressions yielding the same value. (Such expressions are called *equivalent expressions*.) Let CS_i designate a set of common subexpressions. It is possible to eliminate an occurrence $e_j \in CS_i$ if, no matter how the evaluation of e_j is reached during the execution of the program, the value of some $e_k \in CS_i$ would have been already computed. Provision is made to save this value and use it at the place of occurrence of e_j.

Example 6.31

```
                              t := b*c;
a := b*c                      a := t;
- - -                   ⇒    - - -
x := b*c+5.2;                 x := t+5.2;
```

Here CS_i contains the two occurrences of `b*c`. The second occurrence of `b*c` can be eliminated because the first occurrence of `b*c` is always evaluated before the second occurrence is reached during execution of the program. The value computed at the first occurrence is saved in `t`. This value is used in the assignment to `x`.

This optimization is implemented as follows: First, expressions which yield the same value are identified. Many compilers simplify this task by restricting its scope to congruent, i.e. identical, subexpressions. These can be easily identified using triples or quadruples (see indirect triples, Section 6.3.2). Their equivalence is determined by considering whether their operands have the same values in all occurrences. Occurrences of the subexpression which satisfy the criterion mentioned earlier for expressions e_j can be eliminated. Some compilers also use rules of algebraic equivalence in common subexpression elimination. In the following program:

```
..:= b*c ..
d := b;
..:= d*c..
```

`d*c` is a common subexpression since `d` has the same value as `b`. Use of algebraic equivalence improves the effectiveness of optimization. However, it also increases the cost of optimization.

Dead code elimination

Code which can be omitted from a program without affecting its results is called *dead code*. Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the program.

Example 6.32 An assignment `x := <exp>` constitutes dead code if the value assigned to `x` is not used in the program, no matter how control flows after executing this assignment. Note that `<exp>` constitutes dead code only if its execution does not produce side effects, i.e. only if it does not contain function or procedure calls.

Frequency reduction

Execution time of a program can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times. For example, the transformation of *loop optimization* moves loop invariant code out of a loop and places it prior to loop entry.

Example 6.33

```
for i := 1 to 100 do
begin
   z := i;
   x := 25*a;
   y := x+z;
end;
```

⇒

```
x := 25*a;
for i := 1 to 100 do
begin
   z := i;
   y := x+z;
end;
```

Here `x := 25*a;` is loop invariant. Hence in the optimized program it is computed only once before entering the **for** loop. `y := x+z;` is not loop invariant. Hence it cannot be subjected to frequency reduction.

Strength reduction

The strength reduction optimization replaces the occurrence of a time consuming operation (a 'high strength' operation) by an occurrence of a faster operation (a 'low strength' operation), e.g. replacement of a multiplication by an addition.

Example 6.34 In Fig. 6.29, the 'high strength' operator '*' in `i*5` occurring inside the loop is replaced by a low strength operator '+' in `itemp+5`.

```
for i := 1 to 10 do
begin
   - - -
   k := i*5;
   - - -
end;
```

⇒

```
itemp := 5;
for i := 1 to 10 do
begin
   - - -
   k := itemp;
   - - -
   itemp := itemp+5;
end;
```

Fig. 6.29 Strength reduction

Strength reduction is very important for array accesses occurring within program loops. For example, an array reference `a[i,j]` within a Pascal loop gives rise to the

high strength computation `i*n` in the address arithmetic, where n is the number of rows in the array (see Section 6.2.3). Strength of `i*n` can be reduced as in Fig. 6.29. Note that strength reduction optimization is not performed on operations involving floating point operands because finite precision of floating point arithmetic cannot guarantee equivalence of results after strength reduction.

Local and global optimization

Optimization of a program is structured into the following two phases:

1. *Local optimization:* The optimizing transformations are applied over small segments of a program consisting of a few statements,
2. *Global optimization:* The optimizing transformations are applied over a program unit, i.e. over a function or a procedure.

Local optimization is a preparatory phase for global optimization. It can be performed by the front end while converting a source program into the IR (see Fig. 6.36). Local optimization also simplifies certain aspects of global optimization. For example, let a program segment seg_i contain n occurrences of an expression `a+b`. After local common subexpression elimination has been performed over seg_i, global optimization only needs to consider elimination of the first occurrence of `a+b`—other occurrences of `a+b` are either not redundant, or would have been already eliminated!

6.5.2 Local Optimization

Local optimization provides limited benefits at a low cost. The scope of local optimization is a *basic block* which is an 'essentially sequential' segment in the source program. The cost of local optimization is low because the sequential nature of the basic block simplifies the analysis needed for optimization. The benefits are limited because certain optimizations, e.g. loop optimization, are beyond the scope of local optimization.

Definition 6.5 (Basic block) *A* basic block *is a sequence of program statements* $(s_1, s_2, \ldots, s_n)$ *such that only* s_n *can be a transfer of control statement and only* s_1 *can be the destination of a transfer of control statement.*

A basic block b is a program segment with a single entry point. If control reaches statement s_1 during program execution, all statements $s_1, s_2, \ldots, s_n$ will be executed. The 'essentially sequential' nature of a basic block simplifies optimization. Example 6.35 discusses this aspect. We shall also see this aspect while discussing the value numbering technique to perform local optimization.

