PL360, A Programming Language for the 360 Computers

NIKLAUS WIRTH*

Stanford University, Stanford, California

ABSTRACT. A programming language for the IBM 360 computers and aspects of its implementation are described. The language, called PL360, provides the facilities of a symbolic machine language, but displays a structure defined by a recursive syntax. PL360 was designed to improve the readability of programs which must take into account specific characteristics and limitations of a particular computer. It represents an attempt to further the state of the art of programming by encouraging and even forcing the programmer to improve his style of exposition and his principles and discipline in program organization. Because of its inherent simplicity, the language is particularly well suited for tutorial purposes.

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

KEY WORDS AND PHRASES: language, programming language, symbolic machine language, assembly language, phrase structure language, compiler, precedence grammar, S360, error diagnosis, ALGOL-like

CR CATEGORIES: 4.1, 4.10, 4.11, 4.2, 4.21, 4.42

CONTENTS

* Present address: Institut für Operations Research und elektronische Datenverarbeitung der Universität Zürich, Sumatrastrasse 30, Zürich, Switzerland 8006.

1. Introduction, Historical Background, and Aims

In an era of feverish and prolific activity in the design of more and more sophisticated and intricate programming aids, the proposal of a machine-dependent language may seem anachronistic to some readers. In this report we describe an attempt to provide a tool for those applications where it is essential to conceive programs as closely as possible in terms of a specific computer in order to directly take into account its particular capabilities and limitations. These applications often require the use of conventional assembly codes, where it is particularly difficult, if not impossible, to express and clearly exhibit the structure and the characteristics of an algorithm. The language described here is designed with the aim of providing a clear and systematic exposition of the available computing facilities, and a tool which encourages a programmer to write programs in a disciplined, lucid, and readable style while still maintaining control over the optimal use of specific machine characteristics. The result of a systematic language design based on consistent rules is reliability on the part of the implemented translator as well as on the part of the user, who is discouraged from using "tricks" and is less subject to pitfalls and misunderstandings about the nature of complicated or ill-defined facilities.

In the summer of 1965, the author decided to undertake efforts to implement the proposed successor to ALGOL described in [1] on the IBM 360 computer, which at that time had been chosen as Stanford's next generation machine. It was felt that the evolving project should be conducted in a thorough and systematic manner, worthy of an academic endeavor, and making use of the best available methods on compiler construction known. The results should consist of a well-organized system whose structure and principles are sound and precisely understood, and which is intelligibly documented.

After many years of experience with ALGOL, it was clearly recognized that a compiler written in 360 Assembly Language would neither be able to meet the desired documentation standards, nor constitute a sufficiently convenient programming tool. The only other language available on the 360, FORTRAN, was not deemed adequate either. Against the strong arguments of the undesirability of the large amount of additional efforts required to produce a new language and its compiler, it was decided to develop a tool which would (1) allow full use of the facilities provided by the 360 hardware, (2) provide convenience in writing and correcting programs, and (3) encourage the user to write in a clear and comprehensible style.

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

Section 2 contains the definition of the language. It is given in terms of a syntax, and the semantic explanations of the individual syntactic constructions. Knowledge about the nature of the 360 architecture is prerequisite (cf. [2,3]); however, the definition does not require familiarity with the 360 Assembly Language.

In Section 3 several examples of PL360 (sub)programs are given. Since PL360 allows (and requires) the programmer to denote almost every machine instruction explicitly, its use inherently bears some of the tediousness of assembly language

programming. However, the presence of structure in the language, and in particular the introduction of blocks, procedures, and if, for, and while clauses, helps to improve the readability of the language significantly. It drastically reduces the number of necessary program labels and thus of invented names. The method used for assigning base registers and initializing them with the correct values at run time considerably eases the complicated and pitfall-loaded addressing problem of the 360 computer.

We devote Sections 4 and 5 to the implementation of PL360. We exhibit the code which the compiler generates corresponding to various language statements, and the method of segmentation and addressing. In Section 6 we give an account of the organization of the compiler, which relies on a rigorous syntax analysis of the text while at the same time generating the target code. The compiler constitutes a large-scale practical example for the application of the techniques described in [4] which have been extended to process incorrectly constructed texts and to meaningfully diagnose errors. The success of this facility is considered to be a major contribution to making precedence grammars useful in practical applications.

The methods employed in producing the compiler are described in Section 7. A bootstrapping technique was used to make the compiler available on the 360 computer without prior use of any of the languages existing on that machine. Programming the compiler in its own language provided a thorough test for the adequacy of the language to its anticipated purpose.

In Section 8 we give a brief account of the size and the performance of the translator on a 360/50 computer. Concluding remarks about the language and its implementation lead to a brief examination of the appropriateness of the 360 architecture for this experiment.

2. Definition of the Language

2.1. Terminology, Notation, and Basic Definitions

The language is defined in terms of a *computer* which comprises a *processing unit* and a finite set of *storage elements*. Each of the storage elements holds a *content*, also called *value*. At any given time, certain significant *relationships* may hold between storage elements and values. These relationships may be recognized and altered, and new values may be created by the processing unit. The actions taken by the processor are determined by a *program*. The set of possible programs forms 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 currently executed instruction, and a quantity which is called *condition code*.

Storage elements are classified into *registers* and core memory cells, simply called

cells. Registers are divided into three types according to their size and the operations which can be performed on their values. The types of registers are:

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

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

(a) **byte** (a sequence of 8 bits = 1 byte),
(b) **short integer** (a sequence of 16 bits = 2 bytes, interpreted as an integer in two's complement binary notation),
(c) **integer** or **logical** (a sequence of 32 bits = 4 bytes, the former to be interpreted as an integer in two's complement binary notation),
(d) **real** (a sequence of 32 bits = 4 bytes, to be interpreted as a floating point binary number),
(e) **long real** (a sequence of 64 bits = 8 bytes, to be interpreted as a floating point binary number).

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

2.1.2. Relationships

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

Another relationship holds between the cells which are the components of a structured cell, called an array, and the structured cell itself. It is known as *subordination*. Structured cells are regarded as containing the ordered set of the values of the component cells.

A set of relationships between values is defined by *monadic* and *dyadic functions* or operations, which the processor is able to evaluate or perform. The relationships are defined by mappings between values (or pairs of values) known as the operands and values known as the results of the evaluation. These mappings are not to be precisely defined in this report; instead, references will be given to their definition in publications on the system 360 computer [5].

2.1.3. The Program

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

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

(a) Basic symbols constitute the basic *vocabulary* of the language (cf. 1.6.). They are either single characters or underlined letter sequences.
(b) Strings are sequences of characters enclosed in quote marks (").
(c) Comments are sequences of characters (not containing a semicolon) preceded by the basic symbol **comment** and followed by a semicolon (;). It is understood that during execution of a program, all comments are ignored.

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

2.1.4. Syntax

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

Syntactic entities (cf. 2.1.5) are denoted by English words enclosed in the brackets ⟨ and ⟩. These words describe approximately the nature of the syntactic entity, and where these words are used elsewhere in the text, they refer to that syntactic entity. For reasons of notational convenience and brevity, the script letters $\mathcal{A}$, $\mathcal{K}$, and $\mathcal{I}$ are also used in the denotation of syntactic entities. They stand as abbreviations for any of the following words (or pairs):

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

Syntactic rules are of the form

$$\langle \mathrm{A} \rangle ::= \xi$$

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

$$\langle \mathrm{A} \rangle ::= \xi_1 \mid \xi_2 \mid \cdots \mid \xi_n$$

is used as an abbreviation for the n syntactic rules

$$\langle \mathrm{A} \rangle ::= \xi_1 , \quad \langle \mathrm{A} \rangle ::= \xi_2 , \quad \cdots , \quad \langle \mathrm{A} \rangle ::= \xi_n .$$

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

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

is an abbreviation for the set of rules:

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

2.1.5. Syntactic Entities

	Section		*Section*
⟨$\mathcal{A}$ cell assignment⟩	2.2.7	⟨combined condition⟩	2.3.1
⟨$\mathcal{A}$ number⟩	2.2.2	⟨compound condition⟩	2.3.1
⟨alternative condition⟩	2.3.1		
⟨arithmetic operator⟩	2.2.6	⟨digit⟩	2.2.2
		⟨declaration⟩	2.3.5
⟨block body⟩	2.3.5		
⟨block head⟩	2.3.5	⟨for clause⟩	2.3.4
⟨block⟩	2.3.5	⟨for statement⟩	2.3.4
		⟨format code⟩	2.2.8
		⟨fractional number⟩	2.2.2
⟨case clause⟩	2.3.2	⟨function declaration⟩	2.2.8
⟨case sequence⟩	2.3.2	⟨function definition⟩	2.2.8
⟨case statement⟩	2.3.2	⟨function identifier⟩	2.2.1
⟨character sequence⟩	2.2.2	⟨function statement⟩	2.2.9

