7 Subprograms

7.1 Subprograms: procedures and functions

A *subprogram* is a segment of a program that can be invoked from elsewhere within the program. Subprograms are used for various reasons:

- A segment of a program that must be executed at various stages within the computation can be written once as a subprogram and then repeatedly invoked. This saves memory and prevents the possibility of errors caused by copying the code from one place to another.
- A subprogram is a logical unit of program decomposition. Even if a segment is executed only once, it is useful to identify it in a subprogram for purposes of testing, documentation and readability.
- A subprogram can also be used as a physical unit of program decomposition, that is, as a unit of compilation. In Fortran, subprograms (called *subroutines*) are the only units of decomposition and compilation. Modern languages use the module, a group of declarations and subprograms, as the unit of physical decomposition (Chapter 13).

A subprogram consists of:

- A declaration which defines the interface to the subprogram. The subprogram declaration includes the name of the subprogram, the list of parameters (if any)[1] and the type of the value returned (if any).
- Local declarations which are accessible only within the body of the subprogram.
- A sequence of executable statements.

The local declarations and the executable statements form the *body* of the subprogram.

Subprograms that return a value are called *functions*; those that do not are called *procedures*. C does not have a separate syntax for procedures; instead you must write a function that returns void which is a type with no values:

[1] As a point of syntax, subprograms without parameters are usually declared without a parameter list (Ada, Pascal) or with an empty list (C++). C uses the explicit keyword void to indicate an absence of parameters.

```
void proc(int a, float b);
```
C

Such a function has the same properties as a procedure in other languages, so we will use the term procedure even when discussing C.

A procedure is invoked by a *call* statement. In Fortran, there is a special syntax:

```
call proc(x,y)
```
Fortran

while in other languages you simply write the name of the procedure followed by the actual parameters:

```
proc(x,y);
```
C

The semantics of a procedure call is as follows: the current sequence of instructions is suspended; the sequence of instructions within the procedure body is executed; upon completing the procedure body, the execution continues with the first statement following the procedure call. This description ignores parameter passing and scopes which will be the subject of extensive discussion in the next sections.

Since a function returns a value, the function declaration must specify the type of the returned value. In C, the type of a function precedes the function declaration:

```
int func(int a, float b);
```
C

while Ada uses a distinctive syntax for functions:

```
function Func(A: Integer; B: Float) return Integer;
```
Ada

A function call appears not as a statement, but as an element of an *expression*:

```
a = x + func(r,s) + y;
```
C

The result type of the function must be consistent with the type expected in the expression. Note that C does implicit type conversions in many cases, while in Ada the result type must exactly match the context. The meaning of a function call is similar to that of a procedure call: the evaluation of the expression is suspended; the instructions of the function body are executed; the returned value is then used to continue the evaluation of the expression.

The term function is actually very inappropriate to use in the context of ordinary programming languages. When used in mathematics, a function is just a mapping from one set of values to another. To use the technical term, a mathematical function is *referentially transparent*, because its "computation" is transparent to the point at which it is "called". If you have a value 3.6 and you ask for the value of $\sin(3.6)$, you will get the same unique result every time that the function appears in an equation. In programming, a function can perform an arbitrary computation including input-output or modification of global data structures:

C

```
int x,y,z;
int func(void)
{
   y = get( );              /* Modifies a global variable */
   return x*y;              /* Value based on global variable */
}

z = x + func(void) + y;
```

If the optimizer rearranged the order of the computation so that x+y were computed before the function call, a different result will be obtained, because the function modifies the value of y.

Since all C subprograms are functions, C programming style extensively uses return values in non-mathematical situations like input-output subprograms. This is acceptable provided that the possible difficulties with order dependencies and optimization are understood. Programming language research has developed exciting languages that are based on the mathematically correct concept of functions (see Chapter 16).

7.2 Parameters

In the previous section, we defined subprograms as segments of code that may be repeatedly invoked. Almost always, each invocation will require that the code in the subprogram body be executed using different data. The way to influence the execution of a subprogram body is to "pass" it the data that it needs. Data is passed to a subprogram in the form of a sequence of values called *parameters*. The concept is taken from mathematics where a function is given a sequence of *arguments*:[2] $\sin(2\pi r)$.

There are two concepts that must be clearly distinguished:

- A *formal parameter* is a declaration that appears in the declaration of the subprogram. The computation in the body of the subprogram is written in terms of formal parameters.
- An *actual parameter* is a value that the calling program sends to the subprogram.

In the following example:

C

```
int i, j;
char a;
void p(int a, char b)
{
   i = a + (int) b;
}
```

[2]Mathematical terminology uses *argument* for the value passed to a function, while *parameter* is usually used for values that are constant for a specific problem! We will, of course, use the programming terminology.

```
p(i, a);
p(i+j, 'x');
```

the formal parameters of the subprogram p are a and b, while the actual parameters of the first call are i and a, and of the second call, i+j and 'x'.

There are several important points that can be noted from the example. The first is that since the actual parameters are values, they can be constants or expressions, and not just variables. In fact, even when a variable is used as a parameter, what we really mean is "the current value stored in the variable". Secondly, the *name space* of each subprogram is distinct. The fact that the first formal parameter is called a is not relevant to the rest of the program, and it could be renamed, provided, of course, that all occurrences of the formal parameter in the subprogram body are renamed. The variable a declared outside the subprogram is totally independent from the variable of the same name declared within the subprogram. In Section 7.7, we will explore in great detail the relationship between variables declared in different subprograms.

Named parameter associations

Normally the actual parameters in a subprogram call are just listed and the matching with the formal parameters is done by position:

Ada

```
procedure Proc(First: Integer; Second: Character);
Proc(24, 'X');
```

However, in Ada it is possible to use named association in the call, where each actual parameter is preceded by the name of the formal parameter. The order of declaration of the parameters need not be followed:

Ada

```
Proc(Second => 'X', First => 24);
```

This is commonly used together with default parameters, where parameters that are not explicitly written receive the default values given in the subprogram declaration:

Ada

```
procedure Proc(First: Integer := 0; Second: Character := '*');
Proc(Second => 'X');
```

Named association and default parameters are commonly used in the command languages of operating systems, where each command may have dozens of options and normally only a few parameters need to be explicitly changed. However, there are dangers with this programming style. The use of default parameters can make a program hard to read because calls whose syntax is different actually call the same subprogram. Named associations are problematic because they bind the subprogram declaration and the calls more tightly than is usually needed. If you use only positional parameters in calling subprograms from a library, you could buy a competing library and just recompile or link:

```
X := Proc_1(Y) + Proc_2(Z);
```
Ada

However, if you use named parameters, then you might have to do extensive modifications to your program to conform to the new parameter names:

```
X := Proc_1(Parm => Y) + Proc_2(Parm => Z);
```
Ada

7.3 Passing parameters to a subprogram

The definition of the mechanism for passing parameters is one of the most delicate and important aspects of the specification of a programming language. Mistakes in parameter passing are a major source of difficult bugs, so we will go into great detail in the following description.

