

---

```

    printf ("%d", i);
    ...
}
main ( )
{
    ...
    x = fun (i);
    ...
}

```

When function *f* is executed, names *i* and *a* in *fun* denote the same data object. Thus an assignment to *a* would cause the value of *i* printed by *fun* to differ from the value held at the point of call.

Aliasing can easily be achieved through pointers and array elements. For example, the following assignments in C

```

int x = 0;
int* i = &x;
int* j = &x;

```

would make *\*i*, *\*j*, and *x* aliases.

## 2.5 An abstract semantic processor

To describe the operational semantics of programming languages, we introduce a simple abstract processor, called SIMPLESEM, and we show how language constructs can be executed by sequences of operations of the abstract processor. In this section, we provide the main features of SIMPLESEM; additional details will be introduced incrementally, as additional language features are introduced.

In its basic form, SIMPLESEM consists of an *instruction pointer* (the reference to the instruction currently being executed), a *memory*, and a processor. The memory is where the instructions to be executed and the data to be manipulated are stored. For simplicity, we will assume that these two parts are stored into two separate memory sections: the *code memory* (C) and the *data memory* (D). Both C's and D's initial address is 0 (zero), and both programs and data are assumed to be stored from the initial address. The instruction pointer (ip) is always used to point to a location in C; it is initialized to 0.

We use the notation  $D[X]$  and  $C[X]$  to denote the values stored in the *X*-th cell of D and C, respectively. Thus *X* is an *l\_value* and  $D[X]$  is the corresponding *r\_value*. Modification of the value stored in a cell is performed by instruction

set, with two parameters: the address of the cell whose contents is to be set, and the expression evaluating the new value. For example, the effect on the data memory of instruction

set 10, D[20]  
is to assign the value stored at location 20 into location 10.

Input/output in SIMPLESEM is achieved quite simply by using the set instruction and referring to the special registers read and write, which provide for communication of the SIMPLESEM machine with the outside world. For example,

set 15, read  
means that the value read from the input device is to be stored at location 15;

set write, D[50]  
means that the value stored at location 50 is to be transferred to the output device.

We are quite liberal in the way we allow values to be combined in expressions; for example,  $D[15] + D[33] * D[41]$  would be an acceptable expression, and

set 99,  $D[15] + D[33] * D[41]$   
would be an acceptable instruction to modify the contents of location 99.

As we mentioned, ip is SIMPLESEM's instruction pointer, which is initialized to zero at each new execution and automatically updated as each instruction is executed. The machine, in fact, operates by executing the following steps repeatedly, until it encounters a special halt instruction:

1. Get the current instruction to be executed (i.e.,  $C[ip]$ );
2. Increment ip;
3. Execute the current instruction.

Notice, however, that certain programming language instructions might modify the normal sequential control flow, and this must be reflected by SIMPLESEM. In particular, we introduce the following two instructions: jump and jumpnt. The former represents an unconditional jump to a certain instruction. For example,

---

jump 47

forces the instruction stored at address 47 of C to be the next instruction to be executed; that is, it sets *ip* to 47. The latter represents a conditional jump, which occurs if an expression evaluates to true. For example, in:

jump 47, D[3] > D[8]

the jump occurs only if the value stored in cell 3 is greater than the value stored in cell 8.

SIMPLESEM allows *indirect addressing*. For example:

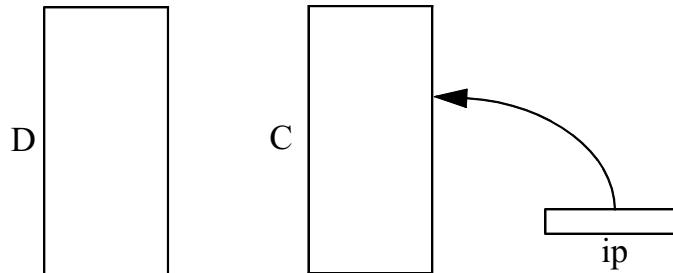
set D[10], D[20]

assigns the value stored at location 20 into the cell whose address is the value stored at location 10. Thus, if value 30 is stored at location 10, the instruction modifies the contents of location 30. Indirection is also possible for jumps. For example:

jump D[13]

jumps to the instruction stored at location 88 of C, if 88 is the value stored at location 13.

SIMPLESEM, which is sketched in Figure 11, is quite simple. It is easy to understand how it works and what the effects of executing its instructions are. In other terms, we can assume that its semantics is intuitively known; it does not require further explanations that refer to other, more basic concepts. The semantics of programming languages can therefore be described by rules that specify how each construct of the language is translated into a sequence of SIMPLESEM instructions. Since SIMPLESEM is perfectly understood, the semantics of newly defined constructs becomes also known. As we will see, however, SIMPLESEM will also be enriched as new programming language concepts are introduced. This will be done in this book incrementally, as we address the semantics of new concepts.



**FIGURE 11.**The SIMPLESEM machine

## 2.6 Execution-time structure

In this section we discuss how the most important concepts related to the execution-time processing of programming languages may be explained using SIMPLESEM. We will proceed gradually, from the most basic concepts to more complex structures that reflect what is provided by modern general-purpose programming languages. We will move through a hierarchy of languages that are based on variants of the C programming language. They are named C1 through C5.

Our discussion will show that languages can be classified in several categories, according to their execution-time structure.

### *Static languages.*

Exemplified by the early versions of FORTRAN and COBOL, these languages guarantee that the memory requirements for any program can be evaluated before program execution begins. Therefore, all the needed memory can be allocated before program execution. Clearly, these languages cannot allow recursion, because recursion would require an arbitrary number of unit instances, and thus memory requirements could not be determined before execution. (As we will see later, the implementation is not required to perform the memory allocation statically. The semantics of the language, however, give the implementer the freedom to make that choice.)

Section 2.6.1 and Section 2.6.2 will discuss languages C1, C2, and its variant C2', all of which fall under the category of static languages.

---

*Stack-based languages*

Historically headed by ALGOL 60 and exemplified by the family of so-called Algol-like languages, this class is more demanding in terms of memory requirements, which cannot be computed at compile time. However, their memory usage is predictable and follows a last-in-first-out discipline: the latest allocated activation record is the next one to be deallocated. It is therefore possible to manage SIMPLESEM's D store as a stack to model the execution-time behavior of this class of languages. Notice that an implementation of these languages need not use a stack (although, most likely, it will): deallocation of discarded activation records can be avoided if store can be viewed as unbounded. In other terms, the stack is part of the semantic model we provide for the language; strictly speaking, it is not part of the semantics of the language.

Section 2.6.3 and Section 2.6.4 discuss languages C3 and C4, which fall under the category of stack-based languages.

*Fully dynamic languages*

These languages have unpredictable memory usage; i.e., data are dynamically allocated only when they are needed during execution. The problem then becomes how to manage memory efficiently. In particular, how can unused memory be recognized and reallocated, if needed. To indicate that store D is not handled according to a predefined policy (like a FIFO policy for a stack memory), the term “heap” is traditionally used. This class of languages is illustrated by language C5 in Section 2.6.5.

### **2.6.1 C1: A language with only simple statements**

Let us consider a very simple programming language, called C1, which can be seen as a lexical variant of a subset of C, where we only have simple types and simple statements (there are no functions). Let us assume that the only data manipulated by the language are those whose memory requirements are known statically, such as integer and floating point values, fixed-size arrays, and structures. The entire program consists of a main routine (`main ( )`), which encloses a set of data declarations and a set of statements that manipulate these data. For simplicity, input/output is performed by invoking the opera-

tions get and print to read and write values, respectively.

```
main ( )
{
    int i, j;
    get (i, j);
    while (i != j)
        if (i > j)
            i -= j;
        else
            j -= i;
    print (i);
}
```

FIGURE 12.A C1 program

A C1 program is shown in Figure 12 and its straightforward SIMPLESEM representation before the execution starts is shown in Figure 12. The D portion shows the activation record of the main program, which contains space for all variables that appear in the program. The C portion shows the SIMPLESEM code.

