CS 53

A PROGRAMMING LANGUAGE FOR THE 360 COMPUTERS

BY

NIKLAUS WIRTH

TECHNICAL REPORT NO. CS53
DECEMBER 20, 1966

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

A PROGRAMMING LANGUAGE FOR THE 360 COMPUTERS

by

Niklaus Wirth

December 20, 1966

Abstract

A programming language for the IBM 360 computers and is implementation are described. The language, called PL360, provides the facilities of a symbolic machine language, but displays a structure defined by a recursive syntax. The compiler, consisting of a precedence syntax analyser and a set of interpretation rules with strict one-to-one correspondence to the set of syntactic rules directly reflects the definition of the language.

k-th syntax rule	k-th interpretation rule
$S_0 ::= S_1S_2\ldots S_n$	$V_0 := f_k(V_1,V_2,\ldots, V_n)$

PL360 was designed to improve the readability of programs which must take into account specific characteristics and limitations of a particular computer. It represents an attempt to further the state of the art of programming by encouraging and even forcing the programmer to improve his style of exposition and his principles and discipline in program organization, and not by merely providing a multitude of "new" features and facilities. The language is therefore particularly well suited for tutorial purposes.

The attempt to present a computer as a systematically organized entity is also hoped to be of interest to designers of future computers.

A Programming Language for the 360 Computers

Page

Contents

I. Introduction, Aims and Purpose

In an era of feverish and prolific activity in the design of more and more sophisticated and intricate programming aids, the proposal of a machine language may seem anachronistic to some readers. This report describes an attempt to provide a tool for those applications where it is essential to conceive the program as closely as possible in terms of an existing computer in order to directly take into account its particular capabilities and limitations. Sophistication has not been an aim in this attempt, but emphasis was rather put on a clear and conceptually systematic exposition of the available facilities. The result is reliability on the part of the implemented system as well as on the part of the user who is not subject to misunderstandings about the nature of complicated and ill-defined--facilities. None of these objectives should be called anachronistic.

In the summer of 1965, the author decided to undertake efforts to implement the proposed successor to ALGOL described in [2] on the IBM 360 computer which at that time had been chosen as Stanford's next generation machine. It was felt that the evolving project should be conducted in a thorough and systematic manner, worthy of an academic endeavour, and making use of the best available methods on compiler construction known. The results should consist of a well-organized system whose structure and principles were sound and precisely understood, and which was intelligibly documented.

After many years of experience with ALGOL, it was clearly recognized that a compiler written in 360 Assembly Language would neither be able to meet the desired documentation standards, nor constitute a sufficiently convenient programming tool. The only other language available on the 360, FORTRAN, was not deemed adequate either. Against the strong arguments of the undesirability of the large amount of additional efforts required to produce a new language and its compiler, it was decided to develop a tool which would:

1. allow full use of the facilities provided by the 360 hardware,
2. provide convenience in writing and correcting programs, and
3. encourage the user to write in a clear and comprehensible style.

As a consequence of 3., it was felt that programs should not be able to modify themselves. The language should have the facilities necessary to express compiler and control programs, and the programmer should be able to determine every detailed machine' operation. In this respect, the language features the property of a conventional assembly code. In its appearance, however, it resembles a high level programming language due to the presence of structure. Being specifically tailored for the 360 computer, the language was appropriately named PL360.

Chapter II is the definition of the language. It is given in terms of a syntax, and the semantic explanations of the individual syntactic constructions. Knowledge about the nature of the 360 architecture is prerequisite (cf. [1]); however, the definition does not require familiarity with the 360 Assembly Language. A few self-explanatory examples of programs are listed in Chapter III.

The following two chapters are devoted to the implementation of PL360. They exhibit the code which the compiler generates corresponding to various language statements, and the method of segmentation and addressing. Chapter VI gives an account of the organization of the compiler, which relies on a rigorous syntax analysis of the text while at the same time generating the target code. The compiler constitutes a large scale practical example for the application of the techniques described in [3], which have been extended to process incorrectly constructed texts and to meaningfully diagnose errors. The success of this facility is considered to be a major contribution to make predecence grammars useful in practical applications.

The methods employed in producing the compiler are described in Chapter VII. A bootstrapping technique was used to make the compiler available on the 360 computer without prior use of any of the languages existing on that machine. Programming the compiler in its own language provided a thorough test for the adequacy of the language to its anticipated purpose.

Chapter VIII gives a brief account of the size and the performance of the translator on a 360/50 computer. Concluding remarks about the language and its implementation lead to a brief examination of the appropriateness of the 360 architecture for this experiment.

II. Definition of the Language

Page

Contents

1. Terminology, notation, and basic definitions

The language is defined in terms of a (/360) *computer* which comprises a number of *processing units* and a finite set of *storage elements*. Each of the storage elements holds a *content*, also called *value*. At any given time, certain significant *relationships* may hold between storage elements and values. These relationships may be recognized and altered, and new values may be created by the processing units. The actions taken by the processors are determined by a *program*. The set of possible programs form the *language*. A program is composed of, and can therefore be decomposed into elementary constructions according to the rules of a *syntax*, or grammar. To each *elementary construction* corresponds an *elementary action* specified as a semantic rule of the language. The action denoted by a program is defined as the sequence of elementary actions corresponding to the elementary constructions which are obtained when the program is decomposed (parsed) by reading from left to right.

1.1. The computer

According to their specific capabilities, processing units are divided into *central processing units* (CPU), input-output processing units (*channels*), and input-output *devices*. At any time, the status of a unit is described by a sequence of bits, called the *program status word* (PSW) for CPUs and the *channel status word* (CSW) for channels. A status word contains, among other information, a pointer to the currently executed instruction. In particular, the program status word also contains a quantity which is called *condition code*.

Storage elements are classified into *registers* and core memory cells, simply called *cells*. Registers are divided into three kinds according to their size and the operations which can be performed on their values. The kinds of registers are:

a. *integer* or *logical* (a sequence of 32 bits)
b. *real* (a sequence of 32 bits)
c. *long real* (a sequence of 64 bits)

Cells are classified into seven types according to their size and the type of value which they may contain. A cell may be structured or simple. The types of simple values and simple cells are:

a. byte, or character (a sequence of 8 bits),

b. short integer (a sequence of 16 bits, usually interpreted as an integer in two's complement binary notation),

c. integer or logical (a sequence of 32 bits, usually interpreted as an integer in two's complement binary notation),

d. real (a sequence of 32 bits to be interpreted as a floating point binary number),

e. long real (a sequence of 64 bits to be interpreted as a floating point binary number),

f. command (a sequence of 64 bits, usually interpreted as a data channel command).

1.2. Relationships

The most fundamental relationship is that which holds between a cell and its value. It is known as containment; the cell is said to contain the value.

Another relationship holds between the cells which are the components of a structured cell, called an array, and the structured cell itself. It is known as subordination. Structured cells are regarded as containing the Cartesian product of the values of the component cells. The component cells themselves are well-ordered.

A set of relationships between values is defined by monadic and dyadic functions or operations, which the processors are able to evaluate or perform. The relationships are defined by mappings between values (or pairs of values) known as the operands and values known as the results of the evaluation. These mappings are not to be precisely defined in this report; instead, references will be given to their definition in official publications on the /360 computer [4].

1.3. The program

A program contains declarations and statements. Declarations serve to list the quantities which are involved in the algorithm denoted by the program, and to associate a name, a so-called *identifier,* with each quantity. Statements specify the operations to be performed on these quantities, to which they refer through use of the identifiers.

A program is a sequence of tokens, which are basic symbols, strings, or comments. Every token is itself a sequence of characters. The following conventions are used in the notation of the present article:

a. basic symbols constitute the basic *vocabulary* of the language (cf. 1.6.). They are either single non-alphanumeric characters or underlined letter sequences;

b. strings are sequences of characters enclosed in quote marks (");

c. comments are sequences of characters (not containing a semicolon) preceded by the basic symbol *comment* and followed by a semicolon (;). It is understood that during execution of a program, all comments are ignored.

In order that a sequence of tokens be an executable program, it must be constructed according to the rules of the syntax.

1.4. Syntax

A sequence of tokens constitutes an instance of a syntactic entity (or construct), if that entity can be derived from the sequence by one or more applications of syntactic substitution rules. In each such application, the sequence equal to the right side of the rule is replaced by the symbol which is its left side.

Syntactic entities (cf. 1.5.) are denoted by English words enclosed in the brackets ⟨ and ⟩ . These words describe approximately the nature of the syntactic entity, and where these words are used elsewhere in the text, they refer to that syntactic entity. For reasons of notational convenience and brevity, the script letters $\mathcal{K}$ and $\mathcal{T}$ are also used in the denotation of syntactic entities. They are understood to stand for any *kind* of register or *type* of cell, possibly subject to restrictions mentioned in the accompanying text of the paragraph.

Syntactic rules are of the form

$$(A) ::= \xi$$

where $\langle A\rangle$ is a syntactic entity (called the left side) and ξ is a finite sequence of tokens and syntactic entities (called the right side of the rule). The notation

$$\langle A\rangle ::= \xi_1 | \xi_2 \mathrm{I} \ \ldots \ | \xi_n$$

is used as an abbreviation for the n syntactic rules

$$(A) ::= \xi_1,\ (A) ::= \xi_2,\ \ldots\ ,\ (A) ::= \xi_n \ .$$

If in the denotations of constituents of the rule the script letters $\mathcal{K}$ or $\mathcal{T}$ occur more than once, they must be replaced consistently. As an example, the syntactic rule

$$\langle\mathcal{K} \text{ register}) ::= \langle\mathcal{K} \text{ register identifier})$$

is an abbreviation for the set of rules

(integer register) ::= (integer register identifier)
(real register) ::= (real register identifier)
(long real register) ::= (long real register identifier)

1.5. Syntactic Entities

(arithmetic operator)	2.6.
(base declaration)	2.11.
(block body)	3.5.
(block head)	3.5.
(block)	3.5.
(case clause)	3.2.
(case sequence)	3.2.
(case statement)	3.2.
(condition)	3.1.
(decimal digit)	2.2.
(decimal integer number)	2.2.
(decimal scale factor)	2.2.
(declaration)	3.5.
(for clause heading)	3.4.
(for clause)	3.4.
(for statement)	3.4.
(fractional number)	2.2.
(function declaration⟩	2.8.
(function heading)	2.8.
(function identifier)	2.1.
(function name)	2.8.
(function statement)	2.9.
(got0 statement)	3.6.
(hexadecimal number)	2.2.
(if clause)	3.1.
(if statement)	3.1.
(increment)	3.4.
⟨𝔎 register assignment)	2.6.
⟨𝔎 register declaration head)	2.3.
⟨𝔎 register declaration)	2.3.
⟨𝔎 register identifier)	2.1.
⟨𝔎 register)	2.6.
(label definition)	3.5.
(limit)	3.4.
(logical operator)	2.6.
(monadic operator)	2.6.
(procedure declaration)	3.7.
(procedure head)	3 .7.
(procedure heading)	3 .7.
(procedure identifier)	2.1.
(procedure name)	3.7
(procedure statement)	3.8.
⟨**program**⟩	3.5.
(relational operator)	3.1.
(shift operator)	2.6.
(simple 𝔎 register assignment)	2.6.
(simple statement)	3.5.
(simple 𝔍 type)	2.4.
(statement)	3.5.
(string)	2.2

(continued)

⟨T cell assignment)	2.7.
⟨T cell declaration)	2.4.
⟨T cell designator)	2.5.
⟨T cell identifier)	2.1.
⟨T number)	2.2,
⟨T subcell designator)	2.5.
⟨T synonym declaration)	2.10.
⟨T type)	2.4.
⟨T value)	2.6.
(true part)	3.1.
(unsigned integer number)	2.2.
(unsigned long real number)	2.2.
(unsigned real number)	2.2.
(while clause)	3.3.
(while statement)	3.3.

1.6. 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|abs|not|shℓℓ|shrℓ|shℓa|shra|

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

begin|end|goto|comment|null|

integer|real|logical||array|haracter|long|short

command|function|procedure|register|syn

segment|base

2. Data manipulation facilities.

2.1. Identifiers

(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⟩

⟨𝔎 register identifier) ::= (identifier)

⟨𝔗 cell identifier) ::= (identifier)

(procedure identifier) ::= (identifier)

(function identifier) ::= (identifier)

An identifier is a 𝔎 register-, 𝔗 cell-, procedure-, or function identifier if it has respectively been associated with a 𝔎 register, 𝔗 cell, procedure, or function (called a quantity) in one of the blocks surrounding its occurrence. This association is achieved by an appropriate declaration. The identifier is said to designate the associated quantity. If the same identifier is associated to more than one quantity, then the considered occurrence designates the quantity to which it was associated in the smallest block embracing the considered occurrence. In any one block, an identifier must be associated to exactly one quantity. In the parse of a program, that association determines which of the rules given above applies.

Any processing computer can be considered to provide an environment in which the program is embedded, and in which some identifiers are permanently declared. Some identifiers are assumed to be known in every environment; they are called standard identifiers, and are listed in the respective paragraphs on declarations.

2.2. Numbers and strings

(decimal digit) ::= 0|1|2|3|4|5|6|7|8|9

(decimal integer number) ::= (decimal digit⟩|
(decimal integer number⟩⟨decimal digit)

(unsigned integer number) ::= (decimal integer number)

(fractional number) ::= (decimal integer number)⟩ . (decimal digit⟩|
(fractional number⟩⟨decimal digit)

(decimal scale factor) ::= (decimal integer number⟩|
(decimal integer number)

(unsigned real number) ::= (fractional number⟩|
(fractional number) E (decimal scale factor⟩|
(decimal integer number) E (decimal scale factor)

(unsigned long real number) ::=
(fractional number) D (decimal scale factor⟩|
(decimal integer number) D (decimal scale factor)

⟨𝒯 number) ::= (unsigned 𝒯 number⟩|_⟨unsigned 𝒯 number)

(hexadecimal digit) ::= (decimal digit⟩|A|B|C|D|E|F|

Here 𝒯 stands for any one of

integer
real
long real

(hexadecimal number) ::= #(hexadecimal digit⟩|
(hexadecimal number⟩⟨hexadecimal digit)

(integer number) ::= (hexadecimal number)

Numbers have their conventional meaning. They can either be given in decimal or hexadecimal notation. The scale factor signifies that the preceding number be multiplied by the indicated power of ten. The symbol _ stands for a minus sign.

