Chapter 12

Some PL concepts

Let us now step back and take a fresh view of programming languages. A PL provides a notation for problem solving. Problem solving consists of transforming input values to output values.
We have discussed different types of values allowed by programming languages: integers, reals, arrays, etc. How are these values specified and manipulated? A language provides different ways for value specification: literals, variables, expressions, and functions. A PL also provides ways of achieving side-effects: statements, blocks, and procedures. Here some recursion kicks in: blocks can contain variable declarations and other blocks, functions can contain function definitions, etc. Because of this complexity, additional concepts are needed to discuss programing languages.

12.1 Literals

Literals are values which stand for themselves. For example, `3.14159` is a literal value. How about the string `"Hello\n"`? Is it a literal? In Perl, this string is considered to be an interpolated string since the character `\n` is interpreted. A literal string literally has a new line! In other languages, this would be considered a literal string.

12.2 Names and binding

Names have a great mnemonic value. Names in programming languages usually consist of a sequence of alphanumeric characters. The meaning of name has to be explicitly declared by the programmer in most languages. In some languages, names have an implicit meaning. In Fortran, names starting with characters `I` to `N` are implicitly treated as integers. It is possible to override this convention in Fortran. In Perl, names staring with `@` are considered arrays. It is not possible to override this convention.
In compiled languages, names exist only at compile time. At runtime, only values and their addresses exist. In interpreted languages, names and values coexist. What is the meaning of

a name? The meaning of a name is its binding. Binding is an association between a name (in the symbol table) and a storage object (an area of memory). Names can stand for either constants or variables.
In PLs variables have multiple purposes: as a shorthand for some value (functional languages), the main vehicle for storing the state of computation (imperative languages), and as a way of expressing constraints (logic languages).
There are many notions associated with names: introduction of a name (declaration), allocation of memory (definition), and use. The definition is also known as a binding occurrence, the use as bound occurrence. The region of the program where a name is bound is called its scope. The following C declaration (actually a definition)

```
int i;
```

allocates storage for the name `i`. Note that a declaration need not always result in storage allocation as in `extern` declarations of C.
Where can variables be declared? In most modern languages, variables can be declared anywhere in the program. The following code is valid in C++, but not in C.

```
int main(void) {
   int a;

   a = 1;

   int b=a; // cannot declare a variable in the middle of a block in C
}
```

Note that a name has no meaning before its declaration. In general, the *scope* of a name is the region of the program where it is meaningful. Can we say that the scope is the complete portion of the program after its declaration? No. While none of the languages allow redeclaration of a name inside its scope, all languages allow nesting of scopes or blocks. While the following code is invalid

```
   int b=a;

   float b; // error. b cannot be redeclared
```

the following code is valid.

```
   int b=a;

   {  // start a new scope or block
     float b=a; // redeclare b inside the new block, old 'a' is used
   }
```

We say that the new declaration causes a hole in the scope of the original name `b`. Note that the occurrence of `a` refers to the declaration outside the new scope. We say that the occurrence of `a` is *free* inside the new block.
What happens when functions have free variables? (We can think of functions as blocks with names and which can be called.) Consider the following Pascal code.

```
var n: char;  // binding occurrence of identifier n

procedure W;
begin
  writeln(n) // free occurrence of identifier n
end;

procedure D;
  var n: char; // another binding occurrence of n
begin
   n := 'D'; // bound occurrence
   W
end;

begin
  n := 'L'; // bound occurrence
  W;
  D
end.
```

The identifer `n` is declared in the first line. Any occurrence of `n` after that is a bound occurrence (unless the occurrence is itself another binding occurrence.) The scope of an identifier is the region where it is bound. In the above example, the scope of `n` is after the first line. In particular, in the statement `n := 'L'`, occurrence within this scope. What about the identifer `n` in procedure `D`? In this case, `n` has another binding occurrence. So procedure `D` is a hole in the scope of the original scope of `n`.
What about the occurrence of `n` in procedure `W`? In the context of the procedure definition, the occurrence is said to be *free*. There are two ways to get binding occurrence of `n` in `W`.

static scoping: In this the binding occurrence is resolved at compile time. The binding occurrence is obtained form the closest lexically enclosing scope. If we run the above Pascal program, the output is `L, L`.

dynamic scoping: Here the binding is resolved at runtime. The binding occurrence is the closest binding occurrence at runtime. If we run the above program with dynamic binding, the output is `L, D`.

