ZEM
40

Jones

(FIL

TECHNICAL REPORT

TR 25.139
20 December 1974

A FORMAL DEFINITION OF A PL/I SUBSET

PART I

H. BEKIĆ
D. BJØRNER
W. HENHAPL
C. B. JONES
P. LUCAS

IBM LABORATORY VIENNA

IBM LABORATORY VIENNA, Austria

A FORMAL DEFINITION OF A PL/I SUBSET

PART I

by

H. Bekić
D. Bjørner
W. Henhapl
C. B. Jones
P. Lucas

ABSTRACT

This report provides a formal definition of large portions of the ECMA/ANSI proposed Standard PL/I language. The metalanguage used is described in the style of the "Mathematical Semantics". That is, the definition of PL/I is given by generating a function from a source program. A commentary is also provided to cover the less clear parts of the chosen model. For the convenience of the reader who wishes to have the commentary side by side with the formulae, the report is divided into two parts: Part I contains the description of the notation, the commentary and a cross-reference; Part II contains all the formulae.

NOTE

This document is not an official PL/I language specification. The language defined is based on the working documents (BASIS/1-9 to BASIS/1-11 [1]) of the joint ECMA/ANSI working group. It has not, however, been offered to them for review and has in no way been approved. Furthermore the subset chosen is not an indication of any IBM product plan.

TR 25.139

20 December 1974

A FORMAL DEFINITION OF A PL/I SUBSET

CONTENTS

PART I

PART II

Introduction

The aim of this report is to illustrate ideas about language definition on a "real" programming language. The language chosen is a subset of PL/I as defined in [1]. The main language features excluded are

CONTROLLED	storage
AREA	data
BY NAME	assignment
DEFINED	variables (other than overlay)
ALIGNED	attribute
REPEAT	option on DO
some Builtin	functions
PICTURE	attribute
ENTRY	statement

The (limited) parts covered of Input/Output have been written up separately and will be made available later. Certain detailed restrictions are given below in lines marked "BASIS-11".

The current definition differs in a number of respects from the earlier ones (e.g. [2]) written in the Vienna Laboratory. The need for change was largely observed in the attempts to base implementation proofs on "VDL" definitions (see [3]).

The removal of some of the shortcomings which had been noticed was attempted in [4]. The period since 1969 has also seen the development of "Mathematical Semantics" as proposed by D.Scott and C.Strachey ([5]). The definition given below follows this style by defining PL/I -programs via a mapping to the functions they denote. Although not fully described in the same style, the extension of these concepts to parallel computation has been the particular interest of one of the authors (see [6]). This report should be seen as summarising "work in progress" in the area of applying formal definition to compiler development.

The report is divided into two major parts: Chapter N of Part I describes the meta-language used in the definition; Chapter C of Part I contains a commentary on the more difficult parts of the model; the model is contained in Part II. A cross-reference of all the formulae is included as Chapter X of Part I.

Acknowledgements

The authors are grateful to the following for their contributions

H.Izbicki	collected from the BASIS document all of the "static checks" which are defined in D1.2;
V.Kudielka	produced an early draft of F5 and co-ordinated the commentaries section;
F.Schwarzenberger and M.Stadler	controlled the updates to the documents;
F.Mayrhofer, E.Moser and W.Plöchl	reviewed F3;
W.Pachl	provided frequent and very thorough reviews of the consistency of the formulae, he also wrote the cross-reference program;
K.Walk	reviewed D2.2;

F.Weissenböck co-operated in the production of the commentary for F5.

Last, but by no means least, the accurate data entry of the formulae from our somewhat varied handwritings was performed by Mrs. H.Neiss.

References

[1] ECMA.TC10/ANSI.X3J1
PL/I BASIS/1-11
European Computer Manufacturers Association
Feb.1974, 346 p.

[2] K.Walk,K.Alber,M.Fleck,H.Goldmann,P.Lauer,E.Moser,P.Oliva,
H.Stigleitner,G.Zeisel
Abstract Syntax and Interpretation of PL/I (ULD Version III)
Techn. Report TR 25.098, IBM Lab. Vienna,
Apr.1969.

[3] P.Lucas
On Program Correctness and the Stepwise Development of Implementations
Proceedings of the Congress on Theoretical Informatics, Pisa,
March 1973, pp.219-251

[4] C.D.Allen,D.N.Chapman,C.B.Jones
A Formal Definition of ALGOL 60
Techn. Report TR 12.105, IBM UK Labs Ltd.,
Aug.1972, 197 p.

[5] D.Scott,C.Strachey
Toward a Mathematical Semantics for Computer Languages
Techn. Monograph PRG-6, Oxford Univ. Computing Lab.
Aug.1971, 42 p.

[6] H.Bekić
Semantics of Parallel Programs
Techn. Report, IBM Lab. Vienna (forthcoming)

[7] P.J.Landin
The Mechanical Evaluation of Expressions
The Computer Journal, Vol.6(1964) No.4; pp. 308-320

[8] H.Bekić,K.Walk
Formalization of Storage Properties
Symposium on Semantics of Algorithmic Languages
Springer Lecture Notes in Mathematics, No. 188 (1970),p.28-61

THIS PAGE HAS BEEN INTENTIONALLY LEFT BLANK.

Notation

0. Introduction

The purpose of this Part is to document the intended meaning of the metalanguage used in Part II to define PL/I: the list of "non-objectives" is rather longer!

Firstly, it should be made clear that the description given below is not intended to be tutorial. It has been written for an audience which is assumed to have been already exposed to Formal Definition ideas. In particular no attempt is made to introduce those parts of the notation which are in common use. (One of the authors hopes to produce a more tutorial guide in the future).

Secondly, it can not be claimed that the metalanguage is the final word of the authors: even in the PL/I definition the construct used to express arbitrary ordering is not defined in a completely satisfactory manner. Moreover, although application to new problems has been considered, it is likely that other constructs would be proposed for a more general specification language.

A related, but perhaps less credible, restriction to our aims is that there is no wish to fix a <u>notation</u>. The approach to the definition and its use in justifying implementations has lead us to certain <u>concepts</u>. It has, of course, been necessary to agree a notation to employ these concepts.

That brings us to the subject of how the definition is written. The definitions written in "VDL" (Vienna Definition Language, cf. [2]) notation were abstract interpreters. The interpreting machine was made rather powerful because of the inclusion of a Control Component which could be explicitly manipulated. Subsequent work aimed at proving implementations correct (see [3]) showed that not only the control, but a number of other concepts were inconvenient: in nearly all cases the need was to make the definitions even more abstract by giving only properties required by the language. Ideas already existed for removing the need for explicit changes to the control as a model for GOTO (cf. [4]). Furthermore, the whole field of Mathematical Semantics style definitions of languages had been developed (cf. [5]).

PL/I is defined here by showing how to map any (abstract) program to a "transformation", that is, a function from states to states.

Classes of objects (including programs) can be described by Abstract Syntax Descriptions: such descriptions are discussed in Section 2. Section 1 describes the other classes of objects used, for instance, to describe states.

The functions which define the generation of transformations, and the transformations themselves are defined by means of a notation which is defined in terms of the lambda calculus in sections 3 and 4. The arbitrary order parts of the meta-language are discussed in 4.4.

The created transformations are defined by recursive equations with the intention that their value is the minimal fixed point. A constructive way of obtaining this is discussed in section 5.

The Appendix defines the concrete syntax of the metalanguage.

1. Objects

In order to achieve a language definition which shows only properties required by the language, the objects on which the definition is based should be as abstract as possible. For example, a set would be preferred to a list where no essential use was made of the ordering. The objects to be considered in the language definition are states and other arguments/results of functions. In order to provide appropriate abstractions for all of these, three different ways of forming composite objects are given in section 1.2; elementary objects are discussed in section 1.1.

The classes of <u>elementary</u> and <u>composite objects</u> are disjoint and together form the class of <u>objects</u>.

The only operators defined on all objects are the two infix relations

$o_1 = o_2$	equality
$o_1 \neq o_2$	inequality

1.1 Elementary Objects

In defining a language, certain classes of objects are required whose elements can be considered to be elementary in the sense that any structure they might have has no effect. Examples from the PL/I definition include the integers and the set of identifiers. The set of truth values is defined:

$B = \{\underline{true}, \underline{false}\}$

Other, individual, elementary objects are written with underlining:

FIX nil

1.2 Composite Objects

Composite objects are constructed from other objects (i.e. elementary or composite). In contrast to elementary objects, the structure of composite objects is considered: for each of the classes below, both the method of defining instances and the operations thereon are given. A further class of composite objects will be introduced in section 2. (Note that the following constructs are expressed in terms of values: section 4.5 discusses the use of transformations).

1.2.1 Sets

This section characterizes the class SET. Sets can be defined by enumerating their elements:

$\{x_1, x_2, \ldots, x_n\}$

A special case of this is the empty set:

$\{\}$

Sets can also be defined implicitly by a predicate:

$\{x \mid p(x)\}$

or more generally:

$\{f(x) \mid p(x)\}$

(using context to determine which variable(S) are bound).

The domain from which elements x are chosen, can be constrained by:

$\{x \in X \mid p(x)\}$

The notation for closed integer interval is (unusually):

$$\{m:n\} = \{i \mid m \le i \le n\}$$

The operators:

$x \in S$	test for membership
$S \subset T$	proper subset
$S \subseteq T$	subset (including equality)
$S \cup T$	union
$S \cap T$	intersection
$S \setminus T$	difference
$\underline{B}S$	Boolean or power set

are used with their conventional meaning.

1.2.2 Lists

This section characterizes the class LIST. Lists can be defined by an enumeration of their elements, where the written order gives the order of the defined list:

$$\langle x_1, x_2, \ldots, x_n \rangle$$

A special case of this is the empty list:

$$\langle\rangle$$

Lists can be defined implicitly, in which case the order of the defined list is:

$$\langle f(i) \mid i \in \{m:n\} \rangle = \langle f(m), f(m+1), \ldots, f(n) \rangle$$

This is also written:

$$\langle f(i) \mid m \le i \le n \rangle$$

The special case where f does not depend on i, simply provides a list of $n - m + 1$ identical elements:

$$\langle x \mid i \in \{m:n\} \rangle$$

All instances of lists will be of finite length.

The usual operators:

$\underline{l}\,l$	length
$\underline{h}\,l$	head (l non-empty)
$\underline{t}\,l$	tail (l non-empty)
$l[i]$	i-th element ($1 \le i \le \underline{l}\,l$)
$l_1 \frown l_2$	concatenation
$\underline{conc}\ L$	$L[1] \frown \ldots \frown L[\underline{l}\,L]$ (L a list of lists)

are used to define composition and decomposition of lists.

Lists of known length (tuples) can also be decomposed via $\underline{let}$, see section 2.

In order to provide a convenient notation, a list can be viewed as a map with domain $\{1:n\}$: the operators $\underline{D}$ and $\underline{R}$, defined below for maps in general, can be used:

$$\underline{D}l = \{1:\underline{l}l\}, \qquad \underline{R}l = \{l[i] \mid 1 \le i \le \underline{l}l\}$$

1.2.3 Maps

The subject of functions is discussed in section 3; the distinction which prompts the convention of using the term "map" is whether the graph of a function is a finite set of pairs. A more pragmatic distinction is that in the case of a map the graph is assumed to be computed at definition time (cf. section 5.3).

One way of defining a map is by enumeration:

$$[d_1 \rightarrow r_1, d_2 \rightarrow r_2, \ldots, d_n \rightarrow r_n]$$

(the d mutually different), of which a special case is the empty map:

$$[\;]$$

Another way is by implicit definition:

$$[d \rightarrow f(d) \mid p(d)]$$

or more generally:

$$[g(x) \rightarrow h(x) \mid p(x)]$$

(error if the resulting set of pairs does not have mutually different first members).