A string is any sequence of characters enclosed by quote marks, within which a single quote mark (") is always denoted by a pair of adjacent quote marks (""). Examples:

"ABC" denotes the sequence ABC
"A""Z" denotes the sequence A"Z
" ""A"""" denotes the sequence "A"

Examples:

integer numbers

0 1066 _5 #A #F00

real numbers

1.0 0.1 -3.1416 2.7E5 1E10

long real numbers

5.37861289D0   _1D10   8.9D_5

strings

"A STRING IS A CHARACTER-SEQUENCE"

"DATE: 29/9/ 1966"

2.3. Register declarations

In the following rules, the letter 𝔎 must be replaced by any one of the following words (or word pairs):

integer
real
long real

⟨𝔎 register declaration head) ::=
    (simple 𝔎 type) REGISTER ⟨identifier⟩|
    ⟨𝔎 register declaration⟩,⟨identifier⟩
⟨𝔎 register declaration) ::=
    ⟨𝔎 register declaration head⟩(⟨integer number))

Every identifier in a 𝔎 register declaration is associated with the 𝔎 register specified by the integer number enclosed in parentheses following the identifier. It herewith becomes a 𝔎 register identifier. This number must designate one of the existing integer (or logical) registers numbered 0-15, or one of the existing real or long real registers numbered 0, 2, 4, and 6.

Examples:

integer register count(1), m(2), n(3)
long real register sum(4), product(0)

The following are standard register identifiers:

R0, R1,..., R9, RA,..., RF

designating the 16 integer registers, and

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

designating the 4 real and long real registers respectively. .

2.4. Cell declarations

(simple integer type) ::= integer|logical
(simple short integer type) ::= short integer
(simple real type) ::= real
(simple long real type) ::= long real
(simple byte type) ::= byte|character
(simple command type) ::= command
⟨$\mathcal{T}$ type) ::= (simple $\mathcal{T}$ type⟩|
    array ((integer'number)) ⟨simple $\mathcal{T}$ type)
⟨$\mathcal{T}$ cell declaration) ::= ⟨$\mathcal{T}$ type⟩⟨identifier⟩|
    ⟨$\mathcal{T}$ cell declaration) , (identifier)\
    ⟨$\mathcal{T}_0$ cell declaration) (⟨$\mathcal{T}_1$ number))|
    (character cell declaration) ((string))

Every identifier occurring in a cell declaration is associated to one unique cell of the indicated type, if that type is simple, or otherwise to a unique array of cells of the indicated type. The number of cells in an array is given by the number enclosed in parentheses following the symbol array .

If a cell declaration is followed by one or more numbers or strings within parentheses, then the cell is declared to contain those numbers or strings as its values. $\mathcal{T}_0$ and $\mathcal{T}_1$ must either be identical, or be selected from the following combinations:

$\mathcal{T}_0$	$\mathcal{T}_1$
short integer	integer
byte	integer
command	integer

The number of such values must not exceed the number of declared elements in the array. A string can only be assigned to a character cell, and the number of characters must not exceed the number of indicated array elements. This assignment of values must be understood to take place only upon the first time the block, in which the cell declaration occurs, is entered.

Examples:

```
    byte a g
    short integer i, j
    integer, age(21), hight(68)     .
    long real x, y, z
    array (3) integer size(36)(23)(37)
    array (1000) real quant, price
    array (8) byte flags
    array (132) character line(" ")
```

Note : The symbols integer and logical, and byte and character are treated as synonymous in the language.

2.5. Cell designators

⟨𝒯 subcell designator) ::= ⟨𝒯 cell identifier⟩(⟨integer number))
⟨𝒯 cell designator) ::= ⟨𝒯 cell identifier⟩|⟨𝒯 subcell designator⟩|
    ⟨𝒯 cell identifier) ((integer register)) |
    ⟨𝒯 subcell designator) ((integer register))

A cell identifier which is followed by a number or an integer register enclosed in parentheses (called a subscript), must designate an array of cells. When n is the subscript (number or current value of register), then the construct designates that cell of the array which is located n memory unit positions (1) from the beginning of the array, if the subscript is preceded by the cell identifier, or (2) from the designated position, if the subscript is preceded by a subcell designator. The number of memory units occupied by cells of various types are: character (1), byte (1), shortinteger (2), integer (4), logical (4), real (4), long real (8). The subscript used to designate any element of an array must therefore be a multiple of the appropriate number.

Note: A subscript must not specify register 0 .

Examples of cell designators:

```
    age
    size(2)
    prize(R1)
    line(16)(R2)
```

2.6. Register assignments

⟨𝔎 register) ::= ⟨𝔎 register identifier)

⟨𝔗 value) ::= ⟨𝔗 number⟩|⟨𝔗 cell designator)

(integer value) ::= ⟨string⟩

A 𝔎 register designates the value contained in the identified register. A value is either a constant, i.e., a number or a string, or the content of a designated cell. In the case of a logical value being a string, that string must consist of not more than 4 characters. If it consists of fewer than 4 characters, the string is extended to the left with null characters. The bit representation of characters is defined in [1] (EBCDIC).

(simple 𝔎 register assignment) ::=

(1) ⟨𝔎 register identifier) := ⟨𝔗 value⟩|

⟨𝔎 register identifier) := ⟨𝔎 register⟩|

(2) ⟨𝔎 register identifier) := (monadic operator⟩⟨𝔗 value)

⟨𝔎 register identifier) := (monadic operator⟩⟨𝔎 register)

A simple register assignment is said to specify a register, namely the one designated by the register identifier to the left of the assignment operator (:=) . To this register is assigned the -value designated by the construct to the right of the assignment symbol. That designated value may be obtained through execution of a monadic operation specified by a monadic operator.

The following are legal combinations of kinds and types to be substituted respectively for the letters 𝔎 and 𝔗 in the rules (1) and (2):

𝔎	𝔗
integer	integer
integer	short integer
integer	command
real	real
long real	real
long real	long real

(monadic operator) ::= abs | neg | neg abs

The monadic operations are those of taking the absolute value, of sign inversion, and of sign inversion after taking the absolute value.

Examples of simple register assignments:

```
R0  := i
R2  := RA
R6  := age
F0  := quant(R1)
F23 := x
F45 := neg F01
RD  := abs hight
F0  := neg abs F6
```

⟨𝒦 register assignment) ::= (simple 𝒦 register assignment⟩|
(3) ⟨𝒦 register assignment⟩⟨arithmetic operator⟩⟨𝒯 value⟩|
⟨𝒦 register assignment⟩⟨arithmetic operator⟩⟨𝒦 register⟩|
(integer register assignment⟩⟨logical operator⟩⟨integer value⟩|
(integer register assignment⟩⟨logical operator⟩⟨integer register⟩|
(integer register assignment⟩⟨shift operator⟩⟨unsigned integer number⟩|
(integer register assignment⟩⟨shift operator⟩⟨integer register identifier)

A register assignment specifies a register, namely the one which is specified by the simple register assignment or the register assignment from which it is derived. To this register is assigned the value obtained by applying a dyadic operator to the current value of that specified register and the value designated by the construct following the operator. The operations are the arithmetic operations of addition (+), subtraction (-), multiplication (*), and division (/), the logical operations of conjunction (and), exclusive and inclusive disjunction (xor, or), and those of shifting to the left and right, as implemented in the /360 system. The operators ++ and -- denote "logical" or unnormalized addition and subtraction when applied to integer or real registers respectively.

(arithmetic operator) ::= +|-|*|/|++|--
(logical operator) ::= and|or|xor
(shift operator) ::= shlℓ|shℓa|shrℓ|shra

In the syntactic rule (3), the same combinations of $\mathfrak{K}$ and $\mathfrak{T}$ are permitted as specified for rules (1) and (2).

Examples of register assignments:

```
R0  := R3
R1  := 10
RA  := i + age - R3 + size(R1)
R9  := R8 and R7 shll 8 or R6
F2  := 3.1416
F0  := quant(R1) * price(R1)
F45 := F45 + F01
```

Note : The syntax implies that sequences of operators (including assignment) are executed strictly in sequence from left to right. Thus

```
R1  := R2 + R1
```

is not equivalent to

```
R1  := R1 + R2
```

but rather to the two statements

```
R1  := R2; R1  := R1 + R1
```

2.7. Cell Assignments

⟨$\mathfrak{T}$ cell assignment⟩ ::=
    ⟨$\mathfrak{T}$ cell designator⟩ := ⟨$\mathfrak{K}$ register⟩

The value of the designated $\mathfrak{K}$ register is assigned to the designated $\mathfrak{T}$ cell. The allowable combinations of cell-type and register kind are indicated in the table of section 2.6.

Examples of cell assignments:

```
i := R0
price(R1) := F0
x := F67
```

2 . 8unction declarations

(function name) ::= <u>function</u> ⟨identifier⟩|
    (function declaration) , (identifier)

(function heading) ::= (function name⟩(⟨integer number))

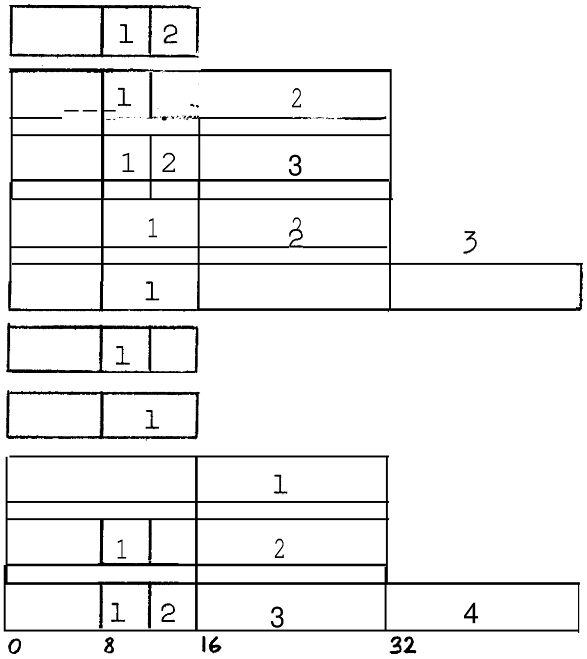
(function declaration) ::= (function heading⟩(⟨integer number))

There exist various data manipulation facilities in the 360 computer which cannot be expressed by an assignment. To make these facilities amenable to the language, the function statement is introduced (cf. 2.9.), which uses an identifier to designate an individual computer instruction. The function declaration serves to associate this identifier, which thereby becomes a function identifier, with the desired computer instruction code, and to define the meanings of the parameters of the function, i.e., to specify the format of the instruction. While the number in the function heading specifies the format (cf. table below) and is called the format code, the number in the function declaration specifies the first two bytes of the instruction code. In the following examples, the identifiers were chosen to be the symbolic codes used in [4], and they are standard identifiers.

Examples

```
    function   MVI(4 )(#9200),        CLI(4)(#9500),
               MVC(5)(#D200),         CLC(5)(#D500),
               STM(3)(#9000),         LM(3)(#9800),
               SRDL(9)(#8C00),        SLDL(9)(#8D00),
               IC(2)(#4300),          STC(2)(#4200),
               LA(2)(#4100),          TEST (8 )(#95FF ),
               SET(8)(#92FF),         RESET(8)(#9200),
               CVD(2)(#4E00),         UNPK(10)(#F300),
               ED(5)(#DE00),          EX(2)(#4400)
```

Format Code	No. of parameter fields in function	Assignment of fields in instruction
1	2	
2	2	
3	3	
4	2	
5	3	
6	1	
7	1	
8	1	
9	2	
10	4	



2.9. Function statements

(function statement) ::= (function identifier))

(function statement) ((integer number⟩)|

(function statement⟩(⟨K register identifier⟩)|

(function statement⟩(⟨J cell designator⟩)|

(function statement⟩(⟨string⟩)

A function statement represents the computer instruction designated by the function identifier. The sequence of quantities enclosed in parentheses specifies the parameter'fields of the function statementin accordance with its format, to which the fields must comply.

Examples

```
SET(flag)
RESET(flag)
LA(R1)(line)
MVC(15)(line)(buffer)
STM(RO)(RF)(save)
MVI("*")(line)
IC(RO)(flags(R1))
```

2.10. Synonym declarations

⟨$\mathcal{T}_0$ synonym declaration) ::=
⟨$\mathcal{T}_0$ type⟩⟨identifier⟩ syn ⟨$\mathcal{T}_1$ cell designator⟩|
⟨$\mathcal{T}_0$ type⟩⟨identifier⟩ syn (integer number⟩|
⟨$\mathcal{T}_0$ synonym declaration) , (identifier) syn ⟨$\mathcal{T}_1$ cell designator))
⟨$\mathcal{T}_0$ synonym declaration) , (identifier) syn (integer number)

A synonym declaration serves to associate identifiers with the cell which is designated immediately following the symbol syn, either by a previously established cell designator or by an integer number representing its absolute address in the computer's core memory.

Examples:

```
integer xlow syn x(4)
array (32768) integer mem syn 0
logical CAW syn 72
integer B1 syn mem(R1)
```

Note: The synonym declaration can be used to associate several different types with a single cell. Each type is connected with a distinct identifier.

Example:

```
long real x(#4E00000000000000)
integer xlow syn x(4)
```

A conversion operation from a number of type integer contained in register RO to a number of type long real contained in register F01 can now be denoted by

```
xlow := RO; F01 := x
```

and a conversion vice-versa by

```
F01 := F01 ++ zero; x := F01; RO := xlow
```

No initialization can be achieved by a synonym declaration.

2.11. Segment base declarations

(segment base declaration) ::= segment base (integer register identifier)

A base declaration causes the compiler to reference the specified register as a base address for all cells subsequently declared in the block in which the base declarationoccurs. Upon entrance to this block, the appropriate base address is assigned to the specified register. (cf. v.2).

3. Control facilities

3.1. If statements

(relational operator) ::= = | ¬= | < | <= | >= | >

(condition) ::= ⟨𝒦 register⟩⟨relational operator⟩⟨𝒥 value⟩|
⟨𝒦 register⟩⟨relational operator)@ register⟩|
(relational operator⟩|overflow

The 360 computer records one of four possible states in the so-called condition code. A condition specifies one or more of these states, which are numbered 0, 1, 2, 3. The relational operators and the symbol overflow designate the following states:

operator	states
=	0
¬=	1,2
<	1
< =	0,1
> =	0,2
>	2
overflow	3

If a relational operator is enclosed by two operands, then those operands are compared, and the condition code is set to state 0, if equality holds, to state 1 if the first operand is numerically smaller, and to state 2 if it is greater than the second operand.

```
(if clause) ::= if (condition) then
(true part) ::= <simple statement) else
(if statement) ::= (if clause⟩⟨statement⟩|
     (if clause⟩⟨true part⟩⟨statement⟩
```

The if statement permits the conditional execution of statements:

1. (if clause⟩⟨statement⟩

   The statement is executed if and only if the condition code is in one of the states designated by the condition in the if clause.

2. (if clause⟩⟨true part⟩⟨statement⟩

   The simple statement in the true part is executed and the statement following it is ignored, if and only if the condition code is in one of the states designated by the condition in the if clause; otherwise the true part is ignored and the statement following it is executed.

