P L 3 6 0 REFERENCE MANUAL

Stanford University

fetched from ftp://lindy.stanford.edu/pub/pl360.tar.gz
and slightliy reformatted from ANSI carriage lineprinter
control to ASCII characters and using form feed characters.

This manual was written and formatted for a lineprinter
with a wide carriage. Examples in appendix A are likely
to get truncated in printing.

If you are reading this text as a pdf file, you will
see appendix a in landscape mode without truncation.

CONTENTS

TABLES

REFERENCES

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

[2] SCIP/Academic Computing Services Program Libraries, Polya Hall Stanford University.

[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 Journal of Research and Development 8 (1964) 87.

[5] G. A. Blaauw et al. "The structure of System/360," IBM Systems Journal 3 (1964) 119.

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

[7] "IBM System/360 OS Assembler Language," IBM C28-6514.

[8] MTS Vol. I, University of Michigan Computation Center, Ann Arbor.

[9] "IBM System/360 Linkage Editor and Loader" IBM C28-6538.

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

[11] "IBM System/360 DOS System Control and System Service Programs," IBM C24-5036

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

[13] "IBM System/360 Disk Operating System Supervisor and Input/Output Macros," IBM C24-5037.

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

FOREWORD

The intent of this manual is to provide a reference tool for programmers using PL360. Although it is not a textbook, it has been organized in such a way that each section introduces new material dependent on information covered in preceding sections. In that sense, it can serve as a self-teaching aid.

Those readers not familiar with Bacus-Naur Form (BNF), may find the syntactic rules used to describe the language difficult to understand. However, the textual descriptions and examples associated with a set of syntactic rules should serve to clarify those rules. Also, the sample programs of Appendix A further clarify the language structure.

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 authors have drawn heavily upon the (anonymous) PL360 Programming Manual published by the University of Newcastle upon Tyne, Computing Laboratory [10].

SECTION 1. INTRODUCTION

PL360 is a programming language designed specifically for the IBM System/360 computers. It provides the facilities for 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 Stanford Program Library [2]. This compiler 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 were carried out at the University of Newcastle by James Eve. These changes [3,10] were aimed primarily at relaxing the linkage constraints on separately compiled programs, enabling for example direct communication with programs using OS/360 type linkages. Michael Malcolm of the Stanford Computer Science Department made several modifications to the version of the compiler produced by James Eve. These extensions made it possible to run the compiler and compiled programs under DOS operating systems. Assembly language subroutines were also written for both OS and DOS to facilitate input-output with sequential tape and disk files. Dick Guertin of the Stanford Center for Information Processing extended the syntax of PL360, primarily to increase programming convenience. He has also written assembly language interfaces to allow interactive use of both the PL360 compiler and PL360 programs under the ORVYL time-sharing monitor at Stanford. Andrew Koening of Columbia University also contributed improvements to the compiler.

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.

SECTION 2. DEFINITION OF THE PL360 ENVIRONMENT

2.1 Terminology, Notation, and Basic Definitions

The language is defined in terms of a computer which is comprised of 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 exist 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 or character (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, interpreted as a base-16 floating-point number)
e. long real (a sequence of 64 bits = 8 bytes, 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 this manual. Likewise, byte and character are equivalent and only byte is mentioned.

2.1.2 Relationships

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

Another relationship exists 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 manual; 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, 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. 2.2.2). They are either single characters, or uppercase letter sequences.
b. Strings are sequences of characters enclosed in quote marks i.e. "string" (cf. 3.4).
c. Comments are sequences of characters (not containing a semicolon) preceded by the basic symbol COMMENT and followed by a semicolon (;). Comments may also be written as a sequence of characters between vertical bars (!). Thus, ! this is a comment ! 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 syntatic 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. Appendix D, E) are denoted by english words enclosed in brackets < and >. These words describe approximately the nature of the syntatic 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 uppercase letters A, K, and T are also used in the denotation of syntactic entities. They stand as abbreviations for any of the following words (or pairs):

A	K	T
long real	long real	long real
real	real	real
integer	integer	integer
short integer		short integer
		byte

Syntactic rules are of the form <E> ::= & where <E> is a syntactic entity (called the left side) and & is a finite sequence of tokens and syntactic entities (called the right side of the rule). The notation

<E> ::= &1 ! &2 ! ... ! &n

is used as an abbreviation for the n syntactic rules

<E> ::= &1, <E> ::= &2, ..., <E> ::= &n

If in the denotations of constituents of the rule the uppercase letters A, K, or 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

<K-register> ::= <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>

2.2 Identifiers and Basic Symbols

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

Throughout this section, user defined identifiers are shown in lowercase letters to distinguish them from standard identifiers and basic symbols. In actual practice, all identifiers are constructed from uppercase letters.

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
<digit> ::= 0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9
<identifier> ::= <letter> ! <identifier><letter> ! <identifier><digit>
<K-register> ::= <identifier>
<T-cell identifier> ::= <identifier>
<procedure identifier> ::= <identifier>
<function identifier> ::= <identifier>
<integer value identifier> ::= <identifier>
```

An identifier is a K-register, T-cell, procedure, function, or integer value identifier, if it has respectively been associated with a K-register, T-cell, procedure, function, or integer value (called a quantity) in one of the blocks surrounding its occurrence (cf. 4.1). 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 Section 2.2.3.

2.2.2 Basic Symbols

Basic symbols which consist of uppercase letter sequences shown below are denoted by the same letter sequences without further distinction. Such letter sequences are called reserved words and cannot be used as identifiers. 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 or other non-alphanumeric. 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 REGISTER
CHARACTER PROCEDURE
```

2.2.3 Standard Identifiers

The following identifiers are predeclared in the language but may be redeclared due to block structure. Their predefined meanings are specified in Section 5, Section 7.1, or Section 9.1.

```
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
CARRY FALSE MIXED OFF ON OVERFLOW STRING TRUE
CANCEL GET KLOSE OPEN PAGE PRINT PUNCH PUT READ WRITE
```

SECTION 3. VALUES

3.1 Hexadecimal Values

Values may be expressed in hexadecimal notation.

```
<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
```

Note: If hexadecimal values are used in conjunction with arithmetic or logical operators in a program, they must be considered as a sequence of bits which constitute 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].

3.2 Decimal Values

```
<unsigned integer number> ::= <digit>
                            ! <unsigned integer number> <digit>
<unsigned short integer number> ::= <unsigned integer number> S
<fractional number> ::= <unsigned integer number> .
                        ! <fractional number> <digit>
<scale factor> ::= <integer number>
<floating-point number> ::= <fractional number>
                          ! <fractional number> ' <scale factor>
                          ! <unsigned integer number> ' <scale factor>
<unsigned real number> ::= <floating-point number>
                         ! <unsigned integer number> R
<unsigned long real number> ::= <floating-point number> L
                              ! <unsigned integer number> 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 underscore symbol "_". (Note: the underscore is used so as not to confuse negative values with the subtract operator "-", which is never part of a number.)

Note: A-number is an abbreviation for long real number, real number, short integer number and integer number as defined in section 2.1.4 as a notational convenience.

3.3 Numeric Values

```
<byte value> ::= <integer number> 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:             2X     _5X
 short integer values:   10S     #FF00S
 integer values:          0      #106C      _1     size
 real values:            1.0     _3.14      2.7'8  #46000001R
 long real values:       0L      3.14159265359L    #4E00000000000001L
```

3.4 String Values

There are also string values, but these are not generally used in conjunction with arithmetic or logical operators.

```
<string> ::= " <character sequence> "
           ! <hexadecimal value> X
<character sequence> ::= <character>
                       ! <character sequence> <character>
<character> ::= <any EBCDIC character except "> ! ""
```

When a string is a character sequence enclosed in quote marks, the string is limited to a total of 255 characters. If a quote mark (") is to be a character of the sequence, it is represented by a pair of consecutive quote marks.

When a string is a hexadecimal value ending in X, up to 16 hexadecimal digits may be specified. Each pair of hexadecimal digits represents one character. If the number of hexadecimal digits specified is odd, a hexadecimal 0 is prefixed to the specified value to make the total even.

```
Examples:  "ABC"          denotes the sequence  ABC
           "A""Z"         denotes the sequence  A"Z
           #C1C2C3X       denotes the sequence  ABC
```

SECTION 4. PROGRAM 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. Character position 1 may contain any character except '$' or any other character (e.g., /) that would signal a compiler control statement or job control statement.

4.1 Block Structure

```
<program> ::= <block> . !
     GLOBAL <simple procedure heading>;<statement> . !
     GLOBAL <simple procedure heading> BASE <integer register>;<statement>
<block> ::= <block body> END
<block body> ::= <block head> ! <block body><statement>; !
     <block body><label definition>
<block head> ::= BEGIN ! <block head><declaration>;
<declaration> ::= <T-cell declaration> !
     <function declaration> ! <procedure declaration> !
     <T-cell synonym> ! <K-register synonym> !
     <integer value synonym> !
     <segment base declaration> ! <segment close declaration>
<label definition> ::= <identifier> :
<statement> ::= <simple statement> ! <IF statement> !
     <WHILE statement> ! <FOR statement>
<simple statement> ::= <K-register assignment> !
     <T-cell assignment> ! <function designator> !
     <procedure statement> ! <CASE statement> ! <GOTO statement> !
     <block> ! NULL
```

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. END may be labeled. The two main purposes of a block are:

1. To enclose 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.

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
```

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 the 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 <= d < 4096

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
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 the program are to constitute segments. The program may then be organized in a way that minimizes the number of cross-references between segments.

It should be noted that a programmer's knowledge about segment sizes and occurrences of cross-reference is quite different for programs than for data. In the latter case the programmer is aware of the precise amount of storage needed for the declared quantities, and knows precisely where in the program references to a specific data segment occur. In the former case, knowledge about the eventual size of a compiled program section is only vague, and the programmer 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; this decision also conformed with the desirability of keeping program and data apart as separate entities.

4.2 Program Segmentation

A program segment corresponds to a CSECT in assembly language. The outermost block of a program is always compiled as a segment. Since by its very nature control lies in exactly one segment at any instant, one register is designated to hold the base address of the program segment currently executing. Register R15 is usually used for this purpose (however, cf. 8.1, 10.1.5). Branching to another segment is accomplished with a procedure statement which causes the program segment base register of the destination segment to be loaded with its base address before branching into that segment (cf. 8.2).

The natural unit for a program segment is the procedure (cf. 8.1). The normal way to enter a procedure is via a procedure statement (cf. 8.2), and the normal way to leave it is at its end, or by a call to another procedure which does not return. An explicit GOTO statement cannot be used for branching from one segment to another, but may be used to branch out of a local procedure within a segment. The fact that no implicitly generated instructions can ever lead control outside of a segment mimimizes the number of cross references in a natural way. Only relatively large procedures should constitute program segments, and a facility is provided to designate such procedures explicitly. A procedure to be compiled as a program segment must contain the symbol SEGMENT or GLOBAL in its heading.