Dynamic scoping involves runtime overhead (searching for the most recent scope). In addition to runtime overhead, the programs can be difficult to understand since one has to trace the execution of the program to understand its behavior – mere reading is not sufficient. Some people consider dynamic binding to be convenient. Consider a function which calculates the sum of elements of an array.

```
int sumArray(int* a) {
   int sum=0;
   for (i=0; i < n; i++) sum += a[i]; // n is free, works for dynamic binding
   return sum;
}

int xxx() {
   int* a = new int[n];
   s = sumArray(a); // under dynamic scoping, the current value of n is used
}
```

The same effect can be achieved by using a global variable. (How?)
We can characterize static scoping by *renaming invariance*. If we rename all the local variables consistently, the result of the program does not change. For example, in the original Pascal program, suppose we rename the variables of `procedure D`.

```
procedure D;
  var m: char;
begin
   m := 'D';
   W
end;
```

The output of the program under static scope is the same. But the output under dynamic scoping is different. Variable renaming is also called *alpha conversion*.

12.3 Lifetimes

Where do variables live? Can we allocate a block of memory for each function to keep its variables? This results in waste of memory if some functions are not called. Also allocating a *single* blocks of memory to each function is not sufficient if the function is recursive. All the modern languages have a stack based memory allocation for local variables of functions. The local variables become alive only during function activations.
Consider the execution of the recursive function

```
fun length([]) = 0
 | length(_::rest) = 1+length(rest);
```

for the input `[1, 2, 3]`. The stack looks as below

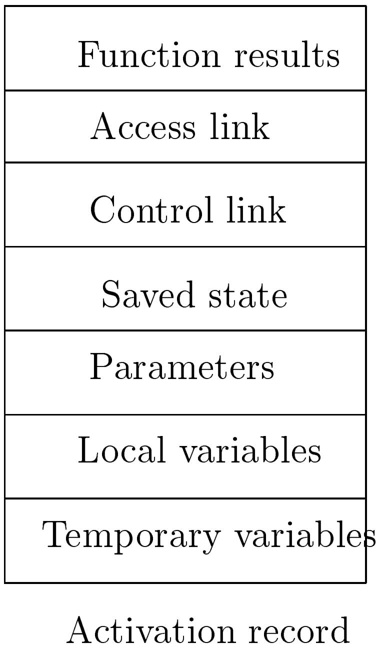
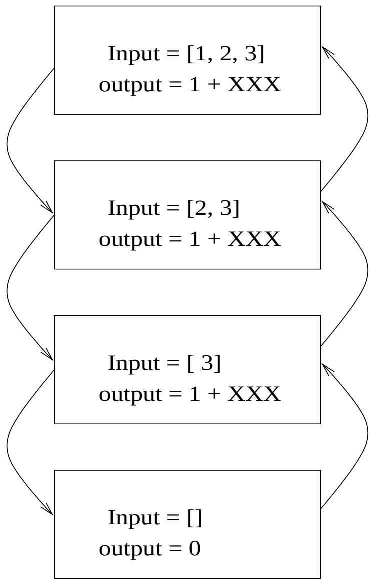


Figure 12.1: Activation record for Pascal



Note the multiple instances of local variables in the stack. The activation frame actually consists of the fields shown in figure 12.1.
Since local variables become alive only during function activations, the following code is meaningless.

```
int* copyPointer(int a) {
```

