Compiler Construction

Rationale

Language translation techniques play a vital role in many fields of Computer Technology. The need for language translators arises because the language in which one wants to express ones computational needs is significantly different from the language recognised by a real machine. Development of translators, therefore, have resulted in two important fallouts. On one hand it has a strong influence on the design of new programming languages, on the other the issue of efficient translation of programming languages is inherently linked with the design of new architectures for higher throughputs. Language translation also plays an important role in the design of command languages and user interfaces of software systems, which require knowledge of scanning and parsing techniques.

In the current context, it is essential for computer science undergraduate students to develop a thorough knowledge of the theory and techniques of language translation. This course is aimed at imparting this knowledge through a detailed treatment of the theory of language translation, and a familiarity with the use of currently available software tools for automatic generation of various modules of a language translator.

Objectives

At the end of this course on Compiler Construction, the student should be able to

1. Understand the two models of language processing, viz. compilers and interpreters, and realize their strengths and limitations.
2. Understand the influence of various features of programming languages and machine architectures on the complexity and efficiency of language translation.
3. Appreciate the role of lexical analysis in compilation, understand the theory behind design of lexical analysers and lexical analyser generators, and be able to use Lex to generate lexical analysers.
4. Be proficient in writing grammars to specify syntax, understand the theories behind different parsing strategies – their strengths and limitations, understand how generation of parsers can be automated, and be able to use Yacc to generate parsers.

5. Understand the error detection capabilities of a compiler and how to report them in a user friendly manner.
6. Understand the need for various static semantic analyses such as declaration processing, type checking, and intermediate code generation, and be able to perform such analysis in a syntax directed fashion through the use of attributed definitions.
7. Understand the issues involved in allocation of memory to data objects, and the relation of such issues with higher level language features such as data types, scope and nesting rules, and recursion.
8. Understand the key issues in the generation of efficient code for a given architecture.
9. Understand the role played by code optimisation.

Compiler Construction

Course Modules

Module 1 : Introduction to Language Processing

Module 2 : Lexical Analysis

Module 3 : Syntax Analysis

Module 4 : Error Analysis

Module 5 : Static Semantics and Intermediate Code Generation

Module 6 : Run-time Environments

Module 7 : Code Generation

Module 8 : Code Optimisation

Preface

In this module, we shall first study the need for runtime storage management and see the issues involved in terms of source language features. To start with, we shall understand the concept of an environment and how it is captured by an activation record. We shall see what goes inside an activation record and how it is set during a procedure call. Then we shall look at ways of handling different parameter passing mechanisms. Finally, we shall study memory allocation techniques for dynamic data structures.

Contents

1 Introduction

The data objects (represented by variables) come into existence during the execution of the program. However, the code generated by a compiler must refer to the data objects using their addresses, which must be resolved during compilation. The addresses of data objects depend on their organization in memory, which, to a great extent, is decided by source language features. In this chapter we shall look at how source language features affect organization of data objects, so that their addresses can be resolved at compile time.

We shall restrict our study to the following two types of scenarios:

1. *Fortran like environments:* By this we mean programming languages, in which (i) the sizes of objects and their positions in memory are known at compile time, (ii) recursion is not permitted, and (ii) dynamic creation of data structures is not allowed. We call the memory allocation strategy for such languages, *static allocation.*

2. *Pascal like environments:* In which (i) the sizes of objects and their *relative* positions in memory are known at compile time, and (ii) recursion is permitted. We call this *stack allocation.* As in Pascal, we shall also allow (iii) Data structures to be created (under program control) at run time.

2 Basic Issues

To illustrate the influence of source language features on allocation strategy, we shall consider the example Fortran and Pascal programs shown in the following pages. The organization of data is largely determined by answers to questions like:

1. *Does the language permit recursion?* There may be many incarnations of `factorial` active at the same time, each with its copy of local variables and parameters which must be allocated storage. Moreover, the number of such incarnations will be decided at run-time.

2. *What are the parameter passing mechanisms?* Inside `factorial`, the access mechanisms for the formal parameters `r` and `x` have to be different. This is because `r` is *call-by-reference*, whereas `x` is *call-by-value.*

EXAMPLE 1: Fortran program

```
      PROGRAM MAIN
      DIMENSION A(100)
      READ *, (A(I), I = 1,100)
      AVGVAL = AVG(A,100)
      STDDEV = STD(A,100)
      PRINT *, AVGVAL, STDDEV
      END
      FUNCTION AVG(B,N)
      DIMENSION B(N)
      SUM = 0
      DO 20 I = 1,N
   20 SUM = SUM + B(I)
      AVG = SUM/N
      END
      FUNCTION STD(C,M)
      DIMENSION C(M)
      SUM = 0
      AV = (*) AVG(C,M)
      DO 30 J = 1,M
   30 SUM = SUM + (C(J)-AV)**2
      STD = SQRT(SUM)/M
      END
```