The domain and range of a map $m = [d_1 \to r_1, \ldots, d_n \to r_n]$ are:

$$\underline{D}m = \{d_1, \ldots, d_n\}, \qquad \underline{R}m = \{m(d) \mid d \in \underline{D}m\}$$

The value of a map for some argument in its domain is obtained by application:

$$m(d) = r$$

Further operators used on maps are:

$m_1 + m_2 = [d \to (\underline{if}\ d \in \underline{D}m_2\ \underline{then}\ m_2(d)\ \underline{else}\ m_1(d)) \mid d \in \underline{D}m_1 \cup \underline{D}m_2]$ (a left-associative overwriting)

$m_1 \cup m_2$ = same, but assuming $\underline{D}m_1 \cap \underline{D}m_2 = \{\}$ (union)

$m \setminus S = [d \to m(d) \mid d \in \underline{D}m \setminus S]$ (removal of a set of pairs)

2. Abstract Syntax Descriptions.

The purpose of an abstract syntax description is to define classes of (composite) objects, together with constructors, selectors, and predicates for composing, decomposing, and testing for the objects. Methods for composing objects are those described in the preceding sections, and also a method for forming trees. The syntax description may be supplemented by "constraints" narrowing the classes defined thereby.

2.1 Rules with =.

An abstract syntax description consists of a set of rules, one for each "non-terminal" name N. Rules are of two kinds. The first kind has the form:

$$N = A$$

where A is an *expression* composed from (names for) elementary objects, non-terminal names, and the operators described in the following. By interpreting elementary objects as unit sets:

$\underline{ABC}$ ~ $\{\underline{ABC}\}$

all the rules can be interpreted as equations defining sets (and N as a name for the set).

(Here and in the following sections, "~" is used to explain a new notation in terms of more basic notation.)

Union is denoted by | :

$A \mid B$ ~ $A \cup B$

Tupling is denoted by juxtaposition:

$A_1 \ldots A_n$ ~ $\{\langle a_1,\ldots,a_n\rangle \mid a_1 \in A_1 \wedge \ldots \wedge a_n \in A_n\}$

(Note this is not associative: ABC are triples, (AB)C are pairs whose first elements are pairs. In certain contexts formation of trees rather than tuples is implied, see 2.3).

Lists or *sets* over a given class are denoted by:

A^* ~ lists with elements in A
A^+ ~ non-empty lists with elements in A
A-set ~ $\underline{B}A$

Maps of given type are:

$A \rightarrow B$ ~ $\{m \in MAP \mid \underline{D}m \subseteq A \wedge \underline{R}m \subseteq B\}$

(which is a sub-class of the *function* space A->B, see Section 3).

Optional components are indicated by [ ]:

[A] ~ A | $\underline{nil}$

Type Clauses. All the above conventions for forming set-expressions are also used in type clauses, except that A->B usually stands for the full function space.

2.2 Trees, Constructors.

The second kind of rule has the form

$$N :: A_1 \ldots A_n$$

where the A are expressions as described above. Such a rule defines the class of trees constructed from given components by a given constructor:

$$N :: A_1 \ldots A_n \qquad \sim \qquad N = \{mk\text{-}N(a_1,\ldots a_n) \mid a_1 \in A_1 \wedge \ldots \wedge a_n \in A_n\}$$

The assumption on the constructors mk-N is that different constructors yield different objects and that components can be retrieved uniquely:

$$mk\text{-}N(al)=mk\text{-}N'(al') \supset N=N' \wedge al=al'$$

The class of all trees is denoted by TREE. Because of unique decomposition, trees can be treated very much like tuples. The notation:

$$\underline{let}\ mk\text{-}N(a_1,\ldots,a_n) = o$$

or even (omitting the constructor where it is obvious):

$$\underline{let}\ \langle a_1,\ldots,a_n\rangle = o$$

can be used to introduce names for the components of a tree of type N, similarly in parameter positions. Components which will not be referenced need not be named:

$$\underline{let}\ \langle a_1,\ ,a_3,\ldots,a_n\rangle = o$$

2.3 Selectors, Predicates.

A rule of the second kind may prefix the components by (simple) selectors, i.e. names starting with "s-" :

$$N :: s\text{-}a_1{:}A_1\ s\text{-}a_2{:}A_2 \ldots s\text{-}a_n{:}A_n$$

This provides another way of selecting components from an object o of type N:

$s\text{-}a_1(o)$.

Where no explicit selectors are given, <u>implied selectors</u> $s\text{-}A_1, \ldots, s\text{-}A_n$ formed by prefixing "s-" to the name of the component class are assumed. (In any other case than, possibly postfixed, nonterminal names, such selectors are not used for decomposing objects, but their existence is required for formation of composite selectors).

<u>Trees in nested positions</u>. If $(A_1 \ldots A_n)$ or $[A_1 \ldots A_n]$ appear in a position nested within the right-hand side of a rule, the conventions about de-tupling imply that it does not matter whether tree- or tuple-formation is assumed. Trees <u>are</u> assumed, however, whenever explicit selectors are used.

The functional style of selector application leads to a notion of <u>composite selector</u>:

$$s\text{-}x_1 \circ s\text{-}x_2 \circ \ldots s\text{-}x_n(o) \quad \sim \quad s\text{-}x_1(s\text{-}x_2(\ldots s\text{-}x_n(o)\ldots))$$

($n \geq 0$; the case $n=0$ gives the identity selector <u>I</u>).

The set of composite selectors applicable to a given object (and yielding elementary or composite objects) is

$$\begin{aligned} &\text{comp-sels}(o) = \\ &\quad o \text{ composite} \rightarrow \{I\} \cup \{sel \circ s \mid s \in \text{imm-sels}(o) \wedge sel \in \text{comp-sels}(s(o))\} \\ &\quad T \qquad\qquad\quad \rightarrow \{I\} \end{aligned}$$

where imm-sels(o) is the set of simple selectors applicable to the composite object o: the set of simple (explicit or implied) selectors to the immediate components of a tree o, resp. the set of (implied) simple selectors uniquely selecting the elements of a set, the elements of a list, and the values assumed by a map (e.g. = o itself for a set o, = <u>D</u>o for a list or map o). Sometimes one wants to consider only those selectors which give an object in a class θ:

$$\text{comp-}\theta\text{-sels}(o) = \{sel \in \text{comp-sels}(o) \mid \text{is-}\theta(sel(o))\}$$

The set of contained objects of class θ themselves is given by

$$\text{comp-}\theta\text{s}(o) = \{sel(o) \mid sel \in \text{comp-}\theta\text{-sels}(o)\}$$

The following predicate tests whether object o_1 is contained in object o:

$$\text{is-contained}\ (o_1, o) = (\exists sel \in \text{comp-sels}(o))(o_1 = sel(o))$$

Names for <u>predicates</u> testing for membership in an object class are derived by prefixing "is-" to the name of the class:

is-N(o) ~ $o \in N$
is-ABC(o) ~ $o = \underline{ABC}$

3. Functions and Expressions

The operators for sets, lists and maps have been discussed in section 1; logical and arithmetic operators are covered in 3.3 and 3.4 respectively, ways of forming expressions which are applicable for any type are covered in 3.2. Firstly, however, 3.1 discusses functions.

3.1 Functions

It will be necessary in the PL/I definition to consider functions which accept or return functions, including themselves. The following definitions are based, therefore, not on the conventional set-theoretic notion of function, but on the (interpreted) lambda-calculus (cf. [5]). Function, then, means partial, continuous (with respect to the ordering "is less defined than") function, and self-referential equations for functions can be solved in terms of the minimal fixed point (but see section 4.4 for a necessary elaboration).

(The maps introduced in section 1.2.3 are a special case of functions. They are simpler in one sense, but the kind of circularity due to self-applicable or self-returning functions arises also there, see section 5.3.)

The basic operations, then, are abstraction and application. As in [7], liberal addition of syntactic sugar makes such functions more palatable. It is the purpose of this section, 3.2, and 4 to explain these sugared forms in terms of the basic notions. In some cases this explanation requires multi-stage expansion. This is done not only for economy of writing, it is also fairly natural since the constructs may already be familiar.

The set of functions from D to R is denoted by D -> R.

When defining a function to which a name is given, the parameter is shown with the function name:

f(d) = e ~ $f = \lambda d.e$

A function of several variables is viewed as function over tuples:

$f(d_1,\ldots,d_n) = e \qquad \sim \qquad f = \lambda\langle d_1,\ldots,d_n\rangle.e$

Also, constructors may be used in the parameter list, see section 2.

Given two functions:

$f : D_1 \rightarrow D_2$
$g : D_2 \rightarrow D_3$

their composition is written:

$g^\circ f : D_1 \rightarrow D_3 \qquad \sim \qquad \lambda d.g(f(d))$

3.2 Expressions (general)

For all forms of expressions the concrete syntax shows that it is possible to give conditional expressions and local definitions. The simplest of the three forms of conditional expression is:

$\underline{\text{if}}\ b\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2$

(which produces a defined value even if the non-selected alternative is undefined).

The "McCarthy conditional" is defined in terms of the above in order to fix the ordering:

$$\begin{array}{lcl}
(b_1 \rightarrow e_1, & \sim & \underline{\text{if}}\ b_1\ \underline{\text{then}}\ e_1 \\
\ b_2 \rightarrow e_2, & & \quad \underline{\text{else}}\ \underline{\text{if}}\ b_2\ \underline{\text{then}}\ e_2 \\
\ \cdot & & \qquad\qquad \cdot \\
\ \cdot & & \qquad\qquad \cdot \\
\ \cdot & & \qquad\qquad \cdot \\
\ T \rightarrow e_n) & & \qquad\qquad \underline{\text{else}}\ e_n
\end{array}$$

Only if all cases are covered should the "T" clause be omitted.

The <u>cases</u> construct provides a very suitable way of structuring some conditionals:

```
(cases e:                        ~        ((∃al) (e=mk-a(al)) ->
 mk-a(al) -> e1,                              (let mk-a(al)=e; e1),
 .                                          .
 .                                          .
 .                                          .
 T         -> en)                           T -> en)
```

The let is used to locally define a value:

```
let r = e;                       ~        (λr.B)(e)
B
```

If e depends on r, the Y (minimal fixed point) operator is implied:

```
let r = f(r)                     ~        let r = Yλr.f(r)
```

Simultaneous recursion is defined with the use of ",":

```
let a = f(a,b),
    b = g(a,b);
```

The construct:

```
let r be s.t. p(r)
```

arbitrarily chooses an r satisfying p (error if there is none).

3.3 Logical Expressions

The logical operators are used with their "conditional expression" meanings (cf.[2]).

¬	not	prefix
∧	and	infix
∨	or	
⊃	implies	
≡	equivalence	

Quantifiers are normally used with bounded domains:

$(\exists x \in S)(p(x))$	there exists
$(\forall x \in S)(p(x))$	for all

where no bound is given, it is implied by the choice of name for the bound variable. (The constraint to a set over which p is defined, avoids the problems of "three-valued" interpretations).

Also used are:

$(\exists! x \in S)(p(x))$	there exists exactly one
$(\iota x \in S)(p(x))$	the unique object such that (only defined where $(\exists! x \in S)(p(x))$

The usual relational operators are also used.

3.4 Arithmetic Expressions

Apart from the usual prefix and infix operators:

$$\underline{prod}\langle v_1, v_2, \ldots, v_n\rangle = v_1 * v_2 * \ldots * v_n$$

$$\underline{sum}\langle v_1, v_2, \ldots, v_n\rangle = v_1 + v_2 + \ldots + v_n$$

$$\underline{max}(v_1, v_2) = (v_1 \leq v_2 \rightarrow v_2, \quad T \rightarrow v_1)$$

are used.