Example 6.35 Consider the following program segment:

```
                    a := x*y;                    t := x*y;
                    - - -          ⇒             a := t;
                    b := x*y;                    - - -
labi :              c := x*y;          labi :    b := t;
                                                 c := x*y;
```

where lab_i is a label. Local optimization identifies two basic blocks in the program. The first block extends up to the statement `b := x*y;`. Its optimization leads to elimination of the second occurrence of `x*y`. The third occurrence is not eliminated because it belongs to a different basic block. If the label lab_i did not exist, the entire program segment would constitute a single basic block and the third occurrence `x*y` can also be eliminated during local optimization.

Value numbers

Value numbers provide a simple means to determine if two occurrences of an expression in a basic block are equivalent. The value numbering technique is applied on the fly while identifying basic blocks in a source program. A value number vn_{alpha} is associated with variable `alpha`. It identifies the last assignment to `alpha` processed so far. Thus, the value number of variable `alpha` changes on processing an assignment `alpha := ...`. Now two expressions e_i and e_j are equivalent if they are congruent and their operands have the same value numbers.

For simplicity all statements of a basic block are numbered in some convenient manner. If statement n, the current statement being processed, is an assignment to `alpha`, we set vn_{alpha} to n. A new field is added to each symbol table entry to hold the value number of a variable. The IC for a basic block is a list of quadruples stored in a tabular form. Each operand field in a quadruple holds the pair (*operand, value number*). A boolean flag *save* is associated with each quadruple to indicate whether its value should be saved for use elsewhere in the program. The flag is initialized to *false* in every new quadruple entered in the table.

Let expression e be a subexpression of e', the expression being compiled. While forming a quadruple for e, the value numbers of its operands are copied from the symbol table. The new quadruple is now compared with all existing quadruples in the IC. Existence of a matching quadruple q_i in the IC implies that the current occurrence of expression e has the same value as a previous occurrence represented by quadruple q_i. If a match is found, the newly generated quadruple is not entered in the IC. Instead, the result name of q_i is used as an operand of e'. In effect, this occurrence of e is eliminated from the program. The result name of q_i should now become a compiler generated temporary variable. This requirement is noted by setting the *save* flag of q_i to *true*. During code generation, this flag is checked to see if the value of q_i needs to be saved in a temporary location.

Example 6.36 Figure 6.30 shows the symbol table and the quadruples table during local optimization of the following program:

Symbol table

Symbol	...	*Value number*
y		0
x		15
g		14
z		0
d		5
w		0

Quadruples table

	Operator	*Operand 1*		*Operand 2*		*Result name*	*Use flag*
		Operand	*Value no.*	*Operand*	*Value no.*		
20	:=	g	–	25.2	–	t_{20}	*f*
21	+	z	0	2	–	t_{21}	*f*
22	:=	x	0	t_{21}	–	t_{22}	*f*
23	*	x	15	y	0	t_{23}	*ft*
24	+	t_{23}	–	d	5	t_{24}	*f*
	⋮						
57	:=	w	0	t_{23}	–	t_{57}	*f*

Fig. 6.30 Local optimization using value numbering

stmt no.	*statement*
14	`g := 25.2;`
15	`x := z+2;`
16	`h := x*y+d;`
..	...
34	`w := x*y;`

Local optimization proceeds as follows: All variables are assumed to have the value numbers '0' to start with. Processing of Statements 14 and 15 leads to generation of quadruples numbered 20–22 shown in the table. Value numbers of g, x and y at this stage are 14, 15 and 0, respectively (see the symbol table). Quadruple for `x*y` is generated next, and the value numbers of x and y are copied from the symbol table. This quadruple is assigned the result name t_{23}, and its *save* flag is set to *false*. This quadruple is entered in entry number 23 of the quadruple table. When the statement `w := x*y` (i.e., Statement 34) is processed, the quadruple for `x*y` is formed using the value numbers found in the symbol table entries of x and y. Since these are still 15 and 0, the new quadruple is identical with quadruple 23 in the table. Hence it is not entered in the table. Instead, the *save* flag of quadruple 23 is set to *true*. The only quadruple generated for this statement is therefore the assignment to w (quadruple number 57). While generating code for quadruple 23, its save flag indicates that the

value of expression x*y needs to be saved in a temporary location for later use. t_{23} can itself become the name of this temporary location.

This schematic can be easily extended to implement *constant propagation*, which is the substitution of a variable *var* occurring in a statement by a constant *const*, and constant folding. When an assignment of the form *var* := *const* is encountered, we enter *const* into a table of constants, say in entry *n*, and associate the value number '$-n$' with *var*. Constant propagation and folding is implemented while generating a quadruple if each operand is either a constant or has a negative value number.