4.3 Data Segmentation

For data, the programmer is aware of the precise amount of allocated memory and of the instances where references are made to these quantities. A base declaration was therefore introduced which implies that all quantities declared thereafter 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 (except for dummy base declarations and BASE R0); however, its previous contents are neither saved nor restored upon exit from the block.

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 also 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 diagnostic.

4.4 Main Program

A PL360 program which is a block is considered to be embedded in a global procedure such as the following: (cf. 8.1)

```
GLOBAL PROCEDURE SEGN001 (R14) BASE R15;
BEGIN  STM(R14,R12,B13(12));  R14 := R13;
   BEGIN  GLOBAL DATA SEGN000 BASE R13;
             ARRAY 18 INTEGER B13;
      B13(4) := R14;  B14(8) := R13;
      B14(16) := B14(16) XOR B14(16);

      BEGIN  COMMENT  Main program block;
      END;

      R13 := B13(4);  LM(R14,R12,B13(12));
   END;
END.
```

The 18 integer area is reserved to conform to procedure calling conventions (cf. 9.1). If the PL360 program is a global procedure, there is no implicit base declaration for the data area (cf. 4.3).

When a program is defined as a block, the compiler supplies a transfer address for the linkage editor or loader [9], and provides the necessary entry and exit code for linking with a standard operating system (cf. 10.1.3).

When a program is defined as a global procedure, no transfer address is supplied, and all linkage code must be written by the programmer.

Both types of program are included in the sample programs of Appendix A.

SECTION 5. DECLARATIONS

5.1 Register Synonym Declarations

The System/360 processor has 16 registers which contain integer numbers and are said to be of type integer (or logical). They are designated by the standard register identifiers: R0 through R15 (cf. 2.2.3).

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 real, 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 other identifiers can be associated with these registers. Reference to specific registers in the text apply to register synonyms also.

```
<K-register synonym> ::=
     <simple K-type> REGISTER <identifier> SYN <K-register> !
     <K-register synonym> , <identifier> SYN <K-register>
```

5.2 Seqment Base Declarations

```
<segment base declaration> ::=
     <segment base heading> BASE <integer register>
<segment base heading> ::= SEGMENT ! GLOBAL DATA <identifier> !
     EXTERNAL DATA <identifier> ! COMMON DATA <identifier> !
     COMMON ! DUMMY
<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. The segment is terminated either by the END of 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 except for the dummy base declaration and base declarations that specify BASE R0 (cf. 4.3).

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. 10.4).

Note: Dummy base declarations permit the description of data areas which are created during the execution of the PL360 program. Any integer register may be specified in a dummy base declaration. When R0 (or a synonym to R0) is specified in any base declaration, the subsequent identifiers are understood to have displacements and no base register (or index register).

5.3 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 belonging to the currently active base declaration segment (cf. 4.3). The scope of validity of these cell identifiers is the block in whose heading the declaration occurs (cf. 4.1). 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.

A cell may be initialized to numeric values, strings, relative or absolute addresses. The number of bytes appropriate for the type of the declared cell is taken for each (numeric) 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 to 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 <= 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 = @basepoint
    LONG REAL x,y,z = 27'3L
    ARRAY 3 INTEGER size = (36,23,37),parmlist = (@@a,@@b,@@erproc)
    ARRAY 132 BYTE blank = 132(" "),buff = 33(" ",2("*")," ")
    ARRAY fbsize LOGICAL area = fbsize(0)
```

5.4 Cell Designators

```
    <T-cell designator> ::= <T-cell identifier>
        ! <T-cell identifier> ( <index> / <integer value expression> )
        ! <T-cell identifier> ( <index> )
    <index> ::= <integer value expression>
        ! <integer register expression>
        ! <integer register expression> + <integer value expression>
        ! <integer register expression> - <integer value expression>
    <integer register expression> ::= <integer register>
        ! <integer register> + <integer register>
    <integer value expression> ::= <integer value>
        ! <integer value expression> + <integer value>
        ! <integer value expression> - <integer value>
```

Note: The second form of <T-cell designator> may be specified at any time, but only has meaning for the first <T-cell designator> of

```
    <T-cell assignment> ::= <T-cell designator> := <T-cell value>
```

and

```
    <condition> ::= <T-cell designator> <relation> <T-cell value>
```

In these two cases, the integer value expression following the slash (/) specifies the number of bytes to be moved or compared (cf. 6.3, 6.5).

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 or index/length. 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. If a length is to be specified when no index is required, an index value of 0 must be specified before the slash; e.g., cell(0/length).

Note: Register R0 or synonym (cf 5.1) 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)
        val(0/20)           status(R1/len-1)
```

5.5 Cell Synonym Declarations

```
    <T-cell synonym> ::=
          <T-type><identifier><synonymous cell> !
          <T-cell synonym> , <identifier><synonymous cell>
    <synonymous cell> ::= SYN <T-cell designator> ! SYN <integer value>
```

Cell synonyms serve to associate synonymous identifiers with previously (i.e., preceding in the text) declared cells. 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 = #4E0000000000000L
    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 := F01 - F01 + x;
```

and a conversion vice-versa by

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

No initialization can be achieved by a synonym declaration.

5.6 EQUATE Declarations

```
<integer value synonym> ::=
     EQUATE <identifier><synonymous integer value> !
     EQUATE <identifier> SYN <string> !
     EQUATE <identifier> SYN <register name> !
     <integer value synonym>,<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> ::= <T-cell designator> - <T-cell designator>
```

Integer value synonyms 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, 3.3). When the difference of two cell designators is specified, the cell identifiers must both have the same base register (cf. 5.2). 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. 4.1).

See sections 6.1 and 6.2 for definitions of monadic, arithmetic, logical and shift operators.

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

SECTION 6. STATEMENTS

6.1 Register Assignments

```
<T-primary> ::= <T-value> ! <T-cell designator>
<K-primary> ::= <K-register>
```

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

```
<simple K-register assignment> ::=
     <K-register> := <A-primary> !
     <K-register> := <monadic operator><A-primary> !
     <integer register> := <string> !
     <integer register> := @ <T-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 taking the absolute value, sign inversion, and 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 (#00X) are appended at the left of the string. The bit representation of characters is defined in EBCDIC [4,5,6].

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 K and A in preceding and subsequent rules of this section are given in Table 6.1.

K	A
integer	integer
integer	short integer
real	real
long real	real
long real	long real

Table 6.1 - Allowable Cell and Register Type Combinations

Examples of simple register assignments:

```
R0  := i                R2  := "xyz"
R2  := R10              F45 := NEG F01
R6  := age              R13 := ABS height
F0  := quant(R1)
```

6.2 Register Assignment Expressions

```
<K-register assignment> ::= <simple K-register assignment>
    ! <K-register assignment> <arithmetic operator> <A-primary>
    ! <K-register assignment> =: <K-register>
    ! <K-register assignment> =: <A-cell designator>
    ! <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. The reverse-assignment operator (=:) specifies that the contents of the assigned register are to be stored in the register or cell following the operator.

Examples of register assignments:

```
R0  := R3
R1  := 10 * x =: x
R10 := i + age - R3 AND size(8)
R9  := R8 AND R7 SHLL 8 OR R6
F2  := 3.1416
F0  := quant(R1) * price(R1)
F45 := F23 + F01
```

Note: 1. The syntax implies that sequences of operators, including assignment, are executed strictly 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 participates in the division and contains the
              remainder.
```

6.3 Cell Assignments

```
<T-cell assignment> ::= <A-cell designator> := <K-register>
                      ! <T-cell designator> := <T-cell value>
                      ! <T-cell assignment> <logical operator> <T-cell value>
<T-cell value> ::= <T-cell designator>
                 ! <T-value>
                 ! <string>
```

In the first assignment, the value in the K-register is assigned to the designated A-cell. The allowable combinations of cell and register types are indicated in Table 6.1. Cells may be indexed.

In the second assignment, the T-cell, T-value or string is assigned to the designated T-cell. The third form is a logical assignment expression in which the assigned cell is logically combined with the specified T-cell, T-value or string. Cell designations must not include an index register (cf. 5.4). For cell to cell, the cell types must be identical or the assigned cell must include a length specification (cf. 5.4). For string to cell, the entire string is moved to or logically combined with the assigned cell regardless of cell type when a length is not specified, or the shorter of the string length or specified length is used. For value to cell, the allowable combinations of cell and value are indicated in the following table:

Note: Length specifications should not be used for value to cell.

T-cell	T-value
long real	long real
real	real, integer
integer	integer, real
short integer	integer*, short integer
byte	integer*, short integer*, byte

Table 6.2 - Allowable Cell and Value Combinations

* unused portions of the T-value must be all 0 or all 1 bits (sign).

Examples of cell assignments:

```
i := R0
price(R1) := F0
x := F67
price(R1) := price(R2)
y := 30
j := "A"
z(0/5) := z XOR z(5)
```

6.4 GOTO Statements and Labels

```
<GOTO statement> ::= GOTO <identifier>
```

The interpretation of a GOTO statement proceeds with 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.
3. Otherwise, execution of the block is regarded as terminated and the innermost block surrounding it is considered.
4. If this block is in the same program segment as the previous blocks, then step 2 is repeated.
5. Otherwise, the identifier is undefined (cf. 4.2).

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. GOTO statements may refer to such points (cf. 4.1). The identifier can be chosen freely, with the restriction that no two points in the same block may be designated by the same identifier.

6.5 Conditions and Compound Conditions

```
<condition> ::= <T-cell designator> <relation> <T-cell value>
              ! <byte cell designator>
              ! ^ <byte cell designator>
              ! <K-register> <relation> <A-primary>
              ! <integer register> <relation> <string>
              ! <relation>
              ! <integer value>
              ! ^ <integer value>
<relation> ::=  = ! ^= ! < ! <= ! >= ! >
```

A condition is said to be met or not met. In the first condition, the T-cell preceding the relation is compared to the T-cell, T-value, or string specified after the relation. The comparison is logical (unsigned). The condition is met if the specified relation holds between the values of the compared quantities. The same restrictions apply regarding combinations allowable as apply to the second form of T-cell assignment (cf. Table 6.2). A condition specified as a byte cell (or a byte cell preceded by ^) is met if the value of the byte cell is #FF (or not #FF). The condition consisting of a relation enclosed by a register and a primary is met if the specified relation holds between the current values of the register and the primary. When an integer register is compared to a string, the comparison is logical (unsigned), and the string must consist of not more than four characters. If it consists of fewer than four characters, the string is right justified and null characters (#00X) are prefixed at the left to form a four character string. The condition is met if the specified relation holds between the register and the string. A condition consisting of only a relation is met if the condition code of the processor (cf. 2.1.1) is in a state specified by the symbols of Table 6.3 on the following page. A condition consisting of an integer value (or an integer value preceded by ^) is met if the condition code of the processor is in a state (or not in a state) specified by summing the integer components from Table 6.3 to arrive at the specified integer

value. Table 6.3 also contains predeclared integer value identifiers which may be used as specified (or preceded by ^ to obtain all states except the specified state).