2.1.6. Basic Symbols

A | *B* | *C* | *D* | *E* | *F* | *G* | *H* | *I* | *J* | *K* | *L* | *M* | *N* | *O* | *P* | *Q* | *R* | *S* | *T* | *U* | *V* | *W* | *X* | *Y* | *Z* |
a | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *i* | *j* | *k* | *l* | *m* | *n* | *o* | *p* | *q* | *r* | *s* | *t* | *u* | *v* | *w* | *x* | *y* | *z* |
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
+ | − | * | / | < | = | > | ¬ | := | , | . | ; | : | (|) | @ | ℵ | ' | " | _ |
and | **or** | **xor** | **abs** | **neg** | **shll** | **shrl** | **shla** | **shra** |
if | **then** | **else** | **case** | **of** | **while** | **do** | **for** | **step** | **until** |
begin | **end** | **goto** | **comment** | **null** |
integer | **real** | **logical** | **byte** | **long** | **short** | **array** |
function | **procedure** | **register** | **syn** | **overflow** |
segment | **base**

2.2. Data Manipulation Facilities

2.2.1. Identifiers

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

An identifier is a ℋ register-, 𝔍 cell-, procedure-, or function-identifier if it has respectively been associated with a ℋ register, 𝔍 cell, procedure, or function (called a quantity) in one of the blocks surrounding its occurrence. This association is achieved by an appropriate declaration. The identifier is said to designate the associated quantity. If the same identifier is associated to more than one quantity, then the considered occurrence designates the quantity to which it was associated

in the smallest block embracing the considered occurrence. In any one block, an identifier must be associated to exactly one quantity. In the parse of a program, that association determines which of the rules given above applies.

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

2.2.2. Values

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨unsigned integer number⟩ ::= ⟨digit⟩ | ⟨unsigned integer number⟩⟨digit⟩
⟨unsigned short integer number⟩ ::= ⟨unsigned integer number⟩ *S*
⟨fractional number⟩ ::= ⟨integer number⟩.⟨digit⟩ | ⟨fractional number⟩⟨digit⟩
⟨scale factor⟩ ::= ⟨integer number⟩
⟨unsigned real number⟩ ::= ⟨fractional number⟩ | ⟨unsigned integer number⟩ *R*|
⟨fractional number⟩'⟨scale factor⟩ | ⟨unsigned integer number⟩'⟨scale factor⟩
⟨unsigned long real number⟩ ::= ⟨fractional number⟩ *L* | ⟨unsigned integer number⟩ *L*|
⟨fractional number⟩'⟨scale factor⟩ *L* | ⟨unsigned integer number⟩'⟨scale factor⟩ *L*
⟨¢ number⟩ ::= ⟨unsigned ¢ number⟩ | ¬ ⟨unsigned ¢ 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, it is preceded by the symbol "¬".

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

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

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

⟨string⟩ ::= "⟨character sequence⟩"
⟨character sequence⟩ ::= ⟨character⟩ | ⟨character sequence⟩⟨character⟩

A string is a sequence of characters enclosed in quote marks. The set of characters depends on the implementation (cf. [2,3]). If the character " is to be an element of the sequence, it is represented by a pair of consecutive quote marks.

Examples:

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

⟨byte value⟩ ::= "⟨character⟩" | ⟨hexadecimal value⟩*X*
⟨short integer value⟩ ::= ⟨short integer number⟩ | ⟨hexadecimal value⟩*S*
⟨integer value⟩ ::= ⟨integer number⟩ | ⟨hexadecimal value⟩
⟨real value⟩ ::= ⟨real number⟩ | ⟨hexadecimal value⟩*R*
⟨long real value⟩ ::= ⟨long real number⟩ | ⟨hexadecimal value⟩*L*

Examples:

byte values:	"B"	"?"	#1FX	
short integer values:	10S	#FF00S		
integer values:	0	1066	_1	#1F2E3D4C
real values:	1.0	_3.1416	2.7'8	#46000001R
long real values:	3.14159265359L		#4E00000000000001L	

Note. If hexadecimal values are used in conjunction with arithmetic operators in a program, they must be considered as the sequence of bits which constitutes the computer's representation of the number on which the operator is applied. Hexadecimal values followed by the letter R or L may be used to denote numbers in unnormalized floating point representation [2,3].

2.2.3. Register Declarations

The System/360 computer features 16 registers which contain integer numbers and are said to be of type **integer** (or **logical**). They are designated by the following standard register identifiers (cf. 2.1):

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

Moreover, the computer features 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 no further register declarations can be made in a program (cf. also 2.2.10).

2.2.4. Cell Declarations

⟨simple byte type⟩ ::= **byte**
⟨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 number⟩⟨simple 𝔗 type⟩
⟨𝔗 cell declaration⟩ ::= ⟨𝔗 type⟩⟨item⟩ | ⟨𝔗 cell declaration⟩, ⟨item⟩
⟨item⟩ ::= ⟨identifier⟩ | ⟨identifier⟩ = ⟨initial value⟩ | ⟨identifier⟩ = (⟨initial value list⟩)
⟨initial value list⟩ ::= ⟨initial value⟩ | ⟨initial value list⟩, ⟨initial value⟩
⟨initial value⟩ ::= ⟨𝔗 value⟩ | ⟨string⟩

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

If a simple type is preceded by the symbol **array** and an integer number, say n, then the declared cell is an array (ordered set) of n cells of the specified simple type.

An initial value list with $m \leq n$ entries specifies the initial values of the first m elements of the array.

A string is understood to stand as an abbreviation for a sequence of byte values, i.e :

$$''\langle char\text{-}1\rangle\langle char\text{-}2\rangle \cdots \langle char\text{-}n\rangle''$$

is an abbreviation of

$$''\langle char\text{-}1\rangle'', ''\langle char\text{-}2\rangle'', \cdots, ''\langle char\text{-}n\rangle''$$

Examples:

byte *flag*
short integer *i, j*
integer *age* = 21, *height* = 68
long real *x, y, z* = 27'3*L*
array 3 **integer** *size* = (36, 23, 37)
array 1000 **real** *quant, price*
array 8 **byte** *flags*
array 132 **byte** *line*

2.2.5. Cell Designators

⟨𝔍 cell designator⟩ ::= ⟨𝔍 cell identifier⟩ | ⟨𝔍 cell identifier⟩(⟨index⟩)
⟨index⟩ ::= ⟨integer number⟩ | ⟨integer register⟩ | ⟨integer register⟩ + ⟨integer number⟩ | ⟨integer register⟩ − ⟨integer number⟩

Cells are denoted by cell designators. In the case of cells of simple type, the designator consists of the identifier associated with that cell; in the case of arrays of cells, the identifier associated with the array is followed by an index. The current value of that index, divided by the number of memory units (bytes) occupied by each array element (cf. 2.1.1), is taken as the ordinal number of the selected element cell.

Note. The remainder resulting from that division must be 0, and register *R*0 must not be specified as an index constituent.

Examples:

age
size (8)
price (*R*1)
line (*R*2 + 15)

2.2.6. Register Assignments

⟨𝔍 primary⟩ ::= ⟨𝔍 value⟩ | ⟨𝔍 cell designator⟩
⟨𝔎 primary⟩ ::= ⟨𝔎 register⟩

A primary is either a value or the content 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⟩ := @ ⟨𝔍 cell designator⟩

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**

TABLE I

$\mathfrak{K}$	$\mathfrak{A}$
integer	integer
integer	short integer
real	real
long real	real
long real	long real

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

If a string is assigned to a register, that string must consist of not more than four characters. If it consists of fewer than four characters, null characters are appended at the left of the string. The bit representation of characters is defined in [2, 3] (EBCDIC).

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

The legal combinations of types to be substituted respectively for the letters $\mathfrak{K}$ and $\mathfrak{A}$ in preceding and subsequent rules of this paragraph are given in Table I.