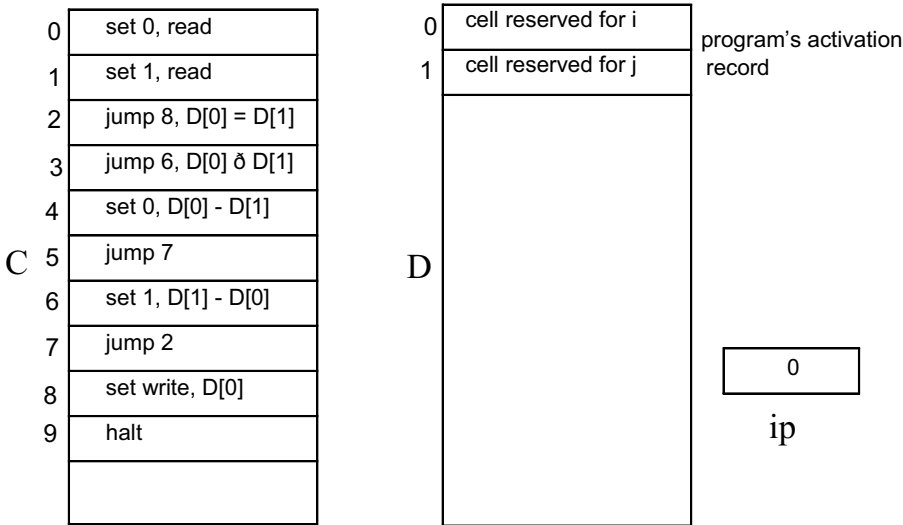


FIGURE 13.Initial state of the SIMPLESEM machine for the C1 program in Figure 12

2.6.2 C2: Adding simple routines

Let us now add a new feature to C1. The resulting language, C2, allows rou-

tines to be defined in a program and allows routines to declare their own local data. A C2 program consists of a sequence of the following items:

- a (possibly empty) set of data declarations (*global data*);
- a (possibly empty) set of routine definitions and or declarations;
- a *main routine* (`main ()`), which contains its local data declarations and a set of statements, that are automatically activated when the execution starts. The main routine cannot be called by other routines.

Routines may access their local data and any global data that are not redeclared internally. For simplicity, we assume that routines cannot call themselves recursively, do not have parameters, and do not return values (these restrictions will be removed later).

Figure 14 shows an example of a C2 program, whose main routine gets called initially, and causes routines beta and alpha to be called in a sequence.

```

int i = 1, j = 2, k = 3;
alpha ( )
{
    int i = 4, l = 5;
    ...
    i+=k+l;
    ...
};
beta ( )
{
    int k = 6;
    ...
    i=j+k;
    alpha ( );
    ...
};
main ( )
{
    ...
    beta ( );
    ...
}

```

**FIGURE 14.** A C2 program

Under the assumptions we made so far, the size of each unit's activation record can be determined at translation time, and all activation records can be allocated before execution (*static allocation*). Thus each variable can be bound to a D memory address before execution. Static allocation is a straightforward

implementation scheme which does not cause any memory allocation overhead at run time, but can waste memory space. In fact, memory is allocated for a routine even if it is never invoked. Since our purpose is to provide a semantic description, not to discuss an efficient implementation scheme, we assume static allocation. The run-time model described in Section 2.6.3 could be adapted to provide dynamic memory allocation for the C2 class of languages.

Figure 15 shows the state of the SIMPLESEM machine after instruction  $i += k + 1$  of routine *alpha* has been executed. The first location of each activation record (offset 0) is reserved for the *return pointer*. Starting at location 1, space is reserved for the local variables. In general, for an instance of unit *A*, the return pointer will contain the address of the instruction that should be executed after unit *A* terminates. This does not apply to *main*, which does not return to a caller. On real computers, however, *main* is called by the operating system, and after termination *main* must return control to the operating system. Also notice that we maintain an activation record to keep global data at the low address end of store *D*.



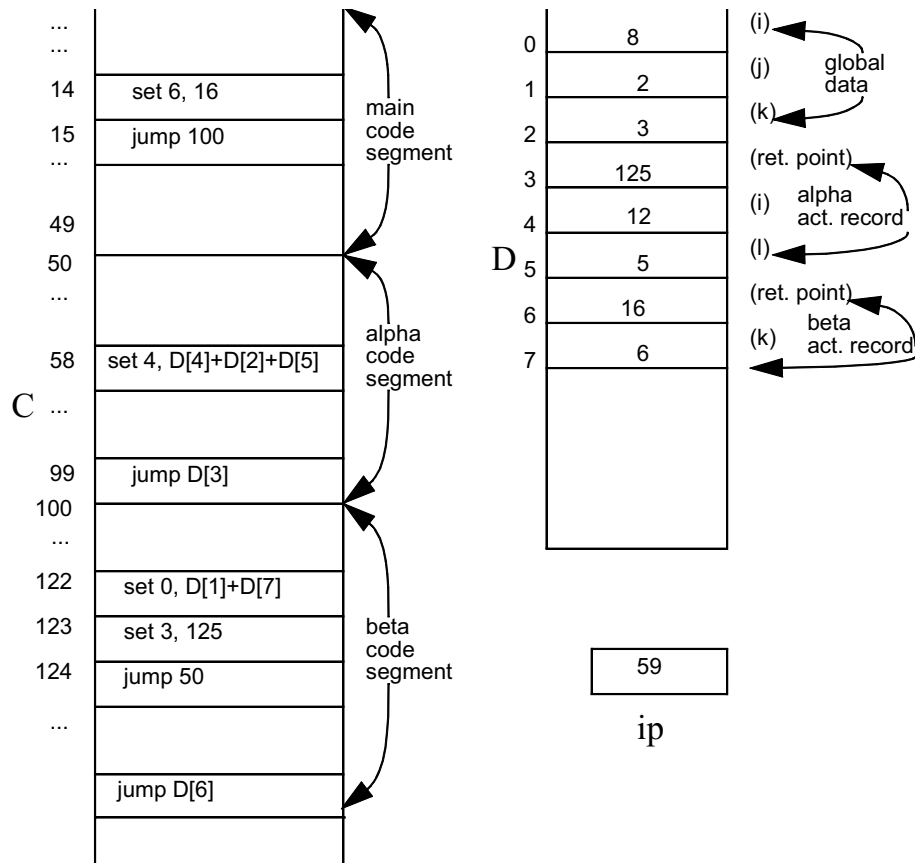


FIGURE 15. State of the SIMPSESEM executing the program of Figure 14

So far, we implicitly assumed that in C2 the main program and its routines are compiled in one monolithic step. It may be convenient instead to allow the various units to be compiled independently. This is illustrated by a variant of C2 (called C2') which allows program units to be put into separate files, and each file to be separately compiled in an arbitrary order. The file which contains the main program may also contain global data declarations, which may then be imported by other separately compiled units, which consist of single routines. If any of such routines needs to access some globally defined data, it must define them as external. Figure 16 shows the same example of Figure

14, using separate compilation.

<i>file 1</i>	<i>file 2</i>	<i>file 3</i>
int i = 1, j = 2, k = 3;	extern int k;	extern int i, j;
extern beta ();	alpha ()	extern alpha ();
main ()	{ ...	beta ()
{	}	{ ...
...		alpha ();
beta ();		...}
...}		

**FIGURE 16.** Program layout for separate compilation

As in the case of C2, a SIMPLESEM implementation can reserve the first location of each activation record (except for main) for the pointer to the caller's instruction, to be executed upon return. Further consecutive locations are then reserved for local variables, which can be bound to their offset within the activation record, as each routine is independently compiled. Independent compilation, however, does not allow variables to be bound to their absolute addresses. Because of independent compilation, imported global variables cannot even be bound to their offsets in the global activation record. Similarly, routine calls cannot be bound to the starting address of the corresponding code segments.

To resolve such unresolved addresses, a *linker* is used to combine the independently translated modules into a single executable module. The linker assigns the various code segments and activation records to stores C and D and fills any missing information that the compiler was unable to evaluate.

From this discussion we see that C2 and C2' do not differ semantically. Indeed, once a linker collects all separately compiled components, C2' programs and C2 programs cannot be distinguished. Their difference is in terms of the user-support they provide for the development of large programs. C2' allows parallel development by several programmers, who might work at the same time on different units.

Independent compilation, as offered by C2', is a simplified version of the facility offered by several existing programming languages, such as FORTRAN and C.