```
<compound condition> ::= <combined condition>
                       ! <alternative condition>
<combined condition> ::= <stat condition>
                       ! <combined condition> AND <stat condition>
<alternative condition> ::= <stat condition>
                          ! <alternative condition> OR <stat condition>
<stat condition> ::= <condition>
                   ! <statement> ; <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 the constituent conditions are met, or of the form 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. Note that each condition may be prefaced by a statement and semi-colon. In such a case, the statement is done before the associated condition is tested.

identifier	state
overflow	3
on	3
off	0
mixed	1
carry	1
integer component	**state**
8	0
4	1
2	2
1	3
symbol	**state**
=	0
^ =	1 or 2
<	1
< =	0 or 1
> =	0 or 2
>	2

Table 6.3 - Condition Code States

6.6 IF Statements

```
<IF statement> ::= <if clause> <statement>
     ! <if clause> <true part> <statement>
<if clause> ::= IF <compound condition> THEN
<true part> ::= <simple statement> ELSE
```

The IF statement specifies the conditional execution of statements:

```
<if clause> <statement>
```

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

```
<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. A simple statement is any statement except an IF, WHILE or FOR statement.

```
Examples:   IF R0 < 10 THEN R1 := 1
            IF F2 > _3.75 AND F2 < 3.75 THEN F0 := F2 ELSE F0 := 0R
            IF < THEN SET(flags(1)) ELSE SET(flags(2))
```

Note: If the condition consists of just a relation or integer value, then the decision is made on the basis of the condition code as determined by a previous instructions.

```
Examples:  EX(R1,CLC(0,B2,B3));  IF = THEN ...
           IF  TM(#80,flags);  ON  THEN ...
```

6.7 WHILE Statements

```
<WHILE statement> ::= <while clause><statement>
<while clause> ::= WHILE <compound condition> DO
```

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
```

6.8 FOR Statements

```
<FOR statement> ::= <for clause><statement>
<for clause> ::= FOR <integer register assignment> STEP <increment>
                 UNTIL <limit> DO
<increment> ::= <integer value>
<limit> ::= <integer primary> ! <short integer primary>
```

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(R1,lines(R1))
           FOR R2 := R1 STEP 4 UNTIL R0 DO
                  BEGIN F23 := quant(R2) * price(R2);
                      F01 := F01 + F23;
                  END
```

6.9 CASE Statements

```
<CASE statement> ::= <case sequence> END
<case sequence> ::= <case clause> BEGIN !
      <case sequence><statement> ;
<case clause> ::= CASE <integer register> OF
```

CASE statements permit the selection of one of a sequence of statements according to the current value of the integer register (other than register R0) 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;
              F01 := F01 + F23;
              F01 := F01 - F23;
              F01 := F01 * F23;
              F01 := F01 / F23;
              F01 := NEG F01;
              F01 := ABS F01;
          END
```

SECTION 7. FUNCTIONS

7.1 Function Declarations

```
<function declaration> ::= FUNCTION <function definition> !
     <function declaration> , <function definition>
<function definition> ::=
     <identifier> ( <format code> , <instruction code> )
<instruction code> ::= <integer value>
<format code> ::= <integer value>
```

Various data manipulation facilities in the 360 computer cannot be expressed by an assignment. To make these facilities available in the language, the function statement is introduced (cf. 7.2), using 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 7.1 on the following page. 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 derived from the symbolic machine code used in assembly language [7].

```
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)
```

7.2 Function Statements

```
<function designator> ::= <function identifier> !
       <function identifier> ( <parameter list> )
<parameter list> ::= <parameter> ! <parameter list> , <parameter>
<parameter> ::= <T-value> ! <string> ! <K-register> !
       <T-cell designator> ! <function designator>
```

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. 7.1).

Examples:

```
    SET(flag)                STM(R0,R15,save)
    RESET(flag)              SVC(255)
    LA(R1,"message")         IC(R0,flags(R1))
    UNPK(3,7,B2,worker)      EX(R1,MVC(0,lines,buffer))
```

Format Code	Number of Parameters	Instruction Fields 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 = K-register
- C = 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 = T-value or string or function designator (the address of the value is used in the instruction field)

Table 7.1 - Instruction Format

SECTION 8. PROCEDURES

8.1 Procedure Declarations

```
<procedure declaration> ::= <procedure heading> ; <statement>
<procedure heading> ::= <simple procedure heading> !
      COMMON <simple procedure heading> !
      <separate procedure heading> !
      <separate procedure heading> BASE <integer register>
<separate procedure heading> ::=
      SEGMENT <simple procedure heading> !
      GLOBAL <simple procedure heading> !
      EXTERNAL <simple procedure heading>
<simple procedure heading> ::=
      PROCEDURE <identifier> ( <integer register> )
```

A procedure declaration serves to associate an identifier, which thereby becomes a procedure identifier, with a statement (cf. 4.1) 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. 4.2, 6.4 and 10.4). 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. 4.2); otherwise, the current program base register is used (usually this is R15, however, cf. 10.1.5). This register must not be R0. When the procedure is invoked, the specified (or assumed) base register is assigned the entry point address.

The instructions associated with the statement of both a simple PROCEDURE and COMMON PROCEDURE are local to the program segment containing these procedure declarations. However, a COMMON PROCEDURE also declares the procedure identifier as an additional entry point to the program segment. Such entry points are normally called upon from separately compiled programs through an EXTERNAL PROCEDURE declaration.

Examples:

```
      PROCEDURE NEXTCHAR(R3);
      BEGIN IF R5 < 71 THEN R5 := R5 + 1 ELSE
           BEGIN R0 := @CARDS;  READ;  R5 := R5--R5;
           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 parenthesis 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.

8.2 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 the invoked procedure's program base register (usually R15) are transferred to the specified integer register and the condition code is set by the transfer. This facilitates the convention of passing return codes in the invoked procedure's program base register (usually R15, cf. 8.1, 10.1.5).

SECTION 9. THE RUN-TIME LIBRARY

This section describes a set of global procedures written in PL360 which perform commonly needed tasks. These subroutines are predeclared as external procedures in the PL360 compiler. 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 cataloged procedure PL360CG.

9.1 Standard Procedures

A set of standard procedures is defined for elementary unit record input and output operations (the first set below), for elementary disk and tape input and output operations using sequential files (the second set), and for ease in communicating with the operating system (the last). The implicit procedure declarations are as follow:

```
EXTERNAL PROCEDURE READ (R14) BASE R15;  NULL;
EXTERNAL PROCEDURE WRITE (R14) BASE R15;  NULL;
EXTERNAL PROCEDURE PAGE (R14) BASE R15;  NULL;
EXTERNAL PROCEDURE PUNCH (R14) BASE R15;  NULL;
EXTERNAL PROCEDURE PRINT (R14) BASE R15;  NULL;

EXTERNAL PROCEDURE OPEN(R14) BASE R15; NULL;
EXTERNAL PROCEDURE GET(R14) BASE R15; NULL;
EXTERNAL PROCEDURE PUT(R14) BASE R15; NULL;
EXTERNAL PROCEDURE KLOSE(R14) BASE R15; NULL;

EXTERNAL PROCEDURE CANCEL(R14) BASE R15; 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. (ABEND 95 or 96)
- **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. (ABEND 95)
- **PAGE** Give the next output record transmitted by a WRITE to the system listing data set 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. (ABEND 95)
- **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. (ABEND 95)

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. (ABEND 97)

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 DFT 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) <= 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.

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. 4.4) 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.

9.2 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. 3). 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 terminated by a blank. These procedures are predeclared in a manner similar to those described in Section 9.1.

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 R0, F0, F01, F0 and F2, or F01 and F23, depending on the type (in that order).

A return code is left in R15:
- 0 -> successful conversion
- 1 -> field size too small
- 2 -> invalid field size

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

All registers, except R14 and 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 VALTOBCD)

The resulting value is left in 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 terminator
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 terminator. Registers R2-R13 are restored.

9.3 Data Manipulation Procedures

The first procedure described in this section does an in-core indirect sort using logical comparisons. The second companion routine searches a sorted list for a specified element. Neither procedure is predeclared.

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

```
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 address 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, except R14, are restored.

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

```
R0 = the number of entries in the sorted table
R1 = the address of the index array (see SHELSORT)
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 and R14, are preserved.

SECTION 10. COMPILER CONTROL FACILITY

10.1 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 2-72.

10.1.1 Listing Control

$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.

$STITLE This directive provides a sub-title line. The sub-title will remain in effect until the next $TITLE or $STITLE card. $STITLE cards may change the sub-title without affecting the main $TITLE. $STITLE also causes a page eject.

$SPACE # This directive allows the user to line space a listing by # lines where # is a number from 1 to 99. If # is blank, then a single line space is assumed. If the number of lines remaining on the page is less than #, then a page eject is done instead of line spacing.

$EJECT This directive is equivalent to $PAGE.

$ON This directive enables the printing of all $-control cards except $TITLE, $STITLE, $EJECT, $PAGE, and $SPACE.

$OFF This directive disables the printing of all $-control cards. This is the default condition at the start of compilation.

10.1.2 Listing Options

$XREF All subsequent instances of identifiers are listed in an alphabetical cross-reference listing together with the line numbers at which they are defined or referenced in the source program. The cross-reference listing follows the program listing if $LIST is in effect at the end of the program. If there is not enough free storage to allocate the cross-reference tables, the $XREF instruction is ignored. The cross-reference listing will be single spaced unless $XREF 2 is specified to double space the listing.

$NOXREF This causes the previous option to be turned off (initial option). Any accumulated cross-references will be listed following the program as described above for $XREF.

$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 declared identifiers and associated value as each 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.

10.1.3 Operating System Control

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

10.1.4 Identification

$XYY# This directive must precede the first non-control card. All compiler generated segment names will commence with XYY rather than SEG, and all object deck cards are identified by XYY in columns 73 through 75 followed by the letter N and a four digit number. X signifies any alphabetic and Y any alphanumeric characters. (cf. 10.4).