Example 6.37 In the following program, variable a is given a negative value number, say '−10', on processing the first statement.

```
a := 27.3;          a := 27.3;
- - -        ⇒      - - -
b := a*3.0;         b := 81.9;
```

This leads to the possibility of constant propagation and folding in the third statement. The value of a, viz. '27.3', is obtained from the 10^{th} entry of constants table. Its multiplication with 3.0 yields 81.9. This value is treated as a new constant and a quadruple for `b := 81.9` is now generated.

6.5.3 Global Optimization

Compared to local optimization, global optimization requires more analysis effort to establish the feasibility of an optimization. Consider global common subexpression elimination. If some expression x*y occurs in a set of basic blocks SB of program P, its occurrence in a block $b_j \in$ SB can be eliminated if the following two conditions are satisfied for every execution of P:

1. Basic block b_j is executed only after some block $b_k \in$ SB has been executed one or more times.
2. No assignments to x or y have been executed after the last (or only) evaluation of x*y in block b_k. (6.7)

Condition 1 ensures that x*y is evaluated before execution reaches block b_j, while condition 2 ensures that the evaluated value is equivalent to the value of x*y in block b_j. The optimization is realized by saving the value of x*y in a temporary location in all blocks b_k which satisfy condition 1.

To ensure that *every possible execution* of program P satisfies conditions 1 and 2 of (6.7), the program is analysed using the techniques of *control flow analysis* and *data flow analysis*. Note the emphasis on the words 'every possible execution'. This requirement is introduced to ensure that the meaning of the program is unaffected by the optimization. In this section we use the word 'always' to imply 'in every possible evaluation'.

6.5.3.1 Program Representation

A program is represented in the form of a *program flow graph.*

Definition 6.6 (Program flow graph (PFG))

A program flow graph *for a program P is a directed graph* $G_P = (N, E, n_0)$ *where*

N	:	*set of basic blocks in P*
E	:	*set of directed edges* (b_i, b_j) *indicating the possibility of control flow from the last statement of* b_i *(the* source node*) to the first statement of* b_j *(the* destination node*)*
n_0	:	*start node of P.*

A basic block, which is a sequence of statements $s_1, s_2, \ldots s_n$, is visualized as a sequence of *program points* $p_1, p_2, \ldots p_n$ such that a statement s_i is said to exist at program point p_i. This notion is used to differentiate between different occurrences of identical statements. For example, an occurrence of e at program point p_i is distinct from the occurrence of e at program point p_j.

6.5.3.2 Control and Data Flow Analysis

The techniques of control and data flow analysis are together used to determine whether the Conditions governing an optimizing transformation are satisfied in a program, e.g. conditions 1 and 2 of (6.7).

Control flow analysis

Control flow analysis analyses a program to collect information concerning its structure, e.g. presence and nesting of loops in the program. Information concerning program structure is used to answer specific questions of interest, e.g. condition 1 of (6.7). The control flow concepts of interest are:

1. *Predecessors and successors:* If $(b_i, b_j) \in E$, b_i is a *predecessor* of b_j and b_j is a *successor* of b_i.
2. *Paths:* A path is a sequence of edges such that the destination node of one edge is the source node of the following edge.
3. *Ancestors and descendants:* If a path exists from b_i to b_j, b_i is an ancestor of b_j and b_j is a descendant of b_i.
4. *Dominators and post-dominators:* Block b_i is a dominator of block b_j if every path from n_0 to b_j passes through b_i. b_i is a post-dominator of b_j if every path from b_j to an exit node passes through b_i.

Table 6.6 Data flow concepts

Data flow concept	*Optimization in which used*
Available expression	Common subexpression elimination
Live variable	Dead code elimination
Reaching definition	Constant and variable propagation

Control flow concetps can be used to answer certain questions in a straightforward manner, e.g. the question posed by condition 1 of (6.7). However, this incurs the overheads of control flow analysis for every expression in the program. It may sometimes be possible to reduce the overheads by restricting the scope of optimization. For example, only those expressions may be considered which occur in a block b_j and a dominator block b_k of b_j. Now condition 1 of (6.7) is automatically satisfied.

Data flow analysis

Data flow analysis techniques analyse the use of data in a program to collect information for the purpose of optimization. This information, called *data flow information*, is computed at the entry and exit of each basic block in G_P. It is used to decide whether an optimizing transformation (see Section 6.5.1) can be applied to a segment of code in the program.

Design of the global optimization phase begins with the identification of an appropriate *data flow concept* to support the application of each optimizing transformation. The data flow information concerning a program entity—for example a variable or an expression—is now a boolean value indicating whether the data flow concept is applicable to that entity. Table 6.6 contains a summary of important data flow concepts used in optimization. The first two data flow concepts are discussed in detail.

Available expressions

The transformation of global common subexpression elimination can be defined as follows: Consider a subexpression `x*y` occurring at program point p_i in basic block b_i. This occurrence can be eliminated if

1. Conditions 1 and 2 of (6.7) are satisfied at entry to b_i.
2. No assignments to `x` or `y` precede the occurrence of `x*y` in b_i.