### 2.6.3 C3: Supporting recursive functions

Let us add two new features to C2: the ability of routines to call themselves (*direct recursion*) or to call one another in a recursive fashion (*indirect recursion*), and the ability of routines to return values, i.e., to behave as functions. These extensions define a new language, C3, which is illustrated in Figure 17 through an example.

```
int n;
int fact ( )
{
    int loc;
    if (n > 1) {
        loc = n--;
        return loc * fact ( );
    }
    else
        return 1;
}
main ( )
{
    get (n);
    if (n >= 0)
        print (fact ( ));
    else
        print ("input error");
}
```

FIGURE 17. A C3 example

As we mentioned in Section 2.4, in order to support mutual recursion between two routines—say, A and B—the program must be written according to the following pattern:

```
A's declaration (i.e., A's header);
B's definition (i.e., B's header and body);
A's definition;
```

Let us first analyze the effect of the introduction of recursion. Although each unit's activation record has a known and fixed size, in C3 it is not known how many instances of any unit will be needed during execution. As an example, for the program shown in Figure 17, at a given point of execution two activations are generated for function `fact` if the read value of `n` is greater than or equal to two. All different activations have the same code segment, since the code does not change from one activation to another, but they need different activation records, storing the different values of the local environment. As

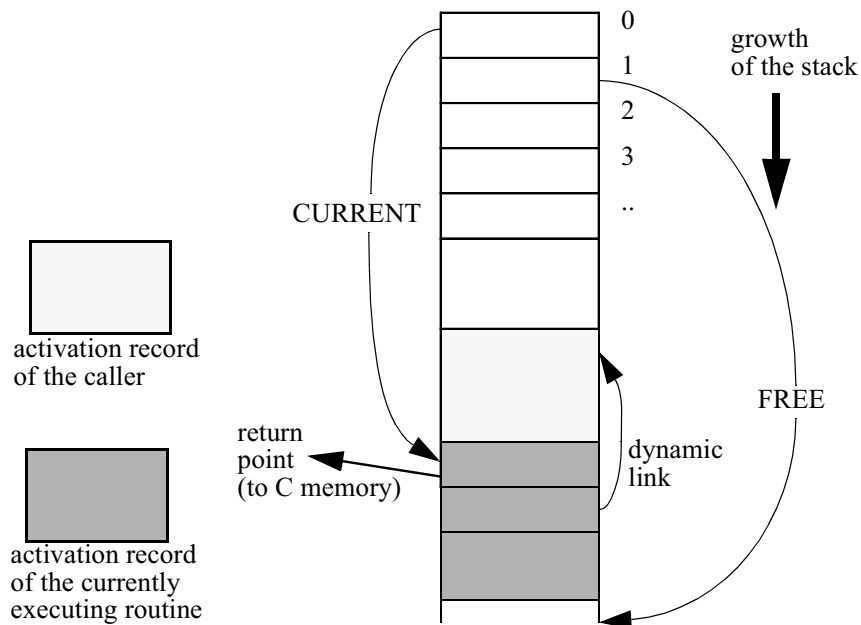
for C2, the compiler can bind each variable to its offset in the corresponding activation record. However, as opposed to C2, it is not possible to perform the further binding step which transforms it into an absolute address of the D store until execution time. In fact, an activation record is allocated by the invoking function for each new invocation, and each new allocation establishes a new binding with the corresponding code segment to form a new activation of the invoked function. Consequently, the final binding step which adds the offset of a variable—known statically—to the starting address (often called *base address*) of the activation record—known dynamically—can only be performed at execution time. To make this possible, we will use the cell at address zero in D to store the base address of the activation record of the currently executing unit (we also call this value CURRENT).

When the current instance of a unit terminates, its activation record is no longer needed. In fact, no other units can access its local environment, and the semantic rule of function invocation requires a new activation record to be freshly allocated. Therefore, after a function completes its current instance, it is possible to free the space occupied by the activation record and make it available to store new activation records in the future. For example, if A calls B which then calls C, the activation records for functions are allocated in the order A, B, C. When C returns to B, C's activation record can be discarded; B's activation record is discarded next, when B returns to A. Because the activation record that is freed is the one that was most recently allocated, activation records can be allocated with a *last-in/first-out* policy on a stack-organized storage.

In order to make return from an activation possible, the following necessary information is stored in the activation record: address of the instruction to be executed (*return point*) and base address of activation record to become active upon return. In the case of C2, only the return point needed to be saved, because through it the (unique) activation record associated with the callee also becomes known. If more than one activation may exist for a given unit, this more general solution becomes necessary. Therefore we assume that the cell at offset 0 of activation records contains the return point, while the cell at offset 1 contains a pointer to the base address of the caller's activation record—this pointer is called the *dynamic link*. The chain of dynamic links originating in the currently active activation record is called the *dynamic chain*. At any time, the dynamic chain represents the dynamic sequence of unit activations.

In order to manage SIMPLESEM's D store as a stack, it is necessary to know, at run time, the address of the first free cell of D, since a new activation record is allocated from that point on. We will use D's cell at address 1 to keep this information (we call this value FREE). Finally, it is necessary to provide memory space for the value returned by the routine, if it behaves as a function. Since the routine's activation record is deallocated upon return, the returned value must be saved into the caller's activation record. That is, when a functional routine is called, the caller's activation record is extended to provide space for the return value, and the callee writes the returned value into that space (using a negative offset, since the location is in the caller's activation record) before returning<sup>1</sup>.

Figure 18 provides an intuitive view of SIMPLESEM's D store. Activation records are allocated one on top of the previous, and the allocated memory grows from the upper part of the store (corresponding to low addresses) downwards.



1. For further details concerning the management of return values, see Exercise 21.

**FIGURE 18.** Structure of the SIMPLETEM D memory implementing a stack  
 Since recursive routines are the main additional features of C3, we now show how the semantics of routine call and return are specified in terms of SIMPLETEM instructions.

### *Routine call*

set 1, D[1] + 1	assume one cell is sufficient to hold the returned value
set D[1], ip + 4	set the value of the return point in the callee's activation record
set D[1] + 1, D[0]	set the dynamic link of the callee's activation record to point to the caller's activation record
set 0, D[1]	set CURRENT, the address of the currently executed activation record
set 1, D[1] + AR	set FREE (AR is the size of the callee's activation record)
jump start_addr	start_addr is an address of C where the first instruction of the callee's code is stored.

### *Return from routine*

set 1, D[0]	set FREE
set 0, D[D[0] + 1]	set CURRENT
jump D[D[1]]	jump to the stored return point

We assume that before the execution of a C3 program starts, ip is set to point to the first instruction of main ( ) and the D memory is initialized to contain space for the global data and for the activation record of main ( ). Such activation record contains space just for local main's variables (if any); space for the address of the return instruction and for the dynamic link are not needed, since the main routine does not return to a caller. Its termination simply means that the execution terminates. The values stored in D[0] and D[1] are also assumed to be initialized before execution. D[0] is set to the address of the first location of main's activation record and D[1] must be set to the address of the first free location after main's activation record.

As an exercise, let us show how the program of Figure 17 is executed by the SIMPLETEM machine. The code stored in the C memory is the following:

---

0	set 2, read	reads the value of n; 2 is the absolute address where global variable n is stored
1	jump 10, D[2] < 0	tests the value of n
2	set 1, D[1] + 1	call to fact starts here; space for the result saved
3	set D[1], ip + 4	
4	set D[1] + 1, D[0]	
5	set 0, D[1]	
6	set 1, D[1] + 3	3 is the size of fact's activation record
7	jump 12	12 is the starting address of fact's code
8	set write, D[D[1] - 1]	D[1] - 1 is the address where the result of the call to fact is stored
9	jump 11	end of the call
10	set write, "input error"	
11	halt	this is the end of the code of main
12	jump 23, D[2] $\delta$ 1	tests the value of n
13	set D[0] + 2, D[2]	assigns n to loc
14	set 2, D[2] - 1	decrements n
15	set 1, D[1] + 1	call to fact starts here; space for the result saved
16	set D[1], ip + 4	
17	set D[1] + 1, D[0]	
18	set 0, D[1]	
19	set 1, D[1] + 3	3 is the size of fact's activation record
20	jump 12	12 is the starting address of fact's code
21	set D[0] - 1, D[D[0] + 2] * D[D[1] - 1]	the returned value is stored in the caller's activation record
22	jump 24	
23	set D[0] - 1, 1	return 1
24	set 1, D[0]	this and the next 2 instructions correspond to the return from the routine
25	set 0, D[D[0] + 1]	
26	jump D [D[1]]	

Figure 19 provides two snapshots of the D memory: immediately after the first call to fact (case (a)) and at the return point from the third activation of fact when the initially read input value is 3 (case (b)). The reader is urged to try the example on paper, going through all intermediate steps of execution.

Note that the stack-based abstract implementation scheme discussed in this section also can be used for implementing C2. We discussed C2 in terms of static memory allocation, but this was simply an implementation choice. The advantage of a stack-based implementation would be that only the minimum amount of data store is allocated at any given time. The disadvantage, of course, is that a more complicated memory management scheme is needed.