3. ***How does a procedure refer to the non-local names?*** **The rules of static scoping decide that the** `z` **being referenced in** `factorial` **refers to the** `z` **in the main program. Therefore, apart from the local variables and formal parameters,** `factorial` **should be able to access this variable at run-time.**

4. ***Does the language permit creation of dynamic data structures?*** **Space must be created at run-time, each time** `new(y)` **is executed.**

2.1 Initial Division of Memory

We assume that the underlying hardware and operating system provides a division of memory into the following.

EXAMPLE 2: Pascal program

```
type t = ^integer;
var z: integer;
procedure  factorial (var r: t; x: integer);
   var y: t;
   begin if x = 0 then r^ := 1
        else begin
                new(y);
                factorial(y,x-1);
           (#)  r^ := y^ * x; writeln(z)
             end
   end;
procedure start;
var p:t, z:integer;
    begin
      z := 4; new(p);
      factorial(p, z)
(*) end;
begin (*main*)
z:=2; start;
end.
```

(*) and (#) have been used to mark program points for future reference.

1. ***Code area:*** **This would contain the generated target code.**

2. ***Static area:*** **This would contain data whose absolute addresses can be determined at compile time. As examples, in languages like Fortran, the addresses of all variables can be determined statically, and therefore can be kept in the static area. Whereas, in languages like Pascal, the local variables of the outermost procedure (for example `z` in `main`) can be kept in the static area since these are the only variables whose addresses can be determined at compile time.**

3. ***Stack area:*** **This area is for data objects of procedures which can have more than one incarnations active at the same time. The local variables and parameters of all Pascal procedure (except for the main program) would reside in this area.**

4. ***Heap area:*** This is data created at run time, such as objects pointed to by pointer types.

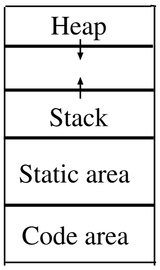

Figure 1: Initial division of memory

Since the sizes of the stack and the heap cannot be estimated at run time, they are arranged to grow in opposite directions as shown in figure ??.

3 Activation Records

An ***activation record*** is a conceptual aggregate of data which contains all the information required for a single activation of a procedure. Activation records are held in the static area for languages like Fortran and in the stack area for languages like Pascal. Typically space must be provided in the activation record for:

1. ***Local variables:*** Part of the local environment of the currently activated procedure.

2. ***Parameters:*** Also part of the local environment of the current procedure.

3. ***Return address:*** To return to the appropriate control point of the calling procedure.

4. ***Saved registers:*** If the called procedure wants to use the registers used by the calling procedure, these have to be saved before and restored after the execution of the called procedure.

5. ***Static link:*** This is required in languages like Pascal, which have a non-trivial nesting structure. A procedure in such a language may access variables not locally declared, i.e. variables which are global or declared in surrounding procedures. The static link of a procedure points to the