Examples of simple register assignments:

R0 := *i* *F0* := *quant* (*R1*) *RD* := **abs** *height*
R2 := *R10* *F23* := *x*
R6 := *age* *F45* := **neg** *F01*

⟨$\mathfrak{K}$ register assignment⟩ ::= ⟨simple $\mathfrak{K}$ register assignment⟩ |
⟨$\mathfrak{K}$ register assignment⟩⟨arithmetic operator⟩⟨$\mathfrak{A}$ primary⟩ |
⟨integer register assignment⟩⟨logical operator⟩⟨integer primary⟩ |
⟨integer register assignment⟩⟨shift operator⟩⟨unsigned integer number⟩ |
⟨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 /360 system. The operators ++ and −− denote "logical" or unnormalized addition and subtraction when applied to integer or real registers respectively.

Examples of register assignments:

R0 := *R3*
R1 := 10
R10 := *i* + *age* − *R3* + *size* (8)
R9 := *R8* **and** *R7* **shll** 8 **or** *R6*

F2 := 3.1416
F0 := *quant*(*R1*) * *price*(*R1*)
F45 := *F45* + *F01*

Notes. 1. The syntax implies that sequences of operators, including assignment, are executed strictly in sequence from left to right. Thus

$$R1 := R2 + R1$$

is not equivalent to

$$R1 := R1 + R2$$

but rather to the two statements

$$R1 := R2;\ \ R1 := R1 + R1.$$

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

Example:

$R3 := x * y + z$
$R2$ is affected by the multiplication.

2.2.7. Cell Assignments

⟨𝔄 cell assignment⟩ ::= ⟨𝔄 cell designator⟩ := ⟨𝔎 register⟩

The value of the 𝔎 register is assigned to the designated 𝔄 cell. The allowable combinations of cell and register types 𝔄 and 𝔎 are indicated in Table I.

Examples of cell assignments:

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

2.2.8. Function Declarations

⟨format code⟩ ::= ⟨unsigned integer number⟩
⟨instruction code⟩ ::= ⟨integer value⟩
⟨function definition⟩ ::= ⟨identifier⟩(⟨format code⟩, ⟨instruction code⟩)
⟨function declaration⟩ ::= **function** ⟨function definition⟩ |
⟨function declaration⟩, ⟨function definition⟩

There exist various data manipulation facilities in the 360 computer which cannot be expressed by an assignment. To make these facilities amenable to the language, the function statement is introduced (cf. 2.2.9), which uses an identifier to designate an individual computer instruction. The function declaration serves to associate this identifier, which thereby becomes a function identifier, with the desired computer instruction code, and to define the instruction fields which correspond to the parameters given in function statements. The format code defines the format of the instruction according to Table II. The instruction code defines the first two bytes of the instruction.

In the following example, the identifiers were chosen to be the symbolic codes used in [5], and they are standard function identifiers.

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

TABLE II

Format code	Number of parameter fields in function	Definition of parameter fields (R = ℜ register, P = ℑ primary, L = integer value (literal))
0	0	
1	2	R R
2	2	R P
3	3	R R P
4	2	L P
5	3	L P P
6	1	R
7	1	L
8	1	P
9	2	R P
10	4	L L P P
		0 8 16 32

2.2.9. Function Statements

⟨parameter⟩ ::= ⟨ℑ value⟩ | ⟨string⟩ | ⟨ℜ register⟩ | ⟨ℑ cell⟩
⟨parameter list⟩ ::= ⟨parameter⟩ | ⟨parameter list⟩, ⟨parameter⟩
⟨function statement⟩ ::= ⟨function identifier⟩ | ⟨function identifier⟩(⟨parameter list⟩)

Examples:

SET(*flag*)
RESET(*flag*)
LA(*R*1, *line*)
MVC(15, *line*, *buffer*)
STM(*R*0, *RF*, *save*)
MVI("*", *line*)
IC(*R*0, *flags*(*R*1))

2.2.10. Synonym Declarations

⟨ℑ cell synonym declaration⟩ ::= ⟨ℑ type⟩⟨identifier⟩⟨synonymous cell⟩ |
⟨ℑ cell synonym declaration⟩, ⟨identifier⟩⟨synonymous cell⟩
⟨synonymous cell⟩ ::= **syn** ⟨ℑ cell designator⟩ | **syn** ⟨integer value⟩
⟨ℜ register synonym declaration⟩ ::= ⟨simple ℜ type⟩ **register** ⟨identifier⟩ **syn** ⟨ℜ register⟩
⟨ℜ register synonym declaration⟩, ⟨identifier⟩ **syn** ⟨ℜ register⟩

Synonym declarations serve to associate synonymous identifiers with previousl (i.e., preceding in the text) declared cells or registers. The types associated wit the synonymous cell identifiers need not necessarily agree.

If a synonymous cell is specified by an integer value, then that integer valu represents the displacement and base register part of the cell's machine addres

Examples:

integer *a*16 **syn** *a*(16)
array 32768 **short integer** *mem* **syn** 0
integer *timer* **syn** ∉50
integer *B*1 **syn** *mem*(*R*1)

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

Example:

long real x(#4E00000000000000L)
integer *xlow* **syn** x(4)

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

xlow := $R0$; $F01$:= x

and a conversion vice versa by:

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

No initialization can be achieved by a synonym declaration.

2.2.11. Segment Base Declarations

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

A base declaration causes the compiler to use the specified register as the base address for all cells subsequently declared in the block in which the base declaration occurs. Upon entrance to this block, the appropriate base address is assigned to the specified register (cf. 5.2).

2.3. Control Facilities

2.3.1. If Statements

⟨relation⟩ ::= = | ¬= | < | <= | >= | >
⟨condition⟩ ::= ⟨ℜ register⟩⟨relation⟩⟨𝔄 primary⟩ | ⟨byte cell⟩ | ¬ ⟨byte cell⟩|
⟨relation⟩ |**overflow**

A condition is said to be met or not met. A condition consisting of a relation enclosed by a register and a primary is met if and only if the specified relation holds between the current values of the register and the primary. A condition specified as a byte cell (or a byte cell preceded by ¬) is met if and only if the value of the cell is #*FF* (or not #*FF*). A condition consisting of a relation or the symbol **overflow** is met if the condition code of the processor (cf. 2.1.1) is in a state specified by Table III.

TABLE III

Symbol	*State*
=	0
¬=	1 or 2
<	1
<=	0 or 1
>=	0 or 2
>	2
overflow	3

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

A compound condition is either of the form

*c*1 **and** *c*2 **and** *c*3 **and** ··· **and** *cn*

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

*c*1 **or** *c*2 **or** *c*3 **or** ··· **or** *cn*

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

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

The if statement specifies the conditional execution of statements:

1. ⟨if clause⟩⟨statement⟩

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

2. ⟨if clause⟩⟨true part⟩⟨statement⟩

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

Examples:

```
if R0 < 10 then R1 := 1
if F2 >_3.75 and F2 < 3.75 then F0 := F2 else F0 := 0
if < then SET(flags(0)) else
    if = then SET(flags(1)) else SET(flags(2))
```

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

Example:

```
CLC(15,a,b); if = then ···
```

2.3.2. Case Statements

⟨case clause⟩ ::= **case** ⟨integer register⟩ **of**
⟨case sequence⟩ ::= ⟨case clause⟩ **begin** | ⟨case sequence⟩⟨statement⟩;
⟨case statement⟩ ::= ⟨case sequence⟩ **end**

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

Example:

```
case R1 of
begin 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
```

2.3.3. While Statements

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

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

Examples:

```
while F0 < prize(R1) do R1 := R1 + 4

while R0 < 10 do
begin R0 := R0 + 1;  F01 := F01 * F01;  F23 := F23 * F01;
end
```

2.3.4. For Statements

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

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

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

Examples:

```
for R1 := 0 step 1 until n do STC(R0, line(R1))

for R2 := R1 step 4 until R0 do
    begin F23 := quant(R2) * price(R2);
        F01 := F01 + F23;
    end
```

2.3.5. Blocks

⟨declaration⟩ ::= ⟨𝔍 cell declaration⟩ | ⟨function declaration⟩ | ⟨procedure declaration⟩ | ⟨𝔍 cell synonym declaration⟩ | ⟨𝔎 register synonym declaration⟩ | ⟨segment base declaration⟩
⟨simple statement⟩ ::= ⟨𝔎 register assignment⟩ | ⟨𝔍 cell assignment⟩ | ⟨function statement⟩ | ⟨procedure statement⟩ | ⟨case statement⟩ | ⟨block⟩ | ⟨go to statement⟩ | **null**
⟨statement⟩ ::= ⟨simple statement⟩ | ⟨if statement⟩ | ⟨while statement⟩ | ⟨for statement⟩
⟨label definition⟩ ::= ⟨identifier⟩ :
⟨block head⟩ ::= **begin** | ⟨block head⟩⟨declaration⟩;
⟨block body⟩ ::= ⟨block head⟩ | ⟨block body⟩⟨statement⟩; | ⟨block body⟩⟨label definition⟩
⟨block⟩ ::= ⟨block body⟩ **end**
⟨program⟩ ::= ⟨statement⟩.