4. Transformations.

In this Section we introduce ways of expressing *transformations*, i.e. functions of type $\Sigma \rightarrow \Sigma$ or (*value-returning transformations*) $\Sigma \rightarrow \Sigma\ R$, where Σ is the set of *states*. A state is a mapping from a set of *references* to other objects. References are elementary objects. We write REF for the set of all references, $\underline{ref}\ V$ for the set of references whose "contents", in any state σ, are restricted to values in V:

$$(\forall r \in \underline{D}\sigma)(r \in \underline{ref}V \supset \sigma(r) \in V)$$

As an abbreviation in type clauses, we use:

$$\Rightarrow R \;\sim\; \Sigma \to \Sigma\, R, \qquad D \Rightarrow R \;\sim\; D \to (\Sigma \to \Sigma\, R)$$

(similarly if R is omitted). Thus transformations become $\Rightarrow$, value-returning transformations become $\Rightarrow R$, and functions from D to transformations (like the int/eval functions, see also section 5) become $D \Rightarrow$ or $D \Rightarrow R$.

4.1 Declaration, Contents, Assignment.

Programming languages provide a "variable-free" notation for state-transformations, i.e. a way of writing state-transformations without explicit reference to the state σ, and this is very much what the "combinators" introduced in this and the following subsections achieve. We assume types:

$$s\colon \Rightarrow, \qquad e\colon \Rightarrow R \text{ (for various R)}$$

(s for "statement", e for "expression"), similarly for s_1, s_2, $s(i)$, e_1, e_2, $e(i)$.

Declaration extends the state:

$$\begin{array}{lll} (\underline{dcl}\ r := v; & & \lambda\sigma.(\underline{let}\ r\ \underline{be}\ \underline{s.t.}\ \neg(r \in \underline{D}\sigma); \\ \ f(r) \quad) \colon \Rightarrow & \sim & \quad \underline{let}\ \sigma' = f(r)(\sigma \cup [r \to v]); \\ & & \quad \sigma' \setminus \{r\}) \end{array}$$

(for $f(r)\colon \Rightarrow$, similarly $f(r)\colon \Rightarrow R$).

Contents takes the value of σ at a given reference $r \in \underline{D}\sigma$:

$$\underline{c}r\colon \Rightarrow V \qquad \sim \qquad \lambda\sigma.\langle \sigma, \sigma(r) \rangle$$

(this has been made of type $\Rightarrow V$ - returning the unchanged σ - rather than $\Sigma \to V$ in order to be usable by the other combinators).

Assignment changes the contents of a reference $r \in \underline{D}\sigma$:

$$(r := v)\colon \Rightarrow \qquad \sim \qquad \lambda\sigma.\ \sigma + [r \to v]$$

Derived references. Given a reference r to a mapping $m\colon I \to V$, we sometimes use $i \circ r$ (for $i \in I$) as a "derived reference" to $m(i)$:

$\underline{c}(i^{\circ}r) \sim (\underline{c}r)(i)$, $\quad (i^{\circ}r := v) \sim r := \underline{c}r + [i \to v]$

(see 4.5 for the use of $\underline{c}r$ in a position where its result m is intended).

4.2 Sequencing.

First we introduce a variant of <u>let</u> (distinguished by the use of ":" instead of "=") which permits side-effects:

For e: =>V, f:V=> , we have:

$$\begin{array}{lcl} (\underline{let}\ v\!:\ e; & \sim & \lambda\sigma.(\underline{let}\langle\sigma',v\rangle = e(\sigma); \\ \ f(v)\quad) : \Rightarrow & & \qquad f(v)(\sigma')) \end{array}$$

(similarly for f:V=>R).

The <u>return</u> statement raises a value v∈V to a transformation:

$$(\underline{return}\ v) : \Rightarrow V \quad \sim \quad \lambda\sigma.\langle\sigma,v\rangle$$

The following all are transformations of type =>.

<u>I</u> is identity on states:

$$\underline{I} \quad \sim \quad \lambda\sigma.\sigma$$

<u>Semicolon</u> is sequential execution:

$$s_1;s_2 \quad \sim \quad \lambda\sigma.s_2(s_1(\sigma))$$

(similarly s;e).

The <u>conditional</u>

$$\underline{if}\ b\ \underline{then}\ s_1\ \underline{else}\ s_2$$

(for b:B), similarly <u>if</u> b <u>then</u> e_1 <u>else</u> e_2, and its variants are as described for expressions in general (see 3.2). We also allow:

$$\underline{if}\ b\ \underline{then}\ s \quad \sim \quad \underline{if}\ b\ \underline{then}\ s\ \underline{else}\ \underline{I}$$

and

```
if e then $s_1$ else $s_2$          ~     let b: e;
                                          if b then $s_1$ else $s_2$
```

(for e: =>B - a special case of the convention described in 4.5).

Iterative statements and expressions are

```
for i = m to n do s(i)              ~     s(m); ... ;s(n)

<e(i) | for i = m to n>             ~     let v(m): e(m);
                                          ...
                                          let v(n): e(n);
                                          return <v(i) | m≤i≤n>
```

(for m,n:intg), and:

```
while e do s                        ~     let w = (let b: e;
                                                   if b then s;w else I);
                                          w
```

(for e: =>B). See 4.4 for the for all statement.

4.3 Exit

The exit mechanism described in this section deals with the situation that execution of a (sub-)phrase has to be terminated "abnormally", i.e. abandoned; it also permits specification of the action that has to be performed on abnormal termination. This mechanism has been used to model the PL/I GO TO (cf. CF3) and RETURN statements, it can also be used to deal with error situations.

Formally, we can explain abnormal termination by slightly complicating our transformations, i.e. re-interpret =>:

```
D=>    ~  D ->(Σ -> Σ (nil | abn ABN))

D=>R   ~  D ->(Σ -> Σ (res R | abn ABN))
```

(similarly with D omitted); the flags res and abn are used to make normally and abnormally returned values disjoint. Transformations not involving sequencing combinators (like cr, r:=v, I, return v) can be re-interpreted immediately:

```
s                              ~      λσ.<s(σ),nil>

e                              ~      λσ.(let <σ',v> = e(σ);
                                          <σ,<res,v>>)
```

The exit statement returns a value abnormally:

```
exit(abn)                      ~      λσ.<σ,<abn,abn>>
```

The trap exit becomes:

```
(trap exit(abn) with f(abn);   ~      let r: s;
 s              ): =>                 cases r:
                                      (nil -> I, <abn,abn> -> f(abn))
```

(f(abn): =>), similarly with e instead of s:

```
(trap exit(abn) with f(abn);   ~      let r: e;
 e              ): =>R                cases r:
                                      (<res,v> -> return(v),<abn,abn> ->f(abn))
```

(f(abn): =>R).

A variant like trap exit(go,abn') with f(abn') causes a test on the arguments passed to exit, the f(abn') being executed only when the constants match; several trap exit's can be specified for one block as long as the argument ranges do not overlap.

Also semicolon (similarly: let:) have to be slightly more complicated:

```
s1;s2                          ~      let r: s1;
                                      cases r:
                                      (nil -> s2, <abn,abn> -> exit(abn))
```

The error handling is defined by:

```
error: =>                      ~      exit(ERROR)
```

where no trap exit for ERROR is provided.

See next section for exit from a parallel phrase.

4.4 Arbitrary Order.

The comma between two transformations denotes *quasi-parallel* execution of them: the "elementary" steps of the two transformations are merged in arbitrary order, preserving only the two orderings within the given transformations. Which steps are considered elementary is left open, a sensible choice would be to take the "terminal" operations of the metalanguage (like c, :=) as elementary. Thus:

(s_1,s_2): => ~ elementary steps merged in arbitrary order

For e_1: =>V_1, e_2: =>V_2 :

$\langle e_1,e_2\rangle$: =>V_1V_2 ~ same, with pair $\langle v_1,v_2\rangle$ of returned values as returned value

(similar for several e(i)).

The context where this is most used are parallel let's:

```
let $v_1$: $e_1$,                ~      let $\langle v_1,v_2\rangle$: $\langle e_1,e_2\rangle$;
    $v_2$: $e_2$;                       f($v_1$,$v_2$)
f($v_1$,$v_2$)
```

Sometimes the s(i) or e(i) are not enumerated explicitly:

for all $i \in I$ do s(i) ~ execute s(i) in parallel

For e(i): =>V (for each $i \in I$) :

par{e(i) | $i \in I$}: =>V-set ~ execute the e(i) in parallel; return the map

$[i \rightarrow v(i) \mid i \in I]$

where v(i) is returned by e(i)

(The operator par is only used implicitly, see next section.)

Arbitrary order and exit. If in (s_1,s_2) (similarly: $\langle e_1,e_2\rangle$) one of the two transformations, say s_1, terminates abnormally with value abn, then an (implied) exit(abn) is executed in s_2, which will cause execution of the relevant trap exit's in s_2. If this eventually terminates s_2 abnormally with the same value abn, the whole transformation (s_1,s_2) terminates abnormally with abn. (If the implied exit leads to normal termination of s_2, or to abnormal termination with a different value abn', then (s_1,s_2) terminates with error; no such, error producing, use of exit in parallel transformations has been made in the PL/1-Definition.)

Non-determinism and recursion. The arbitrary choice operator let v be s.t. p(v) (see 3.2) introduces an element of non-determinism and thus, strictly speaking, forces transformations to be functions from states to sets of states, rather than from states to states. The quasi-parallel merging operator, which also introduces non-determinism, additionally complicates transformations because now we have to consider their component steps, rather than the functional product of those steps. A particular problem arises with recursive definitions and non-determinism: The ordering relation ("v is less defined than v'") on which the familiar way of solving recursive equations is based does not immediately carry over from elements to sets. One way to solve this problem is to evaluate the expressions of the metalanguage under an additional hidden parameter serving as a "choice tape" (e.g. an infinite sequence of truth values); evaluation under a given choice tape is deterministic, the set of all solutions is obtained by considering all choice tapes (cf.[6]).

4.5 Value-Returning Transformations in Value Positions.

Often it is convenient to write a value-returning transformation in a place where a value is required, with the understanding that the transformation is executed as a side-effect. Thus:

For f:V->R, e: =>V , we have:

```
f(e): =>R                  ~        let v: e;
                                    return(f(v))
```

This makes f(e) a transformation (of type =>R) whereas the context requires a value (of type R), and so we can apply the same rule to this context, say g(f(e)). Eventually we will come to a context which is intended to produce a transformation (e.g. g(v) might be return (v)); this is covered by the analoguous rule:

For g:V=>R, e: =>V , we have:

```
g(e): =>R                  ~        let v: e;
                                    g(v)
```

(For f or g with several arguments we get a parallel transformation as the right-hand side of the let.)

Where the e(i) are given implicitly, par is implied:

For e(i): =>V (for each i∈I):

```
{e(i) | i∈I}: =>V-set          ~        let m: par{e(i) | i∈I};
                                        return Rm

<e(i) | i∈I>: =>V*             ~        let m: par{e(i) | i∈I};
                                        return <m(i) | i∈I>
```

(where I is an interval {m:n} - this really only re-explains <e(m),...,e(n)>).

```
[i → e(i) | i∈I]: =>(I->V)     ~        let m: par{e(i) | i∈I};
                                        return m
```

5. Constructive Interpretation of the Metalanguage.

5.1 Macro-Expansion.

The int/eval functions of the PL/I Definition are correspondences from text-classes θ (and auxiliary parameters) to transformations:

$$\begin{aligned} \text{int-}\theta:&\quad \theta\ \mathrm{ENV}\ \ldots \rightarrow (\Sigma \rightarrow \Sigma)\\ \text{eval-}\theta:&\quad \theta\ \mathrm{ENV}\ \ldots \rightarrow (\Sigma \rightarrow \Sigma\ V)\end{aligned}$$

The aim of this section is to outline a method to constructively interpret the highly recursive definitions of these correspondences. The idea is to (1) *macro-expand*, for given t∈θ, env, ..., the call int-θ(t,env,...), i.e. replace it by its definition, similarly for nested calls of int/eval functions; this will eventually lead to a description of the corresponding transformation which no longer refers to any int/eval functions, whereupon (2) application of this transformation (description) to a given state can be left to a conventional *call-by-value* interpreter.

5.2 "Unfounded" Uses of int-θ.