The data flow concept of *available expressions* is used to implement common subexpression elimination. An expression e is *available* at program point p_i if a value equivalent to its value is always computed before program execution reaches p_i. Thus, the concept of available expressions captures the essence of Conditions 1 and 2 of (6.7). The availability of an expression at entry or exit of basic block b_i is computed using the following rules:

1. Expression e is available at the exit of b_i if

   (i) b_i contains an evaluation of e which is not followed by assignments to any operands of e, or
   (ii) the value of e is available at the entry to b_i and b_i does not contain assignments to any operands of e. (6.8)

2. Expression e is available at entry to b_i if it is available at the exit of each predecessor of b_i in G_P.

Available expressions is termed a *forward* data flow concept because availability at the exit of a node determines availability at the entry of its successor(s). It is an *all paths* concept because availability at entry of a basic block requires availability at the exit of all predecessors.

We use the boolean variables $Avail\_in_i$ and $Avail\_out_i$ to represent the availability of expression e at entry and exit of basic block b_i, respectively. Further, we associate the following boolean properties with block b_i to summarize the effect of computations situated in it:

$Eval_i$: 'true' only if expression e is evaluated in b_i and none of its operands are modified following the evaluation

$Modify_i$: 'true' only if some operand of e is modified in b_i.

$Eval_i$ and $Modify_i$ are determined solely by the computations situated in b_i. Hence they are called local properties of block b_i. $Avail\_in_i$ and $Avail\_out_i$ are 'global' properties which are computed using the following equations (see Eq. (6.8)):

$$Avail\_in_i = \Pi_{b_j \in pred(b_i)} Avail\_out_j \quad (6.9)$$

$$Avail\_out_i = Eval_i + Avail\_in_i \,.\, \neg Modify_i \quad (6.10)$$

where $\Pi_{b_j \in pred(b_i)}$ is the boolean 'and' operation over all predecessors of b_i. This operation ensures that $Avail\_in_i$ is true only if $Avail\_out$ is true for all predecessors of b_i. Equations (6.9) and (6.10) are called *data flow equations*.

It is to be noted that every basic block in G_P has a pair of equations analogous to (6.9)–(6.10). Thus, we need to solve a system of simultaneous equations to obtain the values of $Avail\_in$ and $Avail\_out$ for all basic blocks in G_P. Data flow analysis is the process of solving these equations. Iterative data flow analysis is a simple method which assigns some initial values to $Avail\_in$ and $Avail\_out$, and iteratively recomputes them for all blocks according to Eqs. (6.9)–(6.10) until they converge onto consistent values.

The initial values are:

$$Avail\_in_i = \begin{cases} true & \text{if } b_i \in N - \{n_o\} \\ false & \text{if } b_i = n_o \end{cases}$$

$$Avail\_out_i = true \quad \forall b_i$$

After solving the system of equations (6.9)–(6.10) for all blocks, an evaluation of expression e can be eliminated from a block b_i if

1. $Avail\_in_i = true$, and
2. The evaluation of e in b_i is not preceded by an assignment to any of its operands.

Example 6.38 Available expression analysis for the PFG of Fig. 6.31 gives the following results:

a*b : *Avail_in* = *true* for blocks 2,5,6,7,8,9
Avail_out = *true* for blocks 1,2,5,6,7,8,9,10

x+y : *Avail_in* = *true* for blocks 6,7,8,9
Avail_out = *true* for blocks 5,6,7,8,9

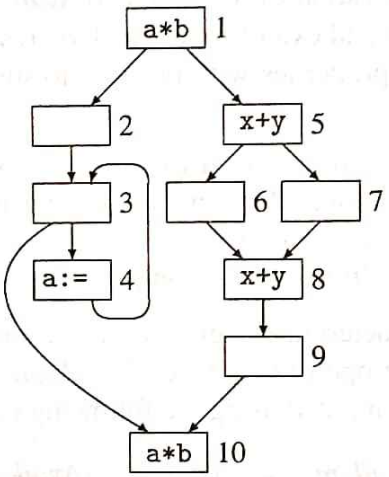


Fig. 6.31 A PFG

The values of *Avail_in* and *Avail_out* for a*b can be explained as follows: The assignment a := .. in block 4 makes $Avail\_out_4 = false$ (eq. (6.10)). This makes $Avail\_in_3 = false$ (eq. (6.9)), which makes $Avail\_out_3 = Avail\_in_{10} = false$. For x*y, $Avail\_in_1 = false$ makes $Avail\_out_1 = false$, which makes *Avail_in* = *Avail_out* = *false* for blocks 2, 3, 4 and 10.

Live variables

A variable *var* is said to be *live* at a program point p_i in basic block b_i if the value contained in it at p_i is likely to be used during subsequent execution of the program. If *var* is not live at the program point which contains a definition *var* := ..., the value assigned to *var* by this definition is redundant in the program. Such a definition constitutes dead code which can be eliminated from the program without changing its meaning.

