PL/I

TECHNICAL REPORT

TR 25.084
28 June 1968

ULD VERSION II

CONCRETE SYNTAX OF PL/I

K. ALBER
P. OLIVA
G. URSCHLER

IBM LABORATORY VIENNA

IBM LABORATORY VIENNA, Austria

CONCRETE SYNTAX OF PL/I

by

K. ALBER
P. OLIVA
G. URSCHLER

ABSTRACT

This report supplements the semantical definition of PL/I given in "Abstract Syntax and Interpretation of PL/I" and the specification of abstract syntax given in "Translation of PL/I into Abstract Syntax" (IBM Laboratory Vienna, TR 25.082 and TR 25.086) by a syntactical definition. The syntactical form of concrete PL/I program text is defined by means of an extended Backus notation, which is described by a meta syntax.

Locator Terms for IBM Subject Index

PL/I
Backus Notation
Formal Definition
Syntax, concrete
21 PROGRAMMING

TR 25.084
28 June 1968

IBM LABORATORY VIENNA, Austria

CONCRETE SYNTAX OF PL\I

by

K. ALBER
P. OLIVA
G. URSCHLER

ABSTRACT

This report supplements the semantical definition of PL\I given in "Abstract Syntax and Interpretation of PL\I" and the specification of abstract syntax given in "Translation of PL\I into Abstract Syntax" (IBM Laboratory Vienna, TR 25.082 and TR 25.086) by a syntactical definition. The syntactical form of concrete PL\I program text is defined by means of an extended Backus notation, which is described by a meta syntax.

Locator Terms for IBM Publication Index

PL\I
Backus Notation
Formal Definition
Syntax, concrete
Z) PROGRAMMING

TR 25.084
25 June 1968

PREFACE

This document is part of a series of documents which represent the formal definition of syntax and semantics of PL/I issued by 28 June 1968:

/1/ LUCAS, P., LAUER, P., STIGLEITNER, H.: Method and Notation for the Formal Definition of Programming Languages.-
IBM Laboratory Vienna, Techn. Report TR 25.087.

/2/ FLECK, M., NEUHOLD, E.: Formal Definition of the PL/I Compile Time Facilities.-
IBM Laboratory Vienna, Techn. Report TR 25.080.

/3/ ALBER, K., OLIVA, P., URSCHLER, G.: Concrete Syntax of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.084.

/4/ ALBER, K., OLIVA, P.: Translation of PL/I into Abstract Text.-
IBM Laboratory Vienna, Techn. Report TR 25.086.

/5/ LUCAS, P., ALBER, K., BANDAT, K., BEKIĆ, H., OLIVA, P., WALK, K., ZEISEL, G.: Informal Introduction to the Abstract Syntax and Interpretation of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.083.

/6/ WALK, K., ALBER, K., BANDAT, K., BEKIĆ, H., CHROUST, G., KUDIELKA, V., OLIVA, P., ZEISEL,G.: Abstract Syntax and Interpretation of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.082.

The method and notation for these documents are essentially taken over from the first version of a formal definition of PL/I issued by the Vienna Laboratory:

/7/ PL/I Definition Group of the Vienna Laboratory : Formal Definition of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.071, 30 December 1966.

/ / ALBER, K.: Syntactical Description of PL/I Text and its Translation into Abstract Normal Form.-
IBM Laboratory Vienna, Techn. Report TR 25.074, 14 April 1967.

An outline of the method is given in /1/, which document also contains the appropriate references to the relevant literature. The basic ideas and their application to PL/I have been made available through several workshops on the formal definition of PL/I, and presentations inside and outside IBM.

The language defined in this present version is PL/I as specified in the official PL/I Language Specifications Form No. Y33-6003 with the exception of the following features which are not included:

optimizing attributes (they are included in the concrete syntax but not in the abstract syntax; they are only tested for compatibility with other attributes and used for implication of default attributes),

implicit conversion between offsets and pointers,

the REFER option,

the implicit rules for ordering initializing actions in the prologues of blocks and procedures.

The draft for this document was completed by 15 March 1968. It has been subject to validation by members of the PL/I Language Department of IBM UK Laboratories Hursley, England. The results of the checking effort conducted in Hursley have been taken into account in this present corrected form.

The formal definition given here includes more details than are given in the Specifications. These details have been confirmed as far as possible by the PL/I Language Department Hursley during the validation process. Some amendments and clarifications to the Specifications were generated during this process and will be published as Technical News Letters to the Specifications.

