On the formal definition of PL/I

by K. BANDAT
IBM Laboratory Vienna
Vienna, Austria

INTRODUCTION

This paper describes a formal definition of PL/I which has been produced by a group in the IBM Vienna Laboratory. The paper contains the outlines of the method rather than details of PL/I. The definition currently exists as a Technical Report "Formal Definition of PL/I" which contains the abstract syntax and the semantics for PL/I program text. A second version of this Technical Report is under preparation and will complete the description of PL/I in these areas where the first version omitted language features, or showed major deviations from the current language. The language described in the Report is PL/I as specified in the PL/I Language Specifications of the IBM Systems Reference Library, supplemented by additional information.

Needs for language descriptions

The new potential user of a programming language—somebody who is assumed to be familiar with high level programming languages in general—needs information on the language on several levels of precision and completeness. At the first contact with the language he would like to know the salient features of the language, the parts of the language which are similar to languages he is familiar with, and the new concepts in the language which mark the step forward in programming language development. The user would like to see the potential areas of application for the new language. He should find all this information in an introductory document to the language like a primer. This primer in an intuitive way explains the concrete representation of programs and data, the various data types and data structures of the language, the structuring of programs by blocks and procedures, the operations which can be used in expressions, etc. A primer need neither be a complete description of the language nor need it be precise in all details. For tutorial purposes simplifications and omissions can be appropriate.

A description of the language with an increased level of completeness and precision is required for the actual user who starts to write programs in the language. He needs to have the full information on the concepts of the language and a complete set of rules for writing programs which will be accepted by a compiler. He would also want to know what result the execution of his program will give. Traditionally these needs for most programming languages were served by language manuals which state fairly precisely how a program has to be written. For the meaning of a program manuals frequently supply a natural language description which explains semantics of the language in an informal way, leaving a certain amount of questions open to the intuition of the reader who has to generalize from examples in the manuals and from the interpretation of programs on existing compilers.

These means will not satisfy the more advanced programmer who wants to know properties of the language or of a program in all details, say, e.g., in order to clarify whether two programs written to solve the same problem are in fact equivalent.

The highest level of precision in language description is needed for the implementer of a language. He requires a reference to the language which for every conceivable question can deliver a complete and correct answer. The implementer should neither be forced nor be allowed to answer questions on the language by his personal interpretation of a manual. The reference tool can also serve as the communication tool by which the language designer conveys the complete information on the language to the implementer. This makes it necessary that the method used in producing the reference document allows easy modifications of the description for the incorporation of language changes and language extensions.

We are convinced that the methods developed establish a tool for describing programming languages and PL/I specifically with a degree of precision which could not be achieved up to now by using informal methods.

Formal methods for language definition

If we accept the need for a rigorous, complete and
unambiguous definition of a high level programming language we have to find which methods and metalanguages can be used for the purpose.

It is frequently claimed that it is appropriate and more convenient for the user to describe a programming language with the aid of a natural language as the describing metalanguage. This may be tolerable for a language manual which has to serve a tutorial purpose as well as to give a description of a language. However, for achieving a precise and unambiguous definition the use of natural language is inappropriate. Natural languages are not well defined languages, they are lacking an exact syntax which allows the unambiguous parsing of sentences and clauses, and they employ words with multiple or vaguely defined meaning. Thus it can never be guaranteed that rules formulated in a natural language, unless applying rigorous restrictions, will have a well defined and unambiguous meaning. Current experience with language manuals written in English has shown up this fact very extensively. Only a formal method used in defining a programming language will yield the required precision and unambiguity. Although formalization is a well established method in the foundation of mathematics and logic, it was only recently that attempts have been made to apply similar methods to programming languages.

In defining a language we have to distinguish between the syntax of the language, i.e., the set of formation rules defining all strings which are well-formed programs, and the semantics of a program, i.e., the meaning of a well-formed program.

For the definition of the syntax of programming languages Backus Normal Form or some of its equivalents are commonly accepted tools.

For the definition of the semantics of a programming language two groups of methods are known, translation and interpretation.