Let us start from the definition we gave above: the value of the actual parameter is passed to the formal parameter. The formal parameter is just a variable declared within the subprogram, so the obvious mechanism is to copy the value of the actual parameter into the memory location allocated to the formal parameter. This mechanism is called *copy-in semantics* or *call-by-value*. Figure 7.1 demonstrates copy-in semantics, given the procedure definition:

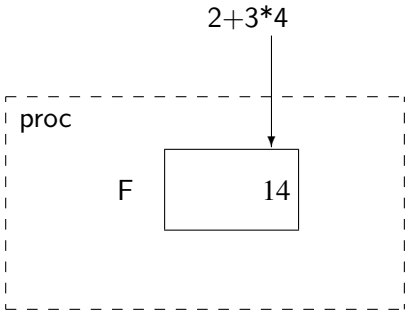

Figure 7.1: Copy-in semantics

```
procedure Proc(F: in Integer) is
begin
   ...;
end Proc;
```
Ada

and the call:

```
Proc(2+3*4);
```
Ada

The advantages of copy-in semantics are:

- Copy-in is the safest mechanism for parameter passing. Since only a copy of the actual parameter is passed, the subprogram cannot cause any damage to the actual parameter, which of course "belongs" to the calling program. If the subprogram modifies the formal parameter, only the copy is modified and not the original.
- Actual parameters can be constants, variables or expressions.
- Copy-in can be extremely efficient because once the initial overhead of the copy is done, all accesses to the formal parameter are to the local copy. As we shall see in Section 7.7, accesses to local variables are extremely efficient.

If copy-in semantics is so good, why are there other mechanisms? The reason is that we will often want to modify the actual parameter despite the fact that such modification is "unsafe":

- A function can only return a single result, so if the result of a computation is more complex, we may want to return several results. The way to do so is to provide a procedure with several actual parameters which can be assigned the results of the computation. Note that this situation can often be avoided by defining a function that returns a record as a result.
- Similarly, the purpose of the computation in the subprogram may be to modify data that is passed to it rather than to compute a result. This is common when a subprogram is maintaining a data structure. For example, a subprogram to sort an array does not compute a value; its only task is to modify the actual parameter. There is no point in sorting a copy of the actual parameter!
- A parameter may be so big that it is inefficient to copy. If copy-in is used for an array of 50,000 integers, there may simply not be enough memory available to make a copy, or the overhead of the copy may be excessive.

The first two situations can easily be solved using *copy-out semantics*. The actual parameter must be a variable, and the subprogram is passed the address of the actual parameter which it saves. A temporary local variable is used for the formal parameter, and a value must be assigned to the formal parameter[3] at least once during the execution of the subprogram. When the execution of the subprogram is completed, the value is copied into the variable pointed to by the saved address. Figure 7.2 shows copy-out semantics for the following subprogram:

Ada

```
procedure Proc(F: out Integer) is
begin
   F := 2+3*4;            -- Assign to copy-out parameter
end Proc;

A: Integer;
Proc(A);                  -- Call procedure with variable
```

[3] Ada 83 does not allow the subprogram to read the contents of the formal parameter. The more common definition of copy-out semantics, which is followed in Ada 95, allows normal computation on the (uninitialized) local variable.

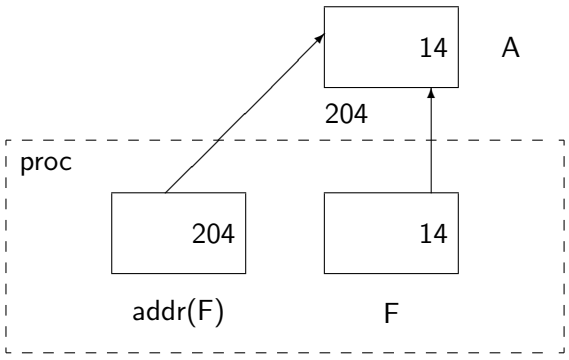

Figure 7.2: Copy-out semantics

When modification of the actual parameter is needed as in sort, *copy-in/out semantics* can be used: the actual parameter is copied into the subprogram when it is called and the final value is copied back upon completion.

However, copy-based parameter passing mechanisms cannot solve the efficiency problem caused by large parameters. The solution, which is known as *call-by-reference* or *reference semantics*, is to pass the address of the actual parameter and to access the parameter indirectly (Figure 7.3). Calling the subprogram is efficient because only a small, fixed-sized pointer is passed for each parameter; however, accessing the parameter can be inefficient because of the indirection.

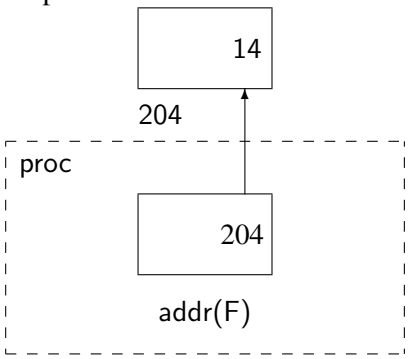

Figure 7.3: Reference semantics

In order to access the actual parameter, its address must be loaded and then an additional instruction is needed to load the value. Note that when using reference (or copy-out) semantics, the actual parameter must be a variable, not an expression, because a value will be assigned to it.

Another problem with call-by-reference is that it may result in *aliasing*: a situation in which the same variable is known by more than one name. In the following example, within the function f the variable global is also known by the alias *parm:

C

```
int global = 4;
int a[10];

int f(int *parm)
```

```
{
   *parm = 5;              /* Same variable as "global" */
   return 6;
}

x = a[global] + f(&global);
```

In the example, if the expression is evaluated in the order in which it is written, its value is a[4]+6, but because of aliasing, the value of the expression may be 6+a[5] if the compiler chooses to evaluate the function call before the array indexing. Aliasing is an important cause of non-portable behavior.

The real disadvantage of call-by-reference is that the mechanism is inherently unsafe. Suppose that for some reason the subprogram thinks that the actual parameter is an array whereas in fact it is just a single integer. This can cause an arbitrary amount of memory to be smeared, since the subprogram is working on the actual parameter, and not just on a local copy. This type of bug is extremely common, because the subprogram will typically have been written by a different programmer than the one calling the subprogram and misunderstandings always occur.

Safety of parameter passing can be improved by insisting on strong type checking which ensures that the types of the formal and actual parameters are compatible. Nevertheless, there is still room for misunderstanding between the programmer who wrote the subprogram and the programmer whose data is being modified. Thus we have an excellent parameter passing mechanism that is not always efficient enough (copy-in semantics), together with mechanisms that are necessary but unsafe (copy-out and reference semantics). The choice is complicated by constraints placed on the programmer by various programming languages. We will now describe the parameter passing mechanisms of several languages in detail.