10.1.5 Program Base Register Control

$BASE=xx This directive must precede the first non-control card. Program segments following this directive are compiled with xx taken as the program base register. This includes main programs, global procedures, segment procedures, and external procedures (which do not specify BASE). 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, and all predeclared external procedure declarations always have base register R15. It is recommended that this compiler directive only be used for programs which make use of SVC instructions that do not preserve the contents of register R15.

10.1.6 Object Deck Control

$GEN If this directive precedes the first error detected (if any), then object decks are still produced if any have been requested. Otherwise object decks are suppressed after encountering an error.

$NOGO Compile, but suppress the GO step.

10.1.7 Copy Facility

$COPY ddname
$COPY ddname(member)

These control cards specify that a sequential data set or member of a partitioned data set is to be copied into the compilation. The compiler temporarily suspends input from the standard input medium and continues compilation with the data set defined by the $COPY control card. When end-of-information is encountered on that data set, compilation continues from the standard input with the card image immediately following the $COPY control card. Note: $COPY is ignored in the data set being copied, i.e., $COPY may not nest. As many $COPY control cards as desired may occur in the standard input. When compiling under ORVYL, ddname or member is assumed to be the ORVYL data set name. An account number may follow to indicate a data set belonging to a different account.

10.1.8 Conditional Compile Directives

At the start of compilation of each program (cf. 4.1), an array of flags is reset by the compiler. The following directives use this array. The array flags are specified by individual characters in the directives, and any characters may be used, including blank. Upper and lowercase characters are considered equivalent. The directives must be in uppercase in columns 1 through 4 on the control card.

$SET a where 'a' is any character in column 6.
This directive sets the 'a' flag.

$IFT a b
$IFF a b

where 'a' is any character in column 6, and 'b' is any character in column 8.
These directives examine the 'a' flag. If the 'a' flag is set for $IFT, or reset for $IFF, this directive takes no action and compilation continues normally.

If the 'a' flag is reset for $IFT, or set for $IFF, the compiler skip-reads source cards until a $END directive is encountered with its 'b' character matching the 'b' character of the $IFT or $IFF. Compilation then continues from that point.

Note: '$IFF a b' is an unconditional skip to '$END b' if '$SET a' has occurred. '$IFT a b' is an unconditional skip to '$END b' if '$SET a' has not occurred.

$END b where 'b' is any character in column 6.
This directive terminates $IFT or $IFF directives.

```
$RESET a    where 'a' is any character in column 8.
            This directive resets the 'a' flag.
```

Examples of Conditional Compile:

```
1.  $SET Z
       .
       .
    $IFT Z
       COMMENT Compile this if 'Z' is $SET;
       .
       .
    $END
    $IFF Z
       COMMENT Compile this if 'Z' is not $SET;
       .
       .
    $END

2.  $SET 1
       .
       .
    $IFF 0 X
    $IFF 1 X
    $IFT 2 Q
    $END X
       COMMENT Compile this if '0' or '1' or '2' is $SET;
       .
       .
    $END Q

3.  $SET -
    $SET +
       .
       .
    $IFT +
    $IFT -
       COMMENT Compile this if both '+' and '-' are $SET;
       .
       .
    $END
```

10.2 Compiler Listing Output

If listing is specified, each non-control 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 (in decimal) and the data relative address (in hexadecimal). The fifth number is the statement number of the source record. The final number, the BEGIN/END level count, shows the excess of BEGIN symbols over END symbols at the beginning of the next line following an occurrence of BEGIN/END. This count is only printed when the BEGIN/END level changes. In addition, each page begins with a heading which includes the page number, date, time, and an optional title (cf. 10.1.1). Examples of compiler listings are given in Appendix A.

10.3 Error Messages of the Compiler

Errors detected by the compiler are indicated by a message and a vertical bar below the point where the error was detected. After about 50 errors, a message is provided, and further diagnostic messages are counted but not listed. The following is a list of error diagnostics and their meanings:

Error Number	Message	Meaning
00	SYNTAX	The source program violates the PL360 syntax.
01	VAR MIX 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, or the limit is not a register, cell, or number of the integer types.
03	REG ASS TYPES	The types of the 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 has been referenced. 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 the number of elements declared in an array, or a string attempts to initialize beyond the declared limits of a variable or 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.
l9	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 T-cell designators have different base registers.
27	SEG NO OFLOW	The maximum allowed segment numbers has been exceeded. The limit is generally set at 255.
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.

10.4 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 [8], the IBM OS linkage editor or loader [9], and the IBM DOS linkage editor [11]. 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] and 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 labeled 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 for SEGMENT declarations 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. 10.1.4).

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

SECTION 11. 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.

11.1 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 NULL. 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 *,l
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 l = 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. 4.4), and by the @@ operator which facilitates the construction of OS-type parameter lists at compile time.

11.2 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. 10.1.5), or by explicitly specifying a base register in the procedure heading (cf. 8.1).

11.3 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. 8.1, 10.1.5) 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 block 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. 10.1.4);
    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. 4.4, 10.1.3)
    d. at exit, all registers are restored from this save area with R15 set equal to zero (R15=0)

2. Global and external names violate the rules of scope established by the PL360 block structure (cf. 2.2.1, 4). By pairing global and external declarations, a name can be given arbitrary scope. Recursive procedures and co-routines can be programmed using this feature; however, this ability should be used carefully and sparingly.

Consider the following example:

```
GLOBAL PROCEDURE p1 (R1) BASE R7;
   BEGIN GLOBAL DATA d1 BASE R10;
      INTEGER a;
      COMMON PROCEDURE p2 (R2);
      BEGIN  R0 := a;
      END;
      COMMON 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.

Also note that when common and external procedures are paired, return registers must be identical and any base register specified in the external declaration must match the base register of the global or segment procedure containing the common procedure declaration. Thus,

```
    EXTERNAL PROCEDURE p2 (R2) BASE R7; NULL;
```

would be the proper declaration for p2 in a separately compiled segment considering the above example.

SECTION 12. PL360 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 [12]. This version is made possible through a special ORVYL-PL360 interface module written in Assembly Language using the ORVYL macro instructions [12].

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 as an ORVYL unload file. To use it, just type:

```
? PL360
```

You will then receive the message:

```
-WELCOME TO PL360, Joe User
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 should place the object modules from the subsequent compilation. This file can be either new or old. Appending " SCR" to the file name will cause an old file to be scratched for reuse; otherwise, you will be prompted:

```
SCRATCH?
```

A "NO" response will cause the file naming process to be repeated. The next thing PL360 asks is:

```
LISTING?
```

If you respond "YES", then you will again be asked to supply an ORVYL file to receive the PL360 Compiler list output. PL360 then asks:

```
-?
```

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, collect lines, edit lines, use files, copy files, etc.) until your WYLBUR working data set contains the PL360 program(s). You then type "COMPILE" immediately after a -? prompt and PL360 will begin compiling the program(s) contained in your WYLBUR working data set.

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

You can type in a program directly by responding "COMPILE X" to a -? prompt, where X represents any non-blank character. PL360 responds:

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

You can now type in 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. Also, leading blanks are stripped from each input line. For these reasons, it is usually best to compile from a WYLBUR working data set. When typing the program in directly, you can leave PL360 at any time by typing "/*" or by simply hitting the ATTN button at the terminal.

As you are leaving PL360, ORVYL core memory is automatically cleared. The WYLBUR working data set is not cleared. 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 (except in response to a prompt). PL360 will then ask:

```
DO YOU WANT FURTHER ERROR MESSAGES TYPED?
```

A "NO" will cause the compilation to continue with no further error messages typed at the terminal. A "YES" will cause compilation to continue as before. In either case, the listing produced in the ORVYL file (if any) will be unaffected.

After leaving PL360, you can retrieve the object deck by typing:

```
GET <file name> CARD CLEAR
```

You can retrieve 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 line printer listing looks like a batch PL360 compilation listing. The ORVYL version of PL360 has several advantages: Waiting for the batch queue is completely eliminated. Errors are printed at the terminal, and thus can usually be fixed immediately and another compilation can be made in a minute or two. Paper is saved since listings with errors are seldom listed offline. Finally, the ORVYL version of the runtime library 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 3000 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

The standard input-output subroutines using the same linkage conventions as the READ and WRITE subroutines described in Section 9.1 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 input area 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 the attention key is typed with no preceding characters, the condition code is set to 2, otherwise it is set to 0.

WRITE This subroutine works exactly like the subroutine described in Section 9.1; i.e., the address of a 132 byte output area is passed through register R0 and all registers are preserved upon return. The output area is typed at the terminal.

The following discussion assumes that the ORVYL system is being used at Stanford where the ORVYL READ and WRITE subroutines and the library subroutines listed in Section 9 are stored in object module form in the WYLBUR file T000.PL360.RUNLIB on SYS10. To run a PL360 program in ORVYL, just follow this simple process. First, compile the program. This may be achieved either in batch or with the ORVYL version of the PL360 compiler. The program must be a statement with segment name SEGN001 (cf. Section 4). Place the object module output of the PL360 compiler in the WYLBUR working data set and type:

```
COPY ALL TO END FROM &T000.PL360.RUNLIB ON SYS10
LOAD TEXT
```

Your program is now ready to be executed. You could either unload the program as an ORVYL UNLOAD file and/or type the command ENTER to begin execution.

Note that file I/O is not provided for in the ORVYL runtime routines.

