CS33

A PROGRAMMING LANGUAGE FOR THE 360 COMPUTERS

BY

NIKLAUS WIRTH

TECHNICAL REPORT CS33
DECEMBER 24, 1965

COMPUTER SC IENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UN IVERS ITY

The PL360 compiler on the B5500 computer.

The compiler is a syntax directed one pass compiler designed according to the principles outlined in Technical Report CS20. The following rules and restrictions apply to the version-of Dec. 1, 1965:

1. The length of the significant part of identifiers is 6 characters,
2. No real numbers and no strings are available.
3. Base register assignment is fixed: Registers RA - RF are used as base address registers and should therefore not be used within the program.
4. Every program must be terminated by an @ character.
5. Composite basic symbols, such as begin, end, etc., are written as BEGIN, END, etc., i.e., have the form of identifiers and may therefore not be used as such. Note: goto is written as GOTO without space between GO and TO.
6. No blank spaces may occur within numbers and identifiers.
7. The compiler is on the USE tape and is called by the ALGOL statement

ZIP("-PL360-", "MCP-USE")

8. The following function identifiers are defined in the compiler:

LA, EX, CVB, CVD, STM, LM, MVI, MVC, TR, ED,

IC, STC, SRDA, SRDL, SLDA, SLDL, SVC, SPM

A Programming Language for the 360 Computers

Table of Contents

A Programming Language for the 360 Computers*

by

Niklaus Wirth

Introduction

This paper is a preliminary definition of a programming language which is specifically designed for use on IBM 360 computers, and is therefore appropriately called PL360.

The intention is to present a programming tool which (a) closely reflects the particular structure of the 360 computer, and (b) is a superior notation to present Assembly Codes with respect to presentation and documentation of algorithms. As a consequence of (a), it enables a programmer to design programs mentioning explicitly features of this machine in a degree impossible in "higher level" languages.

It is also felt that a highly structured language is most appropriate (a) to promote the intelligibility of texts for the human user and (b) to encourage this user to properly structure his algorithms not on paper only, but in his mind as well. The language is therefore a phrase structure language containing many constructions which quite obviously correspond to a single 360 machine instruction (cf. [1]).

Moreover, it is hoped that through certain conventions (not mentioned in this preliminary paper) concerning the use of general registers as base address registers, programs written in PL360 can be efficiently run under a time-sharing monitor without requiring the presence of additional sophisticated relocation hardware (Model 67).

---

*/ This work was partially supported by the National Science Foundation (GP 4053) and the Computation Center of Stanford University.,

Presently, a compiler for PL360 is available on the B5500 computer. This compiler is mainly intended to serve as a temporary tool for a bootstrapping process: The compiler is being rewritten in its own language and then becomes automatically available on the 360 computer. Indeed, the primary purpose of this project is to obtain a convenient tool for the development of other compilers (in particular ALGOL X) and monitor systems, where a considerable degree of machine-orientation and -dependence is desirable, but where an adequate standard of program documentation is of no less importance.

Reference:

[1] IBM System/360 Principles of operation. IBM Systems Reference Library, A22-6821-1.

1. Definitions, Notation

The structure of the language PL360 is defined by a phrase structure system. Its productions have the general form

$$\langle A \rangle ::= x_1 \mid x_2 \mid \ldots \mid x_n$$

which is an abbreviation for the set of productions

$$\langle A \rangle ::= x_1$$
$$\langle A \rangle ::= x_2$$
$$\ldots$$
$$\langle A \rangle ::= x_n$$

and where $\langle A \rangle$ is a single nonterminal symbol, and x_i is a string of terminal and nonterminal symbols.

Terminal symbols of the phrase structure system are either so-called basic symbols or character strings. Basic symbols may consist of one or more characters, i.e., typographical entities of a lower order than basic symbols; the set of characters and the decompositions of basic symbols into characters are not defined here, and may depend on the hardware available to a particular implementation. Character strings are sequences of characters delineated by string quotes.

The set of basic symbols is defined in section 2.