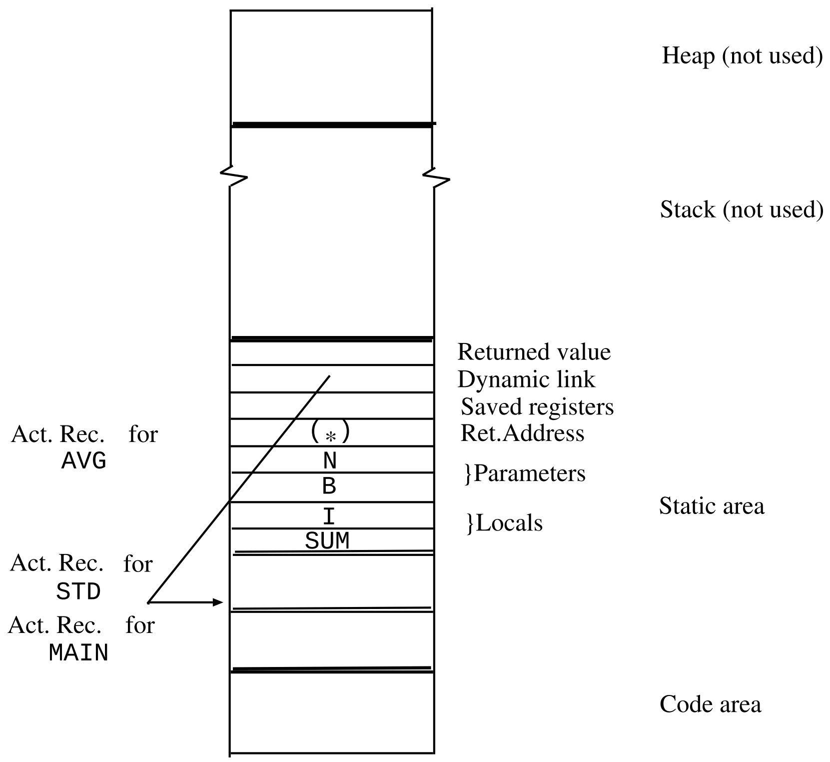

Figure 2: Activation record for the Fortran example

latest activation record of the immediately enclosing procedure. Static links may be avoided by using a display mechanism. Obviously static links are not required for Fortran like languages, which do not allow nest of procedures.

6. *Dynamic link:* Pointer to activation record of calling procedure.

7. *Returned value:* Used to store the result of a function call.

We first look at examples of activation records of the two examples shown earlier at certain points of their execution. For the Fortran program shown earlier, the activation record of `AVG` during its invocation from `STD` is shown in figure ??.

Similarly, for the Pascal program, the activation record of `factorial` during its first invocation is shown in figure ??. This is one of many possible organizations of an activation record. We shall see a different organization later.

Having seen what activation records look like, we shall see how the information inside activation records are accessed, and then we shall see how they are set up when a procedure is invoked.

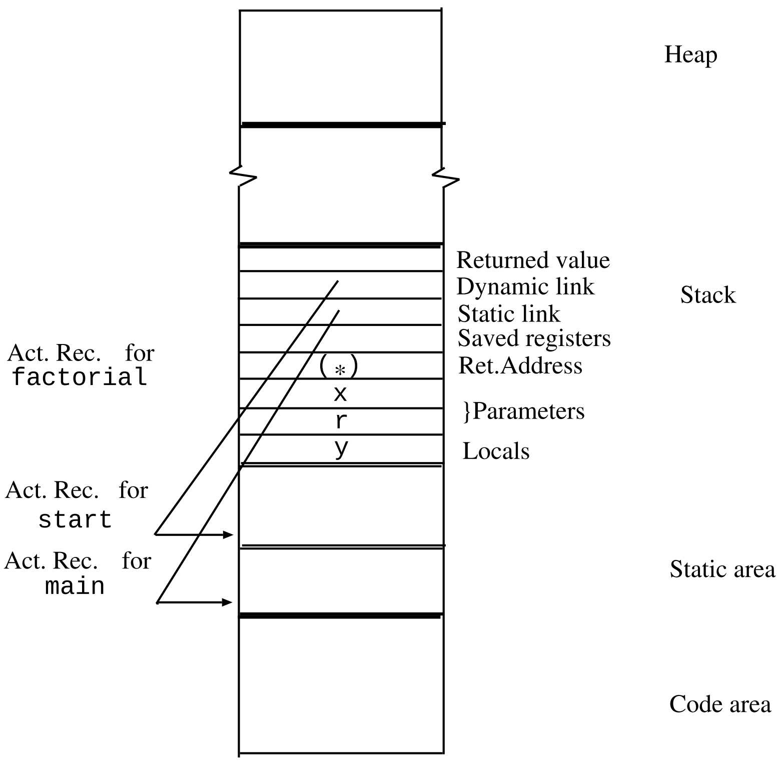

Figure 3: Activation record for the Pascal example

3.1 Accessing Information Using Activation Records

Within an activation record, any information is referred to using an offset relative to a pointer (BP) pointing to the base of the activation record. Thus, in figure ??, the address of the local variable `y` is BP (contents of register BP) and the parameter `x` has the address BP+2. For simplicity we shall assume that every information in the activation record takes unit space. Some machines use a dedicated register for holding the base pointer.

However, as stated earlier, a procedure may sometimes refer to variables which are not local to it, and therefore will not be available in its own activation record. To see how they are accessed, consider an outline of a program fragment with a non-trivial nesting structure:

```
procedure S;
   var a,x: _
   procedure R;
      var i: _
      procedure T;
         var m,n: _
```