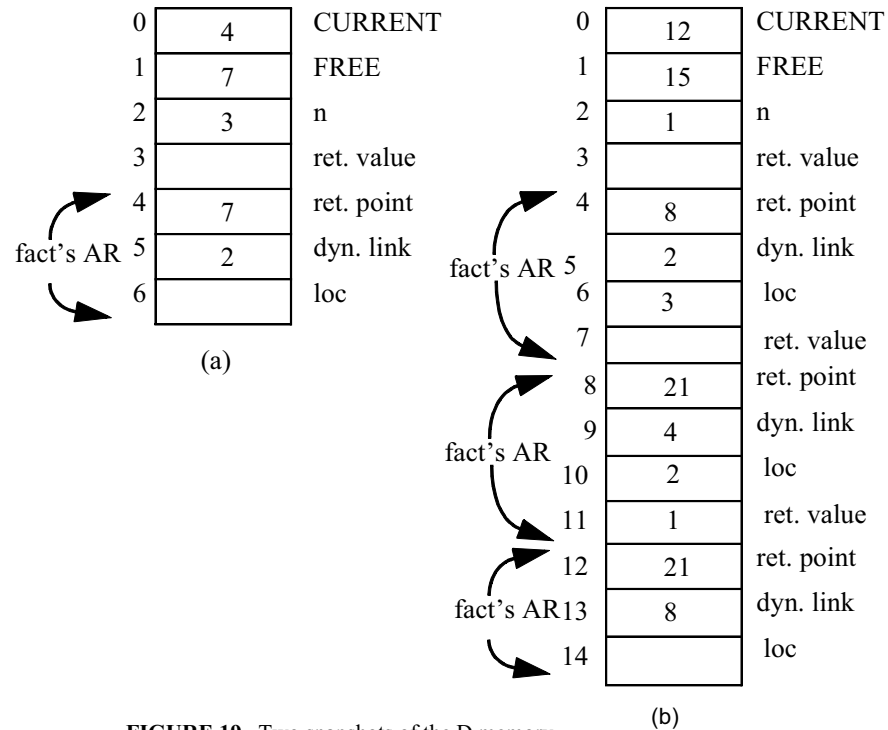


FIGURE 19. Two snapshots of the D memory

#### 2.6.4 C4: Supporting block structure

The structuring facilities offered by C3 allow programs to be defined as a sequence of global declarations of data and routines. Routines may call themselves in a recursive fashion. In this section we discuss a new extension to our language family, which collectively define a new family C4. The family C4 contains two members: C4' and C4". C4' allows local declarations to appear within any compound statement. C4" supports the ability to nest a routine definition within another. Conventionally, the new features offered by C4' and C4" are collectively called *block structure*. Block structure is used to control the scope of variables, to define their lifetime, and to divide the program into smaller units. Any two blocks in the program may be either disjoint (i.e., they have no portion in common) or nested (i.e., one block completely encloses the other).



#### 2.6.4.1 Nesting via compound statements

In C4', blocks have the following form of compound statement, which can appear wherever a statement can appear:

```
{<declaration_list>; <statement_list>}
```

It is easy to realize that such compound statements follow the aforementioned rule of blocks: they are either disjoint or they are nested. A compound statement defines the scope of its locally declared variables: such variables are visible within the compound, including any compound statement nested in it, provided the same name is not redeclared. An inner declaration masks an external declaration for the same name. Figure 20 shows an example of a C4' function having nested compound statements. Function *f* has local declarations for *x*, *y*, and *w*, whose scope extends from //1 to the entire function body, with the following exceptions:

- *x* is redeclared in //2. From that declaration until the end of the while statement the outer *x* is not visible;
- *y* is redeclared in //3. From that declaration until the end of the while statement, the outer *y* is not visible;
- *w* is redeclared in //4. From that declaration until the end of the if statement, the outer declaration is not visible.

Similarly, //2 declares variables *x* and *z*, whose visibility extends from the declaration until the end of the statement, with one exception. Since *x* is redeclared in //4, the outer *x* is masked by the inner *x*, which extends from the dec-

laration until the end of the if statement.

```

int f();
{
    int x, y, w;           //block 1
    while (...)           //1
    {
        int x, z;         //block 2
        ...
        while (...)
        {
            int y;        //block 3
            ...
        }                //end block 3
    }                    //end block 2
    if (...)
    {
        int x, w;         //block 4
        ...
    }                    //end block 3
}                        //end block 2
if (...)
{
    int a, b, c, d;       //block 5
    ...
}                        //end block 5
}                        //end block 1

```

**FIGURE 20.** An example of nested blocks in C4'

A compound statement also defines the lifetime of locally declared data. Memory space is bound to a variable  $x$  as the block in which it is declared is entered during execution. The binding is removed when the block is exited.

In order to provide an abstract implementation of compound statements for the SIMPLESEM machine, there are two options. One consists of statically defining an activation record for a routine with nested compound statements; another consists of dynamically allocating new memory space corresponding to local data as each compound statement is entered during execution. The former scheme is simpler and more time efficient, while the latter can lead to a more space efficient implementation. We will discuss the former scheme, and leave the latter to the reader as an exercise, which can easily be solved after reading Section 2.6.4.2 (see Exercise 27).

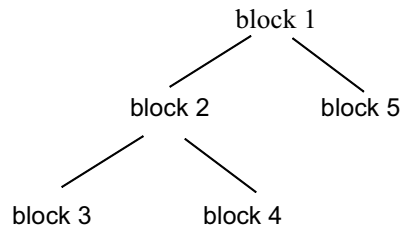
Let us refer to the example of Figure 20. Note that the while block that

declares variables  $x$  and  $z$  and the if block that declares  $a$ ,  $b$ ,  $c$ , and  $d$  are disjoint; similarly, the while block that declares variable  $y$  and the if block that declares  $x$  and  $z$  are disjoint. Since two disjoint blocks cannot be active at the same time, it is possible to use the same memory cells to store their local values. Thus, the activation record of function  $f$  can be defined as shown in Figure 21. The figure shows that the same cells may be used to store  $a$  and  $x$ ,  $b$  and  $w$ ,  $c$  and  $w$ , etc.; i.e., operator “--” denotes an overlay. The definition of overlays can be done at translation time. Having done so, the run-time behavior of  $C4'$  is exactly the same as was discussed in the case of  $C3$ .

return pointer
dynamic link
$x$ in //1
$y$ in //1
$w$ in //1
$x$ in //2-- $a$ in //5
$z$ in //2-- $b$ in //5
$y$ in //3-- $x$ in //4-- $c$ in //5
$w$ in //4-- $d$ in //5

FIGURE 21. An activation record with overlays

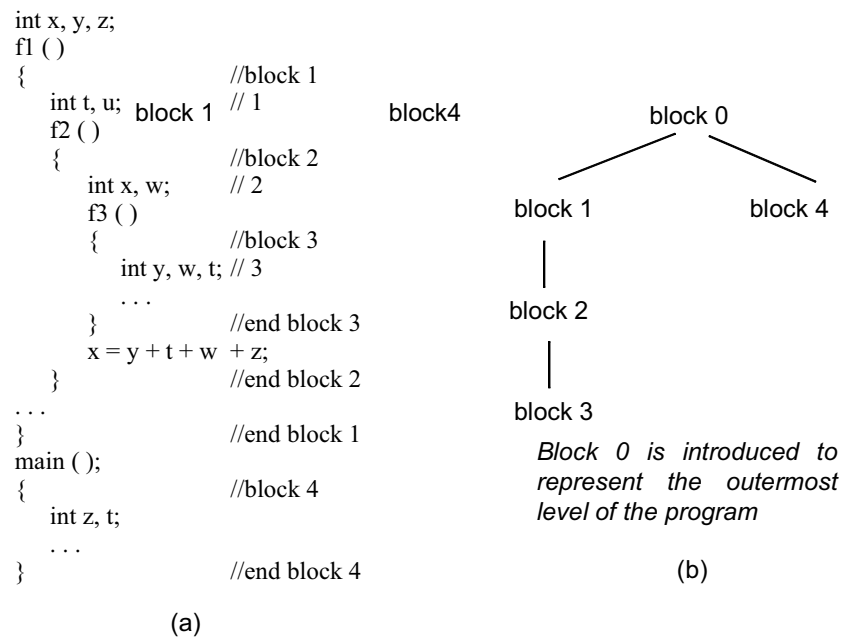
A block structure can be described by *static nesting tree* (SNT), which shows how blocks are nested into one another. Each node of a SNT describes a block; descendants of a node  $N$  which represents a certain block denote the blocks that are immediately nested within the block. For example, the program of Figure 20 is described by the static nesting tree of Figure 22.



**FIGURE 22.** Static nesting tree for the block structure of Figure 20

#### 2.6.4.2 Nesting via locally declared routines

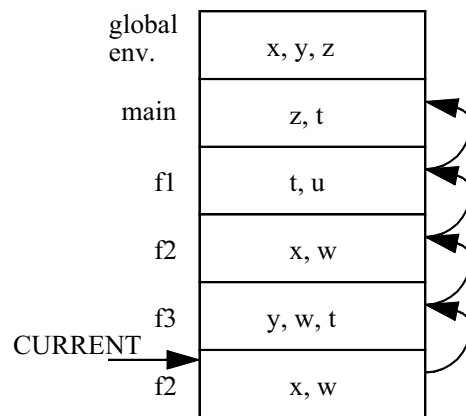
As we mentioned, block structure may result from the ability to nest compound statements within unnested routines, to nest routine definitions within routines, or both. C and C++ only support the nesting of compound statements within routines. Pascal and Modula-2 allow routine nesting, but do not support nesting of compound statements. Ada allows both.