Nonterminal symbols, sometimes also called "syntactic entities", are denoted by letter strings enclosed by the brackets $\langle$ and $\rangle$. In addition to these letter strings, the script letters $\mathcal{T}$, $\mathcal{U}$, and $\mathcal{V}$ may occur; a production containing one or more of these letters stands for the set of productions in each of which this letter has been replaced by a terminal word produced from this letter according to the following syntax:

$\mathcal{V}$ ::= $\mathcal{U}$|byte

$\mathcal{U}$ ::= integer|long integer|real|long real

$\mathcal{T}$ ::= general|floating|floating double

If the same letter occurs more than once in the production, then all occurrences of the letter have to be replaced by the same terminal word.

Example:

The production

⟨$\mathcal{V}$ variable⟩ ::= ⟨$\mathcal{V}$ variable identifier⟩

(cf. section 6) stands for the five productions

(integer variable) ::= (integer variable identifier)

(long integer variable) ::= (long integer variable identifier)

(real variable) ::= (real variable identifier)

(long real variable) ::= (long real variable identifier)

(byte variable) ::= (byte variable identifier)

In order to provide explanations for the meaning (semantics) of PL360 texts, the letter sequences denoting syntactic entities (nonterminal symbols) have been chosen to be English words describing approximately the nature of that entity. Where words which have appeared in this manner are used elsewhere in the text, they refer to the corresponding syntactic definition.

Definition: A sequence of basic symbols (and character strings) is a PL360 program, if and only if it can be produced from the symbol (block) by the productions listed in sections 3-8, and a meaning can be attributed to it by the accompanying semantic explanations.

2. Basic Symbols

A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|

a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|

0|1|2|3|4|5|6|7|8|9|

+|-|*|/|and|or|xor|shl|shr|shll|shrl|

<|≤|=|≠|≥|>|:=|neg|abs|

goto|if|then|else|while|do|for|step|until|case|of|

begin|end|(|)| ,|.|;|:|

integer|real|byte|long|array|procedure|

overflow|#|

3. Identifiers

3.1. Syntax

(letter) ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

(identifier) ::= ⟨letter⟩|⟨identifier⟩⟨letter⟩| (identifier)(digit)

⟨𝔙 identifier) ::= (identifier)

⟨𝔉 register⟩ ::= (identifier)

(procedure identifier) ::= (identifier)

(function identifier) ::= (identifier)

3.2. Semantics

Identifiers have no inherent meaning, but serve for the identification of registers, variables, procedures and labels. They may be chosen freely, with the exception of

R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,RA,RB,RC,RD,RE,RF

which designate the 16 general registers, and

F0,F2,F4,F6,F01,F23,F45,F67

which designate the floating- and floating double registers respectively.

Every identifier in a program must be defined. If it designates a register, definition is implied; if it designates a variable or a procedure, then this occurs through appropriate declarations (cf. section 5), or if it designates a label, then it occurs through a label definition (cf. 7.3.).

The applicability of the rules given in the syntax (cf. 3.1.) depends upon the definition of the identifier under consideration as follows:

(a) If the identifier is Rn, where n is either 0,1,2,..., 9, A,..., F, it designates the n'th general register (the letters A ... F have to be understood as numbers in hexadecimal notation, i.e., 10 . . . 15). Otherwise,

(b) if the identifier is Fn, where n = 0,2,4, or 6, then it designates the n'th floating register. Otherwise,

(c) if the identifier is Fnm, where n = 0,2,4, or 6, and m = n+1, then it designates the n'th floating double register. Otherwise,

(d) if the identifier has been defined in a $\mathfrak{v}$ variable declaration in the smallest block embracing the given occurrence, then it identifies that $\mathfrak{v}$ variable and is said to be a $\mathfrak{v}$ variable identifier; otherwise, if it has been defined in a procedure heading in that block, then it identifies that procedure and is said to be a procedure identifier; otherwise, the rules under (d) are applied considering the smallest block embracing the previously considered block, if there exists one. Otherwise,

(e) if the identifier occurs in the listing of function identifiers (cf. 7.5.), then it identifies that function.

3.3. Examples:

P

cat

RO

4. Quantities, values, and types.

The following kinds of quantities are distinguished: registers, variables, and constants. Every quantity is said to possess a value. The value of a constant is determined by its denotation. (cf. 4.1.-4.2.). The value of a register or a variable is the one most recently assigned to it. Every value is said to be of a certain type. The following types are distinguished:

integer , long integer : the value is an integer,

real , long real : the value is a real number,

byte : the value is a character.