Examples

```
if RO < 10 then Rl := 1
if FO>= F2 then F2 := FO else FO := F2
if < then SET(flags(0)) else
     if = then SET(flags(1)) else SET(flags(2))
```

Note: if the condition consists of a relational operator without operands, then the decision is made on the basis of the condition code as determined by a previous instruction.

Example:

```
CLC(15)(a)(b); if = then ...
```

3.2. Case statements

```
(case clause) ::= case (integer register) of
(case sequence) ::= (case clause) begin |
    ⟨case sequence⟩⟨statement⟩;
(case statement) ::= (case sequence) end
```

Case statements permit the selection of one of a sequence of statements according to the current value of the integer register (other than register 0) specified in the case clause. The statement whose ordinal number is equal to the register value is selected for execution, and the other statements in the sequence are ignored. The value of that register is thereby multiplied by 4.

Example:

```
case R1 of begin
     R1 := R2;
     R2 := R3;
     R3 := R4;
     R4 := R5;
end
```

3.3 . While statements

```
(while clause) ::= while (condition) do
(while statement) ::= (while clause)⟨statement⟩
```

The while statement specifies the repeated execution of a statement as long as the condition code is in one of the states specified by the condition in the while clause.

Examples:

```
while FO < prize(R1) do R1 := R1 + 4

while >= do
begin RO := RO + 1; R1 := R1 - R2;
end
```

Note that in the second example the condition code is set by the subtraction operation and then tested for being in states 0 or 2 .

3.4. For statements

(for clause heading) ::= for (integer register assignment)

(increment) ::= step ⟨integer number)

(limit) ::= until (integer register⟩| until ⟨𝒯 value)

(for clause) ::= (for heading⟩⟨increment⟩⟨limit⟩ do

(for statement) ::= (for clause⟩⟨statement⟩

𝒯 must be replaced by either of the types

integer
short integer

The for statement specifies the repeated execution of a statement, while the content of the integer register specified by the for heading takes on the values of an arithmetic progression. That register is called the control register. The execution of a for statement occurs in the following steps:

1. the register assignment in the for heading is executed;
2. if the number specifying the increment is not negative/negative, then if the value of the control register is not greater/not less than the value specified as the limit, then the process continues with step 3, otherwise the execution of the for statement is terminated;
3. the statement following the for clause is executed;
4. the increment is added to the control register, and the process resumes with step 2.

Examples:

```
for R1 := 0 step 1 until n do STC(".")(line(R1))
for R2 := R1 step _4 until R0 do
    F23in     := quant(R2) * price(R2);
         F01 := F01 + F23;
    end
```

3.5. Blocks

(declaration) ::= ⟨𝒦 register declaration⟩|⟨𝒯 cell declaration⟩|
    (function declaration⟩|⟨procedure declaration⟩|
    (synonym declaration⟩|⟨segment base declaration)

(simple statement) ::= ⟨𝒦 register assignment⟩|⟨𝒯 cell assignment⟩|
    ⟨function⟩|⟨procedure statement⟩|⟨case statement⟩|⟨block⟩|
    ⟨goto statement⟩| null

(statement) ::= (simple statement⟩| (if statement⟩|
    (while statement⟩| (for statement)

(label definition) ::= (identifier) :

(block head) ::= begin |
    (block head⟩⟨declaration⟩;

(block body) ::= (block head⟩| (block body⟩⟨statement⟩;|
    ⟨block body⟩⟨label definition)

(block) ::= (block body) end

(program) ::= (block) @

A block has the form

begin D; D; . . . ; D; S; S; . . . ; S; end

where the D's stand for declarations and the S's for statements. The two main purposes of a block are:

1. To embrace a sequence of statements into a structural unit which as a whole is classified as a simple statement. The constituent statements are executed in sequence from left to right.

2. To introduce new quantities and associate identifiers with them. These identifiers may be used to refer to these quantities in any of the declarations and statements within the block, but are not known outside the block.

Label definitions serve to label certain points in a block. The identifier of the label definition is said to designate the point in the block where the label definition occurs. Go to statements may refer to such points. The identifier can be chosen freely, with the restriction that no two points in the-same block must be designated by the same identifier.

The symbol null denotes a simple statement which implies no action at all.

Example:

```
        begin integer bucket;
              TEST(flag); if = then
              begin bucket  := RO;  RO := Rl; Rl := R2;
                    R2  := bucket;
              end else
              begin bucket  := R2; R2 := Rl; Rl := RO;
                    RO := bucket;
              end;
              RESET(flag);
        end
```

3.6. Go to statements

⟨go to statement) ::= goto (identifier)

The interpretation of a goto statement proceeds in the following steps:

1. Consider the smallest block containing the goto statement.
2. If the identifier designates a program point within the considered block, then program execution resumes at that point. Otherwise, execution of the block is regarded as terminated and the smallest block surrounding it is considered, Step 2 is then repeated.

3.7. Procedure declarations

(procedure name) ::= procedure ⟨identifier⟩|
segment procedure (identifier)
(procedure heading) ::= (procedure name⟩(⟨integer register identifier))
(procedure head) ::= (procedure heading);
(procedure declaration) ::= (procedure head⟩⟨statement⟩

A procedure declaration serves to associate an identifier, which thereby becomes a procedure identifier, with a statement (cf. 3.5.) which is called procedure body. This identifier can then be used as an abbreviation for the procedure body anywhere within the scope of the declaration. The integer register specified in the procedure heading is assigned the return address of control when the statement is invoked by such an abbreviation (procedure statement). It must not be register 0.

If the symbol procedure is preceded by the symbol segment, the procedure body is compiled as a separate program segment (cf. chapter V.1). It has no influence on the meaning of the program.

Examples

```
procedure  P(R1); RO := RO + 1

procedure  SWAP(RA);
   begin long real t;
      t := FO1; FO1 := F23; FO1 := t;
   end
```

Note: The code corresponding to a procedure body is followed by a branch instruction taking the program address from the register specified in the procedure heading, where the invoking procedure statement had deposited the return address. Thus, the programmer must either not use that register within the procedure, or explicitly store and reload its value in the beginning and end of the procedure body.

3.8. Procedure statements

(procedure statement) ::= (procedure identifier)

The procedure statement invokes the execution of the procedure body designated by the procedure identifier. A return control address is assigned to the register specified in the heading of the designated procedure declaration.

III. Examples

```
procedure  magicsquare(R6);
comment This procedure establishes a magic square of order n,  if n
        is odd, and 1 < n <16.  X is the matrix in linearized form.
        Registers 0...6 are used, and register 0 initially contains
        the parameter n .  Algorithm 118(Comm. ACM, Aug. 1962);
begin short integer nsqr;
    integer, register n(0), i(1), j(2), k(5);
    nsqr := n; R1 := n*nsqr; nsqr := R1;
    i := n + 1 shrℓ 1; j := n;
    for k := 1 step 1 until nsqr do
    Begin  3 := i shℓℓ 6; R4 := j shℓℓ 2 + R3; R3 := X(R4);
        if R3 ¬= 0 then
        begin i := i - 1; j := j - 2;
            if  i < 1 then i := k + n;
            if j < 1 then j := j + n;
            R3 := i  shℓℓ 6; R4 := j shℓℓ 2 + R3;
        end;
        X(R4) := k;
        i := k + 1; if i > n then i := i - n;
        j := j + 1; if j > n then j :=j - n;
    end;
end
```

```
procedure  inreal(R4);
comment This procedure reads characters forming a real number according
        to the PL360 syntax. A procedure "nextchar" is used to obtain,
        the next character in sequence in register 0 . The result is
        left in the long real register F01 . Registers 0 ... 4 and
        all real registers are used;
```

```
byte sign, exposign;  short integer ten (10);
long real fcon1(#4E00000000000000), fcon2(#4700000000000000);
integer  fcon1low syn fcon1(4);
function  SRDL(9)(#8C00), LTR(1)(#1200);
while R0 < "0" do
begin  f R0 = "-" then SET(sign) else RESET(sign); nextchar;
end;
comment  Accumulate the integral part in R1;
R1 := R0 and #F; nextchar;
while R0 >= "0" do
Begin  O := R0 and #F; R1  := R1 * ten + R0; nextchar;
end;
R2 := 0; comment R2 is the decimal scale factor;
fcon1low  := R1; F01 := fcon1 + 0D0; comment F01 := R1;
if R0 = "." then
begim m e n t  Process fraction.  Accumulate number in F01;
      nextchar;
      while R0 >= "0" do
      begin R0  := R0 shll 4; STC(R0)(fcon2(4));
            F01  := F01 * 10D0 + fcon2; R2 := R2 - 1; nextchar;
      end;
end;
if R0 = "E" then
begin comment  Add the scale factor to R2;
      nextchar; if R0 = "-" then
           begin SET(exposign); nextchar;
           end else  RESET(exposign);
      R1  := R0 and #F; nextchar;
      while R0 >= "0" do
      begin R0  := R0 and #F; R1  := R1 * ten + R0; nextchar;
      end;
      TEST(exposign);
      if = then R2  := R2 - R1 else R2  := R2 + R1;
end;
if R2 ¬= 0 then
```

```
          begin comment Compute F45 := 10 ↑ R2;
               if  R2 < 0 then
               begin R2  := abs R2; SET(exposign);
               end else RESET(exposign);
               F23  := 10D0; F45  := 1D0; F67  := F45;
               while R2 ¬= 0 do
               begin SRDL(R2)(1); F23 := F23 * F67; F67 := F23;
                    LTR(R3)(R3); if < then F45 := F45 * F23;
               end;
               TEST(exposign);
               if = then F01  := F01/F45 else F01 := F01 * F45;
          end;
          TEST(sign); if = then F01 := neg F01;
     end

     procedure  Binary Search (R8);
     comment A binary search is performed for an identifier in a table via an
               alphabetically ordered directory containing for each entry the
               length (no. of characters) of the identifier, the address of the
               actual identifier, and a code number.  The global declarations

                    array ( ) integer directory
                    array ( ) short integer tag syn directory (0)
                    array ( ) short integer length syn directory (2)
                    array ( ) integer address. syn directory (4)
                    integer n

               are assumed.  Upon entry,  R1 contains the length of the given
               identifier,  R2 contains its address. Upon exit,  R3 contains
               the code number, if a match is found in the table,  0 otherwise.
               Registers 1-8 are used;
     begin integer register  ℓ(1), ℓow(3),  i(4), high(5), x(6), m(7);
          array (3) short integer  compare (#D500)(#2000)(#6000);
          high := n; ℓow := 8; comment  index step in directory is 8;
     while ℓow <= high do
     begin i := Row + high shrℓ 4 shℓℓ 3; x := address(i);
          if ℓ = length(i) then
          EX(ℓ)(compare);          if = then goto found;
               if < then high := i-8 else tow :=i+8;
          end else
          if ℓ < length(i) then
          EX(ℓ)(compare);
               if <= then high := i-8 else low :=i+8;
          end else
          begin   := length(i); EX(m)(compare);
               if < then high := i-8 else Low := i+8;
          end;
     end;
     i := 0;
found: R3 := tag(i);
end
```

Assembly Language Code corresponding to the procedure'Magic square' (first example):

```
MAGICSQR STH    0,NSQR
         LR     1,0
         MH     1,NSQR
         STH    1,NSQR
         LR     1,0
         A      1,ØNE
         SRL    1,1
         LR     2,0
         L      5,ØNE
         B      L7
L1       LR     3,ØNE
         SLL    3,6
         LR     4,2
         SLL    4,2
         AR     4,3
         L      3,X(4)
         C      3,ZERØ
         BC     8,L4
         S      1,ØNE
         S      2,TWØ
         C      1,ØNE
         BC     11,L2
         AR     1,0
L2       C      2,ØNE
         BC     11,L3
         AR     2,0
L3       LR     3,1
         SLL . . 396
         LR     4,2
         SLL    4,2
         AR     4,3
L4       ST     5,X(4)
         A      1,ØNE
         CR     1,0
         BC     13,L5
         SR     1,0
L5       A      2,ØNE
         CR     2,0
         BC     13,L6
         SR     2,0
L6       LA     5,1(5)
L7       CH     5,NSQR
         BC     12,L1
         BR     6

NSQR     DC     H
ZERØ     DC     F"0"
ØNE      DC     F"1"
TWØ      DC     F"2"
```

IV. The object code

Two principal postulates were used as guidelines in the design of the language.

1. Statements which express operations on data must in an obvious way correspond to machine instructions. Their structure must be such that they decompose into structural elements, each corresponding directly to a single instruction.

2. The control of sequencing should be expressible implicitly by the structure of certain statements. (e.g., through prefixing them with clauses indicating their conditional or iterative execution).

Register assignments, cell assignments, and function statements strictly comply to postulate 1, as illustrated by the following example (cf. also II.2.4, II.2.6.):

```
                    RA := I + AGE - R3 + SIZE(R1)

Code:                  LH  10,I
                       A   10,AGE
                       SR  10,3
                       A   10,SIZE(1)
```

The following sections serve to exhibit the target code corresponding to constructions classified as "control facilities" in the definition of the language. The code is described in terms of 360 symbolic assembly language.

1. Construct: if (condition) then (statement)

```
Code:                  code for condition
                       BC c,L
                       code for statement
                    L  . . .
```

c is determined by the form of the condition, whose corresponding code may be empty or consisting of a C or CR instruction.

Example:

```
                 if R1 < R2 then R0 := R3

                    CR 1,2
                    BC     10,L
                    LR 0,3
                  L  ...
```

2. Construct: if (condition) then (simple statement) else (statement)

Code :

```
                    code for condition
                    BC c,L1
                    code for simple statement
                    B  L2
                 L1 code for statement
                 L2 . . .
```

3. Construct:

```
                      case Rm of begin
                           (statement-1);
                           (statement-2);
                              ........
                           (statement-n);
                      end
```

Code:

```
                    SLL m,2
                    B   L(m)
                 L1 code for statement-1
                    B   LX
                 L2 code for statement-2
                    B   LX
                    .....
                 Ln code for statement-n
                 L  B   LX
                    B   L1
                    B   L2
                    ...
                    B   Ln
                 LX . . .
```

4. Construct: while (condition) do (statement)

```
Code:            L1 code for condition
                      BC c,L2
                      code for statement
                      B    L1
                 L2 . . .
```

c is determined by the form of the condition, whose code may either be empty or a C or CR instruction. Note that the condition is established before the statement is ever executed.