Usually, int-θ(t,...) is defined by structural induction on the text t, i.e. in terms of int-θi(ti,..) where the ti are the (immediate) components of t, so that the expansion process will get to ever smaller components and eventually stop. There are, however, a few cases where int-θ(t,..) itself recurs in its definition.

Example 1 (while-statement) (cf F3; this and the following examples are somewhat simplified extracts from corresponding examples in the F Chapters):

```
int-wh-st(<e,st>,env) =
   let b: eval-expr(e,env);
   if b then (int-st(st,env); int-wh-st(<e,st>,env)) else I
```

We can formally avoid the recurring use of int-wh-st by a (recursive!) let:

```
int-wh-st(<e,st>,env) =
   let f = (let b: eval-expr(e,env);
            if b then (int-st(st,env); f) else I;
   f
```

Example 2 (compound-statement,omitting env, cf. F3):

```
int-cpd-st(t) = cue-int-cpd-st(t,lab1)

cue-int-cpd-st(t,lab) =
   trap exit(abn) with if ... then cue-int-cpd-st(t,abn) else exit(abn);
   int-st(t[lab]);
   if ... then cue-int-cpd-st(t,lab+1) else I
```

(lab1 = label of first statement, lab+1 = label of next statement), which becomes

```
int-cpd-st(t) =
   let f(lab) = (trap exit(abn) with if ... then f(abn) else exit(abn);
                 int-st(t[lab]);
                 if ... then f(lab+1) else I);
   f(lab1)
```

(Of course the let f style could be used directly. In Example 1, the re-use of int-wh-st has actually been avoided by using the while-construction of the metalanguage.)

5.3 Recursive "let:" - clauses.

Whereas the rewriting just discussed was only necessary to get a closed expression for the resulting transformation ("to stop the expansion") but did not make any difference for an interpreter, the case

```
let v: f(v)
```

of a transformation defined in terms of the value (to be) returned by its execution is more serious:

Example 3 (blocks, cf. F1):

```
int-bl(<dcls,procs,st>,env) =
   let lenv: [id → eval-dcl(dcls(id),env) | id ∊ Ddcls] ∪
             [id → eval-proc(procs(id),env+lenv) | id ∊ Dprocs];
   int-st(st,env+lenv)
```

(assuming dcls is a map from id to dcl, similarly for procs),

```
eval-dcl(dd,env) =
   let edd: ...dd...;
   alloc(edd)

eval-proc(<idl,st>,env) =
   λlocl.(let penv = [idl[i] → locl[i] | 1≤i≤lidl];
          int-st(st,env+penv))
```

Observe that call-by-value would immediately run into a loop with int-bl: to compute lenv, we would first have to compute lenv in order to pass it to the eval-proc calls. Expanding we get:

```
int-bl(<dcls,procs,st>,env) =
   let lenv: [id → (let edd: ...dcls(id)...; alloc(edd)) | id ∊ Ddcls] ∪
             [id → λlocl.(let penv = [s-idl(procs(id))[i] → locl[i] | ...];
                          int-st(s-st(procs(id)),env+lenv+penv))
                   | id ∊ Dprocs];
   int-st(st,env+lenv)
```

(note that side effects in eval-dcl, e.g. alloc(edd), are to be executed at let lenv - time), and we see that the use of lenv within the definition is now "shielded" by occuring within the scope of λlocl. This can be dealt with by a call-by-value interpreter (provided a λ-expression is not evaluated before it gets applied). (It is easy to see that this is the only kind of unshielded use in the PL/I-Definition - the only recursive definitions in PL/I are procedure declarations.)

5.4 Distinction between Static and Dynamic Properties.

We started this section by asking for a constructive interpretation of the very implicit definitions, but the process of expansion we have described is of interest also in other respects. It gives a (more) closed description of the transformation denoted by given t under given env,... . It exploits, and makes visible, the distinction between static and dynamic case distinctions, i.e. between decisions that are made to arrive at the transformation, and decisions that are part of the transformation. Obviously, this distinction is important for deriving a compiler from the language definition, but it is also relevant for showing which "steps" make up the resulting transformation (see 4.2). One could go further in the expansion (and sharpen the distinction), e.g. expand the let-clause for the local environment lenv into several let-clauses, one per local identifier, arriving at a description which does not use env at all. All the uses made of the for all and (implied) par operator (cf. 4.4) are such that they can be statically expanded into $(s_1,...,s_n)$ resp. $(e_1,...,e_n)$.

APPENDIX : Concrete Syntax

The concrete syntax of the meta-language is given in the notation of ref. [2]. The class "expr" is subdivided only for the definition of the operators - it is otherwise context-free.

Written definitions use a number of relaxations on this syntax which are not formally defined.

a) Brackets around blocks and cond-stmts as well as commas are omitted where identation or line breaks makes the result unambiguous.

b) A cases style cond-stmt may define more than one condition per expression by using "|".

c) Comments, enclosed in "/* */" may be used freely - in particular assertions will be written in comments.

d) Where an expression occurs after let etc. , "result is" can be used.

e) The order of precedence of operators (standard) is modified by use of blanks and line-breaks in order to avoid excessive bracketing.

```
opn-defn ::= int-id {(defs)}••• : st-block |
             eval-id {(defs)}••• : e-block

int-id ::= id             usually begins "int-"

eval-id ::= id            usually begins "eval-"

defs ::= [,•def•••]

def ::= l-id | <{,•def•••}> | constructor

l-id ::= id

constructor ::= "mk-"    as-class-nm(defs)

e-block ::= block         block is value returning

st-block ::= block        block is a transformation
```

```
block ::= ([exit-spec]
           [declaration•••]
           [let-cl•••]
           {;•stmt•••})

exit-spec ::= trap exit (defs) with stmt;

declaration ::= dcl l-id [:=expr];

let-cl ::= let {,•let-body•••};

let-body ::= def:expr |
             l-id(defs):block


stmt ::= st-block|cond-stmt|iter-stmt|ld-stmt|assign-stmt|
         int-stmt|return-stmt|I|exit-stmt|error

cond-stmt ::= if expr then stmt[else stmt] |
              ({,•{expr -> stmt}•••}[,T -> stmt] |
              (cases expr:
               {,•{expr -> stmt}•••}[,T -> stmt])

iter-stmt ::= for l-id = expr to expr do stmt |
              for all def ∈ expr do stmt |
              while expr do stmt

ld-stmt ::= (local-defs;stmt)

assign-stmt ::= l-id := expr

int-stmt ::= int-id {(args)}•••

args ::= [,•expr•••]

return-stmt ::= return (expr)
                       pure expr

exit-stmt ::= exit (args)
```

```
fn-defn ::= fn-id(defs) = expr
                          pure expr

expr ::= e-block|eval-stmt|(expr)|cond-expr|ld-expr|
         prefix-expr|infix-expr|quant-expr|fn-ref|
         var-ref|const

eval-stmt ::= eval-id{(args)}•••

cond-expr ::= if expr then expr else expr |
              ({,•{expr -> expr}•••}[,T -> expr]) |
              (cases expr:
               {,•{expr -> expr}•••}[,T -> expr])

ld-expr ::= (local-defs;expr)

local-defs ::= {;•local-def•••}

local-def ::= let {,•loc-let-body•••}

loc-let-body ::= def{= | be s.t.}expr |
                 l-id(defs) = expr

fn-ref ::= fn-id(args)

var-ref ::= [c] l-id {[args] | (args)}
```

"pure expr" does not contain (directly) e-block
                                        eval-stmt
                                        c

logical expressions

```
prefix-expr ::= ¬log-expr

infix-expr ::= log-expr {∧|∨|⊃|≡} log-expr |
               expr ∈ set-expr |
               set-expr {⊂|⊆} set-expr |
               arith-expr {<|≤|≥|>} arith-expr |
               expr {=|≠} expr

quant-expr ::= ({∀|∃|∃!} def [∈ set-expr]) (log-expr)

const ::= true | false | "is-" as-class-nm(expr)
```

arithmetic expressions

```
prefix-expr ::= llist-expr | - arith-expr | mod (arith-expr arith-expr) |
                {sum|prod} list-expr | max(arith-expr,arith-expr)

infix-expr ::= arith-expr {+|-|*|/|↑} arith-expr

const ::= ints
```

general expressions

```
prefix-expr ::= h list-expr | "s-" as-class-nm(expr)

quant-expr ::= (ιdef[∈ set-expr]) (log-expr)

const ::= constructor
```

set-expr

```
prefix-expr ::= union set-expr | Bset-expr |
                {D|R} map-expr

infix-expr ::= set-expr {∪|∩|\} set-expr

const ::= B | {[,•expr•••]} |
          {expr | log-expr} |
          {arith-expr:arith-expr}
```

list-expr

```
prefix-expr ::= tlist-expr | conc list-expr

infix-expr ::= list-expr^list-expr

const ::= <[,∘expr•••]> |
          <expr ⊥ l-id∈set-expr> |
          <expr ⊥ for l-id := arith-expr to arith-expr>
```

map-expr

```
infix-expr ::= map-expr {+|∪} map-expr |
               map-expr \ set-expr

const ::= [{,∘{expr -> expr}•••}] |
          [expr -> expr ⊥ log-expr]
```

THIS PAGE HAS BEEN INTENTIONALLY LEFT BLANK.

Commentary

C0. Overview

The purpose of the commentary part of the report is to provide a description of some of the less obvious aspects of the model given in the formal part. Given a knowledge of the meta-language the reader is assumed to be able to interpret the formulae as such, and no translation into words is attempted. (If at any place there should be some contradiction, it is the formulae rather than the text which define the model.)

One aid to reading provided by the commentary is the elucidation of the abbreviations used. The set given at the end of this section apply uniformly to function names and names of abstract syntax classes. The use of names for locally defined objects has been less consistent: these are defined at the point of uniform usage (e.g. section, sub-section or formula).

The structure of the commentary is the same as that of the formal part. The major division is between the objects which are manipulated by, and the defining functions themselves. The first of these ("Domains") separates abstract programs from the other objects. A set satisfying is-prog is first defined by abstract syntax rules. A large number of static properties can be described for valid programs. Assuming that these properties are fulfilled makes it possible to write the defining functions in a clearer way. (It is also a way of making the language properties clearer than if they were mixed with the dynamic tests). The class of programs to be used as the domain of int-prog, is, then, defined as a subset of is-prog whose members also satisfy the given context conditions. The "States" portion of the Domains section describes the other objects manipulated. Principal among these is "Storage" which models PL/I variables. The reasons for choosing an implicit definition for this are discussed below.

The "Functions" section is divided into six parts, in this way formulae relating to a particular language concept are grouped together. The section on input/output is only a place holder for the required functions. (The actual formulae will be the subject of a separate report).

The functions themselves are sometimes supported by pre and post conditions and assertions. When given the function is only defined over a restriction of the domain given in the type clause: those elements satisfying pre. That the function is only used over this restricted domain results from other constraints. Post conditions and assertions provide an insight into the formula by stating relations the authors were trying to preserve.