The translational approach requires the existence of a completely defined language $L$. If a translator can be designed, which translates any well-formed program, written in the language $L'$ to be defined, into a program in the language $L$, then the definition of the language $L$ together with the translator $L'$ to $L$ completely define the semantics of $L'$. In order to be precise and unambiguous, the rules translating the program have to be written in a metalanguage which itself is completely defined.

The methods of semantic definition by interpretation have in common that they specify a function or a process which for any given set of input data and any given program text yields the output data. This can be achieved by designing an abstract mechanism which serves the purpose of a machine for which the language to be defined is the machine language. The complete logical description of the programming language contains the description of the possible states of the abstract machine and the way these states are changed by interpreting pieces of program text.

Language definition by a compiler

Occasionally it is claimed that an implementation of a language is sufficient as the definition of the language and that all questions about the language which cannot be answered by the manual, can be answered by processing sample programs with the compiler. This of course can be claimed only for the time when a language has already been implemented and not for the development phase of the language.

A compiler designed as a tool for converting a program written in a high level language into an object program in machine or assembly language which can be executed on a machine, has several drawbacks when used as the definition of the language. A compiler is defined and operable only in connection with its environment, i.e., the machine or assembly level target language and the actual machine. If a compiler should be used as the reference for a language, it has to be ensured that for the period of time while the compiler is used as the reference both this environment and the compiler itself remain unchanged. Currently this can hardly be guaranteed to its full extent for any implementation. Furthermore an implementation of a language defines many details in the semantics of a program which are not defined by the semantics of the programming language. Thus for PL/I the compiler contains a specific choice for the order of evaluation of operands of an expression, whereas the language leaves this order explicitly undefined. When a question concerning a language is solved by processing a characteristic program it cannot be distinguished how far the result reflects the situation of the language and how far the result reflects specific implementation or hardware properties.

It seems feasible to design a compiler which avoids these problems, for reference purposes only. This compiler—besides requiring a fixed environment—would need information on all points where information in addition to the semantics of the language is needed for interpreting a program. In fact this would be an implementation of an interpretive formal definition.

The system constituting the formal definition of PL/I

In designing the system for defining PL/I the required precision of definition, the presumptive user and the impact on the language itself had to be con-
considered. It was desirable to isolate the concepts and properties of PL/I from one another avoiding unnecessary interrelations and cross references and allowing separate consideration and evaluation of the concepts. Specifically a clear separation of all problems of program representation and notational conventions from the functional concepts of the language had to be achieved. Furthermore, it had to be shown in which areas the language requires a specific choice and additional definitions for a specific implementation.

The system designed for the definition of PL/I consists of several stages of processing and interpreting program text. The block diagram of Fig. 1 shows the elements of this system. The compile time facilities of PL/I are considered as a separate sublanguage, defining PL/I program text as the result of processing PL/I source text in a compile time preprocessor. A compile time concrete syntax defines well-formed source text. The concrete syntax of PL/I program text defines well-formed programs for semantic interpretation. Before the interpretation of program text all semantically irrelevant representation properties are removed by converting concrete program text to abstract text, a tree representation for a program. The PL/I machine interpreting abstract text is so designed as to reflect the concepts of PL/I by the various constituents of its state.

Trees and operations on trees

For the definition and handling of pieces of abstract text and parts of the state of the PL/I machine a notation has been developed which, free from redundancy, only reflects relevant properties of the considered objects.

A class of abstract objects has been defined which can be represented by trees. For this class of objects functions are given which enable the generation of trees from elementary objects and the modification of trees by deleting or changing subtrees. Properties of classes of abstract objects can be defined by an abstract syntax.

The basic functions will be explained with the help of examples. In these examples the characters enclosed in <> are meta-names used for discussion only and denote an elementary or a composite object. Capital characters denote elementary objects. The abstract objects <x> and <x'> used as examples are shown in their tree representation in Fig. 2 and Fig. 3.

Selectors on trees

In trees—or, more precisely, in the abstract objects they represent—a subtree attached to a node is selected by a selector function.