**FIGURE 23.** A C4" example (a) and its static nesting tree (b)

Let us examine how routine nesting might be incorporated into our language. The resulting variation will be called C4". As shown in Figure 23 (a), in C4" a routine may be declared within another routine. Routine  $f_3$  can only be called within  $f_2$  (e.g., it would not be visible within  $f_1$  and main). A call to  $f_3$  within  $f_2$ 's body would be a local call (i.e., a call to a locally declared routine). Since  $f_3$  is internal to  $f_2$ ,  $f_3$  can also be called within  $f_3$ 's body (direct recursion). Such a call would be a call to a nonlocally declared routine, since  $f_3$  is declared in the outer routine  $f_2$ . Similarly,  $f_2$  can be called within  $f_1$ 's body (local call) and both within  $f_2$ 's and  $f_3$ 's bodies (nonlocal calls). Moreover, the data declared in //1 are visible from that point until the end of  $f_1$  (i.e., //6), with one exception. If a declaration for the same name appears in internally declared routines (i.e., in //2 or in //3), the internal declarations mask the outer declaration. Also, within a routine, it is possible to access both the local variables, and nonlocal variables declared by enclosing outer routines, if they are not masked. In the example, within  $f_3$ 's body, it is possible to access the non-local variables  $x$  (declared in //2),  $u$  (declared in //1) and the global variable  $z$ .

As shown in Figure 23 (b), the concept of a static nesting tree can be defined in this case too. Block 0 is introduced to represent the outermost level of the program, which contains the declarations of variables  $x$ ,  $y$ ,  $z$ , and functions  $f_1$  ( ) and  $\text{main}$  ( ).



**FIGURE 24.** A sketch of the run-time stack (dynamic links are shown as arrowed lines)

Let us examine the effect of the following sequence of calls:  $\text{main}$  calls  $f_1$ ,  $f_1$

calls  $f_2$ ,  $f_2$  calls  $f_3$ ,  $f_3$  calls  $f_2$ . Figure 24 shows a portion of the activation record stack corresponding to the example. The description is highly simplified, for readability purposes, but shows all the relevant information. For each activation record, we indicate the name of the corresponding routine, the dynamic link, and the names of variables whose values are kept in it. Let us suppose that the execution on the SIMPLESEM machine reaches the assignment  $x = y + t + w + z$  in  $f_2$ . The translation process we discussed so far is able to bind variables  $x$  and  $w$  to offsets 2 and 3 of the topmost activation record (whose initial address is given by  $D[0]$ , i.e.,  $CURRENT$ ); but what about variables  $y$ ,  $t$ , and  $z$ ? For sure, they should not be bound according to the most recently established binding for such variable names, since such binding were established by the latest activations of routines  $f_3$  ( $y$  and  $t$ ) and  $main$  ( $z$ ). However, the scope rules of C4" require variables  $y$  and  $z$  referenced within  $f_2$  to be the ones declared globally, and variable  $t$  to be the one declared locally in  $f_1$ . In other words, the sequence of activation records stored in the stack represent the sequence of unit instances, as they are dynamically generated at execution time. But what determines the nonlocal environment are the scope rules of the language, which depend on the static nesting of routine declarations.

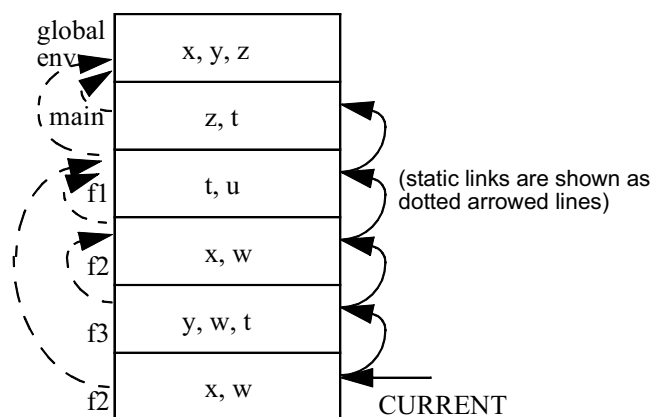


FIGURE 25. The run-time stack of Figure 24 with static links

One way to make access to nonlocal variables possible is for each activation record to contain a pointer (*static link*) up the stack to the activation record of the unit that statically encloses it in the program text. We will use the location

of the activation record at offset 2 to store the value of the static link. Figure 25 shows the static links for the example of Figure 24. The sequence of static links that can be followed from the active activation record is called the *static chain*. Referencing nonlocal variables can be explained intuitively as a search that traverses the static chain. To find the correct binding between a variable and a stack location, the static chain is searched until a binding is found. In our example, the reference to *t* is bound to a stack location within *f1*'s activation record, whereas references to *y* and *z* are bound to a stack location within the global environment—as indeed it should be. Notice that in this scheme, the global environment is accessed in the same uniform way as any other nonlocal environment. In such a case, the value of the static link for *main*'s activation record is assumed to be set automatically before execution. In order to use the cell at offset 2 of *main*'s activation record to hold the value of the static link, as we do for any other activation record, the cells at offsets 0 and 1 are kept unused. Alternatively, access to the global environment can be treated as a special case, by using absolute addresses.

In practice, searching along the static chain, which would entail considerable run-time overhead, is never necessary. A more efficient solution is based on the fact that the activation record containing a variable named in a unit *U* is always a fixed distance from *U*'s activation record along the static chain. If the variable is local, the distance is obviously zero; if it is a variable declared in the immediately enclosing unit, the distance is one; if it is a variable declared in the next enclosing unit, the distance is 2, and so on. In general, for each reference to a variable, we can evaluate a *distance attribute* between that reference and the corresponding declaration. This distance attribute can be evaluated and bound to the variable at translation time. Consequently, each reference may be statically bound to a pair (distance, offset) within the activation record.

Based on the pair (distance, offset), it is possible to define the following addressing scheme for SIMPLESEM. If *d* is the value of the distance, starting from the address of the current activation record (*CURRENT*, the value stored in *D[0]*), we traverse *d* steps along the static chain. The value of the offset is then added to the address so found, and the result is the actual run-time address to the nonlocal data object. We can define this formally in terms of a recursive function *fp* (*d*), which can then be easily translated into SIMPLESEM. Function *fp* (*d*), which stands for the frame pointer—a pointer to an activation record—that is *d* static links away from the active activation record,

can be defined as:

$fp(d) = \text{if } d=0 \text{ then } D[0] \text{ else } D[fp(d-1)+2]$

For example,  $fp(0)$  is simply  $D[0]$ , i.e., the address of the current (topmost) activation record; and  $fp(1)$  is  $D[D[0]+2]$ .

Using  $fp$ , access to a variable  $x$ , with  $\langle \text{distance, offset} \rangle$  pair  $\langle d, o \rangle$ , is provided by the following address:

$D[fp(d)+o]$

The semantics of function call defined in Section 2.6.3 needs to be modified in the case of C4", in order to take into account the installation of static links in activation records. This can be done in the following way. First, notice that, as we did for variables, one can define the concept of distance between a routine call and the corresponding declaration. Thus, if  $f$  calls a local routine  $f_1$ , then the distance between the call and the declaration is 0. If  $f$  contains a call to a function declared in the block enclosing  $f$ , the distance is 1. This, for example, would be the case if  $f$  calls itself recursively. If  $f$  is local to function  $g$  and  $f$  contains a call to a function  $h$  declared in the block enclosing  $g$ , the distance between the call and the declaration is 2, and so on. Therefore, the static link to install for activation record of the callee, if the callee is declared at distance  $d$ , should point to the activation record that is  $d$  steps along the static chain originating from the caller's activation record.

In conclusion, the semantics of routine call can be defined by the following SIMPLESEM code:

#### *Routine call*

set 1, $D[1] + 1$	set space for the result of the function call (assume 1 cell needed)
set $D[1]$ , $ip + 5$	set the value of the return point in the callee's activation record
set $D[1] + 1$ , $D[0]$	set the dynamic link of the callee's activation record to point to the caller's activation record
set $D[1] + 2$ , $fp(d)$	set the static link of the callee's activation record
set 0, $D[1]$	set CURRENT, the address of the currently executed activation record



<pre>set 1, D[1] + AR jump start_addr</pre>	<pre>set FREE (AR is the size of the callee's activation record) start_addr is an address of memory C where the first instruction of the callee's code is stored.</pre>
---	---

### 2.6.5 C5: Towards more dynamic behaviors

So far we assumed that the data storage requirements of each unit are known at compile time, so that the required amount of memory can be reserved when the unit is allocated. Furthermore, the mapping of variables to storage within the activation record can be performed at compile time; i.e., each variable is bound to its offset statically. In this section we discuss language features that invalidate this assumption, and we show how to define semantics of such features.