Parameters in C and C++

C has only one parameter-passing mechanism, copy-in:

C

```
int i = 4;                 /* Global variables */

void proc(int i, float f)
{
   i = i + (int) f;        /* Local "i" */
}

proc(j, 45.0);             /* Function call */
```

In proc, the variable i that is modified is a local copy and not the global i.

In order to obtain the functionality of reference or copy-out semantics, a C programmer must resort to explicit use of pointers:

C

```
int i = 4;                  /* Global variables */

void proc(int *i, float f)
{
   *i = *i + (int) f;       /* Indirect access */
}

proc(&i, 45.0);             /* Address operator needed */
```

After executing proc, the value of the global variable i will be modified. The requirement that pointers be used for reference semantics is unfortunate, because beginning programmers must learn this relatively advanced concept at an early stage of their studies.

C++ has corrected this problem so that true call-by-reference is available using *reference parameters*:

C++

```
int i = 4;                  // Global variables

void proc(int & i, float f)
{
   i = i + (int) f;         // By reference access
}

proc(i, 45.0);              // No address operator needed
```

Note that the programming style is natural and does not use pointers artificially. This improvement in the parameter passing mechanism is so important that it justifies using C++ as a replacement for C.

You will often want to use pointers in C or references in C++ to pass large data structures. Of course, unlike copy-in parameters, there is a danger of accidental modification of the actual parameter. Read-only access to a parameter can be specified by declaring them const:

```
void proc(const Car_Data & d)
{
   d.fuel = 25;             // Error, cannot modify const
}
```

const declarations should be used whenever possible both to clarify the meaning of parameters to readers of the program, and to catch potential bugs.

Another problem with parameters in C is that arrays cannot be parameters. If an array must be passed, the address of the first element of the array is passed, and the procedure has the responsibility for correctly accessing the array. As a convenience, using an array name as a parameter is automatically considered to be the use of a pointer to the first element:

C

```
int b[50];                  /* Array variable */

void proc(int a[])          /* "Array parameter" */
{
   a[100] = a[200];         /* How many components ? */
}

proc(&b[0]);                /* Address of first element */
proc(b);                    /* Address of first element */
```

C programmers quickly get used to this but it is a source of confusion and bugs. The problem is that since the parameter is actually a pointer to a single element, *any* pointer to a variable of a similar type is accepted:

C

```
int i;
void proc(int a[]);         /* "Array parameter" */
proc(&i);                   /* Any pointer to integer is OK !! */
```

Finally, in C no type checking is done between files so that it is possible to declare:

C

```
void proc(float f) { ... }  /* Procedure definition */
```

in one file and:

C

```
void proc(int i);           /* Procedure declaration */
proc(100);
```

in another file, and then spend a month looking for the bug.

The C++ language requires that parameter type checking be performed. However, the language does not require that implementations include a library facility as in Ada (see Section 13.3) that can ensure type checking across separately compiled files. C++ compilers implement type checking by cooperation with the linker: parameter types are encoded in the external name of the subprogram (a process called *name mangling*), and the linker will make sure that calls are linked only to subprograms with the correct parameter signature.[4] Unfortunately, this method cannot catch all type mismatches.

Parameters in Pascal

In Pascal, parameters are passed by value unless reference semantics is explicitly requested:

[4]For details, see Section 7.2c of the Annotated Reference Manual.

Pascal

```
procedure proc(P_Input: Integer; var P_Output: Integer);
```

The keyword var indicates that the following parameter is called by reference, otherwise call-by-value is used even if the parameter is very large. Parameters can be of any type including arrays, records or other complex data structures. The one limitation is that the result type of a function must be a scalar. The types of actual parameters are checked against the types of the formal parameters.

As we discussed in Section 5.4, there is a serious problem in Pascal because the array bounds are considered part of the type. The Pascal standard defines *conformant array parameters* to solve this problem.

Parameters in Ada

Ada takes a novel approach in defining parameter passing in terms of intended use rather than in terms of the implementation mechanism. For each parameter you must explicitly choose one of three possible *modes*:

in The parameter may be read but not written (default).

out The parameter may be written but not read.

in out The parameter may be both read and written.

For example:

Ada

```
procedure Put_Key(Key: in Key_Type);
procedure Get_Key(Key: out Key_Type);
procedure Sort_Keys(Keys: in out Key_Array);
```

In the first procedure, the parameter Key must be read so that it can be "put" into a data structure (or output device). In the second, a value is obtained from a data structure and upon completion of the procedure, the value is assigned to the parameter. The array Keys to be sorted must be passed as in out, because sorting involves both reading and writing the data of the array.

Ada restricts parameters of a function to be of mode in only. This does not make Ada functions referentially transparent because there is still no restriction on accessing global variables, but it can help the optimizer to improve the efficiency of expression evaluation.

Despite the fact that the modes are not defined in terms of implementation mechanisms, the Ada language does specify some requirements on the implementation. Parameters of elementary type (numbers, enumerations and pointers) must be implemented by copy semantics: copy-in for in parameters, copy-out for out parameters, and copy-in/out for in out parameters. The implementation of modes for composite parameters (arrays and records) is not specified, and a compiler may choose whichever mechanism it prefers. This introduces the possibility that the correctness of an

Ada program depends on the implementation-chosen mechanism, so such programs are simply not portable.[5]

Strong type checking is done between formal and actual parameters. The type of the actual parameter must be the same as that of the formal parameter; no implicit type conversion is ever performed. However, as we discussed in Section 5.4, the subtypes need not be identical as long as they are compatible; this allows an arbitrary array to be passed to an unconstrained formal parameter.

Parameters in Fortran

We will briefly touch on parameter passing in Fortran because it can cause spectacular bugs. Fortran can pass only scalar values; the interpretation of a formal parameter as an array is done by the called subroutine. Call-by-reference is used for all parameters. Furthermore, each subroutine is compiled independently and no checking is done for compatibility between the subroutine declaration and its call.

The language specifies that if a formal parameter is assigned to, the actual parameter must be a variable, but because of independent compilation this rule cannot be checked by the compiler. Consider the following example:

Fortran

```
Subroutine Sub(X, Y)
Real X,Y
X = Y
End

Call Sub(−1.0, 4.6)
```

The subroutine has two parameters of type Real. Since reference semantics is used, Sub receives pointers to the two actual parameters and the assignment is done directly on the actual parameters (Figure 7.4). The result is that the memory location storing the value −1.0 is modified! There

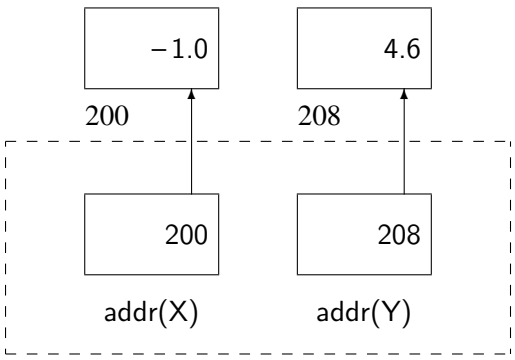

Figure 7.4: Smearing a constant in Fortran

[5] Ada 95 requires that certain categories of parameters be passed by reference; these include task types and tagged types (Section 14.5).

is literally no way to "debug" this bug, since debuggers only allow you to examine and trace variables, not constants. The point of the story is that correct matching of actual and formal parameters is a cornerstone of reliable programming.