In the computer, every value is represented by a number of binary digits in a suitably encoded manner (cf. [1]). The number of bits used in the representation of the different types of values is given as follows:

integer	16	(half word)
long integer	32	(word)
real	32	(word)
long real	64	(double word)
byte	8	(byte)

Subsequently, the denotation of constants is defined.

4.1. Numbers

4.1.1. Syntax

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

⟨unsigned integer⟩ ::= ⟨digit⟩ | ⟨unsigned integer⟩⟨digit⟩

⟨decimal integer⟩ ::= ⟨unsigned integer⟩| _ ⟨unsigned integer⟩

⟨hexadecimal digit⟩ ::= ⟨digit⟩|A|B|C|D|E|F

⟨hexadecimal integer⟩ ::= #⟨hexadecimal digit⟩|

⟨hexadecimal integer⟩⟨hexadecimal digit⟩

⟨integer number⟩ ::= ⟨decimal integer⟩|⟨hexadecimal integer⟩

⟨fraction⟩ ::= .⟨unsigned integer⟩

⟨unscaled real⟩ ::= ⟨unsigned integer⟩⟨fraction⟩|⟨fraction⟩

⟨scale factor 1⟩ ::= E⟨decimal integer⟩

⟨unsigned real⟩ ::= ⟨unscaled real⟩|⟨unscaled real⟩⟨scale factor 1⟩|
⟨unsigned integer⟩⟨scale factor 1⟩

⟨real number⟩ ::= ⟨unsigned real⟩| _ ⟨unsigned real⟩

⟨scale factor 2⟩ ::= D⟨decimal integer⟩

⟨long unsigned real⟩ ::= ⟨unscaled real⟩|⟨unscaled real⟩⟨scale factor 2⟩|
⟨unsigned integer⟩⟨scale factor 2⟩

⟨long real number⟩ ::= ⟨long unsigned real⟩| ⟨long unsigned real⟩

4.1.2. Semantics

Integers have either decimal or hexadecimal notation. Real and long real numbers use decimal notation only. _ denotes a monadic minus sign. The scale factor is expressed as an integral power of 10.

4.1.3. Examples:

0	1E8
1066	5.37861289001D0
3.1416	#7AB3

4.2. Strings

4.2.1. Syntax

⟨string⟩ ::= ⟨sequence of characters enclosed by string quotes⟩

5. Declarations

5.1. Variable Declarations

5.1.1. Syntax

⟨𝔳 simple type⟩ ::= integer|long integer|real|long real|byte

⟨𝔳 type⟩ ::= ⟨𝔳 simple type⟩|

array ((unsigned integer)) ⟨𝔳 simple type⟩

⟨𝔳 variable declaration⟩ ::= ⟨𝔳 type⟩⟨identifier⟩|

⟨𝔳 variable declaration⟩ , ⟨identifier⟩|

@variable declaration) (⟨𝔲 number⟩)

⟨𝔳 variable declaration⟩(⟨string⟩)

5.1.2. Semantics

A variable declaration associates an identifier and a type with one or several quantities. If the type of the declaration is a simple type, then one quantity is declared, otherwise the unsigned integer between parentheses following the symbol array indicates the number of declared quantities of the specified simple type. The individual quantities can then be identified by subscripts (cf. 6.2.). The ensemble of the quantities is called an array. If a declaration is followed by one or several parenthesized numbers, then this implies that the declared quantity be initialized with the given number(s). The type of these numbers must be identical to the type of the declaration, This initial assignment of values is understood to take place only upon the first time the block in which the declaration occurs is entered.

5.1.3. Examples:

```
integer i,j
long integer m,n,q
real x,y
long real z,w
integer i(1)
array (100) integer a
array (5) integer I (21)(0)(8)(17)(39)
```

5.2. Procedure Declarations

5.2.1. Syntax'

(procedure heading) ::= procedure (identifier) ((general register));

(procedure declaration) ::= (procedure heading)(statement)

5.2.2. Semantics

Execution of the statement following a procedure heading is invoked by procedure statements (cf. 7.4.). The procedure identifier defined by the procedure heading is assumed to be unknown within the procedure declaration. Moreover, the value of the register designated in the procedure heading must not be altered during the execution of the statement following the procedure heading.

