PL360
(REVISED)
A PROGRAMMING LANGUAGE FOR THE IBM 360

BY

MICHAEL A. MALCOLM

STAN-CS-71-215
MAY, 1971

COMPUTER SCIENCE DEPARTMENT

School of Humanities and Sciences

STANFORD UNIVERSITY

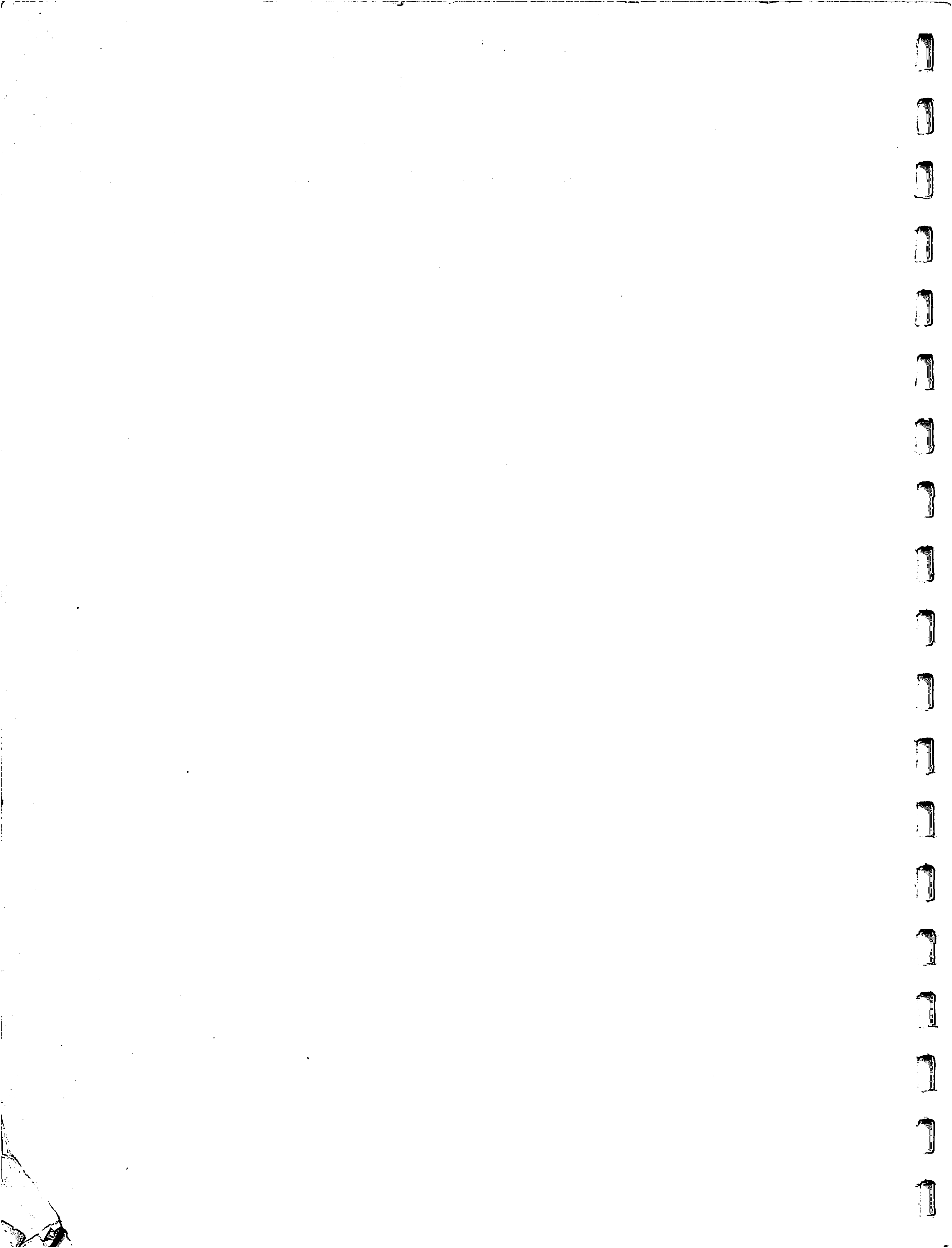
PL360

(REVISED)

A PROGRAMMING LANGUAGE FOR THE IBM 360

Michael A. Malcolm

This work has been supported in part by the National Science Foundation (Grant GJ 408), the Atomic Energy Commission AT(04-3) 326, PA # 30, the Office of Naval Research N00014-67-A-0112-0029 (NR 044-211), the Committee to Elect Stuart McLean to Congress, Lissner Computer Services (San Jose), and the Santa Clara Valley Democratic Coalition.



Abstract

In 1968, N. Wirth (Jan. JACM) published a formal description of PL360, a programming language designed specifically for the IBM 360. PL360 has an appearance similar to that of Algol, but it provides the facilities of a symbolic machine language. Since 1968, numerous extensions and modifications have been made to the PL360 compiler which was originally designed and implemented by N. Wirth and J. Wells. Interface and input-output subroutines have been written which allow the use of PL360 under OS, DOS, MTS and Orvyl.

A formal description of PL360 as it is presently implemented is given. The description of the language is followed by sections on the use of PL360 under various operating systems, namely OS, DOS and MTS. Instructions on how to use the PL360 compiler and PL360 programs in an interactive mode under the Orvyl time-sharing monitor are also included.

Keywords: Compilers
Computer Languages
IBM 360 Language Processors
Interactive Language Processors

Table of Contents

1. Introduction, Historical Background and Aims 1

2. Definition of the Language 3

  1. Terminology, Notation, and Basic Definitions 3

    1. The Processor 3
    2. Relationships 4
    3. The Program 4
    4. Syntax 5
    5. Syntactic Entities 6
    6. Basic Symbols 7

  2. Data Manipulation Facilities 8

    1. Identifiers 8
    2. Values 9
    3. Register Declarations 10
    4. Cell Declarations 11
    5. Cell Designators 12
    6. Register Assignments 13
    7. Cell Assignments 16
    8. Function Declarations 17
    9. Function Statements 19
    10. Synonym Declarations 19
    11. Segment Base Declarations 21

  3. Control Facilities 22

    1. If Statements 22
    2. Case Statements 24
    3. While Statements 25
    4. For Statements 25
    5. Blocks 26
    6. Goto Statements 27
    7. Procedure Declarations 27
    8. Procedure Statements 29

3. Examples 30

4. The Object Code 36

5. Addressing and Segmentation 42

   1. Program Segmentation 43
   2. Data Segmentation 44

6. The PL360 Compiler 45

   1. The Language 45

      1. Symbol Representation 45
      2. Standard Identifiers 46
      3. Restriction 46
      4. Standard Procedures 46

   2. Input Format 49
   3. Instructions to the Compiler 49
   4. Compiler Listing Output 51
   5. Error Messages of the Compiler 51
   6. Compiler Object Program Output 54
   7. Performance 55

7. Linkage Conventions 57

   1. Calling External Routines from PL360 57
   2. Requesting Supervisor Services 58
   3. Calling PL360 Procedures from External Routines 59

8. Operating System Interface and Configuration Requirements . . . 61

   1. Register Conventions 61
   2. Subroutine Specifications 62
   3. Linkage to the Compiler 63
   4. Configuration Requirements 64

9. Use as an OS Language Processor 65

   1. Data Set Requirements 65
   2. Processing Options 66
   3. Return Code 67
   4. JCL Statements 67
   5. Library 68

10. Use as a DOS Language Processor 69

1. System Configuration Requirements 69
2. Processing Options 70
3. JCL Statements 70
4. DTF Tables 72
5. Library 75

11. Use as an MTS Language Processor 76

1. Data Set Requirements 76
2. Processing Options 77
3. MTS Library 77
4. MTS Commands 77

12. Use as an ORVYL Language Processor 78

1. Using the PL360 Compiler with ORVYL 78
2. Input-Output Subroutines for Interactive PL360 Programs . . 80

13. The Run-Time Library 82

1. Number Conversion Procedures 82
2. Data Manipulation Procedures 84

14. Format of PL360 Programs 86

1. Indentation 86
2. Spacing 87
3. Choice of Identifiers 88
4. Comments 88
5. Miscellaneous 89

15. Acknowledgments 90

16. References 91

1. Introduction

PL360 is a programming language designed specifically for the IBM System/360 computers. It provides the facilities of a symbolic machine language but displays a structure similar to that of Algol. A formal description of an earlier version of the language has been published by Niklaus Wirth [1] who directed the development of the PL360 language and its compiler at the Computer Science Department of Stanford University. Although PL360 was originally designed for writing logically self-contained programs, subsequent extensions permit communication with separately compiled programs.

An efficient one pass "in core" compiler, written by Niklaus Wirth, Joseph W. Wells, Jr. and Edwin Satterthwaite, Jr., which incorporates these extensions is available through the IBM Contributed Program Library [2]. This compiler runs under the OS operating system and translates PL360 source code into object modules in the format required by several 360 operating systems (OS and MTS for example). The documentation issued with the compiler includes several amendments to the original language definition. Further extensions have recently been carried out at the University of Newcastle by James Eve. These changes [3,11] were aimed primarily at relaxing the linkage constraints on separately compiled programs, enabling for example direct communication with programs using OS/360 type linkages. The present author has made several modifications to the version of the compiler produced by James Eve. These extensions have made it possible to run the compiler and compiled programs under DOS operating systems. Assembly language subroutines have been written for both OS and DOS to facilitate input-output with sequential tape and disk files. With the aid of Dick Guertin of Stanford, the author has extended the syntax of PL360, primarily to increase programming convenience. We have recently written assembly language interfaces to allow interactive use of both the PL360 compiler and PL360 programs under the Orvyl time-sharing monitor at Stanford. These recent extensions made at Stanford have been documented in personal letters, memos or not at all.

The dispersed nature and inconvenient form of the PL360 documentation is an undoubted hindrance to more extensive use of this powerful and elegant tool. To remedy this, the language definition and compiler

description incorporating all changes are given in this manual. For a full discussion of the background underlying the development of PL360 and a description of the organization and development of the compiler together with some perceptive comments on the 360 Architecture, reference must still be made to [1], where the aims of the language are summarized:

"... 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 supervisor programs, and the programmer should be able to determine every detailed machine operation."

Knowledge of the 360 architecture [4, 5 or 6] is a prerequisite for understanding the language definition and some familiarity with the 360 Assembly Language [7] and Linkage Editor [8] is assumed in the description of the object code produced by the compiler.

In writing this manual, the author has drawn heavily upon the (anonymous) PL360 Programming Manual published by the University of Newcastle upon Tyne, Computing Laboratory [11].

2. Definition of the Language

2.1 Terminology, Notation, and Basic Definitions

The language is defined in terms of a 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.

2.1.1 The Processor

At any time, the state of the processor is described by a sequence of bits called the program status word (PSW). The status word contains, among other information, a pointer to the next instruction to be executed, and a quantity which is called the condition code.

Storage elements are classified into registers and core memory cells, simply called cells. Registers are divided into three types according to their size and the operations which can be performed on their values. The types 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 five 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 (a sequence of 8 bits = 1 byte),

b. short integer (a sequence of 16 bits = 2 bytes, interpreted as an integer in two's complement binary notation),

c. integer or logical (a sequence of 32 bits = 4 bytes, interpreted as an integer in two's complement binary notation),

d. real (a sequence of 32 bits = 4 bytes, to be interpreted as a base-16 floating-point number),

e. long real (a sequence of 64 bits = 8 bytes, to be interpreted as a base-16 floating-point number).

The types integer and logical are treated as equivalent in the language, and consequently only one of them, namely integer, will be mentioned throughout the report.

2.1.2 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 ordered set of the values of the component cells.

A set of relationships between values is defined by monadic and dyadic functions or operations, which the processor is 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 precisely defined in this report; instead, see [6].

2.1.3 The Program

A program contains declarations and statements. Declarations serve to list the cells, registers, procedures, and other quantities which are involved in the algorithm described by the program, and to associate names, so-called identifiers, with them. Statements specify the operations to be performed on these quantities, to which they refer through the use of 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:

a. Basic symbols constitute the basic *vocabulary* of the language (cf. 2.1.6). They are either single 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 comments have no effect on the execution of a program.

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

2.1.4 *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. 2.1.5) are denoted by English words enclosed in the brackets $\langle$ and $\rangle$. 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 $\mathcal{A}$, $\mathcal{K}$, and $\mathcal{T}$ are also used in the denotation of syntactic entities. They stand as abbreviations for any of the following words (or pairs):

$\mathcal{A}$	$\mathcal{K}$	$\mathcal{T}$
long real	long real	long real
real	real	real
integer	integer	integer
short integer		short integer
		byte

Syntactic rules are of the form $\langle A \rangle ::= \xi$ 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

$$\langle A \rangle ::= \xi_1 | \xi_2 | \cdots | \xi_n$$

is used as an abbreviation for the n syntactic rules

$$\langle A \rangle ::= \xi_1 \ , \ \langle A \rangle ::= \xi_2 \ , \ \ldots, \ \langle A \rangle ::= \xi_n \ .$$

If in the denotations of constituents of the rule the script letters $\mathcal{A}$, $\mathcal{K}$, or $\mathcal{T}$ occur more than once, they must be replaced consistently, or possibly according to further rules given in the accompanying text. As an example, the syntactic rule

⟨$\mathcal{K}$ register⟩ ::= ⟨$\mathcal{K}$ register identifier⟩

is an abbreviation for the set of rules

⟨long real register⟩ ::= ⟨long real register identifier⟩
⟨integer register⟩ ::= ⟨integer register identifier⟩
⟨real register⟩ ::= ⟨real register identifier⟩

2.1.5 Syntactic Entities

Syntactic Entity	Section	Syntactic Entity	Section
⟨$\mathcal{A}$ cell assignment⟩	2.2.7	⟨for statement⟩	2.3.4
⟨$\mathcal{A}$ number⟩	2.2.2	⟨format code⟩	2.2.8
⟨alternative condition⟩	2.3.1	⟨fractional number⟩	2.2.2
⟨arithmetic operator⟩	2.2.6	⟨function declaration⟩	2.2.8
⟨block body⟩	2.3.5	⟨function definition⟩	2.2.8
⟨block head⟩	2.3.5	⟨function identifier⟩	2.2.1
⟨block⟩	2.3.5	⟨function statement⟩	2.2.9
⟨case clause⟩	2.3.2	⟨goto statement⟩	2.3.6
⟨case sequence⟩	2.3.2	⟨hexadecimal digit⟩	2.2.2
⟨case statement⟩	2.3.2	⟨hexadecimal value⟩	2.2.2
⟨character sequence⟩	2.2.2	⟨identifier⟩	2.2.1
⟨combined condition⟩	2.3.1	⟨if clause⟩	2.3.1
⟨compound condition⟩	2.3.1	⟨if statement⟩	2.3.1
⟨condition⟩	2.3.1	⟨increment⟩	2.3.4
⟨digit⟩	2.2.2	⟨index⟩	2.2.5
⟨declaration⟩	2.3.5	⟨instruction code⟩	2.2.8
⟨fill value⟩	2.2.4	⟨integer value identifier⟩	2.2.1
⟨for clause⟩	2.3.4	⟨integer value synonym declaration⟩	2.2.10

