

ALGOL 60

Introduced

Formal description

- BNF Language Description**

Nested structure

- Block Structured**

- Lexical scoping**

Multiple types of procedure arguments

- "call by name"**

- "call by value"**

Orthogonal design

All variables declared as to type

- "own" variables**

Recursive procedures

- Procedures could be declared in procedures with downward scope**

Also defined a publication language so that programs would look nicer in print

But

- No IO specification**

Popular in Europe - not so much in the US

Many compilers built

Appealed to the researcher but not the business programmer

Not supported by IBM

BNF

The language is described by metalinguistic formulas like

$\langle ab \rangle ::= (\mid [\mid \langle ab \rangle (\mid \langle ab \rangle \langle d \rangle$

$\langle d \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The idea is that the syntactic entity on the left of the ::= can be replaced by any of the alternatives on the right separated by the | symbol

If the result has more syntactic entities enclosed in < > then further substitutions can be done

When no further substitutions can be done then the result is an ALGOL 60 program

The above example can produce

[[(((1(37(

(12345(

[86

Interesting features of ALGOL 60

Used many special symbols not found on the I/O devices of the day (or today)

ALGOL 60 had reserved words that were considered to be a token and not made up of its individual characters

Generally represented as bold underlining

begin end true false

The = sign was reserved for logical comparisons (like == in C++ or Java) and not for assignment

The := symbol was used for assignment

The language had a goto (as did most languages of this time period)

Array subscripts could be declared with lower and upper bounds defined at run time

The semicolon separated statements - it did not terminate statements

Comments looked like statements beginning with the keyword comment and ending with a semicolon

Some expressions had types that could change at runtime

An integer to an integer power was an integer for positive powers and real for negative powers

own variables were like static variables in C++ - there was only one instance of such a variable even though declared in a recursive procedure

procedures without arguments were called by naming them - no following () in the declaration or the call

Designational expressions were expressions whose value was a suitable target of a goto

A switch was an array of designational expressions

Procedures

Types of arguments did not have to be specified unless they were passed by value

There were two ways to pass parameters to procedures (there was nothing like the FORTRAN "call by reference")

Call by value

The local variable in the procedure was initialized by the calling argument

Modification of this local variable would not affect anything external to the procedure

Call by name

The actual argument effectively replaced every occurrence of the variable in the body of the procedure before the body was executed

This meant that the code to evaluate the actual argument was executed every time the argument was referenced

Example

```
begin procedure assign(a, i);  
    integer i;  
    begin i := i + 1;  
        a[i] := 6.5  
    end assign;  
  
    integer j;  
    array x[3:12];  
    j := 5;  
    assign(x, j);  
end
```

The call assign(x, j) would increment j to 6 and then assign 6.5 to x[6]

The last assignment to a variable with the same name as that of the procedure was used as the return value of the procedure

Labels could be arguments to a procedure

The for Statement

The for loop allowed concatenating several looping constructs (separated by commas) which were executed in sequence

When one looping construct finished the next one was executed
`for j := 1,5 step 3 until 11, j+1 while j < 50 do xxx;`

This statement would execute the controlled statement xxx with

`j = 1`

`j = 5; j = 8; j = 11;`

`j = 12; j = 13; ... j = 49;`