5. Construct: for (integer register assignment)
step (integer number) until (integer value) do
(statement)

The corresponding code depends on the sign of the number following the symbol step. That number will be denoted by i below, and the assumption is made that the assignment after the symbol for specifies register m .

```
Code:            (i ≥ 0)
                      code for assignment
                 L1 C   m,V
                      BC 2,L2
                      code for statement
                      LA m,i(m)
                      B  L1
                 L2 . . .
```

```
Code:            (i < 0)
                      code for assignment
                 L1  C  m,V
                      BC 4,L2
                      code for statement
                      S  m,I
                      B  L1
                 L2 . . .
```

Note : The instruction labeled L1 is a CR instruction, if a register is specified as limiting value; V denotes the cell containing the limit value, I denotes the cell containing the decrement i .

The BXH and BXLE instructions were not used in the construction. The intricate rules about register assignment for control-, increment-, and limit values were considered to be too restrictive, and furthermore these instructions do not permit the testing of the initial value with the limit without altering the initial control value. They are entirely inappropriate for the case $i < 0$.

6. Construct: <u>procedure</u> ⟨identifier⟩(Rn); ⟨statement⟩

```
Code:              P    code for statement
                        BR  n
```

7. Construct: ⟨procedure identifier⟩

```
Code:                   BAL n,P         or

                        L   15, newsegmentbase
                        BAL n,P
                        L   15, oldsegmentbase
```

where n and P are specified by the procedure declaration.

V. Addressing and segmentation

The addressing mechanism of the 360 computers is such that instructions can indicate addresses only relative to a base address contained in a register. The programmer must insure that

1. every address in his program specifies a "base"-register;
2. the specified register contains the appropriate base address whenever an instruction is executed which contains an address;
3. the difference d between the desired absolute address and the available base address satisfies

$$0 \leq d < 4096 \quad .$$

This places a heavy burden upon the programmer, and it was considered to be unquestionably the duty of a compiler to ease the difficult task, and to provide certain checking facilities against errors.

The solution adopted here was that of program segmentation. The program is subdivided into individual parts, so-called segments. Every quantity defined within the program is known by the number of the segment in which it occurs and by its address relative to the origin of that segment, which serves as its base address. The problem then consists of subdividing the program and choosing base registers in such a way that a. the compiler can reference the appropriate register automatically when it compiles addresses, b. the compiler can assure that each base register contains the desired base address during execution, and c. the number of times base addresses are reloaded into registers is reasonably small.

First, it must be decided whether the process of subdividing the program should be performed by the programmer or by the compiler. In the latter case, a fixed number of registers must be set aside to serve as base registers which the compiler has freely at its disposal. This was considered undesirable. Furthermore, a program using a number of segments much larger than that of available base registers would be subject to considerable inefficiencies due to the necessity of loading base addresses very frequently. It was therefore decided that the programmer should designate the parts of his program which were to constitute segments. He has then the possibility of organizing the program in a way which minimizes the number of crossreferences between segments.

It should be noted that the programmer's knowledge about segment sizes and occurrences of crossreferences is quite different in the cases of program and data. In the latter case he is exactly aware of the amount of storage needed for the declared quantities, and he knows precisely in what places of the program references to a specific data segment occur. In the former case, his knowledge about the eventual size of a compiled program section is only vague, and he is in general unaware of the occurrence of branch instructions implicit in certain constructs of the language. It was therefore decided to treat programs and data differently, and this decision was also in conformity with the desirability of keeping program and data apart as separate entities.

1. Program segmentation

Due to the fact that the language does not allow programs to modify themselves, branches are the only instructions referring to locations within program segments. Since control lies by its very nature in exactly one segment at any instant, it seemed appropriate to designate one fixed register to hold the base address of the program segment currently under execution. A branch leading into another segment must then always be preceded by an instruction loading that register with the base address 'of the destination segment. Register 15 was chosen for this purpose.

An obvious approach to the problem of segmentation requires the compiler to automatically generate a new segment, when the currently generated segment's length exceeds 4096 bytes. This solution was rejected because of two reasons:

1. The programmer is not aware of the position of segment boundaries, and therefore has no way to minimize branches from one to another segment.
2. In most cases, the destination of an implicit branch (in if-, case-, while-, for statements) is not known to the compiler at the time of its generation. Therefore it is not known whether it will consist of one or two machine instructions.

The approach taken consists in connecting segment structure with the obvious program structure. The natural unit for a program segment is the procedure. The only way to enter a procedure is via a procedure statement, and the only way to leave-it is at its end or by an explicit go to statement. The fact that no implicitly generated instruction can ever lead control outside of a procedure minimizes the number of cross-references in a natural way. Since only relatively large procedure bodies should constitute segments, a facility was provided to designate such procedures explicitly: a procedure to be compiled as a program segment must contain the symbol segment in its heading. In practice, the requirement that such procedures be explicitly designated has proven to be no handicap. It is relatively easy for a programmer to guess which procedure exceeds the prescribed size, or otherwise to insert the symbol segment after the compiler has provided an appropriate comment in the first compilation attempt, Obviously, the outermost block is always compiled as a segment.

2. Data segmentation

In the case of data, the programmer is precisely aware of the amount of allocated memory as well as of the instances where reference is made to these quantities. A base declaration was therefore introduced which implies that all quantities declared thereafter, but still within the same block and preceding another base declaration, refer to the specified register as their base. These quantities form a data segment. At the place of the base declaration code is compiled which ensures that the register is loaded with the appropriate segment address. However its previous contents are neither saved nor restored upon exit from the block.

A base declaration is implicit in the heading of the outermost block. It always designates register 14.

Obviously, data segments declared in parallel (i.e., not nested) blocks, can safely refer to the same base register. Data segments declared within nested blocks should refer to different base registers.

If they do not, it is the programmer's responsiblity to ensure that the register is appropriately loaded when data in either of the segments is accessed.

There is no limit to the size of data segments. All cell designators must, however, refer to cells whose addresses differ from the segment base address by less than 4096. If they don't, the compiler can provide an appropriate indication.

3. Program loading

A scheme using program and data segments as described above results in an extremely simple relocating loader program, since the segments can be loaded without modification. It was felt that this benefit provided by a computer incorporating a base register scheme should be put to full advantage. Although the 360 computer still makes use of absolute addresses in a few instances (program status words, data channel commands), it was decided, not to allow for absolute addresses in a program. They can, however, be generated at execution time. Consequently, the functions of the loader are reduced to:

a. reading program and data segments into memory,

b. assigning the origin address of each segment to an entry in the segment address table, and

c. transfering control to the program segment representing the outermost block.

The base address table must be available from any point in the program. It was therefore placed in the low end of the first data segment, whose origin address is contained in register 14.

4. Problems connected with Input-output programming

The direct programming of input-output operations in PL360 is impractical in the scheme described so far for the following reasons:

1. Input-output operations on the 360 are designed to use the interrupt mechanism to signal termination of processes performed by data channels and devices in parallel with CPU operations,

In order to use the interrupt feature, it is necessary to create program status words (PSW) and store them in certain fixed locations of memory. A PSW contains the absolute address of a point in the program, which is a quantity that cannot be generated by a PL360 program.

2. Particularly in routines servicing interrupts, but also in some other cases, it is desirable to be able to dispense of a program base register. This could be done by locating these routines within the first 4096 bytes of core memory. The loader described above, however, chooses the absolute location of a segment on its own.

These two shortcomings can be overcome in many ways. The following is suggested:

1. A facility is introduced to designate a segment as an interrupt service routine, with the effect that the compiler supplies information to the loader, causing the loader to assign the segment's base address to the appropriate PSW cell instead of the segment address table. The compiler itself terminates this segment with an LPSW instead of a BR instruction (cf. V.6.). This approach forces a programmer to make explicit the fact that an interrupt routine is conceptually a closed segment, and it circumvents the undesirable introduction of a facility to generate labels as manipulatable objects.

2. A provision is introduced to cause the compiler not to refer to a base register in the branch instructions contained in the interrupt service segment. The loader is at the same time instructed to allocate this segment within the first 4096 bytes of core memory.

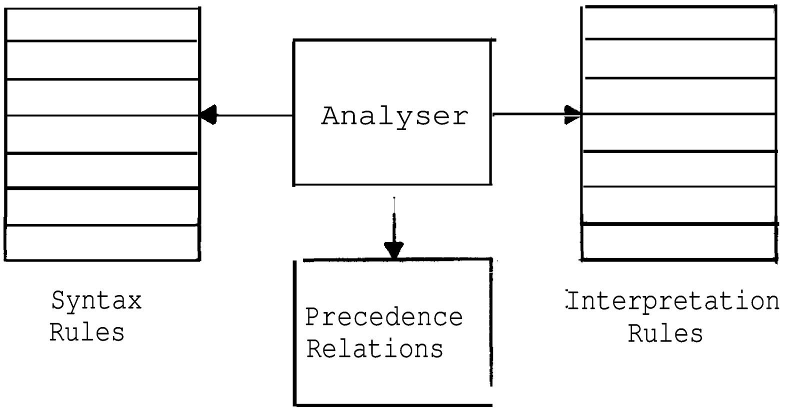
Usually, however, these facilities are not needed, because the program is executed in the environment of an operating system (whose choice is normally not up to the individual programmer) which executes programs in the program-mode where input-output instructions are not executable. The form which statements communicating with such an environment assume is determined by that particular environment and cannot be defined as part of the language proper.

VI. Compiler methodology

1. General organization

The compiler is a strictly syntax directed one-pass translator. Its design served as a major test for the applicability of the techniques described in [3] to practical programming languages. The language was designed to conform to the rules of simple precedence grammars as postulated in [3]. The development of a precedence syntax to whose individual rules the meaning of the language could be properly attached was no easy task. Interestingly enough, however, this design process provided many insights into the nature of various conceptual elements, led to their clarification and often simplification, and contributed a great deal to the systematic structure of the resulting language.

The algorithm for syntactic analysis constitutes the core of the compiler. It operates on the basis of a table containing the rules of syntax and a table containing the precedence relations among input tokens, and evokes the execution of an interpretation rule whenever a parsing step is taken. The input tokens are obtained by calling a procedure called "insymbol", which scans the sequence of input characters in the manner of a finite state machine, and yields as a result either a basic symbol of the language, an identifier, a number, or a string. It automatically suppresses comments. It should be noted, that in the implemented language no equivalent for the underlining of basic symbols is



provided, and that therefore a sequence of letters and digits, starting with a letter and not containing blanks, may constitute a basic symbol. Any such sequence must be matched by the **insymbol** routine against a table containing the representations of all "letter-symbols". If a match is found, the result is a basic symbol, otherwise an identifier. As a consequence, identifiers could not be constructed by the syntax analyser itself upon receiving merely a sequence of letters and digits. The consideration of numbers as tokens, on the other hand, was not a necessity but rather a convenience.

The syntax analyser makes use of a stack (called "symbol stack") to store not yet reduced symbols. Whenever a reduction takes place, the interpretation rule corresponding to the applied syntactic rule is activated. These interpretation rules make use of a second stack (called "value stack") to store information about each syntactic entity occurring in the reduction process. To each entry in the symbol stack corresponds an entry in the value stack, and vice-versa. Ideally, an interpretation rule should exclusively reference data in those entries of the value stack which correspond to symbols in the symbol stack being reduced by the applying syntactic rule. This principle has been followed in the simple example presented in [3]. Here, however, a deviation from it was made by the introduction of conventional identifier tables, one containing identifiers denoting program points (labels), one for all declared identifiers.

2. Identifier tables

The presence of identifer tables simplifies the search for identifiers and eliminates the need for the specific right recursive definition of the declaration structure used in [3]. The separation of the table into one containing declared identifiers and one containing labels has its reason in the fact that labels are the only identifiers which can occur in a statement before being defined in the program, and must therefore be treated differently as discussed below.

It should first be noted that the presence of the syntax rules

(1) ⟨J cell identifier) ::= (identifier)

(2) (function identifier) ::= (identifier)

(3) (procedure identifier) ::= (identifier)

etc.

constitutes a violation of the requirement that in an unambiguous precedence grammar no two rules should have identical right parts. This violation required a slight complication of the analysis algorithm with the effect that an interpretation rule may cause an otherwise applicable syntactic rule to be rejected. In the given example, the interpretation rules specify that the considered identifier be located in the identifier table. If location is successful, then rule 1 is rejected unless the table indicates that the identifier indeed designates a $\mathcal{J}$ cell, rule 2 is rejected unless it designates a function, etc. This decision of the applicability of a syntactic rule on grounds of essentially semantic information reflects the argument that "Algol-like languages" are strictly speaking not context free, i.e., cannot be described by a phrase structure grammar alone.

The above identifier search implies that the entire block-structured identifier table be searched. The following program demonstrates that labels cannot be subjected to the same process, and that therefore

(4) (label) ::= (identifier)

must not be a rule of the language.

```
A:  begin. . .
B:       begin goto L;
         ...
              L:
         end;
    end
```

In this example, rule 4 applying to L after the symbol goto would detect L as present in the identifier table, because L was defined as a label in the outer block (A). This would, however, be an erroneous assumption, since a local L is defined later in the inner block (B), to which goto L should refer. Consequently, searches for labels must be confined to the innermost block, and such a restricted search must be represented by an interpretation rule connected with a distinct syntactic rule with a different right part. In the language, that rule is

(go to statement) ::= goto (identifier)

Identifiers in the label table are marked as either defined or not yet defined. Upon exit of a block, all undefined entries are. collected and considered as entries in the outer block, where some of them may be found as already defined. This process made the use of a separate label table desirable.

The compiler is designed to read the source program from cards or tape; it produces (optionally) a listing, each line containing a corresponding target program address. The code is compiled into core memory, and as soon as a segment is closed, it is written onto secondary storage. The segment is preceded by a record indicating the kind of the segment (program or data), its number, and its length. The program loader later collects the segments from the secondary storage, lists the base address which it assigns to each segment, and assigns it to the corresponding entry in the segment address table.

3. Handling of syntactic errors

The syntax analysis algorithm described in [3] makes the assumption that analysed programs are syntactically valid. This assumption is not tenable in the practical world of computer programming. Syntactic errors are detected by the fact that for some string recognized as reducible there is no matching entry in the table of productions. After an error has been encountered, it is in most cases desirable to continue compilation in order that subsequent errors may be located and indicated. A method has to be devised to let the analysis algorithm proceed after having made some assumption about the nature of the error.

This is in general a rather hopeless task. An investigation of a large number of programs containing syntactic errors reveals, however, that most of the committed errors exhibit strong similarities and can be diagnosed by a relatively simple algorithm. In most cases, syntactic errors are due to omission or wrong use of symbols merely conveying information about structural properties of the program, such as interpunctuation symbols and the various kinds of brackets. Omission of elements explicitly stating program activities, such as operators and operands, are rare.