APPENDIX A. EXAMPLE PROGRAMS AND LISTINGS

See Section 10 for details and descriptions of compiler control.

```
PL360 COMPILATION            Sample Program Demonstrating Extensions to PL360              ORVYL 10/28/74    PAGE   1
                                $STITLE card to provide a sub-title   (Koenig)
                                   $GEN     Generate object decks even on errors.    (Guertin)                    6.
                                   $BASE=12 Specifies default program base register  (Malcolm)                   7.
                                   $SET a   Sets condition 'a'                       (Guertin)                    8.
                                   $IFF b 1   Conditional compile test               (Guertin)                    9.
                                   $IFT a 2   Continue compiling if condition 'a' was $SET                       10.
                                   $END 1     End of conditional range 1             (Guertin)                   11.
001 0000   000 0000        0001              ! Vertical bar comments                  (Guertin) !                 12.
001 0000   000 0000        0002              ! These comments compiled if 'a' or 'b' was $SET   !                 13.
                                   $SET b   Sets condition 'b'                                                   14.
                                   $END 2     End of conditional range 2                                         15.
                                   $RESET a   Turn off condition 'a'                 (Koenig)                    16.
                                   $XREF 1  Turn on Cross-Reference capability       (Guertin)                   17.
001 0000   000 0000        0003                                                                                  18.
001 0000   000 0000        0004      BEGIN  !coding includes establishing proper base register (Malcolm)!       19.
001 001A   000 0048        0005 01   !coding includes XC instruction to set saved R15 to 0     (Koenig) !       20.
                                   $3                                                                          21.
001 001A   000 0048        0006      !EQUATE declarations for integer value identifiers        (Guertin)!       22.
001 001A   000 0048        0007         EQUATE LEN SYN 3;                                                      23.
         00000003  LEN
001 001A   000 0048        0008         ARRAY LEN BYTE ALPHA;                                                  24.
             D048  ALPHA
001 001A   000 004B        0009         INTEGER BETA, GAMMA;                                                   25.
             D04C  BETA
             D050  GAMMA
001 001A   000 0054        0010      !EQUATES allows monadic, arithmetic, Boolean and shift ops(Guertin)!       26.
001 001A   000 0054        0011         EQUATE LO SYN 9,  LOMASK SYN 1 SHLL LO - 1,                            27.
         00000009  LO
001 001A   000 0054        0012                HI SYN NEG LO + 16,  HIMASK SYN 1 SHLL HI - 1;                  28.
         000001FF  LOMASK
         00000007  HI
         0000007F  HIMASK
001 001A   000 0054        0013      !EQUATES allows cell-cell                                 (Guertin)!       29.
001 001A   000 0054        0014         EQUATE SIZE SYN GAMMA(4) - ALPHA + 3 AND _4;                           30.
         0000000C  SIZE
001 001A   000 0054        0015                                                                                  31.
001 001A   000 0054        0016      !Data Segment Declarations with BASE R0       (Guertin & Koenig)   !       32.
001 001A   000 0054        0017         DUMMY BASE R0;                                          !(Guertin)!       33.
001 001A   014 0000        0018            INTEGER MEM;                                                        34.
             0000  MEM
001 001A   014 0004        0019         CLOSE BASE;                                                            35.
001 001A   000 0054        0020                                                                                36.
001 001A   000 0054        0021         GLOBAL DATA SPACE BASE R0;                              !(Koenig) !       37.
001 001A   015 0000        0022            INTEGER SOMECELL;                                                   38.
             0000  SOMECELL
001 001A   015 0004        0023         CLOSE BASE;                                                            39.
                              SPACE     ENTRY (SD) AT 0000
```

```
001 001A   000 0054        0025         GLOBAL PROCEDURE SAMPLE (R15) BASE R6;                                    42.
                SAMPLE
016 0000   000 0054        0026         BEGIN                                                                     43.
016 0000   000 0054        0027 02    !COMMON procedures to allow additional entry points.      (Guertin)!      44.
016 0000   000 0054        0028             COMMON PROCEDURE ENTRY (R15);                                        45.
                ENTRY
016 0008   000 0054        0029             BEGIN  !has all the properties of local procedure.!                  46.
016 0008   000 0054        0030 03    !New FUNCTION types 14 & 15.  See compiled code.          (Guertin)!      47.
016 0008   000 0054        0031                 FUNCTION  MOVE4(14,#D203),  BRANCH(15,#47F0);                    48.
            D203  MOVE4
            47F0  BRANCH
016 0008   000 0054        0032                 MOVE4(GAMMA,BETA);  BRANCH(B6(R4+10));                           49.
016 0012   000 0054        0033                 R6 := R6-R6;  !return condition in BASE!        !(Guertin)!      50.
016 0014   000 0054        0034             END;                                                                 51.
016 0016   000 0054        0035 02          ENTRY(R3);  !See code for BAL to ENTRY, condition to R3!             52.
016 0022   000 0054        0036         END;                                                                     53.
              0020    600607FF  00000000
                         SAMPLE    ENTRY (SD) AT 0000
                         ENTRY     ENTRY (LD) AT 0004
001 001A   000 0054        0038         EXTERNAL PROCEDURE ENTRY (R15) BASE R6;  NULL;        !(Malcolm)!      55.
                ENTRY
001 001A   000 0054        0039         ENTRY;    !See code for  BALR  to COMMON procedure.    (Guertin)!      56.
001 0024   000 0054        0040                                                                                  57.
001 0024   000 0054        0041       !Cell references may specify Base, Index, and expression  (Guertin)!      58.
001 0024   000 0054        0042         BEGIN  INTEGER REGISTER RB SYN R1,  RX SYN R2;                          59.
            0001  RB
            0002  RX
001 0024   000 0054        0043 02          SOMECELL(RB+RX+LEN+1) := R3;                                        60.
001 0028   000 0054        0044         END;                                                                     61.
001 0028   000 0054        0045 01                                                                               62.
001 0028   000 0054        0046       !Cell-to-cell assignment and conditions.                  (Guertin)!      63.
001 0028   000 0054        0047         B1 := B2;   B3 := 16;   B5 := "string";                                 64.
001 003A   000 0054        0048         WHILE R2 >= R1 AND B2 = " " DO                                          65.
001 0048   000 0054        0049         BEGIN  FUNCTION REDUCE(6,#0600);  REDUCE(R2);  END;                     66.
            0600  REDUCE
001 004E   000 0054        0050 01      IF GAMMA ^= BETA THEN GOTO TAG;                                         67.
001 0058   000 0054        0051                                                                                  68.
001 0058   000 0054        0052       !Cell-to-cell assignment allows Boolean expression        (Guertin)!      69.
001 0058   000 0054        0053         BETA := BETA AND GAMMA OR "    ";                                      70.
001 0064   000 0054        0054                                                                                  71.
001 0064   000 0054        0055       !Cell-to-cell may specify length expression after /       (Guertin)!      72.
001 0064   000 0054        0056         GAMMA(4-LEN/LEN) := GAMMA(4-LEN) AND BETA(4-LEN) OR ALPHA;              73.
001 0070   000 0054        0057         ALPHA := " ";  ALPHA(1/LEN-1) := ALPHA;                                 74.
```

```
001 007A   000 0054        0059        WHILE R1 := R1+R2; R1 < 16 AND READ; ^= DO R2 := R2-4;                77.
001 009A   000 0054        0060                                                                              78.
001 009A   000 0054        0061      !Pre-declared EQUATES for use with assignments and tests. (Koenig) !     79.
001 009A   000 0054        0062        BEGIN  EQUATE OVERFLOW SYN 1,  ON SYN 1,  MIXED SYN 4,  OFF SYN 8,    80.
         00000001  OVERFLOW
         00000001  ON
         00000004  MIXED
001 009A   000 0054        0063 02                   CARRY SYN 3,  TRUE SYN _1,  FALSE SYN 0;                81.
         00000008  OFF
         00000003  CARRY
         FFFFFFFF  TRUE
         00000000  FALSE
001 009A   000 0054        0064                                                                              82.
001 009A   000 0054        0065      !<condition> allowed to be <integer> or ^<integer>        (Koenig) !    83.
001 009A   000 0054        0066            IF EX(R4,TM(0,B1)); MIXED THEN GOTO TAG;                          84.
001 00A2   000 0054        0067            IF ^ON THEN GOTO TAG;                                             85.
001 00A6   000 0054        0068            ALPHA := TRUE;  IF ALPHA(1) = FALSE THEN GOTO TAG;                86.
001 00B2   000 0054        0069          END;                                                                87.
001 00B2   000 0054        0070 01                                                                           88.
001 00B2   000 0054        0071      !$SPACE control card to space listing                   (Koenig) !    89.
001 00B2   000 0054        0073          BETA(1) := #4096CX;  GAMMA := #40202120X;                           92.
001 00BE   000 0054        0074                                                                              93.
001 00BE   000 0054        0075      !STRING equate which has length of last "string"        (Guertin)!    94.
001 00BE   000 0054        0076          LA(R1,"This is a test.");  R2 := STRING;                            95.
001 00C6   000 0054        0077          IF R3 = 0 THEN B1(STRING-5) := "mess";                              96.
001 00D2   000 0054        0078                                                                              97.
001 00D2   000 0054        0079      !CASE statement now uses halfword vector table. (Malcolm & Guertin)!    98.
001 00D2   000 0054        0080          CASE R1 OF BEGIN   !See compiled code.!                             99.
001 00D2   000 0054        0081 02           R5 := R5 + R3;                                                 100.
001 00DE   000 0054        0082              R5 := R5 - R3;                                                 101.
001 00E4   000 0054        0083              R5 := R5 * R3;                                                 102.
001 00EA   000 0054        0084              R5 := R5 / R3;                                                 103.
001 00F0   000 0054        0085          END;               !See compiled code.!                            104.
001 00FC   000 0054        0086 01                                                                          105.
                                     $COPY DDNAME          Copy from sequential data set       (Guertin)   106.
001 00FC   000 0054        0087        !This is copy code from a WYLBUR Edit format data set.!               $COPY
001 00FC   000 0054        0088        ! //DDNAME DD DSN=SYS1.DDNAME,VOL=SER=TEMP01,UNIT=2314,     !      $COPY
001 00FC   000 0054        0089        ! //  DISP=OLD,DCB=(RECFM=U,BLKSIZE=3156)                  !      $COPY
001 00FC   000 0054        0090        !This is the end of the $COPY DDNAME!                               $COPY
                                     $COPY DATA(MEMBER)    Copy from partitioned data set      (Koenig)    107.
001 00FC   000 0054        0091        !This is code from MEMBER of DATA.!                                 $COPY
001 00FC   000 0054        0092        ! //DATA DD DSN=SYS1.DATA,VOL=SER=TEMP01,UNIT=2314,DISP=OLD !       $COPY
001 00FC   000 0054        0093        !This is the end of the $COPY DATA(MEMBER)!                         $COPY
```

```
001 00FC   000 0054        0094      TAG: END.                                                    110.
                              SEGN000   ENTRY (SD) AT 0000
                  0020    58C0F118  50321004  D2031000  2000D203   3000C130  D2055000  C1061921  4750C04E
                  0040    95402000  4770C04E  062047F0  C03AD503   D050D04C  4760C0FC  D403D04C  D050D603