The liveness property of a variable can be determined as follows:

1. Variable v is live at the entry of b_i if

   (i) b_i contains a use of e which is not preceded by assignment(s) to v, or
   (ii) v is live at the exit of b_i and b_i does not contain assignment(s) to v.

2. v is live at exit of b_i if it is live at the entry of some successor of b_i in G_P.

Data flow information concerning live variables can be collected as follows:

$Live\_in_i$	:	*var* is live at entry of b_i
$Live\_out_i$	:	*var* is live at exit of b_i
Ref_i	:	*var* is referenced in b_i and no assignment of *var* precedes the reference
Def_i	:	An assignment to *var* exists in b_i

$$Live\_in_i = Ref_i + Live\_out_i \,.\, \neg Def_i \tag{6.11}$$

$$Live\_out_i = \Sigma_{b_j \in succ(b_i)} Live\_in_j \tag{6.12}$$

where $\Sigma_{b_j \in succ(b_i)}$ is the boolean 'or' operation over all successors of b_i.

Live variables is termed a *backward* data flow concept because availability at the entry of a block determines availability at the exit of its predecessor(s). It is an *any path* concept because liveness at the entry of one successor is sufficient to ensure liveness at the exit of a block. The data flow problem can be solved by iterative data flow analysis using the initializations $Live\_in_i = Live\_out_i = false \; \forall b_i$.

Example 6.39 In the PFG of Fig. 6.31, variable b is live at the entry of all blocks, a is live at the entry of all blocks excepting block 4 and variables x, y are live at the entry of blocks 1,5,6,7 and 8.

EXERCISE 6.5

1. Write a program to generate code from a table of quadruples (see Fig. 6.30).
2. Build a program flow graph for the following program:

```
z := 5;
w := z;
for i := 1 to 100 do
    x := a*b;
    y := c+d;
    if y < 0 then
        a := 25;
        f := c+d;
    else
        g := w;
        h := a*b+f;
        d := z+10;
```

```
end;
g := c+d;
print g,h,d,x,y;
```

Apply the following transformations to optimize the program:

(a) common subexpression elimination
(b) dead code elimination
(c) constant propagation
(d) frequency reduction.

3. Apply the data flow concepts of available expressions and live variables to the flow graph of the program in problem 2 and verify whether you obtain the same answers as in parts (a) and (b) of problem 2.
4. A definition def_i is an assignment of the form *var* := <*exp*>. def_i located at program point p_i is said to reach a program point p_j if the value assigned to *var* by def_i may be the value of *var* when program execution reaches p_j.

   Write the data flow equations to collect the information concerning reaching definitions. (*Hint:* Consider whether the data flow concept is a forward or backward concept and any path or all paths concept.)
5. Constant propagation is the substitution of a variable *var* by a constant $const_i$ in a statement. State the conditions under which constant propagation can be performed. (*Hint:* See problem 4.)
6. Loop invariant movement of an assignment *var* := <*exp*> (see Ex. 6.33) is performed if the following conditions are met:

   (a) <*exp*> is loop invariant.
   (b) The assignment is the only assignment to *var* inside the loop.
   (c) The assignment dominates all loop exist.
   (d) *var* is not live on entry to the loop.

   Discuss the need for each condition.

6.6 INTERPRETERS

In Section 1.2.2 we have seen that use of interpretation avoids the overheads of compilation. This is an advantage during program development, because a program may be modified very often—in fact, it may be modified between every two executions. However, interpretation is expensive in terms of CPU time, because each statement is subjected to the interpretation cycle. Hence, conventional wisdom warns against interpreting a program with large execution requirements. However, to make an informed decision in practice, we need a quantitative basis for a comparison of compilers and interpreters. We introduce the following notation for this purpose:

t_c : average compilation time per statement
t_e : average execution time per statement
t_i : average interpretation time per statement

Note that both compilers and interpreters analyse a source statement to determine its meaning. During compilation, analysis of a statement is followed by code generation, while during interpretation it is followed by actions which implement its meaning. Hence we could assume $t_c \cong t_i$. t_e, which is the execution time of the compiler generated code for a statement, can be several times smaller than t_c. Let us assume $t_c = 20.t_e$.

Consider a program P. Let $size_P$ and $stmts\_executed_P$ represent the number of statements in P and the number of statements executed in some execution of P, respectively. We use these parameters to compute the CPU time required to execute a program using compilation or interpretation. Example 6.40 illustrates how this can be done.

Example 6.40 Let $size_P = 200$. For a specific set of data, let program P execute as follows: 20 statements are executed for initialization purposes. This is followed by 10 iterations of a loop containing 8 statements, followed by the execution of 20 statements for printing the results. Thus, $stmts\_executed_P = 20 + 10 \times 8 + 20 = 120$. Thus,

Total execution time using the compilation model
$$= 200.t_c + 120.t_e$$
$$\cong 206.t_c.$$
Total execution time using the interpretation model
$$= 120.t_i$$
$$\cong 120.t_c.$$

Clearly, interpretation is beneficial in this case.