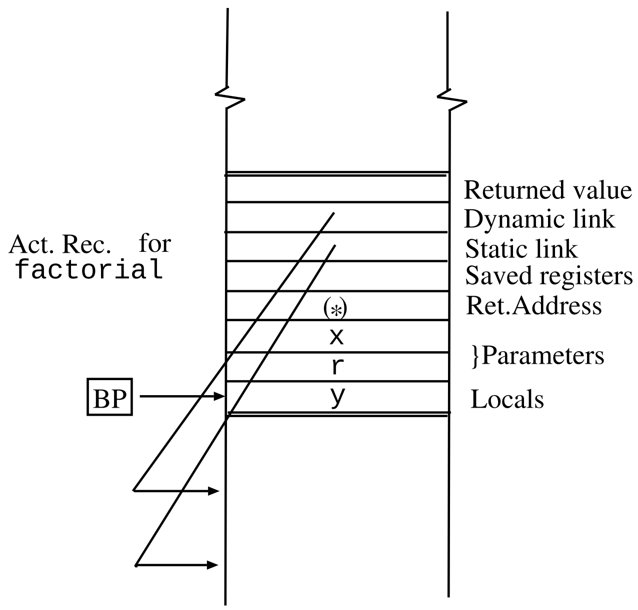

Figure 4: Accessing information in an activation record

```
      body of T;
   body of R;
procedure E;
   body of E;
procedure Q;
   var k,v: _
   function P(y,z:_): _
      var i,j: _
      body of P;
   body of Q;
body of S;
```

Assuming that the language uses *static scoping* rules, the names which can be referred to (visible) at a program point are (i) The names declared locally in the procedure, of which this program point is a part, (ii) the names of all enclosing procedures, and (iii) the names declared immediately within such procedures. An additional rule is that a name declared in more closely nested procedure overrides the same name declared in an outer procedure.

For example, the names visible in the body of `T` are:

1. **`T, m, n` – names declared in `T`.**

2. **`R` and `S` – enclosing procedure names.**

3. `i`, `a`, `x`, `E` and `Q` – names declared immediately within `R` and `S`.

The non-local variables of a procedure are accessed by chasing a chain of pointers starting from the base of the activation record of the procedure. As an example, consider a call `S` → `R` → `T` → `Q` → `Q` → `P`. The stack contents and the static link chains at this point of execution are shown in figure ??. At this point of execution, the procedure `P` finds its local variables in its own activation record. However, to access the variables of `Q`, it goes down the static link and reaches the base of the activation record of `Q`. It can then access the local variables of `Q`. Similarly, to access the variables of `S`, it goes down the static link twice.

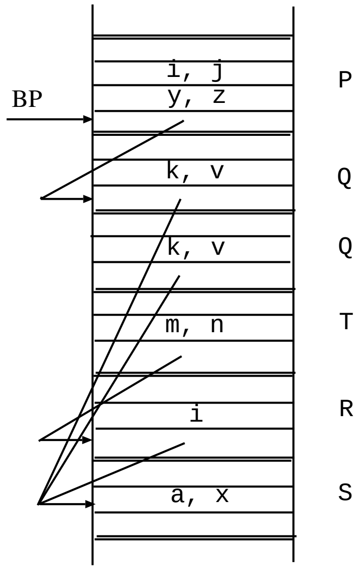

Figure 5: Accessing non-local variables through static link

Also notice the following important fact. Though there may be more than one activations of an enclosing procedure, the variables in the *latest activation* are accessed. In the example being discussed, if we want to access a variable of `Q`, during execution of `P`, the variables in the activation record of the second activation of `Q`, and not the first, will be accessed.

Assuming that the static link has been set up, we have seen how non-local variables are accessed. But how does one set up the static link during a procedure call? It depends on the relative *levels* of the calling and called procedure. The level of a procedure is calculated using the following rules:

1. The level of the main program is 1.

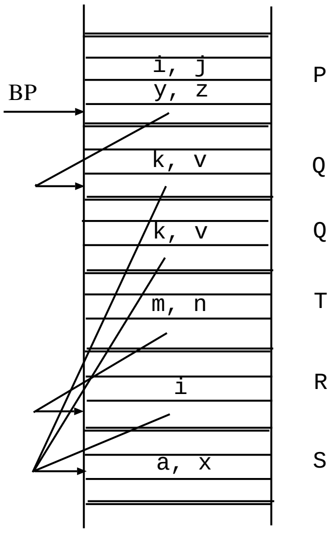

Figure 6: Setting up the static link

2. If the level of an enclosing procedure is l, then the level of the enclosed procedure is $l + 1$.

The level of procedures in the previous example (ignoring the procedures outside S) is:

S — 1
R — 2
T — 3
E — 2
Q — 2
P — 3

In terms of levels, the rules for setting the static link are as follows. If the calling procedure is at level l_1 and the called procedure is at level l_2, then starting from the activation record of the calling procedure the static link is traversed $l_1 - l_2 + 1$ times. The static link of the called procedure is made to point to the base of the activation record thus reached.