A second important consideration is that an incorrect construction should be detected as early as possible, i.e., before further steps are taken on the basis of the incorrect text. The precedence grammar technique is an excellent scheme in this respect, because it is based upon relations existing among symbol pairs. That none of the relations denoted by $\lessdot$, $\doteq$, 3 exists between two symbols implies the impossibility of these two symbols being adjacent in any sentence of the language. The empty relation (denoted by $\odot$) shall be defined as holding whenever none of the others hold. On a left-to-right scan, its encounter constitutes the earliest possible detection of an erroneous construction.

It should be noted that the use of two precedence functions instead of the precedence relations implies that the analysis algorithm is based on a condensation of the information contained in the matrix of relations. This condensation relies on the assumption that empty relations can simply be ignored. The above considerations lead to the conclusion that for practical reasons it is advantageous to have the relation matrix at the disposal of the analyser rather than the functions.

The algorithm for diagnosing of and recovery from errors described subsequently is a heuristic solution rather than one based on rigorous theoretical principles. It is contended here that any such scheme must make a very drastic selection from all the possible forms which errors may assume. The important aspect is that those situations are mastered intelligently which are likely to occur often,, Since a frequency statistic of errors reflects the behavior of the human users, such a selection must by definition be based on heuristics.

There exist two places in the analysis process, where illegal constructions may be detected (cf. [3], p. 18):

1. The empty relation holds between the symbol on top of the stack and the incoming symbol:

$$S_i \odot P_k$$

In this case a list I of insertion symbols is scanned. If for some m, $S_i \phi I_m$ and $I_m \phi P_k$, then I_m is inserted into the scanned string in front of P_k. Since this insertion may lead to a correct program (in about 90% of the tested cases it did), an according comment' must be delivered to the programmer.

If for no m, $S_i \phi I_m$ and $I_m \phi P_k$, then the symbol P_k is stacked.

2. The value of the function

$$\text{Leftpart}(S_j \ldots S_i)$$

is undefined (Ω), i.e., there exists no syntactic rule whose rightpart is $S_j \ldots S_i$. This situation may occur even if for all k ($j < k < i$) $S_k \phi S_{k+1}$.

In this case a table of <u>erroneous productions</u> is scanned for a rightpart identical to $S_j \ldots S_i$. If a match is found, an error message corresponding to that rule can be printed, and the analysis can proceed with the statement

$$S_j := \text{Leftpart}(S_j \ldots S_i) \ .$$

The augmented algorithm for syntactic analysis is then described as follows:

```
procedure Invalid pair;
begin integer m; m := 1;
    while $m < n$ A $(S_j \odot I_m \vee I_m \odot P_k)$ do m := m+1;
    if $m \leq n$ then $(P_k \; . \; . \; . \; P_{z+1}) := I_m$ cat $(P_k \; . \; . \; . \; P_z)$
end;

while $P_k \neq$ "⊥" do
begin i := j := j+1; $S_j := P_k$; k := k+1;
    while $S_j \bullet >\circ P_k$ do
    begin if $S_j \odot P_k$ then Invalid pair;
        while $S_{j-1} \doteq\odot S_j$ do j := j-1;
        $S_j$ := Leftpart($S_j \; . \; . \; . \; S_i$); i := j
    end
end
```

In the specific case of the PL360 language, the selected insertion symbols I_m are

; ( )

The following are the selected erroneous productions:

0 ⟨𝔎 register assignment) ::=
⟨𝔗 cell designator) := ⟨𝔗 value)

1 ⟨𝔎 register assignment) ::=
⟨𝔗 cell designator) := (monadic operator⟩⟨𝔗 value)

2 ⟨𝔎 register assignment) ::=
⟨𝔗 cell designator := (monadic operator⟩⟨𝔗 register)

3 (blockbody) ::= ⟨blockbody⟩⟨statement⟩; else

4 (case sequence) ::= (case sequence⟩⟨statement⟩; else

5 (function statement) ::= (function statement))

6 ⟨𝔗 cell designator) ::= ⟨𝔗 cell designator))

7 (procedure head) ::= (procedure name);

8 (condition) ::=
⟨𝔗 cell identifier⟩⟨relational operator⟩⟨𝔗 value)

9 (condition) ::= ⟨𝔗 cell identifier⟩⟨relational operator⟩⟨𝔎 register)

10 (block head) := (block body⟩⟨declaration⟩;

11 ⟨𝔗 cell designator) ::=
⟨𝔗 cell designator⟩(⟨𝔗 number))

12 (simple 𝔗 type) ::=
(simple 𝔗 type) array ((integer number))

13 (procedure identifier) := (procedure identifier)@ register))

14 (statement) ::= ⟨blockbody⟩⟨statement⟩

The following table of messages accompanies the erroneous productions. If some erroneous production is found to be applicable, the corresponding message is transmitted to the programmer.

0,1,2 Assignment must occur either to or from a register.

3,4 else must not be preceded by a semicolon.

5,6) without matching (.

7 A register specification is missing in the procedure heading.

8,9 The first comparand must be a register.

10 A statement cannot be followed by a declaration.

11 Write "(cell designator⟩(⟨integer number⟩)(⟨integer register))"
instead of
"(cell designator⟩(⟨integer register⟩)(⟨integer number))"

12 array should be the first symbol in the declaration.

13 Procedure statement must not have a parameter.

14 The symbol end is missing.

With these limited facilities, the syntax analyser was able to parse and correctly diagnose the texts in which the following erroneous constructions were contained. The produced diagnostic messages are indicated by their number enclosed in parentheses at the right margin, while arrows indicate the position where the analyser detected and diagnosed the error:

```
begin real x; RO := a end                         missing ;
                        ↑
begin real x RO := a; end                         missing ;
              ↑

if R1 = a then R1 := b; else R1 := c              (3 )
                               ↑

P; P(R1);↑                                        (13)

LA(RO( R1); LA(RO)(R1));                          missing )
      ↑                 ↑                         (5)

array (5) integer m (12)(23) 34)(45(56);          missing (
                               ↑   ↑              missing )

RO := R1; real x, y;                              (10)
                    ↑

a := b; a := abs b; a := abs RO; b := neg R1;     (0)(1)(2)(2)
       ↑           ↑            ↑            ↑

x(R1)) := b;                                      (6)(0)
         ↑  ↑

begin if a = b then goto L;                       (8)
                  ↑
if a<R1 then goto L; else goto K end              (9)(3)
            ↑                 ↑ ↑
                                                  missing;
```

As can be seen from the later examples, the analyser is able to correctly diagnose even nested errors and relate them to their context. The diagnostic messages are meaningful, because the analyser has found applicable an erroneous production which was anticipated by the compiler designer, who in turn was able to associate an appropriate comment, knowing the reasons why human programmers inadvertantly use such a construction. It was found to be helpful to let the compiler list, in addition to the message, the symbols currently in the parsing stack. They represent all the unfinished syntactical entities in the parse, and give the programmer valuable guidance toward understanding of his misuse of the syntax.

The choice of the appropriate insertion symbols and erroneous productions requires a thorough-understanding of the analysis algorithm on on the part of the compiler designer, as well as a subtle feeling to anticipate frequent misuses of the syntax. Of course, further insertion symbols and productions can easily be added to the tables in order to increase the diagnostic capabilities. of the analyser. If a compiler is capable of gathering statistical information about encountered erroneous situations, this information could be evaluated from time to time in order to expand the tables. As a result the compiler would truly seem to adapt itself to its imperfect human environment in order to gradually become a better and better teacher.

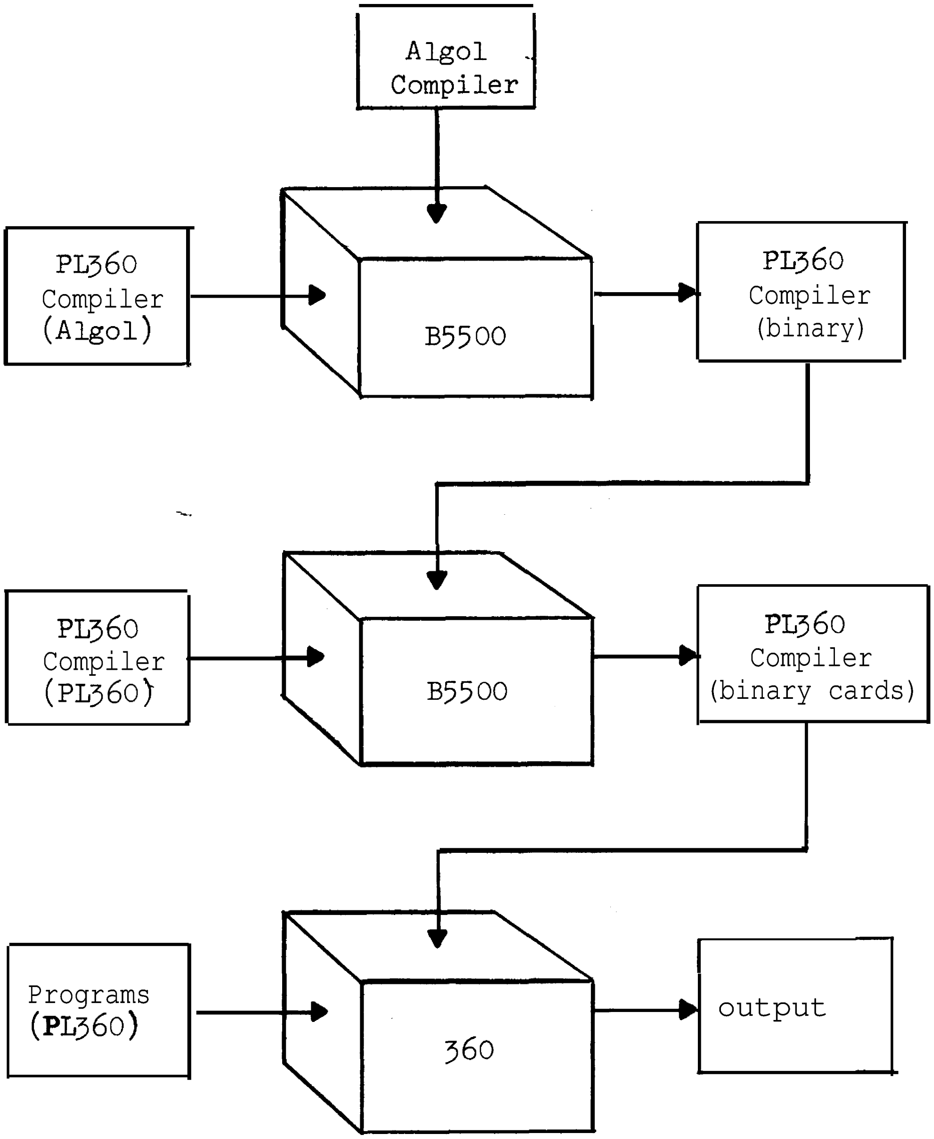
VII. The development of the compiler

At the time when the project to develop a compiler for PL360 was started, no 360 computer was available to the author, nor did the facilities promised with the forthcoming machine look too enticing to use. It was therefore decided to use the available Burroughs B5500 computer for the design and testing of the compiler, which was completed by the author within two months of part time work. It accepted a preliminary version of PL360 as described in [5] which contained the basic features of the presently described language.

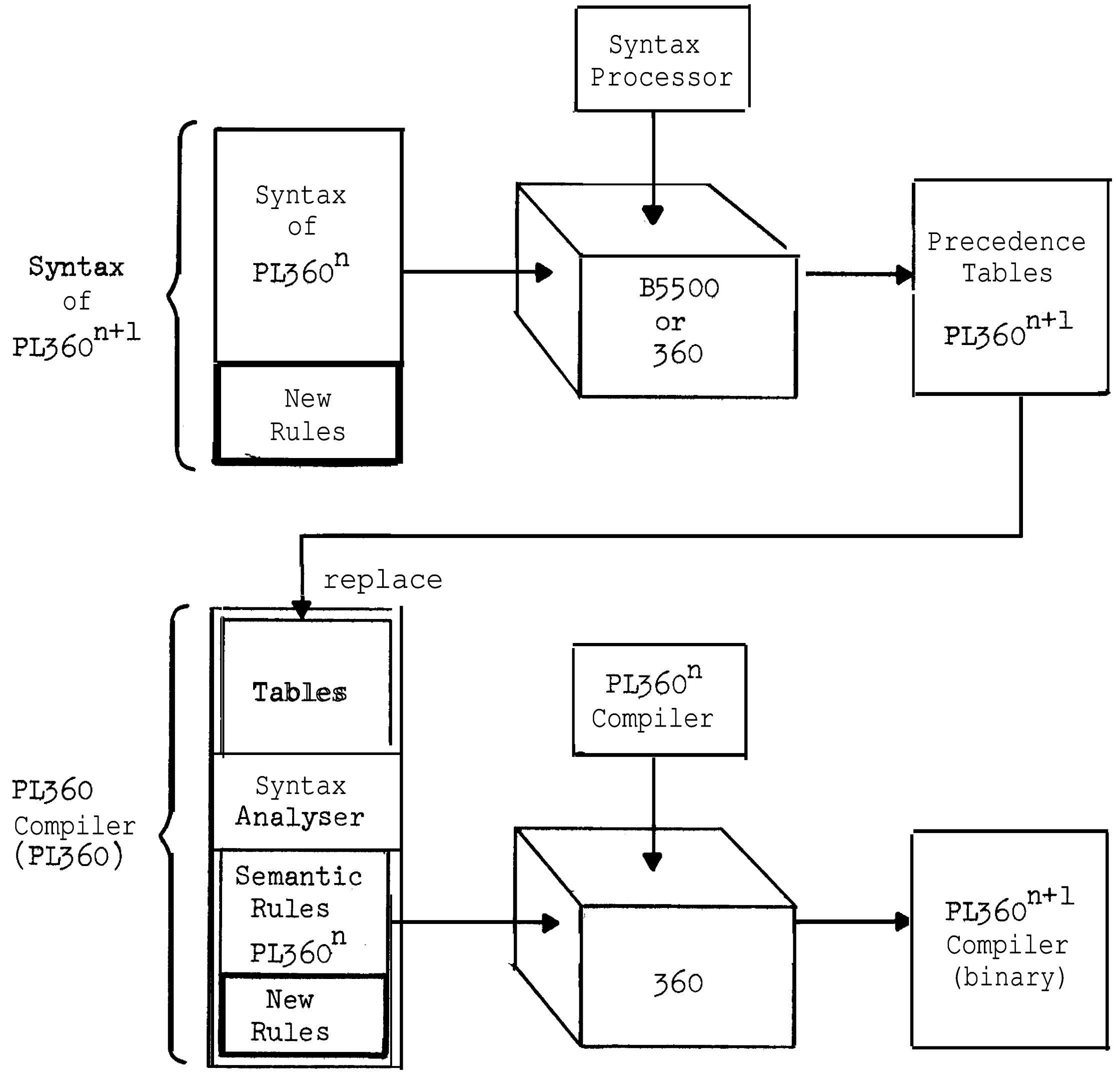
The compiler was then reprogrammed in its own language. Through a loader and supervisor program (written in assembly code), the program, recompiled on the B5500, became immediately available on the 360 computer.