Contribution to the document:

Authors: their main contributions are given by chapters

K. Alber, P. Oliva	3.
G. Urschler	2.

Coordination of production: F. Schwarzenberger, H. Hoja, W. Pachl

Production of the cross-reference index: K.F. Koch

Typing: H. Deim, W. Schatzl

Special graphics: G. Lehmayer

Validation: P. Seaman, R.W. Thomas

C O N T E N T S

C O N T E N T S

1. INTRODUCTION

This document is a supplement to /4/ and /6/ and contains the complete syntactical description of concrete PL/I text. It is intended for readers interested in syntactical questions who are familiar with the syntax notation of the Backus Normal Form (as used in the Revised Report on the Algorithmic Language ALGOL 60). Readers interested in the translation of concrete PL/I programs into abstract programs who are familiar with the concept of abstract objects, are referred to /4/.

The concrete syntax of PL/I is given by a set of formal production rules for writing PL/I program text in a 60-character alphabet. These rules are written in an extended Backus notation. The syntactic form and the meaning of this extended Backus notation is given in chapter 2, the production rules are listed in chapter 3.

Compile time facilities /2/ are not included in this paper. That means that not a program containing compile time macros, but a text possibly produced by them, is considered.

The concrete syntax presented in chapter 3 is a revision of the concrete syntax given in /8/. The relatively short form is a result of nearly two years' cooperation of Hursley and Vienna Laboratories.

1. INTRODUCTION

2. SYNTAX NOTATION

2.1 Semantics of the Extended Backus Notation

The attempt to give a clear and readable description of the concrete syntax of PL/I has been accompanied by the search for new notation possibilities. The extended Backus notation, which has been proved to meet all demands, is a short form of the wellknown Backus notation.

In the following the meaning of the extended forms is explained by giving the equivalent forms in Backus notation.

Point of departure shall be a grammar, whose production rules have the general form:

$$V ::= S_1 \mid S_2 \mid \ldots \mid S_n$$

('V' is to be replaced by one of the alternatives 'S_1' or 'S_2' or ... or 'S_n')

Note: In this chapter with 'V' variables, with 'S_i' arbitrary strings and with 'T_j' strings different from the null-string are denoted. Each of these strings may consist of a certain number of not nearer specified syntactical units, denoted by 'U_r'.

Then the introduction of the metalinguistic signs '{', '}', '[', ']', '•' (the last is a fat dot) is determined according to the following definitions:

(i) '$V ::= S_1\ T_1\ S_2 \mid S_1\ T_2\ S_2 \mid \ldots \mid S_1\ T_n\ S_2$'

may be replaced by

'$V ::= S_1\ \{\ T_1 \mid T_2 \mid \ldots \mid T_n\ \}\ S_2$' and vice versa

(note, that this rule remains, valid also for the case n = 1)

Example: goto-statement ::= { GOTO | GO TO } reference ;

instead of: goto-statement ::= GOTO reference ; | GO TO reference ;

(ii) 'V ::= S_1 S_2 | S_1 T_1 S_2 | ... | S_1 T_n S_2'

may be replaced by

'V ::= S_1 [T_1 | ... | T_n] S_2' and vice versa.

Example: return-statement ::= RETURN [(expression)] ;

instead of:return-statement ::= RETURN ; | RETURN (expression) ;

(iii) 'V ::= U | V U' or

'V ::= U | U V'

may be replaced by

'V ::= U••• ' and vice versa.

Note : For the inversion it is to be regarded that if 'U•••' occurs in the grammar and no production rule has the form 'V ::= U••• ', this missing rule is to be added to the grammar with a not yet used variable 'V' before the replacement can be performed.

Example: integer ::= digit•••

instead of:integer ::= digit | integer digit

(iv) 'V ::= S_1 T_1 [{ T_2 T_1 } •••] S_2' may be replaced by

'V ::= S_1 { T_2 • T_1••• } S_2' and vice versa.

Note: Instead of 'S_1 [{ T_2 • T_1••• }] S_2' also

'S_1 [T_2 • T_1•••] S_2' may be written.