7.4 Block structure

A *block* is an entity consisting of declarations and executable statements. A similar definition was given for a subprogram body and it is more precise to say that a subprogram body *is* a block. Blocks in general and procedures in particular can be nested within one another. This section will discuss the relationships among nested blocks.

Block structure was first defined in the Algol language which includes both procedures and unnamed blocks. Pascal contains nested procedures but not unnamed blocks; C contains unnamed blocks but not nested procedures; and Ada returns to support both.

Unnamed blocks are useful for restricting the scope of variables by declaring them only when needed, instead of at the beginning of a subprogram. The trend in programming is to reduce the size of subprograms, so the use of unnamed blocks is less useful than it used to be.

Nested procedures can be used to group statements that are executed at more than one location within a subprogram, but refer to local variables and so cannot be external to the subprogram. Before modules and object-oriented programming were introduced, nested procedures were used to structure large programs, but this introduces complications and is not recommended.

The following is an example of a complete Ada program:

Ada

```
procedure Main is
   Global: Integer;

   procedure Proc(Parm: in Integer) is
      Local: Integer;
   begin
      Global := Local + Parm;
   end Proc;

begin -- Main
   Global := 5;
   Proc(7);
   Proc(8);
end Main;
```

An Ada program is a *library* procedure, that is, a procedure that is not enclosed within any other entity and hence can be stored in the Ada library. The procedure begins with a procedure declaration for Main,[6] which serves as a definition of the interface to the procedure, in this case the

[6] Unlike C, the main procedure need not be called main.

external name of the program. Within the library procedure there are two declarations: a variable Global and a procedure Proc. Following the declarations is the sequence of executable statements for the main procedure. In other words, the procedure Main consists of a procedure declaration and a block. Similarly, the local procedure Proc consists of a procedure declaration (the procedure name and the parameters) and a block containing variable declarations and executable statements. Proc is said to be *local* to Main or *nested* within Main.

Each declaration has associated with it three properties:[7]

Scope The scope of a variable is the segment of the program within which it is defined.

Visibility A variable is visible within some subsegment of its scope if it can be directly accessed by name.

Lifetime The lifetime of a variable is the interval during the program's execution when memory is assigned to the variable.

Note that lifetime is a dynamic property of the run-time behavior of a program, while scope and visibility relate solely to the static program text.

Let us demonstrate these abstract definitions on the example above. The scope of a variable begins at the point of declaration and ends at the end of the block in which it is defined. The scope of Global includes the entire program while the scope of Local is limited to a single procedure. The formal parameter Parm is considered to be like a local variable and its scope is also limited to the procedure.[8]

The visibility of each variable in this example is identical to its scope: each variable can be directly accessed in its entire scope. Since the scope and visibility of the variable Local is limited to the local procedure, the following is not allowed:

Ada

```
begin -- Main
   Global := Local + 5;    -- Local is not in scope here
end Main;
```

However, the scope of Global includes the local procedure so the access within the procedure is correct:

Ada

```
procedure Proc(Parm: in Integer) is
   Local: Integer;
begin
   Global := Local + Parm;      -- Global is in scope here
end Proc;
```

The lifetime of a variable is from the beginning of the execution of its block until the end of the execution of its block. The block of the procedure Main is the entire program so Global exists for

[7] To keep the discussion concrete, it will be given in terms of variables even though the concepts are more general.

[8] Ada allows named parameter associations, so this statement is not completely precise.

the duration of the execution of the program. Such a variable is called *static*: once it is allocated it lives until the end of the program. The local variable has two lifetimes corresponding to the two calls to the local procedure. Since these intervals do not overlap, the variable may be allocated at a different location each time it is created. Local variables are called *automatic* because they are automatically allocated when the procedure is called (the block is entered), and released when the procedure returns (the block is left).

Hiding

Suppose that a variable name that is used in the main program is repeated in a declaration in a local procedure:

Ada

```
procedure Main is
   Global: Integer;
   V: Integer;                    -- Declaration in Main

   procedure Proc(Parm: in Integer) is
      Local: Integer;
      V: Integer;                 -- Declaration in Proc
   begin
      Global := Local + Parm + V; -- Which V is used ?
   end Proc;

begin -- Main
   Global := Global + V;          -- Which V is used ?
end Main;
```

In this case, the local declaration is said to *hide* the global declaration. Within the procedure, any reference to V is a reference to the locally declared variable. In technical terms, the scope of the global V extends from the point of declaration to the end of Main, but its visibility does not include local procedure Proc.[9]

Hiding of variable names by inner declarations is convenient in that the programmer can reuse natural names like Current_Key and not have to invent strange-sounding names. Furthermore, it is always possible to add a global variable without worrying that this will clash with some local variable name used by one of the programmers on your team. The disadvantage is that a variable name could be accidentally hidden, especially if large include-files are used to centralize global declarations, so it is probably better to avoid hiding variable names. However, there is no objection to reusing a name in different scopes since there is no way of accessing both variables simultaneously, regardless of whether the names are the same or different:

[9] In Ada (but not in Pascal) the hidden variable is accessible using the syntax Main.V. Similarly, in C++ (but not in C), ::V can be used to access a hidden global variable.

Ada

```
procedure Main is
   procedure Proc_1 is
      Index: Integer;              -- One scope
      ...
   end Proc_1;
   procedure Proc_2 is
      Index: Integer;              -- Non-overlapping scope
      ...
   end Proc_2;
begin -- Main
   ...
end Main;
```

Depth of nesting

There is no conceptual limit to the depth of nesting, though a compiler may arbitrarily limit the depth. Scope and visibility are determined by applying the rules given above: a variable's scope is from its point of declaration to the end of the block, and its visibility is the same unless hidden by an inner declaration. For example:

Ada

```
procedure Main is
   Global: Integer;

   procedure Level_1 is
      Local: Integer;       -- Outer declaration of Local

      procedure Level_2 is
         Local: Integer;    -- Inner declaration of Local
      begin -- Level_2
         Local := Global; -- Inner Local hides outer Local
      end Level_2;

   begin -- Level_1
      Local := Global;      -- Only outer Local in scope
      Level_2;
   end Level_1;

begin -- Main
   Level_1;
   Level_2;                 -- Error, procedure not in scope
end Main;
```

The scope of the variable Local defined in procedure Level_1 extends until the end of the procedure, but it is hidden within procedure Level_2 by the declaration of the same name.