#### 2.6.5.1 Activation records whose size becomes known at unit activation

Let us first introduce language C5', by relaxing the assumption that the size of all variables is known at compile time. Such is the case for dynamic arrays, that is, arrays whose bounds become known at execution time, when the unit (routine or compound statement) in which the array is declared is activated.

For example, in the Ada programming language, it is possible to define the following type:

```
type VECTOR is array (INTEGER range <>); --defines arrays with unconstrained index
and declare the following variables:
```

```
A: VECTOR (1..N);
B: VECTOR (1..M); --N and M must be bound to some integer value when these two dec-
larations are processed at execution time
```

The abstract implementation that defines the semantics for this case is rather straightforward. At translation time, storage can be reserved in the activation record for the descriptors of the dynamic arrays. The descriptor includes one cell in which we store a pointer to the storage area for the dynamic array and one cell for each of the lower and upper bounds of each array dimension. As the number of dimensions of the array is known at translation time, the size of the descriptor is known statically. All accesses to a dynamic array are translated as indirect references through the pointer in the descriptor, whose offset is determined statically.


At run time, the activation record is allocated in several stages.

1. The storage required for data whose size is known statically and for descriptors of dynamic arrays are allocated.
2. When the declaration of a dynamic array is encountered, the dimension entries in the descriptors are entered, the actual size of the array is evaluated, and the activation record is extended (that is, FREE is increased) to include space for the variable. (This expansion is possible because, being the active unit, the activation record is on top of the stack.)
3. The pointer in the descriptor is set to point to the area just allocated.


In the previous example, let us suppose that the descriptor allocated when variable A is declared is at offset m. The cell at offset m will point at run time to the starting address of A; the cells at offsets m+1 and m+2 will contain the lower and upper bounds, respectively, of A's index. The run-time actions corresponding to entry into the unit where A's declaration appears will update the value of D[1] (i.e., FREE) to allocate space for A, based on the known value of N, and will set the values of the descriptor at offsets m, m+1, and m+2.

Any access to elements of A are translated to indirect references. Assuming each integer occupies one location of D and supposing that I is a local variable stored at offset s, instruction A [I] = 0 would be translated into SIMPLESEM as:

set [D[D[0] + m] + D[D[0] + s]], 0



denotes the base  
address of A



denotes the value of I

#### 2.6.5.2 Fully dynamic data allocation

Now let us consider another language variation, called C5", in which data can be allocated explicitly, through an executable allocation instruction. In most existing languages, this is achieved by defining pointers to data, and by providing statements that allocate such data in a fully dynamic fashion.

For example, in C++ we can define the following type for nodes of a binary tree:

```
struct node {
    int info;
    node* left;
    node* right;
};
```

The following instruction, which may appear in some code fragment:

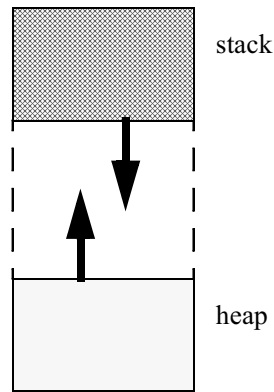
---

```
node* n = new node;
```

explicitly allocates a structure with the three fields `info`, `left`, and `right`, and makes it accessible via the pointer `n`.

According to this allocation scheme, data are allocated explicitly as they are needed. We cannot allocate such data on a stack, as do automatically allocated data. For example, suppose that a function `append_left` is called to generate a new node and make its accessible through field `left` of node pointed by `n`. Also, suppose that `n` is visible by `append_left` as a nonlocal variable. If the node allocated by `append_left` would be allocated on the stack, it would be lost when `append_left` returns. The semantics of these dynamically allocated data, instead, is that their lifetime does not depend on the unit in which their allocation statement appears, but lasts as long as they are accessible, i.e., they are referred to by some existing pointer variables, either directly or indirectly.

An abstract implementation of this concept using SIMPLESEM can be very simple, and consists of allocating dynamic data in `D` starting from the high-address end. This area of `D` is also called the *heap*. New data are allocated in the heap as the allocation instructions are executed, and we can assume that the size of the SIMPLESEM `D` store is sufficient to hold all data that are dynamically allocated via the `new` instruction. Figure 26 gives an overall view of how memory `D` is handled, in order to support both a stack and a heap. We will return to the practical issue of actually implementing dynamic allocation in a memory-efficient way in Chapter 3. From a semantic viewpoint, this simple implementation scheme can be sufficient.



**FIGURE 26.** Management of the D memory

#### Sidebar start The Structure of Dynamic Languages

The term “dynamic languages” implies many things. In general, it refers to those languages that adopt dynamic rather than static rules. For example, APL, SNOBOL4 and several LISP variants use dynamic typing and dynamic scope rules. In principle, of course, a language designer can make these choices independently of one another. For example, one can have dynamic type rules but static scope rules. In practice, however, dynamic properties are often adopted together.<sup>1</sup>

In this sidebar, we will examine how the adoption of dynamic rules changes the semantics of the language in terms of run-time requirements. In general, a dynamic property implies that the corresponding bindings are carried out at run time and cannot be done at translation time. We will examine dynamic typing and dynamic scoping.

In a language that uses *dynamic typing*, the type of a variable and therefore the methods of access and the allowable operations cannot be determined at translation time. In Section 2.6.5, we saw that we need to keep a run-time descriptor for dynamic arrays variables, because we cannot determine the size of or starting address of such variables at translation time. In that case, the

---

<sup>1</sup> We already mentioned that there are dynamically typed languages (like ML and Eiffel) that support static type checking.

---

descriptor has to contain the information that cannot be computed at translation time, namely, the starting address and the array bounds. It was possible to keep the descriptor in the activation record because the size of the descriptor was fixed and known at translation time. In the case of dynamically typed variables, we also need to maintain the type of the variable in the descriptor. If the type of a variable may change at run time, then the size and contents of its descriptor may also change. For example, if a variable changes from a two-dimensional array to a three-dimensional array, then the descriptor needs to grow to contain the values of the bounds for the new dimension. This is in contrast to the descriptors of dynamic arrays whose contents were fixed at unit activation time. Every access to a dynamic variable must be preceded by a run-time check on the type of the variable, followed by appropriate address computation, depending on the current type of the variable.

What is maintained for each variable in the activation record for a unit? Since not only the variable's size may change during program execution, but so may the size of its descriptor, descriptors must be kept in the heap. For each variable, we maintain a pointer in the activation record that points to the variable's descriptor in the heap which, in turn, may contain a pointer to the object itself in the heap.

In order to discuss the effect of dynamic scope rules, let us consider the example program of Figure 27. The program is written using a C-like syntax, but it will be interpreted according to an APL-like semantics, i.e., according to

dynamic scope rules.

```

sub2 ( )
{
    declare x;
    ...
    ... x ...;
    ... y ...;
    ...
}
sub1 ( )
{
    declare y;
    ...
    ... x ...;
    ... y ...;
    sub2 ( );
    ...
}
main ( )
{
    declare x, y, z;
    z = 0;
    x = 5;
    y = 7;
    sub1;
    sub2;
    ...
}

```

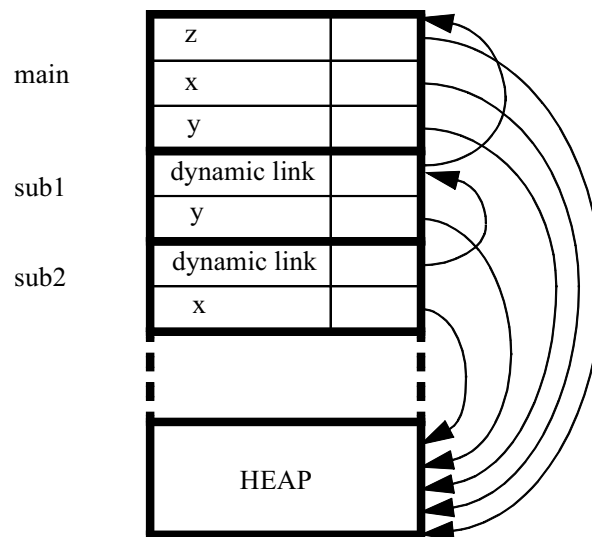
**FIGURE 27.** An example of a dynamically scoped language

A program consists of a number of routines and a main program. Each routine declares its local variables ( $y$  in the case of `sub1`,  $x$  in the case of `sub2`,  $x, y, z$  in the case of `main`). Any access to a variable that is not locally declared is implicitly assumed to be an access to a nonlocal variable. A variable declaration does not specify the variable's type: it simply introduces a new name. Routine names are considered as global identifiers.