Thus in the example of Fig. 2 the sub-objects of the composite object <x> are selected as follows:

- \(<r> = \text{sel-1} (<x>)\)
- \(<s> = \text{sel-2} (<x>)\)
- \(<t> = \text{sel-3} (<x>)\)
As \(<r>\) again is a composite object, its parts are selected by:

\[
\begin{align*}
\langle u \rangle &= \text{sel-1} \,(\langle r \rangle) \\
\langle v \rangle &= \text{sel-2} \,(\langle r \rangle)
\end{align*}
\]

The object \(<u>\) can be obtained as a sub-object of \(<x>\) by a composite selector, where the dot serves for the functional composition:

\[
\langle u \rangle = \text{sel-1} \cdot \text{sel-1} \,(\langle x \rangle)
\]

The application of a selector to an object which has not been specified as selecting a proper object, yields the null object \(\Omega\).

\[
\text{sel-4} \,(\langle x \rangle) = \Omega
\]

The function \(\text{is-}\Omega\) allows checking whether a selector on an object yields a proper or a null object:

\[
\begin{align*}
\text{is-}\Omega \,(\text{sel-3} \,(\langle x \rangle)) &= \text{FALSE and} \\
\text{is-}\Omega \,(\text{sel-4} \,(\langle x \rangle)) &= \text{TRUE}
\end{align*}
\]

It is to be noted that selectors are functions that are local to the object for which they are defined, the same selector name \(\text{sel-1}\) is used for the object \(<x>\) and for its sub-object \(<r>\).

Selectors on the same level do not imply an ordering. Thus a question “which is the first branch attached to the node” has no defined answer.

Nodes of a tree do not according to the definition possess names. The meta-names in \(<>\) used here serve only for the explanation. If a node requires to be named, that name has to be attached as a separate branch. In the example of Fig. 3 the object \(<r>\) possesses the name \(R\), accessible by the selector \(\text{s-name}\).

**Generation of a tree**

For the generation of an abstract object from its elementary objects, a generation function \(\mu_0\) has been designed which establishes the arrangement of selectors and selected objects to form a new composite object. For the object of Fig. 2 the generating function is written as:

\[
\langle x \rangle = \mu_0 \,(\langle \text{sel-1:}\mu_0 \,(\langle \text{sel-1:}\langle u \rangle),
\langle \text{sel-2:}\langle v \rangle),
\langle \text{sel-3:}\langle t \rangle)
\]

The pair selector:selected object is enclosed in pointed brackets, where the selected object may itself be a composite object, generated by a \(\mu_0\)-function, as is the case in the example given for the object selected by \(\text{sel-1}\) on the first level.

**Modification of a tree**

For the modification of objects the modification function \(\mu\) has been introduced. It contains as the first argument the object to be modified, while the other arguments are pairs of selectors and objects as in the generation function \(\mu_0\). The function \(\mu\) can be applied in several ways. If a selector of an argument has already been defined for the object to be modified, the object paired with the selector replaces the old selected sub-object. If the selector has not yet been defined for the object, the \(\mu\)-function establishes a new branch for the object with the object from the pair \(\langle \text{selector:object}\rangle\) as a new sub-object.

The modification of the tree in Fig. 2 to form the tree in Fig. 3 would be written as:

\[
\langle x' \rangle = \mu(\langle x \rangle; \langle \text{s-name:}\text{sel-1:}\langle R \rangle)
\]
Another possible modification of a tree is the replacement of a proper sub-object by the null object $\Omega$. Selector functions yielding $\Omega$ are not represented in the tree representation of objects. Thus replacing a sub-object by $\Omega$ means the deletion of the respective selector from the tree. The modification of the tree $<x'>$ from Fig. 3 into the tree $<x>$ from Fig. 2 would be written as:

$$< x > = \mu (<x'>; <\text{s-name} \cdot \text{sel-1}: \Omega>)$$

**Predicates of objects**

For the definition of the structure and properties of all objects belonging to a specific class, a notation has been introduced which allows giving an abstract syntax for a class of objects. This notation in the definition of PL/I has been applied for the definition of states of the PL/I machine and for the syntax of PL/I programs in a generalized form. An abstract syntax for a class of objects again can be represented as a tree.