The experiment of describing the compiling algorithm in PL360 itself proved to be the most effective test on the usefulness and appropriateness of the language, and it influenced the subsequent development of the language considerably. During this process, several features which seemed desirable were added to the language, and many were dropped again after having proved to be either dubious in value,, inconsistent with the design criteria, or too involved and leading to misconceptions. The leading principle and guideline was to produce a conceptually simple language and to keep the number of features and facilities minimal. The "bootstrapping" method in combination with the described compiling technique proved to be very successful for experimentation with and alteration of the language. The process of incorporation of a new feature consists of representing the new feature in the syntax of the language, and of defining the compiler actions corresponding to the new constructs in the form of additional interpretation rules. These rules must of 'course be denoted in terms of previously available facilities.

In general, a significant drawback of the bootstrapping technique is the fact that programming errors are easily proliferated. However, the combination of the bootstrapping method with the rigorous approach to systematic compiler organization by means of strict syntax analysis proved to be very successful, since the latter constitutes an enormous step towards reliability, which can never be achieved by common heuristic methods of compiler design.



Process of bootstrapping initial version of PL360 Compiler from B5500 to 360 computer.



Process of bootstrapping compiler version
n into version n+1 .

VIII. Performance

The development of a job control and supervisor program was undertaken in parallel with the construction of the compiler. The following performance figures reflect the operation of the compiler under that supervisor. It should be noted that the supervisor considers the compiler in the same way as a regular user's program.

Size (in bytes)

Supervisor	3 500	
Job control	3 700	
		7 200
Compiler program	12 700	
Various compiler data	5 400	18 100
Identifier tables	14 400	
Output area	24 600	
	30 000	39 000
		64 300

Timing

The processing of a job consists of the following steps, described in terms of the present implementation on a 360/50 computer:

1. Loading of the compiler from tape
2. Compilation, with input from cards or tape, and output to tape (and optionally to cards)
3. Loading of the compiled program from tape (or cards)
4. Execution of the program.

Steps 1 and 3, constituting what is usually called "overhead", take 4.7 secs. execution time. Compilation proceeds at the speed of the card reader (1000 cpm). If the source program is read from tape and the program listing is suppressed, the compiler (about 1500 card records) recompiles itself in 39 secs (with listing in 109 secs). The time required to load the system initially is 2 secs.

IX. Reflections on the 360 architecture

Based on the experiences drawn from the compiler development, it can be concluded that the objective to make direct machine programming more convenient by providing a tool which is superior to common assembly codes with respect to readability and writability, is commendable and important. It can also be concluded that PL360 is fairly successful in meeting this objective. The decisive factor, in the author's opinion, is the simplicity, frugality, and coherence of the language. A limiting factor to this is the architecture of the underlying machine. In this respect, the question "how well is the computer suited for this kind of language?" becomes more significant than the opposite question "how well is the language suited for the machine?". The author feels indeed strongly about this point, and recommends future hardware designers to confront themselves seriously with the first question, before yielding to the well-known policy of answering every problem with the common and omnipotent reply: "There is a bit somewhere".

As a matter of fact, the relatively systematic architecture of the 360 computer series provided a strong encouragement to devise a tool in the sense of PL360. It seems nevertheless worth while to locate some of its less fortunate features:

1. The idea of a "two-dimensional instruction set" with one coordinate specifying the operation, the other the type of operand, is very commendable, and is properly reflected in PL360. But, the better a principle is, the worse are its violations. There exist operands of type full word integer, half word integer, full word logical, short and long floating point, and byte in the 360 system. Operations on them are more or less grouped into columns in the matrix of instructions. However, instructions on logical and full word integer operands occur in the same column, certain operations are missing in the half word format, and operations on bytes differ radically from all others. A striking example is the inconsistency of the LH and STH instructions, the first of which performs the function of assigning an integer to a register, the second one that of assigning a halfword logical quantity to a memory cell. This is not merely an unfortunate feature, but a conceptual flaw.

2. The fact that many instructions are indexable only through misuse of the base register field is very unfortunate. It is one reason why none of those instructions fits into the scheme of the PL360 assignment statement.

3. The more complex a single instruction,the more debatable becomes the choice of its detailed form. The BCT, BXLE, BXH are good examples, none of which fitted into the scheme of PL360 structures.

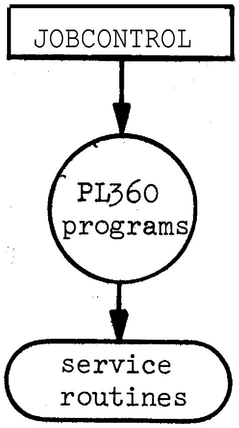
4. The 360 instructions exhibit a remarkable consistency in the scheme of condition code setting, with the very peculiar exception of the TM instruction.

This short list of architectural misfits is by no means complete. It omits,e.g., mentioning some dismal properties of the floating point arithmetic and of the input-output mechanism. However, these have no immediate effect on the structure of the PL360 language.

X. How to use PL360

This chapter is intended to serve as a reference manual for the user of the PL360 language as implemented on the GSG/SRD 360/50 computer at SLAC. It describes the facilities and the usage of the compiler and operating system, version Nov. 1966.

The operating system consists of a batch processing jobcontrol program, and a set of elementary input output service routines with associated interrupt programs. The jobcontrol program incorporates a loader, reading binary programs from either tape or cards, and it treats programs to be executed, including the PL360 compiler itself, as subroutines.



The jobcontrol program and the service routines are executed in the supervisor mode and are storage protected. Together they occupy the first 8000 bytes of core memory.

1. The language

The implemented language is that described in Chapter II,$_I$ with the following symbol representations, restrictions, and extension:

a) Symbol representation

Only capital letters are available. Basic symbols which are denoted by underlined letter sequence in Chapter II are denoted by the same sequence of capital letters. Such sequences may not be used as identifiers. They are tabulated in X.8.

b) Restrictions

No go-to statement may refer to a label in a segment different from the one where the goto statement occurs.

Oniy the first 10 characters of identifiers are significant.

c) Extension

To facilitate program debugging, a dump statement has been introduced.

Syntax:

(simple statement) ::= (dump statement)

(dump statement) ::= (dump heading⟩(⟨length part))

(dump heading) ::= dump (⟨𝒯 cell designator))

(length part⟩ ::= (integer register⟩|⟨integer cell designator)\
(short integer cell designator⟩|⟨integer number)

The dump statement causes the listing in hexadecimal form of the values of the n consecutive memory cells (-bytes), the first of which is designated by the 𝒯 cell designator. n is the value of the length part.

d) Additional standard functions

A set of standard functions is defined as supervisor calls for elementary input and output operations. The referenced supervisor routines make use of parameter registers as specified below. They set the condition code to 0, unless otherwise specified. Input-output devices are designated by logical unit numbers (cf. X.8.).

READ — Read a card, assign the 80 character record to the memory area designated by the address in register 0 , Set the condition code to 1, if the end of the card file is encountered.

READ026 — Same as READ, with the addition of a character code translation as specified in section X.8. The translation maps 026 punched characters into their 029 equivalents.*

WRITE Write the record of 132 characters designated by the address in register 0 on the line printer. Set the condition code to 1, if the next line to be printed appears on the top of a new page.

PUNCH Punch the record of 80 characters designated by the address in register 0 on the card punch.

READTAPE Read a record from the tape unit specified by the logical unit number is register 2 . The length of the record in bytes in specified by register 1, and it is assigned to the memory area designated by the address in register 0 . Set the condition code to 1, if a tape mark is encountered, Register 1 is assigned the number of bytes actually read.

WRITETAPE Write a record on the tape unit specified by the logical unit number in register 2 . The length of the written record in bytes is specified by register 1; the record is designated by the address in register 0 .

PAGE Skip to the next page on the line printer.

The following are tape handling functions. They affect the tape unit specified by the logical unit number in register 2 .

MARKTAPE: Write a tape mark.
REWIND: Rewind the tape.
BSPREC: Backspace one record.
FSPREC: Forwardspace one record.
BSPTM: Backspace to the previous tape mark,
FSPTM: Forwardspace to the next tape mark.

A program interruption (cf. X.5.) due to arithmetic operations records the interruption code in the byte cell FPI. This cell, being part of the supervisor, is memory protected, and cannot be reset by the user's program directly.

FPIRESET: Reset the value of the cell FPI to 0 .

2. Compiler instructions

The compiler accepts instructions occurring anywhere in the sequence of input records. A compiler instruction card is marked by a $ character in column 1, and an instruction in columns 2-4. Columns 5-80 of such a record are ignored.

$026	The compiler assumes subsequent source cards to be punched on 026 keypunches.
$029	The compiler assumes subsequent source cards to be punched on 029 keypunches.
$LIST	Subsequent source records are listed on the printer.
$NØLIST	Subsequent source records are not listed.
$PUNCH	Computed program and data segments are punched on cards.
$PAGE	A page is skipped in the listing.
$0	No trace output is listed.
$1	The relative address of all variables and procedures are listed when they are declared.
$2	Addresses are listed as after $1, and the produced machine code is listed in hexadecimal notation.
$TAPEn	The subsequent source records are read from the tape unit with logical number n .

3. Compiler error messages

Errors are indicated by the compiler with a message and a bar below the character which was read last.

Error No.	Message	Meaning
00	SYNTAX	The source program violates the PL360 syntax. Analysis continues with the next statement,
01	VAR ASS TYPES	The type of operands in a variable assignment are incompatible.

Error No.	Message	Meaning
02	FOR PARAMETER	A real register instead of an integer register is specified in a for clause.
03	REG ASS TYPES	The types of operands in a register assignment are incompatible.
04	BIN OP TYPES	The types of operands of an arithmetic or logical operator, are incompatible.
05	SHIFT OP	A real instead of an integer register is specified in a shift operation.
06	COMPARE TYPES	The types of comparands are incompatible.
07	REG TYPE OR #	Incorrect register specification.
08	UNDEFINED ID	An undeclared identifier is encountered.
09	MULT LAB DEF	The same identifier is defined as a label more than once in the same block.
10	EXC IN1 VALUE	The number of initializing values exceeds the number of elements in the array.
11	NOT INDEXABLE	The function argument does not allow for an index register.
12	DATA OVERFLOW	The address of the declared variable in the data segment exceeds 4095.
13	NO OF ARGS	An incorrect number of arguments is used for a function.
14	ILLEGAL CHAR	An illegal character was encountered; it is skipped.
15	MULTIPLE ID	The same identifier is declared more than once in the same block.
16	PROGRAM OFLOW	The current program segment is too large.
17	INITIAL OFLOW	The area of initialized data in the compiler is full. This can be circumvented by suitable segmentation.

Error No.	Message	Meaning
18	ADDRESS OFLOW	The number used as index is such that the resulting address cannot be accommodated.
19	INTEGER OFLOW	The integer number is too large in magnitude.
20	MISSING @	An end of file has been read before a program terminating @ was encountered.
21	STRING LENGTH	The length of a string is either 0 or > 256 .
22	DUMP TYPE	The length part does not specify an integer.
23	FUNC DEF NO.	The format number in a function declaration is illegal.

At the end of each program segment, undefined labels are listed with an indication where they occurred.

4. Jobcontrol instructions, the form of input card decks

Cards containing a 0-2-8 punch in column 1 are recognized by the "READ" and "READ026" supervisor routines as jobcontrol cards, and give rise to an end of file indication. Information contained in columns 2-9 (left adjusted) of such cards is interpreted by the job control routine as follows:

PL360 Control is given to the compiler to process the subsequent source program.

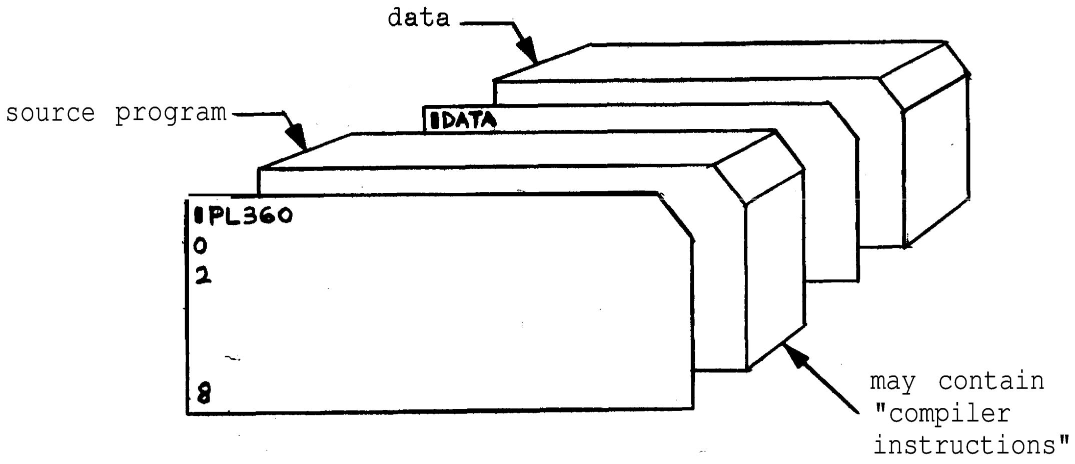
DATA Control is given to the previously compiled and/or loaded program. If the preceding compilation detected any errors, the subsequent data cards are skipped.

LOAD Control is given to the loader routine, which loads subsequent "binary" program cards.

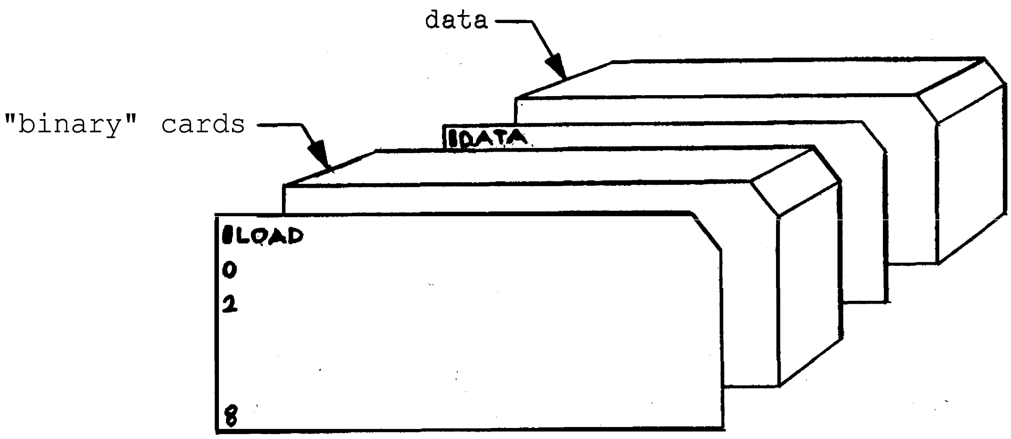
PAUSE The operator is notified, and the system waits for the operator's instructions given via the operator console typewriter (cf. x.6.).