Syntactic Entity	Section	Syntactic Entity	Section
⟨item⟩	2.2.4	⟨separate procedure heading⟩	2.3.7
⟨$\mathcal{K}$ primary⟩	2.2.6	⟨shift operator⟩	2.2.6
⟨$\mathcal{K}$ register assignment⟩	2.2.6	⟨simple $\mathcal{K}$ register assignment⟩	2.2.6
⟨$\mathcal{K}$ register synonym declaration⟩	2.2.10	⟨simple procedure heading⟩	2.3.7
⟨$\mathcal{K}$ register⟩	2.2.1	⟨simple statement⟩	2.3.5
⟨label definition⟩	2.3.5	⟨simple $\mathcal{T}$ type⟩	2.2.4
⟨letter⟩	2.2.1	⟨statement⟩	2.3.5
⟨limit⟩	2.3.4	⟨string⟩	2.2.2
⟨logical operator⟩	2.2.6	⟨synonymous cell⟩	2.2.10
⟨monadic operator⟩	2.2.6	⟨synonymous integer value⟩	2.2.10
⟨parameter⟩	2.2.9	⟨syn cell value⟩	2.2.10
⟨parameter list⟩	2.2.9	⟨$\mathcal{T}$ cell declaration⟩	2.2.4
⟨procedure declaration⟩	2.3.7	⟨$\mathcal{T}$ cell designator⟩	2.2.5
⟨procedure heading⟩	2.3.7	⟨$\mathcal{T}$ cell identifier⟩	2.2.1
⟨procedure identifier⟩	2.2.1	⟨$\mathcal{T}$ cell synonym declaration⟩	2.2.10
⟨procedure statement⟩	2.3.8	⟨$\mathcal{T}$ primary⟩	2.2.6
⟨program⟩	2.3.5	⟨$\mathcal{T}$ type⟩	2.2.4
⟨relation⟩	2.3.1	⟨$\mathcal{T}$ value⟩	2.2.6
⟨repetition list⟩	2.2.4	⟨true part⟩	2.3.1
⟨scale factor⟩	2.2.2	⟨unsigned $\mathcal{A}$ number⟩	2.2.2
⟨segment base declaration⟩	2.2.11	⟨while clause⟩	2.3.3
⟨segment base heading⟩	2.2.11	⟨while statement⟩	2.3.3
⟨segment close declaration⟩	2.2.11		

2.1.6 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|neg|shll|shrl|shla|shra|

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

begin|end|goto|comment|null|

integer|real|logical|byte|character|long|short|array|

function | procedure | register | syn | overflow |
segment | base | data | global | external | common | dummy | close |
equate

2.2 Data Manipulation Facilities

2.2.1 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⟩
⟨$\mathcal{K}$ register⟩ ::= ⟨identifier⟩
⟨$\mathcal{T}$ cell identifier⟩ ::= ⟨identifier⟩
⟨procedure identifier⟩ ::= ⟨identifier⟩
⟨function identifier⟩ ::= ⟨identifier⟩
⟨integer value identifier⟩ ::= ⟨identifier⟩

An identifier is a $\mathcal{K}$ register, $\mathcal{T}$ cell-, procedure-, function-, or integer value identifier, if it has respectively been associated with a $\mathcal{K}$ register, $\mathcal{T}$ cell, procedure, function, or integer value (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 with more than one quantity, then the considered occurrence designates the quantity to which it was associated in the innermost block embracing the considered occurrence. In any one block, an identifier must be associated with exactly one quantity. In the parse of a program, that association determines which of the rules given above applies.

Any processing computer and operating system 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.

2.2.2 Values

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

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

⟨unsigned short integer number⟩ ::= ⟨unsigned integer number⟩ S

⟨fractional number⟩ ::= ⟨integer number⟩ . ⟨digit⟩|
    ⟨fractional number⟩⟨digit⟩

⟨scale factor⟩ ::= ⟨integer number⟩

⟨unsigned real number⟩ ::= ⟨fractional number⟩|
    ⟨unsigned integer number⟩R|⟨fractional number⟩'⟨scale factor⟩|
    ⟨unsigned integer number⟩'⟨scale factor⟩

⟨unsigned long real number⟩ ::= ⟨fractional number⟩L|
    ⟨unsigned integer number⟩L|⟨fractional number⟩'⟨scale factor⟩L|
    ⟨unsigned integer number⟩'⟨scale factor⟩L

⟨*a* number⟩ ::= ⟨unsigned *a* number⟩|_⟨unsigned *a* number⟩

Integer, real, and long real numbers are represented in decimal notation. The latter two can be followed by a scale factor denoting an integral power of 10 . Short integers are distinguished from integers by the letter S following the number. In order to denote a negative number, an unsigned number is preceded by the symbol "_" .*/

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

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

A hexadecimal value denotes a sequence of bits. Each hexadecimal digit stands for a sequence of four bits defined as follows:

0 = 0000	4 = 0100	8 = 1000	C = 1100
1 = 0001	5 = 0101	9 = 1001	D = 1101
2 = 0010	6 = 0110	A = 1010	E = 1110
3 = 0011	7 = 0111	B = 1011	F = 1111

⟨string⟩ ::= "⟨character sequence⟩"

⟨character sequence⟩ ::= ⟨character⟩|⟨character sequence⟩⟨character⟩

---

*/ Note that the underline is used here. The minus sign (-) is used only as a dyadic operator — never as part of a number.

A string is a sequence of characters enclosed in quote marks. The set of characters depends on the implementation (cf. 6.1.1). If a quote mark (") is to be an element of the sequence, it is represented by a pair of consecutive quote marks.

Examples:	"ABC"	denotes the sequence	ABC
	"A""Z"	denotes the sequence	A"Z
	"""A"""	denotes the sequence	"A"

⟨byte value⟩ ::= "⟨character⟩"|⟨hexadecimal value⟩ X

⟨short integer value⟩ ::= ⟨short integer number⟩|⟨hexadecimal value⟩ S

⟨integer value⟩ ::= ⟨integer number⟩|⟨hexadecimal value⟩|
⟨integer value identifier⟩

⟨real value⟩ ::= ⟨real number⟩|⟨hexadecimal value⟩ R

⟨long real value⟩ ::= ⟨long real number⟩|⟨hexadecimal value⟩ L

Examples:

byte values	"B"	"?"	#1FX	
short integer values:	10S	#FF00S		
integer values:	0	#106C	_1	size
real values:	1.0	_3.146	2.7'8	#46000001R
long real values:	3.14159265359L		#4E00000000000001L	

Note: If hexadecimal values are used in conjunction with arithmetic operators in a program, they must be considered as the sequence of bits which constitutes the computer's representation of the number on which the operator is applied. Hexadecimal values followed by the letter R or L may be used to denote numbers in unnormalized floating-point representation [4,5,6].

2.2.3 Register Declarations

The System/360 processor has 16 registers which contain integer numbers and are said to be of type <u>integer</u> (or <u>logical</u>). They are designated by the following standard register identifiers (cf. 2.2.1):

R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15

The processor also has four registers which contain real numbers or long real numbers. If those registers are used in conjunction with real numbers, they are said to be of type <u>real</u>, and are designated by the standard register

identifiers

F0, F2, F4, F6 .

If they are used in conjunction with long real numbers, they are said to be of type long real, and are designated by the standard register identifiers

F01, F23, F45, F67 .

The above register identifiers are assumed to be predeclared, and no further register declarations can be made in a program; however other identifiers can be associated with these registers (cf. 2.2.10).

2.2.4 Cell Declarations

```
⟨simple byte type⟩ ::= byte|character
⟨simple short integer type⟩ ::= short integer
⟨simple integer type⟩ ::= integer|logical
⟨simple real type⟩ ::= real
⟨simple long real type⟩ ::= long real
⟨T type⟩ ::= ⟨simple T type⟩|array ⟨integer value⟩⟨simple T type⟩
⟨T cell declaration⟩ ::= ⟨T type⟩⟨item⟩|⟨T cell declaration⟩,⟨item⟩
⟨item⟩ ::= ⟨identifier⟩|⟨identifier⟩ = ⟨fill value⟩
⟨fill value⟩ ::= ⟨T value⟩|⟨string⟩|
     @⟨procedure identifier⟩|@@⟨procedure identifier⟩|
     @⟨T cell designator⟩|@@⟨T cell identifier⟩|
     ⟨repetition list⟩⟨fill value⟩)
⟨repetition list⟩ ::= (|⟨integer value⟩(|⟨repetition list⟩⟨fill value⟩,
```

A cell declaration introduces identifiers and associates them with cells of a specified type. The scope of validity of these cell identifiers is the block in whose heading the declaration occurs (cf. 2.3.5). Moreover, a declaration may specify the assignment of an initial value to the introduced cell. This assignment is understood to have occurred before execution of the program.

Cells may be initialized to numerical values, strings, relative or absolute addresses. The number of bytes appropriate for the type of the declared cell is taken for each (numerical) T value. Strings are never expanded or truncated, each character of the string occupies one byte,

initialization starting with the leftmost byte. A short integer or integer type cell can be initialized to the relative address (i.e., base register and displacement) corresponding to a 𝒯 cell identifier or to the relative (entry point) address corresponding to a procedure identifier by means of the @ operator. The @ operator also permits the initialization of an integer type cell with the relative address (i.e., index register, base register and displacement) of a 𝒯 cell designator. The @@ operator enables integer type cells to be initialized with absolute addresses corresponding to 𝒯 cell identifiers or the entry point of procedure identifiers.

If a simple type is preceded by the symbol array and an integer value, say n, then the declared cell is an array (ordered set) of n cells of the specified simple type. An initial value list with $m \leq n$ entries specifies the initial values of the first m elements of the array. A list may be specified as a list of sublists. Repetition of a sequence of elements may be specified by making the sequence into a list and preceding it by an integer value, say k, specifying the number of times the list is to be used. If no integer value precedes a list, it is used once. Absolute addresses may not occur in lists where $k > 1$. Integer values n and k must be positive.

Note. Boundary alignment is performed for a cell declaration (according to the simple type) and not for each initializing value; because strings are never expanded or truncated, care is needed in initializing with combinations of strings and other values.

Examples:

```
byte flag
short integer i,j,k = 10S,m = (5),baddr = @base
long real x,y,z = 27'3L
array 3 integer size = (36,23,37),paramlist = (@@a,@@b,@@errproc)
array 132 byte blank = 132(" "),buff = 33(" ",2("*")," ")
array fbsize logical area = fbsize(0)
```

2.2.5 Cell Designators

⟨𝒯 cell designator⟩ ::= ⟨𝒯 cell identifier⟩|
    ⟨𝒯 cell identifier⟩(⟨index⟩)

⟨index⟩ ::= ⟨integer value⟩|⟨integer register⟩|
    ⟨integer register⟩ + ⟨integer value⟩|
    ⟨integer register⟩ - ⟨integer value⟩|
    ⟨integer register⟩ + ⟨integer register⟩|
    ⟨integer register⟩ + ⟨integer register⟩ + ⟨integer value⟩|
    ⟨integer register⟩ + ⟨integer register⟩ - ⟨integer value⟩

Cells are denoted by cell designators. The designator for a particular cell consists of the identifier associated with that cell, optionally followed by an index. When an index is used, the address of the designated cell is taken as the address associated with the cell identifier plus the value of the index.

Notes: Register R0 must not be specified as an index constituent. Depending upon the function with which the cell designator is used and the declaration of the cell identifier, the index may have 0 , 1 or 2 integer register constituents. If the cell identifier has no base register associated with it, then the first integer register (if any) in the index is understood to be the base register. If the cell identifier has a base register associated with it, and the context permits an index register, then an integer register in the index is taken as an index register. If the identifier has no associated base register and the context permits indexing, then two integer registers may appear in the index and they are understood to be the base register and index register, respectively.

Examples:

age	B1(1)
size (8)	B14(R2)
price (R1)	MEM(R3 + R7 + 8)
line (R2 + 15)	buff(R1 + R4 - 2)

2.2.6 Register Assignments

⟨$\mathcal{T}$ primary⟩ ::= ⟨$\mathcal{T}$ value⟩|⟨$\mathcal{T}$ cell designator⟩
⟨$\mathcal{X}$ primary⟩ ::= ⟨$\mathcal{X}$ register⟩

A primary is either a value or the content of a designated cell or register.

⟨simple $\varkappa$ register assignment⟩ ::=
⟨$\varkappa$ register⟩ := ⟨α primary⟩|
⟨$\varkappa$ register⟩ := ⟨monadic operator⟩⟨α primary⟩|
⟨integer register⟩ := ⟨string⟩|
⟨integer register⟩ := @ ⟨τ cell designator⟩|
⟨integer register⟩ := @ ⟨procedure identifier⟩

A simple register assignment is said to specify the register appearing 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.

⟨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.

If a string is assigned to a register, that string must consist of not more than four characters. If it consists of fewer than four characters, null characters*/ are appended at the left of the string. The bit representation of characters is defined in [4,5,6] (EBCDIC).

The construction with the symbol @ is used to assign to the specified register the address of the designated cell or the entry point address of the procedure.

The legal combinations of types to be substituted respectively for the letters $\varkappa$ and α in preceding and subsequent rules of this paragraph are given in Table 1.

$\varkappa$	α
integer	integer
integer	short integer
real	real
long real	real
long real	long real

Table 1

*/ Null characters have the bit representation #00X .

Examples of simple register assignments:

R0 := i
R2 := R10
R6 := age
F0 := quant(R1)
R2 := "xyz"
F45 := neg F01
R13 := abs height

⟨χ register assignment⟩ ::= ⟨simple χ register assignment⟩|
⟨χ register assignment⟩⟨arithmetic operator⟩⟨σ primary⟩|
⟨integer register assignment⟩⟨logical operator⟩⟨integer primary⟩|
⟨integer register assignment⟩⟨shift operator⟩⟨integer value⟩|
⟨integer register assignment⟩⟨shift operator⟩⟨integer register⟩

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

A register assignment is said to specify the same register 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 of the primary 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 System/360. The operators ++ and -- denote logical or unnormalized addition and subtraction when applied to integer or real/long real registers respectively. When an integer value is specified following a shift operator, it must be nonnegative and less than 31 .

Examples of register assignments:

```
R0   :=  R3
R1   :=  10
R10  :=  i + age - R3 + size(8)
R9   :=  R8 and R7 shll 8 or R6
F2   :=  3.1416
F0   :=  quant(R1) * price(R1)
F45  :=  F45 + F01
```

Note: 1. 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 .
```

This single aspect of PL360 provides many pitfalls for beginners.

2. Multiplication and division with integer operands can only be specified with a multiplicand or dividend register Rn , where n is odd. The register Rm with m = n-1 is then used to hold the extension to the left of the product and dividend respectively. In the case of division, register Rm will be assigned the resulting remainder.

Examples:

R3 := x * y + z

R2 is affected by the multiplication.

R5 := B1/15

R4 is affected by the division and contains the remainder.

2.2.7 Cell Assignments

⟨α cell assignment⟩ ::= ⟨α cell designator⟩ := ⟨χ register⟩

The value of the χ register is assigned to the designated α cell. The allowable combinations of cell and register types α and χ are indicated in Table 1.

Examples of cell assignments: i := R0

price(R1) := F0

x := F67

2.2.8 Function Declarations

⟨format code⟩ ::= ⟨integer value⟩

⟨instruction code⟩ ::= ⟨integer value⟩

⟨function definition⟩ ::=

    ⟨identifier⟩(⟨format code⟩ , ⟨instruction code⟩)

⟨function declaration⟩ ::= <u>function</u> ⟨function definition⟩|

    ⟨function declaration⟩ , ⟨function definition⟩

There exist various data manipulation facilities in the 360 computer which cannot be expressed by an assignment. To make these facilities available in the language, the function statement is introduced (cf. 2.2.9), 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 instruction fields which correspond from left to right to the parameters given in function statements. The format code defines the format of the instruction according to Table 2. The last two bytes of the instruction code define the first two bytes of the instruction. The following example defines the standard function identifiers, which apart from TEST, SET and RESET, were chosen to be the symbolic machine codes used in [6].