A block has the form

begin D; D; ⋯; D; S; S; ⋯; S; **end**

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

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

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

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

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

Example of a block:

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

2.3.6. Go To Statements

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

The interpretation of a go to statement proceeds in the following steps:

(1) Consider the smallest block containing the go to statement.

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

2.3.7. Procedure Declarations

⟨procedure heading⟩ ::= **procedure** ⟨identifier⟩(⟨integer register⟩);
segment procedure ⟨identifier⟩(⟨integer register⟩);

⟨procedure declaration⟩ ::= ⟨procedure heading⟩⟨statement⟩

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

If the symbol **procedure** is preceded by the symbol **segment**, the procedure body is compiled as a separate program segment (cf. 5.1). It has no influence on the meaning of the program.

Examples:

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

procedure sort (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
```

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

2.3.8. Procedure Statements

⟨procedure statement⟩ ::= ⟨procedure identifier⟩

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

3. Examples

```
procedure Magicsquare (R6);
comment This procedure establishes a magic square of order n, if n is odd and 1 < n < 16.
    X is the matrix in linearized form. Registers R0...R6 are used, and register R0 initially
    contains the parameter n. Algorithm 118 [Comm. ACM 5 (Aug. 1962)];
begin short integer nsqr:
    integer register n syn R0, i syn R1, j syn R2, x syn R3, ij syn R4, k syn R5;
    nsqr := n; R1 := n * nsqr;  nsqr := R1;
    i := n + 1 shrl 1;  j := n;
    for k := 1 step 1 until nsqr do
    begin x := i shll 6;  ij := j shll 2 + x;  x := X(ij);
        if x ¬= 0 then
        begin i := i - 1;  j := j - 2;
            if i < 1 then i := i + n;
            if j < 1 then j := j + n;
            x := i shll 6;  ij := j shll 2 + x;
        end;
        X(ij) := k;
        i := i + 1;  if i > n then i := i - n;
        j := j + 1;  if j > n then j := j - n;
    end;
end
```

```
procedure Inreal(R4);
begin comment This procedure reads characters forming a real number according to the
    PL360 syntax. A procedure nextchar(R3) is used to obtain the next character in sequence
    in register R0. The answer appears in the long real register F01. Registers R0···R4 and
    all real registers are used;
    integer register char syn R0, accum syn R1, scale syn R2, ext syn R3;
    long real register answer syn F01;
    byte sign, exposign;
    long real converted = #4E00000000000000L;
    integer convert syn converted (4);
    function SRDL(9, #8C00), LTR(1, #1200);
    nextchar; RESET(sign);
    while char < "0" do
    begin if char = "-" then SET(sign) else RESET(sign); nextchar;
    end;
    comment Accumulate the integral part in accum;
    accum := char and #F; nextchar;
    while char >= "0" do
    begin char := char and #F; accum := accum * 10S + char; nextchar;
    end;
    scale := 0;
    convert := accum; answer := converted + 0L;
    if char = "." then
    begin comment Process fraction. Accumulate number in answer;
        nextchar;
        while char >= "0" do
        begin char := char and #F; convert := char;
            answer := answer * 10L + converted; scale := scale - 1;
            nextchar;
        end;
    end;
    if char = "E" then
    begin comment Read the scale factor and add it to scale;
        nextchar; if char = "-" then
            begin SET(exposign); nextchar;
            end else
            if char = "+" then
            begin RESET(exposign); nextchar;
            end else RESET(exposign);
        accum := char and #F; nextchar;
        while char >= "0" do
        begin char := char and #F; accum := accum * 10S + char; nextchar;
        end;
        if exposign then scale := scale - accum else scale := scale + accum;
    end;
    if scale ¬= 0 then
    begin comment Compute F45 := 10 ↑ scale;
        if scale < 0 then
        begin scale := abs scale; SET(exposign);
        end else RESET(exposign);
        F23 := 10L; F45 := 1L; F67 := F45;
        while scale ¬= 0 do
        begin SRDL(scale, 1);
            comment divide scale by 2, shift remainder into scale extension;
            F23 := F23 * F67; F67 := F23; LTR(ext, ext);
            if < then F45 := F45 * F23; comment < if remainder is 1;
        end;
```

```
        if exposign then answer := answer / F45 else answer := answer * F45;
    end;
    if sign then answer := neg answer;
end


procedure Outreal (R4);
begin comment This procedure converts the (long) real number in register F01 into a string
    of 14 characters which constitute one of its possible decimal denotations. The character
    pattern is bsd.dddddd'sdd, where b is a blank, s a sign, and d a digit. Registers R0, R2, R3,
    R4, and all real registers are used. Upon entry, register R1 must contain the address of the
    output area. Its value remains unchanged;
    integer register exp syn R0, scale syn R2, ext syn R3;
    long real register x syn F01;
    long real convert;
    integer converted syn convert (4), expo syn convert (0);
    byte sign;
    function LTR(1, #1200);
    array 4 logical pattern = (#4021204B, #20202020, #20207D21, #20200000);
    if x = 0L then MVC(13, B1,"  0      ") else
    begin if x < 0L then SET(sign) else RESET(sign); x := abs x; convert := x;
        comment Obtain an estimated decimal scale factor from the exponent part of the
            floating point representation;
        exp := expo shrl 24 - 64 * 307S; if < then exp := exp + 255;
        exp := exp shra 8 - 1; scale := abs exp;
        comment compute F45 := 10 ↑ scale;
        F23 := 10L; F45 := 1L; F67 := F45;
        while scale ¬= 0 do
        begin SRDL(scale, 1); F23 := F23 * F67; F67 := F23;
            LTR(ext, ext); if < then F45 := F45 * F23;
        end;
        comment Normalize to 1 ≤ x < 10;
        if exp < 0 then
        begin x := x * F45;
            while x < 1L do
            begin x := x * 10L; exp := exp - 1;
            end;
        end else
        begin x := x / F45;
            while x >= 10L do
            begin x := x * 0.1L; exp := exp + 1;
            end;
        end;
        x := x * 1'7L ++ #4E00000000000005L;
        convert := x; ext := converted;
        comment ext is here used to hold the integer resulting from the conversion;
        if ext >= 100000000 then
        begin ext := ext / 10; exp := exp + 1;
            comment adjustment needed when conversion results in rounding up to 10.0
                Note that R2 = 0;
        end;
        MVC(13, B1, pattern); CVD(ext, convert); ED(9, B1, convert(3));
        if sign then MVI("-", B1(1));
        CVD(exp, convert); ED(3, B1(10), convert(6));
        if exp < 0 then MVI("-", B1(11)) else MVI ("+", B1(11));
    end;
end
```