Since scope rules are dynamic, the scope of a name is totally dependent on the run-time call chain (i.e., on the dynamic chain), rather than the static structure of the program. In the example shown in Figure 27 consider the point when the call to `sub1` is issued in `main`. The nonlocal references to  $x$  and  $z$  within the activation of `sub1` are bound to the global  $x$  and  $z$  defined by `main`. When function `sub2` is activated from `sub1`, the nonlocal reference to  $y$  is bound to the

most recent definition of *y*, that is, to the data object associated to *y* in *sub1*'s activation record. Return from routines *sub2* and then *sub1* causes deallocation of the corresponding activation records and then execution of the call to *sub2* from *main*. In this new activation, the nonlocal reference to *y* from *sub2*, which was previously bound to *y* in *sub1*, is now bound to the global *y* defined in *main*.

An abstract implementation mechanism to reference nonlocal data can be quite simple. Activation records can be allocated on a stack and joined together by dynamic links, as we saw in the case of conventional languages. Each entry of the activation record explicitly records the name of the variable and contains a pointer to a heap area, where the value can be stored. Allocation on a heap is necessary because the amount of storage required by each variable can vary dynamically. For each variable—say, *V*—the stack is searched by following the dynamic chain. The first association found for *V* in an activation record is the proper one. Figure 28 illustrates the stack for the program of Figure 20 when *sub2* is called by *sub1*, which is, in turn, called by *main*.



**FIGURE 28.** A view of the run-time memory for the program of Figure 27

Although simple, this accessing mechanism is inefficient. Another approach

is to maintain a table of currently active nonlocal references. Instead of searching along the dynamic chain, a single lookup in this table is sufficient. We will not discuss this solution any further but the reader should note that this technique speeds up referencing nonlocal variables at the expense of more elaborate actions to be executed at subprogram entry and exit. These additional actions are necessary to update the table of active nonlocal references.

Sidebar end

### 2.6.6 Parameter passing

So far we assumed that routines do not have parameters: we only assumed that they can return a value (see Section 2.6.3). We will now remove this limitation by discussing how parameter passing may be abstractly implemented on SIMPLESEM. We first address the issue of data parameters, and then we will analyze routine parameters.

#### 2.6.6.1 Data parameters

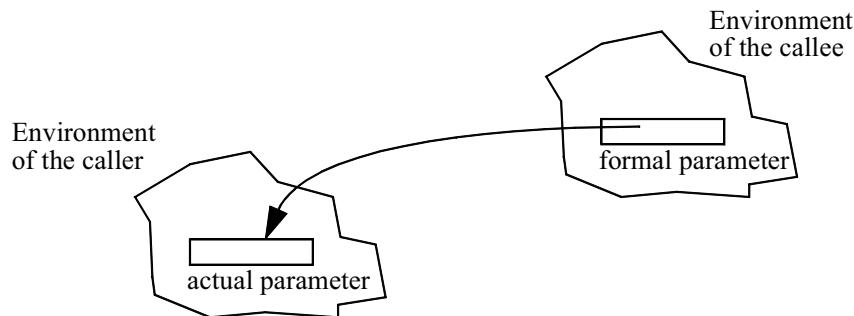
There are different conventions for passing data parameters to routines. The adopted convention is either predefined by the language, and therefore it is part of the language semantics, or can be chosen by the programmer from several options. In either case, it is important to know which convention is adopted, because the choice made affects the meaning of programs. The same program may in fact produce different results under different data parameter passing conventions. Three conventions for data parameters are discussed below: call by reference, call by copy, and call by name. Each of them is first introduced informally, and then defined precisely in terms of SIMPLESEM actions.

##### *Call by Reference (or by Sharing)*

The calling unit passes to the called unit the address of the actual parameter (which is in the calling unit's referencing environment). A reference to the corresponding formal parameter in the called unit is treated as a reference to the location whose address is so passed. The effect of call by reference is intuitively described in Figure 29. If the formal parameter is assigned a value, the corresponding actual parameter changes value. Thus, a variable that is transmitted as an actual parameter is shared, that is, directly modifiable by the subprogram. If an actual parameter is anything other than a variable, for example, an expression or a constant, the subprogram receives the address within the



calling unit's activation record of a temporary location that contains the value of the actual parameter. Some languages treat this situation as an error.



**FIGURE 29.** A view of call by reference

Suppose that call by reference is being added to C4. In order to define a SIMPLESEM implementation that specifies semantics precisely, we need to extend the actions described in Section 2.6.3. The callee's activation record must contain one cell for each parameter. At procedure call the caller must initialize the contents of the cell to contain the address of the corresponding actual parameter. If the parameter cell is at offset *off* and the actual parameter, which is bound to the pair (*d*, *o*), is not itself a by-reference parameter, the following action must be added for each parameter:

set  $D[0] + \text{off}, \text{fp}(d) + o$

If the actual parameter itself is a by-reference parameter, the SIMPLESEM action should be:

set  $D[0] + \text{off}, D[\text{fp}(d) + o]$

When the routine body is executed, parameter reference is performed via indirect addressing. Thus, if *x* is a formal parameter and *off* is its offset, instruction

$x = 0;$

is translated as

set  $D[D[0] + \text{off}], 0$

*Call by copy*

In call by copy—unlike in call by reference—formal parameters do not share

---

storage with actual parameters; rather, they act as local variables. Thus, call by copy protects the calling unit from intentional or inadvertent modifications of actual parameters. It is possible further to classify call by copy into three modes, according to the way local variables corresponding to formal parameters are initialized and the way their values ultimately affect the actual parameters. These three modes are call by value, by result, and by value-result.

In *call by value*, the calling unit evaluates the actual parameters, and these values are used to initialize the corresponding formal parameters, which act as local variables in the called unit. Call by value does not allow any flow of information back to the caller, since assignments to formal parameters (if permitted) do not affect the calling unit.

In *call by result*, local variables corresponding to formal parameters are not set at subprogram call, but their value, at termination, is copied back into the corresponding actual parameter's location within the environment of the caller. Call by result does not permit any flow of information from the caller to the callee.

In *call by value-result*, local variables denoting formal parameters are both initialized at subprogram call (as in call by value) and delivered upon termination (as in call by result). Information thus flows from the caller to the callee (at the point of call) and from the callee to the caller (at the return point).

A description of the semantics of call by value in terms of a SIPLESEM implementation is trivial. The callee's activation record must contain space for by-value parameters, as normal local data. The difference here is that the call must provide for initialization of such data. We leave this and the other two cases of call by copy as exercises for the reader.

One might wonder whether call by reference and call by value-result are equivalent. If this were the case, implementing parameter passing in one mode would be equivalent to implementing it in the other. It can be shown, however, that call by reference may produce a different result from call by value-result in the following cases:

- two formal parameters become aliases (i.e., the two different names denote the same object);

- a formal parameter and a nonlocal variable which is visible both by the caller and by the callee become aliases.

We will provide two examples to motivate these statements. The first case may happen if—say— `a[i]` and `a[j]` are two integer actual parameters corresponding to the formal parameters `x` and `y`, and `i` happens to be equal to `j` at the point of call. In such a case, the effect of call by reference is that `x` and `y` would be aliased, since they would refer to the same array element. If the routine contains the following statements:

```
x = 0;
y ++;
```

the result of the call is that the array element of index `i` (and `j`) is set to 1. In the case of call by value-result, let `a[i]` be 10 at the point of call. The call would initialize `x` and `y` to the 10. Then `x` becomes 0 and `y` becomes 11, due to the above assignment statements. Finally, upon return, first 0 is copied back into `a[i]` and then 11 is copied back into the same cell, if copies are performed in this order. As a result, the array element is set to 11.

As an example of the second case, suppose that a routine is called with one integer actual parameter `a` which corresponds to the formal parameter `x`. Let `a` be visible by the routine as a nonlocal variable. Suppose that the routine contains the following statements:

```
a = 1;
x = x + a;
```

In the case of call by reference, the effect of the call is that `a` is set to 2. In the case of call by value-result, if `a`'s value is 10 at the call point, the value becomes 11 upon return.

#### *Call by name*

As in call by reference, a formal parameter, rather than being a local variable of the subprogram, denotes a location in the environment of the caller. Unlike with call by reference, however, the formal parameter is not bound to a location at the point of call; it is bound to a (possibly different) location each time it is used within the subprogram. Consequently, each assignment to a formal parameter can refer to a different location.

Basically, in call by name each occurrence of the formal parameter is replaced textually by the actual parameter. This may be achieved by a simple kind of macro processing, with one exception that will be discussed below.