Example: declarationlist ::= { , • declaration••• }

instead of: declarationlist ::= declaration [{ , declaration }•••

2.2 The Meta Syntax of the Extended Backus Notation

The expressions 'syntactical unit' and 'string of syntactical units' (called 'unit' and 'sequence' respectively in the following) have not been specified in the last section. We now define them together with the general form of the production rules of the concrete PL/I syntax recursively by means of a meta syntax. The meta syntax itself is written in the well-known Backus normal form.

To uphold the Backus notation also formally we use the metalinguistic signs '<', '>', '::=', '|' in the meta syntax. Therefore we are obliged to change the notation for similar syntactic or PL/I signs. We adapt the same convention as in chapter 3, that each ambiguous sign is marked on the lower syntactic level by a further underlining. This signifies, for this and only for this sub-chapter, that the PL/I signs for colon, equal and or-sign get the forms ':', '=', '|' while the or-sign of the PL/I production rules is denoted by '|'.

Meta Syntax

```
<prod-rule> ::= <not-var>␣:=␣<definition>
<definition>::= <sequence>␣|␣<definition> | <sequence>
<sequence>  ::= <unit>␣<sequence> | <unit>
<unit>      ::= <not-var> | <not-const> |
                {␣<definition>␣} | [␣<definition>␣] | <unit>●●● |
                {␣<not-const>␣●␣<unit>●●●␣} | [␣<not-const>␣●␣<unit>●●●␣]
<not-var>   ::= <sm-letter><not-var> |
                <sm-letter>-<not-var> | <sm-letter>
<sm-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
<not-const> ::= <PL/I-symb><not-const> | <PL/I-symb>
<PL/I-symb> ::= 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 |
                blank | _ | = | + | - | * | / | ( | ) | , | . | ; |
                : | & | | | ¬ | > | < | ? | % | '
```

Note: Each '␣' occurring in the above rules stands for one or more empty spaces (it is the sign ' ', which is not to be mixed up with the PL/I sign 'blank') in the described production rules. The appearance of '␣' in the production rule <sequence> is essential for the unambiguity of the meta syntax. All other occurrences give only a certain freedom of formatting the production rules of the concrete PL/I syntax.

2.3 Generation of a Concrete Program Text

2.3.1 The normal generation process

First of all any implementation must provide production rules for the three implementation dependent notation variables external-option, io-option, and extralingual-character. Since PL/I has context dependent rules for the insertion of blanks and comments, which cannot be expressed by production rules of the form described in 2.1 and 2.2, the generation of a concrete PL/I program text has to be performed in four steps:

1. Starting with the notation variable "program", replacements are to be performed according to the higher level production rules listed in 3.1. This process is to be continued as long as any higher level production rule is applicable.

   It ends up with a text consisting of "PL/I words", which are listed in 3.3. In this respect, all those sequences of PL/I symbols which in the production rules are not separated by empty space are assumed to compose words(notation constants) and not to be split up into their single symbols. So a word is one of the following:

   a single PL/I symbol,
   a keyword, which is a sequence of upper case letters,
   one of the eight composite operators:
   ** || >= <= ¬> ¬= ¬< ->,
   one of the eight notation variables

   identifier,
   integer,
   isub,
   real-constant,
   imaginary-constant,
   simple-string-constant,
   sterling-constant,
   picture-specification.

2. Now "spaces" are inserted into the generated text according to the following rule:

   The 24 words

   = + - * / ( ) , . ; : & | ¬ < > ** || >= <= ¬> ¬=
   ¬< ->

   are "delimiters", all other words "non-delimiters". Between two adjacent non-delimiters the notation variable "space" must be inserted, between other combinations of words or following the last word of the complete program the notation variable "space" may be inserted.

   The production rules for "space" are listed in 3.2.3.

3. Now the replacement is continued by application of the lower level production rules listed in 3.2.

4. Finally all notation constants are split up into their single symbols.

   The complete process ends up with a text consisting of the symbols of the 60 character PL/I alphabet , i.e.

   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 _ blank = + - * / ( ) ,
   . ; : ' & | ¬ > < ? %

   and extralingual characters.

2.3.2 Auxiliary rules for additional facilities

PL/I contains two facilities which in the one case would lengthen unnecessarily the production rules and in the other case cannot be expressed by context independent production rules. Both facilities allow a program text to be replaced by a shorter one, without changing the semantical meaning.