The procedure declarations themselves are considered to have scope and visibility similar to variable declarations. Thus the scope of Level_2 is from its declaration in Level_1 until the end of Level_1. This means that Level_1 can *call* Level_2 even though it cannot access variables within Level_2. On the other hand, Main cannot call Level_2 directly, since it cannot access declarations that are local to Level_1.

Note the potential for confusion since the variable Local accessed by the statement in Level_1 is declared *further* away in the program text than the occurrence of Local enclosed within Level_2. If there were a lot of local procedures, it might be difficult to find the correct declaration. To prevent confusion, it is best to limit the depth of nesting to two or three levels below the main program.

Advantages and disadvantages of block structure

The advantage of block structure is that it provides an easy and efficient method of decomposing a procedure. If you avoid excessive nesting and hidden variables, block structure can be used to write reliable programs since related local procedures can be kept together. Block structuring is especially important when complex computations are being done:

Ada

```
procedure Proc(...) is
      -- Lots of declarations
begin
   -- Long computation 1
   if N ¡ 0 then
         -- Long computation 2 version 1
   elsif N = 0 then
         -- Long computation 2 version 2
   else
         -- Long computation 2 version 3
   end if;
   -- Long computation 3
end Proc;
```

In this example, we would like to avoid writing Long computation 2 three times and instead make it an additional procedure with a single parameter:

Ada

```
procedure Proc(...) is
      -- Lots of declarations
      procedure Long_2(I: in Integer) is
      begin
         -- Access declarations in Proc
      end Long_2;
begin
```

```
    -- Long computation 1
    if N ¡ 0 then Long_2(1);
    elsif N = 0 then Long_2(2);
    else Long_2(3);
    end if;
    -- Long computation 3
end Proc;
```

However, it would be extremely difficult to make Long_2 an independent procedure because we might have to pass dozens of parameters so that it could access local variables. If Long_2 is nested, it needs just the one parameter, and the other declarations can be directly accessed according to normal scope and visibility rules.

The disadvantages of block structure become apparent when you try to program a large system in a language like standard Pascal that has no other means of program decomposition:

- Small procedures receive excessive "promotions". Suppose that a procedure to convert decimal digits to hexadecimal digits is used in many deeply-nested procedures. That utility procedure must be defined in some common ancestor. Practically, large block-structured programs tend to have many small utility procedures written at the highest level of declaration. This makes the program text awkward to work with because it is difficult to locate a specific procedure.

- Data security is compromised. Every procedure, even those declared deeply nested in the structure, can access global variables. In a large program being developed by a team, this makes it likely that errors made by one junior team member can cause obscure bugs. The situation is analogous to a company where every employee can freely examine the safe in the boss's office, but the boss has no right to examine the file cabinets of junior employees!

These problems are so serious that every commercial Pascal implementation defines a (non-standard) module structure to enable large projects to be constructed. In Chapter 13 we will discuss in detail constructs that are used for program decomposition in modern languages like Ada and C++. Nevertheless, block structure remains an important tool in the detailed programming of individual modules.

It is also important to understand block structure because programming languages are implemented using stack architecture, which directly supports block structure (Section 7.6).

7.5 Recursion

Most (imperative) programming is done using *iteration*, that is loops; however, *recursion*, the definition of an object or computation in terms of itself, is a more primitive mathematical concept, and is also a powerful, if often under-used, programming technique. Here we will survey how to program recursive subprograms.

The most elementary example of recursion is the factorial function, defined mathematically as:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \end{aligned}$$

This definition translates immediately into a program that uses a recursive function:

C

```
int factorial(int n)
{
    if (n == 0) return 1;
    else return n * factorial(n – 1);
}
```

What properties are required to support recursion?

- The compiler must emit *pure code*. Since the same sequence of machine instructions are used to execute each call to factorial, the code must not modify itself.
- During run-time, it must be possible to allocate an arbitrary number of memory cells for the parameters and local variables.

The first requirement is fulfilled by all modern compilers. Self-modifying code is an artifact of older programming styles and is rarely used. Note that if a program is to be stored in read-only memory (ROM), by definition it cannot modify itself.

The second requirement arises from consideration of the lifetime of the local variables. In the example, the lifetime of the formal parameter n is from the moment that the procedure is called until it is completed. But before the procedure is completed, another call is made and that call requires that memory be allocated for the new formal parameter. To compute factorial(4), a memory location is allocated for 4, then 3 and so on, five locations altogether. The memory cannot be allocated before execution, because the amount depends on the run-time parameter to the function. Section 7.6 shows how this allocation requirement is directly supported by the stack architecture.

At this point, most programmers will note that the factorial function could be written just as easily and far more efficiently using iteration:

C

```
int factorial(int n)
{
    int i = n;
    result = 1;
    while (i != 0) {
        result = result * i;
        i−−;
    }
    return result;
}
```

So why use recursion? The reason is that many algorithms can be elegantly and reliably written using recursion while an iterative solution is difficult to program and prone to bugs. Examples are the Quicksort algorithm for sorting and data structure algorithms based on trees. The language concepts discussed in Chapters 16 and 17 (functional and logic programming) use recursion exclusively instead of iteration. Even when using ordinary languages like C and Ada, recursion should probably be used more often than it is because of the concise, clear programs that result.

7.6 Stack architecture

A *stack* is a data structure that stores and retrieves data in a Last-In, First-Out (LIFO) order. LIFO constructions exist in the real world such as a stack of plates in a cafeteria, or a pile of newspapers in a store. A stack may be implemented using either an array or a list (Figure 7.5). The advantage of the list is that it is unbounded and its size is limited only by the total amount of

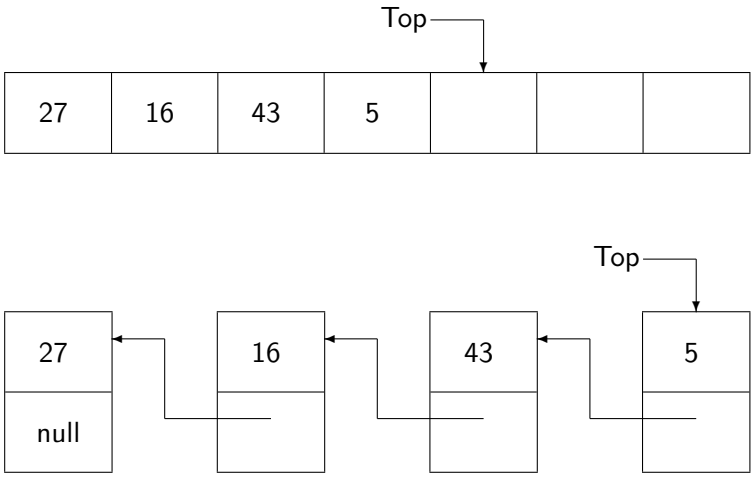

Figure 7.5: Stack implementation

available memory. Arrays are much more efficient and are implicitly used in the implementation of programming languages.