This apparently simple rule can lead to unsuspected complications. For example, the following procedure, which is intended to interchange the values of *a* and *b* (*a* and *b* are by-name parameters)

```
swap (int a, b);
int temp;
{
    temp = a;
    a = b;
    b = temp;
};
```

most likely produces an unexpected result when invoked by the call

```
swap (i, a [i])
```

The replacement rule specifies that the statements to be executed are

```
temp = i;
i = a [i];
a [i] = temp;
```

If *i* = 3 and *a* [3] = 4 before the call, *i* = 4 and *a* [4] = 3 after the call (*a* [3] is unaffected)!

Another trap is that the actual parameter that is (conceptually) substituted into the text of the called unit belongs to the referencing environment of the caller, not to that of the callee. For example, suppose that procedure *swap* also counts the number of times it is called and it is embedded in the following fragment.

```
int c;
...
swap (int a, b);
int temp;
{
    temp = a; a = b;
    b = temp; c ++;
}

y ();
int c, d;
{
    swap (c, d);
};
```

When *swap* is called by *y*, the replacement rule specifies that the statements to be executed are

```
temp = c;
```

```
c = d;  
d = temp;  
c ++;
```

However, the location bound to name *c* in the last statement belongs to *x*'s activation record, whereas the location bound to the previous occurrences of *c* belong to *y*'s activation record. This shows that plain macro processing does not provide a correct implementation of call by name if there is a conflict between names of nonlocals in the routine's body and names of locals at the point of call. This example also shows the possible difficulty encountered by the programmer in foreseeing the run-time binding of actual and formal parameters.

Call by name, therefore, can easily lead to programs that are hard to read. It is also unsuspectedly hard to implement. The basic implementation technique consists of replacing each reference to a formal parameter with a call to a routine (traditionally called *thunk*) that evaluates a reference to the actual parameter in the appropriate environment. One such *thunk* is created for each actual parameter. The burden of run-time calls to *thunks* makes call by name costly.

Due to these difficulties, call by name has mostly theoretical and historical interests, but has been abandoned by practical programming languages.

Call by reference is the standard parameter passing mode of FORTRAN. Call by name is standard in ALGOL 60, but, optionally, the programmer can specify call by value. SIMULA 67 provides call by value, call by reference, and call by name. C++, Pascal and Modula-2 allow the programmer to pass parameters either by value (default case) or by reference. C adopts call by value, but allows call by reference to be implemented quite easily via pointers. Ada defines parameter passing based on the intended use, as either *in* (for input parameters), *out* (for output parameters), or *inout* (for input/output parameters), rather than in terms of the implementation mechanism (by reference or by copy). If the mode is not explicitly specified, *in* is assumed by default. More on this will be discussed in Chapter 4.

#### 2.6.6.2 Routine parameters

Languages supporting variables of type routine are said to treat routines as first-class objects. In particular, they allow routines to be passed as parameters. This facility is useful in some practical situations. For example, a routine *S* that evaluates an analytic property of a function (e.g., derivative at a given

---

point) can be written without knowledge of the function and can be used for different functions, if the function is described by a routine that is sent to *S* as a parameter. As another example, if the language does not provide explicit features for exception handling (see Chapter 4), one can transmit the exception handler as a routine parameter to the unit that may raise an exception behavior.

Routine parameters behave very differently in statically and dynamically scoped languages. Here we concentrate on statically scoped languages. Hints on how to handle dynamically scoped languages are given in a sidebar.

Consider the program in Figure 30. In this program, *b* is called by *main* (line 14) with actual parameter *a*; inside *b*, the formal parameter *x* is called (line 15), which in this case corresponds to *a*. When *a* is called, it should execute normally just as if it had been called directly, that is, there should be no observable differences in the behavior of a routine called directly or through a formal parameter. In particular, the invocation of *a* must be able to access the nonlocal environment of *a* (in this case the global variables *u* and *v*. Note that these variables are not visible in *b* because they are masked by *b*'s local variables with the same names.) This introduces a slight difficulty because our current abstract implementation scheme does not work. As we saw in Section 2.6.4, the call to a routine is translated to several instructions. In particular, it is necessary to reserve space for the activation record of the callee and to set up its static link. In the case of “call *x*” in *b*, this is impossible at translation time because we do not know what routine *x* is, let alone its enclosing unit. This information, in general, will only be known at run time. We can handle this situation by passing the size of the activation record and the needed static

link at the point of call.

```

1  int u, v;
2  a ( )
3  {
4      int y;
5      ...
6  };
7  b (routine x)
8  {
9      int u, v, y;
10     c ( )
11     { ...
12         y = ... ;
13         ...
14     };
15     x ( );
16     b (c);
17     ...
18 }
19 main ( )
20 {
21     b (a);
22 };

```

**FIGURE 30.** An example of routine parameters

In general, how do we know this static link to pass? From the scope rules, we know that in order for a unit  $x$  (in this case, `main`) to pass routine `a` to routine `b`,  $x$  must either:

- (a) Have procedure `a` within its scope, that is, `a` must be nonlocally visible or local (immediately nested); or
- (b) `a` must be a formal parameter of  $x$ , that is, some actual procedure was passed to  $x$  as a routine parameter<sup>1</sup>.

The two cases can be handled in the following way:

*Case (a):* The static link to be passed is `fp(d)`, a pointer to the activation record that is  $d$  steps along the static chain originated in the calling unit, where  $d$  is the distance between the call point where the routine parameter is passed and its declaration (recall Section 2.6.4).

1. Case (b) cannot occur in the case where  $x$  is `main`, since `main` cannot be called by other routines.

*Case (b):* The static link to be passed is the one that was passed to the caller.

We leave the task of formulating these rules in terms of SIMPLESEM as an exercise for the reader.

What about calling a routine parameter? The only difference from calling a routine directly is that both the size of the callee's activation record and its static link are simply copied from the parameter area.

The program in Figure 30 shows another subtle point: when routine parameters are used in a program, nonlocal variables visible at a given point are not necessarily those of the latest allocated activation record of the unit where such variables are locally declared. For example, after the recursive call to *b* when *c* is passed (line 16), the call to *x* in *b* (line 15) will invoke *c* recursively. Then the assignment to *y* in *c* (line 12) will not modify the *y* in the latest activation record for *b* but in the one allocated prior to the latest one. Figure 30 shows this point.

Let us review the impact of procedural parameters. First, we had to extend the basic procedure call mechanism to deal with the additional semantic complexity. Procedure calls now have to deal with different cases of objects. Both the procedure call's semantic description and its implementation have increased in complexity. Contrast this with, say, adding a new arithmetic operator to a language that requires hardly any changes to our semantic description at all. We can say that the ability to pass procedures as parameters adds to the semantic power (and complexity) of a language. On the other hand, it makes the language more uniform in the way the different language constructs are handled: routines are first-class objects, and can be treated uniformly as any other objects of the language.

This is an example of a general property of languages, called *orthogonality*. This term describes the ability of a language to support any combination of basic constructs to achieve any degree of power, without restrictions and without "special cases".

sidebar-start Routine parameters in dynamically scoped languages

Routines passed as parameters sometimes cause a peculiar problem in languages with dynamic scope rules, such as early versions of LISP. If we con-



sider the program in Figure 30 under dynamic scope rules, when routine *a* is called through *x*, references to *u* and *v* in *a* will be bound to the *u* and *v* in *b* and not to those in *main*. This is difficult to use and confusing since when the routine *a* was written, it was quite reasonable to expect access to *u* and *v* in *main* but because *b* happens to contain variables with the same names, they mask out the variables that were probably intended to be used.

Simply stated, the problem is that the nonlocal environment, and therefore the behavior of the routine, is dependent on the dynamic sequence of calls that have been made before it was activated. Consider several programmers working on different parts of the same program. A seemingly innocuous decision, what to name a variable, can change the behavior of the program entirely.

The problem, however, was discovered very early in the development of LISP and a new feature was added to the language to allow a routine to be passed along with its naming environment. If a routine is preceded by the keyword `FUNCTION`, the routine is passed along with its nonlocal environment at the point of call. When such a procedure is invoked, the environment information passed with the parameter is used to set up the current nonlocal environment. This is a rather complicated mechanism, but it seems to be the only reasonable way for procedural parameters to access the nonlocal environment. Of course, a different—and more radical—solution to the problem would be to change the language semantics, and adopt static scope rules for the entire language, as most modern LISPs do.

sidebar end

## 2.7 Bibliographic notes

In this chapter we have studied programming language semantics in an informal but systematic way, by describing the behavior of an abstract language processor. Formal approaches to the definition of semantics are also possible, as we briefly discussed. (Meyer 1991) provides a view of the theoretical foundations of programming languages and their semantics. Our view here is oriented towards language implementation, in order to allow the reader to appreciate the resources that may be needed, and the costs that may be involved, in running a program. We have emphasized the important concepts of binding, binding time, and binding stability. This viewpoint is taken by other textbooks on programming languages, from a classic (Pratt 1984) to a recent one