2.3.2.1 Keyword abbreviations

The following abbreviations may be inserted instead of the corresponding keywords. This replacement has to be performed before step 3 of the generation process described in 2.3 is performed:

keywords:	abbreviations:
ABNORMAL	ABNL
AUTOMATIC	AUTO
BINARY	BIN
BUFFERED	BUF
CHARACTER	CHAR
COLUMN	COL
COMPLEX	CPLX
CONTROLLED	CTL
CONVERSION	CONV
DECIMAL	DEC
DECLARE	DCL
DEFINED	DEF
ENVIRONMENT	ENV
EXCLUSIVE	EXCL
EXTERNAL	EXT
FIXEDOVERFLOW	FOFL
INITIAL	INIT
INTERNAL	INT
IRREDUCIBLE	IRRED
NOCONVERSION	NOCONV
NOFIXEDOVERFLOW	NOFOFL
NOOVERFLOW	NOOFL
NOSTRINGRANGE	NOSTRG
NOSUBSCRIPTRANGE	NOSUBRG
NOUNDERFLOW	NOUFL
NOZERODIVIDE	NOZDIV
OVERFLOW	OFL
PICTURE	PIC
POINTER	PTR
POSITION	POS
PROCEDURE	PROC
REDUCIBLE	RED
SEQUENTIAL	SEQL
STRINGRANGE	STRG

keywords:	abbreviations:
SUBSCRIPTRANGE	SUBRG
UNBUFFERED	UNBUF
UNDEFINEDFILE	UNDF
UNDERFLOW	UFL
VARYING	VAR
ZERODIVIDE	ZDIV

2.3.2.2 Multiple closure of blocks and groups

Assume, that all four steps of the generation process described in 2.3.1 including the insertion of abbreviated keywords have been terminated.

Then a part of this program text is called a compound if it could have been generated by means of the following production rules:

```
compound ::= compoundhead [ compound ] compoundend

compoundhead ::= [ prefixlist ]
    { [ labellist ] BEGIN [ OPTIONS ( { , • external-option••• } ) ] |
    entry-namelist { PROC | PROCEDURE } [ parameterlist ]
                                        [ procedure-optionslist ] |
    [ labellist ] DO [ do-specification | WHILE ( expression ) ] }
    ; [ sentence••• ]

compoundend ::= [ prefixlist ] [ labellist ] END ;
```

Before the rightmost semicolon of a compound, i.e. between 'END' and ';' , one identifier of the labellist or entry-namelist of the appertaining compound-head may be inserted.

Provided that a compound actually has such a compoundend, e. g. 'END identifier$_1$;' it is allowed to omit an immediately preceding compound-end with no prefix- or labellist (i. e. a compoundend immediately to the left) if identifier$_1$ does not occur in the labellist or entry-namelist of the compoundhead appertaining to the omitted compoundend.

2.3.3 Programs in the 48 character set

It is possible to write PL/I programs in the following 48 character set:

```
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 blank = + - * / ( ) . , '
```

If the program shall be written in this character set, in addition to the processes described in 2.3.1 and 2.3.2 the following rules have to be obeyed:

1) From the production rules for "letter", "alphameric-character", "string-character" and "comment-symbol" the following 12 symbols have to be deleted:

@ # _ ; : & | ¬ > < ? %

2) The following 13 PL/I words have to be handled as notation variables and to be replaced by means of the (higher level) production rules:

```
;   ::=  ,.          >=  ::=  GE
¬   ::=  NOT         <=  ::=  LE
&   ::=  AND         ¬>  ::=  NG
|   ::=  OR          ¬<  ::=  NL
>   ::=  GT          ¬=  ::=  NE
<   ::=  LT          ||  ::=  CAT
                     ->  ::=  PT
```

For the insertion of spaces the word ,. is handled as a delimiter and the other 12 words resulting from these replacements as non-delimiters.

3) The 12 sequences of letters

NOT, AND, OR, GT, LT, GE, LE, NG, NL, NE, CAT, PT

are "reserved words", i.e. no identifier must finally be replaced by any of these sequences.

