∼ Topic IV ∼

Block-structured procedural languages
Algol and Pascal

References:

- **Chapters 5 and 7,** of *Concepts in programming languages* by J. C. Mitchell. CUP, 2003.
- **Chapters 10(§2) and 11(§1)** of *Programming languages: Design and implementation* (3RD EDITION) by T. W. Pratt and M. V. Zelkowitz. Prentice Hall, 1999.

- **Chapter 5** of *Programming languages: Concepts & constructs* by R. Sethi (2ND EDITION). Addison-Wesley, 1996.
- **Chapter 7** of *Understanding programming languages* by M Ben-Ari. Wiley, 1996.

Parameters

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.

Example: 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:

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

In Ada it is possible to use *named association* in the call:

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

? What about in ML? Can it be simulated?

This is commonly used together with *default parameters*:

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

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

Parameter passing

The way that actual parameters are evaluated and passed to procedures depends on the programming language and the kind of *parameter-passing mechanisms* it uses.

The main distinction between different parameter-passing mechanisms are:

- the time that the actual parameter is evaluated, and
- the location used to store the parameter value.

NB: The *location* of a variable (or expression) is called its *L-value*, and the *value* stored in this location is called the *R-value* of the variable (or expression).

Parameter passing

Pass/Call-by-value

- In *pass-by-value*, the actual parameter is evaluated. The value of the actual parameter is then stored in a new location allocated for the function parameter.

- Under *call-by-value*, a formal parameter corresponds to the value of an actual parameter. That is, the formal `x` of a procedure `P` takes on the value of the actual parameter. The idea is to evaluate a call `P(E)` as follows:

  `x := E`;

  execute the body of procedure `P`;

  if `P` is a function, return a result.

Parameter passing

Pass/Call-by-reference

- In *pass-by-reference*, the actual parameter must have an L-value. The L-value of the actual parameter is then bound to the formal parameter.

- Under *call-by-reference*, a formal parameter becomes a synonym for the location of an actual parameter. An actual reference parameter must have a location.

Example:

```
program main;
begin
  function f( var x: integer; y: integer): integer;
    begin
      x := 2;
      y := 1;
      if x = 1 then f := 1 else f:= 2
    end;

  var z: integer;
  z := 0;
  writeln( f(z,z) )
end
```

The difference between call-by-value and call-by-reference is important to the programmer in several ways:

- **Side effects.** Assignments inside the function body may have different effects under pass-by-value and pass-by-reference.
- **Aliasing.** Aliasing occurs when two names refer to the same object or location.

  Aliasing may occur when two parameters are passed by reference or one parameter passed by reference has the same location as the global variable of the procedure.

- **Efficiency.** Pass-by-value may be inefficient for large structures if the value of the large structure must be copied. Pass-by-reference maybe less efficient than pass-by-value for small structures that would fit directly on stack, because when parameters are passed by reference we must dereference a pointer to get their value.

Parameter passing

Pass/Call-by-value/result

Call-by-value/result is also known as *copy-in/copy-out* because the actuals are initially copied into the formals and the formals are eventually copied back out to the actuals.

Actuals that do not have locations are passed by value. Actuals with locations are treated as follows:

1. *Copy-in phase*. Both the values and the locations of the actual parameters are computed. The values are assigned to the corresponding formals, as in call-by-value, and the locations are saved for the copy-out phase.
2. *Copy-out phase*. After the procedure body is executed, the final values of the formals are copied back out to the locations computed in the copy-in phase.

Examples:

- A parameter in Pascal is normally passed by value. It is passed by reference, however, if the keyword `var` appears before the declaration of the formal parameter.

  ```
  procedure proc(in: Integer; var out: Real);
  ```

- The only parameter-passing method in C is call-by-value; however, the effect of call-by-reference can be achieved using pointers. In C++ true call-by-reference is available using *reference parameters*.

- Ada supports three kinds of parameters:
  1. `in` parameters, corresponding to value parameters;
  2. `out` parameters, corresponding to just the copy-out phase of call-by-value/result; and
  3. `in out` parameters, corresponding to either reference parameters or value/result parameters, at the discretion of the implementation.

Parameter passing

Pass/Call-by-name

The Algol 60 report describes *call-by-name* as follows:

1. Actual parameters are textually substituted for the formals. Possible conflicts between names in the actuals and local names in the procedure body are avoided by renaming the locals in the body.

2. The resulting procedure body is substituted for the call. Possible conflicts between nonlocals in the procedure body and locals at the point of call are avoided by renaming the locals at the point of call.

Block structure

- In a *block-structured language*, each program or subprogram is organised as a set of nested blocks.

  A *block* is a region of program text, identified by begin and end markers, that may contain declarations local to this region.

- *In-line (or 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.