5.2.3. Examples

```
procedure P (R0); R1 := R1+x
procedure swap02 (RF);
     begin long real t; t := F01; F01 := F23; F23 := t; end
```

6. Variables and Primaries

6.1. Syntax

⟨𝔙 variable⟩ ::= ⟨𝔙 variable identifier⟩|

⟨𝔙 variable identifier⟩ ((unsigned integer))

⟨𝔙 variable identifier⟩ ((general register))

⟨𝔘 primary⟩ ::= ⟨𝔘 variable⟩|⟨𝔘 number⟩

6.2. Semantics

⟨𝔙 variable⟩ designates a declared quantity of type 𝔙. If the variable identifier is followed by an unsigned integer or a general register within parentheses, called a subscript, then the identifier must designate an array, and the integer or the current value of the register identify the individual element of the array. The subscript values designating the elements must be

(a) positive multiples of 2, if the array is of type integer,

(b) positive multiples of 4, if the array is of type long integer or real,

(c) positive multiples of 8, if the array is of type long real,

(d) positive integers, if the array is of type byte.

The first element of any array is designated by a subscript value 0. Register RO must not be used as a subscript. The values of variables may be changed by means of assignment statements (cf. 7.1.).

A primary denotes a quantity, either a variable, or a constant.

6.3. Examples:

Variables: i
I(3)
a(R5)

7. Simple Statements

Syntax

(simple statement) ::= (assignment statement)| (branch statement)|
(block)| (procedure statement)|(function statement)

7.1. Assignment Statements

7.1.1. Syntax

(simple $\mathcal{F}$ register assignment) ::= ⟨$\mathcal{F}$ register⟩ := ⟨$\mathcal{U}$ primary⟩|
⟨$\mathcal{F}$ register⟩ := ⟨$\mathcal{F}$ register⟩| ⟨$\mathcal{F}$ register⟩ := neg ⟨$\mathcal{F}$ register⟩|
⟨$\mathcal{F}$ register⟩ := abs ⟨$\mathcal{F}$ register⟩

(arithmetic operator) ::= +|-|*|/|++|--

(logical operator) ::= and|or|xor

(shift operator) ::= shl|shr|shll|shrl

⟨$\mathcal{F}$ register assignment) ::= (simple $\mathcal{F}$ register assignment)|
⟨$\mathcal{F}$ register assignment)(arithmetic operator)⟨$\mathcal{U}$ primary⟩|
⟨$\mathcal{F}$ register assignment⟩(arithmetic operator)⟨$\mathcal{F}$ register⟩|

(general register assignment) ::=
(general register assignment⟩(logical operator⟩(long integer primary⟩|
(general register assignment⟩(logical operator⟩(general register⟩|
(general register assignment)(shift operator⟩(unsigned integer)/
(general register assignment⟩(shift operator⟩(general register)

(variable assignment) ::= ⟨$\mathcal{U}$ variable⟩ := ⟨$\mathcal{F}$ register⟩

(assignment statement) ::= ⟨$\mathcal{F}$ register assignment⟩|(variable assignment)

7.1.2. Semantics

Execution of an assignment statement causes a new value to be assigned to the quantity designated on the left of the assignment operator (:=).

In the case of a simple register assignment, this value is the current value of a primary, a register, or the negative or the absolute value of a register. The types $\mathcal{U}$ of primaries which may be assigned to a register of type $\mathcal{T}$ are marked in the following Table 1:

$\mathcal{T}$ \ $\mathcal{U}$	integer	long integer	real	long real
general	*	*	*	*
floating			*	*
floating double			*	*

Table 1

The arithmetic, logical and shift operators +, -, *, /, ++, --, and, or, xor (exclusive or), shl, shr (shift left/right), shll, shrl (shift left/right logical) designate operations which are described in detail in Reference [1]. The operators ++ and -- designate unnormalized addition and subtraction if applied to floating registers, "logical' addition/subtraction if applied to general registers (cf. also [1]).

Execution of a register assignment containing one of the arithmetic or logical operators causes the designated operation to be performed on two operands and the result to be assigned to the first operand. The first operand is the register which occurs to the left of the assignment operator, and the second operand is the primary or register following the operator. In the case of a (unary) shifting operation, the operand is the designated first operand, and the number of bit positions it has to be shifted is determined either by the number, or by the current

value of the general register following the shift operator.