```
function BALR(1,#0500),     MVI(4,#9200),      SRDL(9,#8C00),
         CLC(13,#D500),     MVN(5,#D100),      STC(12,#4200),
         CLI(4,#9500),      MVZ(5,#D300),      STH(12,#4000),
         CVB(12,#4F00),     NC(5,#D400),       STM(3,#9000),
         CVD(12,#4E00),     NI(4,#9400),       SVC(7,#0A00),
         ED(5,#DE00),       OC(5,#D600),       TEST(8,#95FF),
         EDMK(5,#DF00),     OI(4,#9600),       TM(4,#9100),
         EX(2,#4400),       PACK(10,#F200),    TR(5,#DC00),
         IC(2,#4300),       RESET(8,#9200),    TRT(5,#DD00),
         LA(2,#4100),       SET(8,#92FF),      TS(8,#9300),
         LH(12,#4800),      SLDA(9,#8F00),     UNPK(10,#F300),
         LM(3,#9800),       SLDL(9,#8D00),     XC(5,#D700),
         LTR(1,#1200),      SPM(6,#0400),      XI(4,#9700),
         MVC(5,#D200),      SRDA(9,#8E00)
```

Table 2

Format Code	Number of Parameter Fields	Instruction Fields (bit positions 0, 8, 16, 32, 48)
0	0	[ ]
1	2	[ ] [R] [R]
2	2	[ ] [R] [LC]
3	3	[ ] [R] [R] [C]
4	2	[ ] [ICS] [C]
5	3	[ ] [ICS] [C] [LC]
6	1	[ ] [R] [ ]
7	1	[ ] [ICS]
8	1	[ ] [ ] [C]
9	2	[ ] [R] [ ] [IC]
10	4	[ ] [I] [I] [C] [LC]
11	2	[ ] [R] [ICS]
12	2	[ ] [R] [C]
13	3	[ ] [ICS] [LC] [LC]
14	2	[ ] [C] [LC]
15	1	[ ] [LC]

Field Definition Codes:

R = χ register

C = $\mathcal{T}$ cell identifier (or designator in the 20 bit field) address

I = Integer value (The value is used directly

S = String in the instruction field)

L = $\mathcal{T}$ value or string or function designator. (The address of the value is used in the instruction field)

2.2.9 Function Statements

⟨parameter⟩ ::= ⟨𝒯 value⟩|⟨string⟩|⟨𝒦 register⟩|⟨𝒯 cell⟩|
    ⟨function designator⟩
⟨parameter list⟩ ::= ⟨parameter⟩|⟨parameter list⟩,⟨parameter⟩
⟨function statement⟩ ::= ⟨function identifier⟩|
    ⟨function identifier⟩(⟨parameter list⟩)

If a function designator is used as a parameter, the first function identifier must correspond to an execute instruction. That is, the first byte of the instruction code must have the value #44X . An example is the predeclared identifier EX (cf. 2.2.8).

Examples:

```
SET(flag)               STM(R0, R15, save)
RESET(flag)             MVI("*", line)
LA(R1, line)            IC(R0, flags(R1))
MVC(1, line, "hi")      EX(R1, MVC(0, line, buffer))
```

2.2.10 Synonym Declarations

⟨𝒯 cell synonym declaration⟩ ::=
    ⟨𝒯 type⟩⟨identifier⟩⟨synonymous cell⟩|
    ⟨𝒯 cell synonym declaration⟩,⟨identifier⟩⟨synonymous cell⟩
⟨synonymous cell⟩ ::= syn ⟨𝒯 cell designator⟩|syn⟨integer value⟩
⟨𝒦 register synonym declaration⟩ ::=
    ⟨simple 𝒦 type⟩ register ⟨identifier⟩ syn ⟨𝒦 register⟩|
    ⟨𝒦 register synonym declaration⟩, ⟨identifier⟩ syn ⟨𝒦 register⟩

Cell and register synonym declarations serve to associate synonymous identifiers with previously (i.e., preceding in the text) declared cells or registers. The types associated with the synonymous cell identifiers need not necessarily agree.

If a synonymous cell is specified by an integer value, then that integer value represents the displacement (and possibly the base register and index register) part of the cell's machine address.

```
Examples:    integer a16 syn a(16)
             array 32768 short integer memory syn 0
             integer timer syn #50
```

The following example defines the standard integer identifiers:

```
integer MEM syn 0,      B5 syn MEM(R5),    B10 syn MEM(R10),
        B1 syn MEM(R1), B6 syn MEM(R6),    B11 syn MEM(R11),
        B2 syn MEM(R2), B7 syn MEM(R7),    B12 syn MEM(R12),
        B3 syn MEM(R3), B8 syn MEM(R8),    B13 syn MEM(R13),
        B4 syn MEM(R4), B9 syn MEM(R9),    B14 syn MEM(R14),
                                           B15 syn MEM(R15)
```

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 = #4E00000000000000L
             integer xlow syn x(4)
```

A conversion operation from a number of type integer contained in register R0 to a number of type long real contained in register F01 can now be denoted by

```
xlow := R0; F01 := x
```

and a conversion vice-versa by

```
F01 := F01 ++ #4E00000000000000L; x := F01; R0 := xlow   .
```

No initialization can be achieved by a synonym declaration.

⟨integer value synonym declaration⟩ ::=
  equate ⟨identifier⟩⟨synonymous integer value⟩|
  ⟨integer value synonym declaration⟩,⟨identifier⟩⟨synonymous integer value⟩
⟨synonymous integer value⟩ ::= syn ⟨integer value⟩|
  syn ⟨syn cell value⟩|syn ⟨monadic operator⟩⟨integer value⟩|
  ⟨synonymous integer value⟩⟨arithmetic operator⟩⟨integer value⟩|
  ⟨synonymous integer value⟩⟨logical operator⟩⟨integer value⟩|
  ⟨synonymous integer value⟩⟨shift operator⟩⟨integer value⟩
⟨syn cell value⟩ ::= ⟨$\mathcal{T}$ cell designator⟩ - ⟨$\mathcal{T}$ cell designator⟩

Integer value synonym declarations serve to associate identifiers with integer values. These integer values are computed at the time the declaration is parsed and the identifiers thus associated can subsequently be used as integer values (cf. 2.2.1). When the difference of two cell designators is specified, the cell identifiers must both have the same base register (cf. 2.2.11); the difference between their relative locations within the segment is taken as the associated integer value. The cell designators must not use index registers. The scope of validity of these integer synonyms is the block in whose heading the declaration occurs (cf. 2.3.5).

Examples:

```
equate a syn 200, b syn a+8, c syn 4
equate d syn a/c and _4
array b byte x,y
equate e syn y-x, f syn e-c shll 2
```

Note: a = 200, b = 208, c = 4, d = 48, e = 208, f = 816.

2.2.11 Segment Base Declarations

⟨segment base heading⟩ ::= segment|global data ⟨identifier⟩|
    external data ⟨identifier⟩|common data ⟨identifier⟩|
    common|dummy

⟨segment base declaration⟩ ::=
    ⟨segment base heading⟩ base ⟨integer register⟩

⟨segment close declaration⟩ ::= close base

A segment base declaration causes the compiler to use the specified register as the base address for the cells subsequently declared in the block in which the base declaration occurs. Such use is terminated either by exit from the block or by the subsequent appearance of a segment close declaration. Upon entrance to this block, the appropriate base address is assigned to the specified base register unless the symbol dummy appears in the declaration (cf. 5.2).

If the symbol data is preceded by any of the symbols global, external or common, the corresponding identifier is associated with the data segment to enable linking of segments in different PL360 programs [8,9,12]. Appearance of the symbol sequence common base causes a blank identification

to be associated with the segment (cf. 6.6).

Note: Dummy base declarations permit the description of data areas which are created during the execution of the PL360 program. The specified base register must be some register other than RO [6], except in the case of a dummy base declaration. When RO is specified in a dummy base declaration, the subsequent identifiers are understood to have displacements and no base register (or index register).

2.3 Control Facilities

2.3.1 If Statements

⟨relation⟩ ::= = | ¬= | < | <= | >= | >

⟨condition⟩ ::= ⟨χ register⟩⟨relation⟩⟨α primary⟩|
    ⟨integer register⟩⟨relation⟩⟨string⟩|
    ⟨byte cell⟩|¬⟨byte cell⟩|⟨relation⟩|overflow

A condition is said to be met or not met. A condition consisting of a relation enclosed by a register and a primary is met, if and only if the specified relation holds between the current values of the register and the primary. When a relation is followed by a string, the string must consist of not more than four characters. If it consists of fewer than four characters, null characters are appended at the left of the string. In this case, the condition is met if and only if the specified relation holds between the current values of the register and the string (a logical comparison is used). A condition specified as a byte cell (or a byte cell preceded by ¬) is met, if and only if the value of the cell is #FF (or not #FF). A condition consisting of a relation or the symbol overflow is met, if the condition code of the processor (cf. 2.1.1) is in a state specified by Table 3.

symbol	state
=	0
¬ =	1 or 2
<	1
< =	0 or 1
> =	0 or 2
>	2
overflow	3

Table 3

⟨combined condition⟩ ::= ⟨condition⟩|
⟨combined condition⟩ and ⟨condition⟩
⟨alternative condition⟩ ::= ⟨condition⟩|
⟨alternative condition⟩ or ⟨condition⟩
⟨compound condition⟩ ::= ⟨combined condition⟩|
⟨alternative condition⟩

A compound condition is either of the form

c1 and c2 and c3 ... and cn

which is said to be met, if and only if all constituent conditions are met, or

c1 or c2 or c3 ... or cn

which is said to be met, if and only if at least one of the constituent conditions is met.

⟨if clause⟩ ::= if ⟨compound condition⟩ then
⟨true part⟩ ::= ⟨simple statement⟩ else
⟨if statement⟩ ::= ⟨if clause⟩⟨statement⟩|
⟨if clause⟩⟨true part⟩⟨statement⟩

The if statement specifies the conditional execution of statements:

1. ⟨if clause⟩⟨statement⟩

The statement is executed, if and only if the compound condition of the clause is met.

2. ⟨if clause⟩⟨true part⟩⟨statement⟩

The simple statement of the true part is executed and the statement is skipped, if and only if the compound condition of the if clause is met. Otherwise the true part is skipped and the statement is executed.

```
Examples:   if RO < 10 then R1 := 1
            if F2 > _3.75 and F2 < 3.75 then FO := F2 else FO := OR
            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 ...
```

2.3.2 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 RO) specified in the case clause. The statement whose ordinal number (starting with 1) 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 modified.

```
Example:    case R1 of
            begin comment interpretation of instruction code R1;
                 FO1 := FO1 + F23;
                 FO1 := FO1 - F23;
                 FO1 := FO1 * F23;
                 FO1 := FO1 / F23;
                 FO1 := neg FO1;
                 FO1 := abs FO1;
            end
```

2.3.3 While Statements

⟨while clause⟩ ::= while ⟨compound condition⟩ do
⟨while statement⟩ ::= ⟨while clause⟩⟨statement⟩

The while statement denotes the repeated execution of a statement as long as the compound condition in the while clause is met.

Examples:

```
while F0 < prize(R1) do R1 := R1 + 4
while R0 < 10 do
begin R0 := R0 + 1; F01 := F01 * F01; F23 := F23 * F01;
end
```

2.3.4 For Statements

⟨increment⟩ ::= ⟨integer value⟩
⟨limit⟩ ::= ⟨integer primary⟩|⟨short integer primary⟩
⟨for clause⟩ ::= for ⟨integer register assignment⟩ step ⟨increment⟩ until ⟨limit⟩ do
⟨for statement⟩ ::= ⟨for clause⟩⟨statement⟩

The for statement specifies the repeated execution of a statement, while the content of the integer register specified by the assignment in the for clause 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 clause is executed;
2. if the increment is not negative (negative), then if the value of the control register is not greater (not less) than the limit, 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
    begin F23 := quant(R2) * price (R2);
        F01 := F01 + F23;
    end
```

2.3.5 Blocks

```
⟨declaration⟩ ::= ⟨𝒯 cell declaration⟩|
     ⟨function declaration⟩|⟨procedure declaration⟩|
     ⟨𝒯 cell synonym declaration⟩|⟨𝒦 register synonym declaration⟩|
     ⟨integer value synonym declaration⟩|
     ⟨segment base declaration⟩|⟨segment close declaration⟩
⟨simple statement⟩ ::= ⟨𝒦 register assignment⟩|⟨𝒯 cell assignment⟩|
     ⟨function statement⟩|⟨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⟩ ::= ⟨statement⟩ . |
     global ⟨simple procedure heading⟩;⟨statement⟩ . |
     global ⟨simple procedure heading⟩ base ⟨integer register⟩;⟨statement⟩ .
```

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 optionally preceded by label definitions. 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 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 may be designated by the same identifier.

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

Example of a block:

```
        begin integer bucket;
            if flag then
            begin bucket := R0; R0 := R1; R1 := R2;
                R2 := bucket;
            end else
            begin bucket := R2; R2 := R1; R1 := R0;
                R0 := bucket;
            end;
            RESET(flag);
        end
```

2.3.6 Go To Statements

⟨go to statement⟩ ::= goto ⟨identifier⟩

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

1. Consider the innermost 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 innermost block surrounding it is considered. If this block is in the same program segment as the previous blocks, then step 2 is repeated; otherwise, the identifier is undefined (cf. 5.1).

2.3.7 Procedure Declarations

⟨simple procedure heading⟩ ::=
    procedure ⟨identifier⟩(⟨integer register⟩)

⟨separate procedure heading⟩ ::=
    segment ⟨simple procedure heading⟩|
    global ⟨simple procedure heading⟩|
    external ⟨simple procedure heading⟩

⟨procedure heading⟩ ::= ⟨simple procedure heading⟩|
    ⟨separate procedure heading⟩|
    ⟨separate procedure heading⟩ base ⟨integer register⟩
⟨procedure declaration⟩ ::= ⟨procedure heading⟩;⟨statement⟩

A procedure declaration serves to associate an identifier, which thereby becomes a procedure identifier, with a statement (cf. 2.3.5) which is called a procedure body. This identifier can then be used as an abbreviation for the procedure body anywhere within the scope of the declaration. When the procedure is invoked, the register specified in parentheses in the procedure heading is assigned the return address of the invoking procedure statement. This register must not be R0 .

If the symbol procedure is preceded by the symbol segment, global, or external, the procedure body is compiled as a separate program segment. If the symbol is global or external, the corresponding identifier is associated with the procedure segment to enable linking of segments in possibly different PL360 programs [8,9,12]. These symbols have no other influence on the meaning of the program with the exception of restricting the scope of goto statements (cf. 2.3.6, 5.1 and 6.6). If a base register is specified in the procedure heading, the procedure body is compiled using the specified register for the program segment base register (cf. 5.1); otherwise the current program base register is used (usually this is R15, however cf. 6.3). This register must not be R0. When the procedure is invoked, the specified (or assumed) base register is assigned the entry point address.

Examples:

```
procedure nextchar (R3);
begin if R5 < 71 then R5 := R5 + 1 else
    begin R0 := @ card; read; R5 := 0 ;
    end;
    IC(R0, card(R5));
end
```

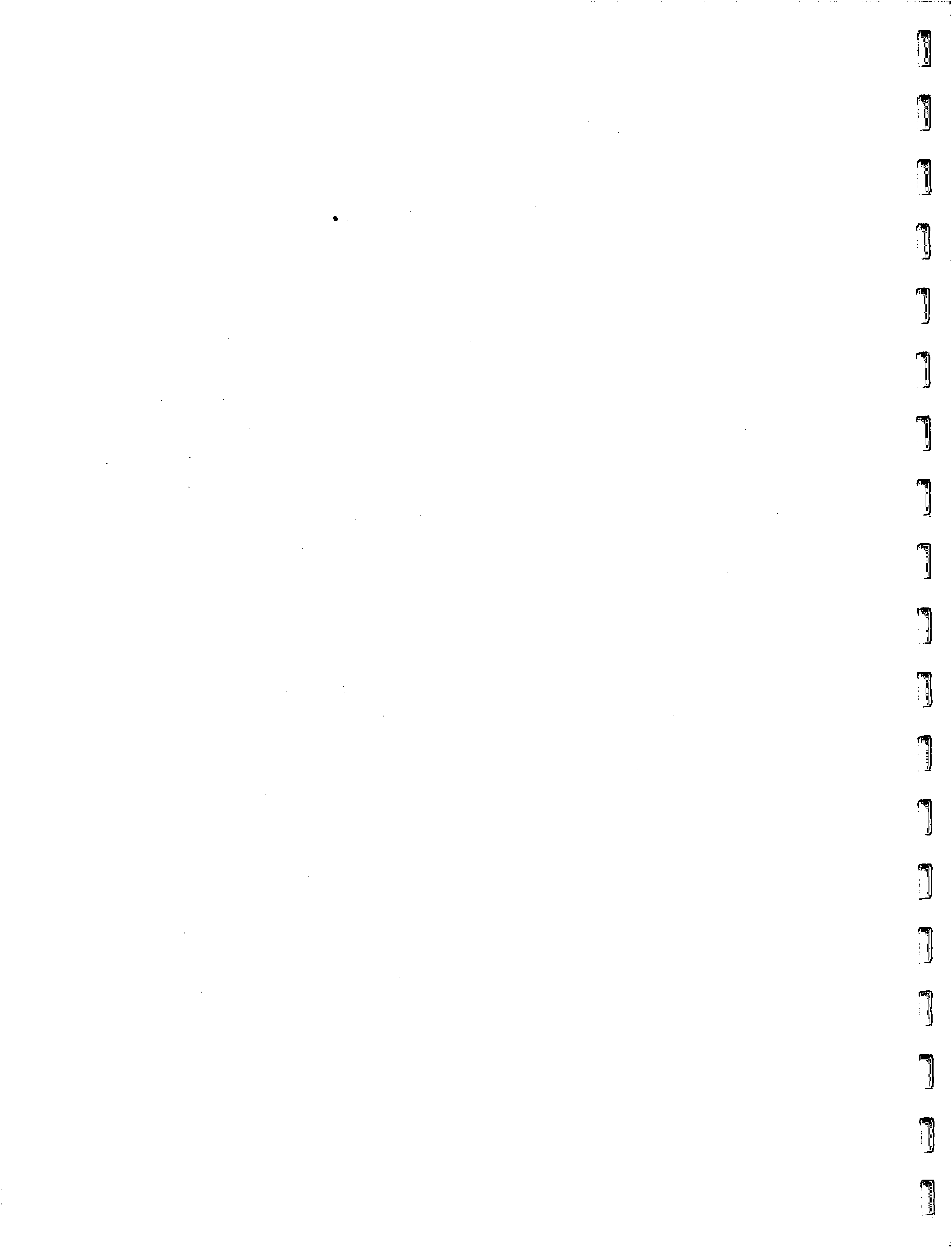
```
procedure slowsort (R4);
for R1 := 0 step 4 until n do
begin R0 := a(R1);
    for R2 := R1 + 4 step 4 until n do
    if R0 < a(R2) then begin R0 := a(R2); R3 := R2; end;
    R2 := a(R1); a(R1) := R0; a(R3) := R2;
end

external procedure searchdisk (R14) base R12; null;
```

Note: The code corresponding to a procedure body is terminated by a branch-on-register instruction specifying the register designated in the procedure heading. A procedure statement places a return address in this register when invoking the procedure. In order to return properly, the programmer must either not change the contents of that register, or explicitly save and restore its contents during the execution of the procedure.

2.3.8 Procedure Statements

⟨procedure statement⟩ ::= ⟨procedure identifier⟩|
    ⟨procedure identifier⟩(⟨integer register⟩)

The procedure statement invokes the execution of the procedure body designated by the procedure identifier. A return address is assigned to the register specified in the heading of the designated procedure declaration. If an integer register is specified in the procedure statement, on return from the procedure the contents of R15 is transferred to the specified integer register and the condition code is set by the transfer. This facilitates the convention of passing return codes in register R15.



3. 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
         R0 ... R6 are used, and register R0 initially contains the
         parameter n.  Algorithm 118 (Comm. ACM, Aug. 1962);
begin short integer nsqr;
    integer register n syn R0, i syn R1, j syn R2, x syn R3, ij syn R4,
    k syn R5;
    nsqr := n; R1 := n * nsqr; nsqr := R1;
    i := n + 1 shrl 1; j := n;
    for k := 1 step 1 until nsqr do
    begin x := i shll 6; ij := j shll 2 + x; x := X(ij);
        if x ≠ 0 then
        begin i := i - 1; j := j - 2;
            if i < 1 then i := i + n;
            if j < 1 then j := j + n;
            x := i shll 6; ij := j shll 2 + x;
        end;
        X(ij) := k;
        i := i + 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(R3)' is used to obtain the
         next character in sequence in register R0.  The answer appears in
         the long real register F01.  Registers R0 ... R4 and all real
         registers are used;
begin external procedure nextchar(R3); null;
integer register char syn R0, accum syn R1, scale syn R2; ext syn R3;
long real register answer syn F01;
byte sign, exposign;
long real converted = #4E00000000000000L;
integer convert syn converted (4);
nextchar; RESET(sign);
while char < "0" do
begin if char = "_" then SET(sign) else RESET(sign); nextchar;
end;
comment  Accumulate the integral part in accum;
accum := char and #F; nextchar;
while char >= "0" do
begin char := char and #F; accum := accum * 10S + char; nextchar;
end;
scale := 0;
convert := accum; answer := converted + 0L;
if char = "." then
begin comment  Process fraction.  Accumulate number in answer;
     nextchar;
     while char >= "0" do
     begin char := char and #F; convert := char;
          answer := answer * 10L + converted; scale := scale - 1;
          nextchar;
     end;
end;
if char = "'" then
begin comment  Read the scale factor and add it to scale;
     nextchar; if char = "-" then
```

```
            begin SET(exposign); nextchar;
            end else
            if char = "+" then
            begin RESET(exposign); nextchar;
            end else RESET(exposign);
        accum := char and #F; nextchar;
        while char >= "0" do
        begin char := char and #F; accum := accum * 10S + char; nextchar;
        end;
        if exposign then scale := scale - accum else scale := scale + accum;
    end;
    if scale ¬= 0 then
        begin comment  Compute F45 := 10 ↑ scale;
            if scale < 0 then
            begin scale := abs scale; SET(exposign);
            end else RESET(exposign);
            F23 := 10L;  F45 := 1L;  F67 := F45;
            while scale ¬ = 0 do
            begin SRDL(scale, 1);
                comment  divide scale by 2, shift remainder into scale
                extension, making ext < 0 if remainder is 1;
                F23 := F23 * F67; F67 := F23;
                if ext < 0 then F45 := F45 * F23;
            end;
            if exposign then answer := answer / F45
            else answer := answer * F45;
        end;
        if sign then answer := neg answer;
    end

    procedure Outreal (R4);
    begin comment  This procedure converts the (long) real number in register F01
        into a string of 14 characters which constitute one of its possible
        decimal denotations.  The character pattern is bsd.dddddd'sdd, where b
        is a blank, s a sign, and d a digit.  Registers R0, R2, R3, R4, and
        all real registers are used.  Upon entry, register R1 must contain the
        address of the output area.  Its value remains unchanged;
```

```
integer register exp syn R0, scale syn R2, ext syn R3;
long real register x syn F01;
long real convert;
integer converted syn convert (4), expo syn convert (0);
byte sign;
array 4 logical pattern =
     (#4021204B, #20202020, #20207D21, #20200000);
if x = 0L then MVC(13,B1,"  0          ") else
begin if x < 0L then SET(sign) else RESET(sign);
     x := abs x; convert := x;
     comment  Obtain an estimated decimal scale factor from the exponent
     part of the floating point representation;
     exp := expo shrl 24 - 64 * 307S; if < then exp := exp + 255;
     exp := exp shra 8 - 1; scale := abs exp;
     comment  compute F45 := 10 ↑ scale;
     F23 := 10L;  F45 := 1L;  F67 := F45;
     while scale ¬= 0 do
     begin SRDL(scale,1); F23 := F23 * F67; F67 := F23;
          if ext < 0 then F45 := F45 * F23;
     end;
     comment  normalize to 1 ≤ x < 10;
     if exp < 0 then
     begin x := x * F45;
          while x < 1L do
          begin x := x * 10L;  exp := exp - 1;
          end;
     end else
     begin x := x / F45;
          while x >= 10L do
          begin x := x *0.1L; exp := exp + 1;
          end;
     end;
     x := x * 1'7L ++ #4E00000000000005L;
     convert := x; ext := converted;
          comment  ext is used here to hold the integer resulting from the
          conversion;
```

```
            if ext >= 100000000 then
            begin ext := ext / 10; exp := exp + 1;
                comment  adjustment needed when conversion results in
                rounding up to 10.0.  Note that R2 = 0;
            end;
            MVC(13, B1, pattern); CVD(ext, convert); ED(9, B1, convert(3));
            if sign then MVI("-", B1(1));
            CVD(exp, convert); ED(3, B1(10), convert(6));
            if exp < 0 then MVI("-", B1(11)) else MVI("+", B1(11));
        end;
    end

    procedure  BinarySearch (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 N integer directory
               array N short integer code syn directory (0)
               array N short integer length syn directory (2)
               array N integer address syn directory (4)
               integer n
             are assumed.   n equals 8 times the number N of entries in the
             table, which appear as directory(8), directory(16), ...,
             directory(n).  This assumption can easily be changed by changing
             the value of size in the equate declaration.  It is assumed that
             code(0) = 0.  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 R1-R8 are used;
    begin integer register L syn R1, low syn R3, i syn R4, high syn R5,
        m syn R7;  equate size syn 8, mask syn neg size;
        high := n; low := size; comment  index step in directory is size;
        while low <= high do
        begin i := low + high shrl 1 and mask; R6 := address(i);
            if L = length(i) then
```

```
          begin EX(L, CLC(0, B2, B6)); if = then goto found;
               if < then high := i - size else low := size + i;
          end else
          if L < length(i) then
          begin EX(L, CLC(0, B2, B6));
               if <= then high := i - size else low := size + i;
          end else
          begin m := length(i); EX(m, CLC(0, B2, B6));
               if < then high := i - size else low := size + i;
          end;
     end;
     i := 0;
found: R3 := code(i);
end
```

4. The Object Code

Three principal postulates were used as guidelines in the design of the language:

1. Statements which express operations on data must correspond to machine instructions in an obvious way. Their structure must be such that they decompose into structural elements, each corresponding directly to a single instruction.
2. No storage element of the computer should be hidden from the programmer. In particular, the usage of registers should be explicitly expressed by each program.
3. 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).

The following paragraphs serve to exhibit the machine code into which the various constructs of the language are translated. The mnemonics of the 360 Assembly Language [7] are used to denote the individual instructions. The notation {A} serves to denote the code sequence corresponding to the construct ⟨A⟩. It is assumed that R15 is the program base register (cf. 5.1, 6.3).

1. ⟨$\mathcal{X}$ register⟩ := ⟨α primary⟩

The code consists of a single load instruction depending on the types of register and primary (cf. Table 4, column 1).

2. ⟨$\mathcal{X}$ register assignment⟩⟨operator⟩⟨α primary⟩

The code consists of a single instruction depending on the operator and the types of register and primary. It is determined according to Table 4, columns 2-7.

3. ⟨α cell⟩ := ⟨κ register⟩

The code consists of a single store instruction depending on the types of cell and register as indicated by Table 4, column 8.

4. if ⟨condition-1⟩ and ... and ⟨condition-n-1⟩ and ⟨condition-n⟩ then ⟨simple statement⟩ else ⟨statement⟩

```
        {condition-1}
        BC c1,L1
          ...
        {condition-n-1}
        BC cn-1,L1
        {condition-n}
        BC cn,L1
        {simple statement}
        B L2
L1      {statement}
L2
```

c_i is determined by the i-th condition, which itself either translates into a compare instruction depending on the types of compared register and primary (cf. Table 4, col. 9), or has no corresponding instruction, if it merely designates condition code states.

Example: if R1 < R2 then R0 := R3 else R0 := R4

```
        CR 1,2
        BC 10, L1
          ...
        LR 0,3
        B L2
L1      LR 0,4
L2
```

Operands		Operators								
$\mathcal{K}$ register (type)	a primary (type)	1 :=	2 +	3 -	4 *	5 /	6 ++	7 --	8 :=	9
integer	integer register	LR	AR	SR	MR	DR	ALR	SLR		CR
integer	integer cell	L	A	S	M	D	AL	SL	ST	C
integer	short integer cell	LH	AH	SH	MH				STH	CH
real	real register	LER	AER	SER	MER	DER	AUR	SUR		CER
real	real cell	LE	AE	SE	ME	DE	AU	SU	STE	CE
long real	real register	LER	AER	SER	MER	DER	AUR	SUR		CER
long real	long real register	LDR	ADR	SDR	MDR	DDR	AWR	SWR		CDR
long real	real cell	LE	AE	SE	ME	DE	AU	SU	STE	CE
long real	long real cell	LD	AD	SD	MD	DD	AW	SW	STD	CD

Table 4

5. <u>if</u> ⟨condition-1⟩ <u>or</u> ... <u>or</u> ⟨condition-n-1⟩ <u>or</u> ⟨condition-n⟩ <u>then</u> ⟨simple statement⟩ <u>else</u> ⟨statement⟩

```
        {condition-1}
        BC c1,L1
          ...
        {condition-n-1}
        BC cn-1,L1
        {condition-n}
        BC cn,L2
L1      {simple statement}
        B L3
L2      {statement}
L3
```

6. <u>case</u> ⟨integer register-m⟩ <u>of</u>
<u>begin</u> ⟨statement-1⟩;
⟨statement-2⟩;
...
⟨statement-n⟩;
<u>end</u>

```
      AR    m,m
      LH    m,SW(m)
      B     0(m,15)
L1    EQU   *-ORIGIN
      {statement-1}
      B     LX(15,0)
L2    EQU   *-ORIGIN
      {statement-2}
      B     LX(15,0)
      :     :
Ln    EQU   *-ORIGIN
      {statement-n}
      B     LX(15,0)
```

```
SW    EQU   *-2
      DC    Y(L1)
      DC    Y(L2)
      :     :
      DC    Y(Ln)
LX    EQU   *-ORIGIN
```

ORIGIN is the address of the beginning of the program segment and register 15 is assumed to contain this address (cf. 5.1).

7. <u>while</u> ⟨condition⟩ <u>do</u> ⟨statement⟩

```
L1    {condition}
      BC    c,L2
      {statement}
      B     L1
L2
```

If the condition is compound, then code sequences similar to those given under 4 and 5 are used.

8. <u>for</u> ⟨integer register assignment⟩ <u>step</u> ⟨increment⟩ <u>until</u> ⟨limit⟩ <u>do</u> ⟨statement⟩

```
      {integer register assignment}
      B     L2
L1    {statement}
      A     m,INC
L2    C     m,LIM
      BC    c,L1
```

Rm is the register specified by the assignment, INC the location where the increment is stored, and LIM the location where the limit is stored. The compare instruction at L2 may be either a C , CH , or CR instruction depending on the type of limit. Moreover, c depends on the sign of the increment.

9. <u>procedure</u> ⟨identifier⟩(⟨integer register⟩);⟨statement⟩

```
        P       {statement}
                BR      m
```

It is assumed that the integer register enclosed in parentheses is Rm .