Use of interpreters

Use of interpreters is motivated by two reasons—efficiency in certain environments and simplicity. The findings of Ex. 6.40 concerning efficiency can be generalized as follows: It is better to use interpretation for a program P if P is modified between executions, and $stmts\_executed_P < size_P$. These conditions are satisfied during program development, hence interpretation should be preferred during program development. In all other situations, it is best to use compilation.

It is simpler to develop an interpreter than to develop a compiler because interpretation does not involve code generation (see Section 6.6.1). This simplicity makes interpretation more attractive in situations where programs or commands are not executed repeatedly. Hence interpretation is a popular choice for commands to an operating system or an editor. User interfaces of many software packages prefer interpretation for similar reasons.

6.6.1 Overview of Interpretation

In this section we discuss an interpretation schematic which implements the meaning of a statement without generating code for it. The interpreter consists of three main components:

1. *Symbol table:* The symbol table holds information concerning entities in the source program.
2. *Data store:* The data store contains values of the data items declared in the program being interpreted. The data store consists of a set of components $\{comp_i\}$. A component $comp_i$ is an array named $name_i$ containing elements of a distinct type $type_i$.
3. *Data manipulation routines:* A set of data manipulation routines exist. This set contains a routine for every legal data manipulation action in the source language.

On analysing a declaration statement, say a statement declaring an array `alpha` of type `typ`, the interpreter locates a component $comp_j$ of its data store, such that $type_j$ = `typ`. `alpha` is now mapped into a part of $name_j$ (this is 'memory allocation'). The memory mapping for `alpha` is remembered in its symbol table entry. An executable statement is analysed to identify the actions which constitute its meaning. For each action, the interpreter finds the appropriate data manipulation routine and invokes it with appropriate parameters. For example, the meaning of statement `a := b+c;` where `a, b, c` are of the same type can be implemented by executing the calls

```
add (b, c, result);
assign (a, result);
```

in the interpreter.

This schematic has two important advantages. First, the meaning of a source statement is implemented through execution of the interpreter routines rather than through code generation. This simplifies the interpreter. Second, avoiding generation of machine language instructions helps to make the interpreter portable. For example, if the interpreter is itself coded in a higher level programming language it can be ported to a new computer system with ease.

6.6.2 A Toy Interpreter

This section discusses the design and operation of an interpreter for Basic written in Pascal. The interpreter is itself a Pascal program (we will call this the *interpreter program*), which is compiled by a Pascal compiler. The data store of the interpreter consists of two large arrays named `rvar` and `ivar` which are used to store real and integer values respectively. Last few locations in the `rvar` and `ivar` arrays are used as stacks for expression evaluation with the help of the pointers `r_tos` and `i_tos` respectively. (Note that these stacks grow 'upwards' in the arrays.)

The interpreter program contains a set of data manipulation routines for use in expression evaluation (see Fig. 6.32)—one routine for each kind of legal subexpression in the language. A routine performs all subtasks involved in the evaluation of a subexpression, namely type compatibility checks, type conversion and evaluation of the subexpressions. Since the interpreter is a Pascal program, it can be ported to any computer system possessing a Pascal compiler.

```
program interpreter (source,output);
  type
    symentry = record
      symbol : array [1..10] of character;
      type : character;
    address : integer
    end;
  var
    symtab : array [1..100] of symentry;
    rvar : array [1..100] of real;
    ivar : array [1..100] of integer;
    r_tos : 1..100;
    i_tos : 1..100;
  procedure assignint (addr1 : integer; value : integer);
    begin
      ivar[addr1] := value;
    end;
  procedure add (sym1, sym2 : symentry);
    begin
    ...
    if (sym1.type = 'real' and sym2.type = 'int') then
      addrealint(sym1.address, sym2.address);
    ...
    end;
  procedure addrealint (addr1, addr2 : integer);
    begin
      rvar[r_tos] := rvar[addr1] + ivar[addr2]
    end;
  begin { Main Program }
    r_tos := 100;
    i_tos := 100;
    {Analyse a statement and call
     appropriate procedure}
  end.
```

Fig. 6.32 Basic interpreter written in Pascal

Example 6.41 Consider the basic program

```
real a, b
integer c
let c = 7
let b = 1.2
a = b+c
```

Figure 6.33 illustrates the symbol table for the program and the memory allocation for variables a, b and c. Each symbol table entry contains information about the type and memory allocation for a variable. The values 'real' and '8' in the symbol table entry of a indicate that a is allocated the word rvar [8].