4) In the final text, each colon ':' is to be replaced:

a) when immediately following a dot '.' by means of the production rules

```
:  ::=  space .
```

b) else by means of the production rule

```
:  ::=  . .
```

3. CONCRETE SYNTAX

3.1 Higher level production rules

(1) program ::=
    procedure•••

(2) procedure ::=
    [prefixlist] entry-namelist PROCEDURE
    [parameterlist] [procedure-optionslist] ; sentencelist

(3) entry-namelist ::=
    { identifier : }•••

(4) parameterlist ::=
    ({ , • identifier••• })

(5) procedure-optionslist ::=
    { function-attribute | RECURSIVE |
    OPTIONS ({ , • external-option••• }) }•••

(6) sentencelist ::=
    [sentence•••] end-clause

(7) end-clause ::=
    [prefixlist] [labellist] END ;

(8) sentence ::=
    statement | procedure | entry | declaration-sentence |
    format-sentence

3.1.1 Declarations:

(9) declaration-sentence ::=
    [labellist] DECLARE declarationlist ;

(10) declarationlist ::=
{ , • declaration••• }

(11) declaration ::=
[integer] { identifier | (declarationlist) }
[dimension-attribute] [attribute•••]

(12) dimension-attribute ::=
([, • bound-pair•••])

(13) bound-pair ::=
[expression :] expression | *

3.1.1.1 Attributes:

(14) attribute ::=
data-attribute | non-data-attribute |
scope-attribute | like-attribute

(15) data-attribute ::=
arithmetic-attribute | string-attribute | VARYING |
picture-attribute | area-attribute | label-attribute |
POINTER | offset-attribute | TASK | EVENT | CELL |
storage-class-attribute | defined-attribute | based-attribute |
PACKED | ALIGNED | SECONDARY | NORMAL | ABNORMAL |
initial-attribute

(16) arithmetic-attribute ::=
{ REAL | COMPLEX | DECIMAL | BINARY | FLOAT |
FIXED } [(integer [, signed-integer])]

(17) signed-integer ::=
[+ | -] integer

(18) string-attribute ::=
{ BIT | CHARACTER } [({ expression | * })]

```
(19)  picture-attribute ::=
          PICTURE picture-specification

(20)  area-attribute ::=
          AREA [ ( { expression | * } ) ]

(21)  label-attribute ::=
          LABEL [ ( { , • identifier••• } ) ]

(22)  offset-attribute ::=
          OFFSET [ ( reference ) ]

(23)  storage-class-attribute ::=
          AUTOMATIC | STATIC | CONTROLLED

(24)  defined-attribute ::=
          DEFINED basic-reference | POSITION ( integer )

(25)  based-attribute ::=
          BASED [ ( reference ) ]

(26)  initial-attribute ::=
          INITIAL { initial-call | initial-itemlist }

(27)  initial-call ::=
          CALL identifier [ argumentlist ]

(28)  initial-itemlist ::=
          { { , • initial-item••• } )

(29)  initial-item ::=
          initial-iteration | initial-constant | simple-string-constant | *

(30)  initial-iteration ::=
          ( expression ) { initial-constant | initial-itemlist }

(31)  initial-constant ::=
          identifier | replicated-string-constant |
          arithmetic-initial-constant | sterling-constant
```

```
(32)  arithmetic-initial-constant ::=
          [ + | - ] real-constant
          [ { + | - } imaginary-constant ] |
          [ + | - ] imaginary-constant

(33)  non-data-attribute ::=
          entry-name-attribute | file-name-attribute |
          BUILTIN | generic-attribute

(34)  entry-name-attribute ::=
          ENTRY [ ( descriptorlist ) ] |
          RETURNS ( function-attribute••• ) |
          { USES | SETS } ( { , • uses-sets-item••• } ) |
          REDUCIBLE | IRREDUCIBLE

(35)  descriptorlist ::=
          descriptor [ , descriptorlist ]

(36)  descriptor ::=
          [ integer ] [ dimension-attribute ] [ attribute••• ]

(37)  function-attribute ::=
          arithmetic-attribute | string-attribute | VARYING |
          picture-attribute | area-attribute | POINTER | offset-attribute

(38)  uses-sets-item ::=
          unsubscripted-reference | integer | *

(39)  file-name-attribute ::=
          FILE | file-attribute | ENVIRONMENT ( io-option )

(40)  file-attribute ::=
          STREAM | RECORD | INPUT | OUTPUT | UPDATE |
          SEQUENTIAL | DIRECT | BUFFERED | UNBUFFERED |
          KEYED | PRINT | BACKWARDS | EXCLUSIVE

(41)  generic-attribute ::=
          GENERIC ( declarationlist )
```