10.

```
                ⟨procedure identifier⟩
                BAL     m,P

        or      L       15,newbase
                BALR    m,15
                L       15,oldbase
```

It is here assumed that P designates the procedure to be called, and Rm is the return address register specified in its declaration. The first version of code is obtained whenever the segment in which the procedure is declared is also the one in which it is invoked. If the procedure call is of the form

⟨procedure identifier⟩(Rn)

then the instruction sequences become

```
        BAL     m,P
        LTR     n,15
        BALR    15,0
        L       15,oldbase

or      L       15,newbase
        BALR    m,15
        LTR     n,15
        BALR    15,0
        L       15,oldbase
```

5. 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 is loaded with the appropriate base address whenever an instruction whose address refers to it is executed,
3. the difference d between the desired absolute address and the available base address satisfies

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

This scheme not only increases the amount of 'clerical' work in programming, but also constitutes a rich source of pitfalls. PL360 is designed to ease the tedious task of base address assignment, and to provide checking facilities against errors.

The solution adopted here is that of program segmentation. The program is subdivided into individual parts, called segments. Every quantity defined within the program is known by the number of the segment in which it occurs and by its displacement relative to the origin of that segment. The problem then consists of subdividing the program and choosing base registers in such a way that

a. the compiler knows which register is used as base for each compiled address,

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.

It was decided [1] that the programmer should express explicitly which parts of his program were to constitute segments. He has then the possibility of organizing the program in a way which minimizes the number of cross-references between segments.

It should be noted that the programmer's knowledge about segment sizes and occurrences of cross-references 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 sometimes unaware of the occurrence of branch instructions implicit in certain constructs of the language. It was therefore decided [1] to treat programs and data differently, and this decision also conformed with the desirability of keeping program and data apart as separate entities.

5.1 Program Segmentation

Since control lies by its very nature in exactly one segment at any instant, one fixed register is designated to hold the base address of the program segment currently under execution. Register R15 is usually used for this purpose, (however, cf. 2.3.7, 6.3).

Branching to another segment is accomplished with a procedure statement which causes R15 to be reloaded with the base address of the destination segment before branching to that segment.

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. An explicit go to statement cannot be used for branching to another segment. 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. Only relatively large procedure bodies should constitute segments. A facility is provided to designate such procedures explicitly: A procedure to be compiled as a <u>program segment</u> must contain the symbol <u>segment</u> or <u>global</u> in its heading. It is relatively easy for a programmer to guess which procedure exceeds the prescribed size, or otherwise to insert the symbol <u>segment</u> after the compiler has provided an appropriate comment in the first compilation attempt. Obviously, the outermost block is always compiled as a segment.

5.2 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, an instruction is compiled which loads the register with the appropriate segment address. However, its previous contents are neither saved nor restored upon exit from the block.

A PL360 program which is a statement is considered to be embedded in a block containing the implicit declarations

global data SEGN000 base R13;
array 18 integer savearea;

However, the identifier "savearea" is not considered predeclared. The 18-word "savearea" is merely reserved to conform with procedure calling conventions (cf. 6.1.4). If the PL360 program is a global procedure, there is no implicit base declaration.

Obviously, data segments declared in parallel (i.e., not nested) blocks, can safely refer to the same base register. Data segments declared within the same block usually refer to different base registers. Data segments declared within nested blocks should normally refer to different base registers. If they do not, it is the programmer's responsibility to ensure that the register is appropriately loaded when a segment is addressed.

There is no limit to the size of data segments. All cell identifiers must, however, refer to cells whose addresses differ from the segment base address by less than 4096. If they do not, the compiler provides an appropriate indication.

6. The PL360 Compiler

The PL360 compiler is itself written in PL360. The current version of the compiler is neither re-entrant nor serially reusable. This in no way inhibits the writing of PL360 programs with these attributes.

6.1 The Language

The PL360 programming language is described in Section 2 of this document. Details pertinent to the present implementation (e.g., symbol representations, standard identifiers, and specific limitations) are contained in subsequent paragraphs of this section.

6.1.1 Symbol Representation

Only capital letters are available. Basic symbols which consist of underlined letter sequences in Section 2 are denoted by the same letter sequences without further distinction. As a consequence, they cannot be used as identifiers. Such letter sequences are called reserved words. Embedded blanks are not allowed in reserved words, identifiers, and numbers. Adjacent reserved words, identifiers, and numbers must be separated by at least one blank. Otherwise, blanks may be used freely. The basic symbols are:

```
+  -  *  /  (  )  =  <  >  ¬
,  ;  .  :  @  #  _  "  '
:=
DO  IF  OF  OR
ABS  AND  END  FOR  NEG  SYN  XOR
BASE  BYTE  CASE  DATA  ELSE  GOTO  LONG  NULL
REAL  SHLA  SHLL  SHRA  SHRL  STEP  THEN
ARRAY  BEGIN  CLOSE  DUMMY  SHORT  UNTIL  WHILE
COMMON  EQUATE  GLOBAL
COMMENT  INTEGER  LOGICAL  SEGMENT
EXTERNAL  FUNCTION  OVERFLOW  REGISTER
CHARACTER  PROCEDURE
```

6.1.2 Standard Identifiers

The following identifiers are predeclared in the language but may be redeclared due to block structure. Their predefined meaning is specified in Section 2 or in Section 6.1.4.

MEM

B1 B2 B3 B4 B5 B6 B7 B8 B9 B10 B11 B12 B13 B14 B15

R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15

F0 F2 F4 F6

F01 F23 F45 F67

BALR CLC CLI CVB CVD ED EDMK EX IC

LA LH LM LTR MVC MVI MVN MVZ NC NI OC OI PACK

RESET SET SLDA SLDL SPM SRDA SRDL STC STH STM SVC

TEST TM TR TRT TS UNPK XC XI

CANCEL GET KLOSE OPEN PAGE PRINT PUNCH PUT READ WRITE

6.1.3 Restriction

The implementation imposes the restriction that only the first 10 characters of identifiers are recognized as significant.

6.1.4 Standard Procedures

A set of standard procedures is defined for elementary unit record input and ouput operations. The implicit procedure declarations are as follows:

external procedure READ (R14); null;
external procedure WRITE (R14); null;
external procedure PAGE (R14); null;
external procedure PUNCH (R14); null;
external procedure PRINT (R14); null;

Suitable externally compiled or assembled routines must be provided in the link/loading process; the specifications of these routines are:

READ — Read an 80 character record from the system input data set and assign that record to the memory area designated by the address in register R0. Set the condition code to 2 if no record could be returned due to an end of file condition; otherwise, to 0.

WRITE — Write a 133 character record to the system listing data set. A 132 character record is taken from the memory area designated by the address in register R0 and prefixed by an appropriate carriage control character. A control character indicating a new page is used after 60 lines have been written on a page, otherwise a control character indicating the next line is used. The first line is written on a new page.

PAGE — Cause the next output record transmitted by a WRITE to the system listing data set to have a control character indicating a new page.

PUNCH — Write the 80 character record designated by the address in register R0 to the system punch data set.

PRINT — Write the 133 character record designated by the address in register R0 to the system listing data set. The calling program provides a USASI control character as the first character.

All of these procedures assume that register R13 contains the address of an 18 word save area and all registers are restored before return. Each of the data sets is opened upon initial reference and is closed by the operating system at the end of a job step.

A set of standard procedures is defined for elementary disk and tape input and output operations using sequential files. The implicit procedure declarations are as follows:

```
external procedure OPEN(R14); null;
external procedure GET(R14); null;
external procedure PUT(R14); null;
external procedure KLOSE(R14); null;
```

Suitable externally compiled or assembled routines must be provided in the link/loading process; the specifications of these routines are:

OPEN — At entry, register R0 must be 0 if the file is to be an output file or 1 if the file is to be an input file. Register R2 must contain the address of an 8-byte area containing a unique file name. (This is taken as the ddname in an OS environment and as the symbolic file name in a DOS environment.) In an OS environment, register R1 must contain the address of a 100-byte full word-aligned area which, following the open, will contain the data control block. In a DOS environment, register R1 must contain the address of a separately assembled DTF table which describes the file. The file is made ready for input/output operations. All registers are restored.

GET — At entry, register R1 must contain the address of a table which describes the file. (In an OS environment this table is called the data control block and in a DOS environment it is called the DTF table.) Upon return, register R1 contains the address of the next logical record in the file. (The first call of GET returns with the address of the first logical record.) When an end-of-file is reached, the condition code is set to 2; normally it is set to 0. All registers, except R1, are restored.

PUT — At entry, register R1 must contain the address of a table which describes the file. Upon return, register R1 contains the address of an area in which the next logical record to be output is to be built. All other registers are restored.

KLOSE — At entry, register R1 must contain the address of a table which describes the file. The corresponding file is closed and no further input-output operations can be performed with it unless it is opened again. In an OS environment, the contents of register R0 denoted by (R0) is also an input parameter to this subroutine: If $(R0) = 0$, the DISP option of the DD statement is used to determine final volume positioning; if $(R0) \leq 0$, the volume is positioned at the end of the data set. If $(R0) > 0$, the volume is positioned at the beginning of the data set. All registers are restored.

One additional standard procedure is defined for ease in communicating with the operating system. The implicit procedure declaration is as follows:

external procedure CANCEL(R14); null;

A suitable externally compiled or assembled routine must be provided in the link/loading process; the specification of this routine is:

CANCEL The job, including all future job steps, is cancelled.

All of these procedures assume that register R13 contains the address of an 18 word save area (cf. 5.2).

6.2 Input Format

Compiler input records consist of 80 characters. The first 72 characters of each record are processed as part of a PL360 program; characters 73 through 80 are listed but not otherwise processed. Character 72 of one record is considered to be immediately followed by character 1 of the next record. Strings and comments should be arranged so that the character '$' does not appear in character position 1.

6.3 Instructions to the Compiler

The compiler accepts instructions inserted anywhere in the sequence of input records. These instructions affect subsequent records. A compiler instruction record is marked by the character '$' in column 1 and an instruction in columns 2-9.

$NOGO Compile, but mark the output non-executable.

$LIST List source records (initial option)

$NOLIST Do not list source records.

$PAGE Start a new page with the next listing record.

$TITLE Start a new page with the next listing record, and use the contents of columns 10 through 62 as the title for that and subsequent pages.

$XYYY# If this directive precedes the first source record then compiler generated segment names will commence with XYY rather than SEG, otherwise the directive is ignored. X signifies any alphabetic and Y any alphanumeric character: (cf. 6.6).

$OS Subsequent PL360 programs which are statements are compiled with entry and exit instruction sequences which conform with the program calling conventions of an OS environment. This is a default option when the compiler is used with the OS interface.

$DOS Subsequent PL360 programs which are statements are compiled with entry and exit instruction sequences which conform with the program calling conventions of a DOS environment. This is a default option when the compiler is used with the DOS interface.

$XREF All subsequent instances of identifiers are listed in an alphabetical cross-reference listing together with the line numbers at which they are referenced in the source program. The cross-reference listing follows the program listing. If there is not enough free storage to allocate to the cross-reference tables, the cross-reference listing is not made and the $XREF instruction is ignored.

$NOXREF This causes the previous option to be turned off (initial option).

$0 Print a summary line at the close of each segment (initial option).

$1 Print a summary line and list of external symbol dictionary entries at the close of each segment.

$2 List the address of each variable and procedure as it is declared, as well as the information specified in $1.

$3 List the object text in hexadecimal notation at the close of each segment, as well as the information specified in $2.

$BASE=xx New program segments following this instruction are compiled with xx taken as the program segment base register. This includes main programs, global procedures, segment procedures, and external procedures. Procedure calls to such segments automatically set the specified base register to the entry point address. The decimal number xx must be between 01 and 15 . Programs which are statements must not be compiled with base registers 13 or 14. The initial option is xx=15 . It is recommended that this compiler instruction only be used for programs which make use of SVC instructions which do not preserve the contents of register R15.

6.4 Compiler Listing Output

If listing is specified, each source record is listed as it is read. Source records in which errors are detected are always listed. Four sets of numbers appear at the left of each line. The first set consists of the current internal program segment number (in decimal) followed by the program object code relative address (in hexadecimal); the second set, of the current internal data segment number and data relative address. The fifth number is the number of the source record (excluding compiler instructions). The final number, the begin/end level count, shows the excess of begin symbols over end symbols at the beginning of the line on which it appears; it is only printed after lines which cause its value to change.

In addition, each page begins with a heading which includes the page number, date, time, and an optional title (cf. 6.3).

6.5 Error Messages of the Compiler

Errors detected by the compiler are indicated by a message and a vertical bar below the character which was last read. After 51 errors are detected in a program, a message is provided, and further diagnostic messages are counted but not listed. Following is a list of error diagnostics and their meanings:

Error No.	Message	Meaning
00	SYNTAX	The source program violates the PL360 syntax. Analysis continues with the next statement.
01	VAR ASS TYPES	The types of operands in a variable assignment are incompatible.
02	FOR PARAMETER	In a for clause, the register is not an integer register, the step is not an integer or short integer number, or the limit is not a register, cell, or number of the integer types.
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 or number is specified in a shift operation.
06	COMPARE TYPES	The types of operands in a comparison are incompatible.
07	REG TYPE OR #	Either the type or the number of the register used is incorrect.
08	UNDEFINED ID	An undeclared identifier is encountered. The identifier is treated as if it were 'R1'. This may generate other errors.
09	MULT LAB DEF	The same identifier is defined as a label more than once in the same block.
10	EXC INI VALUE	The number of initializing values exceeds the number of elements in the array.
11	NOT INDEXABLE	An index register is not allowed for the cell designator in this context.
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. This occurrence of the identifier is ignored.
16	PROGRAM OFLOW	The current program segment is too large. It must be resegmented.
17	INITIAL OFLOW	The area of initializing data in the compiler is full. This can usually be circumvented by suitable data segmentation or by re-ordering initialized data within the segment.
18	ADDRESS OFLOW	The number used as index is such that the resulting relative address is less than 0 or greater than 4095.
19	NUMBER OFLOW	The integer number is too large in magnitude.
20	MISSING .	An end-of-file is encountered before a '.' terminating the program. The problem may be a missing string quote.
21	STRING LENGTH	The length of a string is either 0 or greater than 256.
22	AND/OR MIX	A compound condition must not contain both ANDs and ORs.
23	FUNC DEF NO.	The format number in a function declaration is illegal.
24	ILLEGAL PARAM	A parameter is incompatible with the specifications of the function.
25	NUMBER	A number has been used that has an illegal type or value.
26	SYN MIX	Synonym declarations cannot mix cell and register declarations, or J cell designators have different base registers.
27	SEG NO OFLOW	The maximum allowed segment number has been exceeded. The limit is generally set at 75.
28	ILLEGAL CLOSE	A segment close declaration is encountered when no data segment is open in the corresponding block head.

29	NO DATA SEG	A variable is declared with no open data segment. A dummy data segment is opened.
30	ILLEGAL INIT	Initialization is specified in a common data segment or replicates an absolute address.

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

6.6 Compiler Object Program Output

The PL360 compiler is designed to be used in conjunction with link/loader programs which resolve symbolic cross-references between the segments of one or more programs. Examples of programs capable of such resolution are the MTS loader [9], the IBM OS linkage editor or loader [8], and the IBM DOS linkage editor [12]. The remainder of this section uses the terminology of these programs.

The output of the PL360 compiler is a sequence of object modules. Each object module contains a single control section corresponding to a PL360 segment. It consists of 80 character records in the standard format of external symbol dictionary (ESD), text (TXT), relocation dictionary (RLD) and an end (END) (cf. [10] Appendix B).

Every PL360 segment (except a dummy data segment) is associated with an object module in the following fashion:

1. If the symbol segment appears in the segment declaration, an object module is produced for this segment; the control section name is generated by the compiler as described below.
2. If the symbol global appears in the segment declaration, an object module is produced for this segment; the control section name is the first 8 bytes of the identifier appearing in the declaration.
3. If the symbol external occurs in the segment declaration, no object module is produced; instead the first 8 bytes of the identifier in the declaration is assumed to be the name of a control section independently generated and is used to indicate this in the object module created for the segment containing the external declaration.

4. If the symbol common appears in the segment declaration then an object module is created in the form of a labelled or blank common control section according to whether the common declaration contains an identifier or not.