Abbreviations

aa	activation identifiers
abs	absolute
act	action
addr	address
ag	aggregate
aid	activation identifier
alloc	allocate
approx	approximate
arg	argument
arith	arithmetic
ass	assign, assigment
atm	atomic
augm	augment[ed]
auto	automatic
bi	builtin
bif	builtin function
bin	binary
bl[s]	block[s]
bool	boolean
boe	block on-establishment
bp	bound-pair
bpl	bound-pair list
bs	base
c	cond-pref-set,context
cat	concatenate
cbif	condition built-in function
ccn	comp-cond-nm
c-nm	non-io-cond-nm or io-cond
ceil	ceiling
char	character
cl	class
clng	closing
cmp	composite
cn	cond-nm
comp	computational, component (the latter used more often!)
compar	compare
cond	condition
conn	connected
const	constant
constr	construct
cont	content

conv	conversion, convert
cprefs	cond-pref-set
cpv	cond-pv
ctl	control
ctld	controlled
cur	current
dcl	declaration
dcls	declaration set
dd	data-description
dec	decimal
def	defined
der	derived
descr[s]	descriptor[s]
dft	default
digitl	digit list
dim	dimension
distr[ib]	distributive
div	divide
dsgn	designator
dtp	data-type
ebp	evaluated bound-pair
edd	evaluated data description
el	element, elementary
elem	element
enab	enabled
env	environment
eq	equal
eu	executable-unit
evd	evaluated
eval	evaluate
eval-l	eval-to-left-value
ex-unit	executable-unit
expr	expression
ext	external
fact	factor
fct	function
f	field
fix	fixed
flt	float
fofl	fixed overflow
fuid	unique file identifier
ge	greater or equal
gen	generate
grp	group

gt	greater
hbound	high bound
id[s]	identifier[s]
im	immediate
impl	implementation
indep	independent
indices	set of index lists
indl	list of indices
inf	infix, information
init	initial, initialize
init-wh-do	DO statement with init and while
int	internal
int	interpret(in fn names)
intg	integer
io	input-output
ioc	io-cond
iter	iterative, iteration
l	list
l-	location
lab	label
lb	lower bound
lbound	low bound
le	less than or equal
len	length
loc[s]	location[s]
locr	locator
loe[r]	local on-establishment [by reference]
lt	less than
l-to-r	left to right
max	maximum
maxl	maximum length
min	minimum
mod	modulo
mult	multiply
ne	not equal
nm[s]	name[s]
nmd	named
nod	number-of-digits
num	number,numeric
obs	objects
ofl	overflow
onsource	onsource location
op	operator
opng	opening

opt[s]	option[s]
ou	on-unit
parm[s]	parameter[s]
pdd	parameter data descriptor
pos	position
pos-cond-nm	positive-condition-name
prec	precision
pref[s]	prefix[es]
proc	procedure
prog	program
prom	promote
prop	proper
ps	proper statement
ptr	pointer
pv	pseudo variable
qual	qualifier
r	right
rec	recursive
recity	recursivity
ref	reference
rel	relevant
rep	representation
res	result
ret	return
ret-descr	returns-descriptor
rev	revert
sc	scalar
sdd	statically determined data description
scomp-	static computational-
sdtp	static data description
sect	section
sels	selector set
sentry	static entry data description
sig	signal
snap	SNAP or nil
snms	statement names
snon-comp-	static non-computational-
source	onsource-char-str-val
spec	specification
st[s]	statement[s]
step-do	DO statement with TO
stg	storage
str	string
strg	string range

struct	structure
strz	stringsize
subjs	subjects
subr	subroutine
subrg	subscript range
subscr	subscript
substr	substring
subt	subtract
t	text
targ	target
term	terminal
tp	type
truth	truth (truth value)
ub	upper bound
udf	undefined
ufl	underflow
uid	unique identifier
unal	un-aligned
v-	value
val	value
var	variable
varity	variability
vary	variability
vr	value reference
wh	while
zdiv	zerodivide
1-loc	level-one location

CD1.1 Abstract Programs

The explicit selectors are those used in chapters D2 through F5; selectors used in D1.2 are explicit when given, otherwise implicit.

The reader should note the choice of defining symbol (:: or =, cf. N2) used in the various rules. The construction has been rather cautious in that elements of unions (unless themselves unions) have usually been given constructors, even if one could have shown this not to be necessary to ensure disjointness.

In a number of places (e.g. dcl-set) it is now thought that rather than use a set, a mapping (in this case id -> dcl-tp) would provide a shorter definition.

ad 31 entry: Deviating from BASIS-11, we distinguish between: no requirements on parameters, and empty paramter list required. In concrete syntax: ENTRY vs. ENTRY().

CD1.2 Context Conditions and Functions

CD1.2.1 Static Data Descriptions

Static data descriptions are used to capture the declarative information available in the program text. Thus it is not, in general, possible to know more than the dimensionality of an array: the bounds may be computed only in relation to a particular storage. (Some benefit could be gained by combining the definitions of dd, sdd, pdd and edd).

CD1.2.2 Rule-by-Rule Conditions and Functions

Within the class of objects satisfying is-prog there are some which can be considered "statically wrong" programs (e.g. using variables which are not declared). It would be possible to build checks for such errors into the defining function. It was felt by the current authors that it was better to show such properties statically. The predicates of this section define a subclass of is-prog, as follows: for each phrase class θ defined in terms of $\theta_1, \theta_2, \ldots, \theta_n$ there is a rule which is either provided explicitly or by default is:

$$\text{is-wf-}\theta(o,env) = \text{is-wf-}\theta_1(\text{s-e1}(e),env) \wedge \ldots \wedge \text{is-wf-}\theta_n(\text{s-e}_n(e),env)$$

The env component contains the declaration or procedure for each known identifier. This, together with the function el-sdd (see below) provides the way of checking those context conditions governing types.

Another important class of context conditions is those which simply express a context-free subset of the abstract syntax (e.g. is-no-refers): these are expressed as predicates of c-comp-θ, although they could have been handled by duplicating rules. Those context conditions prohibiting duplication of names are defined using the predicate is-unique-ids. Certain consistency checks are made (e.g. locator qualifiers must be available by default if not explicit). There are also "geometrical" (e.g. is-refer-geom) and value (e.g. EXTERNAL dd's must evaluate to same edd) constraints.

The numbers used for the functions are those of the abstract syntax.

ad 1 is-wf-prog: to simplify notation, quantifiers over contained objects (here: p1, p2) have been omitted throughout this section.

ad 39 is-wf-bl: notice that passing the old environment minus local names (nenv'), prevents for example automatic declarations relying on block local quantities.

CD1.2.3 Auxiliary Functions

The function el-sdd yields the sdd of an expression. That is, it determines the descriptions of its atomic elements and applies rules for combining operand types with particular operators. In the uses of this function outside D1, the second argument (textual environment env, which can always be determined statically) has been omitted. For a discussion of the distribution mechanism see CF5.

CD2. States, Auxiliary Parameters.

Programs denote functions from states to states, and this Section defines the notion of (PL/I-)*state*. As explained in N4, a state is a map from references ("variables") to other objects (the "contents" of the variable). The five major state components are treated in the first three sub-sections: AA and PA (dealing with activation identifiers) in 2.1, storage in 2.2, external storage and file state (dealing with input/output) in 2.3.

The int/eval functions establishing the correspondence between texts and state-transformations need *auxiliary parameters*. These are defined in the remaining sub-sections, namely the environment in 2.4, the on-establishment (dealing with on-conditions) in 2.5, and the cbif-part (dealing with condition builtin functions) in 2.6.

Abbreviations for this Section:

Σ	set of states
σ	a state
S	[ref to] storage
ES	[ref to] external storage
FS	[ref to] file state
AA	[ref to] active aid's
PA	[ref to] previous aid's
aid	activation identifier

1 Σ:

See N4. The present definition is more specific in that it enumerates the object classes over which the contents of a state component can range. The first four alternatives are due to the major (global) state components (AID arising twice); OE (cf 2.5) and the "other" objects arise as the contents of local state components.

2 (major state components):

By systematic ambiguity, these five names are used both as *references* (e.g. when appearing on the left of :=, or as argument of *c*) and as names for the *sets* over which the references range (e.g. in syntax rules).

CD2.1 Activation Identifiers

An activation identifier (aid) serves to uniquely identify the activation of a block or procedure; it is needed to make the denotation of a label unique, and also for discovering uses of "dead" label- and entry values. PA records all aid's used so far (it is never decreased), AA the currently active ones.

CD2.2 Values, Locations, and Storage.

The storage model used here is a version of the general model described in [8], specialised to the needs of PL/I. The basic idea behind the model is quite simple: Storage is, essentially, a function f from locations to values:

$$f: L \rightarrow V$$

thus associating with each location l in L a value v = f(l), the "contents" of the location, with the following two properties:

1. f is range-respecting: each location has associated with it a certain range, i.e. subset of V, and can contain values from this subset only.

2. f is structure-preserving: a location may have components, and then the contents of the component location is the "corresponding" component of the contents of the whole location.

The present model is more explicit than the general model. For example, composite locations (and values) are defined explicitly as lists or maps. Also, the only instance of "flexible" locations (i.e. ones whose active components depend on the current contents) is provided by VARYING strings. (A price to pay for this explicitness is that "width zero" locations, e.g. string locations of length 0, now seem to be over-specified, at least in connection with pointers, see CD2.2.3). Still, many notions are characterised implicitly, by axioms. For example, there is no need to say what an elementary location "is"; also, PL/I pointers are so implementation-defined that they are best described by (incomplete) axioms.

u-edd	unit-edd (of an array-edd)
ebpl	evaluated bound pair list
tp	string-type
vy	variability
v	value
vl	value list
vals	values
l	location
m, mm'	map
-l	-list

CD2.2.1 Evaluated Data Descriptions

Evaluated data descriptions (edd's) arise from dd's by evaluating expressions for array bounds and string lengths (and dropping initial and REFER elements); they serve, among other things, to represent the range of a location, see values(edd) below.

Indices are used to select components from locations or values, see CD2.2.2 and CD2.2.3 below. (The functions given here are purely auxiliary).

19 width:
counts characters, bits, and non-string scalars.

20 is-all-str:
tests whether edd consists of NONVARYING characters or bits only (tp = CHAR or BIT).

CD2.2.2 Values

An array value is a map from a multi-dimensional rectangle of integers (the subscripts) to values of a given type. A structure value is the list of its field values. Note that, by use of ::, the empty character string value and the empty bit string value are different. An entry value is a function (cf. CF1) together with an identifier (needed for entry comparison) and an aid (needed for checking against "dead" entries).

28 mk-STR-VAL:
needed where not statically known whether to use mk-CHAR-STR-VAL or mk-BIT-STR-VAL.

39 udf-val,
40 is-defined-val:

Undefined values are needed to discover uses of uninitialised variables. One value ? is used for single elements of NONVARYING strings, and for the other scalars (note that the undefined VARYING string is *one* ? - nothing about the current length of the string is known!); composite undefined values are composed of ?'s; a value is *defined* if it contains no ?.

41 values:

Connects edd's with "ranges", i.e. value sets. Note that the preceding rules for VAL etc. did not list ? with STR-VAL or the alternatives of ELEM-VAL, so it has to be added to the ranges here.

43 v-augm-indices,
44 v-indices:

The indices returned by v-indices(v) are integer-lists selecting the elements of an array, integers selecting the (immediate) fields of a structure, and pairs <i,i> selecting the one-element substrings of a string; v-augm-indices also includes * for arrays (for forming cross-sections) and <i,j> for strings, it is only used in the pre of comp-val.

CD2.2.3 Locations

LOC is the set of *all* (potential) locations, not only the currently allocated ones. The definitions are analogous to those for VAL; *atomic* locations (elementary, i.e. non-string scalar locs, and single character and bit locs) are left unanalyzed - except that elementary locs have an edd extracted by the function l-edd . Note that CHAR-LOC and BIT-LOC are not subsets of LOC - they cannot be denoted in PL/I.

Due to distribution over all its arguments, the SUBSTR pseudo-variable (see F2) can generate "inhomogenous" array locations violating the constraint given for ARRAY-

LOC. The use of this constraint in the definition of l-edd could be avoided by associating with array locations a map from indices to edd's, rather than the present array-edd (which would arise as the special case of a constant map).

53 CHAR-STR-LOC,
54 BIT-STR-LOC:

The corresponding constructors mk-CHAR-STR-LOC and mk-BIT-STR-LOC must be assumed as non-unique in the case of (at least the VARYING) empty string, see "Independence" below.

The function comp-loc is completely analoguous to comp-val.

66 sub-loc:
m' is an array-loc, except that the elements may be arrays again; this is rectified by the function array-loc.

68 ordered-sc-locs,
69 sc-locs:
give list/set of scalar sub-locations.

70 ordered-atm-locs,
71 atm-locs:
similar for atomic locations, with an irregularity for VARYING strings explained presently.