In addition to the array (or list), a stack contains an additional piece of data—the *top-of-stack pointer*. This is an index to the first available empty position in a stack. Initially, a variable top will point to the first position in the stack. The two possible operations on a stack are push and pop. push is a procedure that receives an element as a parameter, which it places on the top of the stack, incrementing top. pop is a function that returns the top element in the stack, decrementing top to indicate that that position is the new empty position.

The following C program implements a stack of integers as an array:

C

```
#define Stack_Size 100
int stack[Stack_Size];
```

```
int top = 0;

void push(int element)
{
    if (top == Stack_Size) /* Stack overflow, do something! */
    else stack[top++] = element;
}

int pop(void)
{
    if (top == 0) /* Stack underflow, do something! */
    else return stack[--top];
}
```

A stack can underflow if we try to pop from an empty stack, and it can overflow if we try to push onto a full stack. Underflow is always due to a programming error since you store something on a stack if and only if you intend to retrieve it later on. Overflow can occur even in a correct program if the amount of memory is not sufficient for the computation.

Stack allocation

How is a stack used in the implementation of a programming language? A stack is used to store information related to a procedure call, including the local variables and parameters that are automatically allocated upon entry to the procedure and released upon exit. The reason that a stack is the appropriate data structure is that procedures are entered and exited in a LIFO order, and any accessible data belongs to a procedure that occurs earlier in the chain of calls.

Consider a program with local procedures:

Ada

```
procedure Main is
    G: Integer;

    procedure Proc_1 is
        L1: Integer;
    begin ... end Proc_1;

    procedure Proc_2 is
        L2: Integer;
    begin ... end Proc_2;

begin
    Proc_1;
    Proc_2;
end Main;
```

When the Main begins executing, memory must be allocated for G. When Proc_1 is called, additional memory must be allocated for L1 without releasing the memory for G (Figure 7.6(a)). The memory for L1 is released before memory is allocated for L2, since Proc_1 terminates before

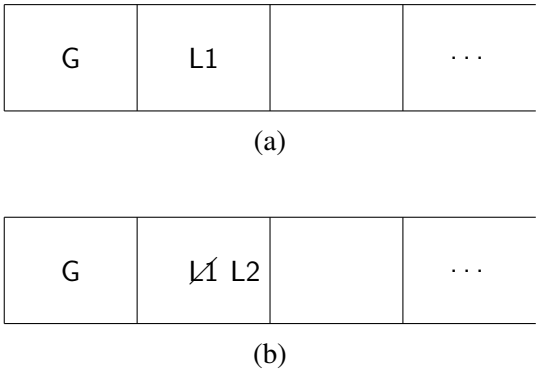

Figure 7.6: Allocating memory on a stack

Proc_2 is called (Figure 7.6(b)). In general, no matter how procedures call each other, the first memory element to be released is the last one allocated, so memory for variables and parameters can be allocated on a stack.

Consider now nested procedures:

Ada

```
procedure Main is
   G: Integer;

   procedure Proc_1(P1: Integer) is
      L1: Integer;

      procedure Proc_2(P2: Integer) is
         L2: Integer;
      begin
         L2 := L1 + G + P2;
      end Proc_2;

   begin -- Proc_1
      Proc_2(P1);
   end Proc_1;

begin -- Main
   Proc_1(G);
end Main;
```

Proc_2 can only be called from within Proc_1. This means that Proc_1 has not terminated yet, so its memory has not been released and the memory assigned to L1 must still be allocated (Fig-

ure 7.7). Of course, Proc_2 terminates before Proc_1 which in turn terminates before Main, so memory can be freed using the pop operation on the stack.

Figure 7.7: Nested procedures

Activation records

The stack is actually used to support the entire procedure call and not just the allocation of local variables. The segment of the stack associated with each procedure is called the *activation record* for the procedure. In outline,[10], a procedure call is implemented as follows (see Figure 7.8):

1. The actual parameters are pushed onto the stack. They can be accessed as offsets from the start of the activation record.
2. The *return address* is pushed onto the stack. The return address is the address of the statement following the procedure call.
3. The top-of-stack index is incremented by the total amount of memory required to hold the local variables.
4. A jump is made to the procedure code.

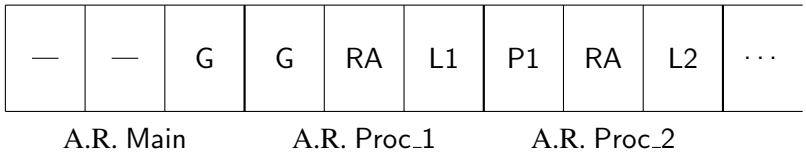

Figure 7.8: Activation records

Upon completion of the procedure the above steps are reversed:

1. The top-of-stack index is decremented by the amount of memory allocated for the local variables.
2. The return address is popped and the instruction pointer reset.
3. The top-of-stack index is decremented by the amount of memory allocated for the actual parameters.

[10]See Section 7.8 for more details.

While this code may seem complicated, it can actually be done very efficiently on most computers. The amount of memory needed for variables, parameters and call overhead is known at compile-time, and the above processing just requires modification of the stack index by a constant.

Accessing values on the stack

In a true stack, the only permissible operations are push and pop. The execution stack we have described is a more complex structure because we want to be able to efficiently access not only the most recent value pushed, but also all the local variables and all the parameters. One possibility would be to access these data relative to the top-of-stack index:

```
stack[top − 25];
```
C

However, the stack may hold other data besides that associated with a procedure call (such as temporary variables, see Section 4.7), so it is customary to maintain an additional index called the *bottom pointer* which points to the start of the activation record (see Section 7.7). Even if the top-of-stack index varies during the execution of the procedure, all the data in the activation record can be accessed at fixed offsets from the bottom pointer.

Parameters

There are two methods for implementing the passing of parameters. The simpler method is just to push the parameters themselves (whether values or references) onto the stack. This method is used in Pascal and Ada because in those languages the number and type of each parameter is known at compilation time. From this information, the offset of each parameter relative to the beginning of the activation record can be computed at compile-time, and each parameter can be accessed at this fixed offset from the bottom pointer index:

```
load    R1,bottom-pointer
add     R1,#offset-of-parameter
load    R2,(R1)         Load value whose address is in R1
```

If the bottom pointer is kept in a register, this code can usually be collapsed into a single instruction. When leaving the subprogram, *stack clean-up* is done by having the subprogram reset the stack pointer so that the parameters are effectively no longer on the stack.

There is a problem using this method in C, because C allows a procedure to have a variable number of arguments:

```
void proc(int num_args, ...);
```
C

Since the subprogram does not know how many parameters there are, it cannot clean-up the stack. The responsibility for stack clean-up is thus shifted to the caller which does know how many parameters were passed. This causes some memory overhead because the clean-up code is duplicated at *every* call instead of being common to all calls.