In all cases a control section has a single entry point; the entry point name and the control section name are identical. In the case of a PL360 program which is a statement, a transfer address to the entry point is provided in the END card of the object module for the implicit segment corresponding to this statement. This transfer address is used by a loader to determine where to begin execution.

The task of the linkage editor/loader includes matching global and external declarations, inserting absolute address constants and completing tables of segment base addresses, contained within each control section for a program segment, in accordance with the external symbol dictionary and relocation dictionary generated by the compiler for that control section.

For PL360 programs which are statements, control section names generated by the compiler are of the form SEGNnnn where nnn is the decimal internal segment number. If the PL360 program is a global procedure, the first three characters of the procedure identifier (extended on the right by NN if necessary) are used in place of the characters 'SEG'. These naming conventions may be overruled by use of the compiler directive $XYY# (cf. 6.3).

Each END card of the object module output of the compiler has the name "PL360" followed by the date and time of compilation.

6.7 Performance

In an OS environment on a 360/67 with spooled input and output files, the compiler will recompile itself in about 25 seconds. The compiler is approximately 2700 card images. Thus, when the OS scheduler time is subtracted from the execution time given above, it is seen that the compiler runs at a speed in excess of 100 cards per second (for dense code).

In a DOS environment on a 360/30, the compiler is limited only by the speed of the card reader. The compiler has successfully recompiled itself on a 64K 360/30 at a rate of 1200 cards per minute (the speed of the card

reader). This is impressive when compared to the time required for the DOS Assembler to assemble the interface module which consists of under 250 cards. When the macro instructions are expanded, the DOS interface has 972 card images and the Assembler takes 15 minutes for the assembly.

7. Linkage Conventions

Although PL360 was designed for writing logically self-contained programs, it is possible to communicate with separately compiled programs if appropriate linkage and coding conventions are observed. These conventions are summarized below.

7.1 Calling External Routines from PL360

Addresses which correspond to external symbolic names and which are to be supplied by linkage editing can be specified by the external or common declarations of PL360. Entry to the block containing a data segment declaration causes the specified base register to be loaded with the corresponding address. External names appearing in procedure declarations are assumed to designate entry points to subroutines. In such declarations, the procedure body is normally the statement <u>null</u>. The call of the external procedure P2 from the procedure P1 is equivalent to the following 360 Assembler coding:

```
USING P1,15
...
L       ℓ,=V(P2)
DROP    15
BALR    n,ℓ
USING   *,n
L       15,=A(P1)
USING   P1,15
DROP    n
```

This linkage implies the following restrictions upon the called routine:

1. At entry, the base register specified (or assumed) in the external procedure declaration (ℓ) contains the address of the entry point, unless $\ell = n$.
2. At entry, the register specified in the external procedure declaration (n) contains the return address.
3. Before return, the return address must be restored to that designated register.

Any additional, non-conflicting conventions may be established by the programmer.

If the called procedure (P2) uses R15 to return information to the calling routine (P1), the procedure statement in P1 is usually of the form P2(Rm) , indicating that the return linkage must move the contents of R15 to Rm , thus setting the condition code before re-establishing the base address of P1 in R15. The equivalent 360 Assembler coding for this type of call differs from that already given only in the last four lines which become

```
LTR     m,15
BALR    15,0
USING   *,15
L       15,=A(P1)
USING   P1,15
```

OS type linkages are facilitated by the fact that if the calling PL360 program is a statement, the first 18 words of the implicit data segment (base register R13) are available for use as a save area (cf. 5.2), and by the @@ operator which facilitates the construction of OS-type parameter lists at compile time.

7.2 Requesting Supervisor Services

SVC instructions are available in PL360 programs through the function statement. It should be noted, however, that in many operating systems the contents of R15 are destroyed by execution of some SVC instructions. In such cases, it is essential that saving and immediately restoring R15 be explicitly programmed. This tedious job of preserving the contents of the program base register can be avoided by using the $BASE compiler instruction (cf. 6.3), or by explicitly specifying a base register in the procedure heading (cf. 2.3.7).

7.3 Calling PL360 Procedures from External Routines

Symbolic names and corresponding addresses to be made known to routines external to the PL360 program are specified by the global and common declarations of PL360. Global names specified in procedure declarations are associated with the corresponding procedure entry point. The external invocation of PL360 procedures must satisfy the following restrictions:

1. At entry to a PL360 procedure, the procedure base register (usually R15, but cf. 2.3.7, 6.3) must contain the procedure entry address and the register specified in the procedure declaration must contain the return address.
2. At exit from a correct PL360 procedure, the register specified in the procedure declaration will contain the return address.

In addition, the following points should be noted:

1. If the PL360 program was compiled from a statement and not a global procedure declaration,

   a. the symbolic name of the entry point will normally be SEGN001, the symbolic name of the implicit data segment (with base register R13) will normally be SEGN000 (cf. 6.3);
   b. the return register will be R14;
   c. at entry, R13 must contain the address of an 18 word save area, if the $OS option is in effect (cf. 6.3);
   d. at exit, all registers are restored from this save area.

2. Immediately prior to exit from a PL360 procedure, R15 may be loaded with a return code.
3. Global and external names violate the rules of scope established by the PL360 block structure (cf. 2.2.4). By pairing global and external declarations, a name can be given arbitrary scope. Recursive procedures and coroutines can be programmed using this feature; however, this ability should be used carefully and sparingly.

Consider the following example.

```
global procedure P1 (R1);
   begin global data D1 base R10;
      integer A;
      global procedure P2 (R2);
      begin R0 := A;
      end;
      global procedure P3 (R2);
      begin external data D1 base R10;
         integer A;
         R0 := A;
      end;
      R0 := A+1;
   end.
```

The procedure P2 can be entered with the base register for data segment D1 incorrectly loaded, since it is possible to bypass the entry code of the block containing the base declaration. In procedure P3, however, the external declaration causes register loading, but all declarations must be repeated. In general, procedures which are to be entered independently should be declared as separate programs whenever possible.

It should be noted that the registers specified in corresponding global and external procedure declarations must be identical, while the registers specified in corresponding global, external, and common data segment declarations may be different.

8. Operating System Interface and Configuration Requirements

The PL360 compiler contains no direct calls to an operating system, nor does it contain any code dependent upon any specific operating system environment. Instead, subroutines which interface with a particular operating system must be separately assembled and merged with the compiler object modules by suitable linkage editing. Consequently, any operating system using 360 standard object modules (e.g. MTS, OS, BOS, TOS, DOS) can accommodate the compiler. The PL360 compiler uses the following external names for entry points to such routines:

READ	SYSINIT
WRITE	SYSTERM
PUNCH	

The following information is intended to facilitate the writing of appropriate subroutines.

8.1 Register Conventions

The following conventions apply to all the above entry points:

1. R13 contains the address of a standard 18 word save area.
2. R14 contains the return address.
3. R15 contains the address of the entry point.

In addition, other registers and the condition code are used for input or output parameters in those cases specified below. Before return to the PL360 compiler, all registers (except R15 and any output parameter registers) must be restored.

8.2 Subroutine Specifications

1. SYSINIT

Function: system initialization, including

a. any required parameter list decoding,
b. opening required data sets,
c. obtaining free storage (at least 12,000 bytes),
d. supplying system or job identification.

Input: none supplied by PL360.
(Registers R1-R5 will be unchanged from the point of entry to the compiler.)

Output: R1 -- address of a 16 byte character string to be used as identification in compilation listing headings.

R3 -- address of first byte of free storage available for use by the compiler.

R4 -- address of first byte past the end of the free storage area supplied.

R15 -- set to #FF if the $OS option is to be used, set to 0 of the $DOS option is to be used (cf. 6.3).

2. SYSTERM

Function: system termination, including

a. release of free storage,
b. closing required data sets.

Input: none

Output: R15 is set to 0 if the object module output from the compiler was discarded by the PUNCH routine. R15 should be set to some nonzero value if this is not the case. The compiler uses this information in setting a return code when it terminates.

3. READ

Function: transmission of a card image record to the compiler (source program input).

Input: R0 -- address of 80 byte buffer into which the record is to be moved.

Output: Condition code set to
- 2 if no record was transmitted (input file exhausted),
- 0 otherwise.

4. WRITE

Function: transmission of a line image record from the compiler (listing output).
Records are 133 bytes in length; the first byte is a USASI control character (" ", "0", or "1"), and the last 12 bytes may be ignored without substantial information loss.

Input: R0 -- address of 133 byte output record.

Output: none

5. PUNCH

Function: transmission of a card image from the compiler (object module output).

Input: R0 -- address of 80 byte output record.

Output: none

8.3 Linkage to the Compiler

The PL360 compiler assumes the calling conventions outlined in 8.1. That is, the compiler is always compiled with the $0S option (cf. 6.3). Parameters to be interpreted by SYSINIT can be supplied in R1 through R5. Upon exit from the compiler,

1. R15 is set to 16 if any compilation errors were detected, to 8 if the return code from SYSTERM is 0, and to 24 if both conditions exit; otherwise R15 is set to 0.
2. all other registers are restored.

8.4 Configuration Requirements

The compiler requires:

1. A System/360 processor with at least the scientific instruction set.
2. At least 52,000 bytes of main storage (for the compiler and free storage used for table space) plus whatever is required for the interface module and input-output buffer space.
3. A reader and either a punch or a device accommodating 80 character records with EBCDIC encoding.
4. A printer or device accommodating 133 character records with EBCDIC encoding of the PL/1 60-character print set.

9. Use as an OS Language Processor

This section describes the use of the PL360 compiler, with the standard interface routines, in the environment of Operating System/360 (OS). An effort has been made to keep the Job Control Language statements and processor options similar to those for the IBM OS Assembler (E, F).

9.1 Data Set Requirements

The PL360 compiler uses the data sets described below, identified by the DDNAMEs required. All data sets are sequential with fixed blocked format. Unless supplied by the system or by data set labels, DCB parameters for physical block size (BLKSIZE) and number of buffers (BUFNO) must be specified in the DD statement, except for SYSPRINT and SYSPUNCH. These two data sets will use a default block size equal to the logical record size if no value is specified elsewhere. Through selection of compilation options (cf. 9.2), reference to any or all of the output data sets can be prevented. In such cases, no corresponding DD statement is required.

1. SYSIN

   This data set, consisting of compiler instructions (e.g. $NOLIST) and one or more PL360 source programs, constitutes the input to the compiler. The logical record length is 80 bytes. Concatenation of data sets with unlike attributes is supported; however, space for buffers and access method routines must never exceed that required for the first of the concatenated data sets.

2. SYSPRINT

   This data set contains the compiler output listing, including all diagnostic messages. The logical record length is 133 bytes; the first byte of each record is a control character.

3. SYSPUNCH

This data set contains compiler output in the form of a sequence of object modules. Some or all of the object modules corresponding to programs in which errors were detected will be missing. The logical record length is 80 bytes. The data set is closed with a disposition of LEAVE.

4. SYSGO

This data set contains object module output identical to that described for SYSPUNCH. It is closed with a disposition of REREAD for further processing, such as linkage editing.

9.2 Processing Options

The production of listing and object module output by the compiler is controlled explicitly by compiler instructions (cf. 6.3) or implicitly (through error detection) by the input stream. Independent control of the transfer of this output to OS data sets is provided by the following compiler options, which can be specified in the PARM field of the job step EXEC statement. In each case, the unqualified parameter causes transfer to the indicated data set; the prefix "NO" inhibits such transfer.

Options	Data Set
LIST, NOLIST	SYSPRINT
LOAD, NOLOAD	SYSGO
DECK, NODECK	SYSPUNCH

LOAD and DECK options are not mutually exclusive. Options may be specified in any order; in the case of conflict, the rightmost specification is used. Default options are equivalent to

```
PARM='LIST,LOAD,NODECK'
```

9.3 Return Code

The return code supplied by the compiler is > 0 if any errors are detected, or if the $NOGO directive (cf. 6.3) is detected, or if the NOLOAD option is in effect, 0 otherwise (cf. 8.3).

9.4 JCL Statements

The catalogued procedure used at Stanford is listed in this section as an illustration of typical job control language. At Stanford, the compiler is available as a load module in the partitioned data set T123.PLLIB on SYS21. The input-output routines and other run-time object modules which may be used by PL360 programs (cf. 13) are available in the SYSLIB partitioned data set T123.PLSYSLIB on SYS21. The linking-loader automatically accesses this library to resolve external references. To compile, link and execute a PL360 program using the catalogued procedure, the following JCL is sufficient:

```
//TESTPROG JOB ...
//  EXEC  PL360CG
//PL360.SYSIN  DD  *
       [PL360 source programs]
/*
//GO.SYSIN  DD  *
       [data]
/*
```

The text of the catalogued procedure follows:

```
//PL360 EXEC PGM=PL360
//STEPLIB DD DSN=T123.PLLIB,UNIT=2314,VOL=SER=SYS21,DISP=OLD
//SYSGO DD DSN=SYS1.UT2,UNIT=2314,DISP=(OLD,PASS),
//      DCB=(KEYLEN=0,BLKSIZE=1600)
//SYSPRINT DD SYSOUT=A
//SYSPUNCH DD SYSOUT=B
//GO EXEC PGM=LOADER,PARM='MAP',COND=(0,NE,PL360)
//SYSLOUT DD SYSOUT=A
//SYSLIN DD DSN=*.PL360.SYSGO,DISP=(OLD,KEEP)
//SYSLIB DD DSN=T123.PLSYSLIB,DISP=OLD,UNIT=2314,VOL=SER=SYS21
//SYSPRINT DD SYSOUT=A
//SYSPUNCH DD SYSOUT=B
//SYSUDUMP DD SYSOUT=A
```

9.5 Library

The standard procedures described in Sections 6.1.4 and 13 are included in the SYSLIB library. The input-output subroutines all use the queued sequential access method (QSAM). The unit record input-output routines assume fixed blocked records. A default value equal to the logical record size is used for the block size unless one is supplied by data set label or by the DCB parameter BLKSIZE of the corresponding DD statement. The ddname correspondences for these procedures are:

Procedure	ddname
READ	SYSIN
WRITE	SYSPRINT
PAGE	SYSPRINT
PRINT	SYSPRINT
PUNCH	SYSPUNCH

The other input-output routines assume nothing about the DCB information. This information must be supplied either by data set labels or by the DCB parameters in the DD statements. Corresponding ddnames are chosen by the programmer.

Abnormal termination of a job (ABEND) may occur with these subroutines in the following situations:

ABEND	U0095	A unit record file could not be opened upon the first occurrence of a READ, WRITE, PRINT or PUNCH.
ABEND	U0096	A READ was attempted after reaching an end-of-file.
ABEND	U0097	An attempted OPEN was unsuccessful.
ABEND	U0098	A GET, PUT or KLOSE was attempted with an unopened DCB.
ABEND	U0099	A GET was attempted after reaching an end-of-file.

If a SYSUDUMP DD statement is included for the job step, a dump will follow the above ABENDs.

10. Use as a DOS Language Processor

This section describes the use of the PL360 compiler with the standard interface routines, in the environment of the Disk Operating System/360 (DOS).

10.1 System Configuration Requirements

The Disk Operating System is usually used on small 360 machines. The PL360 is an "in core" compiler and cannot be run on many of the smaller 360 computers due to the core memory requirements. The PL360 compiler together with the DOS interface and buffer space require approximately 55,000 bytes of core memory. Since DOS requires about 10K of memory, this dictates a minimum memory capacity of 64K for running PL360.

Following is a list of the logical files used by the DOS-PL360 interface. These files are usually assigned to the proper devices; however, default assignments can easily be overridden with job control statements.

1. SYSIPT

   This file contains the primary input to the compiler; namely compiler instructions (e.g. $NOLIST) and one or more PL360 source programs.

2. SYSPCH

   This file is used for compiler output in the form of a sequence of object modules. Some or all of the object modules corresponding to source programs in which errors were detected will be missing.