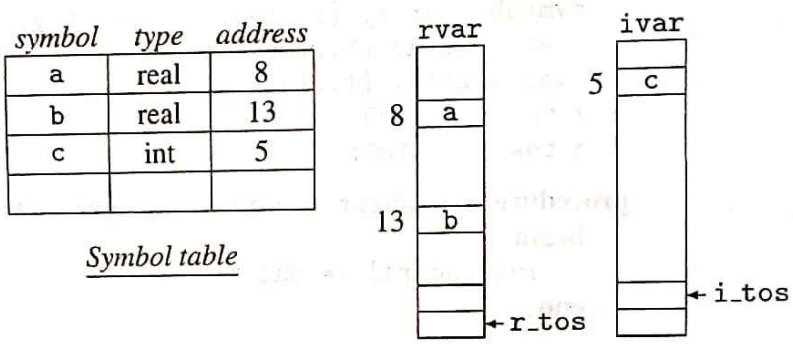


Fig. 6.33 Interpreter data structures

The operation of the interpreter can be explained as follows: The interpreter analyses a source statement to determine its meaning. It implements the meaning by invoking appropriate routines. Consider the assignment statement c = 7 in the Basic program, where both c and '7' are of type integer. The Pascal procedure `assignint` of Fig. 6.32 is invoked with two parameters. The first parameter is the address of c in `ivar` and the second parameter is the RHS value. Execution of procedure `assignint` effectively executes the Pascal statement

```
ivar [5] := 7;
```

Similarly, the assignment to b is executed as `rvar [13] := 1.2;` by a routine `assignreal`.

Interpretation of `a = b+c` proceeds as follows: The interpreter procedure `add` is called with the symbol table entries of b and c. `add` analyses the types of b and c and decides that procedure `addrealint` will have to be called to realize the addition. `addrealint` now executes a single Pascal statement which is equivalent to

```
rvar [r_tos] := rvar [13] + ivar [5];
```

Note that b+c involves a type conversion operation, which is performed implicitly by the above Pascal statement. In other words, while compiling the expression

```
rvar [addr1] + ivar [addr2]
```

in the interpreter program, the Pascal compiler would have made provision for type conversion of the second operand from integer to real. The interpreter simply arranges to execute this statement under the right conditions.

6.6.3 Pure and Impure Interpreters

The schematic of Fig. 6.34(a) is called a *pure* interpreter. The source program is retained in the source form all through its interpretation. This arrangement incurs substantial analysis overheads while interpreting a statement.

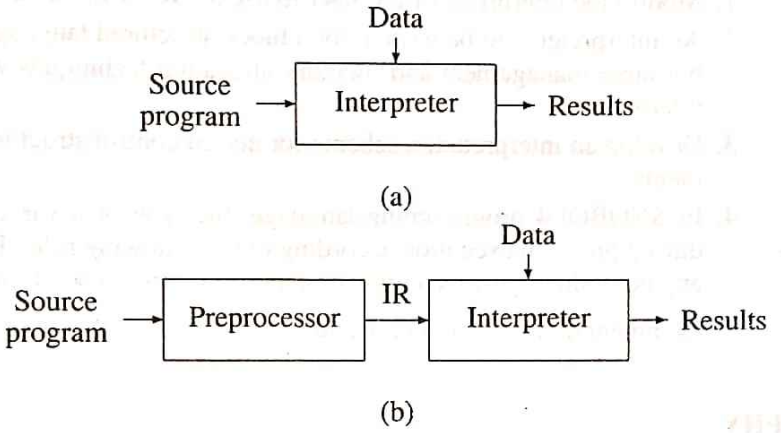


Fig. 6.34 Pure and impure interpreter

An *impure interpreter* performs some preliminary processing of the source program to reduce the analysis overheads during interpretation. Figure 6.34(b) contains a schematic of impure interpretation. The preprocessor converts the program to an intermediate representation (IR) which is used during interpretation. This speeds up interpretation as the code component of the IR, i.e. the IC, can be analysed more efficiently than the source form of the program. However, use of IR also implies that the entire program has to be preprocessed after any modification. This involves fixed overheads at the start of interpretation.

Example 6.42 Postfix notation is a popular intermediate code for interpreters. The intermediate code for a source string `a+b*c` could look like the following:

S #17	S #4	S #29	*	+

where each IC unit resembles a token (see Section 1.3.1.1).

IC of Ex. 6.42 eliminates most of the analysis during interpretation excepting type analysis to determine the need for type conversion. Even this can be eliminated if the preprocessor performs type analysis before generating IC.

Example 6.43 The preprocessor of an interpreter performs type analysis to generate following IC for the expression `a+b*c`, where `a`, `b` are of type real and `c` is of type integer

S #17	S #4	S #29	$t_{i \to r}$	$*_r$	$+_r$

where the unary operator $t_{i \to r}$ indicates type conversion of an operand (in this case, c) from 'integer' to 'real'. The arithmetic operators are also type specific now.Thus '$*_r$' indicates multiplication in the 'real' representation. This eliminates most analysis during interpretation.

EXERCISE 6.6

1. Modify the interpreter of Ex. 6.41 to use the IC of Ex. 6.43.
2. An interpreter is to be written for a block structured language. Comment on the symbol table management and memory allocation techniques which can be used by the interpreter.
3. Develop an interpretation scheme for nested control structures, e.g. nested `for` statements.
4. In SNOBOL4 programming language, the type of a variable changes dynamically during program execution according to the following rule: The type of a variable x at any point during the execution of a program is the type of the last value assigned to it.

   Comment on the compilation and interpretation of this feature of SNOBOL4.