![Figure 4 - Abstract syntax of a class of objects](image)

A tree of the form shown in Fig. 4 would define the class of objects for which the predicate pred-1 is true, as the class of all those composite objects for which the application of the selectors sel-1 and sel-2 yield sub-objects for which pred-2 and pred-3, respectively, are TRUE. The application of a selector other than sel-1 or sel-2 on an object for which pred-1 = TRUE, yields $\Omega$.

The formula representation of the predicate tree of Fig. 4 in the notation used in the definition of PL/I is written as:

$$\text{is-pred} = (\text{sel-1}: \text{is-pred-2}, \text{sel-2}: \text{is-pred-3})$$

**Special objects**

Several places in the abstract syntax of program text as explained in the next section, required the representation of lists. As the sub-objects of an object are unordered, specific selectors $\text{elem}(i)$ where $i$ is integer, have been introduced to achieve an ordering of elements. This is shown as an example in Fig. 5 for an object of type pred-1, being a list of three elements.

![Figure 5 - Abstract syntax of a list of three elements](image)

A specific notation is used for the formula representation of a list, the example given would be represented by:

$$\text{pred-l} = (\langle \text{elem}(i): \text{pred-1}> | 1 \leq i \leq 3)$$

For objects whose sub-objects are attached by selectors of the type $\text{elem}(i)$ normal functions on lists as length, head and tail have been defined.

The structure of the PL/I machine involves major parts belonging to a class of objects called directories. A directory is an object whose sub-objects are selected by unique names. A directory of type pred containing sub-objects of type pred-1 would be defined as:

$$\text{is-pred} = (\langle n: \text{pred-1} > \| \langle \text{is-unique-name}(n) \rangle)$$

While in the definition of a list the condition following the vertical stroke defines the number of elements of the list, the condition following the two vertical strokes in the definition of a directory leaves the number of elements open but specifies the type of selectors.

**Abstract PL/I program text and syntax**

For the definition of the meaning of a PL/I program an abstract form of this program is interpreted.

The idea of representing a program in an abstract form has been shown by J. McCarthy. A well-formed concrete program has only one corresponding abstract form, whereas the abstract form of a program may have a multiplicity of concrete representations.

The abstract form for a PL/I program—its abstract text—is the result of processing the derivation of PL/I program text—a PL/I program not containing compile time statements—on the translator as shown in Fig. 1. By a set of rewriting rules the translator eliminates all semantically irrelevant notations like delimiters, and explicitly establishes those declarations which in the concrete text are defined by default rules, factored attributes, and implicit and contextual declarations. The rewriting rules also establish expansions as defined for the LIKE attribute and for not
fully qualified structure references and insert system-defined initial condition enabling prefixes. The rewriting rules eliminate the information on the ordering of branches of the derivation tree of concrete text where the order is semantically irrelevant. The abstract text can be represented as a tree.

For the abstract text a set of rules—the abstract syntax of PL/I—is given which define properties and structure of the tree representing the text. The rules define the branches which build the tree and the selectors which give access to these branches.

As an example the normal form of an assignment statement could be defined by the abstract syntax as:

\[
\text{is-assign-stmt} = (\text{<s-st:is-assign>}, \\
\text{<s-lab-list:is-name-list>}, \\
\text{<s-pref:is-cond-name-set>}, \\
\text{<s-left-part:is-reference-list>}, \\
\text{<s-right-part:is-expression>}, \\
\text{<s-name-opt:is-option>}),
\]

As explained for the tree notation used in the formal definition, selectors on the same level of an object have no implied ordering. Thus the selectors s-left-part and s-right-part in the definition of the assignment statement no longer reflect any ordering in a string. The mnemonic names only hint at the origin of the branches in the program text. In places where ordering is relevant it is reflected in the abstract text, thus, e.g., the ordering of statement labels is kept in the label list selected by s-lab-list, while the semantically irrelevant ordering of condition prefixes is no longer shown, the prefixes being in a set of names, i.e., in an unordered collection of elements.