3. SYSLST

   This file is used for the compiler output listing, including all diagnostic messages. The record length is 133 bytes and the first byte of every record is a control character.

4. SYSLNK

   This file receives object module output identical to that described for SYSPCH except that it is written in the special variable record length format required by the DOS linkage editor. It is assumed by the interface that this file resides on a 2314 disk; however, this assumption can easily be changed (to a 2311, for example) in the source code of the interface.

10.2 Processing Options

The production of listing and object module output by the compiler is controlled explicitly by compiler instructions (cf. 6.3) or implicitly (through error detection) by the input stream. Independent control of the transfer of this output to DOS files is provided by the following compiler options, which can be altered through the first three bits of the UPSI byte. In each case, the unqualified option cuases the transfer to take place; the prefix "no" inhibits the transfer.

Options	File Name
list, nolist	SYSLST
load, noload	SYSLNK
deck, nodeck	SYSPCH

The default options are (list,load,nodeck) . These default options can be changed with the job control statement

```
// UPSI ijk
```

where i,j,k may be either 0 or 1 :

0 - take the default option,
1 - reverse the default option.

If no UPSI statement is included in the job control input stream, then

```
// UPSI 000
```

is assumed. The load and deck options are not mutually exclusive.

10.3 JCL Statements

The following sample of job control statements which can be used for invoking the PL360 compiler, link editing the output and executing the resulting program assume that the compiler is available in the Core-Image Library under the name PLDOS (digits cannot be used in the name, unfortunately), the elementary unit record input-output subroutines (READ, WRITE, PRINT, PAGE, PUNCH) and CANCEL are available in the Relocatable Library under the

name PL360IO, the tape and disk input-output subroutines (OPEN, GET, PUT, KLOSE) are available in the Relocatable Library under the name PLTAPEIO, and the run-time library (cf. 13) is available in the Relocatable Library under the name RUNLIB.

```
// JOB [jobname]    [comments]
// UPSI ijk
        if any, (cf. 10.2)
// OPTION {LINK
           CATAL}
```

If the linkage editor is to be used, this statement must be included. The option LINK is for compile-and-go jobs, while the CATAL option also retains the core-image module and catalogues it in the Core-Image Library.

```
// PHASE [program name],S+m
```

This statement is necessary only if option LINK or CATAL are in affect. The integer m is set equal to 80ℓ where ℓ is the number of standard data set labels to be processed during the EXEC step.

```
// EXEC PLDOS
```

This statement invokes the PL360 compiler.

```
[source program]
/*
```

The following statements are necessary only if option LINK or CATAL are in affect.

```
INCLUDE PL360IO
```

Includes the READ, WRITE, PAGE, PUNCH, PRINT and CANCEL subroutines.

```
INCLUDE PLTAPEIO
```

Includes the OPEN, GET, PUT and KLOSE subroutines.

```
INCLUDE RUNLIB
```

Includes the run time library of subroutines (cf. 13).

```
ENTRY
```

```
// EXEC LNKEDT
```

This invokes the linkage editor.

[JCL for particular input-output requirements of the job; for example, label processing, etc.]

```
// EXEC
```

This executes the core-image module produced by the linkage editor.

[card input for the program, if any]

```
/*
/&
```

A typical job with lineprinter output and no input would have a deck setup as follows:

```
// JOB TESTJOB
// OPTION LINK
// PHASE T,S
// EXEC PLDOS
{source program}
/*
 INCLUDE PL360IO
 ENTRY
// EXEC LNKEDT
// EXEC
/&
```

10.4 DTF Tables

This section describes the conventions which must be followed when coding DTF macro instructions for the tape and disk input-output subroutines. (Refer to [14] for a complete description of the necessary macro instructions.)

Generally, the only macro instructions needed are: DTFMT (Define The File for Magnetic Tape), MTMOD (Magnetic Tape input/output control section MODule), and DTFSD and SDMODxx for sequential disk files. The DOS routines for GET and PUT require the following conventions in declaring the DTF table:

```
EOFADDR = ENDRDR,
IOREG = (2) .
```

Since the DTF macro instruction is assembled by the IBM assembler as a separate module, an

```
EXTRN   ENDRDR
```

statement must be included in the assembly. ENDRDR is actually a subroutine in the PLTAPEIO module.

The name of the DTF table must be accessible to your PL360 program. Thus, it must be specified as an ENTRY point. The easiest way to access it in the PL360 program is with an external procedure declaration. For example, if the DTF table is called INFILE, one might code

```
external procedure INFILE (R14); null;
integer DTFADDR = @@ INFILE;
```

in the declarations of the PL360 main program. An external data declaration may be used instead.

If variable-length records are to be written, the RECSIZE parameter must be used with its value being the maximum possible length of a record. Also,

```
VARBLD = (3)
```

must be specified. The PUT subroutine uses these parameters as follows: If the remaining length in a buffer becomes less than RECSIZE, a TRUNC macro instruction is automatically issued to write the physical block and the subsequent record is started at the beginning of the next buffer area (IOAREA).

The example on the following page is for variable-length input from magnetic tape with blocks of at most 4000 bytes. The subroutine that actually does the input is called a "logic module" and is generated by the MTMOD macro instruction. When expanded, this assembly produces 314 card images. Assembly time on a model 30 is about 5-10 min. The necessary job control language has been included in the example. Notice that buffer areas must be explicitly declared.

If you are not familiar with the hazards of writing these macro instructions, be very careful and read every word of the instructions contained in the Supervisor and IO Macro Manual [14].

```
//   JOB  IOASM
//   OPTION  LOG,DECK,LIST,XREF
//   EXEC    ASSEMBLY
INFILE     DTFMT BLKSIZE=4004,                                    X
                 DEVADDR=SYS010,                                  X
                 EOFADDR=ENDRDR,                                  X
                 FILABL=STD,                                      X
                 ERROPT=IGNORE,                                   X
                 HDRINFO=YES,                                     X
                 IOAREA1=PACKIN,                                  X
                 IOAREA2=PACKIN2,                                 X
                 IOREG=(2),                                       X
                 MODNAME=IJFVZZZY,                                X
                 RDONLY=YES,                                      X
                 RECFORM=VARBLK
           EXTRN ENDRDR
           ENTRY INFILE
PACKIN     DS    4004C
PACKIN2    DS    4004C
IJFVZZZY   MTMOD ERROPT=YES,                                      X
                 RDONLY=YES,                                      X
                 READ=FORWARD,                                    X
                 RECFORM=VARBLK
           END
/*
/&
```

10.5 Library

The standard procedures described in Sections 6.1.4 and 13 are available for the DOS operating system. The input-output subroutines all use the sequential access method (SAM).

Abnormal termination of a job may result from any of the following conditions:

1. A READ or GET was attempted after reaching an end-of-file.
2. An attempted OPEN was unsuccessful.
3. A GET, PUT or KLOSE was attempted with an unopened file.

Any of these conditions will result in a core dump.

11. Use as an MTS Language Processor

This section describes the use of the PL360 compiler, with the standard interface routines in the environment of the MTS Operating System.

11.1 Data Set Requirements

The PL360 compiler uses the logical files SCARDS, SPRINT and SPUNCH and the device PUNCH1.

1. SCARDS
   This input file consists of compiler instructions (e.g. $NOLIST) and one or more PL360 source programs.
2. SPRINT
   This file contains the compiler output listing, including all diagnostic messages.
3. SPUNCH
   This file contains the object modules output by the compiler. Some or all of the object modules corresponding to programs in which errors are detected will be missing.
4. PUNCH1
   Provides the object modules on cards (batch runs only).

11.2 Processing Options

The production of listing and object module output by the compiler is controlled explicitly by compiler instructions (cf. 6.3) or implicitly (through error detection) by the input stream. Independent control of these data transfers is provided by the following compiler options, which can be specified in the PAR field of the $RUN command. In each case the unqualified parameter causes the transfer to take place; the prefix 'NO' inhibits the transfer.

Options	File or Device
LIST,NOLIST	SPRINT
LOAD,NOLOAD	SPUNCH
DECK,NODECK	PUNCH1

The DECK option is available only in batch runs; LOAD and DECK are not mutually exclusive. Options may be used in any order, in case of conflict the rightmost specification is used. The default options are

PAR=LIST,LOAD,NODECK

11.3 MTS Library

The procedures READ, WRITE, PAGE and PUNCH described in Section 6.1.4 are included in the file *PL360SLIB. An alternative version of the procedure WRITE is available in MTS, its specifications correspond to those for the procedure WRITE used by the compiler. This version of WRITE together with READ, WRITE and PAGE are available in the file *PL360LIB.

11.4 MTS Commands

1. To compile in the batch (source on cards, listing to the printer and the object program to a temporary file -T):
   $RUN *PL360 SPUNCH= -T
2. To compile from a terminal, (source on a file MYSOURCE, listing to a file MYLISTING and object program to a file MYOBJECT):
   $RUN *PL360 SCARDS=MYSOURCE SPRINT=MYLISTING SPUNCH=MYOBJECT
3. To execute the program created in Example 2, using the standard library taking data from *SOURCE* and sending printed output to a file RESULTS:
   $RUN MYOBJECT+*PL360SLIB SPRINT=RESULTS

When working from a terminal the compiler directives $0 and $NOLIST can be used. Only error messages (and their program context of one line) and one line summaries of the coding for each segment are produced.

12. Use as an Orvyl Language Processor

This section contains a brief narrative description of how one uses the interactive version of PL360 which runs under the Orvyl time-sharing monitor [13]. This version is made possible through a special Orvyl-PL360 interface module written in Assembly Language using the Orvyl macro instructions [13].

12.1 Using the PL360 Compiler with Orvyl

This section assumes that the Orvyl system is being used at Stanford where the Orvyl-PL360 compiler is saved in object module form in the Wylbur data set T123.PL360 on SYS21. To use it, just type:

```
USE &T123.PL360 ON SYS21 LOAD
```

You will then receive the message:

```
-WELCOME TO PL360
DO YOU WANT AN OBJECT DECK?
```

If your account has been activated for Orvyl files, then you can type "YES" and PL360 will respond with:

```
FILE NAME?
```

You should then type the name of an Orvyl file in which PL360 will place the object modules from subsequent compilations. This file can be either new or old. The next thing PL360 asks is:

```
DO YOU WANT A LISTING?
```

If you respond "YES", then you will again be asked to supply a file name. Thus, the listing is placed into an Orvyl file. The final question asked by PL360 is:

```
DO YOU WANT WYLBUR?
```

If your response is "NO", you will get the message:

```
BEGIN TYPING PL360 PROGRAM
-?
```

You can begin typing a PL360 program and each line will be compiled as you go. Unfortunately, if you make a mistake, you must start over since the old lines are not saved. For this reason, it is usually best to compile from a Wylbur working dataset. To do this, say "YES" you want Wylbur and PL360 will give the prompt

    -?

You can now type Wylbur commands which will be passed to and executed by Wylbur. You can continue to pass commands to Wylbur (for example, you collect lines, edit lines, use files, copy files, etc.) until your Wylbur working data set contains a PL360 program. You then type "COMPILE" immediately after a -? prompt and PL360 begins compiling the program contained in your Wylbur working data set.

Any error messages and the line on which they occur are printed at the terminal as the compilation proceeds. Each time a segment is closed a message is printed at the terminal.

When compiling from a Wylbur working data set, the compiler terminates at the end of the data set and types

    -LEAVING PL360

When typing the program in directly, you can leave PL360 by typing a "/*" in the first two columns of a line. As you are leaving PL360, the Orvyl core memory and your Wylbur working data set are cleared automatically.

If the program you are compiling has numerous errors and you wish to suppress the typing of error messages at the terminal, then simply hit the ATTN button at the terminal. Orvyl will respond (as usual) with

    DO YOU WANT YOUR PROGRAM?

Respond with a "YES". PL360 will then ask:

    DO YOU WANT FURTHER ERROR MESSAGES TYPED?

You can respond to this question with either a "YES", "NO" or by hitting the ATTN button. The ATTN button will cause PL360 to terminate and return you to Wylbur. A "NO" will cause the compilation to continue with no error messages typed at the terminal. A "YES" will cause the compilation to continue as before. In either case, (except for ATTN) the listing produced in the Orvyl file (if any) will be unaffected.

After leaving PL360, you can get the object deck by typing

GET ⟨file name⟩ CARD

You may get the listing by typing

GET ⟨file name⟩ PRINT CLEAR

The listing has 133-byte records, the first byte of which is a carriage control character. Thus, when the listing is printed offline, the following Wylbur command should be used:

LIST OFF BIN XXX UNN (0)

The (0) part of the LIST command causes the first byte to be treated as a carriage control character. The resulting lineprinter listing looks like any other PL360 compilation listing. The Orvyl version of PL360 has several advantages: Waiting for the batch queue is completely eliminated. Syntax error messages are printed at the terminal, thus syntax errors can usually be fixed immediately and another compilation can be made within a minute or two. Paper is saved since listings with syntax errors are seldom made. Finally, the Orvyl versions of the READ and WRITE routines can be used to run and test the program immediately at the terminal. In this way, Orvyl's debugging tools can be used and debugging takes far less time.

Most short compilations can be done in about a second or two of Orvyl compute time (less than 50¢). This is a significant savings over batch compilations. The PL360 compiler, which is about 2700 cards long, compiles in 37 seconds of Orvyl compute time at a cost of about $6.20.

12.2 Input-Output Subroutines for Interactive PL360 Programs

Standard input-output subroutines using the same linkage conventions as the READ and WRITE subroutines described in Section 6.4 are available for input-output operations directly at the terminal when running a PL360 program under the Orvyl monitor. A description of the parameter passing conventions of these subroutines follows:

READ The address of a 132 byte buffer should be provided in R0 prior to calling READ. Upon return, all registers are preserved except R15 which contains the number of non-blank characters typed by the user (counting imbedded blanks). All details such as error messages for illegal use of tabs or waiting too long to respond are taken care of by the READ subroutine. If a "/*" has been typed in column 1 then the condition code is set to 2, otherwise it is set to 0.

WRITE This subroutine works exactly like the subroutine described in Section 6.4. I.e., the address of a 132 byte buffer is passed through register R0 and all registers are preserved upon return.

The following discussion assumes that the Orvyl system is being used at Stanford where the Orvyl READ and WRITE subroutines are stored in object module form in the Wylbur file T123.PL360.IO on SYS21 and the library subroutines listed in Section 13 are stored in T123.PL360.RUNLIB on SYS21. To run a PL360 program in Orvyl, just follow this simple procedure: First, compile the program. This may be achieved either in batch or with the Orvyl version of PL360. Put the object module output of the PL360 compiler in a Wylbur working dataset and type:

```
COPY ALL TO END FROM &T123.PL360.IO ON SYS21
LOAD TEXT
```

Your program will then begin execution.

13. The Run-Time Library

This section describes a set of global procedures written in PL360 which perform commonly needed tasks. These subroutines are not predeclared as external procedures in the PL360 compiler; thus they must be explicitly declared in the calling program. In all cases, the procedure linkage is done with register R14, and R15 should contain the address of the entry point upon entry. At Stanford, the linkage editor automatically adds the required subroutines if you are using the catalogued procedure PL360CG (cf. 9.4).

