159.341 Programming Languages, Algorithms & Concurrency

Imperative Programming (Part 3)

Daniel Playne
d.p.playne@massey.ac.nz

Program Composition

To construct real programs we must be able to combine our declarations and statements together.

There are (at least) four hierarchical units to discuss:

- Blocks
- Routines (procedures, functions and operators)
- Modules/packages
- Programs

Program Composition

Routines and ***programs*** have been recognised since the earliest days of programming.

Blocks of statements where introduced around 1960 in Algol60.

Modules appeared later in the mid 1970's after it was found that routines did not scale as well as expected.

Code Blocks

Blocks are the smallest grouping of declarations and statements, generally delimited with some special syntax such as `begin end` or `{ }`.

A code block defines a ***scope*** or a set of names that are only known locally within that block (may override identifiers from a higher-level of scope).

Most languages have an outermost level of declaration - the ***global scope*** where globals are declared and are generally accessible to all parts of your program.

Code Blocks

A variable declared within a code block will go out of scope at the end of that block.

```
{
  int t;
  // swap x and y
  t = x; x = y; y = t;
}
```

```
declare
  T: Integer;
begin
  - swap x and y
  T := X; X := Y; Y := T;
end;
```

Some languages such as C++ allow declarations and statements to be freely mixed together within the code block whereas others like Ada require separate `declare`...`begin`...`end` parts of a block.

Scope Rules

When there are ***nested blocks***, each block defines its own level of scope, so there may be situations where a given name may have several overlapping declarations.

The scope rules must determine which declaration is referred to for each use of an identifier.

"If there there are lots of variables of the same name, which one does the programmer really mean, in a particular scope?"

Scope Rules

Most languages use a simple “start from the innermost and work out” approach:

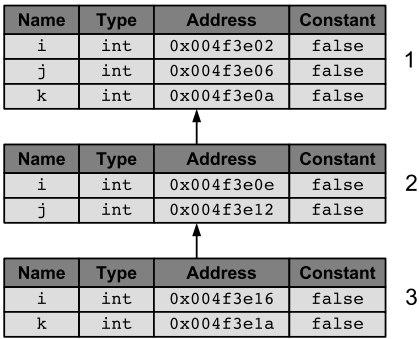
```
{
   int i, j, k;
   ...
   {
      int i, j;
      ...
      {
         int i, k;
         ...
      }
   }
}
```

```
declare
   I, J, K: Integer;
begin
   ...
   declare
      I, J: Integer;
   begin
      ...
      declare
         I, K: Integer;
      begin
         ...
      end;
   end;
end;
```

Scope Rules

Scope is often represented in the symbol table as either a list of different symbol tables or markers within a symbol table representing a different level of scope.

Name	Type	Address	Constant	
i	int	0x004f3e02	false	1
j	int	0x004f3e06	false	
k	int	0x004f3e0a	false	
i	int	0x004f3e0e	false	2
j	int	0x004f3e12	false	
i	int	0x004f3e16	false	3
k	int	0x004f3e1a	false	



Binding

The process of determining which declaration an identifier refers to is called ***binding***

If the process is determined at compile time it is called ***static binding***

Some (mostly interpreted) languages allow ***dynamic binding*** where the binding is determined at run-time.

Binding

Binding usually considers the following:

- ***block*** of a name - the block it was declared in.
- ***scope*** of a declaration and of the name in it - the part of the block over which the declaration extends.
- ***visibility range*** (or just visibility) of a declaration is the parts of the scope in which that identifier will be bound to the declaration.
- ***contextual scope rules*** - context may help resolve the appropriate declaration.

Contextual Scope Rules

If we have no overloading in our language (like C) we do not need anything other than syntactic scope rules.

However, to resolve overloading then ***contextual scope rules*** will be needed (sometimes also known as ***semantic scope rules***).

Contextual Scope Rules

Ada needs these to resolve what `Cents` means in the following example:

```
declare
  function Cents(V: Float) return Integer is
  begin
    return Integer(V*100.0);
  end Cents;

  function Cents(S: String) return Integer is
  begin
    -- convert a string like "19.95" to 1995 and return
  end Cents;
begin
  Put(Cents(18.95));
  Put(Cents("18.95"));
end
```

Subprograms

Subprogram is a collective name for those groupings of statements and expressions that must be invoked explicitly and which guarantee to return the flow of control to the calling point.

- ***Procedures***, which return no value.
- ***Functions***, which return a value.
- ***Operators***, which are like functions but have different syntax.

The collective term for procedures and functions is ***routine*** (although sometimes the word ***procedure*** is used regardless of whether a value is returned or not).