                  0060    D04CC10C  D402D051  D04DD602  D051D048   9240D048  D201D049  D0481A12  5910C130
                  0080    47B0C09A  58F0C144  05EF58C0  E0AE4790   C09A5B20  C13447F0  C07A4440  C12A4740
                  00A0    C0FC47E0  C0FC92FF  D0489500  D0494780   C0FCD202  D04DC110  D203D050  C1134110
                  00C0    C1174120  000F1233  4770C0D2  D203100A   C1261A11  4811C0F2  47F1C000  1A5347FC
                  00E0    00FC1B53  47FC00FC  1C4347FC  00FC1D43   47FC00FC  00DC00E2  00E800EE  58D0D004
                  0100    98ECD00C  07FEA2A3  99899587  40404040   04096C40  202120E3  8889A240  89A24081
                  0120    40A385A2  A34B9485  A2A29100  10004710   00000010  00000004  00000000  00000000
                  0140    00000000  00000000
                              SEGN001   ENTRY (SD) AT 0000
                              SEGN000   EXTERNAL REFERENCE
                              ENTRY     EXTERNAL REFERENCE
                              READ      EXTERNAL REFERENCE
```

```
BETA        0009  0032  0050  0053  0053  0056  0073
BRANCH      0031  0032
B1          0047  0066  0077
B2          0047  0048
B3          0047
B5          0047
B6          0032
CARRY       0063
ENTRY       0028  0035  0038  0039
EX          0066
FALSE       0063  0068
GAMMA       0009  0014  0032  0050  0053  0056  0056  0073
HI          0012  0012
HIMASK      0012
LA          0076
LEN         0007  0008  0043  0056  0056  0056  0056  0057
LO          0011  0011  0012
LOMASK      0011
MEM         0018
MIXED       0062  0066
MOVE4       0031  0032
OFF         0062
ON          0062  0067
OVERFLOW    0062
RB          0042  0043
READ        0059
REDUCE      0049  0049
RX          0042  0043
R0          0017  0021
R1          0042  0048  0059  0059  0059  0076  0080
R15         0025  0028  0038
R2          0042  0048  0049  0059  0059  0059  0076
R3          0035  0043  0077  0081  0082  0083  0084
R4          0032  0066
R5          0081  0081  0082  0082  0083  0083  0084  0084
R6          0025  0033  0033  0033  0038
SAMPLE      0025
SIZE        0014
SOMECELL    0022  0043
SPACE       0021
STRING      0076  0077
TAG         0050  0066  0067  0068
TM          0066
TRUE        0063  0068
```

```
PL360 COMPILATION           RIGHT TRIANGLE PROBLEM                                          ORVYL 04/09/74    PAGE  1
001 0018   000 0048        0002 01   *  MAKES USE OF MOST OF THE FEATURES OF PL360.                                  33.
001 0018   000 0048        0003      *  THE PROGRAM READS THE SIDES OF A RIGHT TRIANGLE,                             34.
001 0018   000 0048        0004      *  COMPUTES THE HYPOTENUSE, AND WRITES THE RESULT. --;                          35.
001 0018   000 0048        0005                                                                                    36.
001 0018   000 0048        0006      COMMENT -- DECLARE EXTERNAL PROCEDURES, FUNCTIONS,                            37.
001 0018   000 0048        0007      *  AND VARIABLES FIRST. --;                                                     38.
001 0018   000 0048        0008                                                                                    39.
001 0018   000 0048        0009         EXTERNAL PROCEDURE VALTOBCD (R14);  NULL;                                 40.
               VALTOBCD
001 0018   000 0048        0010         EXTERNAL PROCEDURE BCDTOVAL (R14);  NULL;                                 41.
               BCDTOVAL
001 0018   000 0048        0011                                                                                    42.
001 0018   000 0048        0012         PROCEDURE SQRT (R14);  IF F01 > 0L THEN                                   43.
               SQRT
001 0022   000 0048        0013         COMMENT THIS PROCEDURE TAKES THE SQUARE ROOT OF THE VALUE IN F01;         44.
001 0022   000 0048        0014         BEGIN  LONG REAL FCON;                                                    45.
             D048  FCON
001 0022   000 0050        0015 02         FCON := F01;  R1 := R1-R1;                                            46.
001 0028   000 0050        0016            IC(R1,FCON);  R1 := R1 - #40S SHRA 1 + #40S;                          47.
001 0038   000 0050        0017            STC(R1,FCON);  F45 := FCON;  F6 := 1R;                                48.
001 0044   000 0050        0018            WHILE F67 > 10'_6L DO                                                 49.
001 004C   000 0050        0019            BEGIN  F23 := F45;                                                    50.
001 004E   000 0050        0020 03            F45 := F01/F23 + F23 / 2L;                                         51.
001 0058   000 0050        0021               F67 := F45 - F23;  F67 := ABS F67;                                  52.
001 005E   000 0050        0022            END;  F01 := F45;                                                     53.
001 0064   000 0050        0023 02      END;                                                                       54.
001 0066   000 0050        0024 01                                                                                 55.
001 0066   000 0050        0025      COMMENT -- READ & WRITE ARE ALREADY KNOWN --;                                 56.
001 0066   000 0050        0026         FUNCTION REDUCE (6,#0600);  COMMENT -- SUBTRACT 1 FROM REGISTER --;       57.
             0600  REDUCE
001 0066   000 0050        0027                                                                                    58.
001 0066   000 0050        0028         ARRAY 134 BYTE OUTPUT = (                                                 59.
             D050  OUTPUT
001 0066   000 00D6        0029            " HYPOTENUSE =         FOR SIDES OF",100(" "));                       60.
001 0066   000 00D6        0030         BYTE CARD SYN OUTPUT(35), ANSWER SYN OUTPUT(14);                          61.
             D073  CARD
             D05E  ANSWER
001 0066   000 00D6        0031                                                                                    62.
001 0066   000 00D6        0032      COMMENT -- MAIN CODE --;                                                      63.
001 0066   000 00D6        0033      LOOP: R0 := @CARD;  READ;  IF ^= THEN GOTO EXIT;                              64.
001 0078   000 00D6        0034         R1 := @CARD;  R2 := 3;  BCDTOVAL;  F67 := F01 * F01;                       65.
001 008E   000 00D6        0035         BCDTOVAL;  F01 := F01 * F01 + F67;                                         66.
001 009C   000 00D6        0036         SQRT;   COMMENT -- TAKE SQUARE ROOT OF VALUE IN F01 --;                    67.
001 00A0   000 00D6        0037         R1 := @ANSWER;  R3 := 7;  VALTOBCD;                                        68.
001 00B2   000 00D6        0038         R0 := @OUTPUT;  WRITE;  GOTO LOOP;                                         69.
001 00C4   000 00D6        0039      EXIT: END.                                                                    70.
                 0070    D6C64040  40404040  40404040  40404040   40404040  40404040  40404040  40404040
                 0090 TO 00CC      40404040
                 00D0    40404040  4040
                             SEGN000   ENTRY (SD) AT 0000
```

```
0020    F0646000  D0481B11  4310D048  4B10F0CE   8A100001  4A10F0CE  4210D048  6840D048
0040    7860F0D0  6960F0F0  47D0F062  28242840   2D422A42  6D40F0F8  28642B62  206647F0
0060    F0442804  07FE4100  D07358F0  F0DC05EF   58F0E064  4760F0C4  4110D073  41200003
0080    58F0F0E0  05EF58F0  E04E2860  2C6058F0   F0E005EF  58F0E040  2C002A06  45E0F01C
00A0    4110D05E  41300007  58F0F0E4  05EF58F0   E0264100  D05058F0  F0E805EF  58F0E018
00C0    47F0F066  58D0D004  98ECD00C  07FE0040   41100000  00000000  00000000  00000000
00E0    00000000  00000000  00000000  72565520   3CA7C5AC  471B4784  41200000  00000000
```

```
SEGN001   ENTRY (SD) AT 0000
SEGN000   EXTERNAL REFERENCE
READ      EXTERNAL REFERENCE
BCDTOVAL  EXTERNAL REFERENCE
VALTOBCD  EXTERNAL REFERENCE
WRITE     EXTERNAL REFERENCE
```

```
                TRTEST
014 0000   000 0000        0002 01   COMMENT THIS ROUTINE TESTS AN INPUT STRING                                  5.
014 0000   000 0000        0003      * AGAINST A TRANSLATE TABLE.                                               6.
014 0000   000 0000        0004      * ENTER WITH R1 = @ OF STRING TO BE TESTED.                                7.
014 0000   000 0000        0005      *            R2 = @ OF TABLE.                                              8.
014 0000   000 0000        0006      *            R3 = LENGTH OF STRING TO BE TESTED.                           9.
014 0000   000 0000        0007      * EXITS WITH R1 = LENGTH OF TRANSLATED STRING.                            10.
014 0000   000 0000        0008      *            R2 = TRANSLATE TABLE CHARACTER WHICH                          11.
014 0000   000 0000        0009      *                 STOPPED TRANSLATION.                                     12.
014 0000   000 0000        0010      *      ALSO, CONDITION CODE SET BASED ON R2;                               13.
014 0000   000 0000        0011         FUNCTION REDUCE(6,#0600);                                             14.
          0600  REDUCE
014 0000   000 0000        0012         STM(R3,R6,B13(12)); COMMENT SAVE REGISTERS;                           15.
014 0004   000 0000        0013         R4 := R2; R5 := @B1; R2 := R2-R2; R1 := R2;                           16.
014 000E   000 0000        0014         IF R3 > 0 THEN                                                        17.
014 0014   000 0000        0015         BEGIN REDUCE(R3); R6 := R2;                                           18.
014 0018   000 0000        0016 02         FOR R3 := R3 STEP _256 UNTIL 256 DO                                19.
014 0018   000 0000        0017            BEGIN TRT(255,B5,B4); IF ^= THEN                                   20.
014 0026   000 0000        0018 03            BEGIN R1 := @B1(R6)-R5; GOTO EXIT;                              21.
014 0030   000 0000        0019 04            END ELSE                                                        22.
014 0030   000 0000        0020 03            BEGIN R6 := @B6(256); R5 := @B5(256);                           23.
014 003C   000 0000        0021 04            END;                                                            24.
014 003C   000 0000        0022 03         END; EX(R3,TRT(0,B5,B4));                                          25.
014 004C   000 0000        0023 02         IF = THEN R1 := @B5(R3+1);                                         26.
014 0054   000 0000        0024            R1 := @B1(R6) - R5;                                                27.
014 005A   000 0000        0025         END;                                                                  28.
014 005A   000 0000        0026 01   EXIT: LM(R3,R6,B13(12)); LTR(R2,R2);                                    29.
014 0060   000 0000        0027      END.                                                                     30.
                0020    40004790  F0344116  10001B15  47F0F05A   47F0F03C  41606100  41505100  5A30F068
                0040    5930F06C  47A0F01C  4430F062  4770F054   41135001  41161000  1B159836  D00C1222
                0060    07FEDD00  50004000  FFFFFF00  00000100
                     TRTEST    ENTRY (SD) AT 0000
```

```
                OPTION
014 0000   000 0000        0002      BEGIN BALR(R10,R0);  R1 := 2;  R10 := R10-R1;                                  2.
014 0008   000 0000        0003 01      BEGIN PROCEDURE TPUT (R9);                                                   3.
                TPUT
014 000C   000 0000        0004 02         BEGIN R0 := 1;  SVC(246);  R0 := 1;  SVC(242);                            4.
014 0018   000 0000        0005 03         END;  PROCEDURE WYLBUR (R9);                                              5.
                WYLBUR
014 001A   000 0000        0006 02         BEGIN R1 := NEG R1;  R0 := R0-R0;  SVC(254);  END;                        6.
014 0022   000 0000        0007 02         LA(R1,#5F1CX);  R15 := 2;  TPUT;                                          7.
014 002E   000 0000        0008            LA(R1,"SET TERSE");  R15 := STRING;  WYLBUR;                              8.
014 003A   000 0000        0009            LA(R1,"SET NUM");  R15 := STRING;  WYLBUR;                                9.
014 0046   000 0000        0010            LA(R1,"SET VOL SYS19");  R15 := STRING;  WYLBUR;                         10.
014 0052   000 0000        0011            LA(R1,"SET NOTIME");  R15 := STRING;  WYLBUR;                            11.
014 005E   000 0000        0012            LA(R1,"SET WIDTH 72");  R15 := STRING;  WYLBUR;                          12.
014 006A   000 0000        0013            SVC(253);                                                                13.
014 006C   000 0000        0014         END;                                                                        14.
014 006C   000 0000        0015 01   END.                                                                          15.
```