The types of a register ($\mathcal{T}$) and of a primary ($\mathcal{U}$) which may simultaneously be operands of an arithmetic operator are defined in the following Table 2 (the type of a register assignment is said to be the type of the register occurring to the left of its assignment operator):

$\mathcal{T}$ \ $\mathcal{U}$	integer	long integer	real	long real
general	*(1)	*		
floating			*	*
floating double			*	*

Table 2

Note (1): The combination of general register and integer primary is only permissible in the connection with the operators +, -, and * .

Execution of a variable assignment causes the current value of the designated register to be assigned to a variable. The types of the variable ($\mathcal{U}$) to which the value of a register of type $\mathcal{T}$ may be assigned, are designated in Table 1.

7.1.3. Examples

Register assignments:

```
R1   := R3
R1   := 5
RF   := i+j-m+a(R1)
R9   := R9 and R10 shll 8 or R1
F2   := F3 + 3.1416
F01  := z*w+w
```

Variable assignments:

i := R0

x := F0

w := F23

a(R1) := RF

7.2. Branch Statements

7.2.1. Syntax

⟨branch statement⟩ ::= goto ⟨identifier⟩

7.2.2. Semantics

A branching statement determines that execution of the program be continued at the place of the definition of the identifier following the symbol goto . This definition is identified by the following rules:

(1) If some label definition (cf. 7.3 .) within the smallest block embracing the branch statement contains that identifier, then this label definition designates the place where execution has to be continued. Otherwise,

(2) Rule (1) is applied considering the smallest block embracing the previously considered block.

7.3. Blocks

7.3.1. Syntax

⟨block heading⟩ ::= begin | ⟨block heading⟩⟨variable declaration⟩; |
⟨block heading⟩⟨procedure declaration⟩;

⟨label definition⟩ ::= ⟨identifier⟩:

⟨block body⟩ ::= ⟨block heading⟩ | ⟨block heading⟩⟨statement⟩; |
⟨block heading⟩⟨label definition⟩⟨statement⟩;

⟨block⟩ ::= ⟨block body⟩ end | ⟨block body⟩⟨label definition⟩ end

7.3.2. Semantics

A block introduces a new level of nomenclature: identifiers defined in variable declarations or procedure headings in the block heading or in label definitions in the block body are said to be local to that block.

Execution of a block begins with the execution of the first statement following the block heading. Upon termination of the execution of a statement, the next statement in textual sequence is executed (except in the case of a goto statement).

7.3.3. Examples

Innerproduct program with summation in double precision:

```
begin long real s; array (100) real x, y;
     FO1 := ODO;
     for R1 := 0 step 4 until 396 do
     begin F23 := x(R1) * y(R1); FO1 := FO1 + F23;
     end;
     s := FO1
end
```

Bubble sorting program:

```
begin array (100) real a;
     for R1 := 396 step -4 until 0 do
     begin R5 := R1 - 4;
          for R2 := 0 step 4 until R5 do
          Begin  6 := R2+4; FO := a(R2); F2 := a(R6);
                 if FO > F6 then
                 begin a(R2)  := F2; a(R6) := FO;
                 end;
          end;
     end;
end
```

7.4. Procedure Statements

7.4.1. Syntax

(procedure statement) ::= (procedure identifier)

7.4.2. Semantics

Execution of a procedure statement consists of the execution of the statement which, together with the procedure heading in which the procedure identifier is defined, consititutes a procedure declaration (cf. 5.2.). The value of the general register specified in that procedure heading is altered by the procedure statement.

7.5. Functions

7.5.1. Syntax

{function) ::= (function identifier)]

(function) ((integer number⟩)| ⟨function⟩(⟨T register⟩)|

(function) (@'variable))

7.5.2. Semantics

The instruction set of the system/360 processor contains instructions which cannot be expressed by any of the statements of this language (except the function statement). In order that the language be able to express the individual functions corresponding to these instructions in one single simple statement, the function statement is introduced. The individual instructions falling into this class are listed below, They are described in Reference [1].