**The PL/I machine**

The semantics of a PL/I program are defined by interpreting the preprocessed version of this program—its abstract text—on a PL/I machine. The PL/I machine is an abstract sequential machine not defined by hardware but as a set of possible states, together with the functions defining the transition from any state to its successor states.

The set \( \Sigma \) of possible states \( \xi \) which the PL/I machine can adopt is defined using the notation of abstract syntax in the same way as for abstract text.

Fig. 6 shows a simplified structure of a state of the PL/I machine.

The initial state \( \xi_0 \) of the PL/I machine includes the abstract text of the program as an argument of the first instruction in the control part.

A general language function \( \Lambda \) applied to a given state yields a set of successor states. The fact that a given state may have more than one successor state reflects the property of PL/I, that in several cases the order of evaluation of parts of program text is undefined.

![Figure 6 - Structure of a state of the PL/I machine](image)

The iterated application of the language function \( \Lambda \) on the initial state yields a computation:

\[
\text{comp} (\xi_0) = \{ \xi_0, \xi_1, \ldots, \xi_n \}
\]

provided that one successor state \( \xi_{i+1} \) is selected from the set of states defined by \( \Lambda (\xi_i) \). Making all possible selections the application of \( \Lambda \) to an initial state yields a set of computations. The terminal state \( \xi_n \) is element of a set \( \Sigma_e \) of states for which \( \Lambda (\xi_n) \) is undefined, i.e., no successor state exists.

A computation is successful if it terminates on interpreting the logical end of the program. If due to a semantical error in the program the interpretation does not continue until the logical end is reached, the computation terminates unsuccessfully. On the successful termination of a program the control part of the state becomes empty. The set of computations defined for a given initial state may contain both successful and unsuccessful computations.

**Unique name source N**

One of the parts of the PL/I machine is a source for unique names. These names are used to denote specific elements of the abstract storage and to serve as selectors in the directories of the machine. Unique names are also used to distinguish multiple uses of the same identifier for different declarations in a program. Within the scope of one declaration in the environment part \( E \) of the PL/I machine one unique name is associated with the declared identifier.
The storage part S

The storage part $S$ of the PL/I machine has been created to reflect the properties of storage as they are relevant in PL/I. The language requires the definition of contiguity in storage and the definition of pointers which permit reference to parts of storage not using the name for which the storage had originally been allocated. The storage part as described here is a simplified model which does not provide means for describing PL/I cells, areas and offsets and the notion of contiguity.

The storage part $S$ is a directory of the form:

$$\text{is-abstract-storage} = \langle \langle a: (\langle s-value: \text{is-value-representation} \rangle) \rangle, \text{is-unique-name}(a) \rangle$$

The unique name $a$ is an elementary address of storage. The value of a storage element is not directly accessible by the address, but via a second level selector $s-value$. This property serves the purpose of separating for a variable the cases of non-allocated storage from allocated but not initialized storage. For allocated but not initialized storage it holds that:

$$\text{is-value-representation}(s-value \cdot a(S)) = \text{FALSE}.$$  

Block activation and block local state parts

One of the salient features of PL/I is the possibility of controlling the scope of names, the scope of condition enabling prefixes, and storage allocation by the procedure and begin-block structure of a program.

The control part $C$, the dump $D$, and the block local state parts $E$, $CS$, and $EI$ are those parts of the PL/I machine which represent specifically the purpose of interpreting the effects of block activation.

An identifier is declared in a block by linking the identifier with a set of attributes in a declaration. The same identifier can be redeclared with a new set of attributes denoting a new entity in an inner block and denotes the new entity in the scope of this block. Thus an identifier may have various uses throughout a program.

For resolving the problem of multiple use of an identifier the environment part $E$ is used. In the interpretation of a program on the PL/I machine each identifier in its scope of declaration is associated with a unique name $n$ from the unique name source. The pairs $<\text{identifier}:n>$ for each block activation are collected in the environment part $E$.

On establishing a block activation, $E$ is generated from the environment part of a second block, updating it with the identifiers declared in the block being activated. If the activated block is a begin block, the second block is the dynamically preceding block activation. If the activated block is a procedure block, the second block is that block in which the procedure has been declared. In the united set for a redeclared identifier the association with the new unique name replaces the old one.