Other control cards are recognized and may be used to activate library programs, which are not described in this Report.

Typical card deck compositions are:



Compilation and execution



Loading and execution

5. Program execution errors

The following error conditions can occur:

a. A "program-check" interruption occurred. This is indicated by the message

```
PRG PSW XXXXXXXXXXXXXXXX
```

If interruption occurred due to an arithmetic operation, the interruption code is stored in the byte cell FPI (floating point interruption), and control is returned to the interrupted program? Otherwise, control is given to the job control routine.

b. An attempt is made to read beyond a control card. The message

```
EOF PSW XXXXXXXXXXXXXXXX
```

is printed, and control is returned to the job control routine.

c. An illegal logical unit number has been used for an input-output operation. The message

```
DEV PSW XXXXXXXXXXXXXXXX
```

is printed, and control is returned to the job control routine.

d. The operator intervenes by causing an external interrupt. The message

```
EXT PSW XXXXXXXXXXXXXXXX
```

appears on the line printer and the operator console. (cf. X.6.).

6. Minimal configuration reauirements

Core memory: 65 K bytes, protection feature

1 card reader/punch (2540)

1 line printer (1403)

2 tape units (2401-3)

1 console typewriter (1052) (dev. addr. 009)

*Such interrupts are counted, and the counts are listed (if ≠ 0) after the end of program execution.

7. Loading and operating the system

The process of initial loading consists of the following steps:

a. Reset system

b. Mount system tape on any 9-track unit (usually device 282)

c. Stack jobs on the card reader

d. Make card reader, line printer, and tape 5 (used by the compiler) ready.

e. Select the unit carrying the system tape on the Load Unit Switches'

f. Press the Load Key

Execution of the job sequence stacked on the card reader is immediately started. Control is returned to the operator when either

a. a PAUSE control card is encountered, or

b. the operator presses the external interrupt key.

The computer then accepts instructions from the operator via typewriter. Each message must be terminated with an EOB (end of block) character. The following free-field instructions are accepted:

a. dump XXXXXX, NNNNNN EOB

dump NNNNNN EOB

dump EOB

The values of the registers and of the NNNNNN byte cells starting at the initial address XXXXXX are listed in hexadecimal form. If the initial address is omitted, it is taken as the beginning of the user's data segment area, and if She count is omitted, it is taken as the length of' the user's data, segment area .

b. device XX EOB

Devices are designated by logical numbers, The correspondence between logical numbers and actual device addresses is established by the device table (cf. X.8). The above command causes the address AAA of the device with logical unit number XX to be typed out. Subsequent typing of the device address BBB causes that device to be assigned the logical unit number XX, and the device with address AAA to be given the logical unit number YY, which previously designated device BBB (if any). As a result, every device in the system will always be designated by at most one logical unit number.

before	after
XX : AAA	xx : BBB
YY : BBB	YY : AAA

c. EOB

Processing resumes with the next job in sequence.

The operator is informed about abnormal conditions encountered by the error analysis routines of the elementary input - output programs contained in the supervisor. The following messages are typed:

a. XX YYY NOT RDY
b. XX YYY NOT OP
c. XX YYY I/O ERROR CCCC DDDD
d. XX YYY DEV END CCCC DDDD

XX represents the logical number of the afflicted device, YYY its physical address, CCCC the encountered channel status, and DDDD the device status.

Message a. is given when a device is not ready. Execution resumes when the device is put into the ready state. Messages b., c., and d., are respectively given when a device is not operating, when a malfunction is encountered, or when an error is discovered upon device end interrupt caused by the reader, punch, or printer. The operator must reply with one of the following messages:

a. ignore EOB
b. exit EOB (resume processing with next job)
c. EOB (retry the operation after I/O ERROR; ignore the DEV
END condition)

Note that if a storage dump is desired before processing the next job, then the interrupt key must be pressed first. If the operator response is not recognized by the system, then "RETRY" is typed out. In order to cancel a response, the CANCEL character must be typed before typing EOB. In either case a correct response should then be typed by the operator.

8. Tables

Character code translation table (used in READ026)

holes	026 ---	029	hex.
12-3-8	.	.	4B
12-6-8	<	<	4c
o-4-8	(	(	4D
12-5-8	[	(	4D
12	+	+	4E
o-6-8	<	|	4F
12-0		&	50
11-3-8	$	$	5B
11-4-8	*	*	5C
12-4-8	)	)	5D
11-5-8	|	)	5D
u-6-8	;	;	5E
6-8	X	1	5F
11			60
0-1	/	/	61
o-3-8	,	,	6B
11-7-8		%	6C
o-5-8	←	_	6D
11-0		>	6E
5-8	:	:	7A
12-7-8		#	7B
o-7-8		@	7C
7-8		'	7D
3-8	=	=	7E
4-8	'	"	7F

Letters and digits are represented by the same hole combinations on cards punched on either the 026 or the 029 keypunches, and do therefore not undergo any translation. The column designated "026" lists the characters printed by the Stanford extended 026 keypunches.

BASK SYMBOLS

-
;
+
-
=
<
>
¬

(
)
*
/
:
,
â
:=

DO
IF
OF
OR
ABS
AND
END
FOR
NEG
SYN

XOR
BASE
BYTE
CASE
DUMP
ELSE
GOTO
LONG
NULL
REAL

SHLA
SHLL
SHRA
SHRL
STEP
THEN
ARRAY
BEGIN
SHORT
UNTIL

WHILE
COMMAND
INTEGER
LOGICAL
SEGMENT
FUNCTION
OVERFLOW
REGISTER
CHARACTER
PROCEDURE

NOTE: THESE LETTER SEQUENCES MUST NOT BE USED AS IDENTIFIERS.

STANDARD IDENTIFIERS.

ARRAY() INTEGER	MEN SYN 0·
BYTE	FPI SYN 43
INTEGER	B1 SYN MEM(R1)
INTEGER	B2 SYN MEM(R2)
INTEGER	83 SYN MEM(R3)
INTEGER	84 SYN MEM(R4)
INTEGER	85 SYN MEM(R5)
INTEGER	86 SYN MEM(R6)
INTEGER	B7 SYN MEM(R7)
INTEGER	B8 SYN MEM(R8)
INTEGER	B9 SYN MEM(R9)
INTEGER	BA SYN MEM(RA)
INTEGER	BB SYN MEM(RB)
INTEGER	BC SYN MEM(RC)
INTEGER	BD SYN MEM(RD)

```
INTEGER REGISTER   R0   (0)
INTEGER REGISTER   ii 1 (1)
INTEGER REGISTER   R2   (2)
INTEGER REGISTER   K3   (2)
INTEGER REGISTEK   R3   (3)
INTEGER REGISTER   R4   (4)
INTEGER REGISTER   R5   (5)
INTEGER REGISTER   R6   (6)
INTEGER REGISTER   R7   (7)
INTEGER REGISTER   R8   (8)
INTEGER REGISTER   R9   (9)
INTEGER REGISTER   RA   (A)
INTEGER REGISTER   RB   (B)
INTEGER REGISTER   RC   (C)
INTEGER REGISTEK   RD   (D)
REAL REGISTER      FO   (0)
REAL REGISTER      r-2  (0)
REAL REGISTER      F2   (2)
REAL REGISTER      F4   (4)
REAL REGISTER      F6   (6)
LONG HEAL REGISTER F01(0)
LONG REAL REGISTER F23(2)
LONG REAL REGISTER F45(4)
LONG REAL REGISTER F67 (6)

FUNCTION  LA         (2)(#4100)
FUNCT ION MVI        (4)(#9200)
FUNCTION  MVC        (5)(#D200)
FUNCTION  CLI        (4)(#9500)
FUhCT ION CLC        (5)(#D500)
FUNCTION  LM         (3)(#9800)
FUNCTION  STM        (3)(#9000)
FUNCTION  SLDL       (9)(#8D00)
FUNCTION  SROL       (9)(#8C00)
FUNCTION  IC         (2)(#4300)
FUhCT ION STC        (2)(#4200)
FUNCTION  CVD        (2)(#4E00)
FUhCT ION UNPK       (10)(#F300)
FUNCTION  t o        (5)(#DE00)
FUNCTION  EX         (2)(#4400)
FUNCTION  SET        (8)(#92FF)
FUNCTION  RESET      (8)(#9200)
FUhCT ION TEST       (8)(#95FF)
FUNCTION  READ       (0)(#0A00)
FUNCTION  RfA0026    (C)(#0A01)
FUhCT ION WRITE      (0)(#0A02)
FUNCT ION PUNCH      (0)(#0A03)
FUNCT ION READTAPE   (0)(#0A06)
FUNCT ION WRITETAPE  (0)(#0A07)
FUNCTION  REWIND     (0)(#0A08)
FUNCTION  MARKTAPE   (0)(#0A09)
FUNCTION  FSPTM      (0)(#0A0A)
FUNCT ION FSPREC     (0)(#0A0B)
FUNCTION  BSPTM      (C)(#0A0C)
FUNCTION  BSPREC     (0)(#0A0D)
FUNCTION  PAGE       (C)(#0A0E)
FUNCTION  FPIRESET   (0)(#0A0F)
```

SYNTAX

```
<K REG*>        ::= <ID>
<T CELL IO>     ::= <ID>
<PROC ID>       ::= <ID>
<FUNC ID>       ::= <ID>
<K REG>         ::= <K REG*>
<T CELL*>       ::=<T CELL ID>  (<T   NUMBER>   )
<T CELL>        ::=<T CELL IO>        |
                    <T CELL IO>  (  <K  REG*>  )      |
                    <T CELL*>      |
                    <T  CELL*>  (   <K  REG*>  )
<T VALUE>       ::= <T NUMBER>     |
                    <T CELL>     |
                    <STRING>
<KSI ASS>       ::= <K REG*>  :=   <T VALUE>        |
                    <K  REG*>  :=   <K REG>        1
                    <  K REG*> :=   <UNARY OP>  <T VALUE>        |
                    <K REG*>  :=   <UNARY OP> <K REG>
<UNARY OP>      ::= ABS        |
                    NEG        |
                    NEG ABS
<ARITH OP>      ::= +      |
                           |
                    *      |
                    /      |
                    + +       |
                    - -
<LCG CP>        ::= AND        |
                    CR        |
                    XOR
<SHIFT OP>      ::= SHLA        |
                    SHRA        |
                    SHLL        |
                    SHRL
<K REG ASS*>::=<KSI ASS>       |
                    <K REG ASS*> CARITH OP><T VALUE>        |
                    <K HEG ASS*:>  CARITH OP><K REG>        |
                    <K REG ASS*>   <LOG OP><T VALUE>        |
                    <K REG ASS*><LOG OP>   <K REG>      |
                    <K REG ASS*>   <SHIFT OP><T NUMBER>        |
                    <K REG ASS*><SHIFT CP>   <K REG*>
<K REG ASS>     ::= <K REG ASS*>
<FUNCTION>      ::= <FUNC ID>       |
                    <FUNCTION>  (<T   NUMBER>   )      |
                    <FUNCTICN>  (  <K  REG*>  )
                    <FUNCTION>  (  <T  CELL>  )      |
                    <FUNCTION>  (  <STRING>  )
<DUMP HEAD>     ::= DUMP  (<T   CELL>   )
<CASE SEC>      ::= CASE <K REG*> Of  BEGIN      |
                    <CASE SEQ> <STATEMENT>;
<SIMPLE ST>     ::= <T CELL>:=   <K REG>       |
                    <K REG ASS>       |
                    NULL      |
                    GOTO  <IC>       |
                    CPRGC ID>       |
                    <DUMP HEAD>  (  <T  NUMBER>  )      |
                    <DUMP HEAD>  (  <K  REG*>  )      |
                    <DUMP HEAC>  (  <T  CELL>  )      |
                    <FUNCTION>       |
                    <CASE SEQ> END       |
                    <BLOCK>
```

```
<REL OP>       ::= <      |
                   =      |
                   >      |
                   < =    |
                   > =    |
                   ¬ =
<CONDITION>    ::= <K REG*>  CRELOP><TVALUE>     |
                   <K REG*>  <REL OP>  <K REG>   |
                   CVERFLOW   |
                   <REL OP>
<IF CL>        ::= IF<CONDITION>THEN
<TRUE PART>    ::= <SIMPLEST>ELSE
<WHILE>        ::= WHILE
<CCND DO>      ::= <CCNDITICN> DO
<FOR HEAD>     ::=FCR<KREG ASS>
<INCREM>       ::= STEP<TNUMBER>
<LIMIT>        ::= UNTIL <KREG*>     |
                   UNTIL<TCELL>      |
                   UNTIL  <T NUMBER>
<DO>           ::= CC
<STATEMENT*>   ::=<SIMPLEST>     |
                   <IF CL> <STATEMENT*>     |
                   <IF CL> <TRUE PART>  <STATEMENT*>     |
                   <WHILE><COND DO> <STATEMENT*>     |
                   <FOR HEAD>  <INCREM>  <LIMIT>  <DO>  <STATEMENT*>
<STATEMENT>    ::= <STATEMENT*>
<SI T TYPE>    ::= SHORT  INTEGER     |
                   INTEGER     |
                   LOGICAL     |
                   REAL     |
                   LCNG  REAL     |
                   BYTE     |
                   CHARACTER     |
                   COMMAND
<T TYPE>       ::=<SI T TYPE>     |
                   ARRAY(  <TNUMBER3  )<SI  T  TYPE>
<T CECL*>      •. *=<TTYPE><ID>     |
                   <T DECL>  ,  <ID>     |
                   <T DECL>  (  <T  NUMBER>  )     |
                   CT UECL>  (   <STRING>   )
<T CECL>       ::= <T DECL*>
<K REG CC*>    ::= <SI T TYPE>  REGISTER  <ID>     |
                   <K  REG  DC>  ,  <IC>
<KREG C C >    ::= <K  REG  DC*>  (  <T  NUMBER>  )
<FUNC DECL*>   ::= FUNCTION  <ID>     |
                   <FUNC  DECL>  ,  <ID>
<FUNC CECL->   ::= <FUNC  DECL*>  (  <TNUMBER>)
<FUNC DECL>    ::= <FUNC  DECL->  (  <T  NUMBER>  )
<SYN DECL>     ::=<TTYPE>  <IO>  SYN  <TCELL>     |
                   <T TYPE>  <ID>  SYN<TNUMBER>     |
                   <SYN  DECL>  ,  <ID>  SYN  <T  CELL>     |
                   <SYN  DECL>  ,  <ID>  SYN  <T  NUMBER>
<CECL>         ::= <T DECL>     |
                   <KREGDC>     |
                   <FUNC DECL>     |
                   <SYN DECL>     |
                   SEGMENT BASE <K REG>
<PRCC NAME>    ::= PKCCEDURE  <ID>     |
                   SEGMENT  PROCEDURE  <ID>
<PRCC HEAD*>   ::=<PROC NAME>  (  <K  REG*>  )
```

```
<PROC HEAD>   ::= <PROC  HEAD*>  ;
<LABEL DEF>   ::=  <ID>  :
<BLOCKHEAD>   ::= BEGIN      |
                  <BLOCKHEAD>  <DECL>  ;      |
                  <BLOCKHEAD>  <PROC HEAD> <STATEMENT>   ;
<BLOCKBODY>   ::=   <BLOCKHEAD>    |
                  <BLOCKBODY>  <STATEMENT>  ;      |
                  <BLOCKBODY> <LABEL  DEF>
<BLOCK>       ::= <BLOCKBODY>  END
<PROGRAM>     ::= @  <BLOCK>  @
```