In the example shown in figure ??, assume that at this point of execution procedure T (level 3) calls procedure Q (level 2). Then following the rules stated earlier, we should start from the activation record of T and travel down the static link twice (3-2+1) to reach the base of the activation record of S. This is where the static link of Q should point to. Notice that S is also the immediately enclosing procedure of Q. The reader is urged to verify that the rule stated above is correct in the sense that it always sets the static link of the called procedure to the most recent activation record of the immediately

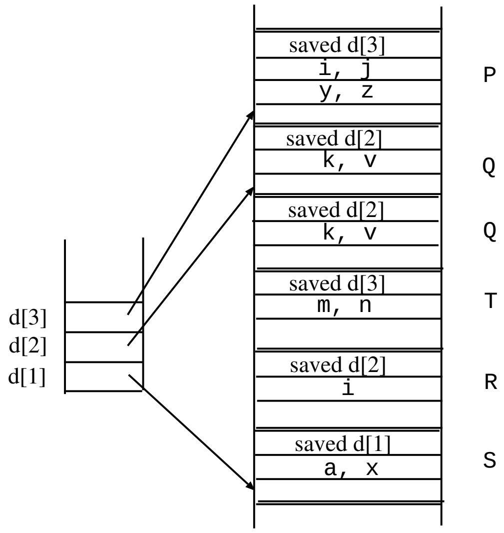

Figure 7: Accessing non-locals through display

enclosing procedure.

As we have seen before, accessing a non-local variable involves going down the static link, possibly more than once. This process can be made more efficient if, instead of chasing pointers, one uses an *array* of pointers to activation records. This array is called a *display.*

In the previous example, for the calling sequence `S` $\rightarrow$ `R` $\rightarrow$ `T` $\rightarrow$ `Q` $\rightarrow$ `Q` $\rightarrow$ `P`**, the display would look as shown in figure ??. From the body of** `P`**, the base of the activation record of the (unique) procedure at level 2 (in this case** `Q`**) would be given by d[2].**

Once again, how does one set up a display during a procedure call? When setting up d[i] to point to the base of the current procedure at level i, the old d[i] is saved within the activation record of the procedure itself. While a new activation record of a procedure (at nesting depth i) is being set up, the current value of d[i] is saved in the new activation record and d[i] is set to point to the new activation record. And at the end of execution of the procedure, d[i] is reset to its old value.

4 Parameter Passing

The common methods to associate an actual parameter with a formal parameter are:

1. ***Call by value:*** **The calling procedure evaluates the actual parameter and places the resulting value in the activation record of the called procedure.**

2. ***Call by reference:*** **The actual parameter is a variable. The calling procedure places the address of this variable in the activation record of the called procedure. A reference to the formal parameter, then, becomes an indirect reference through the address passed to the called procedure.**

If a procedure `P` **declared as**

`P(x:_, var y:_)`

is called as

`P(e,a)`

where `e` **is an expression and** `a` **a variable, then** `e` **is evaluated and the *value* of** `e` **is placed in the location for** `x` **in the procedure** `P`**. Whereas the *address* of** `a` **is placed in the location for** `y`**. Inside** `P`**, any reference to** `x` **translates to a reference to the location for** `x`**, while a reference to** `y` **translates to an indirect reference to** `a` **through the address in the location for** `y`**.**

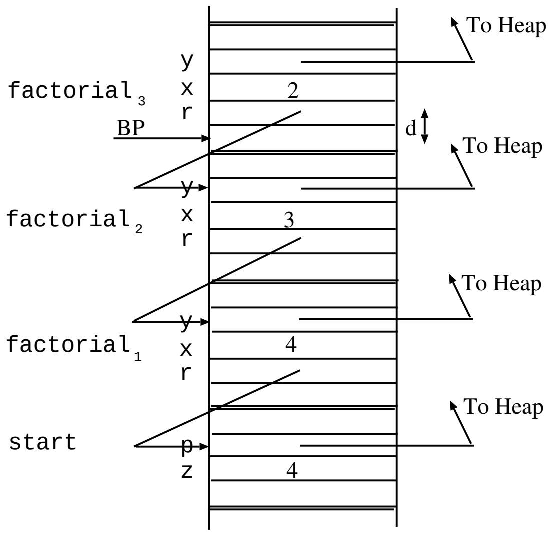

Figure 8: Parameter passing

In the `factorial` **example, the parameters are passed in the manner shown in figure ??. When** $\texttt{factorial}_2$ **calls** $\texttt{factorial}_3$**, the value of** `x-1` **(2 in this case) is evaluated and stored in the location for** `x` **in** $\texttt{factorial}_3$**. However, the address of** `y` **in** $\texttt{factorial}_2$ **is stored in the location for** `r` **in** $\texttt{factorial}_3$**.**

A reference to x now translates to *(BP+d_x) while a reference to r translates to *(*(BP+d_r)), where d_x and d_r are the offsets of r and x in the activation record.