13.1 Number Conversion Procedures

The two subroutines described below are used to convert the EBCDIC representation of a number into an internal representation of that number, or vice-versa. A slightly more conventional number representation is used by these routines than that of the PL360 language (cf. 2.2.2). The numbers must satisfy the following syntax:

⟨long complex number⟩ ::= ⟨long real number⟩ + ⟨imaginary number⟩L
⟨complex number⟩ ::= ⟨real number⟩ + ⟨imaginary number⟩
⟨imaginary number⟩ ::= ⟨real number⟩I | ⟨integer number⟩I
⟨long real number⟩ ::= ⟨real number⟩L | ⟨integer number⟩L
⟨real number⟩ ::= ⟨unscaled real⟩ | ⟨unscaled real⟩⟨scale factor⟩|
    ⟨integer number⟩⟨scale factor⟩|⟨scale factor⟩
⟨unscaled real⟩ ::= ⟨integer number⟩. ⟨integer number⟩|
    .⟨integer number⟩|⟨integer number⟩.
⟨scale factor⟩ ::= '⟨integer number⟩|'⟨sign⟩⟨integer number⟩
⟨integer number⟩ ::= ⟨digit⟩|⟨integer number⟩⟨digit⟩
⟨sign⟩ ::= + | -

Numbers are interpreted according to the conventional decimal notation. A scale factor denotes an integral power of 10 which is multiplied by the unscaled real or integer number preceding it. A number can have no imbedded blanks and must be delimited by a blank.

The parameter passing conventions for the two conversion subroutines are as follows:

VALTOBCD This procedure converts an internally stored value to an EBCDIC representation. At entry,

R1 contains the address of an area to receive the EBCDIC representation

R2 indicates the type:

1 = integer

2 = real

3 = long real

4 = complex

5 = long complex

R3 contains the field length (≥ 1)

The value to be converted is in either R0, F0, F01, F0 and F2, or F01 and F23, depending upon the type.

A return code is left in R15:

0 => successful conversion

1 => field size too small

2 => invalid fieldsize

When the field size is too small to receive the value, the field is filled with stars (*) .

All registers, except R15, are preserved.

BCDTOVAL This procedure converts an EBCDIC representation of a number to an internal number. At entry,

R1 contains the address of the EBCDIC representation (possibly preceded by blanks)

R2 indicates type (see above)

The resulting value is left in either R0, F0, F01, F0 and F2, or F01 and F23, depending upon the type.

A return code is left in R15:

- 0 => successful scan
- 1 => invalid character in input string
- 2 => missing "I" on imaginary part
- 3 => nonblank delimiter
- 4 => number scanned is not assignment compatible (e.g., a decimal point is found when R2 = 1)
- 5 => integer too large

Upon exit, R1 contains the address of the delimiter. Registers R2-R14 are restored.

13.2 Data Manipulation Procedures

The first procedure described in this section does an in-core indirect sort using logical comparisons. The second procedure is a companion routine which searches a sorted list for a specified element.

SHELSORT This procedure sorts character data. The Shell sort technique is used. At entry, registers R0-R3 must be set as follows:

- R0 = the number of items to sort
- R1 = the address of the index array
- R2 = the number of the first byte of the key in each record on which the sort is to be done. (R2 >= 1)
- R3 = the number of bytes in the key on which the sort is to be done.

The index array is a list of 4-byte integers containing the addresses of the items to be sorted. The actual sort is done on the elements of the index array and not the records themselves. That is, only the order of the elements of the index array is modified by the procedure. All registers are restored.

BISEARCH This procedure locates an element in a sorted list. At entry, registers R0-R4 must be set as follows:

R0 = the number of entries in the sorted table

R1 = the address of the index array (see above)

R2 = the number of the first byte of the key field in the records

R3 = the number of bytes in each key field

R4 = the address of the key for which you are looking

At exit, R1 contains the address of an element in the index array that points to a record that contains the desired key. If no match is found, R1 = 0 .

All registers, except R1, are preserved.

14. Format of PL360 Programs

The following rules (except for some minor modifications) were proposed by Wirth [15] during the development of the Algol W compiler (which is written in PL360) as guidelines for producing uniformly readable PL360 programs. They have proved helpful and effective in both programming and debugging. However, they must not be regarded as strict rules to be followed under any circumstances, but rather as guidelines to be followed when no stronger reasons dictate a choice.

14.1 Indentation

(a) Indent lines contained between begin and end by 3 spaces:

```
begin ...
   R1 := R2; ...
   begin ...
      page; R0 := @line; ...
   end; R6 := R5; ...
end;
```

(b) Do not indent after if, for, while clauses, but reserve a separate line for the clause, if it is followed by a lengthy statement:

```
for R1 := 1 step 1 until 100 do
begin ...
   ...
end;
```

However:

```
if R0=1 then R1 := R1+1;
```

(c) In the case of _if then else_ , the two statements should be shown to be of equal "importance", that is:

```
if RO=0 then R1 := 1 else R2 := R1;
```

or

```
if RO=0 then
begin ...
    ...
end else
begin ...
    ...
end;
```

(d) A program sometimes consists of a few very large blocks, each being one or more pages long. In this case, indentation does not make sense because the reader cannot see that the page he is reading uses indentation at all. It is preferable to accompany the _begin_ and the _end_ of such a major block with a short comment linking them together with a common name or number.

14.2 Spacing

(a) Spacing is a powerful tool in grouping things together which should be read together, and to display the structure of a statement. If spaces are used in the same amount everywhere, they are useless and may as well be omitted with the benefit of saving paper. An example may illustrate the idea:

```
R1 := TEMP / 4 + SLAB9 * C ; TEMP := R1 ;
```

is equally as bad as

```
R1:=TEMP/4+SLAB9*C;TEMP:=R1;
```

Instead write

```
R1 := TEMP /4 + SLAB9 *C;  TEMP := R1;
```

The following rule may seem a bit absurd, but nevertheless it has proven useful: Use no space between single letter identifiers and operators, otherwise use one space.

(b) Always use one space before and after the assignment operator.

14.3 Choice of Identifiers

(a) In general, use descriptive words for identifiers (in particular labels). This serves as an implicit comment. However, if the identifier occurs very often, it may be advantageous to use a short (possibly one-letter) identifier.

(b) In this case, the declaration must be accompanied by a comment explaining the nature of the quantity.

(c) Another exception from (3a) is the case where the identified quantity or program location has only extremely local significance, such as temporary storage cells or loop labels. In this case, the one-letter identifier may be used to underscore the auxiliary and local role of the quantity or label.

14.4 Comments

(a) Comments should always be given at key points such as along with declarations, at block entry, in the procedure heading.

(b) If they occur elsewhere, they may represent "snapshots"; they should explain relationships between variables which hold unconditionally when control passes the point of the comment. Such snapshots are sometimes extremely useful in explaining the functioning of a program.

(c) In PL360, comments will sometimes be necessary to explain the role of a sequence of "obscure" function statements.

(d) In block- and procedure headings, it is useful to add a comment indicating which registers are used, or vice versa: which ones are not. Often it is useful to indicate what the registers are used for.

14.5 Miscellaneous

(a) Declare quantities which have local significance only in the block where they belong. Avoid sharing of local variables, in particular avoid sharing "temporary storage cells" among several procedures.

(b) Avoid labels where you can. This is not as easy in PL360 as it is in Algol. Nevertheless, use <u>if</u>, <u>for</u>, and <u>while</u> statements instead of <u>goto</u> statements where appropriate. When a label must be used, always put it in the left margin where it can be easily located. When a <u>goto</u> statement is used in a large program, it is sometimes useful to accompany it with a comment telling the reader approximately where the label is defined.

(c) Use the appropriate type symbols when declaring variables. For example, do not write

<u>integer</u> flag

when that variable is never used as a number, but only as a logical quantity.

(d) Avoid bit manipulation where possible. For flags, use byte variables and the functions SET, RESET and TEST.

(e) Minimize the use of functions.

(f) Avoid the use of subscripted synonyms, such as

<u>integer</u> x <u>syn</u> y(R2)

It is hard to realize that the statement

R1 := x

uses R2 as an index register! Of course

<u>integer</u> x <u>syn</u> y(2)

is o.k.

15. Acknowledgments

After the U.S. invasion of Cambodia in the Spring of 1970, a handful of graduate students in the Computer Science Department at Stanford decided to apply their programming skills to the data processing problems of "working within the System". We decided to do a computer analysis of precinct data to aid a candidate in the local Congressional race. One of the early questions which had to be answered was what language to use? For various reasons we chose PL360. This decision necessitated the design and coding of input-output subroutines for tape and disk units (OPEN, GET, PUT and KLOSE). Many of our jobs were limited in speed by the input-output devices; thus, the DOS interface was implemented to permit "production" runs to be made on a small 360. Through the many nights and weekends of programming and running jobs, we became aware of shortcomings in the compiler, the input-output subroutines and the PL360 language. Many times we were in the unique position of developing and debugging the compiler, the input-output subroutines and problem programs, simultaneously. As a result, I believe that the PL360 system has evolved into a tool that is not only elegant, but useful!

Thanks are due to Edwin Satterthwaite for many discussions and explanations about PL360 and the OS interface.

Special thanks are due to Richard Guertin of the Spires/Ballots project at Stanford who recently took an interest in PL360 and made several improvements. His careful scrutiny of the manuscript kept many of my errors out of print.

The other students who programmed for the congressional race helped in many ways, especially in the design of new features and debugging. They are: Robert Russell, Henry Bauer and Richard Underwood.

Our enthusiasm about PL360 was not dampened by the fact that our candidate lost.

16. References

[1] N. Wirth: PL360. "A Programming Language for the 360 Computers", JACM 15 (1968) 37.

[2] "OS/360 PL360 Compiler", IBM Contributed Library (Type IV) Program Number 360D-03.2.011.

[3] J. Eve: "PL360 Language Extensions", Internal Note, Computing Laboratory. University of Newcastle upon Tyne.

[4] G. M. Amdahl, G. A. Blaauw, F. P. Brooks, Jr.: "Architecture of the IBM System/360", IBM J. of Res. and Dev. 8 (1964) 87.

[5] G. A. Blaauw et al. "The Structure of System/360", IBM Sys. J. 3 (1964) 119.

[6] "IBM System/360 Principles of Operation", IBM Sys. Ref. Lib. A22-6821.

[7] "IBM System/360 OS Assembler Language", IBM Sys. Ref. Lib. Form C28-6538.

[9] MTS Vol. I 290-0 et. seq., University of Michigan Computation Center, Ann Arbor.

[10] "IBM System/360 OS Assembler F Programmers Guide", IBM Sys. Ref. Lib. Form C26-3756.

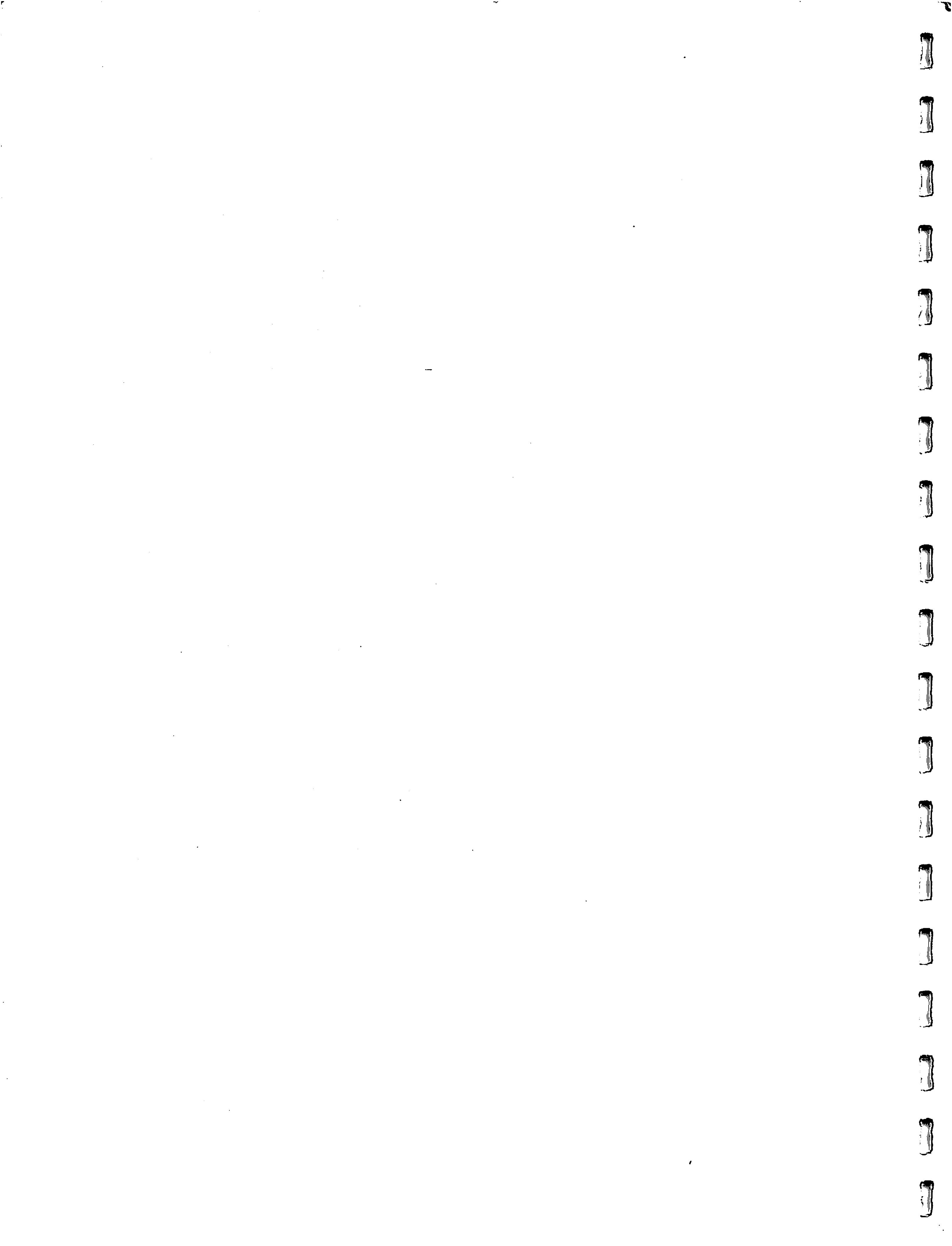
[11] "PL360 Programming Manual", University Computing Laboratory, University of Newcastle upon Tyne, Claremont Tower, Newcastle upon Tyne, NE1 7RU, England, 1970.

[12] "IBM System/360 DOS System Control and System Service Programs", IBM Sys. Ref. Lib. Form C24-5036.

[13] R. Fajman and J. Borgelt, "Orvyl User's Guide", Stanford University Computation Center, 1971.

[14] "IBM System/360 Disk Operating System Supervisor and Input/Output Macros", IBM Sys. Ref. Lib. Form C24-5037.

[15] N. Wirth: "Format of PL360 Programs", Algol W - Project Memo, Stanford University, Sept. 9, 1966.



Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Stanford University	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE

PL360 (REVISED), A PROGRAMMING LANGUAGE FOR THE IBM 360

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Technical Report, May 1971

5. AUTHOR(S) *(First name, middle initial, last name)*

Michael A. Malcolm

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
May, 1971	91	15

8a. CONTRACT OR GRANT NO. ONR N00013-67-A-0112-0029 b. PROJECT NO. NSF GJ-408 AEC AT(04-3) 326PA30 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) STAN-CS-71-215 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* None

10. DISTRIBUTION STATEMENT

Releasable without limitations on dissemination.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Mathematics Program Office of Naval Research Arlington, Virginia 22217

13. ABSTRACT

In 1968, N. Wirth (Jan. JACM) published a formal description of PL360, a programming language designed specifically for the IBM 360. PL360 has an appearance similar to that of Algol, but it provides the facilities of a symbolic machine language. Since 1968, numerous extensions and modifications have been made to the PL360 compiler which was originally designed and implemented by N. Wirth and J. Wells. Interface and input-output subroutines have been written which allow the use of PL360 under OS, DOS, MTS and Orvyl.

A formal description of PL360 as it is presently implemented is given. The description of the language is followed by sections on the use of PL360 under various operating systems, namely OS, DOS and MTS. Instructions on how to use the PL360 compiler and PL360 programs in an interactive mode under the Orvyl timesharing monitor are also included.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Compilers						
Computer Languages						
IBM 360 Language Processors						
Iteractive Language Processors						