When the number of parameters is not known, an alternative method of parameter passing is to store the actual parameters in a separate block of memory, and then to pass the address of this block on the stack. An additional indirection is required to access a parameter, so this method is less efficient than directly pushing parameters on the stack.

Note that it may not be possible to store a parameter directly on the stack. As you will recall, a formal parameter in Ada can be an unconstrained array type whose bounds are not known at compilation time:

Ada

```
procedure Proc(S: in String);
```

Thus the actual parameter cannot be pushed directly onto the stack. Instead a dope vector (Figure 5.5) which contains a pointer to the array parameter is placed on the stack.

Recursion

The stack architecture directly supports recursion because each call to a procedure automatically allocates a new copy of the local variables and parameters. For example, each recursive call of the function for factorial needs one memory word for the parameter and one memory word for the return address. The higher overhead of recursion relative to iteration comes from the extra instructions involved with the procedure entry and exit. Some compilers will attempt an optimization called *tail-recursion* or *last-call* optimization. If the only recursive call in a procedure is the last statement in the procedure, it is possible to automatically translate the recursion into iteration.

Stack size

If recursion is not used, the total stack usage can theoretically be computed before execution by adding the activation record requirements for each possible chain of procedure calls. Even in a complex program, it should not be hard to make a reasonable estimate of this figure. Add a few thousand spare words and you have computed a stack size that will probably not overflow.

However, if recursion is used, the stack size is theoretically unbounded at run-time:

C

```
i = get( );
j = factorial(i);
```

In the Exercises, we describe Ackermann's function which is unconditionally guaranteed to overflow any stack you allocate! In practice, it is usually not difficult to make an estimate of stack size even when recursion is used. Suppose the size of an activation record is about 10 and the depth of recursion no more than a few hundred. Adding an extra 10K to the stack will more than suffice.

Readers who have studied data structures will know that recursion is convenient to use on tree-structured algorithms like Quicksort and priority queues. The depth of recursion in tree algorithms is roughly $\log_2$ of the size of the data structure. For practical programs this bounds the depth of recursion to 10 or 20 so there is very little danger of stack overflow.

Whether recursion is used or not, the nature of the system will dictate the treatment of potential stack overflow. A program might completely ignore the possibility and accept the fact that in extreme circumstances the program will crash. Another possibility is to check the stack size before each procedure call, but this might be too inefficient. A compromise solution would be to check the stack size periodically and take some action if it fell below some threshold, say 1000 words.

7.7 More on stack architecture

Accessing variables at intermediate levels

We have discussed how local variables are efficiently accessed at fixed offsets from the bottom pointer of an activation record. Global data, that is data declared in the main program, can also be accessed efficiently. The easiest way to see this is to imagine that global data is considered "local" to the main procedure; thus memory for the global data is allocated at the main procedure entry, that is, at the beginning of the program. Since this location is known at compile-time, or more exactly when the program is linked, the actual address of each element is known either directly or as an offset from a fixed location. In practice, global data is usually allocated separately (see Section 8.5), but in any case the addresses are fixed.

Variables at intermediate levels of nesting are more difficult to access:

Ada

```
procedure Main is
   G: Integer;

   procedure Proc_1 is
      L1: Integer;

      procedure Proc_2 is
         L2: Integer;
      begin L2 := L1 + G; end Proc_2;
      procedure Proc_3 is
         L3: Integer;
      begin L3 := L1 + G; Proc_2; end Proc_3;

   begin -- Proc_1
      Proc_3;
   end Proc_1;

begin -- Main
   Proc_1;
end Main;
```

We have seen that accessing the local variable L3 and the global variable G is easy and efficient,

but how can L1 be accessed in Proc_3? The answer is that the value of the bottom pointer is stored at procedure entry and is used as a pointer to the activation record of the enclosing procedure Proc_1. The bottom pointer is stored at a known location and can be immediately loaded, so the overhead is an additional indirection.

If deeper nesting is used, each activation record contains a pointer to the previous one. These pointers to activation records form the *dynamic chain* (Figure 7.9). To access a *shallow variable* (one that is less deeply nested), instructions have to be executed to "climb" the dynamic chain. This potential inefficiency is the reason that accessing intermediate variables in deeply nested procedures is discouraged. Accessing the immediately previous level requires just one indirection and an occasional deep access should not cause any trouble, but a loop statement should not contain statements that reach far back in the chain.

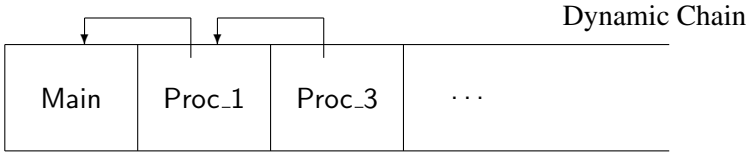

Figure 7.9: Dynamic chain

Calling shallow procedures

Accessing intermediate variables is actually more complicated than the discussion above would indicate because a procedure is allowed to call other procedures that are at the same level of nesting or lower. In the example, Proc_3 calls Proc_2. The activation record for Proc_2 will store the bottom pointer of Proc_3 so that it can be restored, but the variables of Proc_3 are *not* accessible in Proc_2 by the rules of scope.

Somehow the program must be able to identify the *static chain*, the link of activation records that defines the static context of the procedure according to the rules of scope, as opposed to the dynamic chain of procedure calls at execution time. As an extreme example, consider a recursive procedure: there may be dozens of activation records in the dynamic chain (one for each recursive call), but the static chain will consist only of the current record and the record for the main procedure.

One solution is to store the static level of nesting of each procedure in the activation record, because the compiler knows what level is needed for each access. In the example, if the main program is level 0, Proc_2 and Proc_3 are both at level 2. When searching up the dynamic chain, the level of nesting must decrease by one to be considered part of the static chain, thus the record for Proc_3 is skipped over and the next record, the one for Proc_1 at level 1, is used to obtain a bottom index.

Another solution is to explicitly include the static chain on the stack. Figure 7.10 shows the static chain just after Proc_3 calls Proc_2. Before the call, the static chain is the same as the dynamic chain while after the call, the static chain is shorter and contains just the main procedure and Proc_2.