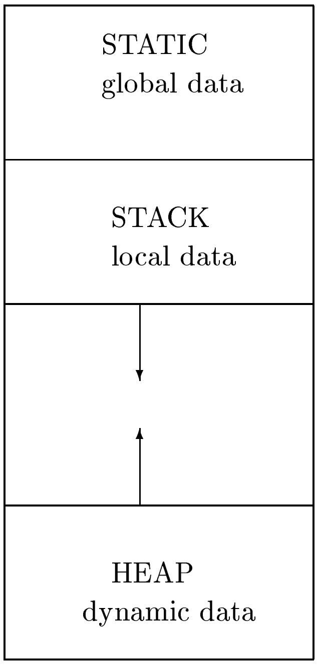


Figure 12.2: Memory layout for a program at runtime. The all dynamic data is allocated on the heap. The local data for function activations is done on the stack. The stack grows with function calls and shrinks with function returns. The heap grows with memory allocations. The size of the heap reduces with memory deallocation and compaction. In most languages, programmer is responsible for deallocation. In languages like Lisp, ML, and Java deallocation is done automatically by the system (*garbage collection*).

```
   int x; // allocated on stack, has no permanent life

   x=a;
   return &x; // cannot use address of "x"
}
```

What should a function do to return a more enduring value? It should allocate memory in the heap using dynamic memory allocation and return a pointer to the heap. (See figure 12.2.)

```
int* testFn(int a) {
   int *x; // x is allocated on stack, has no permanent life

   x = new int[1]; // x points to a permanent location in heap
   *x=a;
   return x;
}
```

The stack allocation scheme combined with nested procedure definitions introduces some complication in scope rule implementation. In static binding, an free reference refers to the closest lexically enclosing block. In nested procedure defintions, different variables can reside in different scopes. Consider the following example.

```
procedure U;
   var u:integer;
   procedure V;
      var v: real;
      procedure W;
      var w: real;
      begin
          w :=  u+v;
      end;
   begin
     W;
   end;
begin
  V;
end;
```

All the local variables are allocated on the stack. In the statement `w :=  u+v`, variables `u` and `v` live in different scopes. How are they accessed at runtime? Each activation record keeps a pointer to the activation record of the enclosing scope. This pointer is called a *access link*. See figure 12.1. Variable `v` is retrieved by following the access link. How's `u` retrieved? The static link is traversed twice: once from the activation record of `W` and once from that of `W` to reach the activation record of `U`. From the activation record of `U`, variable `u` is retrieved. The number of access links to be traversed is the same as the *nesting depth* of the procedure. Recursion adds to the complexity of access links. Suppose the closest enclosing block has multiple activations because of recursion. Which activation is used? We have to use the most recent activation of the closest lexically enclosing block. Consider the following Pascal program.

```
program L(input, output);
var n: integer;

procedure p1(i:integer);
  var n: integer;
  procedure p2;
  begin
    n := n+3; { which n? }
  end;

begin
  n := i;
```

```
  if (i<4) then p1(i+1) { recursive call }
  else
    begin
      p2;
      writeln(n);
    end
end;

begin
  n := 100;
  p1(1);
end.
```

A data structure called *displays* is used to speedup travesal of access links. It is an array of size nesting depth. The entry at index i holds the access link of the scope at distance i. So sequential traversals of access links can be replaced by an array lookup.
Note that finding the most recent activation of each procedure involves runtime overhead. To reduce this overhead, C does not permit nested function defintions. In C, all functions exist in the same global scope.
It is sometimes useful for procedures to have variables which retain their value across invocations. Such variables are called "own" (Ada) or "static" (C) variables.

```
void state (void) {
  static int n=0; // initialized once, value "remembered"
  int m=0; // initialized for each call
/* ... */
}
```

12.4 Constants

Most of the languages allow the notion of constants which cannot be changed. Sometimes the notion of constants is misleading. Consider Java. In Java, all variables hold references to objects. When we declare a name to be `final`, only the reference is held constant – not the object. Consider the following example.

```
class Test {
    int f = 0;

    Test(int a) {
       f = a;
    }

    public static void main(String[] args) {
```

```
        final Test t1 = new Test(10);
        Test t2 = new Test(20);

//      t1 = t2; // compile time error
        t1.f = 100; // OK
    }
}
```