APPENDIX B. 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 show 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. It is assumed that R15 is the program base register (cf. 4.2, 10.1).

Operands		Operators						
K register (type)	A primary (type)	1 :=	2 +	3 -	4 *	5 /	6 ++	
integer	integer register	LR	AR	SR	MR	DR	ALR	S
integer	integer cell	L	A	S	M	D	AL	S
integer	short integer cell	LH	AH	SH	MH			
real	real register	LER	AER	SER	MER	DER	AUR	S
real	real cell	LE	AE	SE	ME	DE	AU	S
long real	real register	LER	AER	SER	MER	DER	AUR	S
long real	long real register	LDR	ADR	SDR	MDR	DDR	AWR	S
long real	real cell	LE	AE	SE	ME	DE	AU	S
long real	long real cell	LD	AD	SD	MD	DD	AW	S

Table B.1 - Object Code Operators

1. <K-register> := <A-primary>

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

2. <K-register assignment><operator><A-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 B.1, columns 2-7.

3. <A-cell> := <K-register>

The code consists of a single store instruction depending on the types of cell and register as indicated by Table B.1, 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
```

ci is determined by the i-th condition, which itself either translates into a compare instruction depending on the types of compared quantities (cf. Table B.1, column 9), or has no corresponding instruction, if it merely designates a relation or integer value.

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
```

5. IF <condition-1> OR ... OR <condition-n-1> OR
<condition-n> THEN <simple statement> ELSE <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. CASE <integer register-m> OF
BEGIN <statement-1>;
<statement-2>;
...
<statement-n>;
END

```
        AR  m,m
        LH  m,SW(m)
        B   0(m,p)
L1      EQU  *-ORIGIN
        (statement-1)
        B    LX(P,0)
L2      EQU  *-ORIGIN
        (statement-2)
        B    LX(P,0)
        .    .
        .    .
        .    .
Ln      EQU  *-ORIGIN
        (statement-n)
        B    LX(P,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 Rp is assumed to contain this address (cf. 5.1, 8.1).

7. WHILE <condition> DO <statement>

```
L1      (condition)
        BC   cond,L2
        (statement)
        B    L1
L2
```

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

8. FOR <integer register assignment>
STEP <increment> UNTIL <limit> DO <statement>

```
        (integer register assignment)
        B    L2
L1      (statement)
        A    m,INC
L2      C    m,LIM
        BC   cond,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, cond depends on the sign of the increment.

9. PROCEDURE <identifier>(<integer register>);<statement>

```
        P    (statement)
             BR  m
```

It is assumed that the integer register enclosed in parentheses is Rm and P is a label corresponding to the procedure identifier.

10. <procedure identifier>

```
             BAL  m,P

        or   L    b,newbase
             BALR m,b
             L    b,oldbase

        or   L    b,newbase
             BAL  m,P
             L    b,oldbase
```

It is here assumed that P designates the relative address of the procedure to be called within the program segment in which it is declared, and m is the return address register specified in its declaration, and b is the program segment's base register. 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,b
             BALR b,0
             L    b,oldbase

        or   L    b,newbase
             BALR m,b
             LTR  n,b
             BALR b,0
             L    b,oldbase

        or   L    b,newbase
             BAL  m,P
             LTR  n,b
             BALR b,0
             L    b,oldbase
```

APPENDIX C. COMPILER CONSTRUCTS

Registers

Reg	R0 thru R15	INTEGER
Freg	F0, F2, F4, F6	REAL
Lreg	F01, F23, F45, F67	LONG REAL

Subscript -d indicates that register assigned on the left side of the assign symbol (:=), thus

```
Reg-d := Expression
```

Cells

Bcell	BYTE	Value X
Scell	SHORT INTEGER	Value S
Icell	INTEGER	Value
Fcell	REAL	Value R
Lcell	LONG REAL	Value L

Note: Values may replace cells in an expression.

```
Reg-d := Icell    could be:   (Icell)
         Reg-d := #FACE      0000FACE
         Reg-d := "DROP"     C4D9D6D7
         Reg-d := _4         FFFFFFFC
```

Conditions

Cond represents, = , ^= , >= , <= , > , < , Number , ^Number

In the following tables, * preceding the CODE indicates the instruction does not change the condition code.

Code	Mnemonic	Compiler Construct
05	BALR	Procname (not local procedure call)
07	BCR	END of any PROCEDURE
10	LPR	Reg-d := ABS Reg
11	LNR	Reg-d := NEG ABS Reg
12	LTR	Reg Cond 0
13	LCR	Reg-d := NEG Reg
14	NR	Reg-d AND Reg
16	OR	Reg-d OR Reg
17	XR	Reg-d XOR Reg
*18	LR	Reg-d := Reg or Reg-d ... =: Reg
	Note: Reg-d := Reg-d generates no code.	
19	CR	Reg-1 Cond Reg-2
1A	AR	Reg-d + Reg
1B	SR	Reg-d - Reg
*1C	MR	Reg-d * Reg
	Note: Reg-d must be odd numbered	
*1D	DR	Reg-d / Reg
	Note: Reg-d must be odd numbered	
1E	ALR	Reg-d ++ Reg
1F	SLR	Reg-d -- Reg
20	LPDR	Lreg-d := ABS Lreg
21	LNDR	Lreg-d := NEG ABS Lreg
22	LTDR	Lreg Cond 0L
23	LCDR	Lreg-d := NEG Lreg
*28	LDR	Lreg-d := Lreg or Lreg-d ... =: Lreg
	Note: Lreg-d := Lreg-d generates no code.	
29	CDR	Lreg-1 Cond Lreg-2
2A	ADR	Lreg-d + Lreg
2B	SDR	Lreg-d - Lreg
*2C	MDR	Lreg-d * Lreg
*2D	DDR	Lreg-d / Lreg
2E	AWR	Lreg-d ++ Lreg
2F	SWR	Lreg-d -- Lreg
30	LPER	Freg-d := ABS Freg
31	LNER	Freg-d := NEG ABS Freg
32	LTER	Freg Cond 0R
33	LCER	Freg-d := NEG Freg
*38	LER	Freg-d := Freg or Freg-d ... =: Freg
	Note: Freg-d := Freg-d generates no code.	
39	CER	Freg-1 Cond Freg-2
3A	AER	Freg-d + Freg
3B	SER	Freg-d - Freg
*3C	MER	Freg-d * Freg
*3D	DER	Freg-d / Freg
3E	AUR	Freg-d ++ Freg
3F	SUR	Freg-d -- Freg

Table C.1 - 2-Byte Instructions

All these instructions allow indexable cells.

Code	Mnemonic	Compiler Construct
*40	STH	Scell := Reg or Reg-d ... =: Scell
*41	LA	Reg-d := @Cell
45	BAL	Procname (local procedure call)
*47	BC	GOTO Tag
		--- THEN
		--- ELSE
		--- DO
*48	LH	Reg-d := Scell
49	CH	Reg Cond Scell
4A	AH	Reg-d + Scell
4B	SH	Reg-d - Scell
*4C	MH	Reg-d * Scell
*50	ST	Icell := Reg or Reg-d ... =: Icell
54	N	Reg-d AND Icell
56	O	Reg-d OR Icell
57	X	Reg-d XOR Icell
*58	L	Reg-d := Icell or Reg-d := @Procname
59	C	Reg-d Cond Icell
5A	A	Reg-d + Icell
5B	S	Reg-d - Icell
*5C	M	Reg-d * Icell
	Note: Reg-d must be odd numbered	
*5D	D	Reg-d / Icell
	Note: Reg-d must be odd numbered	
5E	AL	Reg-d ++ Icell
5F	SL	Reg-d -- Icell
*60	STD	Lcell := Lreg or Lreg-d ... =: Lcell
*68	LD	Lreg-d := Lcell
69	CD	Lreg-d Cond Lcell
6A	AD	Lreg-d + Lcell
6B	SD	Lreg-d - Lcell
*6C	MD	Lreg-d * Lcell
*6D	DD	Lreg-d / Lcell
6E	AW	Lreg-d ++ Lcell
6F	SW	Lreg-d -- Lcell
*70	STE	Fcell := Freg or Freg-d ... =: Fcell
*78	LE	Freg-d := Fcell
79	CE	Freg-d Cond Fcell
7A	AE	Freg-d + Fcell
7B	SE	Freg-d - Fcell
*7C	ME	Freg-d * Fcell
*7D	DE	Freg-d / Fcell
7E	AU	Freg-d ++ Fcell
7F	SU	Freg-d -- Fcell
*88	SRL	Reg-d SHRL Ivalue or Reg
*89	SLL	Reg-d SHLL Ivalue or Reg
8A	SRA	Reg-d SHRA Ivalue or Reg
8B	SLA	Reg-d SHLA Ivalue or Reg

Table C.2 - 4-Byte Instructions

APPENDIX D. SYNTACTIC INDEX

Syntactic Entity	Section	Syntactic Entity	Section
<A-number>	3.2	<label definition>	4.1
<alternative condition>	6.5	<letter>	2.2.1
<arithmetic operator>	6.2	<limit>	6.8
<block>	4.1	<logical operator>	6.2
<block body>	4.1	<monadic operator>	6.1
<block head>	4.1	<parameter>	7.2
<case cause>	6.9	<parameter list>	7.2
<case sequence>	6.9	<procedure declaration>	8.1
<CASE statement>	6.9	<procedure heading>	8.1
<character>	3.4	<procedure identifier>	2.2.1
<character sequence>	3.4	<procedure statement>	8.2
<combined condition>	6.5	<program>	4.1
<compound condition>	6.5	<relation>	6.5
<condition>	6.5	<repetition list>	5.3
<declaration>	4.1	<scale factor>	3.2
<digit>	2.2.1	<segment base declaration>	5.2
<fill value>	5.3	<segment base heading>	5.2
<floating-point number>	3.2	<segment close declaration>	5.2
<for clause>	6.8	<separate procedure heading>	8.1
<FOR statement>	6.8	<shift operator>	6.2
<format code>	7.1	<simple K-register assignment>	6.1
<fractional number>	3.2	<simple procedure heading>	8.1
<function declaration>	7.1	<simple statement>	4.1
<function definition>	7.1	<simple T-type>	5.3
<function designator>	7.2	<stat condition>	6.5
<function identifier>	2.2.1	<statement>	4.1
<GOTO statement>	6.4	<string>	3.4
<hexadecimal digit>	3.1	<syn cell value>	5.6
<hexadecimal value>	3.1	<synonymous cell>	5.5
<identifier>	2.2.1	<synonymous integer value>	5.6
<if clause>	6.6	<T-cell assignment>	6.3
<IF statement>	6.6	<T-cell declaration>	5.3
<increment>	6.8	<T-cell designator>	5.4
<index>	5.4	<T-cell identifier>	2.2.1
<instruction code>	7.1	<T-cell synonym>	5.5
<integer register expression>	5.4	<T-cell value>	6.3
<integer value expression>	5.4	<T-primary>	6.1
<integer value identifier>	2.2.1	<T-type>	5.3
<integer value synonym>	5.6	<T-value>	3.3
<item>	5.3	<true part>	6.6
<K-primary>	6.1	<unsigned A-number>	3.2
<K-register>	2.2.1	<while clause>	6.7
<K-register assignment>	6.2	<WHILE statement>	6.7
<K-register synonym>	5.1		

APPENDIX E. SYNTACTIC ENTITIES