- Block structure was first defined in Algol. Pascal contains nested procedures but not in-line blocks; C contains in-line blocks but not nested procedures; Ada supports both.
- *Block-structured languages* are characterised by the following properties:
  - New variables may be declared at various points in a program.

- Each declaration is visible within a certain region of program text, called a block.
- When a program begins executing the instructions contained in a block at run time, memory is allocated for the variables declared in that block.
- When a program exits a block, some or all of the memory allocated to variables declared in that block will be deallocated.
- An identifier that is not delcared in the current block is considered global to the block and refers to the entity with this name that is declared in the closest enclosing block.

Algol

HAD A MAJOR EFFECT ON LANGUAGE DESIGN

- The Algol-like programming languages evolved in parallel with the LISP family of languages, beginning with Algol 58 and Algol 60 in the late 1950s.

- The most prominent Algol-like programming languages are Pascal and C, although C differs from most of the Algol-like languages in some significant ways. Further Algol-like languages are: Algol 58, Algol W, Euclid, *etc.*

- The main characteristics of the Algol family are:
  - the familiar semicolon-separated sequence of statements,
  - block structure,
  - functions and procedures, and
  - static typing.

Algol 60

- Designed by a committee (including Backus, McCarthy, Perlis) between 1958 and 1963.
- Intended to be a general purpose programming language, with emphasis on scientific and numerical applications.
- Compared with FORTRAN, Algol 60 provided better ways to represent *data structures* and, like LISP, allowed functions to be called recursively.

  Eclipsed by FORTRAN because of the lack of I/O statements, separate compilation, and library; and because it was not supported by IBM.

Algol 60

Features

- Simple statement-oriented syntax.
- Block structure.
- Recursive functions and stack storage allocation.
- Fewer ad hoc restrictions than previous languages (*e.g.*, general expressions inside array indices, procedures that could be called with procedure parameters).
- A primitive *static type system*, later improved in Algol 68 and Pascal.

Algol 60

Some trouble spots

- The Algol 60 type discipline had some shortcomings.

  For instance:

  - Automatic type conversions were not fully specified (*e.g.*, `x := x/y` was not properly defined when `x` and `y` were integers—is it allowed, and if so was the value rounded or truncated?).
  - The type of a procedure parameter to a procedure does not include the types of parameters.
  - An array parameter to a procedure is given type array, without array bounds.

- Algol 60 was designed around two parameter-passing mechanisms, *call-by-name* and *call-by-value*.

  Call-by-name interacts badly with side effects;
  call-by-value is expensive for arrays.

- There are some awkward issues related to control flow, such as memory management, when a program jumps out of a nested block.

Algol 60 `procedure` types[a]

In Algol 60, the type of each formal parameter of a procedure must be given. However, `proc` is considered a type (the type of `proc`edures). This is much simpler than the ML types of function arguments. However, this is really a type loophole; because calls to procedure parameters are not fully type checked, Algol 60 programs may produce run-time errors.

Write a procedure declaration for `Q` that causes the following program fragment to produce a run-time type error:

```
proc P ( proc Q )
  begin Q(true) end;

P(Q);
```

where `true` is a Boolean value. Explain why the procedure is statically type correct, but produces a run-time type error. (You may assume that adding a Boolean to an integer is a run-time error.)

[a] Exercise 5.1 of *Concepts in programming languages* by J. Mitchell, CUP, 2003.

Algol 60 pass-by-name
Copy rule

```
real procedure sum(E,i,low,high); value low, high;
  real E; integer i, low, high;
  begin
    sum:=0.0;  for i := low step 1 until high do sum := sum+E;
  end

integer j; real array A[1:10]; real result;
for j:= 1 step 1 until 10 do A[j] := j;
result := sum(A[j],j,1,10)
```

By the *Algol 60 copy rule*, the function call to `sum` above is equivalent to:

```
  begin
    sum:=0.0;  for j := 1 step 1 until 10 do sum := sum+A[j];
  end
```

Algol 60 pass-by-name[a]

The following Algol 60 code declares a procedure `P` with one pass-by-name integer parameter. Explain how the procedure call `P(A[i])` changes the values of `i` and `A` by substituting the actual parameters for the formal parameters, according to the Algol 60 copy rule. What integer values are printed by the `program`? And, by using pass-by-value parameter passing?

```
begin
  integer i; i:=1;
  integer array A[1:2]; A[1]:=2; A[2]:=3;

  procedure P(x); integer x;
    begin  i:=x; x:=1  end

  P(A[i]);  print(i,A[1],A[2])
end
```

[a] Exercise 5.2 of *Concepts in programming languages* by J. Mitchell, CUP, 2003.

Algol 68

- Intended to remove some of the difficulties found in Algol 60 and to improve the expressiveness of the language.

