(971)

RECEIVED BY TIC 123 26 1973

HOPKINS COMPUTER RESEARCH REPORTS
REPORT # 24
DECEMBER 1972

PRODUCTION SYSTEMS: A FORMALISM FOR DEFINING THE SYNTAX AND
TRANSLATION OF PRACTICAL PROGRAMMING LANGUAGES

BY
HENRY F. LEDGARD

RESEARCH PROGRAM IN COMPUTER SYSTEMS ARCHITECTURE
COMPUTER SCIENCE PROGRAM
THE JOHNS HOPKINS UNIVERSITY
BALTIMORE, MARYLAND

MASTER

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

PRODUCTION SYSTEMS: A FORMALISM FOR DEFINING THE
SYNTAX AND TRANSLATION
OF PRACTICAL PROGRAMMING LANGUAGES * *

Henry F. Ledgard^{*}
Computer Science
Johns Hopkins University
Baltimore, Maryland 21218

*Current Affiliation
Computer and Information Science
University of Massachusetts
Amherst, MA 01002

CR Categories: 4.2, 5.21, 5.23

NOTICE

This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Atomic Energy Commission, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately owned rights.

Keywords: Syntax, Translation, Context-Sensitive Grammars, Compilers, Lambda Calculus, Abstract Syntax, Formal Definition, PL/I, Standards.

**Publication supported by the U. S. Atomic Energy Commission under contract number AT (11-1) 3288.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

See

ABSTRACT

This paper investigates the application of a formalism called production systems to define the syntax of a computer language and its translation into a target language. Several properties appear well-suited to this task:

- (a) The formalism can be used to specify exactly the syntax of a computer language, including context-sensitive requirements.
- (b) The specification of the context-free portions of syntax and the context-sensitive portions of syntax can be isolated.
- (c) The same formalism can be used to specify the translation of a language into another.

The following example applications of production systems are given:

- (a) A specification of the syntax of a significant subset of PL/I.
- (b) A specification of the abstract syntax of the PL/I subset.
- (c) A specification of the translation of lambda-calculus expressions into normal form.

I. INTRODUCTION

1.1 Motivation

This paper presents the formalism of production systems and investigates its application to define the *complete syntax* of a computer language and its *translation* into a target language.

The need for suitable methods for formal definition of computer languages is evident. For language designers, implementers, or users, there is a clear need to be able to define rigorously what strings in a language are legal programs and what the programs "mean", possibly in terms of some suitable (for humans or computers) target language. While not all attempts at formal definition appeal to notions of syntax or translation, the notions of syntax and translation are used widely enough to warrant investigation into methods for formalizing them.

The elusive but essential notion we must face in choosing any formal system is its "acceptability" [2] in a particular application. The criteria for judging the acceptability of production systems in their application to define syntax and translation are many. Certainly among these are conciseness of definition, perspicuity of definition, the amount of material needed to understand the formalism, and its ability to adapt from one language to another. I shall discuss each of these criteria in turn, and in the process note what motivated the author to pursue the approach taken here.

Perhaps the most important reason for the widespread use of context-free grammars to define syntax, notably Backus-Naur form, is the conciseness and simplicity with which context-free portions of syntax can be specified. Owing to the more complex nature of context-sensitive requirements on languages and the specification of translation, some additional complexity for production systems must be expected. On the other hand, when viewed as a generative grammar, production systems provide a significant degree of conciseness that synthetic or generative (as opposed to analytic or algorithmic) methods of definition possess.

Some additional conciseness for production systems in the specification of syntax and translation has been obtained by introducing abbreviations to the basic notation. Three principal factors governed the kind of abbreviations introduced: first, reduction in the length of a specification; second, an attempt to isolate the context-free portions of syntax, context-sensitive portions of syntax, and translation; and third, an attempt to develop a conceptual framework facilitating language specification.

Perspicuity of definition appears more important than conciseness of definition. Three factors seem paramount in determining perspicuity: notation, segmentation of the parts of a definition, and the conceptual framework within which the definition is given. It is tempting for the author of a work to introduce notation, terminology and conventions that become convenient for him to use, but which often obscure the work and its contribution to others. In the effort to avoid this temptation, this author has spent many hours in developing the notation and conventions for production systems in the hope that they would be well-suited to computer languages.

Similarly, a good deal of effort has been expended to provide a formalism whereby a user of a may obtain the degree of precision that he is seeking. According, the productions in the sequel are given in "levels". This segmentation of a production system specification certainly adds to the perspicuity of a production system's definition.

The conceptual framework of a formalism is vital to its proposed application in that the conceptual framework either lends itself naturally or unnaturally to the application. Production systems are couched in a conceptual framework of generative productions used to enumerate sets of strings. The conceptual notions of "generative productions", "sets", and "strings" underlies all production systems specifications given here and lends a uniformity of approach. Rather than talk about tables of identifiers, parsing schemes for scanning programs or algorithms for computing functions, we talk about sets of identifiers, sets of programs and sets of n-tuples that define functions.

Superimposed on the basic notation for production systems is a notation for defining functions. Via the function-like notation portions of a productive system *appear* algorithmic in that, given arguments of a function, the productions may be used to *compute* the result. The function-like notation greatly relieves the difficulty with production systems that strictly speaking all sets are defined generatively. Generally, the basic constituents of a language (for example, the class of arithmetic expressions or blocks) are defined here with the basic generative notation. On the other hand, auxiliary constituents needed to complete the specification are defined via the function-like notation. For example, the function-like notation is used to define the mapping of an expression and its local environment into the derived type of the expression.

Considering the complexity involved in the specification of syntax and translation, the amount of material needed to understand basic formalism of production systems is small. While some complexity to basic formalism is introduced by adding abbreviations and alternate notations, the basic simplicity of the formalism remains.

The ability of production systems to adapt from one language to another remains to be judged. The syntax of one small language, PL.1, is defined here with a production system. No production system specifying the syntax or translation of a complete, commonly-used language has been attempted. Hence the acceptability of production systems to adapt to various types of language is untested.

1.2 Background of the Formalism

The mathematical underpinnings of production systems are due to Emil Post [14] in 1943 and Raymond Smullyan [15] in 1961. With suitable syntactic changes production systems are equivalent [7] to Smullyan's "elementary formal systems", which can be used to specify any "recursively enumerable" set [15]. The set of strings comprising all syntactically legal programs in a computer language and the set of pairs of strings comprising all syntactically legal programs in a computer language and their translations into a target language are just two examples of recursively enumerable sets.

The application of a logically modified variant of the formal systems of Post [14] Smullyan [15] and Trenchard More [12] to specify completely the syntax of a computer language was first made by John Donovan [3] in 1966. Donovan applied a formal system called *canonic systems* to specify the set of legal programs in a computer language, including the specification of several context-sensitive requirements on the set of legal programs, like the requirement that all statement labels in a program be different.

The formal system presented here has undergone many stages of evolution in the attempt to tailor the formal system to practical programming languages. Hence the name "production systems" is used to distinguish the formal system presented in this paper from the formal systems of Post, Smullyan, and Donovan. The notation and terminology for production systems presented here stems from Post and Smullyan, but for the most part is new.

1.3 PL.1

Before discussing the formalism of production systems, we present a small subset of PL/I, called PL.1. This subset was chosen before its syntax was defined via production systems. The PL.1 subset was selected to embody several significant aspects of a full PL/I syntax and to reflect several major kinds of typical syntactic requirements, for example block structure, compatibility of declaration and use, use of multiple data types, and conversion rules. A context-free description of its syntax is given in Table I.

TABLE 1: CONTEXT FREE SYNTAX OF PL.1

(1.1) Programs

p	PROGRAM	< i: PROCEDURE; u ₁ u ₂ ... u _n END j; >
u	UNIT	< iu du i' : u >
iu	IMPER UNIT	< <u>ASGT STM</u> <u>GOTO STM</u> <u>IF STM</u> <u>ALLOCATE STM</u> <u>FREE STM</u> <u>RETURN STM</u> <u>BLOCK</u> >
du	DCL UNIT	< <u>ELEMENT DCL</u> <u>STRUCTURE DCL</u> >

(1.2) Imperative Units

	ASGT STM	< r = e; >
	GOTO STM	< GOTO i; >
	IF STM	< IF e THEN iu >
	ALLOCATE STM	< ALLOCATE i; ALLOCATE i SET(j); >
	FREE STM	< FREE i; FREE j+1; >
	RETURN STM	< RETURN; >
	BLOCK	< BEGIN; u ₁ u ₂ ... u _n END; >

(1.3) Declarations

	ELEMENT DCL	< DECLARE i eal; >
	STRUCTURE DCL	< DECLARE l i sal, msl; >
eal	EL ATR LIST	< dta 0 sca 0 sta >
sal	STRUC AIR LIST	< sca 0 sta >
msl	MINOR STRUC LIST	< n ₁ i ₁ succ ₁ , n ₂ i ₂ succ ₂ , ..., n _k i _k succ _k >
succ	STRUC SUCC	< dta n i msl >
dta	DATA TYPE ATR	< A FIXED FLOAT POINTER BIT(n) CHAR(n) >
sca	SCOPE ATR	< A INTERNAL EXTERNAL >
sta	STORAGE ATR	< A STATIC AUTOMATIC BASED(j) >

(1.4) Expressions and Atomic Components

e	EXPRESSION	< n "c ₁ c ₂ ... c _n " r NULL ADDR(r) e ₁ q e ₂ >
r	REFERENCE	< i j+1 i ₁ .i ₂ ... i _n >>
l	LETTER	< A B ... Z >
n	NUMBER	< 0 1 ... 9 >
i, j	IDENTIFIER	< l ₁ l ₂ ... l _n >
c	CHARACTER	< n i >
q	OPERATOR	< + '<' = '>' >

lo**lo****lo****lo**

PL.1 contains numerous context-sensitive requirements on its syntax. Table II contains a list of some relevant statements abstracted from the PL/I Language Reference Manual [18]. These statements illustrate the variety and complexity of the full PL/I syntax. It should be pointed out here that a significant effort is required to organize these requirements so that a coherent view of PL/I syntax results.

Note for PL/I Programmers. In the sequel, a number of PL/I examples are given, some of which contain errors and some of which contain constructs that perhaps could be better replaced by constructs not in the PL.1 subset. In cases where errors occur, we are concerned here only with only syntactic errors, i.e., those that are picked up by a PL/I compiler, and not semantic or run-time errors.

TABLE II SOME REQUIREMENTS ON THE SYNTAX OF PL.1

1. An identifier specified as an element variable or major structure in a DECLARE statement is said to be *explicitly* declared.
2. A label identifier prefixing an imperative statement is said to be *explicitly* declared.
3. An identifier that has not been declared explicitly is *contextually* declared if it appears in the BASED attribute, in a SET option, or on the left-hand side of a pointer qualification symbol is given the POINTER attribute.
4. An identifier that appears in a program and is not explicitly or contextually declared is said to be *implicitly* declared.
5. The scope of a contextual or implicit declaration is determined as if the declaration were made in a DECLARE statement immediately following the PROCEDURE statement of the program.
6. Multiple declarations are in error. That is, within a given scope, an identifier can have one and only one meaning. For example, the same identifier cannot be used for both a pointer and a floating-point variable.
7. If no storage class attribute is specified and the scope is internal, AUTOMATIC is assumed. If no storage class attribute is specified and the scope is external, STATIC is assumed. If neither storage class nor the scope attribute is specified, then AUTOMATIC is assumed.
8. Automatic and based variables can have internal scope only. Storage class attributes cannot be specified for members of structures. The EXTERNAL attribute cannot appear with a based variable declaration.
9. In all of the external declarations for the same identifier, the attributes declared must be consistent. Note: This rule is not enforced by the PL/I (F) compiler.
10. All structure variables in a structure expression must have identical structuring. Identical structuring means that the structure must have the same minor structuring and the same number of contained elements and arrays and that the positioning of the elements and arrays within the structure must be the same. Identifiers do not have to be the same. Data types of corresponding elements do not have to be the same as long as valid conversion can be performed.

Table II (continued)

11. Members of structures always have the INTERNAL attribute and cannot be declared with any scope attribute.
12. In a structure assignment statement, the target variable must be a structure, and all of the structure operands must have the same number of contained items.
13. The based variable appearing in the ALLOCATE statement may be an element variable, an array, or a major structure.
14. The pointer variable qualifying a based variable cannot itself be based. Pointer variables cannot be operands of any operators except the comparison operator "=". Assignment of a pointer variable value can be made only to another pointer variable.

The requirements of Table II rule out many potential PL/I programs.

The program

```
P:    PROCEDURE;  
      L:  X+1;  
      L:  Y+1;  
END P;
```

is illegal due to a multiple label declaration. The program

```
P:    PROCEDURE;  
      DECLARE Q POINTER;  
      Q=1;  
END P;
```

is illegal due to an assignment of an arithmetic value to a pointer.

The program

```
P:    PROCEDURE;  
      X = 1;  
      BEGIN;  
          X = NULL;  
      END;  
      X = X+1;  
END P;
```

is illegal since X is implicitly declared with the attribute FLOAT and hence the assignment of a NULL pointer to X in the BEGIN-END block is illegal.

Consider the procedure BUILD given earlier. The replacement of

```
DECLARE NEWPTR POINTER EXTERNAL;
```

by

```
DECLARE NEWPTR FIXED EXTERNAL;
```

would be illegal since all subsequent uses of NEWPTR require a variable with the attribute POINTER. The replacement of

```
DECLARE TREEPTR POINTER EXTERNAL;
```

by

```
DECLARE TREEPTR POINTER EXTERNAL;
```

would be illegal since TREEPTR would then be implicitly declared with

an arithmetic attribute causing subsequent assignment statements to be in error. The replacement of

```
2 LP POINTER,
```

in the declaration of TREE by

```
2 LP FIXED,
```

would cause a type error in the comparison of TREE.LP with NULL.

One could go on and on listing numerous examples of illegal programs. Suffice it to say that the context-sensitive requirements on PL.1 impose many constraints on the writing of legal programs.

If one investigates the darker corners of the syntax of PL/I as defined by the manual, one finds several instances where requirements are not clear. For example, is the following procedure legal?

```
P:  PROCEDURE;  
    DECLARE P POINTER;  
    P = NULL;  
END P;
```

Does the declaration of the pointer P override the use of P as an entry name or does the declaration cause a multiple declaration of P?

Consider also the following proposed program:

```
P:  PROCEDURE;  
    DECLARE X FIXED;  
    X = 0;  
L:  DECLARE Y FIXED;  
    X = X+3;  
L:  X = X+2;  
    IF X<15 THEN GO TO L;  
END P;
```

Does this program contain a multiple declaration of the label L?

The above discussion indicates that a formal definition of PL/I syntax must ultimately be based upon a particular implementation. Furthermore, the process of writing a formal definition forces one to make choices that might be overlooked in an informal definition. A complete formal definition of syntax must weed out the problems, and a good formal definition must provide a coherent framework within which these problems may be explained.

2. PRODUCTION SYSTEMS

2.1 The Basic Formalism

Formation Rules:

A *production system* consists of a collection of the following items:

1. An alphabet called the *object alphabet*.
2. An alphabet called the *predicate alphabet*, each of whose numbers is assigned a unique positive integer called its degree.
3. An alphabet called the *variable alphabet*.
4. An alphabet called the *punctuation alphabet*.
5. A finite collection of *productions*, each of which is well-formed according to the definition given below.

In a well-formed production (see Figure III), it is necessary to be able to determine the alphabet from which each symbol is drawn. Accordingly the following symbols will be used. (a) strings of capital letters, possibly interlaced with digits, spaces and tuple signs, for predicate alphabet symbols (b) strings of lower case English letters (possibly subscripted or super-scripted) for variable alphabet symbols (c) the symbols

- + implication sign
- , conjunction sign
- : tuple sign
- ' quote sign
- < > left and right bracket signs

for punctuation symbols, and (d) symbols not in the predicate, variable and punctuation alphabets for object alphabet symbols.

FIGURE III

SUMMARY OF PRODUCTION SYSTEM NOTATION AND TERMINOLOGY

A *well-formed term* consists of a concatenated sequence of variable and object alphabet symbols, e.g., " i ", " e_1 ", " $e_1 + e_2$ " and " $r = e;$ ". A *well-formed term tuple* consists of a sequence of n terms each separated by a tuple sign and enclosed by a left and right angle bracket sign, e.g. " $\langle r = e; \rangle$ " and " $\langle i : j \rangle$ ". The number n of terms is called the degree of the term tuple. A *well-formed atomic formula* consists of a predicate alphabet symbol of degree n , followed by a well-formed term-tuple of degree n e.g., "ASGT STM $\langle r = e; \rangle$ " and "NOT IN $\langle i : j \rangle$ ", where "ASGT STM" and "NOT IN" are predicates of degrees 1 and 2 respectively. A *well-formed production* consists of

(a) an atomic formula, or

(b) an atomic formula followed by the implication sign and a

sequence of atomic formulas each separated by the conjunction sign.

An atomic formula preceeding the implication sign or occurring alone is called a *conclusion*. An atomic formula following the implication sign is called a *premise*. A production containing no premises is called an *atomic production*.

In the specification of written expressions in computer languages, it will often be necessary to include production systems

symbols as members of the object alphabet. Since capital letters, digits, the implication sign, and conjunction sign, cannot occur within the brackets of a term tuple as predicate, variable, or punctuation alphabet symbols, these symbols can be used in a term tuple as object alphabet symbols. Furthermore, strings containing

variable alphabet symbols, tuple signs, and bracket signs can also be used as members of the object alphabet provided that the strings are enclosed by the quote signs when used within a production. For example, consider the following productions

ID<A>
ID
:
:
ID<Z>

IDLIST<i> + ID<i>
IDLIST<l,i> + IDLIST<l>, ID<i>

DIFF ID<A:B>
DIFF ID<A:C>
:
:
DIFF ID<Z:Y>

NOT IN <i:j> + ID<i>, ID<j>, DIFF ID<i:j>
NOT IN <i:l,j> + ID<i>, ID<j>, IDLIST<l>, NOT IN <i:l>, DIFF ID <i:j>

DIFF IDLIST <i> + ID<i>
DIFF IDLIST <l,i> + DIFF IDLIST<l>, NOT IN<i:l>

Here, the symbols "A" through "Z" and "," enclosed in angle brackets are object alphabet symbols. The symbols "i", "j", and "l" are variable alphabet symbols.

Finally, the use of the double quote " will be used when the quote symbol ' occurs as an *object* alphabet symbol.

Deductive Rules:

The *derivable conclusions* of a production system are the conclusions that can be obtained from the productions by a finite number of applications of the following two rules.

Rule (1) A production P' can be obtained from a production P by substitution of an object string (possibly null) for *each* occurrence of a variable.

Rule (2) If each premise in a production is derivable, then the conclusion is derivable.

In the case of atomic productions, rule (2) states that its conclusion can be derived immediately. These rules can be applied to the previously given productions to derive the conclusions

ID<A>
IDLIST<A>
IDLIST<A,B,A>
NOT IN <A:B>
NOT IN <A:B,C>
DIFF IDLIST <A,B,C>

Interpretation:

A production system will be interpreted in the following way. A predicate will denote the name of a set. A term tuple of degree n following a predicate of a derived conclusion will be taken as an assertion that the n-tuple is one member of the named set. Productions will be viewed as rewriting rules for enumerating members of sets. In the previously given productions, the set named ID of identifiers contains the 26 letters of the English alphabet, the set "DIFF ID" contains all pairs of *different* identifiers, and the set DIFF IDLIST

contains all lists of identifiers such that each identifier in a list is different from other identifiers in the list.

2.2 Abbreviations to the Basic Notation

Using only the basic notation for production systems, a specification for a computer language often becomes lengthy or unnatural. It will be extremely useful to introduce several notational conventions to alleviate this difficulty. For brevity, each abbreviation will be specified mainly by example. Strictly speaking, each abbreviation should be formally specified so that an abbreviated production system and its unabbreviated counterpart define exactly the same sets.

Abbreviation 1: Factoring Common Predicate Names

Atomic productions defining the same predicate can be abbreviated by listing the term tuples in a single production. For example:

ID<A>
ID
ID<C>

may be abbreviated

ID<A>,,<C>

Similarly premises referring to the same predicate can be abbreviated.

For example:

NOT IN <i:j> ← ID<i>, ID<j>, DIFF ID<i:j>

may be abbreviated

NOT IN <i:j> ← ID<i>,<j>, DIFF ID<i:j>

Abbreviation 2: Factoring Common Variables and Predicates

It is somewhat more transparent to use one or two distinct variables throughout a production system to refer to members of a single predicate. If this convention is strictly followed for a number of predicates, then we may use the following abbreviation to "factor out" the premises referring to these predicates and variables. Let $v_1, v_2, \dots, v_n$ be variables and $P_1, P_2, \dots, P_n$ be predicates. Then $v_1, v_2, \dots, v_n$ can be listed beside the corresponding defined predicate $P_1, P_2, \dots$, or P_n and all premises of the form $P_1\langle v_1 \rangle, P_2\langle v_2 \rangle, \dots$, and $P_n\langle v_n \rangle$ can be deleted. For example, the productions

```
ID<A>,<B>,...,<Z>
NOT IN<i:j> + ID<i>, ID<j>, DIFF ID<i:j>
ID LIST<i> + ID<i>
ID LIST<l,i> + ID LIST<l>, ID<i>
```

can be abbreviated

```
i,j ID<A>,<B>,...,<Z>
    NOT IN<i:j> + DIFF ID<i:j>
    l ID LIST<i>,<l,i>
```

Abbreviation 3: Membership in Lists, Testing for Equality, and Arithmetic Predicates

Consider the predicate NOT IN defined above. This predicate names a set of pairs, where the first element of each pair is an identifier i , and the second element is a list of identifiers not containing i . For example the pairs

$\langle A:C,D,E \rangle$

$\langle X:A,B,B,Y \rangle$

are members of NOT IN, whereas the pairs

$\langle C:C,D,E \rangle$

$\langle B:A,B,B,Y \rangle$

are not members of NOT IN. Similarly, a predicate IN may be defined such that the first element is an identifier i and the second element is a list containing at least one occurrence of i . Furthermore, one may easily extend these definitions to include multi-character identifiers.

The use of these predicates is a frequent occurrence in our application to computer languages. For simplicity, the symbols " ϵ " and " $\notin$ " will be used in place of these predicates. For example,

NOT IN $\langle i:l \rangle$

NOT IN $\langle i: \text{DOMAIN} \langle \rho \rangle \rangle$

IN $\langle i:l \rangle$

IN $\langle \text{BASED}:\rho(i) \rangle$

will be abbreviated

$i \notin l$

$i \notin \text{DOMAIN} \langle \rho \rangle$

$i \in l$

$\text{BASED} \in \rho(i)$

Similarly, let p_1 and p_2 be terms denoting strings and let n_1 and n_2 be terms denoting numbers. The predicates EQ, NEQ, LT, LTE, GT, GTE may be defined such that

EQ $\langle s_1:s_2 \rangle$	iff s_1 denotes a string that is identical to s_2
NEQ $\langle s_1:s_2 \rangle$	iff s_1 denotes a string that is not identical to s_2
LT $\langle n_1:n_2 \rangle$	iff n_1 denotes a number that is less than n_2
LTE $\langle n_1:n_2 \rangle$	iff n_1 denotes a number that is less than or equal to n_2
GT $\langle n_1:n_2 \rangle$	iff n_1 denotes a number that is greater than n_2
GTE $\langle n_1:n_2 \rangle$	iff n_1 denotes a number that is greater than or equal to n_2

For convenience, atomic formulas of the above form will be abbreviated:

$s_1 \equiv s_2$	$s_1 \neq s_2$
$n_1 < n_2$	$n_1 \leq n_2$
$n_1 > n_2$	$n_1 \geq n_2$

For example, the production

RESULT TYPE $\langle t_2::t_2:ARITH \rangle \leftarrow EQ\langle t_2:ARITH \rangle, EQ\langle t_2:ARITH \rangle$

may be abbreviated

RESULT TYPE $\langle t_2::t_2:ARITH \rangle \leftarrow t_1 \equiv ARITH, t_2 \equiv ARITH$

Abbreviation 4: Direct Representation of Set Names

To avoid a profusion of variables representing members of predicates in a production system, it is convenient to replace some variables by their corresponding predicate name. Accordingly, the occurrence of an underlined predicate symbol $\underline{P}$ in a production will be used in place of the occurrence of the variable v in the conclusion and premise $P\langle v \rangle$.

For example, the productions

DCL UNIT<x> ← ELEMENT DCL<x>
DCL UNIT<x> ← STRUCTURE DCL<x>

may be abbreviated

DCL UNIT<ELEMENT DCL>
DCL UNIT<STRUCTURE DCL>

Abbreviation 5: Disjunction and the use of "|"

In many cases the members of a single predicate can be derived from multiple combinations of premises. In these cases, the disjunction symbol "|" is introduced. For example, the productions

- (a) DATA TYPE ATR <A>
DATA TYPE ATR <FIXED>
DATA TYPE ATR <FLOAT>
- (b) DCL UNIT <ELEMENT DCL>
DCL UNIT <STRUCTURE DCL>
- (c) UNIT <iu>
UNIT <du>
- (d) RESULT TYPE < $t_1 : + : t_2 : \text{ARITH}$ > ← $t_1 \equiv \text{ARITH}, t_2 \equiv \text{ARITH}$
RESULT TYPE < $t_1 : + : t_2 : \text{ARITH}$ > ← $t_1 \equiv \text{ARITH}, t_2 \equiv \text{STR}$
RESULT TYPE < $t_1 : + : t_2 : \text{ARITH}$ > ← $t_1 \equiv \text{STR}, t_2 \equiv \text{ARITH}$
RESULT TYPE < $t_1 : + : t_2 : \text{ARITH}$ > ← $t_1 \equiv \text{STR}, t_2 \equiv \text{STR}$

may be abbreviated

- (a) DATA TYPE ATR <A | FIXED | FLOAT>
- (b) DCL UNIT <ELEMENT DCL | STRUCTURE DCL>
- (c) UNIT <iu | du>
- (d) RESULT TYPE < $t_1::t_2$:ARITH> + ($t_1 \equiv$ ARITH | $t_1 \equiv$ STR), ($t_2 \equiv$ ARITH | $t_2 \equiv$ STR)

Using abbreviations 1, 2, and 4 above we can abbreviate the previously given productions for DIFF IDLIST as

i,j	ID	<A B ... Z>
l	IDLIST	<i l,i>
	DIFF ID	<A:B>, <A:C>, ..., <Z:Y>
	NOT IN	<i:j> + DIFF ID <i:j>
	NOT IN	<i:l,j> + DIFF ID <i:j>, NOT IN <i:l>
	DIFF IDLIST	<i>
	DIFF IDLIST	<l,i> + DIFF IDLIST <l>, NOT IN <i:l>

Abbreviation 6: Permutations

In a few cases we will wish to define a set as comprising permutations of elements from other sets. We will use the notation $t_1 \circ t_2 \circ \dots \circ t_n$ to denote any permutation of the terms t_1 through t_n . For example, the productions

EL ATR LIST <dta sca sta | dta sca sta | sca dta sta | sca sta dta |
sta dta sca | sta sca dta>

will be abbreviated

EL ATR LIST <dta o sca o sta>

Abbreviation 7: Notation for Functions

The notation for functions is motivated by the observation that besides thinking in terms of "inductive" or "generative" definitions, we often think of "algorithms" that can be used to "compute" results. The next notational convention reflects this predisposition.

(7a) Let $t_1, t_2, \dots, t_{n-1}$ be terms and t_n be a variable.

If $R\langle t_1: t_2: \dots : t_n \rangle$ is a premise occurring in a production P containing exactly one other occurrence c of t_n , then the premise can be deleted from the production if c is replaced by the string

$$\underline{R}\langle t_1: t_2: \dots : t_{n-1} \rangle$$

(7b) If $R\langle t_1: t_2: \dots : t_n \rangle$ is an atomic formula occurring in a production, then it may be alternately written

$$\underline{R}\langle t_1: t_2: \dots : t_{n-1} \rangle = t_n$$

(7c) In cases where a predicate R is used only as a "function," we may use a greek letter λ in place of the underlined predicate name $\underline{R}$.

For example, using (7a) and (7b) the productions

(a) IF STM<IF e THEN u > $\leftarrow$ COMPUTED TYPE < $e:p:t$ >, CONVERTIBLE < $t:STR$ >

(b) RESULT TYPE< $t_1:+:t_2:ARITH$ > $\leftarrow$ ($t_1 \equiv ARITH | t_1 \equiv STR$), ($t_2 \equiv ARITH | t_2 \equiv STR$)

(c) COMPUTED EXTERNAL ENV< $iu:\Lambda$ >
COMPUTED EXTERNAL ENV< $n \text{ i dta} : [i \rightarrow x]$ > $\leftarrow$ COMPUTED EXTERNAL ENV< $dta:x$ >

(d) COMPUTED TYPE< $i:p:x$ > $\leftarrow$ TYPE ATR < $p(i):x$ >

may be abbreviated

- (a') IF STM<IF e THEN iu> + t = COMPUTED TYPE<e:p>, CONVERTIBLE<t:STR>
- (b') RESULT TYPE<t₁:+:t₂> = ARITH + (t₁≡ARITH|t₁≡STR), (t₂≡ARITH|t₂≡STR)
- (c') COMPUTED EXTERNAL ENV<iu> = Λ
COMPUTED EXTERNAL ENV<n i dta> = [i → COMPUTED EXTERNAL ENV<dta>]
- (d') COMPUTED TYPE<i:p> = TYPE ATR<ρ(i)>

Using (7c) with the replacement of "COMPUTED TYPE" by " ϕ_t " and "COMPUTED EXTERNAL ENV" by " β_e ", productions (a'), (c'), and (d') may be further abbreviated

- (a'') IF STM<IF e THEN iu> + t = ϕ_t <l:p>, CONVERTIBLE<t:STR>
- (c'') β_e <iu> = Λ
 β_e <n i dta> = [i → β_e <dta>]
- (d'') ϕ_t <i:p> = TYPE ATR<ρ(i)>

Abbreviation 8: Sequences and the use of "..."

Consider the following recursive definition of IDLIST

$$l \text{ IDLIST} \langle i \mid l, i \rangle$$

Conceptually, one may alternatively view a member of IDLIST as a sequence of one or more identifiers each separated by a comma. Accordingly, we shall write the definition of IDLIST as

$$l \text{ IDLIST} \langle i_1, i_2, \dots, i_n \rangle$$

In the sequel, the "..." notation will be used in several contexts. In some cases the corresponding recursive definition may be difficult to write. The use of the sequence notation is extremely useful.

We shall ask some indulgence by the reader to accept that a strict production system definition can be written from an abbreviated production system containing the notation for sequences.

3 THE COMPLETE SYNTAX OF PL.1

The complete syntax of PL.1 is indeed complex, and is given in Table C. To make the task lighter, we shall introduce a few concepts relevant to block structured languages. The most important of these is the concept of "syntactic environment."

3.1 A Basic Overview

Conceptually we shall view the concept of a syntactic environment ρ as a function mapping an identifier into its derived attributes. For example, consider the following PL.1 program

```
PROG: PROCEDURE;  
      DECLARE X FIXED;  
      DECLARE Y FLOAT;  
      DECLARE Q POINTER EXTERNAL;  
      DECLARE Z CHAR(5) BASED(Q);  
      X = 1;  
      Y = X + 1;  
      ALLOCATE Z SET(Q);  
      Q → Z = 'SNOW';  
END PROG;
```

The syntactic environment ρ for this program is defined as follows:

```
[ X → ARITH *  
  Y → ARITH *  
  Z → STR, BASED *  
  Q → PTR, EXTERNAL ]
```

In order that the components in a PL.1 expression or statement be compatible with their use, the types of each component must be determined.

(1) Given an identifier i and an environment ρ , we shall define a function APPLY, such that APPLY $\langle\rho:i\rangle$ equals the derived attribute list associated with i in ρ . For example, using ρ above

APPLY $\langle\rho:Z\rangle$ = STR, BASED

For brevity, we shall abbreviate

APPLY< ρ : i >

as

$\rho(i)$

(2) Given a derived attribute list ℓ , we shall define a function TYPE ATR that maps ℓ into the problem data type attribute given in ℓ .

For example

TYPE ATR<STR,BASED> = STR

(3) Given two problem data types t_1 , t_2 and an operator q , we shall define a function RESULT TYPE that maps these three arguments into the result type computed by applying the operator q to two arguments of type t_1 and t_2 , e.g.:

RESULT TYPE<ARITH : q : ARITH> = ARITH $\leftarrow$ PLUS OP< q >
RESULT TYPE<ARITH : q : ARITH> = STR $\leftarrow$ EQUALITY OP< q >

(4) Given an expression e and a syntactic environment ρ , we shall define a function ϕ_t that computes the derived type of e in ρ .

For example, given the previous environment ρ

ϕ_t < $X:\rho$ > = TYPE ATR < $\rho(x)$ > = TYPE ATR <ARITH> = ARITH

ϕ_t < $Y:\rho$ > = TYPE ATR < $\rho(x)$ > = TYPE ATR <ARITH> = ARITH

ϕ_t < $1:\rho$ > = ARITH

ϕ_t < $(X+1):\rho$ > = RESULT TYPE < ϕ_t < $X:\rho$ >::+: ϕ_t < $1:\rho$ >>
= RESULT TYPE <ARITH::+:ARITH>
= ARITH

ϕ_t <'SNOW': ρ > = STR

ϕ_t < $Z:\rho$ > = TYPE ATR < $\rho(Z)$ >
= TYPE ATR <STR, BASED>
= STR

(5) Finally, we shall define a predicate CONVERTIBLE comprising all pairs of derived types t_1 and t_2 such that t_2 can be converted into t_1 according to the rules of PL/1. For example the pairs

<STR : STR>

<ARITH : STR>

are members of CONVERTIBLE whereas

<STR : PTR>

<LABEL : ARITH>

are not.

3.2 Some simple examples

Consider the following proposed program

```
P: PROCEDURE;  
  DECLARE A POINTER;  
  DECLARE A FIXED;  
  A = 1;  
END P;
```

The syntactic environment for this program is

[A → PTR * A → FIXED]

This program is ruled out in the production system of PL.1 by the premise

DIFF IDLIST <DOMAIN< ρ >>

given in the productions for PROGRAM. Here the domain of ρ is the list "A,A", which is not a member of the predicate DIFF IDLIST.

Consider next the proposed program

```
P: PROCEDURE;  
  DECLARE X FLOAT;  
  X = 0.0;  
L: X = X+1;  
  GO TO M;  
END P;
```

The syntactic environment for this program is

```
[ X → ARITH*
  L → LABEL*
  M → ARITH ]
```

since M is implicitly declared to be arithmetic since it is not explicitly or contextually declared. This program is ruled out by the production

GOTO STM<GOTO i;> ← LABEL ∈ ρ(i)

since ρ(M) = ARITH and

LABEL ∉ ARITH

Consider the following proposed programs

```
P: PROCEDURE;
  DECLARE X FIXED BASED(A);
  ALLOCATE X;
  A → X = 1;
  B = A;
END P;
```

```
Q: PROCEDURE;
  DECLARE X FIXED BASED(A);
  DECLARE B POINTER;
  ALLOCATE X;
  A → X = 1;
  B = A;
END Q;
```

The environments ρ_P and ρ_Q for P and Q are

ρ_P ≡ [X → FIXED, BASED *
A → PTR *
B → ARITH]

ρ_Q ≡ [X → FIXED, BASED *
A → PTR *
B → PTR]

The production for assignment statements is

ASGT STM <r=e;> ← t₁=φ_t<r:ρ>, t₂=φ_t<e:ρ>, CONVERTIBLE<t₁:t₂

The instances of the corresponding premises for the statement "B = A;"

P and Q are

t₁ = φ_t<B:ρ_P> = ARITH

t₁ = φ_t<B:ρ_Q> = PTR

t₂ = φ_t<A:ρ_P> = PTR

t₂ = φ_t<A:ρ_Q> = PTR

CONVERTIBLE <ARITH:PTR>

CONVERTIBLE <PTR:PTR>

Since the pair <ARITH:PTR> is not a member of CONVERTIBLE whereas the pair <PTR:PTR> is, program P is illegal whereas program Q is legal.

3.3 Computation of Environments and Block Structure

Given environments ρ and ρ' and a list ℓ of identifiers, we may define the functions DOMAIN, DELETE, AND PLUS such that

- (1) DOMAIN $\langle\rho\rangle$ = the list of identifiers in the domain of ρ
- (2) DELETE $\langle\rho:\ell\rangle$ = the environment computed from ρ by deleting the identifiers in ℓ from the domain of ρ .
- (3) PLUS $\langle\rho:\rho'\rangle$ = the environment computed by "adding" ρ to ρ'

We shall abbreviate (2) and (3) above as

$$(2') \quad \rho - \ell$$

$$(3') \quad \rho + \rho'$$

PL.1 allows three varieties of declarations: explicit, contextual, and implicit. Accordingly, three functions β_e , β_c , and β_i are defined that map a PL.1 unit sequence $u \equiv u_1 u_2 \dots u_n$ into environments corresponding to the identifiers that are candidates for explicit, contextual, or implicit declarations. Since explicit declarations override candidates for contextual declarations, and contextual declarations override candidates for implicit declarations, the resulting environment components ρ_e , ρ_c , and ρ_i for explicit, contextual, and implicit declarations are defined as

$$\rho_e = \beta_e \langle u \rangle$$

$$\rho_c = \beta_c \langle u \rangle - \text{DOMAIN} \langle \rho_e \rangle$$

$$\rho_i = \beta_i \langle u \rangle - \text{DOMAIN} \langle \rho_i + \rho_c \rangle$$

For example, consider the program

```
P:  PROCEDURE;
    DECLARE X FLOAT BASED(Q);
    X = 1;
    IF X>1 THEN ALLOCATE X SET(R);
    Y = X + 1;
    X = Y;
END P;
```

Here the unit sequence u in the program contains one DECLARE statement and five imperative statements. The identifier X is declared explicitly, hence

$$\begin{aligned}\rho_e &\equiv \beta_e \langle u \rangle \\ &= [X \rightarrow \text{ARITH, BASED}]\end{aligned}$$

Two contextual declarations appear in the BASED attribute declaration of Q and the SET option declaration of R . Hence

$$\begin{aligned}\rho_c &\equiv \beta_c \langle u \rangle - \underline{\text{DOMAIN}} \langle \rho_e \rangle \\ &= [Q \rightarrow \text{PTR} * R \rightarrow \text{PTR}] - X \\ &= [Q \rightarrow \text{PTR} * R \rightarrow \text{PTR}]\end{aligned}$$

One remaining identifier Y , is declared implicitly. Hence

$$\begin{aligned}\rho_i &\equiv \beta_i \langle u \rangle - \underline{\text{DOMAIN}} \langle \rho_e + \rho_c \rangle \\ &= [X \rightarrow \text{ARITH} * Q \rightarrow \text{ARITH} * R \rightarrow \text{ARITH} * Y \rightarrow \text{ARITH}] - X, Q, R \\ &= [Y \rightarrow \text{ARITH}]\end{aligned}$$

Hence the total environment ρ is given as

$$\begin{aligned}\rho &= \rho_e + \rho_c + \rho_i \\ &= [X \rightarrow \text{ARITH, BASED} * \\ &\quad Q \rightarrow \text{PTR} * \\ &\quad R \rightarrow \text{PTR} * \\ &\quad Y \rightarrow \text{ARITH}]\end{aligned}$$

Using the total environment ρ , each of the imperative statements in P can be derived as legal statements.

Two basic requirements follow from adding block structure:

(a) an identifier declared local to an inner block has a separate existence within its own scope (b) an identifier declared global to an inner block and not declared locally has a scope that includes the inner block. To reflect these requirements, the environment for a block

BEGIN; $u_1 u_2 \dots u_n$ END;

nested within an environment ρ is defined as

$$\rho'' = \text{OVERRIDE} \langle \rho_e : \rho \rangle$$

where $\rho_e = \rho_e + \rho'$

$$(a) \rho_e = \beta_e \langle u_1 u_2 \dots u_n \rangle$$

$$(b) \rho' = \rho - \text{DOMAIN} \langle \rho_e \rangle$$

All units within the block are then tested for legality using ρ'' .

3.4 Structures

In order to guarantee the appropriate requirements on the use of structure operands, the type-checking functions and predicates must include the use of structures. Consider the following structure declarations:

```
DECLARE 1 X,
  2 A FIXED,
  2 B,
    3 D POINTER,
    3 E POINTER,
  2 C FLOAT;
```

```
DECLARE 1 Y,
  2 A FIXED,
  2 B,
    3 D CHAR(5),
    3 E CHAR(5),
  2 C;
```

```
DECLARE 1 Z,
  2 A FIXED,
  2 B,
    3 D CHAR(6),
    3 E CHAR(6),
  2 C FIXED;
```

The derived data types t_X , t_Y , and t_Z given for X, Y, and Z in the syntactic environment ρ are

$$t_X = [A \rightarrow \text{ARITH}^* \\ B \rightarrow [D \rightarrow \text{PTR}^* \\ E \rightarrow \text{PTR}]^* \\ C \rightarrow \text{ARITH}]$$

$$t_Y = [A \rightarrow \text{ARITH}^* \\ B \rightarrow [D \rightarrow \text{STR}^* \\ E \rightarrow \text{STR}]^* \\ C \rightarrow \text{ARITH}]$$

$$t = [A \rightarrow \text{ARITH}^* \\ B \rightarrow [D \rightarrow \text{STR}^* \\ E \rightarrow \text{STR}]^* \\ C \rightarrow \text{ARITH}]$$

Consider also the statement

$$Y = Y + Z;$$

and the production

$$\text{ASGT STM } \langle r = e; \rangle \leftarrow t_1 = \phi_t \langle r : \rho \rangle, \quad t_2 = \phi_1 \langle e : \rho \rangle, \quad \text{CONVERTIBLE} \langle t_1 : t_2 \rangle$$

For the above assignment statement

$$t_1 = \phi_t \langle Y : \rho \rangle = t_Y$$

$$\begin{aligned} t_1 &= \phi_t \langle Y + Z : \rho \rangle = \text{RESULT TYPE} \langle \phi_t \langle Y : \rho \rangle : + : \phi_t \langle Z : \rho \rangle \rangle \\ &= \text{RESULT TYPE} \langle t_Y : + : t_Z \rangle \\ &= [A \rightarrow \text{ARITH}^* \\ &\quad B \rightarrow [D \rightarrow \text{ARITH}^* \\ &\quad \quad E \rightarrow \text{ARITH}]^* \\ &\quad C \rightarrow \text{ARITH}] \end{aligned}$$

since t_Y and t_Z have identical structuring and the types of each pair of corresponding base elements are compatible with the addition operation. Furthermore t_2 is convertible to t_1 since

$$\text{CONVERTIBLE} \langle t_1 : t_2 \rangle \leftarrow \text{RESULT TYPE} \langle t_1 := t_2 \rangle = \text{STR}$$

and using the productions for RESULT TYPE, the premise

$$\text{RESULT TYPE} \langle t_1 := t_2 \rangle = \text{STR}$$

can be derived.

On the other hand, consider the assignment statement

$$Y = X + Z;$$

Here we have an error in the attempt to add the pointers in X to the character strings in Z. This statement cannot be derived as a legal statement since

$$\begin{aligned} t_2 = \phi_t \langle (X+Z):p \rangle &= \text{RESULT TYPE} \langle \phi_t \langle X:p \rangle :+ : \phi_t \langle Z:p \rangle \rangle \\ &= \text{RESULT TYPE} \langle t_x :+ : t_z \rangle \\ &= \text{undefined} \end{aligned}$$

The root of this issue lies in the production

$\text{RESULT TYPE} \langle t_1 :q : t_2 \rangle = \text{ARITH} + \text{PLUS OP} \langle q \rangle, (t_1 \equiv \text{ARITH} | t_1 \equiv \text{STR}), (t_2 \equiv \text{ARITH} | t_2 \equiv \text{STR})$
applied to the structure elements. This production cannot be used to derive the conclusion.

$\text{RESULT TYPE} \langle \text{PTR} :+ : \text{ARITH} \rangle = \text{ARITH}$

since $t_1 \equiv \text{PTR}$, i.e. $t_1 \not\equiv \text{ARITH}$ and $t_1 \not\equiv \text{STR}$

3.5 The Complete Production System of PL.1

A definition of the complete syntax of PL.1 is given in Table C. The basic concepts underlying the production system have been outlined in the previous sections. Hopefully, the reader who wishes any detail of information on PL.1 may refer to Table C and find the appropriate information.

One additional abbreviation is used in Table C.

The predicates for PL.1 syntactic units

from UNIT down to REFERENCE are of degree 2, with the syntactic environment ρ for each unit given as a second term. For brevity, ρ is omitted in many of the corresponding atomic formulas. For example, the productions

GOTO STM<GOTO i; : ρ > + LABEL $\in \rho(i)$

STRUC ATR LIST<sca o sta: ρ > + SCOPE ATR<sca: ρ >, STORAGE ATR<sta: ρ >,
(sca $\neq$ EXTERNAL | sta $\equiv$ STATIC)

are abbreviated

GOTO STM<GOTO i;> + LABEL $\in \rho(i)$

STRUC ATR LIST<sca o sta> + SCOPE ATR<sca>, STORAGE ATR<sta>,
(sca $\neq$ EXTERNAL | sta $\equiv$ STATIC)

Several comments are in order at this point.

The production system of PL.1 is a complete definition of syntax of a significantly difficult practical language subset. The major thrust of this research has been the attempt to tailor a given mathematical formalism to meet the practical needs of language definition. Subsequent sections will now enlarge upon the problem of language definition to include the definition of abstract syntax and the mapping of lambda-calculus expressions into normal form.

4. APPLICATION TO SPECIFY ABSTRACT SYNTAX

A definition of a class of "abstract" objects is a definition for which no representation of objects is specified. Following the lines of McCarthy [11] and the IBM Vienna Laboratory [9, 10], a definition of a class of abstract object must provide definitions of (a) constructor functions for constructing the variety of objects in the class, (b) predicates for testing whether an object is of a particular variety, and (c) selector functions that when applied to an object of a particular variety yield a particular component of the object.

Clearly, to communicate any definition one must use some symbols. Accordingly, primitive objects will be denoted by meta-linguistic variables, and composite objects containing n components will be denoted by the conventional notation for n -tuples, i.e.,

$$(a_1, a_2, \dots, a_n)$$

or trees, i.e.,

where $a_1, a_2, \dots$ and a_n denote objects. A definition will be considered "abstract" in that all objects will be represented using only

meta-linguistic variables and the notation for n-tuples (or equivalently, trees).

4.1 Definition of Abstract Syntax via Production Systems

The notion of a definition of a class of abstract objects will be couched within production systems in the following way:

(a) Productions specifying primitive objects will be omitted in a production system specification. We shall say only what properties the primitive objects must possess.

(b) Objects constructed from the primitive objects will be specified as n-tuples of the form

$$(a_1, a_2, \dots, a_n)$$

where the a_i , $1 \leq i \leq n$, are variables denoting primitive objects or other constructed objects. The productions defining these n-tuples will be taken as implicit definition of the *constructor functions* for objects in the defined class.

(c) Predicate names of a productions system will be interpreted as *predicates* over the class of abstract objects in that $P\langle b \rangle$ will be interpreted as *true* if b can be derived as a member of the set named P , and *false* otherwise.

(d) The *selector functions* of an abstract definition will be specified as follows. Let $(a_1, a_2, \dots, a_n)$ denote an object in a class C and $S_1, S_2, \dots$ and $S_n, S_i \neq S_j$ for $i \neq j$, be the names of the selector functions over objects in C . Then

$$S_i(a_1, a_2, \dots, a_n) = a_i$$

if and only if the conclusion

$$S_i\langle a_1, a_2, \dots, a_n \rangle = a_i$$

is derivable from the production system.

4.2 The Abstract Syntax of PL.1

In a definition of abstract syntax, we first assume that there are certain classes of primitive objects with certain primitive properties. For PL.1 these comprise the class of identifiers, the class of natural numbers, the class of operators, and the classes of special PL.1 symbols, e.g., the assignment symbol, the goto symbol, the fixed attribute symbol, and the based attribute symbol. The identifiers, numbers, and operators have their conventional properties. The *special* PL.1 symbols have only the property that they must be *different* in order to yield an unambiguous interpretation of a given n-tuple.

For example, a FREE statement with a non-null pointer may be defined as a triple of the form

$$(x,i,j)$$

where x is an ALLOCATE symbol, i is a based variable, and j is a pointer. The class of FREE statements may be given the predicate name "FREE STM" and may be defined by the production

$$(1) \text{ FREE STM}(x,i,j) \rightarrow \text{ALLOC SYM}(x), \text{ID}(i), (j)$$

For each composite object we must define the selector functions for extracting the components of the object. In particular, for a FREE statement we wish to select its marking symbol, its based variable, and its pointer reference. These three functions may be given the names S-SYM, S-BASED-VAR, S-PTR-REF and defined as follows:

- (1a) $\underline{S-SYM}(x,i,j) = x \quad + \quad \text{ALLOC SYM}(x), \text{ID}(i), (j)$
 (1b) $\underline{S-BASED-VAR}(x,i,j) = i \quad + \quad \text{ALLOC SYM}(x), \text{ID}(i), (j)$
 (1c) $\underline{S-PTR-REF}(x,i,j) = j \quad + \quad \text{ALLOC SYM}(x), \text{ID}(i), (j)$

The definition of the production system predicates and selector functions may be considerably shortened by the following abbreviation:

Let P be a sequence of premises, C be the predicate name of a class of n -tuples, $\underline{S_1}, \underline{S_2}, \dots$, and $\underline{S_n}$ be the names of n selector functions over the class of objects, and $t_1, t_2, \dots, t_n$ be terms. Productions of the form

$$\begin{array}{rcl} C(t_1, t_2, \dots, t_n) & + & P \\ \underline{S_1}(t_1, t_2, \dots, t_n) = t_1 & + & P \\ \underline{S_2}(t_1, t_2, \dots, t_n) = t_2 & + & P \\ \vdots & & \vdots \\ \underline{S_n}(t_1, t_2, \dots, t_n) = t_n & + & P \end{array}$$

may be combined into the single production

$$C(\underline{S_1} t_1, \underline{S_2} t_2, \dots, \underline{S_n} t_n) + P$$

Thus the productions 3, 1 through 1c above can be abbreviated

$$(1') \quad \text{FREE STM}(\underline{S-SYM} x, \underline{S-BASED-VAR} i, \underline{S-PTR-REF} j) + \text{ALLOC SYM}(x), \text{ID}(i), (j)$$

The abbreviated notation is more than a shorthand notation in that the abbreviated productions may be viewed as a *simultaneous* definition of the *constructors*, *predicates*, and *selectors* in the definition of the class of abstract objects.

The context-sensitive requirements on the syntax of a language must be specified in the definition of abstract syntax as well as concrete syntax. Consider the requirement on PL.1 that there be no multiple declarations. In terms of the notion of a syntactic environment ρ , the list of identifiers comprising the domain of ρ must each be different. In terms of abstract syntax, the list of identifiers $i_1, i_2, \dots, i_n$ in the domain of ρ must each be different.

Next consider the productions

NOT IN<i:j>	+ DIFF ID<i:j>
NOT IN<i:j,l>	+ DIFF ID<i:j>, NOT IN<i:l>
DIFF IDLIST<i>	
DIFF IDLIST<i,l>	+ NOT IN<i:l>, DIFF IDLIST<l>

As mentioned earlier, the predicate "ID" specifying the class of identifiers is left unspecified in the definition of the abstract syntax. So too, the predicate "DIFF ID" specifying the set of all ordered pairs for which the first element is an identifier and the second element is a *different* identifier is left unspecified. The property of identifiers that we are able to say if two identifiers are different is a *primitive* property of identifiers, and accordingly the predicate "DIFF ID" is left unspecified in a definition of abstract syntax. In terms of the unspecified predicates "ID" and "DIFF ID", the predicate "DIFF IDLIST" defines a set where each element is a list of identifiers such that each identifier in the list is different.

The complete definition of the abstract syntax of PL.1 is given in Table A. There the intuitive role of the predicates parallel those given for the concrete syntax, except that no concrete representation of programs is specified.

The definition of the abstract syntax of PL.1 is considerably simpler than the definition of concrete syntax. The primary reason for this result is that many of the peculiarities of PL/I disappear at the abstract level. For example, in the abstract syntax each identifier must be explicitly declared with all relevant attributes. This eliminates the need for functions like β_e and β_i needed to compute the environment components for contextual and implicit declarations.

5. THE TRANSLATION OF LAMDA-CALCULUS EXPRESSIONS INTO NORMAL FORMS.

As a last example, we consider the mapping of λ -calculus expressions into normal form. In this mapping the full power of production systems must be employed, since the set of pairs

$\langle \lambda\text{-calculus exp} : \text{normal form} \rangle$

is a recursively enumerable set that is not recursive. That is, given a pair $\langle e_1 : e_2 \rangle$ of well-formed lambda expressions, it is not decidable whether e_2 is a normal form of e_1 .

The primary value of the productions for this mapping lies in the definition of the λ -calculus substitution rule ϕ as described by Curry and Feys [2].

The λ -calculus substitution rule has a counterpart in the definition of the call-by-name rule of ALGOL 60. Let i be a formal parameter in ALGOL 60, f be the corresponding actual parameter, e be the λ -expression representing the declared procedure. Let e' be the λ -expression representing the resulting ALGOL 60 code when the procedure e is called with a formal parameter i and actual parameter f . Then the value e' such that

$$\phi \langle i : f : e \rangle = e'$$

represents the code derived from e when i is "substituted" for f .

In particular, consider the productions

$$\begin{aligned} \phi \langle i : f : i \rangle &= f \\ \phi \langle i : f : j \rangle &= j \quad + \text{ DIFF ID} \langle i : j \rangle \end{aligned}$$

If e is an identifier i that is identical to the actual parameter i , then e' is f . If e is the identifier j that is different from i , then e' is j . For example, consider the procedure declaration

```

procedure F(X);
  X := X + Y;
end F

```

and call

```

F(B);

```

The execution of F(B) results in the execution of

```

B := B + Y

```

That is, $e = X$ is replaced by B, but $e = Y$ is left alone.

Next consider the production

$$\phi\langle i : f : (\lambda j.e) \rangle = (\lambda k.\phi\langle i : f : e' \rangle)$$

$$\begin{array}{ll} \text{DIFF ID}\langle i:j \rangle, & j \in \text{FREE IDS}\langle f \rangle, \\ \text{DIFF ID}\langle i:k \rangle, & k \notin \text{FREE IDS}\langle f \rangle, k \notin \text{FREE IDS}\langle e \rangle, \\ e' = \phi\langle j:k:e \rangle & \end{array}$$

the procedure declaration

```

procedure G(Y);
  B := Y + 1;
  begin ; integer X;
    X := 1;
    B := X + Y;
  end;
end G;

```

and the procedure call

```

G(X + A);

```

We wish to consider the parameter replacement of $e' = Y$ by $f = X+A$ in the nested begin-end block. Here the local variable $j = X$ is contained in the list X,A of actual parameters of $f = X+A$. Let $k = Q$ be an identifier, which is different from $i = X$ and is not contained in the list of free identifiers for f or the begin-end block.

Replacing X in the *begin-end* block by Q, we obtain

```
e' = begin; integer Q;  
      Q := 1;  
      B := Q + Y;  
      end;
```

Having thus changed the local variable, the replacement of i by f in the body of the revised begin-end block may be performed without considering a possible clash of identifiers.

6. DISCUSSION

Production systems have placed under a single framework the complete definition of the syntax and translation of a programmed language. Not once was it necessary to step outside the formalism. While the theoretical capability of production systems to define recursively enumerable sets guarantees us that the formalism is sufficiently powerful to define syntax and translation, the overwhelming task of this research was to tailor the formalism to practical languages. Accordingly, the notation, abbreviations, and conceptual view of production systems have undergone several stages of evolution.

Besides simplicity, such attendant qualities like naturalness, perspicuity, and communicativeness have been accorded due allowance. Necessarily, the author has used his personal discretion in weighing these qualities. It is inevitable and desirable that further research will refine the optimal balance of these qualities. Admittedly, there exists no known metrics for measuring these qualities precisely. They are subject to a latitude of interpretations. This fact should not be surprising. Indeed, almost every computer language has at least the theoretical capability of defining any computable algorithm. Why so many computer languages? It is more natural or more concise to define an algorithm in one language than another.

One theoretical difficulty with production systems remains to be resolved: the decidability of the class of strings specified by a production system. A production system specifying syntax defines a class of legal programs, but does not formally define the class of

strings that are illegal. A string is considered illegal only if the reader of a production system is convinced that the string cannot be derived as legal program. While in the production systems given here the classes of illegal strings are quite apparent, it would certainly be desirable in many cases to find some restriction on production systems to limit their definition to decidable sets.

Production systems can be used to specify definitions and string transformations much different from those given here. Outside of the example, given here, and a few others that the author has attempted, little experience other than the definition of syntax and translation with production systems has been obtained. Whether production systems can be fruitfully applied to more general areas of formal definition is a subject that remains to be investigated.

ACKNOWLEDGEMENT

To Edward Glaser, Dana Scott, John Wozencraft, and Michael Flynn, whose mature views of computation have significantly influenced the author.

And to Christopher Wadsworth, Calvin Mooers, Robert Graham, John Donovan, Joseph Davidson, and Lee Hoevel, who devoted considerable thought to the issues herein.

REFERENCES

1. Church, Alonzo
The Calculi of Lambda-Conversion
Annals of Mathematical Studies, Number 6, Princeton
University Press, Princeton, New Jersey 1941.
2. Curry, Haskell B., and Feys, Robert
Combinatory Logic
Volume I, North-Holland Publishing Company, Amsterdam, 1958.
3. Donovan, John J.
Investigations in Simulation and Simulation Languages
Ph.D. dissertation, Yale University, New Haven, Connecticut, 1966.
4. Donovan, John J., and Ledgard, Henry F.
A Formal System for the Specification of the Syntax and
Translation of Computer Languages
AFIPS, *Proceedings of the 1967 Fall Joint Computer Conference*,
Volume 31, Thompson Books, Washington, D.C., 1967.
5. Kleene, Stephen C.
Lambda-Definability and Recursiveness
Duke Mathematical Journal, Volume 2, pp. 340-353, 1936.
6. Landin, Peter J.
A Correspondence Between ALGOL 60 and Church's Lambda-Notation
Communications of the ACM, Volume 8, Number 2 & 3, February 1965.
7. Ledgard, Henry F.
*A Formal System for Defining the Syntax and Semantics of
Computer Languages.*
MAC-TR-60 (Ph.D. dissertation) Project MAC, M.I.T., Cambridge, Mass. 1969.
8. Ledgard, Henry F.
10 Mini-languages: A study of Topical Issues in Programming Languages
Computing Surveys, Volume 3, Number 3, September 1971, pp. 115-146.
9. Lucas, P., Lauer, P., and Stigleitner, H.
Method and Notation for the Formal Definition of Programming Languages
IBM Technical Report 25.087, IBM Laboratory, Vienna 1968.
10. Lucas, P., and Walk, K.
On the Formal Description of PL/I
Annual Review of Automatic Programming Volume 6, Number 3, Pergamon Press 1969
11. McCarthy, John M.
A formal Description of a Subset of ALGOL 60
Formal Language Description Languages for Computer Programming
(T.B. Steel - editor) North Holland Publishing Company, Amsterdam 1966.

12. More, Trenchard
Relations Between Simplicational Calculi
Ph.D. dissertation, MIT, Cambridge, Massachusetts, 1962.
13. Naur, Peter (Editor)
Revised Report on the Algorithmic Language ALGOL 60.
Communications of the ACM, Volume 6, Number 1, pp.1-23, 1963.
14. Post, Emil L.
Formal Reductions of the General Combinatorial Decision Problems
American Journal of Mathematics, Volume 65, pp.197-215, 1943.
15. Smullyan, Raymond M.
Theory of Formal Systems
Annals of Mathematical Studies, Number 47, Princeton University
Press, Princeton, New Jersey, 1961.
16. Turing, A. M.
On Computable Numbers with an Application to the Entscheidungsproblem
Proceedings of the London Mathematical Society,
Volume 42, pp.230-265, 1936.
17. Turing, A. M.
Computability and Lambda-Definability
Journal of Symbolic Logic, Volume 4, pp.153-160, 1937.
18. -----
PL/I (F) Language Reference Manual, GC28-8201-3
International Business Machines Corporation, New York, 1970.

Table C: COMPLETE SYNTAX OF PL.1

(C.1) Programs

p	PROGRAM	$\langle i: \text{PROCEDURE}; u_1 u_2 \dots u_n \text{ END } i; \rangle$ $+ u \equiv u_1 u_2 \dots u_n$ $\rho_e = \beta_e \langle u \rangle$ $\rho_c = \beta_c \langle u \rangle - \text{DOMAIN} \langle \rho_e \rangle,$ $\rho_i = \beta_i \langle u \rangle - \text{DOMAIN} \langle \rho_e + \rho_c \rangle,$ $\rho = \rho_e + \rho_c + \rho_i$ $\text{if } \text{DOMAIN} \langle \rho \rangle,$ $\text{DIFF IDLIST} \langle \text{DOMAIN} \langle \rho \rangle \rangle,$ $\text{UNIT} \langle u_1: \rho \rangle, \langle u_2: \rho \rangle, \dots, \langle u_n: \rho \rangle$
u	UNIT	$\langle \text{IMPER UNIT} \mid \text{DCL UNIT} \mid i': 'u' \rangle$
iu	IMPER UNIT	$\langle \text{ASGT STM} \mid \text{GOTO STM} \mid \text{IF STM} \mid \text{ALLOCATE STM} \mid \text{FREE STM} \mid \text{RETURN STM} \mid \text{BLOCK} \rangle$
du	DCL UNIT	$\langle \text{ELEMENT DCL} \mid \text{STRUCTURE DCL} \rangle$

(C.2) Imperative Units

	ASGT STM	$\langle r = e; \rangle$	$+ t_1 = \phi_t \langle r: \rho \rangle, t_2 = \phi_r \langle e: \rho \rangle, \text{ CONVERTIBLE} \langle t_1: t_2 \rangle$
	GOTO STM	$\langle \text{GOTO } i; \rangle$	$+ \text{ LABEL } ep(i)$
	IF STM	$\langle \text{IF } e \text{ THEN } iu \rangle$	$+ t = \phi_t \langle e: \rho \rangle, \text{ CONVERTIBLE} \langle t: \text{STR} \rangle$
	ALLOCATE STM	$\langle \text{ALLOCATE } i; \rangle$	$+ \text{ BASED } ep(i)$
	ALLOCATE STM	$\langle \text{ALLOCATE } i \text{ SET}(j); \rangle$	$+ \text{ BASED } ep(i), \text{ PTR } ep(j)$
	FREE STM	$\langle \text{FREE } i; \rangle$	$+ \text{ BASED } ep(i)$
	FREE STM	$\langle \text{FREE } j + i; \rangle$	$+ \text{ BASED } ep(i), \text{ BASED } ep(j), \text{ PTR } ep(j)$
	RETURN STM	$\langle \text{RETURN}; \rangle$	
	BLOCK	$\langle \text{BEGIN}; u_1 u_2 \dots u_n \text{ END}; \rangle$ $+ \rho_e = \beta_e \langle u_1 u_2 \dots u_n \rangle$ $\rho' = \text{OVERRIDE} \langle \rho_e: \rho \rangle$ $\text{DIFF IDLIST} \langle \text{DOMAIN} \langle \rho_e \rangle \rangle$ $\text{UNIT} \langle u_1: \rho' \rangle, \langle u_2: \rho' \rangle, \dots, \langle u_n: \rho' \rangle$	

(C.3) Declarations

	ELEMENT DCL	$\langle \text{DECLARE } i \text{ eal}; \rangle$	
	STRUCTURE DCL	$\langle \text{DECLARE } i \text{ sal, msl}; \rangle$	
eal	EL ATR LIST	$\langle \text{dta } o \text{ sca } o \text{ sta} \rangle$	$+ (sca \neq \text{EXTERNAL} \mid sta = \text{STATIC})$
sal	STRUC ATR LIST	$\langle \text{sca } o \text{ sta} \rangle$	$+ (sca \neq \text{EXTERNAL} \mid sta = \text{STATIC})$
ms	MINOR STRUC	$\langle n \text{ i succ} \rangle$	
msl	MINOR STRUC LIST	$\langle n_1 i_1 \text{ succ}_1, n_2 i_2 \text{ succ}_2, \dots, n_k i_k \text{ succ}_k \rangle$ $+ n_1 \geq n_2, n_2 \geq n_3, \dots, n_{k-1} \geq n_k, \text{ DIFF IDLIST} \langle i_1, i_2, \dots, i_k \rangle$	
succ	STRUC SUCC	$\langle \text{dta} \rangle$	
	STRUC SUCC	$\langle n \text{ i msl} \rangle + n \text{ LEVEL NUM} \langle \text{msl} \rangle$	
dta	DATA TYPE ATR	$\langle A \mid \text{FIXED} \mid \text{FLOAT} \mid \text{POINTER} \mid \text{BIT}(n) \mid \text{CHAR}(n) \rangle$	
sca	SCOPE ATR	$\langle A \mid \text{INTERNAL} \mid \text{EXTERNAL} \rangle$	
sta	STORAGE ATR	$\langle A \mid \text{STATIC} \mid \text{AUTOMATIC} \mid \text{BASED ATR} \rangle$	
	BASED ATR	$\langle \text{BASED}(j) \rangle + \text{PTR } ep(j), \text{ BASED } ep(j)$	

(C.4) Expressions and Atomic Components

e	EXPRESSION	$\langle n \mid "c_1 c_2 \dots c_n" \mid r \mid \text{NULL} \rangle$	
	EXPRESSION	$\langle \text{ADDR}(r) \rangle$	$+ \text{ LABEL } \neq \phi_t \langle r: \rho \rangle$
	EXPRESSION	$\langle e_1 q e_2 \rangle$	$+ t_1 = \phi_t \langle e_1: \rho \rangle, t_2 = \phi_t \langle e_2: \rho \rangle, \text{ CONVERTIBLE} \langle t_1: t_2 \rangle$
r	REFERENCE	$\langle i \rangle$	
	REFERENCE	$\langle j + i \rangle$	$+ \text{ BASED } ep(i), \text{ PTR } ep(j), \text{ BASED } ep(j)$
	REFERENCE	$\langle i_1 i_2 \dots i_n \rangle$	$+ \rho' = \text{TYPE ATR} \langle \rho(i_1) \rangle, \rho'(i_2) \dots (i_n) \neq \Lambda$
l	LETTER	$\langle A \mid B \mid \dots \mid Z \rangle$	
n	NUMBER	$\langle 0 \mid 1 \mid \dots \mid 9 \rangle$	
i, j	IDENTIFIER	$\langle i_1 i_2 \dots i_n \rangle$	
c	CHARACTER	$\langle n \mid i \rangle$	
q	OPERATOR	$\langle \text{ADD OP} \mid \text{COMPARISON OP} \rangle$	
	ADD OP	$\langle ++ \rangle$	
	COMPARISON OP	$\langle '<' \mid '=' \mid '>' \rangle$	
	EQUALITY OP	$\langle == \rangle$	

(C.5) Computation of Environments

/* Explicitly declared environment */

β_e	COMPUTED EXTERNAL ENV		
		$\beta_e \langle u_1 u_2 \dots u_n \rangle$	$= \beta_e \langle u_1 \rangle + \beta_e \langle u_2 \rangle + \dots + \beta_e \langle u_n \rangle$
		$\beta_e \langle i \text{ ' : ' } iu \rangle$	$= [i \rightarrow \text{LABEL}]$
		$\beta_e \langle iu \rangle$	$= \Lambda$
		$\beta_e \langle i \text{ ' : ' } du \rangle$	$= \beta_e \langle du \rangle$
		$\beta_e \langle \text{DECLARE } i \text{ eal}; \rangle$	$= [i \rightarrow \beta_e \langle \text{eal} \rangle]$
		$\beta_e \langle \text{DECLARE } i \text{ i sal, msl}; \rangle$	$= [i \rightarrow [\beta_e \langle \text{msl} \rangle], \beta_e \langle \text{sal} \rangle]$
		$\beta_e \langle \text{dta o sca o sta} \rangle$	$= \beta_e \langle \text{dta} \rangle + \beta_e \langle \text{sca} \rangle + \beta_e \langle \text{sta} \rangle$
		$\beta_e \langle A \rangle$	$= \Lambda$
		$\beta_e \langle \text{FIXED} \mid \text{FLOAT} \rangle$	$= \text{ARITH}$
		$\beta_e \langle \text{BIT}(n) \mid \text{CHAR}(n) \rangle$	$= \text{STR}$
		$\beta_e \langle \text{POINTER} \rangle$	$= \text{PTR}$
		$\beta_e \langle \text{INTERNAL} \rangle$	$= \text{INTERNAL}$
		$\beta_e \langle \text{EXTERNAL} \rangle$	$= \text{EXTERNAL}$
		$\beta_e \langle \text{AUTOMATIC} \rangle$	$= \text{AUTOMATIC}$
		$\beta_e \langle \text{STATIC} \rangle$	$= \text{STATIC}$
		$\beta_e \langle \text{BASED}(j) \rangle$	$= \text{BASED}$
		$\beta_e \langle n \text{ i dta} \rangle$	$= i + \beta_e \langle \text{dta} \rangle$
		$\beta_e \langle n \text{ i msl} \rangle$	$= i + [\beta_e \langle \text{msl} \rangle]$
		$\beta_e \langle \text{ms}_1, \text{ms}_2, \dots, \text{ms}_n \rangle$	$= \beta_e \langle \text{ms}_1 \rangle * \beta_e \langle \text{ms}_2 \rangle * \dots * \beta_e \langle \text{ms}_n \rangle$

/* Contextually declared environment */

β_c	COMPUTED CONTEXTUAL ENV		
		$\beta_c \langle u_1 u_2 \dots u_n \rangle$	$= \text{MERGE} \langle \beta_c \langle u_1 \rangle + \beta_c \langle u_2 \rangle + \dots + \beta_c \langle u_n \rangle \rangle$
		$\beta_c \langle i \text{ ' : ' } u \rangle$	$= \beta_c \langle u \rangle$
		$\beta_c \langle \text{GOTO } i; \mid \text{RETURN}; \rangle$	$= \Lambda$
		$\beta_c \langle \text{ALLOCATE } i; \mid \text{FREE } i; \rangle$	$= \Lambda$
		$\beta_c \langle \text{ALLOCATE } i \text{ SET}(j); \rangle$	$= \Lambda$
		$\beta_c \langle \text{FREE } j + i; \rangle$	$= [j \rightarrow \text{PTR}]$
		$\beta_c \langle \text{IF } e \text{ THEN } iu \rangle$	$= \beta_c \langle e \rangle + \beta_c \langle iu \rangle$
		$\beta_c \langle r = e; \rangle$	$= \beta_c \langle r \rangle + \beta_c \langle e \rangle$
		$\beta_c \langle \text{BEGIN}; u_1 u_2 \dots u_n \text{ END}; \rangle$	$= \beta_c \langle u_1 u_2 \dots u_n \rangle - \text{DOMAIN} \langle \beta_e \langle u_1 u_2 \dots u_n \rangle \rangle$
		$\beta_c \langle \text{DECLARE } i \text{ eal}; \rangle$	$= \beta_c \langle \text{eal} \rangle$
		$\beta_c \langle \text{DECLARE } i \text{ i sal, msl}; \rangle$	$= \beta_c \langle \text{sal} \rangle$
		$\beta_c \langle \text{dta o sca o sta} \rangle$	$= [j \rightarrow \text{PTR}] + \text{sta} \equiv \text{BASED}(j)$
		$\beta_c \langle \text{dta o sca o sta} \rangle$	$= \Lambda \quad + \quad (\text{sta} = \Lambda \mid \text{sta} = \text{STATIC} \mid \text{sta} = \text{AUTOMATIC})$
		$\beta_c \langle n \mid "c_1 c_2 \dots c_n" \mid \text{NULL} \rangle$	$= \Lambda$
		$\beta_c \langle i \mid i_1 . i_2 \dots . i_n \rangle$	$= \Lambda$
		$\beta_c \langle \text{ADDR}(r) \rangle$	$= \beta_c \langle r \rangle$
		$\beta_c \langle e_1 \text{ q } e_2 \rangle$	$= \beta_c \langle e_1 \rangle + \beta_c \langle e_2 \rangle$
		$\beta_c \langle j + i \rangle$	$= [j \rightarrow \text{PTR}]$

/* Implicitly declared environment */

β_1	<u>COMPUTED</u>	$\beta_1 \langle u_1 u_2 \dots u_n \rangle$	$= \text{MERGE} \langle \beta_1 \langle u_1 \rangle + \beta_1 \langle u_2 \rangle + \dots + \beta_1 \langle u_n \rangle \rangle$
	<u>IMPLICIT ENV</u>	$\beta_1 \langle i : 'u' \rangle$	$= \beta_1 \langle u \rangle$
		$\beta_1 \langle \text{RETURN}; \rangle$	$= \Lambda$
		$\beta_1 \langle \text{DECLARE } i \text{ eal}; \rangle$	$= \Lambda$
		$\beta_1 \langle \text{DECLARE } i \text{ i eal, mal}; \rangle$	$= \Lambda$
		$\beta_1 \langle \text{GOTO } i; \rangle$	$= [i \rightarrow \text{ARITH}]$
		$\beta_1 \langle \text{ALLOCATE } i; \rangle$	$= [i \rightarrow \text{ARITH}]$
		$\beta_1 \langle \text{ALLOCATE } i \text{ SET}(j); \rangle$	$= [i \rightarrow \text{ARITH}]$
		$\beta_1 \langle \text{FREE } i; \rangle$	$= [i \rightarrow \text{ARITH}]$
		$\beta_1 \langle \text{FREE } j \rightarrow i; \rangle$	$= [i \rightarrow \text{ARITH}]$
		$\beta_1 \langle r = e; \rangle$	$= \beta_1 \langle r \rangle + \beta_1 \langle e \rangle$
		$\beta_1 \langle \text{IF } e \text{ THEN } iu \rangle$	$= \beta_1 \langle e \rangle + \beta_1 \langle iu \rangle$
		$\beta_1 \langle \text{BEGIN}; u_1 u_2 \dots u_n \text{ END}; \rangle$	$= \beta_1 \langle u_1 u_2 \dots u_n \rangle - \text{DOMAIN} \langle \beta_1 \langle u_1 u_2 \dots u_n \rangle \rangle$
		$\beta_1 \langle n \rangle$	$= \Lambda$
		$\beta_1 \langle "c_1 c_2 \dots c_n" \rangle$	$= \Lambda$
		$\beta_1 \langle \text{NULL} \rangle$	$= \Lambda$
		$\beta_1 \langle i_1 \cdot i_2 \cdot \dots \cdot i_n \rangle$	$= \Lambda$
		$\beta_1 \langle i \rangle$	$= [i \rightarrow \text{ARITH}]$
		$\beta_1 \langle j \rightarrow i \rangle$	$= [i \rightarrow \text{ARITH}]$
		$\beta_1 \langle e_1 \text{ q } e_2 \rangle$	$= \beta_1 \langle e_1 \rangle + \beta_1 \langle e_2 \rangle$
		$\beta_1 \langle \text{ADDR}(r) \rangle$	$= \beta_1 \langle r \rangle$

(C.6) Type-checking Predicates

ϵ	<u>DERIVED ATTRIBUTES</u>	DERIVED TYPE ATR <ARITH PTR STR $[i_1 \rightarrow \epsilon_1 i_2 + \epsilon_2 \dots i_n + \epsilon_n]$ > DERIVED NON TYPE ATR <INTERNAL EXTERNAL AUTOMATIC STATIC BASED>
		DERIVED ATR <DERIVED TYPE ATR DERIVED NON TYPE ATR> DERIVED ATR LIST < Λ $da_1, da_2, \dots, da_n$ >
da	<u>CONVERTIBLE</u>	CONVERTIBLE < $t_1 : t_2$ > $\rightarrow \text{RESULT TYPE} \langle t_1 : t_2 \rangle = \text{STR}$
ϕ	<u>COMPUTED TYPE</u>	$\phi_t \langle n : \rho \rangle = \text{ARITH}$ $\phi_t \langle "c_1 c_2 \dots c_n" : \rho \rangle = \text{STR}$ $\phi_t \langle \text{NULL} : \rho \rangle = \text{PTR}$ $\phi_t \langle \text{ADDR}(r) : \rho \rangle = \text{PTR}$ $\phi_t \langle i : \rho \rangle = \text{TYPE ATR} \langle \rho(i) \rangle$ $\phi_t \langle j \rightarrow i : \rho \rangle = \text{TYPE ATR} \langle \rho(i) \rangle$ $\phi_t \langle i_1 \cdot i_2 \cdot \dots \cdot i_n : \rho \rangle = \rho'(i_2) \dots (i_n) + \rho' = \text{TYPE ATR} \langle \rho(i_1) \rangle$ $\phi_t \langle e_1 \text{ q } e_2 : \rho \rangle = \text{RESULT TYPE} \langle t_1 : t_2 \rangle + t_1 = \phi_t \langle e_1 : \rho \rangle, t_2 = \phi_t \langle e_2 : \rho \rangle$
	<u>TYPE ATR</u>	$\text{TYPE ATR} \langle \Lambda \rangle = \text{ARITH}$ $\text{TYPE ATR} \langle da \rangle = da$ $\text{TYPE ATR} \langle da \rangle = \text{ARITH}$ $\text{TYPE ATR} \langle da_1, da_2, \dots, da_n \rangle = da_1$ $\text{TYPE ATR} \langle da_1, da_2, \dots, da_n \rangle = \text{TYPE ATR} \langle da_2, \dots, da_n \rangle + \text{DERIVED NON TYPE ATR} \langle da_1 \rangle$

(C.7) Computation of Result Types

RESULT TYPE

```

/* General */
RESULT TYPE<t1 : q : t2> = RESULT TYPE<t2 : q : t1>

/* Scalars */
RESULT TYPE<t1 : q : t2> = ARITH + PLUS OP<q>, (t1=ARITH | t1=STR), (t2=ARITH | t2=STR)
RESULT TYPE<PTR : q : PTR> = STR + EQUALITY OP<q>
RESULT TYPE<t1 : q : t2> = STR + COMPARISON OP<q> (t1=ARITH | t1=STR), (t2=ARITH | t2=STR)

/* Addition of Structures */
RESULT TYPE<[i1+t1*i2+t2*...*in+tn] : q : [i1'+t1'*i2'+t2'*...*in'+tn']> = [i1 + t1'*i2 + t2'*...*in + tn']
+ PLUS OP<q>, RESULT TYPE<t1:+:t1'> = t1', RESULT TYPE<t2:+:t2'> = t2', ..., RESULT TYPE<tn:+:tn'> = tn'

/* Comparison of Structures */
RESULT TYPE<[i1+t1*i2+t2*...*in+tn] : q : [i1'+t1'*i2'+t2'*...*in'+tn']> = STR
+ COMPARISON OP<q>, RESULT TYPE<t1:q:t1'> = STR,
RESULT TYPE<t2:q:t2'>, ..., RESULT TYPE<tn:q:tn'> = STR

/* Scalar Operations on Structures */
RESULT TYPE<t : q : [i1+t1*i2+t2*...*in+tn]>
= RESULT TYPE<[i1+t*i2+t*...*in+t] : q : [i1+t1:i2+t*...*in+tn]>

```

(C.8) Miscellaneous Predicates

ρ	<u>ENV</u>	ENVIRONMENT< Λ [i ₁ + dal ₁ * i ₂ + dal ₂ * ... * i _n + dal _n]>
	<u>DOMAIN</u>	DOMAIN< Λ > = Λ DOMAIN< ρ > = i ₁ , i ₂ , ..., i _n + $\rho \equiv [i_1 + dal_1 * i_2 + dal_2 * \dots * i_n + dal_n]$
	<u>APPLY</u>	APPLY< ρ :i> = ERROR + i $\notin$ DOMAIN< ρ > APPLY< ρ :i> = dal ₁ + $\rho \equiv [i_1 + dal_1 * i_2 + dal_2 * \dots * i_n + dal_n]$, i=i ₁ APPLY< ρ :i> = dal ₂ + $\rho \equiv [i_1 + dal_1 * i_2 + dal_2 * \dots * i_n + dal_n]$, i=i ₂ ... APPLY< ρ :i> = dal _n + $\rho \equiv [i_1 + dal_1 * i_2 + dal_2 * \dots * i_n + dal_n]$, i=i _n
	<u>PLUS</u>	PLUS< ρ : Λ > = Λ PLUS< ρ : ρ' > = ρ'' + $\rho \equiv [x]$, $\rho' \equiv [y]$, $\rho'' \equiv [x*y]$
	<u>DELETE</u>	DELETE< ρ : Λ > = Λ DELETE< ρ :i> = $\rho_1 + \rho_2$ + $\rho = \rho_1 + [i + dal] + \rho_2$ DELETE< ρ :i, i> = DELETE< ρ' :i> + $\rho' = \text{DELETE}(\rho:i)$ OVERRIDE< ρ : ρ' > = ρ'' + $\rho_1 = \text{DELETE}(\text{DOMAIN}(\rho):\rho')$, $\rho'' \equiv \rho + \rho_1$
	<u>MERGE</u>	MERGE< Λ > = Λ MERGE< ρ > = ρ + DIFF IDLIST<DOMAIN< ρ >> MERGE< ρ > = $\rho_1 + \rho_2$ + $\rho = \rho_1 + [i + dal] + \rho_2$, i $\in$ DOMAIN< $\rho_1 + \rho_2$ >
	<u>LEVEL NUM</u>	LEVEL NUM <n ₁ i ₁ succ ₁ , n ₂ i ₂ succ ₂ , ..., n _k i _k succ _k > = n _k

Table A : ABSTRACT SYNTAX OF PL-1

(A.1) Programs

p	PROGRAM	$\langle (S-ENTR-NAME\ i, S-DCL-PART\ dp, S-IMPER-PART\ ip) \rangle$ $+ \rho = B\langle dp \rangle, i \notin DOMAIN\langle \rho \rangle, DIFF\ IDLIST\langle DOMAIN\langle \rho \rangle \rangle$ $DCL\ PART\langle dp:\rho \rangle, IMPER\ PART\langle ip:\rho \rangle$
dp	DCL PART	$\langle (S-DCL-1\ du_1, S-DCL-2\ du_2, \dots, S-DCL-n\ du_n) \rangle$ $+ DIFF\ IDLIST\langle S-ID\langle du_1 \rangle, S-ID\langle du_2 \rangle, \dots, S-ID\langle du_n \rangle \rangle$
ip	IMPER PART	$\langle (S-IMPER-1\ iu_1, S-IMPER-2\ iu_2, \dots, S-IMPER-n\ iu_n) \rangle$
du	DCL UNIT	$\langle ELEMENT\ DCL \mid STRUCTURE\ DCL \mid LABEL\ DCL \rangle$
iu	IMPER UNIT	$\langle ASGT\ STM \mid GOTO\ STM \mid IF\ STM \mid ALLOCATE\ STM \mid TREE\ STM \mid RETURN\ STM \mid$ $BLOCK \mid LABELED\ UNIT \rangle$
	LABELED UNIT	$\langle (S-LABEL-PREFIX\ i, S-IMPER\ iu) \rangle$

(A.2) Imperative Statements

	ASGT STM	$\langle (S-SYM\ x, S-REF\ r, S-EXP\ e) \rangle$ $+ ASGT\ SYM\langle x \rangle, t_1 = \phi_e\langle r:\rho \rangle, t_2 = \phi_e\langle e:\rho \rangle, CONVERTIBLE\langle t_1:t_2 \rangle$
	GOTO STM	$\langle (S-SYM\ x, S-LABEL-REF\ i) \rangle$ $+ GOTO\ SYM\langle x \rangle, LABEL\in \rho(i)$
	IF STM	$\langle (S-SYM\ x, S-EXP\ e, S-IMPER-UNIT\ iu) \rangle$ $+ IF\ SYM\langle x \rangle, t_1 = \phi_e\langle e:\rho \rangle, CONVERTIBLE\langle t:STR \rangle$
	ALLOCATE STM	$\langle (S-SYM\ x, S-BASED-VAR\ i, S-PTR-REF\ A) \rangle$ $+ ALLOC\ SYM\langle x \rangle, BASED\in \rho(i)$
	ALLOCATE STM	$\langle (S-SYM\ x, S-BASED-VAR\ i, S-PTR-REF\ j) \rangle$ $+ ALLOC\ SYM\langle x \rangle, BASED\in \rho(i), PTR\in \rho(j)$
	FREE STM	$\langle (S-SYM\ x, S-BASED-VAR\ i, S-PTR-REF\ A) \rangle$ $+ FREE\ SYM\langle x \rangle, BASED\in \rho(i)$
	FREE STM	$\langle (S-SYM\ x, S-BASED-VAR\ i, S-PTR-REF\ j) \rangle$ $+ FREE\ SYM\langle x \rangle, BASED\in \rho(i), PTR\in \rho(j), BASED\neq \rho(j)$
	RETURN STM	$\langle (S-SYM\ x,) \rangle + RETURN\ SYM\langle x \rangle$ $\langle (S-IMPER-PART\ ip) \rangle$
	BLOCK	$\langle (S-DCL-PART\ dp, S-IMPER-PART\ ip) \rangle$ $+ \rho_1 = B\langle dp \rangle,$ $\rho' = \overline{OVERRIDE}\langle \rho_1:\rho \rangle$ $DIFF\ IDLIST\langle DOMAIN\langle \rho_1 \rangle \rangle$ $DCL\ PART\langle dp:\rho' \rangle, IMPER\ PART\langle ip:\rho' \rangle$

(A.3) Declarations

	ELEMENT DCL	$\langle (S-ID\ i, S-ATR-LIST\ sal) \rangle$
	STRUCTURE DCL	$\langle (S-ID\ i, S-ATR-LIST\ sal, S-MINOR-STRUC-LIST\ ms1) \rangle$
	LABEL DCL	$\langle (S-ID\ i, S-LABEL-ATR\ x) \rangle + LABEL\ SYM\langle x \rangle$
sal	EL ATR LIST	$\langle (S-DATA-TYPE-ATR\ dta, S-SCOPE-ATR\ sca, S-STORAGE-ATR\ sta) \rangle$ $+ (sca \neq EXTERNAL\ SYM \mid sta = STATIC\ SYM)$
sal	STRUC ATR LIST	$\langle (S-SCOPE-ATR\ sca, S-STORAGE-ATR\ sta) \rangle$ $+ (sca \neq EXTERNAL\ SYM \mid sta = STATIC\ SYM)$
ms1	MINOR STRUC LIST	$\langle (S-EL-1\ ms_1, S-EL-2\ ms_2, \dots, S-EL-n\ ms_n) \rangle$ $+ DIFF\ IDLIST\langle S-ID\langle ms_1 \rangle, S-ID\langle ms_2 \rangle, \dots, S-ID\langle ms_n \rangle \rangle$
ms	MINOR STRUC	$\langle (S-ID\ i, S-SUCC\ succ) \rangle$
succ	STRUC SUCC	$\langle (S-DATA-TYPE-ATR\ dta) \mid (S-MINOR-STRUC-ID\ i, S-MINOR-STRUC-LIST\ ms1) \rangle$
dta	DATA TYPE ATR	$\langle FIXED\ SYM \mid FLOAT\ SYM \mid POINTER\ SYM \mid STRING\ DATA\ ATR \rangle$
	STRING DATA ATR	$\langle (S-STR-BASE\ x, S-LENGTH\ n) \rangle + (CHAR\ SYM\langle x \rangle \mid BIT\ SYM\langle x \rangle)$
sca	SCOPE ATR	$\langle INTERNAL\ SYM \mid EXTERNAL\ SYM \rangle$
sta	STORAGE ATR	$\langle STATIC\ SYM \mid AUTOMATIC\ SYM \mid BASED\ ATR \rangle$
	BASED ATR	$\langle (S-BASED-SYM\ x, S-PTR-REF) \rangle + BASED\ SYM\langle x \rangle, PTR\in \rho(j), BASED\neq \rho(j)$

(A.4) Expressions and Atomic Components

e	EXPRESSION	<NUMBER CHAR STR NULL SYM REFERENCE>
	EXPRESSION	<(S-ADDR-SYM x, S-REF r)> + ADDR SYM x, LABEL $\neq \phi_t<r:p>$
	EXPRESSION	<(S-EXP1 e, S-OP q, S-EXP2 e ₂)> + OPERATOR<q>, $t_1 = \phi_t<e_1:p>$, $t_2 = \phi_t<e_2:p>$, CONVERTIBLE<t ₁ :t ₂ >
r	REFERENCE	<i>
	REFERENCE	<(S-PTR-QUAL j, S-ID i)> + PTR $\in \rho(j)$, BASED $\neq \rho(j)$, BASED $\in \rho(i)$
	REFERENCE	<(S-ID-1 i ₁ , S-ID-2 i ₂ , ..., S-ID-n i _n)> + $\rho' \equiv \text{TYPE ATR}<\rho(i_1)>$, $\rho'(i_2)...(i_n) \neq \Lambda$
	OPERATOR	<ADD OP COMPARISON OP>

(A.5) Computation of Environments

β	$\beta<du_1, du_2, \dots, du_n>$	$= [\beta<du_1> * \beta<du_2> * \dots * \beta<du_n>]$
	$\beta<(i, x)>$	$= i + \text{LABEL} \quad + \text{LABEL SYM}<x>$
	$\beta<(i, eal)>$	$= i + \beta<eal>$
	$\beta<(i, eal, msl)>$	$= i + [\beta<msl>], \beta<eal>$
	$\beta<(dta, sca, sta)>$	$= \beta<dta>, \beta<sca>, \beta<sta>$
	$\beta<(sca, sta)>$	$= \beta<sca>, \beta<sta>$
	$\beta<\text{FIXED SYM}>$	$= \text{ARITH}$
	$\beta<\text{FLOAT SYM}>$	$= \text{ARITH}$
	$\beta<\text{POINTER SYM}>$	$= \text{PTR}$
	$\beta<\text{STRING DATA ATR}>$	$= \text{STR}$
	$\beta<\text{INTERNAL SYM}>$	$= \text{INTERNAL}$
	$\beta<\text{EXTERNAL SYM}>$	$= \text{EXTERNAL}$
	$\beta<\text{AUTOMATIC SYM}>$	$= \text{AUTOMATIC}$
	$\beta<\text{STATIC SYM}>$	$= \text{STATIC}$
	$\beta<\text{BASED ATR}>$	$= \text{BASED}$
	$\beta<(i, dta)>$	$= i + \beta<dta>$
	$\beta<(i, msl)>$	$= i + [\beta<msl>]$
	$\beta<(ms_1, ms_2, \dots, ms_n)>$	$= \beta<ms_1> * \beta<ms_2> * \dots * \beta<ms_n>$

(A.6) Type-checking Predicates

t	DERIVED TYPE ATR<ARITH PTR STR [i ₁ +t ₁ *i ₂ +t ₂ *...*i _n +t _n]>
	DERIVED NON TYPE ATR<INTERNAL EXTERNAL AUTOMATIC STATIC BASED>
da	DERIVED ATR<DERIVED TYPE ATR DERIVED NOW TYPE ATR>
dal	DERIVED ATR LIST<da ₁ , da ₂ , ..., da _n >
	CONVERTIBLE<t ₁ :t ₂ > + RESULT TYPE<t ₁ : = : t ₂ > = STR
d _t	$\phi_t<\text{NUMBER}:p>$ = ARITH
	$\phi_t<\text{CHAR STR}:p>$ = STR
	$\phi_t<\text{NULL SYM}:p>$ = PTR
	$\phi_t<(x, r):p>$ = PTR + ADDR SYM<x>
	$\phi_t<i:p>$ = TYPE ATR< $\rho(i)$ >
	$\phi_t<(j, i):p>$ = TYPE ATR< $\rho(i)$ > + PTR $\in \rho(j)$
	$\phi_t<(i_1, i_2, \dots, i_n):p>$ = $\rho'(i_2)...(i_n)$ + $\rho' = \text{TYPE ATR}<\rho(i_1)>$
	$\phi_t<(e_1, q, e_2):p>$ = RESULT TYPE<t ₁ :q:t ₂ > + $t_1 = \phi_t<e_1:p>$, $t_2 = \phi_t<e_2:p>$
	TYPE ATR<da ₁ , da ₂ , da ₃ > = da ₁

(A.7) Computation of Result Types - Same as in Concrete Syntax

(A.8) Miscellaneous predicates - Same as in Concrete Syntax

Table L: MAPPING OF λ -CALCULUS EXPRESSIONS INTO NORMAL FORM

(L.1) Primitive Predicates

i, j, k	IDENTIFIER	An ordered set of distinct atoms: $a_1, a_2, a_3, \dots$
	DIFF ID	The set of all pairs of different identifiers

(L.2) λ -Calculus Expressions

e, f	EXPRESSION	<u>IDENTIFIER</u> <u>COMBINATION</u> <u>LAMBDA EXP</u>
	COMBINATION	$\langle e_1(e_2) \rangle$
	LAMBDA EXP	$\langle \lambda i.e \rangle$

(L.3) Free and Bound Identifiers

<u>FREE IDS</u>	<u>FREE IDS</u> $\langle i \rangle$	= i
	<u>FREE IDS</u> $\langle e_1(e_2) \rangle$	= <u>UNION</u> $\langle$ <u>FREE IDS</u> $\langle e_1 \rangle$; <u>FREE IDS</u> $\langle e_2 \rangle \rangle$
	<u>FREE IDS</u> $\langle \lambda i.e \rangle$	= <u>REL COMP</u> $\langle$ <u>FREE IDS</u> $\langle e \rangle$; i $\rangle$
<u>BOUND IDS</u>	<u>BOUND IDS</u> $\langle i \rangle$	= Λ
	<u>BOUND IDS</u> $\langle e_1(e_2) \rangle$	= <u>UNION</u> $\langle$ <u>BOUND IDS</u> $\langle e_1 \rangle$; <u>BOUND IDS</u> $\langle e_2 \rangle \rangle$
	<u>BOUND IDS</u> $\langle \lambda i.e \rangle$	= <u>UNION</u> $\langle$ i; <u>BOUND IDS</u> $\langle e \rangle \rangle$

(L.4) λ -Calculus Substitution Rule

ϕ	$\phi \langle i : f : i \rangle$	= f
	$\phi \langle i : f : j \rangle$	= j
	$\phi \langle i : f : e_1(e_2) \rangle$	= $\phi \langle i : f : e_1 \rangle (\phi \langle i : f : e_2 \rangle)$
	$\phi \langle i : f : \lambda i.e \rangle$	= $\lambda i.e$
	$\phi \langle i : f : \lambda j.e \rangle$	= $\lambda j.\phi \langle i : f : e \rangle$
	$\phi \langle i : f : \lambda j.e \rangle$	= $\lambda k.\phi \langle i : f : e' \rangle$
		+ DIFF ID $\langle i:j \rangle$, j $\notin$ <u>FREE IDS</u> $\langle f \rangle$, DIFF ID $\langle i:j \rangle$, j $\in$ <u>FREE IDS</u> $\langle f \rangle$, DIFF ID $\langle i:k \rangle$, k $\notin$ <u>FREE IDS</u> $\langle e \rangle$, k $\in$ <u>FREE IDS</u> $\langle f \rangle$, e' = $\phi \langle j:k:e \rangle$

(L.5) Conversion to Normal Form

	NORMAL FORM $\langle i \rangle$	
	NORMAL FORM $\langle e_1(e_2) \rangle$	+ NORMAL FORM $\langle e_1 \rangle, \langle e_2 \rangle$, (ID $\langle e_1 \rangle$ COMBINATION $\langle e_1 \rangle$)
	NORMAL FORM $\langle \lambda i.e \rangle$	+ NORMAL FORM $\langle e \rangle$
	EXP : NORMAL FORM $\langle e:e \rangle$	+ NORMAL FORM $\langle e \rangle$
	EXP : NORMAL FORM $\langle e_1:e_2 \rangle$	+ NORMAL FORM $\langle e_2 \rangle$, CONV $\langle e_1 \rangle$ = e_2
	CONV $\langle \lambda i.e \rangle$	= $(\lambda j.\phi \langle i:j:e \rangle)$
	CONV $\langle \lambda i.e \rangle (f)$	= $\phi \langle i:f:e \rangle$
	CONV $\langle x e_1 y \rangle$	= x e_2 y + EXP $\langle x e_1 y \rangle$, CONV $\langle e_1 \rangle$ = e_2