```
<A-number> ::= <unsigned A-number> !
      _ <unsigned A-number>
<alternative condition> ::= <stat condition> !
      <alternative condition> OR <stat condition>
<arithmetic operator> ::= + ! - ! * ! / ! ++ ! --
<block body> ::= <block head> !
      <block body> <statement> ; !
      <block body> <label definition>
<block head> ::= BEGIN ! <block head> <declaration> ;
<block> ::= <block body> END
<byte value> ::= <integer number> X
<case clause> ::= CASE <integer register> OF
<case sequence> ::= <case clause> BEGIN !
      <case sequence> <statement> ;
<CASE statement> ::= <case sequence> END
<character> ::= <any EBCDIC character except "> ! ""
<character sequence> ::= <character> !
      <character sequence> <character>
<combined condition> ::= <stat condition> !
      <combined condition> AND <stat condition>
<compound condition> ::= <combined condition> !
      <alternative condition>
<condition> ::= <T-cell designator> <relation> <T-cell value> !
      <byte cell designator> !
      ^ <byte cell designator> !
      <K-register> <relation> <A-primary> !
      <integer register> <relation> <string> !
      <relation> !
      <integer value> !
      ^ <integer value>
<declaration> ::= <T-cell declaration> !
      <procedure declaration> !
      <function declaration> !
      <T-cell synonym> !
      <K-register synonym> !
      <integer value synonym> !
      <segment base declaration> !
      <segment close declaration>
<digit> ::= 0 ! 1 ! 2 ! 3 ! 4 ! 5 ! 6 ! 7 ! 8 ! 9
<fill value> ::= <T-value> !
      <string> !
      @<procedure identifier> !
      @@<procedure identifier> !
      @<T-cell designator> !
      @@<T-cell identifier> !
      <repetition list> <fill value> )
<floating-point number> ::= <fractional number> !
      <fractional number> ' <scale factor> !
      <unsigned integer number> ' <scale factor>
<for clause> ::= FOR <integer register assignment> STEP <increment>
      UNTIL <limit> DO
<FOR statement> ::= <for clause> <statement>
```

```
<format code> ::= <integer value>
<fractional number> ::= <unsigned integer number> . !
      <fractional number> <digit>
<function declaration> ::= FUNCTION <function definition> !
      <function declaration> , <function definition>
<function definition> ::=
      <identifier> ( <format code> , <instruction code> )
<function designator> ::= <function identifier> !
      <function identifier> ( <parameter list> )
<function identifier> ::= <identifier>
<GOTO statement> ::= GOTO <identifier>
<hexadecimal digit> ::= <digit> ! A ! B ! C ! D ! E ! F
<hexadecimal value> ::= # <hexadecimal digit> !
      <hexadecimal value> <hexadecimal digit>
<identifier> ::= <letter> ! <identifier> <letter> ! <identifier> <digit>
<if clause> ::= IF <compound condition> THEN
<IF statement> ::= <if clause> <statement> !
      <if clause> <true part> <statement>
<increment> ::= <integer value>
<index> ::= <integer value expression> !
      <integer register expression> !
      <integer register expression> + <integer value expression> !
      <integer register expression> - <integer value expression>
<instruction code> ::= <integer value>
<integer register expression> ::= <integer register> !
      <integer register> + <integer register>
<integer value expression> ::= <integer value> !
      <integer value expression> + <integer value> !
      <integer value expression> - <integer value>
<integer value identifier> ::= <identifier>
<integer value synonym> ::=
      EQUATE <identifier> <synonymous integer value> !
      EQUATE <identifier> SYN <string> !
      EQUATE <identifier> SYN <register name> !
      <integer value synonym> , <identifier> <synonymous integer value>
<integer value> ::= <integer number> !
      <hexadecimal value> !
      <integer value identifier>
<item> ::= <identifier> ! <identifier> = <fill value>
<K-primary> ::= <K-register>
<K-register assignment> ::= <simple K-register assignment> !
      <K-register assignment> <arithmetic operator> <A-primary> !
      <K-register assignment> =: <K-register> !
      <K-register assignment> =: <A-cell designator> !
      <integer register assignment> <logical operator> <integer primary> !
      <integer register assignment> <shift operator> <integer value> !
      <integer register assignment> <shift operator> <integer register>
<K-register synonym> ::=
      <simple K-type> REGISTER <identifier> SYN <K-register> !
      <K-register synonym> , <identifier> SYN <K-register>
<K-register> ::= <identifier>
<label definition> ::= <identifier> :
<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
<limit> ::= <integer primary> ! <short integer primary>
```

```
<logical operator> ::= AND ! OR ! XOR
<long real value> ::= <long real number> !
      <hexadecimal value> L
<monadic operator> ::= ABS ! NEG ! NEG ABS
<parameter list> ::= <parameter> ! <parameter list> , <parameter>
<parameter> ::= <T-value> !
      <T-cell designator> !
      <K-register> !
      <string> !
      <function designator>
<procedure declaration> ::= <procedure heading> ; <statement>
<procedure heading> ::= <simple procedure heading> !
      COMMON <simple procedure heading> !
      <separate procedure heading> !
      <separate procedure heading> BASE <integer register>
<procedure identifier> ::= <identifier>
<procedure statement> ::= <procedure identifier> !
      <procedure identifier> ( <integer register> )
<program> ::= <block> . !
      GLOBAL <simple procedure heading> ; <statement> . !
      GLOBAL <simple procedure heading> BASE <integer register> ; <statement>
<real value> ::= <real number> !
      <hexadecimal value> R
<relation> ::= = ! ^= ! < ! <= ! >= ! >
<repetition list> ::= ( !
      <integer value> ( !
      <repetition list> <fill value> ,
<scale factor> ::= <integer number>
<segment base declaration> ::=
      <segment base heading> BASE <integer register>
<segment base heading> ::= SEGMENT !
      GLOBAL DATA <identifier> !
      EXTERNAL DATA <identifier> !
      COMMON DATA <identifier> !
      COMMON !
      DUMMY
<segment close declaration> ::= CLOSE BASE
<separate procedure heading> ::=
      SEGMENT <simple procedure heading> !
      GLOBAL <simple procedure heading> !
      EXTERNAL <simple procedure heading>
<shift operator> ::= SHLL ! SHLA ! SHRL ! SHRA
<short integer value> ::= <short integer number> !
      <hexadecimal value> S
<simple byte type> ::= BYTE ! CHARACTER
<simple integer type> ::= INTEGER ! LOGICAL
<simple K-register assignment> ::=
      <K-register> := <A-primary> !
      <K-register> := <monadic operator> <A-primary> !
      <integer register> := <string> !
      <integer register> := @ <T-cell designator> !
      <integer register> := @ <procedure identifier>
<simple long real type> ::= LONG REAL
```

```
<simple procedure heading> ::=
      PROCEDURE <identifier> ( <integer register> )
<simple real type> ::= REAL
<simple short integer type> ::= SHORT INTEGER
<simple statement> ::= <block> !
      <K-register assignment> !
      <T-cell assignment> !
      <function designator> !
      <procedure statement> !
      <GOTO statement> !
      <CASE statement> !
      NULL
<stat condition> ::= <condition> !
      <statement> ; <condition>
<statement> ::= <simple statement> !
      <IF statement> !
      <WHILE statement> !
      <FOR statement>
<string> ::= " <character sequence> " !
      <hexadecimal value> X
<syn cell value> ::= <T-cell designator> - <T-cell designator>
<synonymous cell> ::= SYN <T-cell designator> ! SYN <integer value>
<synonymous integer value> ::= SYN <integer value> !
      SYN <monadic operator> <integer value> !
      SYN <syn cell value> !
      <synonymous integer value> <arithmetic operator> <integer value> !
      <synonymous integer value> <logical operator> <integer value> !
      <synonymous integer value> <shift operator> <integer value>
<T-cell assignment> ::= <A-cell designator> := <K-register> !
      <T-cell designator> := <T-cell value> !
      <T-cell assignment> <logical operator> <T-cell value>
<T-cell declaration> ::= <T-type> <item> ! <T-cell declaration> , <item>
<T-cell designator> ::= <T-cell identifier> !
      <T-cell identifier> ( <index> / <integer value expression> ) !
      <T-cell identifier> ( <index> )
<T-cell identifier> ::= <identifier>
<T-cell synonym> ::=
      <T-type> <identifier> <synonymous cell> !
      <T-cell synonym> , <identifier> <synonymous cell>
<T-cell value> ::= <T-cell designator> !
      <T-value> !
      <string>
<T-primary> ::= <T-value> ! <T-cell designator>
<T-type> ::= <simple T-type> ! ARRAY <integer value> <simple T-type>
<true part> ::= <simple statement> ELSE
<unsigned integer number> ::= <digit> !
      <unsigned integer number> <digit>
<unsigned long real number> ::= <floating-point number> L !
      <unsigned integer number> L
<unsigned real number> ::= <floating-point number> !
      <unsigned integer number> R
<unsigned short integer number> ::= <unsigned integer number> S
<while clause> ::= WHILE <compound condition> DO
<WHILE statement> ::= <while clause> <statement>
```