  It did not entirely succeed however, with one main problem being the difficulty of efficient compilation (*e.g.*, the implementation consequences of higher-order procedures where not well understood at the time).

- One contribution of Algol 68 was its *regular, systematic type system*.

  The types (referred to as *modes* in Algol 68) are either *primitive* (`int`, `real`, `complex`, `bool`, `char`, `string`, `bits`, `bytes`, `semaphore`, `format`, `file`) or *compound* (`array`, `structure`, `procedure`, `set`, `pointer`).

Type constructions could be combined without restriction. This made the type system seem more systematic than previous languages.

- Algol 68 memory management involves a *stack* for local variables and *heap* storage. Algol 68 data on the heap are explicitly allocated, and are reclaimed by *garbage collection*.
- Algol 68 parameter passing is by value, with pass-by-reference accomplished by pointer types. (This is essentially the same design as that adopted in C.)
- The decision to allow independent constructs to be combined without restriction also led to some complex features, such as assignable pointers.

Algol innovations

- Use of BNF syntax description.
- Block structure.
- Scope rules for local variables.
- Dynamic lifetimes for variables.
- Nested `if-then-else` expressions and statements.
- Recursive subroutines.
- Call-by-value and call-by-name arguments.
- Explicit type declarations for variables.
- Static typing.
- Arrays with dynamic bounds.

Pascal

- Designed in the 1970s by Niklaus Wirth, after the design and implementation of Algol W.

- Very successful programming language for *teaching*, in part because it was designed explicitly for that purpose.

  Also designed to be compiled in one pass. This hindered language design; *e.g.*, it forced the problematic `forward` declaration.

- Pascal is a *block-structured* language in which *static scope* rules are used to determine the meaning of nonlocal references to names.

- A Pascal program is always formed from a single main program block, which contains within it definitions of the subprograms used.

  Each block has a characteristic *structure*: a header giving the specification of parameters and results, followed by constant definitions, type definitions, local variable declarations, other nested subprogram definitions, and the statements that make up the executable part.

- Pascal is a *quasi-strong, statically typed* programming language.

  An important contribution of the Pascal *type system* is the rich set of data-structuring concepts: *e.g.* enumerations, subranges, records, variant records, sets, sequential files.

- The Pascal *type system* is more expressive than the Algol 60 one (repairing some of its loopholes), and simpler and more limited than the Algol 68 one (eliminating some of the compilation difficulties).

  A restriction that made Pascal simpler than Algol 68:

```
procedure Allowed( j,k: integer );
procedure AlsoAllowed( procedure P(i:integer);
                        j,k: integer );
procedure
NotAllowed( procedure MyProc( procedure P(i:integer) ) );
```

- Pascal was the first language to propose index checking.
- Problematically, in Pascal, the index type of an array is part of its type. The Pascal standard defines *conformant array parameters* whose bounds are implicitly passed to a procedure. The Ada programmig language uses so-called *unconstrained array types* to solve this problem.

  The subscript range must be fixed at compile time permitting the compiler to perform all address calculations during compilation.

```
procedure
  Allowed( a: array [1..10] of integer ) ;
procedure
  NotAllowed( n: integer; a: array [1..n] of integer ) ;
```

- Pascal uses a mixture of *name* and *structural* equivalence for determining if two variables have the same type.

  Name equivalence is used in most cases for determining if formal and actual parameters in subprogram calls have the same type; structural equivalence is used in most other situations.

- Parameters are passed by value or reference.

  Complete static type checking is possible for correspondence of actual and formal parameter types in each subprogram call.

Pascal variant records

Variant records have a part common to all records of that type, and a variable part, specific to some subset of the records.

```
type
  kind = ( unary, binary) ;
type                              { datatype                        }
  UBtree = record                 {      'a UBtree = record of      }
    value: integer ;              {           'a * 'a UBkind        }
    case k: kind of               { and 'a UBkind =                 }
      unary: ^UBtree ;            {           unary of 'a UBtree }
      binary: record              {         | binary of             }
        left: ^UBtree ;           {             'a UBtree *         }
        right: ^UBtree            {             'a UBtree ;         }
        end
  end ;
```

Variant records introduce *weaknesses* into the type system for a language.

1. Compilers do not usually check that the value in the tag field is consistent with the state of the record.
2. Tag fields are optional. If omitted, no checking is possible at run time to determine which variant is present when a selection is made of a field in a variant.

Summary

- The Algol family of languages established the command-oriented syntax, with blocks, local declarations, and recursive functions, that are used in most current programming languages.
- The Algol family of languages is statically typed, as each expression has a type that is determined by its syntactic form and the compiler checks before running the program to make sure that the types of operations and operands agree.