73 1-LOC:

The set 1-LOC of <u>level-one</u> <u>locations</u> is used as a pool from which to allocate storage for PL/I level-one variables. The dissection into AUTO and BASED locations is used for a test on freeing, see F2.

74 is-indep:

Two locations are *independent* if they have no parts in common. (The reason for including length-zero VARYING string locations in atm-locs above is that they have no atoms yet need to be distinguishable: they have two possible contents, namely the empty string and ?. The latter does not hold in the NONVARYING case). Different level-one locations, and different components of the same given location, are postulated to be independent.

77 is-conn,
79 is-l-to-r-loc:

A location is *connected* if it is a contiguous part of a level-one location. *Left-to-right equivalence* is defined, contrary to BASIS/1-11, down to arbitrarily nested structure levels.

80 PROP-PTR-VAL,
81 addr,
82 constr-loc:

A connected location has an *address* which is a (non-null) *pointer*. Intuitively, the address may be regarded as location "minus" edd, hence the loc should be reconstructable from its addres and its edd (axiom 83). Independent locations have different addr (axiom 84, postulated only for locations not of width 0, in view of the difficulties with the latter; for locs with the same edd, this axiom follows from the previous one). Left-to right equivalent locations have the same addr (axiom 85), and an all-CHAR or all-BIT location has the same addr as its first atomic location (axiom 86; these last two axioms are compatible with the view that addr is the "starting point" of a location).

CD2.2.4 Storage

L_0	currently active level-one locations
f_0	storage, viewed as a map over L_0
L	all locations derivable from L_0
f	storage, viewed as a function over L

87 S:

For finite representation, storage is viewed as a map from (active) level-one locations only. The two properties of storage required in the introduction to CD2 above are ensured, then, first by the constraint to this formula, and second by the way in which the map is extended to L:

91 extend:

The "parts" of a location go down to characters and bits, but not inside VARYING strings; the "current parts" also take into account components of the latter within the current length. The extended set L consists of all locations whose parts are among the current parts of locations in L_0. The given map over L_0 uniquely generates a function over L which satisfies the required properties, i.e. is range-respecting and structure-preserving. (The set of active locations actually expressible in PL/I is a finite subset of L. It seemed better, however, to give a simple extension rule than to enumerate cases).

CD2.2.5 Allocate, Free, Contents, Assignment

Allocation uses env-cond (the "environment" for condition raising, see F4), because the STG condition has to be raised on storage overflow. Only level-one locations can be freed. Contents checks for non-initialised locations.

96 assign:

f_0'	the updated (level-one) storage
f'	the updated extended storage

The updated storage must ascribe contents v to location l, and leave unchanged the contents of locations independent from l. This alone does not ensure that a VARYING string location (dependent from l but) not contained in l has its current length unchanged, which therefore has to be postulated explicitly.

CD2.3 External Storage, File State.

DS	data set
REC	record
K-[R]	keyed [record]
uid	unique identifier

107 c-uid,
108 file-id:

The function c-uid is used in F1 to associate different uid's with different occurrences of file constant declarations, except that external declarations of the same identifier are commoned; the id is retrieved and represented as character string by the function file-id.

CD2.4 Environment

The environment pairs identifiers with denotations: locations for proper variables and parameters (cf F1 and F2), values for named constants (cf. F1), and certain functions or pairs of functions for DEFINED and BASED variables (cf. F2).

For BL-ENV (block environment, used for STATIC and EXTERNAL identifiers) and the function bl-env(pb), see CF1.

CD2.5 On-Establishment

On-establishments are named: oe, oe-0, oe-1, boe(block-), loe(local-) and loer (ref to local). oe's are passed (by value) to the int-bl function 'inside' which the passed oe is named boe. The loe is passed by reference to all functions which can update this loe, these functions then name the passed oe loer; otherwise the loe is passed by value (by taking c of loer) and 'keeps' the name loe (except for the case of int-bl).

ou-ENTRY-VAL is a set of functions -- with many similarities between these and the functions of ENTRY-VAL.

CD2.6 Cbif-Part

Instances of CBIF are named cbif, $cbif_0$, cbif-1. They are all maps. Specifically a cbif is a map from (a finite set of) cond-bif-nm's to either LOCations, NUMber-VALues or CHARacter-STRing-VALues depending on the cond-bif-nm. Cond-bif-nm ONSOURCE maps into a LOC whose type (i.e. edd) is CHAR-STR-VAL.

CF1 Block Structure

This Section deals with program, block, and procedure interpretation. It covers both procedure declaration, which associates with the procedure identifier the function denoted by the procedure, and procedure activation, which applies that function to the evaluated arguments.

Abbreviations for this Section:

acty[s]	ref to activity flag[s] (for non-RECURSIVE procedures)
en-f	entry function (denoted by a proc id)
major	"this is the major proc activation" (truth value)
st-env	static environment
nenv	new environment
abn	value returned on abnormal termination

CF1.1 Programs

1 int-prog

main-id	id of main proc
pb-sels	selectors to contained procs and blocks similarly: non-RECURSIVE procs, STATIC EXTERNAL dcls
st-ext-ids	STATIC EXTERNAL identifiers
st-int-ids	similarly, indexed by declaring proc or block
st-ext-locs	locations for st-ext-ids, indexed by id
st-int-locs	also indexed by sel to declaring proc or block
env-1	pairs EXTERNAL proc ids with their denotations
prog'	prog with dens for STATIC and EXTERNAL ids inserted
main-en-f	entry function denoted by main proc

Syntactically, prog is a set of procedures. Semantically, it behaves very much like a block whose declarations are these (EXTERNAL) procs, and whose body is a call to one of them, identified by main-id (hence the "pre:"). Other declarations which in a sense belong to this artificial outermost block are those of STATIC variables and those of file and EXTERNAL entry constants.

After a few "pure" auxiliary definitions, the first action is initialisation of the state. Like for genuine block activations, an id (aid-0) uniquely characterising the activation is generated. Next, storage for STATIC variables is allocated; the difference between EXTERNAL and INTERNAL is that in the former case declarations of the same id are commoned.

Since prog' has inserted into it, amongst other things, the denotations of the EXTERNAL procedures, the definitions of env-1 and prog' are mutually recursive (cf. N5.3). The third argument of eval-proc-dcl distinguishes the outermost use of the main proc from any other: it is <u>true</u> only for the former. (This is used in the interpretation of the RETURN statement, see below). - The function bl-env'(spb) collects into a bl-env (block-environment) the denotations to be inserted into a given (occurrence of a) block; the actual insertion is done by postulating a context function bl-env(pb) which retrieves the inserted bl-env (cf D2.4). For procedures not declared RECURSIVE, the activity flag (initialised to <u>INACTIVE</u>) is used for testing that they are indeed used non-recursively; this flag is also made part of bl-env.

After all the preparatory actions, the function denoted by the main proc is called, with dummy arguments except that condition names are paired (in oe-1) with their system actions. Finally, STATIC storage is freed and the activation closed.

CF1.2 Blocks

2 int-bl:

Immediately calls int-bl-1, the common part of block and procedure interpretation. By context conditions, it can be assumed that st-env and lenv (in int-bl-1) have disjoint domains.

3 int-bl-1:

lenv	local environment
loer	reference to local oe

Again, the new environment nenv is defined recursively, due to recursive procedures (see again N5.3 for a constructive reading). Note that parameters are not dealt with here, but in eval-proc-dcl. The loer is initialised to the passed oe but can be reset by ON and REVERT statements. The block epilogue is performed both on normal and abnormal termination.

CF1.3 Procedures

5 eval-proc-dcl:

This is a pure function - it returns a function with side effects, en-f, which is (the main part of, see int-bl-1) the denotation of the procedure identifier. Besides a list of locations (the "arguments" in the PL/I sense), en-f has additional arguments, passed to it from the calling block: oe and cbif, whose passing as arguments reflects the dynamic inheritance rules for PL/I on-units, and the statically determinable entry attribute of the entry reference; the latter is checked against the parameter and RETURNS attributes of the actually called procedure, which may be determinable only dynamically.

The function en-f tests and sets the activity flag for non-RECURSIVE procedures, checks the argument attributes against the parameter declarations and the RETURNS descriptor against the result attribute prescribed by the caller, and then sets up the new environment env' to be passed to int-bl-1; again, by context conditions, parameters and STATIC variables are disjoint. Since the RETURN statement terminates intermediate blocks, it is modelled by using the <u>exit</u> mechanism, with <u>ret</u> used to flag the returned value; therefore, this call of int-bl-1 always terminates abnormally (with <u>ret</u> or <u>go</u>), and the epilogue need not be written after it.

7 int-call-st:

Both the en-ref (an expression of type ENTRY) and the argument list are evaluated (see next function), and the value of the former, a function en-f, applied to the value of the latter (and to auxiliary arguments). Dummy locations allocated during argument evaluation have to be freed on termination.

8 eval-proc-ref:

The activation identifier (aid) is used to discover use of a "dead" entry value, i.e. one whose declaring block activation has been terminated. The (syntactic) case distinction in computing the elements of loc-l is made to see whether the argument matches the parameter descriptor without conversion.

11 int-ret-st:

If an expression is specified in the RETURN statement (which, by context conditions, is the case iff the terminated procedure is a function procedure, i.e. has an rdd), the expression value is converted to the completed rdd (RETURNS descriptor). The passing down of "major" from eval-proc-dcl has not been shown explicitly; it is necessary in order to ensure that the FINISH condition is raised (and the relevant on-units are executed) before termination of intermediate blocks.

CF2 Declarations, Reference, Allocation

This section covers the treatment of both declarations and references for variables (see F1.3.1 for procedure declarations). Sub-sections 2.1 to 2.3 discuss proper, based and defined variables respectively: in each section declarations and references are shown together to facilitate reading. Section 2.4 covers pseudo-variables; 2.5 the handling of initial; 2.6 some auxiliary functions. One restriction made to all variable types is that the major structure identifier is assumed to distinguish the declaration.

Abbreviation for this Section

oenv	old environment
nenv	new environment

loe	local oe component
env-ex	triple, see F3
lq	locator qualifier

1 eval-dcl: not used for STATIC (see F1 int-prog), parm (see F1 eval-proc-dcl), BI, LAB (see F1 int-bl-1), file-const and ext-entry (see F1 int-prog).

2 eval-l-ref:

vr	the reference to be evaluated to a location.

3 val-l-var-ref:

id	main identifier of reference
idl	sub-structure qualifications
sl	subscript list
esl	evaluated form of sl
dt	data type of id (statically determined)

The location of the whole variable (main-loc) is determined by the appropriate function and a merged index list is obtained from compose-indl (indl). From context conditions, it is known that the reference must match the declaration. Thus, if indl is not valid with respect to main-loc (see augm-index-lists CD2), only an out of range subscript can be to blame. Notice that there is no normal return from raise-cond for SUBRG.

CF2.1 Proper Variables

The term "proper variables" covers STATIC and AUTO (PARM declarations are considered in eval-proc-dcl). For such variables the environment directly contains the denoted location. Notice that it would be possible to combine the functions eval-static-dcl-tp and eval-auto-dcl-tp.

4 eval-static-dcl-tp:

l	the newly created location

Since STATIC declarations contain only restricted expressions, the empty environment passed from int-prog is adequate.

5 eval-auto-dcl-tp:

1	the newly created location

The environment under which the dd is evaluated is that of the surrounding block. The check that no references are made to redeclared variables is in the Context Conditions.

6 eval-l-prop-ref:
see assertions.