BIBLIOGRAPHY

Elson (1973), Tennent (1981), Pratt (1983) and Maclennan (1983) have discussed different aspects of the design of programming languages and their influence on programming ease, compilation strategies and execution efficiencies. It is recommended that the reader should acquaint himself with the material covered in one of these texts before reading specialized material.

Gries (1971), Weingarten (1973) and Bauer and Eickel (1974) are important early books on compilers. Later books include those by Lewis, Rosenkrantz and Stearns (1976), Barrett and Couch (1977), Dhamdhere (1997), Tremblay and Sorenson (1984), Aho, Sethi and Ullman (1986), Fischer and LeBlanc (1988), and Watson (1989).

Good discussions on storage allocation can be found in books by Gries (1971) and Dhamdhere (1997). Expression compilation is discussed by Hopgood (1967), Rohl (1975), Dhamdhere (1997) and Aho, Sethi and Ullman (1986). Compilation of control structures and code optimization is discussed by Dhamdhere (1997) and Aho, Sethi and Ullman (1986).

(a) Programming languages

1. Elson, M. (1973): *Concepts of Programming Languages*, Science Research Associates, Chicago.
2. Maclennan, B.J. (1983): *Principles of Programming Languages*, Holt, Rinehart & Winston, New York.
3. Pratt, T.W. (1983): *Programming Languages – Design and Implementation*, Prentice-Hall, Englewood Cliffs.
4. Tennent, R.D. (1981): *Principles of Programming Languages*, Prentice-Hall, Englewood Cliffs.

(b) Computer architecture and compilers

1. Doran, R.W. (1979): *Computer Architecture – A Structured Approach*, Academic Press, London.
2. Organick, E.I. (1973): *Computer System Architecture*, Academic Press, London.
3. Wulf, W.A. (1981): "Compilers and computer architecture," *Computer*, **14** (7), 41-48.

(c) Compilers, general

1. Aho, A.V., R. Sethi and J.D. Ullman (1986) : *Compilers – Principles, Techniques and Tools*, Addison-Wesley, Reading.
2. Barrett, W.A. and J.D. Couch (1977): *Compiler Construction*, Science Research Associates, Pennsylvania.
3. Bauer, F.L. and J. Eickel (1974): *Compiler Construction : An Advanced Course*, Springer-Verlag, Berlin.
4. Calingaert, P. (1979): *Assemblers, Compilers and Program Translation*, Computer Science Press, Maryland.
5. Dhamdhere, D.M. (1997): *Compiler Construction – Principles and Practice*, 2^{nd} *edition*, Macmillan India, New Delhi.
6. Fischer, C.N. and R.J. LeBlanc (1988) : *Crafting a compiler*, Benjamin/ Cummings, Menlo Park, California.
7. Gries, D. (1971): *Compiler Construction for Digital Computers*, Wiley, New York.
8. Hansen, P.B. (1985): *Brinch Hansen on Pascal compilers*, Prentice-Hall, Englewood Cliffs, N.J.
9. Hill, U. (1974): "Special run time organization techniques for Algol-68," in *Compiler Construction : An advanced course*, Bauer, F.L., Eickel, J. (eds), Springer-Verlag, Berlin.
10. Lewis, P.M., D.J. Rosenkrantz, and R.E. Stearns (1976): *Compiler Design Theory*, Addison-Wesley, Reading.
11. Peck, J.E.L. (ed.) (1971): *Algol-68 Implementation*, North Holland, Amsterdam.
12. Tremblay, J.P. and P.G. Sorenson (1984): *The Theory and Practice of Compiler Writing*, McGraw-Hill, New York.
13. Watson, D. (1989): *High Level Languages and their Compilers*, Addison-Wesley, Reading.
14. Weingarten, F.W. (1973): *Translation of Computer Languages*, Holden-Day, San Francisco.

(d) Interpreters and related topics

1. Ayres, R.B. and R.L. Derrenbacher (1971): "Partial recompilation," *Proceedings of AFIPS SJCC*, **38**, 497-502.
2. Berthaud, M. and M. Griffiths (1973): "Incremental compilation and conversational interpretation," *Annual Review in Automatic Programming*, **7** (2), 95-114.
3. Brown, P.J. (1979): *Writing Interactive Compilers and Interpreters*, Wiley, New York.
4. Early, J. and P. Caizergues (1972): "A method of incrementally compiling languages with nested statement structure," *Commn. of ACM*, **15** (12), 1040-1044.

5. Klint, P. (1981): "Interpretation techniques," *Software – Practice and Experience*, **11**, 963-973.
6. Kornerup, P., B.B. Kristensen and O.L. Madsen (1980): "Interpretation and code generation based on intermediate languages," *Software – Practice and Experience*, **10** (8), 636-658.
7. McIntyre, T.C. (1978): *Software Interpreters for Micro computers*, Wiley, New York.
8. Ryan, J.L., R.L. Crandall and M.C. Medwedeff (1966): "A conversational system for incremental compilation and execution in a time sharing environment," *Proceedings of AFIPS FJCC*, **29**, 1-22.