```
procedure Binary Search (R8);
comment A binary search is performed for an identifier in a table via an alphabetically or-
    dered directory containing for each entry the length (number of characters) of the identi-
    fier, the address of the actual identifier, and a code number. The global declarations
        array N integer directory
        array N short integer code syn directory (0)
        array N short integer length syn directory (2)
        array N integer address syn directory (4)
        integer n
    are assumed. n equals 8 times the number N of entries in the table, which appear as di-
    rectory(8), directory(16), ..., directory(n). It is assumed that code(0) = 0. Upon entry, R1
    contains the length of the given identifier, R2 contains its address. Upon exit, R3 contains
    the code number if a match is found in the table, 0 otherwise. Registers R1–R8 are used;
begin integer register L syn R1, low syn R3, i syn R4, high syn R5, x syn R6, m syn R7;
    array 3 short integer compare = (#D500S, #2000S, #6000S);
    high := n; low := 8; comment index step in directory is 8;
    while low <= high do
    begin i := low + high shrl 4 shll 3; x := address(i);
        if L = length(i) then
        begin EX(L, compare); if = then goto found;
            if < then high := i - 8 else low := i + 8;
        end else
        if L < length(i) then
        begin EX(L, compare);
            if <= then high := i - 8 else low := i + 8;
        end else
        begin m := length(i); EX(m, compare);
            if < then high := i - 8 else low := i + 8;
        end;
    end;
    i := 0;
found: R3 := code(i);
end
```

4. The Object Code

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

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

(2) No storage element of the computer should be hidden from the programmer. In particular, the usage of registers should be explicitly expressed by each program.

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

The following paragraphs serve to exhibit the machine code into which the various constructs of the language are translated. The mnemonics of the 360 Assembly Language are used to denote the individual instructions. The notation {A} serves to denote the code sequence corresponding to the construct ⟨A⟩.

1. ⟨ℜ register⟩ := ⟨𝔄 primary⟩

TABLE IV

Operands		*Operator*								
ℜ register	*𝔄 primary*	1 :=	2 +	3 −	4 *	5 /	6 ++	7 −−	8 :=	9 ρ
Integer register	Integer register	LR	AR	SR	MR	DR	ALR	SLR		CR
Integer register	Integer cell	L	A	S	M	D	AL	SL	ST	C
Integer register	Short integer cell	LH	AH	SH	MH				STH	CH
Real register	Real register	LER	AER	SER	MER	DER	AUR	SUR		CER
Real register	Real cell	LE	AE	SE	ME	DE	AU	SU	STE	CE
Long real register	Real register	LER	AER	SER	MER	DER	AUR	SUR		CER
Long real register	Long real register	LDR	ADR	SDR	MDR	DDR	AWR	SWR		CDR
Long real register	Real cell	LE	AE	SE	ME	DE	AU	SU	STE	CE
Long real register	Long real cell	LD	AD	SD	MD	DD	AW	SW	STD	CD

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

2. ⟨ℜ 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 IV, cols. 2–7.

3. ⟨𝔄 cell⟩ := ⟨ℜ register⟩

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

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

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

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

Example:

if *R*1 < *R*2 **then** *R*0 := *R*3 **else** *R*0 := *R*4

```
        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

```
        SLL  m,2
        B    SW(m)
L1      {statement-1}
        B    LX
L2      {statement-2}
        B    LX
        ...
Ln      {statement-n}
SW      B    LX
        B    L1
        B    L2
        ...
        B    Ln
LX
```

7. **while** ⟨condition⟩ **do** ⟨statement⟩

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

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

8. **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   c,L1
```

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

9. **procedure** ⟨identifier⟩(⟨integer register-*m*⟩); ⟨statement⟩

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

10. ⟨procedure identifier⟩

```
    BAL  m,P
```

or

```
    L    15,newbase
    BAL  m,P
    L    15,oldbase
```

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

5. Addressing and Segmentation

The addressing mechanism of the 360 computers is such that instructions can indicate addresses only relative to a base address contained in a register. The programmer must insure that (1) every address in his program specifies a "base"-register; (2) the specified register is loaded with the appropriate base address whenever an instruction whose address refers to it is executed; (3) the difference d between the desired absolute address and the available base address satisfies $0 \leq d < 4096$.

This scheme not only increases the amount of "clerical" work in programming, but also constitutes a rich source of pitfalls. A translator should therefore be designed to ease the tedious task of base address assignment, and to provide checking facilities against errors.

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

(a) the compiler knows which register is used as base for each compiled address,
(b) the compiler can assure that each base register contains the desired base address during execution, and
(c) the number of times base addresses are reloaded into registers is reasonably small.

First, it must be decided whether the process of subdividing the program should be performed by the programmer or by the compiler. In the latter case, a fixed number of registers must be set aside to serve as base registers which the compiler has freely at its disposal. This was considered undesirable. Furthermore, a program using a number of segments much larger than that of available base registers would be

subject to considerable inefficiencies due to the reloading of base addresses at points which might have been ill chosen. It was therefore decided that the programmer should express explicitly which parts of his program were to constitute segments. He has then the possibility of organizing the program in a way which minimizes the number of cross references between segments.

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

5.1. Program Segmentation

Due to the fact that the language does not allow programs to modify themselves, branches are the only instructions referring to locations within program segments. Since control lies by its very nature in exactly one segment at any instant, it seemed appropriate to designate one fixed register to hold the base address of the program segment currently under execution. A branch leading into another segment must then always be preceded by an instruction loading that register with the base address of the destination segment. Register $R15$ was chosen for this purpose.

An obvious approach to the problem of segmentation requires the compiler to automatically generate a new segment, when the currently generated segment's length exceeds 4096 bytes. This solution was rejected for two reasons: (1) The programmer is not aware of the position of segment boundaries, and therefore has no way to minimize branches from one to another segment. (2) In most cases, the destination of an implicit branch (in if, case, while, and for statements) is not known to the compiler at the time of its generation. Therefore it is not known whether it will consist of one or two machine instructions.

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

5.2. Data Segmentation

In the case of data, the programmer is precisely aware of the amount of allocated

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

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

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

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

5.3. Program Loading

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

(a) reading program and data segments into memory,
(b) assigning the origin address of each segment to an entry in the segment address table, and
(c) transferring control to the program segment representing the outermost block.

5.4. Problems Connected With Input-Output Programming

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

(1) Input-output operations on the 360 are designed to use the interrupt mechanism to signal termination of processes performed by data channels and devices in parallel with CPU operations. In order to use the interrupt feature, it is necessary to create program status words (PSW) and store them in certain fixed locations of memory. A PSW contains the absolute address of a point in the program, which is a quantity that cannot be generated by a PL360 program.

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

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

(1) A facility is introduced to designate a segment as an interrupt service rou-

tine, with the effect that the compiler supplies information to the loader, causing the loader to assign the segment's base address to the appropriate PSW cell instead of the segment address table. The compiler itself terminates this segment with an LPSW instead of a BR instruction (cf. 5.6). This approach forces a programmer to make explicit the fact that an interrupt routine is conceptually a closed segment, and it circumvents the undesirable introduction of a facility to generate labels as manipulatable objects.

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

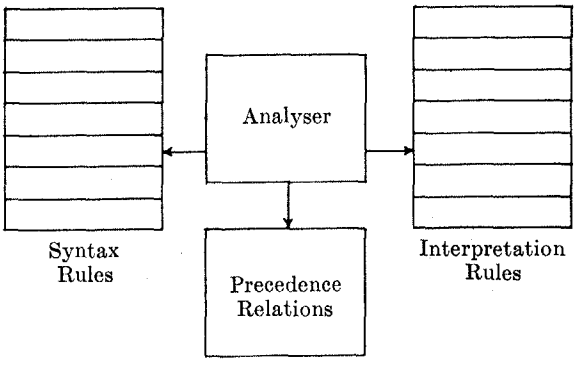
Usually, however, these facilities are not needed, because the program is executed in the environment of an operating system (whose choice is normally not up to the individual programmer) which executes programs in the program-mode where input-output instructions are not executable. The form which statements communicating with such an environment assume is determined by that particular environment and cannot be defined as part of the language proper (cf. also [6]).

6. Compiler Methodology

6.1. General Organization

The compiler is a strictly syntax directed one-pass translator. Its design served as a major test for the applicability of the techniques described in [4] to practical programming languages. The development of a precedence syntax with productions to which the meaning of the language could be properly attached, is no easy task. Interestingly enough, however, this design process provided many insights into the nature of various conceptual elements, led to their clarification and often simplification, and contributed a great deal to the systematic structure of the resulting language.

The algorithm for syntactic analysis constitutes the core of the compiler. It operates on the basis of a table containing the rules of syntax and a table containing the precedence relations among input tokens, and evokes the execution of an interpretation rule whenever a parsing step is taken. The input tokens are obtained by calling a procedure



"insymbol," which scans the sequence of input characters and yields as a result either a basic symbol of the language, an identifier, a number, or a string. It automatically suppresses comments. It should be noted that in the implemented language no equivalent for the **underlining** of basic symbols is provided, and that therefore a sequence of letters and digits, starting with a letter and not containing blanks, may constitute a basic symbol. Any such sequence must be matched by the insymbol routine against a table containing the representations of all "letter-symbols." If a match is found, the result is a basic symbol, otherwise an identifier. As a consequence, identifiers could not be constructed by the syntax analyser itself upon receiving merely a sequence of letters and digits. The consideration of numbers as tokens, on the other hand, was not a necessity but rather a convenience.

The syntax analyser makes use of a stack (called *"symbol stack"*) to store not yet reduced symbols. Whenever a reduction takes place, the interpretation rule corresponding to the applied syntactic rule is activated. These interpretation rules make use of a second stack (called *"value stack"*) to store information about each syntactic entity occurring in the reduction process. To each entry in the symbol stack corresponds an entry in the value stack, and vice versa. Ideally, an interpretation rule should exclusively reference data in those entries of the value stack which correspond to symbols in the symbol stack being reduced by the applying syntactic rule. This principle has been followed in the simple example presented in [4]. Here, however, a deviation from it was made by the introduction of conventional identifier tables, one containing identifiers denoting program points (labels), one for all declared identifiers.

6.2. Identifier Tables

The presence of identifier tables simplifies the search for identifiers and eliminates the need for the specific right recursive definition of the declaration structure used in [4]. The separation of the table into one containing declared identifiers and one containing labels has its reason in the fact that labels are the only identifiers which can occur in a statement before being defined in the program, and must therefore be treated differently as discussed below.

(1) ⟨3 cell identifier⟩ ::= ⟨identifier⟩
(2) ⟨function identifier⟩ ::= ⟨identifier⟩
(3) ⟨procedure identifier⟩ ::= ⟨identifier⟩
etc.

constitutes a violation of the requirement that in an unambiguous precedence grammar no two rules should have identical right parts. This violation required a slight complication of the analysis algorithm with the effect that an interpretation rule may cause an otherwise applicable syntactic rule to be rejected. In the given example, the interpretation rules specify that the considered identifier be located in the identifier table. If location is successful, then rule 1 is rejected unless the table indicates that the identifier indeed designates a 3 cell, rule 2 is rejected unless it designates a function, etc. This decision of the applicability of a syntactic rule on grounds of essentially semantic information reflects the argument that languages of "Algol type" are strictly speaking not context free.

The above identifier search implies that the entire block-structured identifier

table be searched. The following program demonstrates that labels cannot be subject to the same process, and that therefore

(4) ⟨label⟩ ::= ⟨identifier⟩

must not be a rule of the language.

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

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

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

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

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

6.3. Handling of Syntactic Errors

The syntax analysis algorithm described in [4] makes the assumption that analysed programs are syntactically valid. This assumption is not tenable in the practical world of computer programming. Syntactic errors are detected by the fact that for some string recognized as reducible there is no matching entry in the table of productions. After an error has been encountered, it is in most cases desirable to continue compilation in order that subsequent errors may be located and indicated. A method has to be devised to let the analysis algorithm proceed after having made some assumption about the nature of the error.

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

A second important consideration is that an incorrect construction should be

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

It should be noted that the use of two precedence functions instead of the precedence relations implies that the analysis algorithm is based on a condensation of the information contained in the matrix of relations [4]. This condensation relies on the assumption that empty relations can simply be ignored. The above considerations lead to the conclusion that for practical reasons it is advantageous to have the relation matrix at the disposal of the analyser rather than the functions.

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

There exist two places in the analysis process, where illegal constructions may be detected (cf. [4, p. 18]):

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

$$S_i \odot P_k$$

In this case a list I of *insertion symbols* is scanned. If for some m, $S_i \not\odot I_m$ and $I_m \not\odot P_k$, then I_m is inserted into the scanned string in front of P_k. Since this insertion may lead to a correct program (in about 90 percent of the tested cases it did), an according comment must be delivered to the programmer.

If for no m, $S_i \not\odot I_m$ and $I_m \not\odot P_k$, then the symbol P_k is stacked.

2. The value of the function

$$\mathit{Leftpart}\ (S_j \cdots S_i)$$

is undefined (Ω), i.e., there exists no syntactic rule whose rightpart is $S_j \cdots S_i$. This situation may occur even if for all k $(j \leq k < i)$ $S_k \not\odot S_{k+1}$.

In this case a table of *erroneous productions* is scanned for a right-part identical to $S_j \cdots S_i$. If a match is found, an error message corresponding to that rule can be printed, and the analysis can proceed with the statement:

$$i := j$$

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