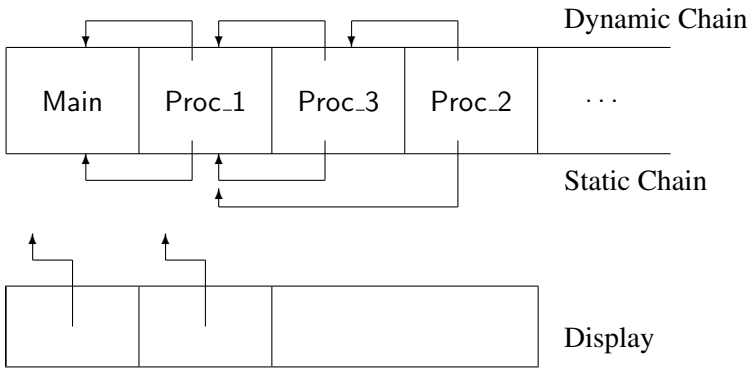

Figure 7.10: Variables at intermediate levels

The advantage of using an explicit static chain is that a static chain is often shorter than a dynamic chain (again think of a recursive procedure as an extreme case). However, we still have to do the search for each access of an intermediate variable. A solution that can be more efficient is to use a *display* which is an array that holds the current static chain, indexed by the nesting level (Figure 7.10). Thus to access a variable at an intermediate level, the nesting level is used as an index to obtain a pointer to the correct activation record, then the bottom pointer is obtained from the record and finally the offset is added to obtain the variable address. The disadvantage of a display is that additional overhead is needed to update the display at procedure entry and exit.

The potential inefficiencies of accessing intermediate variables should not deter the use of nested procedures, but programmers should carefully consider such factors as the depth of nesting and the trade-offs between using parameters as opposed to direct access to variables.

7.8 * Implementation on the 8086

To give a more concrete idea of how a stack architecture is implemented, we describe the actual machine code for procedure entry and exit on the Intel 8086 series of processors. The example program is:

Ada

```
procedure Main is
   Global: Integer;

   procedure Proc(Parm: in Integer) is
      Local1, Local2: Integer;
   begin
      Local2 := Global + Parm + Local1;
   end Proc;

begin
   Proc(15);
end Main;
```

The 8086 has built-in push and pop instructions which assume that the stack grows from higher to lower addresses. Two registers are dedicated to stack operations: the sp register which points to the "top" element in the stack, and the bp register which is the bottom pointer that identifies the location of the start of the activation record.

To call the procedure the parameter is pushed onto the stack and the call instruction executed:[11]

```
mov     ax,#15      Load value of parameter
push    ax          Store parameter on stack
call    Proc        Call the procedure
```

Figure 7.11 shows the stack after executing these instructions—the parameter and the return address have been pushed onto the stack.

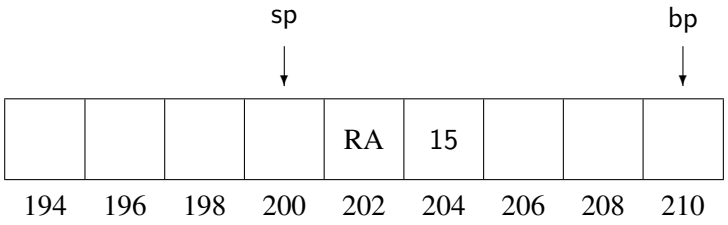

Figure 7.11: Stack before procedure entry

The next instructions are part of the code of the procedure and are executed at procedure entry; they store the old bottom pointer (the dynamic link), set up the new bottom pointer and allocate memory for the local variable by decrementing the stack pointer:

```
push    bp          Save the old dynamic pointer
mov     bp,sp       Set the new dynamic pointer
sub     sp,#4       Allocate the local variables
```

The stack now appears as shown in Figure 7.12.

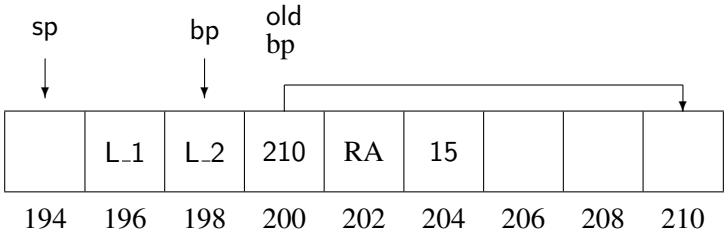

Figure 7.12: Stack after procedure entry

Now the body of the procedure can be executed:

[11] The extra mov instruction is there because stack operations work only to and from registers, in this case ax.

```
mov     ax,ds:[38]      Load the variable Global
add     ax,[bp+06]      Add the parameter Parm
add     ax,[bp-02]      Add the variable Local1
mov     ax,[bp]         Store in the variable Local2
```

Global variables are accessed as offsets from a special area of memory pointed to by the ds (data segment) register. The parameter Parm which is "lower" down in the stack than the start of the activation record is accessed at a *positive* offset from bp. The local variables which are "higher" up in the stack are accessed at a *negative* offset from bp. What is important to note is that since the 8086 processor has registers and addressing modes designed for common stack-oriented computations, all these variables can be accessed in a single instruction.

Procedure exit must reverse the effects of the procedure call:

```
mov     sp,bp           Release all local variables
pop     bp              Restore the old dynamic pointer
ret     2               Return and release parameters
```

The stack pointer is reset to the value of the bottom pointer thus effectively releasing memory allocated for the local variables. Then the old dynamic pointer is popped from the stack so that bp now points at the previous activation record. The only remaining tasks are to return from the procedure using the return address, and to release the memory allocated for the parameters. The ret instruction performs both of these tasks; the operand of the instruction indicates how many bytes of parameter memory must be popped from the stack. To summarize: procedure exit and entry require just three short instructions each, and access to local and global variables and to parameters is efficient.

7.9 Exercises

1. Does your Ada compiler use value or reference semantics to pass arrays and records?
2. Show how last-call optimization is implemented. Can last-call optimization be done on the factorial function?
3. McCarthy's function is defined by the following recursive function:

   Ada

   ```
   function M(I: Integer) return Integer is
   begin
      if I > 100 then return I-10;
      else return M(M(I+11));
   end M;
   ```

   (a) Write a program for McCarthy's function and compute M(I) for $80 \leq I \leq 110$.

   (b) Simulate by hand the computation for M(91) showing the growth of the stack.

   (c) Write an iterative program for McCarthy's function.

4. Ackermann's function is defined by the following recursive function:

Ada

```
function A(M, N: Natural) return Natural is
begin
   if M = 0 then return N + 1;
   elsif N = 0 then return A(M – 1, 1);
   else return A(M – 1, A(M, N – 1));
end A;
```

(a) Write a program for Ackermann's function and check that A(0,0)=1, A(1,1)=3, A(2,2)=7, A(3,3)=61.

(b) Simulate by hand the computation for A(2,2)=7 showing the growth of the stack.

(c) Try to compute A(4,4) and describe what happens. Try the computation using several compilers. Make sure you save your files before doing this!

(d) Write a non-recursive program for Ackermann's function.[12]

5. How are variables of intermediate scope accessed on an 8086?

6. There is a parameter passing mechanism called *call-by-name* in which each access of a formal parameter causes the actual parameter to be re-evaluated. This mechanism was first used in Algol but does not exist in most ordinary programming languages. What was the motivation for call-by-name in Algol and how was it implemented?

[12] Solutions can be found in: Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974, p. 201 and p. 235.