CF2.2 Based Variables

The declaration of a PL/I BASED variable is more complex than, for example, AUTO. In the latter case a single instance of the variable is allocated and its location serves as a denotation. For BASED, any number of instances may be allocated. The storage and retrieval of the LOCS (more strictly PTR-VAL's) is left to the programme. All, then, that is required is the ability to respond to ALLOCATE statements and references. On allocation the dd of the BASED declaration must be evaluated as well as, possibly, the dft-qual. Both of these evaluations must be performed in the <u>prologue</u> environment! The OE and CBIF components are, however, to be those of the reference. As with PROC-DEN (see F 1.3.1) the way of showing such "closure" is by generating a function which can be used to allocate variables. Similar considerations apply to reference and the required function pair is the denotation for a BASED variable. Notice that whilst both of the created functions are state changing, eval-based-dcl-tp is a pure fn.

alloc-based	the function creating allocations
ref-based	the function covering references

7 eval-based-dcl-tp:

dft-qual	default qualifier in the declaration
set-opt-r	the set option of the reference
oe-r	the oe component as at reference time
cbif-r	the cbif component as at reference time
set-opt	the required set option location
l (in alloc-based)	the newly created location
qual-r	the locator qualifier of the reference
qual	the required, evaluated, locator qualifier
dd-sub	the part of dd relevant to this reference
l (in ref-based)	the required location

(re alloc-based) the flag *BASED* is passed to alloc to support the check on freeing (see int-free-st).

Notice that the order in which the set-opt and refer-objects are set is not constrained.

(re ref-based) the agreement of the dd used to generate the LOC with the dd of the BASED declaration referenced is discussed under based-loc

8 init-refer-obs:
  s composite selector into a dd.

Because of the use of the arbitrary order construct, the result of this operation is indeterminate if more than one subject refers to the same REFER object. This function could have been integrated into alloc-based of eval-based-dcl-tp.

9 refers:
  s composite selector into a dd.

The selectors created are also applied to edd which, because of their differing syntaxes, is strictly wrong.

10 based-loc:
  qual the evaluated locator qualifier

For a given qual and dd, the relevant location is generated. The axioms given for constr-loc in D 1.2 define that dd must be left-to-right equivalent to that used to create the location. In particular this will check the validity of refer object values. Notice that if no such location exists we rely on (U1)(*false*) = *error*.

11 is-instance: Only because of the restriction to the language that expressions can occur only with REFER, can this be written as pure function.

12 left-struct-part: Notice that sub-structures within arrays of structures are not considered. This is because the mapping *would* be affected by subsequent fields.

13 eval-l-based-ref:
  m-loc required location
  see discussion above.

14 int-alloc-st:

set-opt	set option from allocate statement
set-loc	evaluated set-opt

Notice that an abnormal exit (GOTO) out of a function call invoked during allocation is defined as an error.

15 int-free-st:

ptr-val	evaluated locator qualifier from free statement
l	location to be freed.

The references in a free statement are evaluated like normal variable references (without sub-structure or subscript list). That evaluation provides the check on REFER object values. Because it is possible, via the ADDR built-in-function, to obtain a PTR-VAL to AUTO or STATIC variables, the free is preceded by a check that l was, indeed a BASED location. The check that l is a level one location is made in free.

CF2.3 Defined Variables

The situation with defined variables is similar to that with BASED, in that the required denotation is again a function. One important language difference is that there is some evaluation to be performed at the time of declaration: this results in eval-def-dcl-tp being a state changing operation.

eval-def-loc	the function covering references

16 eval-def-dcl-tp:

dd	dd of declaration
base	variable reference, on which current variable is based
pos	position
w	width
base-loc	evaluated lcc from base
i	evaluated pos
loc-l	location list
l	the required location

Whilst dd is evaluated immediately, base and pos are evaluated at reference time. Note that the function width is used only with aggregates containing strings. A dynamic test is made that the base location is connected. The length of loc-l is also checked dynamically at reference time. Under the test, the existence of l is guaranted. See extend for validity of such locs.

17 eval-l-def-ref:

m-loc	the required location.

CF2.4

Notice that condition pseudo variables are handled by eval-l-cond-pv.

The possibility to use the substr-pseudo-variable with arrays as second and third arguments can result in the generation of "inhomogenous" aggregate locations in the sense that different element locs have different, though fixed, lengths. Such locs do not satisfy the constraints of D2.2! This problem was noted rather late and has not been corrected.

b-loc	base location for SUBSTR
st	starting values for SUBSTR
len	length values for SUBSTR

18 eval-l-stg-pv-ref:

vr	the stg-pv-ref to be evaluated
atms	the atomic locations used by STR
res-loc	the result location of SUBSTR

19 distrib-substr-pv:

k	current length of b-loc
i	(scalar) value of st
j	(scalar) value of len

The requirement for this function results from the power of SUBSTR which is not reflected in sub-loc (SUBSTR can be applied directly to an aggregate meaning that an aggregate of sub-strings is to be created.) Notice no return from raise-cond of STRG is allowed.

CF2.5

The main difficulty with the definition of INIT is that an initial attribute given for a scalar within an array of structures is to be used to initialise the resulting non-adjacent scalars. The approach taken in the current definition is to generate a list of all index lists to such scalars (sc-indices) and to step through this list at the same time as the values. Firstly an abstract syntax is given for (partially) evaluated initial element lists.

iel	initial element list
s	composite selector into a dd

23 int-init:

l	location to be initialised
indl	list of index lists

Notice arbitrary choice of initial attribute order.

24 init-sc-parts:

indll	list of index lists
eiel	(partially) evaluated initial element list
if	iteration factor
v	value

Notice that the end of either indll or eiel terminates the function. Any iteration factor must, at the latest, be evaluated when required for expansion. By use of eval-init-elem-list the freedom is given to evaluate earlier.

25 sc-indices: no comment

26 eval-init-elem-list: no comment

27 init-comp: no comment

CF2.6

28 eval-dd:

u-dd	unit-dd
ev-u-dd	evaluated u-dd
ev-bpl	evaluated bpl
tp	string type
maxl	maximun length
vy	varyability
e-maxl	evaluated maxl

The extents and lengths are evaluated in arbitrary order. All other parts are unchanged.

29 eval-extent:

v	value of extent
cv	converted value of extent

30 compose-indl:

dd	relevant data declaration
idl	sub-structure qualifying identifiers, to be merged with
esl	evaluated subscript list
un-dd	unit dd
sdd-l	sub-structure dd list

The dd is used to guide the merging of idl and esl. Notice the constraints in the context conditions which guarantee this functions is defined.

31 eval-subscr-list:

CF3 Statements

Abbreviations for Section

cprefs	condition prefix set
snms	statement name set
t	text

lab-t	label part of an abnormal component-target
aid-t	AID part of an abnormal component-target
env-eu	the 5-tuple of environment information for executable units
env-st	the triple of environment information for proper statements
env-ex	the triple of environment information for expression
lab	label
test	logical expression in IF or WHILE
then-u	executable unit from THEN part of IF statement
else-u	executable unit from ELSE part of IF statement
eu-l	executable unit list
cv	control variable
cv-loc	location evaluated from cv
init	initialising expression from DO specification
byto	BY and TO part of DO specification
by	BY expression of DO specification
to	TO expression of DO specification
while	WHILE expression of DO specification
init-val	value evaluated for init
b	boolean value of evaluating a logical expression

General Comments

There are a number of different objects used to define the dynamic "environment" in which the denotation of a piece of text can be obtained (e.g. ENV, OE). These objects are passed as arguments in a way which shows their possible use. An object is only passed at all if it can be read or changed; it is only passed by reference if it can be changed. The relevant objects are

ENV	read to provide denotations for PL/I variables.
local EO	changed by ON or REVERT statements, read wherever an exception can occur.
block OE	read by REVERT statements.
CBIF	read by references to condition built-in-functions or psuedo-variables.
AID	read to test locality of labels wherever a <u>trap exit</u> can occur.
cprefs	read where exceptions can occur (Note: this is obtained statically by context functions).

Remembering that non-simple statements (e.g. IF) can contain other statements, it should now be possible to understand the choice of "environment tuples" for the various functions.

It is suggested that a first reading of this section is made ignoring GOTO, (described subsequently) so one should overlook all "cue-fns", int-goto-st, all *trap exit* units and the assertions.

int-ex-unit: no comment.

4 int-prop-st: note generation of appropriate environment tuples.

5 int-ex-unit-list:

7 iter-ex-unit-list: no comment

8 int-iter-grp:

sp-l	specification list

For DO statements which contain only a WHILE the power of the metalanguage (allowing a "side-effect" predicate) is such that a direct definition is possible. However, the metalanguage does *not* have the full richness of PL/I's DO and the more general forms are explained step by step. The next two functions were split out because of length, not for logical reasons.

9 int-step-do:

cv-dd	data description of cv
by-val	result of evaluating by
to-val	result of evaluating to
cv-val	result of accessing cv-loc
c-cv-val	result of converting cv-val
new-cv-val	new value to be assigned to cv-loc

Notice the arbitrary order of evaluating the initial, by and (if present) to expressions. The use of prom-ass is somewhat too general in that the targets are restricted (see context conditions) to scalars. A *while* construct is again used with a side-effect predicate. In this case the predicate implied by the various combinations of BY, TO and WHILE are defined in a block ending with *return*. The function, as written, assumes that a BY clause is present whenever there is a TO: thus BY1 of appropriate type has been inserted in the abstract program if TO is present without BY.

10 int-init-while-do: no comment.

11 int-if-st: notice that the nested *if* is on a static text property.

13 eval-truth:

bit-str	result of evaluating expr and converting to a BIT string

Returns true if any bit, in the bit string obtained by converting the test expression, is 1 BIT.

15 int-ass-st:

tr-l	target reference list
rhs	expression from right hand side of assignment statement
rhs-v	result of evaluating rhs
targ-loc	result of evaluating a tr to a location

The evaluation of the first target reference is separated so that the arbitrary merging with the evaluation of the right hand side can be shown. The prom-ass function must be called once per targ-loc because the language permits them to be of different shapes or types.

16 eval-targ-ref:

tr	target reference

17 prom-ass:

cval	the result of converting val to the type of loc.

The use of l-edd is somewhat over-dynamic in that the shape (i.e. everything except the bounds) could be deduced statically.

The Model for GOTO

The defining functions have been chosen throughout the definition to closely mirror the phrase structure given by the abstract syntax. The difficulty with the GOTO statement is that its execution cuts accross the phrase structure. Firstly, consider the effect of a GOTO which abnormally terminates execution of a phrase structure, as in

```
        BEGIN;
           .
           .
           BEGIN;
              DO I = 1 TO 10;
                 .
                 .
                 GOTO A;
                 .
                 .
              END;
           END;
           .
           .
          A: ...
        END;
```

The meta-language feature used to model such termination is exit, its effect is to close all of the defining functions until one is found with a trap exit and then to obey the trap exit body. Along with the exit a value can be passed which, in this case, contains the relevant information about the target label. (Notice that the semantics of exit are described in the meta-language by inserting a test after every call of a defining function which could result in exit).

The trap exit routines are given with each defining function which considers a phrase structure capable of introducing labels; the routine checks whether the label is local to the current text and, by means of the aid, if it is the appropriate instance thereof; if so, "normal" execution can be resumed. There are, of course, other places where some special action is required if a phrase structure is left abnormally (e.g. int-bl-1 in SF1): these must also contain trap exit units.

PL/I also permits GOTO statements to transfer control into phrase structure, as in

```
        GOTO B;
        DO;
        IF p THEN B:s1;
             ELSE s2;
        END;
```

Notice that it is not only necessary to begin execution in the right place, but also the necessary actions must follow such execution. The prefix "cue" has been given to the functions which model these two points because, as in acting, they show where to begin. There is considerable overlap between the body of a cue function and the corresponding normal function. An earlier version had been written which capitalised on this overlap by combining them. Unfortunately, this clouds the static nature of the cue mechanism.

It has been pointed out that omission of the trap exit on some levels of the phrase structure (but retaining all cue functions) would not change the overall semantics. In order to define the effect of a GOTO as locally as possible, this is not done. The assertions written show that a GOTO is always handled by the function covering the smallest phrase structure common to it and the target label.

1 int-ex-unit(trap): defines that a GOTO to any label within the current executable unit should be handled by cue-int-ex-unit. Note the check against the passed AID to distinguish between labels of different blocks or different block instances.