On termination of a block all identifiers declared as variables of storage class AUTOMATIC lose their meaning. Storage which has been allocated for these variables has to be freed on block termination. A set of identifiers in $EI$ serves the purpose of preserving, for use in the freeing of storage, all locally declared identifiers until block termination.

The condition status $CS$ for a block activation contains all information on the condition enabling status as established by condition prefixes, and on the action to be performed when a condition is raised as defined by the system action or by executed ON-statements.

The control $C$ in any state of the machine contains the instructions to be executed next. Each block activation has its own level of control, which is established on block activation and deleted on block termination, and is transferred to the corresponding part of the dump if a new block is activated. The last instruction executed before the control for the current block becomes empty, is the instruction performing the block termination.

The dump $D$ of a state of the PL/I machine reflects the nested structure of block activations of the program being interpreted. Whenever a new block is activated, a new dump $D$ is established, where the contents of the local directories, the dump and the control of the activating block, are stacked on top of the old dump $D$. On normal termination of this block the old contents as stored in the dump $D$ are re-established in the local directories and in the control, and the top level of the dump $D$ is deleted.

Parallel task and event part $PA$

In the interpretation of a program containing parallel tasks and I/O events these parallel actions are sequentialized. All active tasks and events have entries in one of the global directories, the parallel task and event part $PA$, containing all information necessary for their interpretation. At appropriate points in the program interpretation a priority evaluation decides which task will execute the next instruction or instructions. Several directories of the PL/I machine shown in Fig. 6 are global for all tasks and I/O events. Others are local to a specific task of I/O event. Thus, e.g., the control part has to contain only instructions belonging to one task. These task-local state parts are established for the selected after the priority evaluation using the information kept for this task in $PA$. When the task loses control, the contents of the task-local directories are saved by transferring them back to $PA$. 

From the collection of the Computer History Museum (www.computerhistory.org)
Meaning of names and global directories

An identifier which has been declared as the name of an entity is associated with a complete set of attributes.

In the abstract text—the normal form of the program to be interpreted in the PL/I machine—all declarations of one block are collected in the declaration part of this block. These declarations are interpreted in the prologue action for each activation of a given block. It is necessary to ensure that declarations can be retrieved during the execution of the program for various purposes like matching of attributes in parameter passing, evaluation of attributes on allocation of controlled storage or finding the block denoted by an entry name. This can be achieved either by searching through the abstract text whenever information on the text has to be retrieved or by transferring information on the text to a specific part of the PL/I machine where it can be accessed without employing a text searching mechanism.

The definition system for PL/I applies the second method. Global directories contain the meaning of names, i.e., what a name denotes, which attributes are associated with it and, if relevant, which storage has been allocated to it.

As already shown in the environment for a block activation each identifier which can be used in a reference is associated with a unique name n. This unique name serves as a selector in the attribute directory AT and in the denotation directory DN, both of which are global directories of the PL/I machine.

The **Attribute Directory** AT associates the unique name of an identifier with the attributes declared with the identifier. The entry in AT is part of the prologue action which has to be performed in the interpretation of a block activation. The attributes declared with an identifier are transferred to the respective element in AT denoted by the unique name, without performing any transformation or evaluation of the declaration. The declaration may contain expressions which for their later evaluation need the meaning of all names appearing in the expression. For this purpose the elements in AT contain an environment part in addition to an attribute part.

The abstract syntax of AT is formally defined as:

\[
\text{is-at} = (\langle n:\langle s\text{-attr}:\text{is-attr}\rangle, s-e:is-e\rangle) \mid \text{is-unique-name}(n))
\]

The attribute part with the predicate is-attr, identical to an attribute part in abstract text, has the abstract syntax:

\[
is\text{-attr} = (\langle s\text{-stg-cl}: (\text{is-static V is-automatic V is-ctrl}),
\langle s\text{-da}: (\text{is-mode}: (\text{is-real V is-compl})),
\langle s\text{-base}: (\text{is-bin V is-dec}),
\langle s\text{-scale}: (\text{is-fix V is-float}),
\langle s\text{-prec}: (\text{is-integer V is-integer-pair})\rangle),
\langle s\text{-init}:is\text{-initial\text{-value}}\rangle)
\]

The **Denotation Directory** DN associates the unique name of an identifier with the entity it denotes. An entry name denotes the body of a procedure and the environment of the declaration, a label denotes a statement list and a block activation identification, variable names denote generations which basically are collections of storage addresses.