DEVICE TABLE

LOG.NO.	DEVICE		ADDRESS	
0	TYPEWRITER	(1052)	009	
1	PRINTER	(1403)	00E	
2	CARDREADER	(2540)	00C	
3	CARDPUNCH	(2540)	00D	
4	SYS TAPE	(2401)	282	
5	TAPE	(2401)	181	(7 TRACK)
6	TAPE	(2401)	182	
7	TAPE	(2401)	183	
8	TAPE	(2401)	184	
9	TAPE	(2401)	283	
10	DISK	(2311)	190	
11	DISK	(2311)	191	
12	DISK	(2311)	192	
13	DISPLAY	(2250)	2E0	

```
PROCEDURE INREAL (R9);
   BEGIN COMMENT READ CHARACTERS VIA "NEXTCHAR". RESULT IN F01;
      LONG REAL FCON1 (#4E00000000000000), FCON2 (#4700000000000000);
      INTEGER FCON1LOW SYN FCON1(4);
      SHORT INTEGER TEN (10); BYTE SIGN, EXPOSIGN;
      WHILE R0 < "0" DO
      BEGIN IF R0 = "-" THEN SET(SIGN) ELSE RESET(SIGN); NEXTCHAR;
      END ;
      BEGIN COMMENT READ THE INTEGRAL PART;
         R1 := R0 AND #F; NEXTCHAR;
         WHILE R0 >= "0" DO
         BEGIN R0 := R0 AND #F; R1 := R1 * TEN + R0; NEXTCHAR;
         END ;
         COMMENT INTEGER IN R1, NEXT CHARACTER IN R0;
         R2 := 0;  COMMENT R2 IS THE DECIMAL SCALE FACTOR;
         FCON1LOW := R1; F01 := FCON1+0D0;     COMMENT F01 := R1;
         IF R0 = "." THEN
         BEGIN COMMENT PROCESS FRACTION. BUILD NUMBER IN F01;
            NEXTCHAR;
            WHILE R0 >= "0" DO
            BEGIN R0 := R0 SHLL 4; STC(R0)(FCON2(4));
               F01 := F01 * 10D0 + FCON2; R2 := R2-1; NEXTCHAR;
            END ;
         END ;
         IF R0 = "E" THEN
         BEGIN COMMENT ADD THE SCALEFACTOR TO R2;
            NEXTCHAR; IF R0 = "-" THEN
               BEGIN SET(EXPOSIGN); NEXTCHAR;
               END ELSE RESET(EXPOSIGN);
            R1 := R0 AND #F; NEXTCHAR;
            WHILE R0 >= "0" DO
            BEGIN R0 := R0 AND #F; R1 := R1 * TEN + R0; NEXTCHAR;
            END ;
            TEST(EXPOSIGN);
            IF = THEN R2 := R2-R1 ELSE R2 := R2+R1;
         END ;
         IF R2 ¬= 0 THEN
            BEGIN COMMENT COMPUTE  F45 := 10 ** R2;
               IF R2 < 0 THEN
               BEGIN R2 := ABS R2; SET(EXPOSIGN);
               END ELSE RESET(EXPOSIGN);
               F23 := 10D0; F45 := 1D0; F67 := F45;
               WHILE R2 ¬= 0 DO
               BEGIN SRDL(R2)(1); F23 := F23*F67; F67 := F23;
                  LTR(R3)(R3); IF < THEN F45 := F45*F23;
               END ;
               TEST(EXPOSIGN);
               IF = THEN F01 := F01/F45 ELSE F01 := F01*F45;
            END ;
         TEST(SIGN); IF = THEN F01 := NEG F01;
      END ;
   END ;
```

```
PROCEDURE CUTREAL (R9);
   BEGIN COMMENT NUMBER IN F01. ADDRESS OF OUTPUT IN R1;
      LONG REAL X, FCON1 (#4E00000000000001);
      INTEGER XHIGH SYN X; INTEGER XLOW SYN X(4);
      SHORT INTEGER Q (307); BYTE SIGN;
      ARRAY (4) LOGICAL PATTERN
         (#4040214B)(#20202020)(#20204F21)(#20200000);
      ARRAY (14) CHARACTER ZERO ("   0            ");

      IF F01 = 0D0 THEN MVC(13)(B1)(ZERO) ELSE
      BEGIN IF F01 < 0D0 THEN SET(SIGN) ELSE RESET(SIGN);
         F01 := ABS F01; X := F01;
         R0 := XHIGH SHRL 24 - 64 * Q; IF < THEN R0 := R0 + 128;
         R0 := R0 SHRA 8 - 1; R2 := ABS R0;
         COMMENT  COMPUTE F45 := 10**R2;
         F23 := 10D0; F45 := 1D0; F67 := F45;
         WHILE R2 ¬= 0 DO
         BEGIN SRDL(R2)(1); F23 := F23*F67; F67 := F23;
            LTR(R3)(R3); IF < THEN F45 := F45*F23;
         END ;
         IF R0 < 0 THEN
         BEGIN F01 := F01*F45;
            WHILE F01 < 1D0 DO
            BEGIN F01 := F01*10D0; R0 := R0-1;
               R7 := R7+1;
            END ;
         END ELSE
         BEGIN F01 := F01/F45;
            WHILE F01 >= 10D0 DO
            BEGIN F01 := F01*0.1D0; R0 := R0+1;
               R8 := R8+1;
            END ;
         END ;
         F01 := F01 * 2D6 ++ FCON1; X := F01; R3 := XLOW SHRL 1;
         IF R3 >= 10000000 THEN
         BEGIN R2 := R2 / 10; R0 := R0+1;
         END ;
         CVD(R3)(X); MVC(13)(B1)(PATTERN); ED(9)(B1)(X(4));
         TEST(SIGN); IF = THEN MVI("-")(B1(1));
         CVD(R0)(X); ED(3)(B1(10))(X(6));
         IF R0 < 0 THEN MVI("-")(B1(11)) ELSE MVI("+")(B1(11));
      END ;
   END ;
```

```
BEGIN COMMENT BINARY ID SEARCH;
   INTEGER N; LONG REAL DEC;
   ARRAY (3) SHORT INTEGER MOVE (#D200)(#4000)(#6000);
   ARRAY (6) BYTE PATTERN (#40)(#20)(#20)(#20)(#20)(#20);
   COMMENT EACH ENTRY IN THE DIRECTCRY CONSISTS OF A TAG, LENGTH,
      ANC ABSOLUTE ACCRESS OF THE IDENTIFIER;
   ARRAY (1CC) INTEGER DIRECTORY;
   ARRAY (100) SHORT INTEGER TAG SYN DIRECTORY(0);
   ARRAY (100) SHORT INTEGER LN  SYN DIRECTORY(2);
   ARRAY (1CO) INTEGER ADR SYN DIRECTCRY(4);
   ARRAY (132) CHARACTER BUF (" ");
   ARRAY (1000) CHARACTER ID ;

PROCEDURE SEARCH (RF!);
   BEGIN COMMENT PARAMETERS: R1 = LENGTH OF IDENTIFIER, R2 = ADDRESS
      OF IDENTIFIER.  RESULT: R3  = TAG(0 I F NOT LOCATED).
      REGISTERS USED: 0 - 8   ;
      INTEGER REGISTER L(1), LOW(3), I(4), HIGH(5), X(6), M(7);
      ARRAY (3) SHORT INTEGER COMP (#D500)(#2000)(#6000);
      COMMENT COMPARE CHARACTER INSTRUCTION;
      HIG 1-i := N; LOW:= 8 ;
      WHILE LOW <= HIGH DO
      BEGIN I := LOW t HIGH  SHHC  4  SHLL 3: X := ADR(I);
         IF L = LN(I) THEN
         BEGIN EX(L)(COMP); IF = THEN GOT0 OUT:
            IF <  THEN HIGH := I-R ELSE LOW := I+8;
         END ELSC
         I f -L < LN(I) THEN
         BEGIN EX(L)(COMP);
            IF <= THEN HIGH := I-8 ELSE LOW:= I+8;
         END ELSE
         BEGIN M := LN(I); EX(M)(COMP);
            IF  <  THEN HIGH:= I-R ELSE LQW:= I+8;
         END ;
      END;
      I :=  0;
OUT: R  3 := TAG(I);
   END ;

   ED(131) (BUF)(BUF); LA(R0)(BUF); R6 := R0;   COMMENT BLANK BUFFER;
   R1 := 0 ; LA(R4)(ID);

   COMMENT READ IDENTIFIERS AND ENTER THEM IN TABLES;
L1: READ026; CLI("$")(BUF); I  F ¬= THEN
   BEGIN R1 := R1+8; TAG(R1) :   = R1;
      R2 := 0; R3 := R2;
L2:   IC(R3)(BUF(R2)); IF R3 ¬= "" THEN
      BEGIN R2 := R2+1; GOTO L2;
      END;
      EX (R2) (MOVE ); ADR(R1) := R4;
      R4 := R4+R2; R2 :   = R2-1; LN(R1) := R2;
      GOTO 1.1;
   END ;
   h : = R1;

   COMMENT READ AN IDENTIFIER AND SEARCH IT IN THE TABLE;
L3: READ026; If-  = THEN
   BEGIN R1 :  =   0  ; R3 := R1; LA(R2)(BUF);
```

```
L4:     IC(R3)(BUF(R1));IF R 3¬=" "THEN
        BEGIN R1 :  =R1+1; GCTO L4;
        ENC ;
        R1 := R1-1; SEARCH;
        CVC(R3)(CEC); MVC(5)(BUF(36))(PATTERN);
        ED(5)(BUF(36))(CEC(5));WRITE; COMMENT WRITE IDENTIFIER AND TAG;
        GOTO L3;
    ENC ;
ENC a
```

```
BEGIN COMMENT MAGIC SQUARE GENERATOR;
    ARRAY (132) CHARACTER LINE (" ");
    ARRAY (8) BYTE PATTERN(#40)(#20)(#20)(#20)(#21)(#21);
    LCNG REAL DEC;
    ARRAY (256) INTEGER X;

PRCCEDURE MAGICSQUARE (R6) ;
    BEGIN SHORT INTEGER NSQR;
        INTEGER REGISTER N(0),I(1),J(2),K(5);
        NSCR := N; R1 := N*NSQR; NSQR := R1;
        I := N+1 SHRL 1; J := N;
        FOR K := 1 STEP 1 UNTIL NSQR 00
        BEGIN R3 :=I SHLL 6; R4 := J SHLL 2 + R3; R3 := X(R4);
            IF R3 ¬= C THEN
            BEGIN I := I-l; J := J-7;
                IF I < 1 THEN I := I+N;
                IF J < 1 Tt-EN J := J+N;
                R3 := 1 SHLL 6; R4 := J SHLL 2 + R3;
            END;
            X(R4):=K ;
            I := I+1; IF I > N THEN I := I-N;
            J : =J+1; IF J > N THEN J := J-h;
        END ;
    ENC ;

PRCCEDURE GETANDPRINT(R8);
    BEGIN R 2:=0; FGR R1 :=0 STEP 4 UNTIL 1020 DO X(R1):=R2;
        MAGICSQUARE; R6 : = R O ;LA(RO)(LINE);
        FOR R1 := 1 STEP 1 UNTIL R6 DO
        BEGIN R4 : =R1 SHLL 6 +4; LA(R5)(LINE(4));
            FUR R2:= 1 STEP 1 UNTIL R6 00
            BEGIN MVC(5)(R5)(PATTERN); R3 := X(R4); CVD(R3)(DEC);
                ED(5)(B5)(CEC(5)); R4 : =R4+4; R5 := R5+7;
            END ;
            WRITE;
        END ;
        FD(131)(LINE)(LINE); WRITE:
    ENC ;

    ED(131)(LINE)(LINE);   CCMMENT BLANK LINE;
    RO := 3 ; GETANDPRINT;
    AC := 5 ; GETANDPRINT;
    KC := 7 ; GETANDPRINT;
ENC a
```

4	3	8
9	5	1
2	7	6

11	10	4	23	17 --
18	12	6	5	24
25	19	13	7	1
2	21	2C	14	8
9	3	22	16	15

22	21	13	5	46	38	30
31	23	15	14	6	47	39
4C	32	24	16	8	7	48
49	41	33	25	17	9	1
2	43	42	34	26	18	10
11	3	44	36	35	27	19
2c	12	4	45	37	29	28

Acknowledgments

The author wishes to express his sincere thanks to Mr. J. W. Wells for his indispensable assistance. Mr. Wells recoded the compiler in its own language, and developed the supporting monitor system. Thanks are also due to the GSG group at the Stanford Linear Accelerator Center for their generous providing of computer time under favourable conditions. And finally, the support of the National Science Foundation under grant GP 4053 is gratefully acknowledged.

References

1. G. M. Amdahl, G. A. Blaauw, F. P. Brook, Jr. : "Architecture of the IBM System/360", IBM J. of Res. and Dev. 8, No. 2, 87-101 (April 1964), and

   G. A. Blaauw, et al.: "The structure of System/360". IBM Sys. J. 3, No. 2 119-164 (1964).

2. N. Wirth and C. A. R. Hoare, "A contribution to the development of Algol", Comm. ACM 9/6, 413-432 (June 1966).

3. N. Wirth and H. Weber, "Euler: A generalization of Algol, and its formal definition: Part I", Comm. ACM 9/1, 13-23 (Jan. 1966).

4. "IBM System/360 principles of operation", IBM Sys. Ref. Lib. A22-6821-2.

5. N. Wirth: "A programming language for the 360 computers", Tech. Report CS 33, Stanford U., Dec. 1965.