In Java, there is no way to define constant objects. C++ allows constant references as well as constant objects. But the syntax can be confusing.

```
int main(void) {
    const char* pc = "hello";
//  pc[2] = 'a';  // error
    pc = "world";

    char *const pd = "hello";
    pd[2] = 'a';
//  pd = "world"; // error

    int a = 1;
    const int c = 2;
    const int* p1 = &c;
    const int* p2 = &a;
    int* p3 = &c; // compiler warning, *p3 is now 2
    *p3 = 10;
}
```

12.5 Equality

There are two types of equality: equality of objects and the equality of references. Consider the following example in Java.

```
    str1 = "Hello";
    str2 = "Hello";
    str3 = str1;

    str1.equals(str2) // true, equality of objects
    str1 == str3      // true, equality of reference
    str1 == str2      // false, inequality of reference
```

In case of RMI, there are some issues in comparing remote references for equality. One should use `equals()` for comparison.

12.6 Function calls

Function calls involve establishing a correspondence between actual parameters and formal parameters. There are many ways of establishing this correspondence.

12.6.1 Call by value

In this, the formal parameters are evaluated and copied to the stack. The function then uses the values in the stack for evaluation. In this manner, the calling function is insulated from the function evaluation. The function evaluation cannot have side effect. This type of calling sometimes prevents writing of useful procedures – like procedures for swapping two values of two variables.

```
void swap(int a, int b) { // does not work
   int tmp;
   tmp = a;
   a = b;
   b = tmp;
}
```

In the above example, since `swap` works with local copies, the function has no effect on the calling function.

12.6.2 Call by reference

To achieve the above functionality, the parameters should be passed by reference.

```
void swap(int& x, int& y) { // passed by reference
   int tmp;
   tmp = x;
   x = y;
   y = tmp;
}

int main(void) {
  int u=5, v = 10;

  swap(u,v);
}
```

Note that C supports only call by value. One should use pointers to write a swap function in C.
Call by reference is also useful when huge values (like arrays, records) have to be passed. Copying the objects to the stack may result in poor performance. So call by reference is used.

12.6.3 Call by name

This is sometimes called *call by macro*. Here the actual arguments are textually substituted in the procedure body. While this sounds like call by reference, it is not. Consider the example.

```
// assumption: parameters are passed by name
   int callByName(int a, int d) { // function definition
     a := a + 1;
     return d;
   }

   i = 1;
   c = callByName(i, b[i]);
```

If the parameters are passed by name, then `i` gets the value `2` and the value returned by the function is `b[2]` (not `b[1]`).
Call by name is not used in any of the new languages. It was available in Algol.

12.6.4 Call by value-result

This is also called *copy-in/copy-out*. In Ada, there are three kinds of parameters: **in**, **out**, and **in out**. At function invocation, the **in** and **in out** parameters are copied from the argument list to the stack. After function terminaion, the **in out** and **out** parameters are copied from the stack to the actual parameters. In most cases, the values are similar to those using call by reference. The following example illustrates the difference.

```
var i, j: int;
procedure foo(in out x, y: int); begin i:= y end;
begin
  i := 2; j := 3;
  foo(i, j);
end.
```

12.7 Exceptions

Exceptions in Java are called *checked* exceptions. Each method either handles all the exceptions in its body or declares the exceptions it throws as part of its declaration. This mechanism makes it easy for the compiler to enforce exception handling. C++ does not insist on declaring exceptions as part of the interface. So exception handling in C++ can be error prone.
Exception handling requires the following steps.

1. Examine the function in which the throw occurred.

2. Determine if the throw occurred in a `try` block.

3. If so, compare the type of the exception against the type of each `catch` clause.

4. If the types match, pass the control to the body of the `catch` clause.

5. If the exception is not within a `try` block or if none of the `catch` clauses match, then

   (a) destroy any active local objects

   (b) unwind the current function from the stack

   (c) go to the next active frame and repeat the above steps.

An exception table is stored in each activation frame which lists the exceptions that are caught in that frame with pointers to corresponding handlers.
Exceptions can make writing code (for the programmer) and generating code (for the compiler) difficult in some cases. Consider the following example.

```
void exceptionTest() {
   Foo *f;
   f = bar(); // object created inside the bar

   Foo f1; // created by the compiler
   f = bar1(f1);
   delete f;
}
```

If an exception is thrown in `bar()`, then the activation record can be just popped from the stack. If an exception is thrown in `bar1()`, then the object `f1` has to be destructed before popping out. So the semantics of different regions in the same block is different.
Now whose responsibility is it to free memory for `f`? It is the programmer's responsbility. So the code should look as follows.

```
void exceptionTest() {
   Foo *f, *g;
   try {
      f = bar(); // object created inside the bar

      Foo f1; // created by the compiler
      g = bar1(f1);
   } catch (...) {
      delete f;
      delete g;
      throw;
   }
```

```
   delete f;
}
```

This code is difficult to write as well as read. Java has two improvements.

1. Automatic garbage collection
2. A `finally` clause which is a single exit point for the `try` clause. An "equivalent" Java code is shown below.

```
void exceptionTest() {
   Foo f, g;
   try {
      f = bar();
      Foo f1 = new Foo();
      g = bar1(f1);
   } catch (Exception e) {
     /*   ... */
   } finally {
     /*  do some cleaning up */
     /* always executed: whether or not exception */
   }
}
```

Note that in Java there is no need to release memory. But if `f` and `g` are some critical resources, they they can be handled in `finally`.

12.8 Life cycle

The following steps are associated with a working program.

design: The task is divided into modules.

development: The modules are developed independently.

compilation: The modules are compiled and tested independently.

loading: For a complete working system, each of the compiled module should be loaded for linking. For example, the default Java loader looks for files in directories in `CLASSPATH`. Java permits the use of custom class loaders which load a class, say, from a remote site.

linking: The compiled modules have typically have external references. Linking is the process of resolving these references. The output of the linking process is an executable file which is self-contained (does not contain external refrences.)

running: The executable is run.

Any of the above steps can result in an error.
A self-contained executable is typically very large. This is especially true if the program uses standard libraries like networking, windowing, etc. Since these standard libraries are available in most locations, they need not be attached to the executable. The linking can take place at runtime. Such libraries are called *dynamically linked libraries* (DLLs).

12.8.1 Dynamic loading example

Execute the following Java code to convince yourself of dynamic loading.

```
public class Flower {
    static {
       System.out.println("Loading Flower");
    }
}
public class Frog {
    static {
       System.out.println("Loading Frog");
    }
}
public class Snake {
    static {
       System.out.println("Loading Snake");
    }
}
class Site {
   public static void main(String[] args) {
      System.out.println("Inside main");
      new Frog();
      System.out.println("After creating frog");
      new Flower();
      System.out.println("After creating flower");
      new Snake();
      System.out.println("After creating snake");
   }
}
```

12.8.2 Name conflicts

If a system is developed through a team of programmer, it is very likely that linking reveals some name conflicts – same name being used in different modules for different objects. Some kind of hierarchical naming convention is required for avoiding name conflicts.

Java provides the notion of packages to qualify names. The name, say, `foo` in packages `p1` and `p2` can be distinguised by fully qualifying it with the package name: `p1.foo` and `p2.foo`.

12.9 Threads

Most operating systems provide a facility for process creation. Processes execute independently. Each process has its own name space. Threads are a lightweight version of processes. All the threads share the same name space.
Each thread at runtime has its own program counter and stack. Mutual exclusion of threads is facilitated by hardware support like test-and-set instruction.
It is helpful to compare the synchronization facility in Java and, say, C. Java has language support for synchronization. C does not. To get the same functionality, one should use libraries. A typical call in C using semaphores looks like

```
sema_t tellers; // semaphore declaration
sema_init(&tellers, /* other args */); // initialization

sema_wait(&tellers); // lock
/* do stuff */
sema_post(&tellers); // release
```

While this code has the same functionality, the rule that every wait should be followed by a post cannot be checked by the compiler. The use of `synchronized(teller) { /* ... */ }` syntax in Java allows the compiler to check for these conditions.