The abstract syntax of DN is:

\[
is\text{-dn} = (\langle n:\langle s\text{-body}:is\text{-body}, s\text{-e}:is\text{-environment}\rangle
\langle s\text{-st-list}:is\text{-statement\text{-list}},
\langle s\text{-bl-ldf}:is\text{-block\text{-identification}}\rangle) \mid \text{is-unique-name}(n))
\]

The **Aggregate Directory** AG has the form:

\[
is\text{-ag} = (\langle n:\text{is\text{-generation\text{-list}}}, \| \text{is\text{-unique\text{-name}}}(n)\rangle)
\]

An element of a generation-list, i.e., one generation has the form:

\[
is\text{-gen} = (\langle s\text{-da}:is\text{-da}, s\text{-addr:is\text{-a}}\rangle)
\]

Aggregates are denoted by unique names which are selectors to the aggregates in the aggregate directory.

The generation serves the purpose of collecting the storage element allocated for a variable and information on the data attributes. The respective parts of the generation are selected by s-addr and s-da. The notion of generation reflects a basic concept derived for the current formal definition of PL/I. In PL/I variables do not always possess private storage but may be based on already existent and allocated variables. This sharing pattern is reflected by the way in which two generations comprise the same elementary storage items a(S).

Example for the interpretation of a PL/I variable

A simple example will show how the various directories are involved in the interpretation of a PL/I variable.