Fixed or Floating Point Arithmetic:

- Load Negative
- Load and Test
- Halve
- Convert to Binary
- Convert to Decimal
- Load Multiple
- Store Multiple

Logical and Branching:

- Compare
- Load Address
- Insert Character
- Store Character

Furthermore, all instructions with SI and SS format belong to this category, as well as all status switching instructions.

The parameters of the function statement correspond in the order from left to right to the operand fields of an instruction.

It is suggested that the mnemonic instruction codes as defined in [1] be used as function identifiers.

7.5.3. Examples:

SVC(0)	SPM(R5)
IC(R0)(A(R1))	CVB(RF)(N)
EX(0)(instruction)	SLDL(R4)(16)
MVI(#1F)(code)	MVC(255)(a)(b)

8. Statements

Syntax

(statement) ::= ⟨simple statement⟩ | ⟨if statement⟩ |
⟨case statement⟩ | ⟨iterative statement⟩

8.1. If Statements

8.1.1. Syntax:

(relational operator) ::= < | ≤ | = | ≠ | ≥ | >

(condition) ::= ⟨𝔗 register⟩⟨relational operator⟩⟨𝔘 primary⟩ |
⟨𝔗 register⟩⟨relational operator⟩⟨𝔗 register⟩ | overflow

(if clause) ::= if (condition) then

(true part) ::= ⟨simple statement⟩ else

(if statement) ::= ⟨if clause⟩⟨statement⟩ |
⟨if clause⟩⟨true part⟩⟨statement⟩

8.1.2. Semantics:

A condition is said to be met, if the relation indicated by the relational operator holds between the two operands. The types 𝔗 and 𝔘 of the operands which may simultaneously be operands of relational operators are defined in Table 2 of section 7.1.2.

The symbol overflow designates a condition which may be met after the occurrence of a result of arithmetic operations which cannot be accepted by the computer.

The if statement expresses that execution of certain statements be dependent on certain conditions. In the construction

(if clause)(statement)

the statement is executed only if the condition contained in the if clause is met. In the construction

⟨if clause⟩⟨true part⟩⟨statement⟩

the simple statement of the true part is executed and the statement following the true part is skipped, if the condition specified by the if clause is met. Otherwise, the true part is skipped, and the statement following it is executed.

8.1.3. Examples

```
if RO > 5 then goto L
if FO < F1 then FO := FO + 1.5
if RA = RB then RO := RO or R1 else RO := RO and R1
if RO = 1 then FO1 := w+z else
     if RO = 3 then F4 := x+y else goto L
```

8.2. Case Statements

8.2.1. Syntax

(case clause) ::= case (general register) of

(case sequence) ::= (case clause) begin | (case sequence)⟨statement⟩;

(case statement, ::= (case sequence) end

8.2.2. Semantics

Execution of the case statement

case (register-k) of

begin (statement-1); . . . ; (statement-i); . . . ; (statement-n); end

consists of the execution of the i-th statement in the case sequence, where i is the current value of the general register specified by the

case clause. This value is supposed to be the ordinal number of some statement in the case sequence. The general register of the case clause must not be RO, and its value becomes undefined through the execution of the case statement.

8.2.3. Example:

case R1 of begin RO := 100; F2 := x; P; goto L; end

8.3. Iterative Statements

8.3.1. Syntax

(while clause) ::= while (condition) do

(step until) ::= step⟨integer number⟩ until (general register) |
step (integer number) until (integer primary)

(for clause) ::= for (general register assignment) (step until) do

(iterative statement-) ::= (while clause)⟨statement⟩ |
(for clause) (statement)

8.3.2. Semantics

a. An iterative statement of the form

while (condition) do (statement)

is equivalent to

L: if (condition) then begin (statement); goto L; end

b. An iterative statement of the form

for (general register) := (initial value) step (increment)
until (limit) do ⟨statement⟩

is equivalent to

```
    (general register) := (initial value);
K:  if (general register) > (limit) then goto L;
                          <
    (statement);
    (general register) := (general register) + (increment);
    goto K;
L:
```

The > sign applies, if (increment) is a positive integer, < applies, if it is a negative integer.

8.3.3. Examples

```
while R1 ≠ a(R1) do R1 := R1 + 1;
for R1 := 0 step 4 until n do R0 := R0 + a(R1);
for R3 := 1 step 1 until k do
   begin R2 := R1/R3; R1 := R1 - 1;
   end
```