2 cue-int-ex-unit: part of the series of functions that model abnormal entry into phrase structure. Notice that an attempt to GOTO into an iterative group is caught here.

3 is-contained-lab: true even for labels within interative DO, see cue-int-ex-unit.

5 int-ex-unit-list(trap): no comment.

6 cue-int-ex-unit-list: no comment.

11 int-if-st(trap): no comment

12 cue-int-if-st: notice that, from the pre condition, if the label is not contained in the then clause, it must be contained in the else clause.

14 int-goto-st:

val-ref	value reference which is evaluated to find the target label

The check of the target aid against the contents of the AA component ascertains whether the block containing the label occurrence is still active. The possibility that it is not arises from the ability to assign a label to a label variable with a greater lifetime. The exit statement initiates abnormal termination.

CF4 Conditions

CF4.1 Condition Handling Functions:

loe	local on establishment by value
loer	local on establishment by reference
boe	value of on establishment of embracing block
oe	formal parameter on establishment
cbit,cbif$_0$,cbif-1	condition built-in function map
env	environment

char-pos	onchar character position in onsource
cn	condition name
cn-l	list of condition names
comp-cn	computational condition name
evd-cn	evaluated condition name
enabled	enablement status, in B
fct,fct1,fct2	let clause function definition names
fuid,fuid'	unique file identifier
ioc	input/output condition
ps	proper statement
snap	nil or SNAP
symbol-l	list of symbols
vr	variable reference, to yield file value

1 int-on-st

assert: the extension-and-override, +, to the loer is always an override in that int-prog initializes oe-1 to contain a suitable system-ou-entry-val (with no 'snap') as range element for any condition name.

2 ou-entry-val

The function, fct1, defined by [SNAP], and to be executed upon on-unit activation, is defined first, only to be immediately installed in the function, fct2, otherwise defined by ps. If ps is other than SYSTEM the on-unit is to be treated as a potential recursive, albeit parameterless procedure; the eval-proc-dcl call serves this requirement by 'dummying' up the corresponding entry-function.

3 system-ou-entry-val

No comments.

4 int-rev-st

This is the only function which requires access to boe.
assert: successively executed rev-st's in the same (PL/I) block activation and over some cond-nm has the same effect, with respect to the loer (only) as execution of one rev-st for that cond-nm provided no corresponding on-st 'slips' in-between!

5 int-sig-st

Other than for the case of disabled comp-cond-nm's this function 'prepares' for the call of an appropriate raise-...-cond function, by 'concocting' suitable arguments; those being of import being for comp-cond-nm CONV and io-cond KEY only!

6 raise-cond

Serves as the main funnel to the functions raise-comp-cond and raise-conv. For other than the KEY io-cond this function can also serve as the funnel into raise-evd-io-cond.

7 raise-comp-cond

The raising of a disabled comp-cond other than through int-sig-st (see this function), is illegal; hence yields error.

8 raise-conv

For CONV cbif has to be extended; and with ONSOURCE a location has first to be allocated: if a PL/I GOTO occurs of the eventually invoked on-unit, see below (after 8), then this location has to be freed. This then is the sole purpose of the exit specification. If calling alloc raises the STG condition then no location will have been allocated for the case the STG on-unit activation terminates abnormally, i.e. with a PL/I GOTO; in other words: the exit due to such a GOTO will not go via the exit trap of the raise-conv.

9 raise-evd-io-cond

Presently called by int-sig-st, raise-cond and numerous functions in F6 -- one could let these latter call raise-cond, but since they have to evaluate the file-variable anyway this approach was adopted, thus 'violating' the desire to let raise-cond serve as main-funnel, etc.

For functions 6, 7, 8 and 9 the actual invocation of the appropriate on-unit (loe(...)) is expressed by a functional reference of the form: (loe(...))(arg-1); one could perhaps have aided readability by writing instead: (let on = loe(...); on(arg-1))!

CF4.2 BIF Value- and Pseudc Variable Location Functions:

cbif-nm	cond-bif-nm
pos	integer position of ONCHAR in ONSOURCE
l,loc	locations
val	value
intg	integer

CF4.3 Enablement Status (Context) Functions:

c,c1,c2	cond-pref-set
t,t1,t2,t3,t',t"	text (of abstract programs)
cprefs	cond-pref-set
c-default	default cond-pref-set
cn	comp-cond-nm

Functions 14 - 19 are statically applicable, i.e. context functions:

15 cur-cond-prefs

Merges the immediate cprefs, c1, of the immediately, statically embracing ex-unit whose prop-st is a bl, or proc, with the combined cprefs, c2, inherited from all further, embracing "bl's" or procs.

16 im-cprefs

If t is a proc or an ex-unit then result is the cprefs of that proc or ex-unit. Otherwise an immediately statically embracing proc or ex-unit, t2, is found and its cprefs is taken as the result provided t2 is not an ex-unit whose prop-st is an on-st of whose cond-nm-list t is part. For that latter case the result is {}.

17 extract-cprefs

This function does most of the work. The inner, recursive call of extract-cprefs of the alternative, <u>else</u> case has the effect of the function first 'worming' its way all out to the external procedure, t ϵ π, then 'retracing' its path, while at each proc and ex-unit bl level merging their cprefs with the cprefs brought in from the outside.

18 merge-cprefs

The 'inner' (cn,enab) takes precedence over any outer (cn,enab').

19 im-embr-eubl-proc

Picks up the innermost statically embracing ex-unit if its prop-st is a bl, else the corresponding proc.

20 is-enab

Together with cur-cond-prefs the only interface functions of this section (F4.3).

CF5. Expressions.

General Comments.

env-op	information necessary for conversion, consisting of on-establishment prefix-set and CBIF's
env-ex	information consisting of environment, on-establishment and CBIF's

The expression evaluation part of the definition has three main entries, namely the functions eval-comp-expr , eval-expr and prom-conv.

Almost all decision are made statically using sdd. Only in the part handling indices and string length are dynamical decisions necessary.

All functions except eval-comp-expr and eval-expr have side-effects only via on-conditions.

There exists no normal return from raise-cond except in STRZ and UFL.

1 eval-comp-expr

Derives in the given environment the value of the expression t that conforms to the given target edd (r-edd). The value of t as a unit "in isolation" is evaluated using the function eval-expr. The result of eval-expr is converted and promoted to conform to r-edd.

2 eval-expr

Determines in env-ex the value of the expression t as a unit "in isolation". The values of subexpressions (operands/arguments) are evaluated in arbitrary order.

In the case of infix-, prefix-expressions or distributable BIF's the static data description (sdd) of the result depends on the operation/BIF-name and the sdd's of the operand(s)/arguments. The sdd of the result is derived by the function el-sdd.

CF5.1 Distribution.

3 distrib-op

el-sdd-l	list of sdd's, one sdd for each immediate operand/argument in the expression
val-l	list of values, one value for each immediate operand/argument in the expression

Evaluates a value of type r-sdd from the value(s) val-l[i] of type el-sdd-l[i] (i = 1,2,.., l val-l).

If the result is a scalar, conversions are performed as necessary (using the function conv) and the operation op is then applied (using the function apply-and-conv).

Operand/argument-lists consisting of aggregates or of aggregates and scalars are decomposed into sets of argument lists consisting of scalars. To each member of this set the necessary conversions (deviating from ANS-11 (9.1.1.6) a scalar value is usually converted more than once) and the operation are applied in any order to get the set of scalar results, which are then composed to the result aggregate value (corresponding to r-sdd).

4 gen-comp-edd

len-l	list of lengths or nil's, one item for each operand/argument. For each string-operand/argument the length of the string value occurs in the list; nil in all other cases.

Derives from the type of operation, sdd-l and len-l the evaluated data descriptions (edd's) to which the operands/arguments must be converted before applying the operation.

CF5.2 Operations.

5 apply-and-conv

opd-l list of scalar values, one for each operand/argument (as decomposed by distrib-op).

Evaluates the scalar result of type r-sdd by applying op to the list of scalar values opd-l. Depending on the type of operator (arithmetic, comparison, string, substring-class, mixed as defined by equations 8-12). The actfal evaluation is performed by the functions num-res, compar, substr-res, string-res, mix-res.

In the case of an arithmetic operator the exact mathematical result (as obtained by num-res) is transformed into a value belonging to the set VAL by the function arith-rep; arith-rep at the same time simulates rounding, truncation and condition raising.

12 arith-rep

The function is used in two ways:

in case of conversion it adjusts num to rdd in a way that represents the special properties of arithmetic in PL/I (rounding, truncation). This operation may raise the condition SIZE , OFL or UFL.

in case of an arithmetic operation it adjusts num to the maximal precision nn . This operation may raise the condition FOFL , OFL or UFL. Since the value is adjusted to the maximal precision num1 can be greater than the value allowed by the precision of rdd.

Deviating from ANS-11, this adjustment to FIXED is not implementation defined.

14 compar

Note that PTR values are addresses and not locations. Therefore pointer comparison is address comparison.

18 eval-non-distrib-bif

Determines the value returned by the BIF call. In the case of DIM , HBOUND and LBOUND the dimension expression (second argument) is evaluated and tested against the statically known number of dimensions before the aggregate (first argument) is evaluated.

In the case of STR all scalar components of the aggregate are converted to strings in any order by the function conv-to-str-ag and then concatenated to one string in left-to-right order by the function concat. Note the difference to evaluation of the STR pseudo variable in which no conversions occur.

CF5.3 Conversions.

21 prom-conv

If t-edd is a scalar edd, the value v is converted to a scalar of type t-edd using conv.

If t-edd is an aggregate edd an aggregate value of type t-edd is constructed, the components of which are built from the corresponding components of the converted value of v . v must be promotable to t-edd.

22 conv

dd is an edd to make padding or truncation of strings possible.

25 conv-to-char

In the case of arithmetic to character conversion the numeric value val is converted to an intermediate decimal value. This conversion guarantees that the resulting num is representable as a character string. Therefore the usage of symbl-to-val is pure functional, no conversion condition can occur.

CF5.4 Translation of Symbol-Lists.

CF5.4.1 Concrete Syntax of Constants.

The list of syntax rules (#26 - #45) facilitates the analysis and the evaluation of the (BIT | CHAR | NUM)-value of a given symbol-list.

CF5.4.2 Translation of symbol-lists.

symbl	list of CHAR-VALs

46 symbl-to-val

pred predicate of the form is-X, where X is a syntactical category name. pred is:

is-c-const	if function called by eval-expr (evaluation of constants),
is-c-num-str	if function called by conv (convert character value to arith),
is-c-prop-num-str	if function called by conv-to-char.

Parsing is performed on a symbol-list, which is checked against the predicate pred, and the value from the value set VAL corresponding to the list symbol is evaluated.

49 parse

Determines that unique tree of the syntactical category c-const whose terminal string (derived using term-str) is the list symbl. Note, that the grammar is unambiguous according to the rules of abstract syntax.

47 test-and-correct

Returns the character value list symbl if it can be parsed according to the syntactic category defined by pred. Otherwise a corrected string is returned. More precisely: In the latter case the condition CONV is raised. If it returns a corrected string using the ONCONV pseudovariable then the test against pred is repeated with the new string.

48 wrong-pos

Returns an integer in the range 0 to $\underline{l}$ symbl.

If the string symbl can be parsed according to the syntactic category defined by pred, 0 is returned.

Otherwise the first erroneous character position is returned. More precisely: either the position in symbl from which point no syntactical correct continuation (conforming to pred) exists, is returned or if symbl has a syntactically correct continuation, but it is incomplete, then the length of symbl is returned.

57 normalized

p	number of digits of the FLOAT representation of num

Tests if the list symbl corresponds to the edit rules for FLOAT to character conversion, which are not expressed by the syntax (#26 - #45).

58 correct-prec

Same as normalized but for FIXED.

THIS PAGE HAS BEEN INTENTIONALLY LEFT BLANK.

1. Abstract Syntax Classes

2. Functions