Routines

A named and parameterised block - comes from a ***routine declaration*** consisting of a ***routine header*** (the name and types of arguments etc) and a ***routine body*** (the statements of the routine).

A routine can be activated by a ***routine call/invocation*** which transfers the parameters and flow of control to the routine.

The routine declaration defines the ***formal parameters*** and the ***actual*** parameters are defined by the routine call.

Routines - Activations

Each routine in an imperative program has exactly one caller (at any one time) and only one routine can be active at once (in a serial program).

The ***activation record*** (all the local variables and linkage information etc) is created on the ***stack***.

This does allow for recursive self-calling, each call has a separate activation record, if data is to be retained between calls it must be done so with ***static data*** or global variables.

When routine ***S*** calls routine ***T***, ***T*** is active while ***S*** is suspended.

Routines - Parameters

Traditionally parameters are passed by matching the type, number and order of the formals with the actuals - known as ***positional parameter passing***

An alternative approach is ***keyword parameter passing***, where the syntax allows us to pass parameters in any order by naming them.

e.g.
`Insert(Source => A, Dest =>B);`
is Ada code that invokes the routine `Insert` with two parameters indicated by the keywords `Source` and `Dest`.

Routines - Parameters

Parameters may be passed by:

- ***Call-by-value*** makes a copy of the item, can be slow for large data items.
- ***Call-by-reference*** allows the routine to access the actual variable - like aliasing.
- ***Call-by-result*** the effect on the actual parameter only occurs when the routine call is finished.

C does not have ***call-by-reference*** (C++ does) but it can be faked by passing a pointer (by value) to the routine. C also does not have call-by-result.

Routines - Parameters

Call-by-sharing is a term first used in the language CLU and is a technique used in languages such as Python, Java, Ruby and others (although these languages often do not use the term).

This type of parameter passing often applies to languages based on objects and not primitive types - in Java an object is not passed to a function but rather a reference to that object.

This is not the same as pass-by-reference as any assignment to the parameter does not affect the object in the caller's scope. However, if the object is mutable then any change to the object in the function will affect the original.

Routines - Parameters

Some languages such as modern Fortran and Ada allow various attributes to state whether a parameter is intended for input, output or both with respect to the routine.

This extra information (often known as the ***intent*** for the parameter) allows for extra compile-time checking such as checking that an input parameter is not assigned to etc.

Routines - Forward Declarations

Declaration-before-use can introduce problems for mutually recursive routines - there is no way to define two mutually recursive routines ***F*** and ***G*** and still have declaration-before-use.

A ***forward-declaration*** of a name ***N*** says that the name ***N*** exists and that a genuine declaration for it will follow.

Compiler can therefore use information about ***N*** even before it is fully defined.

Routines - Forward Declarations

```
void f(void);
void g(void);

void f(void)
{
  ...
  g();      /* use g() */
  ...
}

void g(void)
{
  ...
  f();      /* use f() */
  ...
}
```

```
procedure F;
procedure G;

procedure F is
begin
  ...
  G;        -- use G
  ...
end F;

procedure G is
begin
  ...
  F;        -- use F
  ...
end G;
```

Figure 2.23 The use of forward declarations.

Routines - Stack Regime

A routine's local data exists only between the time it is called and when it returns. This allows efficient memory allocation model based on a stack.

However, it does make it hard for data retention between calls, some data might be conceptually local to the routine so we need some mechanism for this.

Algol has ***own variables***, C uses ***static locals*** - one value for the routine shared and accessible between all recursive calls.

Status of Subprograms

- Can routine declarations nest (can we have a routine inside a routine)
  C: No, Ada: Yes.
- Can routine names be passed as parameters
  C & Ada: Yes.
- Can routine names be stored in data structures
  C & Ada :Yes.
- Are routines data?

Languages differ on many of these issues, e.g. Java does not have function pointers but instead supports reflection to manipulate the names of classes inside programs.

Modules

Blocks and routines force a nesting structure that eventually starts to get in the way rather than help (as the programs get bigger).

Modules allow a program to be separated into independent pieces or ***modules*** that bind together related types, variables, routines etc

A ***module*** consists of two parts:

- ***Specification part*** - describes an interface that is provided by the module.
- ***Implementation part*** - source code seen only by the module programmer(s).

Modules

A well-designed module should be responsible for some distinct functionality which can be separated from the rest of the program.

This can make the program much easier to work with as the internal workings of the module are hidden from the rest of the program.

Two modules that provide the same interface could be interchanged with each other without affecting any other part of the program.

Program Composition Summary

- Blocks
- Scope
- Subprograms
- Parameter Passing
- Modules