Some other methods of parameter passing are:

3. ***Copy restore:*** The actual parameters are evaluated, and the values passed to the called procedure. After execution, the current values of the formal parameters are copied back into the addresses of the actual parameters, if it is meaningful to do so. ***Note that both these activities are done by the caller.***

4. ***Call by name:*** The effect of the call is that of substituting procedure body for the procedure call, with the actuals literally substituted for the formals. Name clashes, if any, are resolved by renaming.

   This strategy is actually implemented by passing a pair consisting of a pointer to the code to evaluate the actual parameter and the environment to resolve accesses to the variables in the actual parameter. Such a pair is called a ***closure***.

It would be appropriate to compare call by reference with the copy restore method. The second method seems to be more secure than the first, particularly in Fortran like contexts. Consider the Fortran subroutine shown below:

```
SUBROUTINE SWITCH (N)
N = 3
RETURN
END
```

Assuming that the parameter passing mechanism is call by reference, if this subroutine is called as `CALL SWITCH(2)`, then (depending on the implementation) it may have the unexpected side effect of modifying the data area for literals. However, if the parameter passing mechanism is copy restore, then the caller would know that the subroutine has been called with a constant. Therefore it would not copy back the modified actual parameter into the formal parameter.

5 Compiling Procedure Calls

During a procedure call, there are many possible ways of dividing the book-keeping work between the caller and the callee. One possible way illustrated in figure ??.

During a procedure call

Caller's activities

1. Makes space on the stack for a return value.
2. Puts the actual parameters on the stack.
3. Sets the static link.
4. Jumps to the called procedure. Return address is saved on the stack.

Callee's activities

1. Sets the dynamic link.
2. Sets the base of the new activation record.
3. Saves registers on the stack.
4. Makes space for local variables on the stack.

Note that if we follow this sequence, the base pointer does not actually point to the base of the activation record, but somewhere in the middle. The actual parameters are now referenced using a negative offset, and the local variables with a positive offset.

During a return from procedure

Callee's activities

1. Restores registers.
2. Sets the base pointer to the activation record of the calling procedure.
3. Returns to the caller.

Caller's Activities

1. Restores stackpointer to the location containing the returned value.

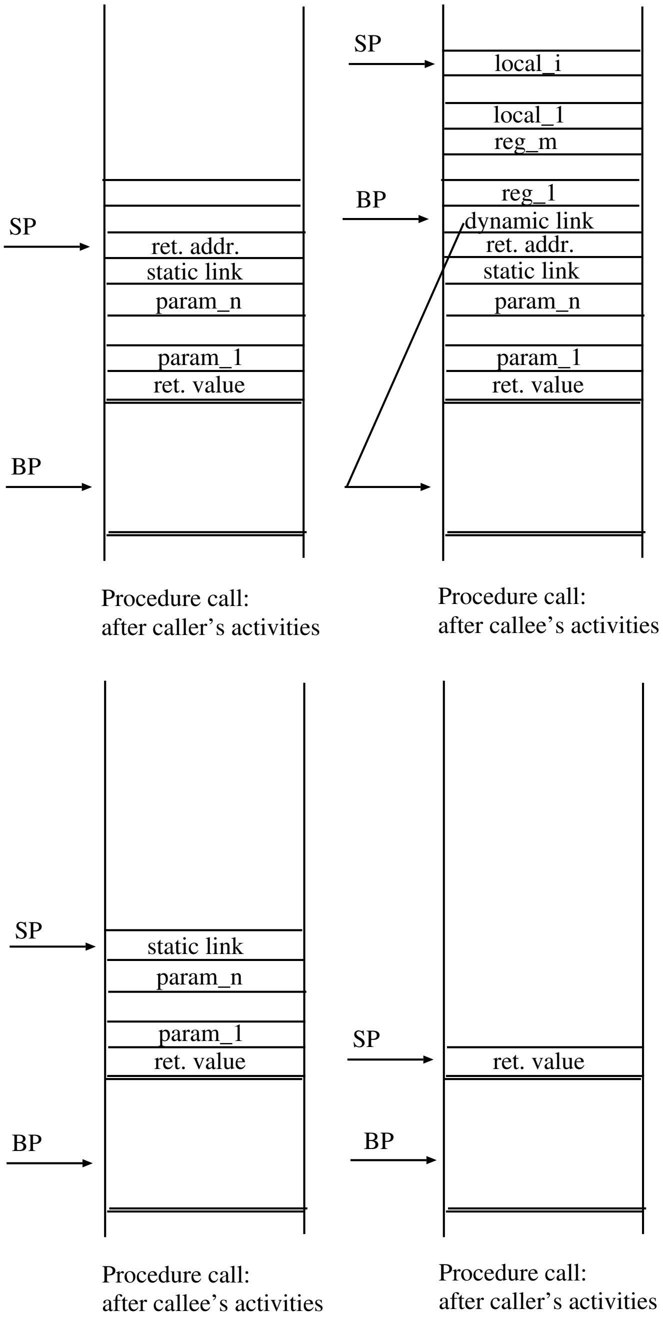

Figure 9: Division of activities during a call

5.1 Machine Support for Compilation

Modern processors have instructions which facilitate compilation. For example the Motorola 68000 processor has:

1. **LINK BP, #disp:**
   - **Pushes the base pointer on the stack.**

- Sets the base pointer to the top of the stack.
- Makes disp amount of space on the top of the stack.

disp is usually negative because, in 68000, the stack grows towards decreasing memory locations.

2. UNLK BP:
   - Resets the stack pointer to the base pointer.
   - Restores the base pointer of the caller.

3. JSR P: Jumps to the procedure P, after having pushed the return address on the stack.

4. MOVEM D0–D7, (SP)/ MOVEM (SP), D0–D7: Saves/restores registers D0–D7 on/from the stack.

5. PEA BP + x: Pushes the effective address of the location BP + x. Useful for passing parameters by call-by-reference.

Shown below are the relevant parts of the M68000 code generated for the recursive call to `factorial`. For this example, it is assumed that (i) the current values of the variables reside in their home locations, (ii) no registers need to be saved, (iii) each data item requires one location, and (iv) no space needs to be allocated for a return value.

```
factorial: LINK BP, #-1    /* for y */
           ...
           ...             /* code for call to factorial */
           PEA BP+1        /* push address of y */
           MOVE 3(BP), R1  /* push value of x-1 */
           SUB #1, R1
           MOVE R1, -(SP)
           MOVE BP, R1   /* set up static link */
           MOVE #1, R2
     LOOP: CMP R2, 0
           BEQ END
           MOVE 2(R1), R1
           SUB #1, R2
           JMP LOOP
```

```
END: MOV R1, -(SP)
     JSR FACTORIAL
     ADD #3 SP        /* clean up act. rec. of callee */
     ...
     ...
     UNLK BP
     RSR
```

6 Handling Dynamic Data Structures

A pointer variable `p` is allocated memory on the stack. However, On execution of a statement such as `new(p)` (in Pascal), memory is allocated from the heap and the home location of the pointer variable (on the stack) is made to point to the allocated location in the heap. Any reference to `p^` is an indirect reference to the heap location through the address on the stack.

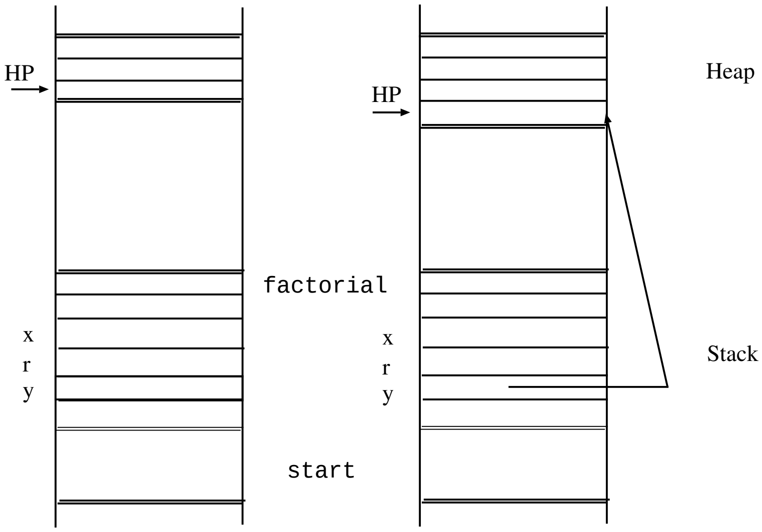

Figure 10: Allocation of memory for dynamic data structures

Figure ?? shows the memory configuration before and after the execution of `new(y)` in `factorial`. A new location has been released from the heap and the contents of `y` has been made to point to it. On running out of heap space, the run-time environment might employ the services of a *garbage collector*.

We summarize by showing the entire memory after completion of the third call to `factorial`.