(42) scope-attribute ::=
INTERNAL | EXTERNAL

(43) like-attribute ::=
LIKE unsubscripted-reference

3.1.1.2 Formats:

(44) format-sentence ::=
[prefixlist] labellist FORMAT formatlist ;

(45) formatlist ::=
({ , • format••• })

(46) format ::=
format-iteration | format-item

(47) format-iteration ::=
{ integer | (expression) } { format-item | formatlist }

(48) format-item ::=
data-format | control-format | remote-format

(49) data-format ::=
real-format | complex-format | string-format |
picture-format

(50) real-format ::=
{ E | F } (expression [, expression [, expression]])

(51) complex-format ::=
C ({ real-format | picture-format }
[, real-format | , picture-format])

(52) string-format ::=
{ B | A } [(expression)]

(53) picture-format ::=
P picture-specification

(54) control-format ::=
{ COLUMN | LINE | PAGE | SKIP | X } [(expression)]

(55) remote-format ::=
R (reference)

3.1.2 Statements:

(56) entry ::=
entry-namelist ENTRY [parameterlist] [attribute•••] ;

(57) statement ::=
if-statement | unconditional-statement

(58) unconditional-statement ::=
begin-block | simple-statement

(59) simple-statement ::=
[prefixlist] [labellist] proper-statement

(60) prefixlist ::=
{ ({ , • prefix-element••• }) : }•••

(61) prefix-element ::=
prefix | no-prefix | check-condition |
no-check-condition

(62) prefix ::=
CONVERSION | FIXEDOVERFLOW | OVERFLOW | SIZE |
SUBSCRIPTRANGE | STRINGRANGE | UNDERFLOW | ZERODIVIDE

(63) no-prefix ::=
NOCONVERSION | NOFIXEDOVERFLOW | NOOVERFLOW |
NOSIZE | NOSUBSCRIPTRANGE | NOSTRINGRANGE | NOUNDERFLOW |
NOZERODIVIDE

(64) labellist ::=
{ { identifier | initial-label } : }•••

(65) initial-label ::=
[{ identifier [({ , • signed-integer••• })] • }•••]
identifier ({ , • signed-integer••• })
[{ • identifier }•••]

(66) proper-statement ::=
group | goto-statement | call-statement |
return-statement | wait-statement | delay-statement |
exit-statement | stop-statement | assignment-statement |
allocate-statement | free-statement | on-statement |
revert-statement | signal-statement | open-statement |
close-statement | stream-io-statement | record-io-statement |
display-statement | null-statement

(67) null-statement ::=
;

3.1.2.1 Block and groups:

(68) begin-block ::=
[prefixlist] [labellist] BEGIN
[OPTIONS ({ , • external-option••• })] ; sentencelist

(69) group ::=
simple-group | iterated-group

(70) simple-group ::=
DO ; sentencelist

(71) iterated-group ::=
DO [do-specification | WHILE (expression)] ;
sentencelist

(72) do-specification ::=
reference = { , • specification••• }

(73) specification ::=
expression [BY expression [TO expression] |
TO expression [BY expression]]
[WHILE (expression)]

3.1.2.2 Flow of control statements:

(74) if-statement ::=
if-clause statement |
if-clause balanced-statement ELSE statement

(75) if-clause ::=
[prefixlist] [labellist] IF expression THEN

(76) balanced-statement ::=
if-clause balanced-statement ELSE
balanced-statement |
unconditional-statement

(77) goto-statement ::=
{ GOTO | GO TO } reference ;

(78) call-statement ::=
CALL identifier [argumentlist] [call-optionslist] ;

(79) call-optionslist ::=
[TASK [(reference)] | PRIORITY (expression) |
EVENT (reference) }•••

```
(80)  argumentlist ::=
          ( { , • expression••• } )

(81)  return-statement ::=
          RETURN [ ( expression ) ] ;

(82)  wait-statement ::=
          WAIT ( { , • reference••• } ) [ ( expression ) ] ;

(83)  delay-statement ::=
          DELAY ( expression ) ;

(84)  exit-statement ::=
          EXIT ;

(85)  stop-statement ::=
          STOP ;
```

3.1.2.3 Storage manipulating statements:

```
(86)  assignment-statement ::=
          { , • reference••• } = expression [ , BY NAME ] ;

(87)  allocate-statement ::=
          ALLOCATE { , • { based-allocate-item |
          controlled-allocate-item }••• } ;

(88)  based-allocate-item ::=
          identifier { SET ( reference ) [ IN ( reference ) ] |
          IN ( reference ) [ SET ( reference ) ] }

(89)  controlled-allocate-item ::=
          [ integer ] identifier [ dimension-attribute ]
          [ { string-attribute | area-attribute | CELL |
          initial-attribute }••• ]
```