Let us assume a block B as

```
BEGIN; . . . DECLARE X FLOAT (8) AUTOMATIC, Y, Z; . . . X = expression; . . . END;
```

```
The example of Fig. 7 shows the same block transformed into abstract text, where all declarations are collected and completed in a declaration part.

When the block B is activated during the interpretation of the abstract text, its declaration part in the prologue action is evaluated and parts are transferred to the local and global directories.

All locally declared identifiers are linked to unique names. The new environment is made up by selecting the unique name of an identifier using the identifier itself as selector.

Continuing in the prologue action an entry is made for each newly declared identifier in the attribute directory AT.

Using the same unique names \( n_x, n_y, n_z \) as selectors, the prologue action establishes entries in the denotation directory DN, taking unique names \( b \) from the set of aggregate names.

When the denotation for the variable X has been established a generation is created. The later is established as element of AG selected by \( b_x \).

The generation-list of \( b_x \) contains only one element, the variable X having been declared AUTOMATIC. Only in the case of CONTROLLED storage class the aggregate would contain a list of generations. The generation contained in the aggregate for X denotes one storage item \( a(S) \) and the data attributes for a value which can be assigned to \( a(S) \).

In interpreting the statement-list of the block B there occurs a reference to X in an assignment statement. The value of the expression has to be assigned...
to the proper location in storage, which is found by
the access chain.

\[ X \rightarrow n_x \rightarrow b_x \rightarrow a \]

Control part and state transitions
The control part \( C \) of the PL/I machine is signifi­
cant for the changes of the state of the machine in the
process of interpreting abstract text. Changes of any
subpart of the PL/I machine—as in the example in
previous section—can only be performed by executing
instructions which are elements of the control part.

Figure 11 – Generation for a variable \( X \)

The control part has the form of a tree where the
nodes contain instructions. The formula representa­
tion equivalent to the control tree in Fig. 12 is used for
representing a control tree in the formal definition.

The PL/I machine is a sequential machine, i.e., no
two actions can be performed in parallel. Thus only
language properties can be described which in their
logical significance can be sequentialized.

All terminals of the control tree designate instruc­
tions which are candidates for execution, but only one
instruction at a time is executed. This choice of one of
several instructions for execution reflects those situa­
tions in PL/I where the order of execution for some
program parts is undefined, as, e.g., in operand evalua­
tion. Each instruction in the control tree can have ar­
guments and a set of successor control trees. Only
those instructions for which the set of successor con­
trol trees is empty, are the proper instructions on the
terminal nodes of the control tree which are candi­
dates for execution. Thus instr-3 in Fig. 12 is no can­
didate for immediate execution. Each instruction is
deleted from the control after it has been executed.

A form of the control tree using specific selectors is
used to allow the evaluation of arguments of an
instruction.

In the control tree of Fig. 13 the arguments \( f_1 \)
and \( f_2 \) are evaluated by the execution of instructions
instr-2 and instr-3.

A proper instruction in the Formal Definition of
PL/I is defined in the following format:

\[ \text{instr-name} \left( x_1, \ldots, x_n \right) = \text{RETURNS}: e_0 \]

Such an instruction on execution returns the value
obtained in evaluating the expression \( e_0 \), e.g. to an ar­

gument place if used as in Fig. 12, and changes parts of
the state of the PL/I machine selected by \( s\text{-part}_n \).

The arguments \( x_1, \ldots, x_n \) may contain pieces of
the abstract text to be interpreted. The expressions
\( e_0, \ldots, e_n \) are expressions in the metalanguage of the
formal definition, i.e. functions on abstract objects
involving \( \mu \)-functions and selectors. The arguments
\( x_1, \ldots, x_n \) can appear in these expressions.

As an example the process of updating \( DN \) with a
new denotation for a declared variable with the unique
name \( n_x \), as shown in the example in the previous sec­
tion would require a control tree for execution:

\[ \text{update-dn} \left( n_x, b_x \right), \]

\[ b_x: \text{un-n} \]

The instruction would be defined as

\[ \text{Def.: un-n} = \text{RETURNS: head} \left( N \right) \]

\[ \text{s-part}_n: \text{tail} \left( N \right) \]

\[ \text{Def.: update-dn} \left( t_1, t_2 \right) = \]

\[ s\text{-dn}: \mu \left( DN; <t_1; t_2> \right) \]
The instruction update-dn \((n_x, b_x)\) can be executed, if the execution of un-n has brought the unique name from the list \(N\) to the argument \(b_x\). In taking off the head element of \(N\), un-n changes the part \(N\) of the PL/I machine to the tail of its previous state. Finally the execution of update-dn with the arguments \((n_x, b_x)\) adds a new element \(b\) to \(DN\), selected by \(n_x\).

The control tree is initialized in the initial state \(\xi_0\) of the PL/I machine with initial-instruction \((t_0)\) where \(t_0\) is the abstract text of the program. If the program is interpreted successfully the interpretation process stops if the control \(C\) becomes empty. For an unsuccessful computation the interpretation terminates before the text is completely interpreted by arriving at a state of the machine for which no successor state is defined by the language function \(\Lambda\).

ACKNOWLEDGMENT

The methods for the formal definition of PL/I as presented in this paper have been developed by P. LUCAS and K. WALK in the IBM Vienna Laboratory. Vital sources of information for the methods involved were J. MCCARTHY, C. C. ELGOT, and P. J. LANDIN.

REFERENCES

1. PL/I definition group of the Vienna laboratory: Formal definition of PL/I
   IBM Laboratory Vienna Technical Report TR 25.071 1966
2. IBM Systems Reference Library:
   Operating system/360 - PL/I language specifications
   Form-No. C28-6571-4 1966 IBM Data Processing Div White Plains N Y
3. J McCarthy
   A formal description of a subset of ALGOL—\(\text{in}\): Formal language description languages for computer programming
   T B Steel ed 1966 pp 1-12 North Holland Publ Company Amsterdam
4. C C Elgot A Robinson
   Random-access stored-program machines. An approach to programming languages
   J ACM 11 1964 no 4 pp 365-399
5. P J Landin
   A correspondence between ALGOL 60 and Church's lambda-notation
   Comm ACM 8 1965 no 2 pp 89-101 and no 3 pp 158-165