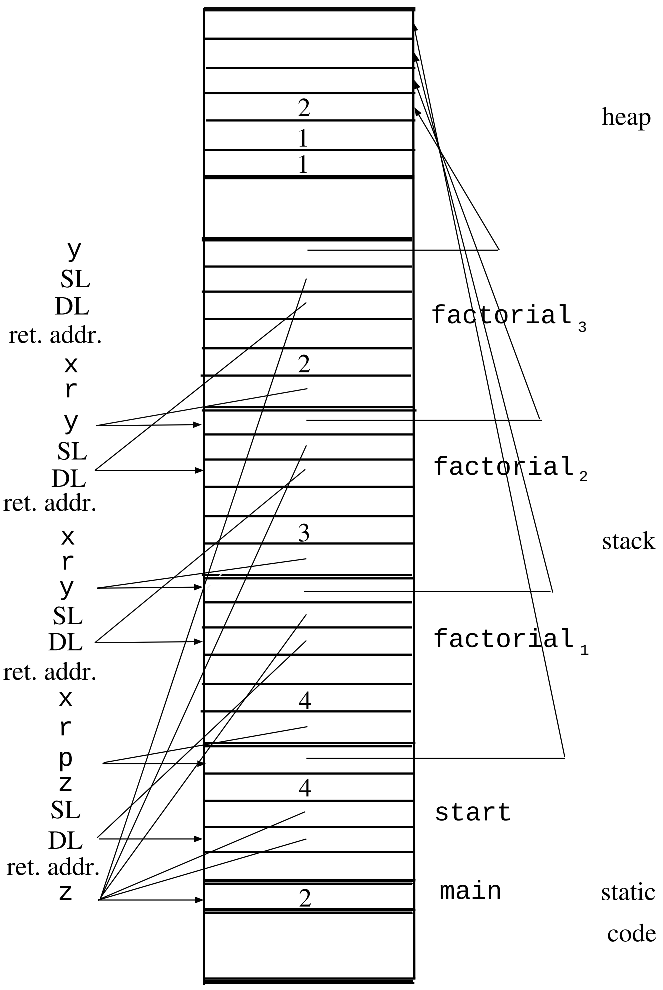

Figure 11: Execution trace of `factorial`

7 Graded Exercises

1. **What is printed by the the program shown below, assuming (a) call-by-value, (b) call-by-reference, (c) copy-restore linkage (d) call-by-name?**

```
program main(input,output);
   procedure p(x, y, z);
      begin
         y := y + 1;
         z := z + x
      end;
   begin
      a := 2;
      b := 3;
```

```
    p(a+b, a, a);
    print a
  end.
```

2 Some languages like C and Algol allow a programmer to define blocks as shown below.

a. How would you implement blocks? In particular, for this program, how would you access the variables at (*) at runtime?

b. Can you allocate storage for all the blocks in a procedure at the point of entering the procedure body? If so, describe how?

```
main()
  {int a = 0;
   int b = 0;
     {int b = 1;
        {int a = 2;
         printf(``%d\n'', a, b);}
        {int b = 3;
   (*)   printf(``%d\n'', a, b);}
      printf(``%d\n'', a, b);}
   printf(``%d\n'', a, b);}
```

3. Consider the Pascal program shown below:

```
var x : integer;
   procedure p(x: integer; var y:integer);
      procedure q(x: integer; var y : integer);
         procedure t(x: integer; var y : integer);
            var r : integer;
            begin (*t*)
               r := y;
               p(x-1,r)
            end;   (*t*)
         begin (*q*)
            t(x,y)
         end;  (*q*)
      begin (*p*)
```

```
          if x = 0 then x := 1 else q(x,y)
       end   (*p*)
begin (*main*)
   x := 2; p(2,x)
end.  (*main*)
```

a. **Assuming an implementation using static links, show the runtime stack when it has grown to its maximum height.**

b. **Generate M68000 code for the program.**

c. **Repeat a. using display instead of static links.**

8 References

1. Aho, A.V., R. Sethi and J.D. Ullman (1986) : *Compilers – Principles, Techniques and Tools*, Addison-Wesley, Reading.

2. Barrett, W.A. and J.D. Couch (1977): *Compiler Construction*, Science Research Associates, Pennsylvania.

3. Bauer, F.L. and J. Eickel (1974): *Compiler Construction : An Advanced Course*, Springer-Verlag, Berlin.

4. Fischer, C.N. and R.J. LeBlanc (1988) : *Crafting a compiler*, Benjamin/Cummings, Menlo Park, California.

5. Gries, D. (1971): *Compiler Construction for Digital Computers*, Wiley, New York.

6. Holub, A.I. (1993): *Compiler Design in C* Prentice-Hall, Englewood-Cliffs.

7. Tremblay, J.P. and P.G. Sorenson (1985): *The Theory and Practice of Compiler Writing*, McGraw-Hill, New York.

8. Watson, D. (1989): *High Level Languages and their Compilers*, Addison-Wesley, Reading.

9. Bruce J. MacLennan (1983): *Principles of Programming Languages: Design, Evaluation, and Implementation*, Holt-Saunders, New York.