(90) free-statement ::=
FREE { , • { [reference ->] identifier [IN (reference)] }•••) ;

3.1.2.4 Condition handling statements:

(91) on-statement ::=
ON condition [SNAP] { unconditional-statement |
SYSTEM ; }

(92) revert-statement ::=
REVERT condition ;

(93) signal-statement ::=
SIGNAL condition ;

(94) condition ::=
prefix | check-condition | AREA | named-io-condition |
ERROR | FINISH | programmer-named-condition

(95) check-condition ::=
CHECK ({ , • unsubscripted-reference••• })

(96) no-check-condition ::=
NOCHECK ({ , • unsubscripted-reference••• })

(97) named-io-condition ::=
io-condition (identifier)

(98) io-condition ::=
ENDFILE | ENDPAGE | KEY | NAME | RECORD | TRANSMIT |
UNDEFINEDFILE

(99) programmer-named-condition ::=
CONDITION (identifier)

3.1.2.5 Input and output statements:

```
(100)  open-statement ::=
           OPEN { , • open-optionslist••• } ;

(101)  open-optionslist ::=
           { file-attribute | FILE ( identifier ) | IDENT ( expression ) |
           TITLE ( expression ) | LINESIZE ( expression ) |
           PAGESIZE ( expression ) }•••

(102)  close-statement ::=
           CLOSE { , • close-optionslist••• } ;

(103)  close-optionslist ::=
           { FILE ( identifier ) | IDENT ( expression ) }•••

(104)  stream-io-statement ::=
           { GET | PUT } stream-optionslist ;

(105)  stream-optionslist ::=
           { FILE ( identifier ) | STRING ( reference ) | data-specification |
           COPY | SKIP [ ( expression ) ] | PAGE | LINE ( expression ) }•••

(106)  data-specification ::=
           data-directed | edit-directed | list-directed

(107)  data-directed ::=
           DATA [ ( datalist ) ]

(108)  edit-directed ::=
           EDIT { ( datalist ) formatlist }•••

(109)  list-directed ::=
           LIST ( datalist )

(110)  datalist ::=
           { , • datalist-element•••}

(111)  datalist-element ::=
           ( datalist DO do-specification ) | expression
```

(112) record-io-statement ::=
{ READ | WRITE | REWRITE | LOCATE | DELETE | UNLOCK }
[identifier] record-optionslist ;

(113) record-optionslist ::=
{ FILE (identifier) | EVENT (reference) | FROM (reference) |
IGNORE (expression) | INTO (reference) | KEY (expression) |
KEYTO (reference) | KEYFROM (expression) | SET (reference) |
NOLOCK }...

(114) display-statement ::=
DISPLAY (expression)
[REPLY (reference) [EVENT (reference)] |
EVENT (reference) REPLY (reference)] ;

3.1.3 Expressions:

(115) expression ::=
expression-six | expression || expression-six

(116) expression-six ::=
expression-five | expression-six & expression-five

(117) expression-five ::=
expression-four | expression-five comparison-operator expression-four

(118) comparison-operator ::=
> | >= | = | < | <= | ¬> | ¬= | ¬<

(119) expression-four ::=
expression-three | expression-four || expression-three

(120) expression-three ::=
expression-two | expression-three { + | - } expression-two

(121) expression-two ::=
expression-one | expression-two { * | / } expression-one

(122) expression-one ::=
primitive-expression | { + | - | ¬ } expression-one |
primitive-expression ** expression-one

(123) primitive-expression ::=
(expression) | reference | constant | isub

(124) reference ::=
[reference ->] basic-reference

(125) basic-reference ::=
{ . • unqualified-reference••• }

(126) unqualified-reference ::=
identifier [({ , • { expression | * }••• })]

(127) unsubscripted-reference ::=
{ . • identifier••• }

(128) constant ::=
real-constant | imaginary-constant | sterling-constant |
simple-string-constant | replicated-string-constant

(129) replicated-string-constant ::=
(integer) simple-string-constant

3.2 Lower level production rules:

3.2.1 Identifiers and constants:

(130) identifier ::=
letter [alphameric-character•••]