```
procedure Invalid pair;
begin integer m;  m := 1;
    while m ≤ n ∧ (S_j ⊙ I_m ∨ I_m ⊙ P_k) do m := m+1;
    if m ≤ n then (P_k ··· P_{z+1}) := I_m cat (P_k ··· P_z)
end;
```

```
while P_k ≠ "⊥" do
begin i := j := j+1; S_j := P_k ; k := k+1;
    while S_j ⩾⊙ P_k do
    begin if S_j ⊙ P_k then Invalid pair;
        while S_{j-1} ≐⊙ S_j do j := j-1;
        t := Leftpart(S_j ··· S_i);
        S_j := if t = Ω then Error(S_j ··· S_i) else t;  i := j
    end
end
```

This technique enables a compiler to deliver meaningful error messages, because the analyser has found an erroneous production to be applicable, which was anticipated by the compiler designer who, in turn, knowing the reasons why programmers inadvertently write such a construction, was in a position to devise an appropriate comment. It was found to be beneficial to accompany the diagnostic message by the list of symbols currently stored in the parsing stack. These symbols represent all unfinished syntactic entries in the current parse, and may give the programmer valuable insights about his misuse of the language.

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

7. Development of the Compiler

At the time when the project to develop a compiler for PL360 was started, no 360 computer was available to the author, nor did the facilities promised with the forthcoming machine look too enticing to use. It was therefore decided to use the available Burroughs B5500 computer for the design and testing of the compiler, which was completed by the author within two months of part time work. It accepted a preliminary version of PL360 as described in [7] which contained the basic features of the presently described language.

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

The experiment of describing the compiling algorithm in PL360 itself proved to be the most effective test on the usefulness and appropriateness of the language, and it influenced the subsequent development of the language considerably. During this process, several features which seemed desirable were added to the language, and many were dropped again after having proved to be either dubious in value, inconsistent with the design criteria, or too involved and leading to misconceptions. The leading principle and guideline was to produce a conceptually simple language and to keep the number of features and facilities minimal. The "bootstrapping" method in combination with the described compiling technique proved to be very

successful for experimentation with and alteration of the language. The process of incorporation of a new feature consists of representing the new feature in the syntax of the language, and of defining the compiler actions corresponding to the new constructs in the form of additional interpretation rules. These rules must of course be denoted in terms of previously available facilities.

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

8. Performance

The development of a job control and supervisor program was undertaken in parallel with the construction of the compiler (cf. [6]). The following performance figures reflect the operation of the compiler under that supervisor. It should be noted that the supervisor considers the compiler in the same way as a regular user's program.

Size (in bytes, approximately)	
Supervisor	4 600
Job control	4 400
	9 000
Compiler program	16 000
Various compiler data	5 400
	21 400
Identifier tables, Output area	adjusted according to available core size.

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

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

Steps 1 and 3, constituting what is usually called "overhead," take 4.7 seconds execution time. Compilation proceeds at the speed of the card reader (1000 cpm). If the source program is read from tape (640 character records, unbuffered) and the program listing is suppressed, the compiler (about 1500 card records) recompiles itself in 39 seconds (with listing in 109 seconds). The time required to load the system initially is 2 seconds.

9. Reflections on the 360 Architecture

Based on the experiences drawn from the compiler development, it can be concluded that the objective to make direct machine programming more convenient by

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

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

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

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

$$\langle 3 \text{ cell} \rangle := \langle 3 \text{ primary} \rangle$$

is not part of the language, and that **string** was not adopted as a simple data type.

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

(4) The 360 instructions exhibit a remarkable consistency in the scheme of condition code setting, with the very peculiar exception of the TM instruction. It is obvious that the condition code plays a somewhat odd role in this language. It can only be altered by *hidden side effects* of arithmetic, logical, comparison, and certain other operations, and be tested by branch instructions. There is no way, however, to copy its value or even to perform operations on it.

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

ACKNOWLEDGMENTS. The author wishes to express his sincere thanks to Mr. J. W. Wells for his indispensable assistance. Mr. Wells recoded the compiler in its own

language, and developed the supporting monitor system. Thanks are also due to the GSG group at the Stanford Linear Accelerator Center for their generous provision of computer time. And finally, the support of the National Science Foundation under grant GP 4053 is gratefully acknowledged.

REFERENCES

1. WIRTH, N., AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9*, 6 (June 1966), 413–432.
2. AMDAHL, G. M., BLAAUW, G. A., AND BROOKS, F. P., JR. Architecture of the IBM System/360. *IBM J. Res. and Dev. 8* (April 1964), 87–101.
3. BLAAUW, G. A., ET AL. The structure of System/360. *IBM Syst. J. 3*, 2 (1964), 119–164.
4. WIRTH, N., AND WEBER, H. EULER: A generalization of ALGOL, and its formal definition: Part I. *Comm. ACM 9*, 1 (Jan. 1966), 13–23.
5. "IBM System/360 principles of operation," IBM Syst. Reference Libr. A22-6821-2.
6. WIRTH, N. (Ed.). The PL360 System. Tech. Rep. CS 68, Stanford U., Stanford, Calif., June 1967.
7. WIRTH, N. A programming language for the 360 computers. Tech. Rep. CS 33, Stanford U., Stanford, Calif., Dec. 1965.

RECEIVED DECEMBER, 1966; REVISED JUNE, 1967

Appendix I. Example of Compiled Code

The Assembly Language Code corresponding to the procedure *Magic square* (cf. Section 3) is as follows:

```
MAGICSQR  STH   0,NSQR
          LR    1,0
          MH    1,NSQR
          STH   1,NSQR
          LR    1,0
          A     1,ONE
          SRL   1,1
          LR    2,0
          L     5,ONE
          B     L7
L1        LR    3,ONE
          SLL   3,6
          LR    4,2
          SLL   4,2
          AR    4,3
          L     3,X(4)
          C     3,ZERO
          BC    8,L4
          S     1,ONE
          S     2,TWO
          C     1,ONE
          BC    11,L2
          AR    1,0
L2        C     2,ONE
          BC    11,L3
          AR    2,0
L3        LR    3,1
          SLL   3,6
          LR    4,2
          SLL   4,2
          AR    4,3
L4        ST    5,X(4)
          A     1,ONE
          CR    1,0
          BC    13,L5
          SR    1,0
L5        A     2,ONE
          CR    2,0
          BC    13,L6
          SR    1,0
L6        LA    5,1(5)
L7        CH    5,NSQR
          BC    12,L1
          BR    6
NSQR      DC    H
ZERO      DC    F''0''
ONE       DC    F''1''
TWO       DC    F''2''
```

Appendix II. Format of PL360 Programs

[Read left-hand column, then right-hand column on this page; next read left-hand column, then right-hand column on p. 71.]

⟨k reg⟩ ::= ⟨id⟩
⟨t cell id⟩ ::= ⟨id⟩
⟨proc id⟩ ::= ⟨id⟩
⟨func id⟩ ::= ⟨id⟩
⟨t cell⟩ ::= ⟨t cell id⟩
⟨t cell1⟩)
⟨t cell2⟩)
⟨t cell1⟩ ::= ⟨t cell2⟩ ⟨arith op⟩ ⟨t number⟩
⟨t cell3⟩ ⟨t number⟩
⟨t cell2⟩ ::= ⟨t cell3⟩ ⟨k reg⟩
⟨t cell3⟩ ::= ⟨t cell id⟩ (
⟨unary op⟩ ::= **abs**
neg
neg abs
⟨arith op⟩ ::= +
−
*
/
+ +
− −
⟨log op⟩ ::= **and**
or
xor
⟨shift op⟩ ::= **shla**
shra
shll
shrl
⟨k reg ass⟩ ::= ⟨k reg⟩ := ⟨t cell⟩
⟨k reg⟩ := ⟨t number⟩
⟨k reg⟩ := ⟨string⟩
⟨k reg⟩ := ⟨k reg⟩
⟨k reg⟩ := ⟨unary op⟩ ⟨t cell⟩
⟨k reg⟩ := ⟨unary op⟩ ⟨t number⟩
⟨k reg⟩ := ⟨unary op⟩ ⟨k reg⟩
⟨k reg⟩ := @ ⟨t cell⟩
⟨k reg ass⟩ ⟨arith op⟩ ⟨t cell⟩
⟨k reg ass⟩ ⟨arith op⟩ ⟨t number⟩
⟨k reg ass⟩ ⟨arith op⟩ ⟨k reg⟩
⟨k reg ass⟩ ⟨log op⟩ ⟨t number⟩
⟨k reg ass⟩ ⟨log op⟩ ⟨t cell⟩
⟨k reg ass⟩ ⟨log op⟩ ⟨k reg⟩
⟨k reg ass⟩ ⟨shift op⟩ ⟨t number⟩
⟨k reg ass⟩ ⟨shift op⟩ ⟨k reg⟩
⟨func1⟩ ::= ⟨func2⟩ ⟨t number⟩
⟨func2⟩ ⟨k reg⟩
⟨func2⟩ ⟨t cell⟩
⟨func2⟩ ⟨string⟩
⟨func2⟩ ::= ⟨func id⟩ (
⟨func1⟩,
⟨case seq⟩ ::= **case** ⟨k reg⟩ **of begin**
⟨case seq⟩ ⟨statement⟩ ;
⟨simple st⟩ ::= ⟨t cell⟩ := ⟨k reg⟩
⟨k reg ass⟩
null
goto ⟨id⟩
⟨proc id⟩
⟨func id⟩
⟨func1⟩)
⟨case seq⟩ **end**
⟨blockbody⟩ **end**
⟨rel op⟩ ::= <
=
>
< =
> =
¬ =
⟨not⟩ ::= ¬
⟨condition⟩ ::= ⟨k reg⟩ ⟨rel op⟩ ⟨t cell⟩
⟨k reg⟩ ⟨rel op⟩ ⟨t number⟩
⟨k reg⟩ ⟨rel op⟩ ⟨k reg⟩

```
                    ⟨k reg⟩ ⟨rel op⟩ ⟨string⟩
                    overflow
                    ⟨rel op⟩
                    ⟨t cell⟩
                    ⟨not⟩ ⟨t cell⟩
⟨comp cond⟩  ::= ⟨condition⟩
                    ⟨comp aor⟩ ⟨condition⟩
⟨comp aor⟩   ::= ⟨comp cond⟩ and
                    ⟨comp cond⟩ or
⟨cond then⟩  ::= ⟨comp cond⟩ then
⟨true part⟩  ::= ⟨simple st⟩ else
⟨while⟩      ::= while
⟨cond do⟩    ::= ⟨comp cond⟩ do
⟨ass step⟩   ::= ⟨k reg ass⟩ step ⟨t number⟩
⟨limit⟩      ::= until ⟨k reg⟩
                    until ⟨t cell⟩
                    until ⟨t number⟩
⟨do⟩         ::= do
⟨statement*⟩ ::= ⟨simple st⟩
                    if ⟨cond then⟩ ⟨statement*⟩
                    if ⟨cond then⟩ ⟨true part⟩ ⟨statement*⟩
                    ⟨while⟩ ⟨cond do⟩ ⟨statement*⟩
                    for ⟨ass step⟩ ⟨limit⟩ ⟨do⟩ ⟨statement*⟩
⟨statement⟩  ::= ⟨statement*⟩
⟨si t type⟩  ::= short integer
                    integer
                    logical
                    real
                    long real
                    byte
⟨t type⟩     ::= ⟨si t type⟩
                    array ⟨t number⟩ ⟨si t type⟩
⟨fill⟩       ::= ⟨string⟩
                    ⟨t number⟩
⟨t decl1⟩    ::= ⟨t type⟩ ⟨id⟩
                    ⟨t decl2⟩ ⟨id⟩
⟨t decl2⟩    ::= ⟨t decl4⟩ ,
⟨t decl3⟩    ::= ⟨t decl1⟩ =
⟨t decl4⟩    ::= ⟨t decl1⟩
                    ⟨t decl3⟩ ⟨fill⟩
⟨func dc1⟩   ::= function
                    ⟨func dc7⟩,
⟨func dc2⟩   ::= ⟨func dc1⟩ ⟨id⟩
⟨func dc3⟩   ::= ⟨func dc2⟩ (
⟨func dc4⟩   ::= ⟨func dc3⟩ ⟨t number⟩
⟨func dc5⟩   ::= ⟨func dc4⟩,
⟨func dc6⟩   ::= ⟨func dc5⟩ ⟨t number⟩
⟨func dc7⟩   ::= ⟨func dc6⟩ )
⟨syn dc1⟩    ::= ⟨t type⟩ ⟨id⟩ syn
                    ⟨si t type⟩ register ⟨id⟩ syn
                    ⟨syn dc3⟩ ⟨id⟩ syn
⟨syn dc2⟩    ::= ⟨syn dc1⟩ ⟨t cell⟩
                    ⟨syn dc1⟩ ⟨t number⟩
                    ⟨syn dc1⟩ ⟨k reg⟩
⟨syn dc3⟩    ::= ⟨syn dc2⟩ ,
⟨proc hd1⟩   ::= procedure
                    segment procedure
⟨proc hd2⟩   ::= ⟨proc hd1⟩ ⟨id⟩
⟨proc hd3⟩   ::= ⟨proc hd2⟩ (
⟨proc hd4⟩   ::= ⟨proc hd3⟩ ⟨k reg⟩
⟨proc hd5⟩   ::= ⟨proc hd4⟩ )
⟨proc hd6⟩   ::= ⟨proc hd5⟩ ;
⟨decl⟩       ::= ⟨t decl4⟩
                    ⟨func dc7⟩
                    ⟨syn dc2⟩
                    ⟨proc hd6⟩ ⟨statement*⟩
                    segment base ⟨k reg⟩
⟨label def⟩  ::= ⟨id⟩ :
⟨blockhead⟩  ::= begin
                    ⟨blockhead⟩ ⟨decl⟩ ;
⟨blockbody⟩  ::= ⟨blockhead⟩
                    ⟨blockbody⟩ ⟨statement⟩ ;
                    ⟨blockbody⟩ ⟨label def⟩
⟨program⟩    ::= . ⟨statement⟩ .
```

Appendix III. PL360 Syntax (In convenient precedence form)

```
 1   <K REG>        ::= <ID>
 2   <T CELL ID>    ::= <ID>
 3   <PROC ID>      ::= <ID>
 4   <FUNC ID>      ::= <ID>
 5   <T CELL>       ::= <T CELL ID>
 6                      <T CELL1>  )
 7                      <T CELL2>  )
 8   <T CELL1>      ::= <T CELL2>  <ARITH OP>  <T NUMBER>
 9                      <T CELL3>  <T NUMBER>
10   <T CELL2>      ::= <T CELL3>  <K REG>
11   <T CELL3>      ::= <T CELL ID>  (
12   <UNARY OP>     ::= ABS
13                      NEG
14                      NEG  ABS
15   <ARITH OP>     ::= +
16                      -
17                      *
18                      /
19                      +  +
20                      -  -
21   <LOG OP>       ::= AND
22                      OR
23                      XOR
24   <SHIFT OP>     ::= SHLA
25                      SHRA
26                      SHLL
27                      SHRL
28   <K REG ASS>    ::= <K REG>  :=  <T CELL>
29                      <K REG>  :=  <T NUMBER>
30                      <K REG>  :=  <STRING>
31                      <K REG>  :=  <K REG>
32                      <K REG>  :=  <UNARY OP>  <T CELL>
33                      <K REG>  :=  <UNARY OP>  <T NUMBER>
34                      <K REG>  :=  <UNARY OP>  <K REG>
35                      <K REG>  :=  @  <T CELL>
36                      <K REG ASS>  <ARITH OP>  <T CELL>
37                      <K REG ASS>  <ARITH OP>  <T NUMBER>
38                      <K REG ASS>  <ARITH OP>  <K REG>
39                      <K REG ASS>  <LOG OP>  <T NUMBER>
40                      <K REG ASS>  <LOG OP>  <T CELL>
41                      <K REG ASS>  <LOG OP>  <K REG>
42                      <K REG ASS>  <SHIFT OP>  <T NUMBER>
43                      <K REG ASS>  <SHIFT OP>  <K REG>
44   <FUNC1>        ::= <FUNC2>  <T NUMBER>
45                      <FUNC2>  <K REG>
46                      <FUNC2>  <T CELL>
47                      <FUNC2>  <STRING>
48   <FUNC2>        ::= <FUNC ID>
49                      <FUNC1>  ,
50   <CASE SEQ>     ::= CASE  <K REG>  OF  BEGIN
51                      <CASE SEQ>  <STATEMENT>
52   <SIMPLE ST>    ::= <T CELL>  :=  <K REG>
53                      <K REG ASS>
54                      NULL
55                      GOTO  <ID>
56                      <PROC ID>
57                      <FUNC ID>
58                      <FUNC1>
```

```
                        <CASE SEQ>  END
                        <BLOCKBODY>  END
<REL OP>        ::= <
                    =
                    >
                    <  =
                    >  =
                    ¬  =
<NOT>           ::= ¬
<CONDITION>     ::= <K REG>  <REL OP>  <T CELL>
                    <K REG>  <REL OP>  <T NUMBER>
                    <K REG>  <REL OP>  <K REG>
                    <K REG>  <REL OP>  <STRING>
                    OVERFLOW
                    <REL OP>
                    <T CELL>
                    <NOT>  <T CELL>
<COMP COND>     ::= <CONDITION>
                    <COMP AOR>  <CONDITION>
<COMP AOR>      ::= <COMP COND>  AND
                    <COMP COND>  OR
<COND THEN>     ::= <COMP COND>  THEN
<TRUE PART>     ::= <SIMPLE ST>  ELSE
<WHILE>         ::= WHILE
<COND DO>       ::= <COMP COND>  DO
<ASS STEP>      ::= <K REG ASS>  STEP  <T NUMBER>
<LIMIT>         ::= UNTIL  <K REG>
                    UNTIL  <T CELL>
                    UNTIL  <T NUMBER>
<DO>            ::= DO
<STATEMENT*>    ::= <SIMPLE ST>
                    IF  <COND THEN>  <STATEMENT*>
                    IF  <COND THEN>  <TRUE PART>  <STATEMENT*>
                    <WHILE>  <COND DO>  <STATEMENT*>
                    FOR  <ASS STEP>  <LIMIT>  <DO>  <STATEMENT*>
<STATEMENT>     ::= <STATEMENT*>
<SI T TYPE>     ::= SHORT  INTEGER
                    INTEGER
                    LOGICAL
                    REAL
                    LONG  REAL
                    BYTE
                    CHARACTER
<T TYPE>        ::= <SI T TYPE>
                    ARRAY  <T NUMBER>  <SI T TYPE>
<T DECL1>       ::= <T TYPE>  <ID>
                    <T DECL2>  <ID>
<T DECL2>       ::= <T DECL7>  ,
<T DECL3>       ::= <T DECL1>  =
<T DECL4>       ::= <T DECL3>  (
                    <T DECL5>  ,
<T DECL5>       ::= <T DECL4>  <T NUMBER>
                    <T DECL4>  <STRING>
<T DECL6>       ::= <T DECL3>
<T DECL7>       ::= <T DECL1>
                    <T DECL6>  <T NUMBER>
                    <T DECL6>  <STRING>
                    <T DECL5>  )
<FUNC DC1>      ::= FUNCTION
                    <FUNC DC7>
```

(Continued on following page)

Appendix III—*Continued*

```
119     <FUNC DC2>    ::= <FUNC DC1>  <ID>
120     <FUNC DC3>    ::= <FUNC DC2>  (
121     <FUNC DC4>    ::= <FUNC DC3>  <T NUMBER>
122     <FUNC DC5>    ::= <FUNC DC4>  ,
123     <FUNC DC6>    ::= <FUNC DC5>  <T NUMBER>
124     <FUNC DC7>    ::= <FUNC DC6>  )
125     <SYN DC1>     ::= <T TYPE>  <ID>  SYN
126                       <SI T TYPE>  REGISTER  <ID>  SYN
127                       <SYN DC3>  <ID>  SYN
128     <SYN DC2>     ::= <SYN DC1>  <T CELL>
129                       <SYN DC1>  <T NUMBER>
130                       <SYN DC1>  <K REG>
131     <SYN DC3>     ::= <SYN DC2>
132     <SEG HEAD>    ::= SEGMENT
133     <PROC HD1>    ::= PROCEDURE
134                       <SEG HEAD>  PROCEDURE
135     <PROC HD2>    ::= <PROC HD1>  <ID>
136     <PROC HD3>    ::= <PROC HD2>  (
137     <PROC HD4>    ::= <PROC HD3>  <K REG>
138     <PROC HD5>    ::= <PROC HD4>  )
139     <PROC HD6>    ::= <PROC HD5>  ;
140     <DECL>        ::= <T DECL7>
141                       <FUNC DC7>
142                       <SYN DC2>
143                       <PROC HD6>  <STATEMENT*>
144                       <SEG HEAD>  BASE  <K REG>
145     <LABEL DEF>   ::= <ID>  :
146     <BLOCKHEAD>   ::= BEGIN
147                       <BLOCKHEAD>  <DECL>  ;
148     <BLOCKBODY>   ::= <BLOCKHEAD>
149                       <BLOCKBODY>  <STATEMENT>  ;
150                       <BLOCKBODY>  <LABEL DEF>
151     <PROGRAM>     ::= .  <STATEMENT>  .
```