(131) 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 | $ | @ | #

(132) alphameric-character ::=
letter | digit | _

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

(134) isub ::=
integer SUB

(135) integer ::=
digit•••

(136) real-constant ::=
{ fixed-constant | float-constant } [B]

(137) fixed-constant ::=
integer [.] | [integer] . integer

(138) float-constant ::=
fixed-constant E [+ | -] integer

(139) imaginary-constant ::=
real-constant I

(140) simple-string-constant ::=
bit-string | character-string

(141) bit-string ::=
' [bit•••] ' B

(142) bit ::=
0 | 1

(143) character-string ::=
' [string-character•••] '

(144) string-character ::=
alphameric-character | blank | '' |
= | + | - | * | / | (|) | , | . | ; |
: | & | ¢ | ¬ | > | < | ? | % |
extralingual-character

(145) sterling-constant ::=
integer . integer . fixed-constant £

3.2.2 Pictures:

(146) picture-specification ::=
' picture-string [F ([+ | -] integer)] '

(147) picture-string ::=
[(integer)] picture-character [picture-string]

(148) picture-character ::=
A | B | C | D | E | G | H | I | K | M | P | R | S |
T | V | X | Y | Z | $ | 1 | 2 | 3 | 6 | 7 | 8 | 9 |
+ | - | * | / | , | .

3.2.3 Blanks and comments:

(149) space ::=
{ blank | comment }•••

(150) comment ::=

/ * [{ [*•••] comment-symbol | / }•••] *••• /

(151) comment-symbol ::=

alphameric-character | blank | '
= | + | - | (|) | , | . | ; |
: | & | ¢ | ¬ | > | < | ? | % |
extralingual-character

3.3 List of PL/I words:

.
<
<=
(
+
|
||
&
*
**
)
;
¬
¬<
¬>
¬=
-
->
/
,
>
>=
:
=
A
ABNORMAL
ALIGNED
ALLOCATE
AREA
AUTOMATIC
B
BACKWARDS
BASED
BEGIN
BINARY
BIT
BUFFERED
BUILTIN
BY
C
CALL
CELL
CHARACTER
CHECK
CLOSE
COLUMN
COMPLEX
CONDITION
CONTROLLED
CONVERSION
COPY
DATA
DECIMAL
DECLARE
DEFINED
DELAY
DELETE
DIRECT
DISPLAY
DO
E
EDIT
ELSE
END
ENDFILE
ENDPAGE
ENTRY
ENVIRONMENT
ERROR
EVENT
EXCLUSIVE
EXIT
EXTERNAL
F
FILE
FINISH
FIXED
FIXEDOVERFLOW
FLOAT
FORMAT
FREE
FROM
GENERIC
GET
GO
GOTO
IDENT
identifier
IF
IGNORE
imaginary-constant
IN
INITIAL
INPUT
integer
INTERNAL
INTO
IRREDUCIBLE
isub
KEY
KEYED
KEYFROM
KEYTO
LABEL
LIKE
LINE
LINESIZE
LIST
LOCATE
NAME
NOCHECK
NOCONVERSION
NOFIXEDOVERFLOW
NOLOCK
NOOVERFLOW
NORMAL
NOSIZE
NOSTRINGRANGE
NOSUBSCRIPTRANGE
NOUNDERFLOW
NOZERODIVIDE
OFFSET
ON
OPEN
OPTIONS
OUTPUT
OVERFLOW
P
PACKED
PAGE
PAGESIZE
PICTURE
picture-specification
POINTER
POSITION
PRINT
PRIORITY
PROCEDURE
PUT
R
READ
REAL
real-constant
RECORD
RECURSIVE
REDUCIBLE
REPLY
RETURN
RETURNS
REVERT
REWRITE
SECONDARY
SEQUENTIAL
SET
SETS
SIGNAL
simple-string-constant
SIZE
SKIP
SNAP
STATIC
sterling-constant
STOP
STREAM
STRING
STRINGRANGE
SUBSCRIPTRANGE
SYSTEM
TASK
THEN
TITLE
TO
TRANSMIT
UNBUFFERED
UNDEFINEDFILE
UNDERFLOW
UNLOCK
UPDATE
USES
VARYING
WAIT
WHILE
WRITE
X
ZERODIVIDE

3.4 Cross Reference Index: