PRODUCTION SYSTEMS: A NOTATION FOR DEFINING SYNTAX AND TRANSLATION

BY

Henry F. Ledgard*

COINS Technical Report 75A-4
(Revised August 1976)

* Computer and Information Science Department
University of Massachusetts, Amherst, MA

Index Terms: Formal Definition, Language Design, Standards, Syntax, Translation, Context-Sensitive Grammars, Compilers, Lambda-Calculus, PL/I

This work was supported in part by the U.S. Army Research Office.

ABSTRACT

This paper presents the formalism of Production Systems and investigates its application to define the syntax and translation of programming languages. Several properties appear well-suited to this task:

(1) The formalism can be used to specify exactly the syntax of a computer language, including context-sensitive requirements.

(2) The specification of the context-sensitive requirements on syntax can be isolated from the context-free requirements.

(3) The same formalism can be used to specify the translation of one language into another.

The notation has been developed with readability as a prime design issue.

The following examples are given:

(1) A specification of the syntax of a small but difficult subset of PL/I.

(2) A specification of the translation of lambda-calculus expressions into normal form.

1. INTRODUCTION

It is almost impossible to overestimate the value of formal definitions in the language area: unduly complicated constructs, omissions of critical detail, different interpretations of a given construct, incompatible implementations of a language standard, and repeated user confusion are commonplace. One major part of this difficulty is due to a poor technology for providing *readable* methods for *complete* definitions.

This paper presents a formal notation capable of fully defining what strings in a language are legal programs and what the legal programs "mean" in terms of some suitable target language. Perhaps the most important reason for the widespread use of context-free grammars, notably Backus-Naur form, is the clarity with which context-free portions of syntax can be specified. Owing to the more complex nature of context-sensitive requirements and the specification of translation, some additional complexity in a formal notation must be expected. For clarity, the conceptual framework of a notation is vital in that the conceptual framework either lends itself naturally or unnaturally to the application.

The conceptual notions of "generative productions", "sets", and "strings" underlie 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 generative notation for Production System is a notation for defining functions. Via this notation, portions of a Production Systems *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 all sets are defined generatively.

The mathematical underpinnings of Production Systems are due to Emil Post [10] and Raymond Smullyan [12]. With suitable syntactic changes Production Systems are equivalent to Smullyan's "elementary formal systems" [12], which can be used to specify any recursively enumerable set. 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 and their translations into a target language are just two examples of recursively enumerable sets. The notation and terminology for Production Systems presented here stems from Post and Smullyan, but for the most part is new. A more detailed history of Production Systems is given in [5,4,3]. A detailed exposition of other formal systems, as well as a discussion of the importance of formal definitions, is given in [8].

2. 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 difficult 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 PL.1 using Production Systems (in abbreviated form) is given in Table 1.

For example, this subset contains the simple procedure,

```
P:  PROCEDURE;
    DECLARE A FIXED;
    DECLARE B FLOAT;
        A = 0;
        B = 8;
    L:  A = A+1;
        IF A > B THEN GOTO L;
END P;
```

as well as, the procedure of Table 2, which constructs binary tree linkages similar to those that might be needed in the symbol table of a compiler. Note: the pointers *NEWPTR* and *TREEPTR* are assumed to point to structures allocated in a calling procedure.

Pl.1 contains numerous context-sensitive requirements on its syntax. Table 3 contains a list of some relevant statements adapted from the PL/I Language standard [13]. 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 syntactic errors, i.e. those that are picked up by a PL/I compiler, and not semantic or run-time errors.

TABLE 1: CONTEXT-FREE SYNTAX OF PL.1

[a. *Programs and Units*]

prog   PROGRAM <id: *PROCEDURE:* unit$_1$ unit$_2$ ... unit$_n$ *END* id;>.

unit   UNIT <exec-unit | dcl-unit>.

exec-unit   EXECUTABLE UNIT <id: exec-unit | stm>.

dcl-unit   DECLARATIVE UNIT <id: dcl-unit | dcl>.

[b. *Imperative Statements*]

stm   ASSIGNMENT STM <ref = exp*;*>.

stm   GOTO STM <*GOTO* id*;*>.

stm   IF STM <*IF* exp *THEN* exec-unit>.

stm   ALLOCATE STM <*ALLOCATE* id*;* | *ALLOCATE* id$_1$ *SET(*id$_2$*);*>.

stm   FREE STM <*FREE* id*;* | *FREE* id$_1$ → id$_2$*;*>.

stm   RETURN STM <*RETURN;*>.

stm   BLOCK <*BEGIN;* unit$_1$ unit$_2$ .. unit$_n$ *END;*>.

[c. *Declarations*]

dcl   ELEMENT DECLARATION <*DECLARE* id elem-atr-list*;*>.

dcl   STRUCTURE DECLARATION <*DECLARE* 1 id struc-atr-list, minor-struc-list*;*>.

elem-atr-list   ELEMENT ATTRIBUTE LIST <type-atr ∘ scope-atr ∘ storage-atr>.

struc-atr-list   STRUCTURE ATTRIBUTE LIST <scope-atr ∘ storage-atr>.

minor-struc-list   MINOR STRUCTURE LIST <n$_1$ id$_1$ succ$_1$, n$_2$ id$_2$ succ$_2$, ... n$_k$ id$_k$ succ$_k$>.

succ   STRUCTURE SUCCESSOR <type-atr | n id minor-struc-list>.

type-atr   DATA TYPE ATR <Λ | *FIXED* | *FLOAT* | *BIT(*n*)* | *CHAR(*n*)*>.

scope-atr   SCOPE ATR <Λ | *INTERNAL* | *EXTERNAL*>.

storage-atr   STORAGE ATR <Λ | *STATIC* | *AUTOMATIC* | *BASED(*id*)*>.

[d. *Expressions and Atomic Components*]

exp   EXPRESSION <n | "c$_1$c$_2$...c$_n$" | *NULL* | ref | *ADDR(*ref*)* | exp$_1$ op$_1$ exp$_2$>.

ref   REFERENCE <id | id$_1$ → id$_2$ | id$_1$.id$_2$ ... .i$_n$>.

op   OPERATOR <+ | ≤ | = | ≥ >.

id   IDENTIFIER <$\ell_1 \ell_2$ ... ℓ_n>.

ℓ LETTER <*A* | *B* | ... | *Z*>.

n   NUMBER <d$_1$d$_2$ ... d$_n$>.

d   DIGIT <*0* | *1* | ... | *9*>.

c   CHARACTER <ℓ | d>.

Table 2: A PROCEDURE TO CONSTRUCT BINARY TREES

```
BUILD:  PROCEDURE;

  DECLARE NEWPTR POINTER EXTERNAL;
  DECLARE TREEPTR POINTER EXTERNAL;
  DECLARE P POINTER;
  DECLARE 1 TREE  BASED(P),
                2  ID  CHAR(6),
                2  VALUE  FIXED,
                2  LEFTPTR  POINTER,
                2  RIGHTPTR  POINTER;
  DECLARE 1 LEAF  BASED(NEWPTR),
                2  ID  CHAR(6),
                2  VALUE  FIXED,
                2  LEFTPTR  POINTER,
                2  RIGHTPTR  POINTER;

    IF TREEPTR = NULL  THEN
       BEGIN;
         TREEPTR = NEWPTR;
         RETURN;
       END;

    P = TREEPTR;

 L: IF TREE.ID < LEAF.ID  THEN
       BEGIN;
         IF TREE.LEFTPTR = NULL
            THEN BEGIN;
                   TREE.LEFTPTR = NEWPTR;
                   RETURN;
                 END;
         P = TREE.LEFTPTR;
         GOTO L;
       END;

    IF TREE.ID = LEAF.ID  THEN
       RETURN;

    IF TREE.ID > LEAF.ID  THEN
       BEGIN;
         IF TREE.RIGHTPTR = NULL
            THEN BEGIN;
                   TREE.RIGHTPTR = NEWPTR;
                   RETURN;
                 END;
         P = TREE.RIGHTPTR;
         GOTO L;
       END;

 END BUILD;
```

TABLE 3: SOME REQUIREMENTS ON THE SYNTAX OF PL.1

1. An identifier specified in a declaration as a structure or variable is said to be *explicitly* declared.

2. A label identifier prefixing a 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. In these cases the identifier is given the *POINTER* type attribute.

4. An identifier that appears in a program and is not *explicitly* or *contextually* declared is said to be *implicitly* declared. In these cases, the identifier is given the *FIXED* type attribute.

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 declared both as a pointer and as a floating-point variable.

7. If no storage class attribute is specified and the scope is internal, the storage class attribute *AUTOMATIC* is assumed. If no storage class attribute is specified and the scope is external, *STATIC* is assumed. If neither a storage class nor a scope attribute is specified, then *AUTOMATIC* is assumed.

8. Automatic and based variables can have *INTERNAL* scope only.

9. Storage class and scope attributes cannot be specified for members of structures.

10. In all of the *EXTERNAL* declarations for the same identifier, the attributes declared must be consistent.

11. All structure variables in a structure expression must have identical structuring. Identical structuring means that structures must have the same minor structuring and the same number of contained elements and arrays. The positioning of the elements and arrays within the structure must be identical. Identifiers of corresponding elements do not have to be the same, and data types of corresponding elements do not have to be the same as long as valid conversion can be performed.

12. In an assignment to a structured variable, all the structure operands on the right-hand side must have the same number of contained items as the structure variable on the left-hand side.

13. The based variable appearing in an *ALLOCATE* statement must be an element variable or a major structure.

14. Pointer variables cannot be operands of any operators except the comparison operator "=". Assignment of a pointer can be made only to another pointer variable.

The context-sensitive requirements of PL.I rule out many potential PL/I programs. The program

```
Q:  PROCEDURE;
        L:  X = 1;
        L:  Y = 1;
END Q;
```

is illegal due to a multiple label declaration. The program

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

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

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

is illegal since *X* is implicitly declared with the attribute *FIXED* 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;
```

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

```
2  LEFTPTR POINTER,
```

in the declaration of *TREE* by

```
2  LEFTPTR FIXED,
```

causes a type error in the comparison of *TREE.LEFTPTR* 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.

The process of writing a formal definition forces one to resolve issues that might easily 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. It is my contention that the complexity of the Production System defining the complete syntax of PL.1 well illustrates an overly complex language design.

3. PRODUCTION SYSTEMS

3.1 Basic 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 members 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 it is necessary to be able to determine the alphabet from which each symbol is drawn. The following symbols are used for Production Systems:

(1) strings of capital letters (possibly interlaced with spaces) for predicate alphabet symbols;

(2) strings of lower case letters (possibly hyphenated, subscripted, or superscripted) for variable alphabet symbols;

(3) the symbols:

←	implication symbol
&	conjunction symbol
:	tuple symbol
< >	left and right tuple bracket symbols
[ ]	left and right comment bracket symbols
.	termination symbol

for punctuation symbols

(4) symbols not in the predicate, variable, or punctuation alphabets for object alphabet symbols.

(*Note:* In the sequel, I will introduce a few additional symbols and conventions relevant to the definition of programming languages.)

A well-formed term consists of a concatenated sequence of variable and object alphabet symbols, e.g. "id", "exp_1 + exp_2", and "ref = exp;". 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. "<ref = exp;>" and "<id_1 : id_2>." The degree of the term tuple is the number of terms, n.

```
IF STM < IF  exp  THEN  exec-unit>      ←    EXPRESSION<exp>   &   EXECUTABLE UNIT<exec-unit>.

                └──┘   └────────┘
               object   variable
               string

└───┘  └───────────────┘                └────────────┘
predi-        term                       atomic formula
cate

└──────────────────────┘                └────────────┘    └──────────────────┘
       conclusion                            premise             premise

└──────────────────────────────────────────────────────────────────────────┘
                                production
```

Table 4: SUMMARY OF PRODUCTION SYSTEM NOTATION AND TERMINOLOGY

A well-formed atomic formula consists of a predicate alphabet symbol of degree n followed by a well-formed term-tuple of degree n, for example,

ASSIGNMENT STM <ref = exp;>
NOT IN <id_1 : id_2>

where "ASSIGNMENT STM" and "NOT IN" are predicates of degrees l and 2 respectively. A well-formed production consists of

(a) an atomic formula followed by a termination symbol, or

(b) an atomic formula followed by the implication sign, a sequence of atomic formulas each separated by the conjunction sign, and a termination bymbol.

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*.

In the specification of written expressions in computer languages, it will often be necessary to use the symbols in the predicate, or variable alphabet as members of the object alphabet. Since capital letters, digits, and the punctuation symbols

← & .

cannot occur within the brackets of a term tuple as predicate, variable, or punctuation alphabet symbols, these symbols can be unambiguously used in a term as object alphabet symbols. Furthermore, strings containing other variable or punctuation symbols will be used as members of the object alphabet provided that the symbols are underlined; for example a̲ or :̲. Table 4 gives a summary of the notation.

Consider the productions of Table 5. Here, the symbols "*A*" through "*Z*" enclosed in angle brackets are object alphabet symbols. The symbols "id_1", "id_2", and "ℓ" are variable alphabet symbols.

Table 5: <u>UNABBREVIATED PRODUCTIONS</u>

ID <*A*>.
ID <*B*>.
⋮
ID <*Z*>.

DIFF ID <*A* :: *B*>.
DIFF ID <*A* : *C*>.
⋮
DIFF ID <*Z* : *Y*>.

IDLIST <id>
  ← ID<id>.

IDLIST <ℓ,id>
  ← IDLIST<ℓ> & ID<id>.

NOT IN <id_1 : id_2>
  ← ID<id_1> & & ID<id_2> &
    DIFF ID<id_1 : id_2>.

NOT IN <id_1 : ℓ,id_2>
  ← ID<id_1> & ID<id_2> &
    IDLIST<ℓ> & DIFF ID<id_1 : id_2> &
    NOT IN<id_1 : ℓ>.

DIFF IDLIST <id>
  ← ID<id>.

DIFF IDLIST <ℓ,id>
  ← DIFF IDLIST<ℓ> & NOT IN<id : ℓ>.

3.2 Deductive Rules and Interpretation

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: *Uniform Replacement Rule:* 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: *Implication:* 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 two rules can be applied to the production 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*>

A Production System will be interpreted in the following way. A predicate will denote the name of a set. Productions will be viewed as rewriting rules for enumerating members of sets. A term tuple of degree n following the predicate of a derived conclusion will be taken as an assertion that the n-tuple is one member of the named set. In the previously given productions of Table 5,

(1) the set named ID of identifiers consists of the 26 letters of the English alphabet,

(2) the set named DIFF ID consists of all pairs of different identifiers,

(3) the set named IDLIST consists of all lists of identifiers,

(4) the set named NOT IN consists of all pairs where the first element is an identifier and the second element is a list of identifiers not containing the first identifier,

(5) the set named DIFF IDLIST consists of all lists of different identifiers.

4. ABBREVIATIONS TO THE BASIC NOTATION

Using only the basic notation for Production Systems, a specification for a programming language would quickly become lengthy and cumbersome. Considerable clarity for Production Systems has been obtained by introducing abbreviations to the basic notation. Three principal factors have governed the kind of abbreviations introduced:

(1) an attempt to develop a conceptual framework facilitating language specification,

(2) an attempt to isolate the context-free portions of syntax, context-sensitive portions of syntax, and translation,

(3) reduction in the length of a specification.

For brevity, each abbreviation will be specified mainly by example.

Abbreviation 1: Factoring Common Variables and Predicates

It is somewhat more transparent to use one variable throughout a Production System to refer to members of a single predicate. If this convention is strictly followed, then we may use the following abbreviation to "factor out" the premises referring to these variables. Let v be a variable and P be its corresponding predicate. Then v can be listed beside the productions defining members of P and all premises of the form P<v> can be deleted. For example, the productions:

```
ID<A>.
ID<B>.
 :
ID<Z>.

IDLIST <id>
   ← ID<id>.

IDLIST <id,ℓ>
   ← ID<id>    &   IDLIST<ℓ>.

NOT IN<id₁ : id₂>
   ← ID<id₁>   &  ID<id₂>    &
     DIFF ID<id₁ : id₂>.

NOT IN<id₁ : id₂,ℓ>
   ← ID<id₁>  &   ID<id₂>   &
     IDLIST<ℓ>  &   DIFF ID<id₁ : id₂>   &
     NOT IN<id₂ : ℓ>.
```

can be abbreviated

```
id    ID <A>.
id    ID <B>.
        :
id    ID <Z>.

ℓ     IDLIST <id>.
ℓ     IDLIST <id,ℓ>.

      NOT IN <id1 : id2>
         ← DIFF ID<id1 : id2>.

      NOT IN <id1 : id2,ℓ>
         ← DIFF ID<id1 : id2>  &
           NOT IN<id2 : ℓ>.
```

Abbreviation 2: <u>Membership is Lists, Testing for Equality, and Arithmetic Predicates</u>

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

<A : C,D,E>   *<X : A,B,B,Y>*

are members of NOT IN, whereas the pairs

<C : C,D,E>   *<B : A,B,B,Y>*

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

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

```
IN <id : ℓ>
IN <BASED : ρ(id)>

NOT IN <id : ℓ>
NOT IN <id : DOMAIN(ρ)>
```

may be abbreviated

$$\begin{array}{l} Id \ \varepsilon \ \ell \\ BASED \ \varepsilon \ \rho(Id) \end{array}$$

$$\begin{array}{l} Id \not\varepsilon \ \ell \\ Id \not\varepsilon \ \underline{DOMAIN}(\rho) \end{array}$$

Similarly, let s_1 and s_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

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

$$\begin{array}{ll} s_1 = s_2 & s_1 \neq s_2 \\ n_1 < n_2 & n_1 \leq n_2 \\ n_1 > n_2 & n_1 \geq n_2 \end{array}$$

For example, the production

$$\begin{array}{l} RESULT\ TYPE \langle type_1 : \overline{+} : \overline{type_2 : ARITH} \rangle \\ \quad \leftarrow EQ\langle type_1 : ARITH \rangle \ \ \& \ \ EQ\langle type_2 : ARITH \rangle . \end{array}$$

may be abbreviated

$$\begin{array}{l} RESULT\ TYPE \langle type_1 : \overline{+ : type_2} : ARITH \rangle \\ \quad \leftarrow type_1 = ARITH \ \ \& \ \ type_2 = ARITH . \end{array}$$

Abbreviation 3: <u>Disjunction and the use of "|"</u>

In many cases identical conclusions can be derived from multiple premises. In these cases, the disjunction symbol "|" is introduced. For example, the productions:

```
RESULT TYPE <type1 : + : type2 : ARITH>
   ← type1 = ARITH  &  type2 = ARITH.
RESULT TYPE <type1 : + : type2 : ARITH>
   ← type1 = ARITH  &  type2 = STRING.
RESULT TYPE <type1 : + : type2 : ARITH>
   ← type1 = STRING  &  type2 = ARITH.
RESULT TYPE <type1 : + : type2 : ARITH>
   ← type1 = STRING  &  type2 = STRING.
```

may be abbreviated

```
RESULT TYPE <type1 : + : type2 : ARITH>
   ← (type1 = ARITH | type1 = STRING)   &
     (type2 = ARITH | type2 = STRING).
```

Similarly, multiple members of predicates can be derived from identical (possibly null) conclusions. Thus,

```
ID <A>.
ID <B>.
 :
ID <Z>.
```

may be abbreviated

```
ID <A | B | ... | Z>.
```

Abbreviation 4: Multiple Conclusions and the Use of "&"

In many cases, multiple conclusions can be derived from identical premises. In these cases, the conclusions can be combined into a single production by separating the conclusions with "&". For example, the productions,

```
GOTO STM <GOTO id;>
   ← LABEL ε ρ(id)
LEGAL <GOTO id; : ρ>
   ← LABEL ε ρ(id).
```

may be abbreviated

```
GOTO STM <GOTO id;>  &  LEGAL <GOTO id; : ρ>
   ← LABEL ε ρ(id).
```

Abbreviation 5: <u>Repeated Strings and the Use of "*"</u>

In many cases compound conclusions are defined over identical strings. To prevent repeated use of identical strings, the symbol "*" is used. For example

```
GOTO STM <GOTO id;> & LEGAL <GOTO id; : ρ>
   → LABEL ε ρ(id).
```

may be abbreviated

```
GOTO STM <GOTO id;> & LEGAL<* : ρ>
   → LABEL ε ρ(id).
```

ABBREVIATION 6: <u>Permutations and the use of "o"</u>

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

```
ELEMENT ATTRIBUTE LIST
   <type-atr  scope-atr  storage-atr  |
    type-atr  storage-atr  scope-atr  |
    scope-atr  type-atr  storage-atr  |
    scope-atr  storage-atr  type-atr  |
    storage-atr  type-atr  scope-atr  |
    storage-atr  scope-atr  type-atr>.
```

may be abbreviated

```
ELEMENT ATTRIBUTE LIST
   <type-atr o scope-atr o storage-atr>
```

Abbreviation 7: <u>Sequences and the use of "..."</u>

Consider the following recursive definition of IDLIST

```
IDLIST <id | ℓ,id>.
```

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

$$\text{IDLIST} < id_1, id_2 \ldots , id_n >$$

In the sequel, the "..." notation will be used in several contexts. In some cases the corresponding recursive definition may be difficult to write. We shall ask some indulgence by the reader to accept that the corresponding unabbreviated productions can be written from such sequences.

Abbreviation 8: <u>Notation for Functions</u>

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.

(8a) Let $t_1, \ldots, t_n$ be terms and v be a variable. If

$$R<t_1 : \ldots : t_n : v>$$

is a premise occurring in a production containing exactly one other occurrence of v, then the premise can be deleted from the production if the other occurrence of v is replaced by the string

$$\underline{R}(t_1 : \ldots : t_n)$$

(8b) If

$$R<t_1 : \ldots : t_n : v>$$

is a conclusion occurring in a production defining the function, then the formula may be written as

$$\underline{R}(t_1 : \ldots : t_n) \equiv v$$

(8c) If

$$R<t_1 : \ldots : t_n : v>$$

is a premise referencing the result v of a function, then the formula may be written as

$$v \equiv \underline{R}(t_1 : \ldots : t_n)$$

For example, the productions

STRUCTURE SUCCESSOR <n id minor-struc-list>
← LEVEL NUM <minor-struc-list : n_1> &
n < n_1.

DECLARED TYPE <n : ρ : ARITH>.

IF STM <*IF* exp *THEN* exec-unit>
← DECLARED TYPE<exp : ρ : $type_1$> &
CONVERTIBLE<$type_1$: STRING>.

may be abbreviated

STRUCTURE SUCCESSOR <n id minor-struc-list>
← n < LEVEL NUM(minor-struc-list).

DECLARED TYPE(n : ρ) ≡ ARITH.

IF STM <*IF* exp *THEN* exec-unit>
← $type_1$ ≡ DECLARED TYPE(exp : ρ) &
CONVERTIBLE<$type_1$: STRING>.

5. THE COMPLETE SYNTAX OF PL.1

The complete syntax of PL.I is indeed complex, and is given in Appendix I. 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."

5.1 A Basic Overview

Conceptually, we shall view a syntactic environment ρ as a function mapping identifiers into attributes. The attributes are derived from the declarations of the identifiers. For example, consider the following explicit PL.I declarations

```
DECLARE A FIXED;
DECLARE B FLOAT;
DECLARE C POINTER EXTERNAL;
DECLARE D CHAR(5);
```

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

```
{ A →  ARITH, INTERNAL, AUTOMATIC,
  B →  ARITH, INTERNAL, AUTOMATIC,
  C →  POINTER, EXTERNAL, STATIC,
  D →  STRING, INTERNAL, BASED   }
```

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

(a) Given an identifier id and an environment ρ, we shall define a function APPLY such that APPLY(ρ:id) equals the derived attribute list associated with id in ρ. For example, using ρ above

APPLY(ρ:*D*) ≡ STRING, INTERNAL, BASED

For brevity, we shall abbreviate

```
APPLY(ρ:id)
```

as

```
ρ(id)
```

(b) Given two type attributes $type_1$, $type_2$ and an operator, we shall define a function RESULT TYPE that yields the data type obtained when the operator is applied to two arguments of $type_1$ and $type_2$, e.g.

```
RESULT TYPE(ARITH : op : ARITH)   ≡  ARITH
   ← PLUS OP<op>.

RESULT TYPE(ARITH : op : ARITH)   ≡  STRING
   ← EQUALITY OP<op>.
```

(c) Given an expression exp and a syntactic environment ρ, we shall define a function DECLARED TYPE that computes the declared type of exp in ρ. For example, given the previous environment ρ

```
DECLARED TYPE(A : ρ)
   ≡ ARITH

DECLARED TYPE(1 : ρ)
   ≡ ARITH

DECLARED TYPE("SNOW" : ρ)
   ≡ STRING

DECLARED TYPE(A + 1 : ρ)
   ≡ RESULT TYPE(DECLARED TYPE(A : ρ)  :  +
                 DECLARED TYPE(1 : ρ))
   ≡ RESULT TYPE(ARITH : + : ARITH)
   ≡ ARITH
```

(e) Finally, we shall define a predicate CONVERTIBLE comprising all pairs of declared types $type_1$ and $type_2$ such that $type_1$ can be converted into $type_2$ according to the rules of PL/I. For example, the pairs

<STRING : STRING> <ARITH : STRING>

are members of CONVERTIBLE, whereas

<STRING : POINTER> <LABEL : ARITH>

are not.

5.2 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 → POINTER, INTERNAL, AUTOMATIC
A → ARITH, INTERNAL, AUTOMATIC }

This program is ruled out in the production [01] of Appendix 1 by the premise

DIFF IDLIST<<u>DOMAIN</u>(ρ)>

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

```
Q:  PROCEDURE;
    DECLARE A FLOAT;
       A = 0.0;
    L: A = A + 1;
       GOTO M;
END Q;
```

The environment for the program is

$$\{ A \to \text{ARITH, INTERNAL, AUTOMATIC}, \quad L \to \text{LABEL, INTERNAL, AUTOMATIC}, \quad M \to \text{ARITH, INTERNAL, AUTOMATIC} \}$$

Note that M is implicitly declared to be arithmetic since it is not explicitly or contextually declared. This program is ruled out by production [08]

$$\text{GOTO STM}\langle \textit{GOTO}\ \text{id};\rangle \ \&\ \text{LEGAL}\langle *:\rho\rangle \to \text{LABEL} \in \rho(\text{id}).$$

since for the statement "*GOTO M;*"

$$\rho(M) \equiv \text{ARITH, INTERNAL, AUTOMATIC}$$
$$\text{LABEL} \notin \text{ARITH, INTERNAL, AUTOMATIC}$$

Consider also the following proposed programs

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

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

The environments ρ_P and ρ_Q for P and Q are

$$\rho_P \equiv \{A \to \text{ARITH, INTERNAL, BASED}, \quad B \to \text{ARITH, INTERNAL, AUTOMATIC}, \quad R \to \text{POINTER,IINTERNAL, AUTOMATIC} \}$$

$$\rho_Q \equiv \{A \to \text{ARITH, INTERNAL, BASED}, \quad B \to \text{POINTER, INTERNAL, AUTOMATIC}, \quad R \to \text{POINTER, INTERNAL, AUTOMATIC} \}$$

The production [07] for assignment statements is

$$
\begin{aligned}
&\text{ASGT STM} \langle \text{ref} := \text{exp};\rangle \quad \& \quad \text{LEGAL}\langle * : \rho\rangle \\
&\leftarrow \text{type}_1 \equiv \underline{\text{DECLARED TYPE}}(\text{ref} : \rho) \quad \& \\
&\quad \text{type}_2 \equiv \underline{\text{DECLARED TYPE}}(\text{exp} : \rho) \quad \& \\
&\quad \text{CONVERTIBLE}\langle \text{type}_2 : \text{type}_1\rangle .
\end{aligned}
$$

The instances of the corresponding premises for the statement "$B = R;$" in P and Q are

$$
\begin{aligned}
\text{type}_1 &\equiv \underline{\text{DECLARED TYPE}}(B:\rho_P) \\
&\equiv \text{ARITH} \\
\text{type}_2 &\equiv \underline{\text{DECLARED TYPE}}(R:\rho_P) \\
&\equiv \text{POINTER}
\end{aligned}
$$

$$
\begin{aligned}
\text{type}_1 &\equiv \underline{\text{DECLARED TYPE}}(B:\rho_Q) \\
&\equiv \text{POINTER} \\
\text{type}_2 &\equiv \underline{\text{DECLARED TYPE}}(R:\rho_Q) \\
&\equiv \text{POINTER}
\end{aligned}
$$

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

5.3 <u>Computation of Environments and Block Structure</u>

Given environments ρ and ρ', we may define the functions <u>OVERRIDE</u> and <u>PLUS</u> such that

(1) <u>PLUS</u>(ρ:ρ') = the environment computed by "adding" ρ to ρ'

(2) <u>OVERRIDE</u>(ρ:ρ') = the environment computed from ρ' by "overwriting" the identifiers declared in ρ.

We shall abbreviate (1) above as

(1') $\rho + \rho'$

PL.1 allows three varieties of declarations. Accordingly, three functions, <u>EXPLICIT ENV</u>, <u>CONTEXTUAL ENV</u>, and <u>IMPLICIT ENV</u> are defined. These functions map a PL.1 unit sequence into environments corresponding to the identifiers that are declared explicitly, contextually, or implicitly. Since contextual declarations are only applied to identifiers not declared explicitly, and implicit declarations are only applied to identifiers not declared explicitly or contextually, the resulting environment components ρ_{exp}, ρ_{cont}, and ρ_{imp} defined over a unit sequence

$$\text{unit-seq} \equiv \text{unit}_1\ \text{unit}_2\ \ldots\ \text{unit}_n$$

are:

$$\rho_{exp} \equiv \underline{\text{EXPLICIT ENV}}(\text{unit-seq})$$
$$\rho_{cont} \equiv \underline{\text{CONTEXTUAL ENV}}(\text{unit-seq} : \rho_{exp})$$
$$\rho_{imp} \equiv \underline{\text{IMPLICIT ENV}}(\text{unit-seq} : \rho_{exp} + \rho_{cont})$$

For example, consider the program

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

Here the identifier *A* is declared explicitly, hence

$$\rho_{exp} \equiv \{A \rightarrow \text{ARITH, INTERNAL, BASED}\}$$

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

$$\rho_{cont} \equiv \{Q \rightarrow \text{POINTER, INTERNAL, AUTOMATIC}, \; R \rightarrow \text{POINTER, INTERNAL, AUTOMATIC}\}$$

One remaining identifier *B* is declared implicitly, hence

$$\rho_{imp} \equiv \{B \rightarrow \text{ARITH, INTERNAL, AUTOMATIC}\}$$

The total environment ρ is thus given as

$$\rho \equiv \{A \rightarrow \text{ARITH, INTERNAL, BASED,} \\ Q \rightarrow \text{POINTER, INTERNAL, AUTOMATIC,} \\ R \rightarrow \text{POINTER, INTERNAL, AUTOMATIC,} \\ B \rightarrow \text{ARITH, INTERNAL, AUTOMATIC}\}$$

Using the total environment ρ, each of the executable 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 outside an inner block and not declared locally has a scope that includes the inner block. To reflect these requirements, the environment for a block

$$\textit{BEGIN;}\ \ \text{unit}_1\ \text{unit}_2\ \ldots\ \text{unit}_n\ \ \textit{END;}$$

nexted within an environment ρ is defined as

$$\rho' \equiv \underline{\text{OVERRIDE}}(\rho_{\text{local}} : \rho)$$

where

$$\text{unit-seq} \equiv \text{unit}_1\ \text{unit}_2\ \ldots\ \text{unit}_n$$

$$\rho_{\text{local}} \equiv \underline{\text{EXPLICIT ENV}}(\text{unit-seq})$$

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

5.4 The Complete Production System of PL.1

A definition of the complete syntax of PL.1 is given in Appendix 1. 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 Appendix 1 and find the appropriate information.

One important issue not discussed here is the use of structures in PL.1. The main impact of structures is in the productions for <u>RESULT TYPE</u>, which is defined for structured types as well as atomic types.

6. THE TRANSLATION OF LAMBDA-CALCULUS EXPRESSION INTO FORM FORM

As a second example, we consider the mapping of λ-calculus expressions into normal form, which is given in Appendix 2. In this mapping the full power of Production Systems must be employed, since the set of pairs

<λ-calculus-exp : normal-form>

is a recursively enumerable set that is not recrusive. That is, given a pair $<exp_1 : exp_2>$ of well-formed lambda expressions, it is <u>not</u> decideable whether exp_2 is a normal form of exp_1. The primary value of the productions for this mapping lies in a precise definition of the λ-calculus substitution rule 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 id be an identifier representing a formal parameter, exp be the corresponding actual parameter, and b be the λ-expression representing the body of the declared procedure. Then the value b' such that

<u>SUBST</u>(id : exp : b) ≡ b'

represents the body derived from b when exp is "substituted" for id.

In particular, consider the productions

<u>SUBST</u>(id : exp : id) ≡ exp

<u>SUBST</u>(id : exp : id_1) ≡ id_1
← DIFF ID<id : id_1>.

If b is an identifier that is identical to the formal parameter, than b' is exp. If b is an identifier that is different from the formal parameter, than b' is b.

For example, consider the procedure declaration

```
PROCEDURE  F(X);
  X := X + Y;
END F
```

and the call

```
F(A);
```

The execution of *F(A)* results in the execution of

```
A := A + Y;
```

That is, exp = *X* is replaced by A, but exp = *Y* is left alone.

Next consider the production

$$
\begin{aligned}
&\underline{\text{SUBST}}(\text{id} : \text{exp} : \lambda \text{id}_1.\text{exp}_1) \equiv \lambda \text{id}_2.\text{exp}_1'' \\
&\quad \leftarrow \text{DIFF ID}<\text{id} : \text{id}_1> \quad \& \\
&\quad \text{id}_1 \in \underline{\text{FREE IDS}}(\text{exp}) \quad \& \\
&\quad \text{DIFF ID}<\text{id}_1 : \text{id}_2> \quad \& \\
&\quad \text{id}_2 \notin \underline{\text{FREE IDS}}(\text{exp}) \quad \& \\
&\quad \text{id}_2 \notin \underline{\text{FREE IDS}}(\text{exp}_1) \quad \& \\
&\quad \text{exp}_1' \equiv \underline{\text{SUBST}}(\text{id}_1 : \text{id}_2 : \text{exp}_1) \quad \& \\
&\quad \text{exp}_1'' \equiv \underline{\text{SUBST}}(\text{id} : \text{exp} : \text{exp}_1').
\end{aligned}
$$

For 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 id ≡ *Y* by exp ≡ *X + A* in the nested BEGIN-END block. Here the local variable id_1 ≡ *X* is contained in the list *X,A* of free identifiers of exp ≡ *X + A*. Let id_2 ≡ *Q* be an identifier, which is different from id = *Y* and is not contained in the list of free identifiers of exp or in the free identifiers of the block body. Replacing *X* in the *BEGIN-END* block by *Q*, we obtain

```
exp1' = BEGIN  INTEGER Q;
               Q := 1;
               B := Q + Y
        END
```

Having thus changed the local variable, the replacement of *Y* by *X + A* in the body of the revised BEGIN-END block may be performed without considering a possible clash of identifiers.

7. DISCUSSION

Production Systems have placed under a single framework the complete definition of the syntax and translation of a programming language. 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 effort was to tailor the formalism to clear language definitions. Accordingly, the notation, abbreviations, and conceptual view of production systems have undergone many stages of evolution.

Besides simplicity of the formal system, human readability has been a major goal. Necessarily, I have used my personal discretion in what constitutes a readable notation. There exist no known metrics for measuring simplicity and readability, for 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 language? It is presumably more natural or more concise to define an algorithm in one language than another.

Admittedly, the definition of PL/.1 is not short, and is quite complex. It is my contention that a good definition mechanism displays those areas where a language is overly complex. The PL/I conventions for default attributes and for contextual and implicit declarations are a major source of complexity in the Production System given here. The complexity of the formal definition is an argument against the utility of these features.

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 a legal program. While in most examples given here the class of illegal strings is quite apparent, it would certainly be desirable to find some constraints on the form of productions to limit their definition to decidable sets. This would guarantee that the Production Systems could be used as the basis for a language processor and also allow one to refer to the complement of defined sets. A recent work by Schuler [11] may offer a solution to this problem.

Production Systems could be used to specify definitions and string transformations much *different* from those given here. Outside of the examples given here, and a few others that I have attempted, little experience other than the definition of syntax and translation with Production Systems has been obtained. Nevertheless, based on the clarity of the definitions given here, I believe that Production Systems provide a solid formal system for readable and precise definitions.

ACKNOWLEDGMENTS

To Michael Marcotty, whose mature views of human engineering and formal definition have significantly influenced this work.

And to Edward Glaser, Dana Scott, John Wozencraft, Michael Flynn, Louis Chmura, Christopher Wadsworth, and Calvin Mooers, who in various ways contributed to the development of Production Systems over the past years.

REFERENCES

1. A. Church, "The Calculi of Lambda-Conversion," Annals of Mathematical Studies, Number 6, Princeton University Press, Princeton, NJ, 1941.

2. H. B. Curry, and R. Feys, Combinatory Logic, vol. I. Amsterdam: North Holland Publishing Co., 1958.

3. J. J. Donovan, "Investigations in Simulation and Simulation Languages," Yale University, New Haven, CT, 1966.

4. J. J. Donovan, and H. F. Ledgard, "A Formal System for the Specification of the Syntax and Translation of Computer Languages," in AFIPS, Proc. of the 1967 Fall Joint Computer Conference, 1967, Volume 31.

5. H. F. Ledgard, "A Formal System for Defining the Syntax and Semantics of Computer Languages," MAC-TR-60, M.I.T., Cambridge, MA, 1969.

6. H. F. Ledgard, "Production Systems: Or Can We Do Better than BNF?", Communications of the ACM, Feb., 1974.

7. P. Lucas, and K. Walk, "On the Formal Description of PL/I," Annual Review of Automatic Programming, vol. 6, no. 3, Pergamon Press, 1969.

8. M. Marcotty, H. Ledgard, and G. Bochmann, "A Sampler of Formal Definitions," Computing Surveys, June 1976.

9. P. Naur, "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, vol. 6, no. 1, pp. 1-23, 1963.

10. E. L. Post, "Formal Reductions of the General Combinatorial Decision Problems," American Journal of Mathematics, vol. 65, pp. 197-215, 1943.

11. P. F. Schuler, "Weakly Context-Sensitive Languages," Acta Informatica, vol. 3, pp. 155-170, 1974.

12. R. M. Smullyan, "Theory of Formal Systems," Annals of Mathematical Studies, no. 47, Princeton University Press, Princeton, NJ, 1961.

13. BASIS/1-12, European Computer Manufacturers Association, and American National Standards Institute, July 1974.

APPENDIX 1: PRODUCTION SYSTEM SPECIFYING THE COMPLETE SYNTAX OF PL.1

[a. *Programs and Units*]

[01] prog PROGRAM <id: *PROCEDURE;* unit$_1$ unit$_2$ ... unit$_n$ *END* id;>

← unit-seq ≡ unit$_1$ unit$_2$ ... unit$_n$ &

ρ_{exp} ≡ EXPLICIT ENV(unit-seq) &

[*Get environment from explicit declarations*]

ρ_{cont} ≡ CONTEXTUAL ENV(unit-seq : ρ_{exp}) &

[*Get contextual environment for identifiers not in* ρ_{exp}]

ρ_{imp} ≡ IMPLICIT ENV(unit-seq : $\rho_{exp} + \rho_{cont}$) &

[*Get implicit environment for identifiers not in* ρ_{exp} *or* ρ_{cont}]

$\rho \equiv \rho_{exp} + \rho_{cont} + \rho_{imp}$ &

[*Get total environment for the program*]

DIFF IDLIST<DOMAIN(ρ)>

[*There must be no multiple declarations*]

LEGAL<unit$_1$:ρ> & LEGAL<unit$_2$:ρ> & ... & LEGAL<unit$_n$:ρ>.

[*Each unit must be legal in* ρ]

[02] unit UNIT <exec-unit | dcl-unit>.

[03] exec-unit EXECUTABLE UNIT <id: exec-unit> & LEGAL<* : ρ>

← LEGAL<exec-unit : ρ>.

[04] exec-unit EXECUTABLE UNIT <stm>.

[05] dcl-unit DECLARATIVE UNIT <id: dcl-unit> & LEGAL<* : ρ>

← LEGAL<dcl-unit : ρ>.

[06] dcl-unit DECLARATIVE UNIT <dcl>.

[b. *Imperative Statements*]

[07] stm ASGT STM <ref = exp;> & LEGAL<* : ρ>

← type$_1$ ≡ DECLARED TYPE(ref : ρ) &

type$_2$ ≡ DECLARED TYPE(exp : ρ) &

CONVERTIBLE<type$_2$: type$_1$>.

[*The type of the expression must be convertible to the type of the left hand side reference*]

[08] stm GOTO STM <*GOTO* id;> & LEGAL<* : ρ>

← LABEL ε ρ(id).

[*The identifier id must correspond to a label*]

[09] stm IF STM <*IF* exp *THEN* exec-unit> & LEGAL * : ρ>

← type$_1$ ≡ DECLARED TYPE(exp : ρ) &

CONVERTIBLE<type$_1$: STRING>.

[*The expression must be convertible to a (bit) string*]

[10] stm ALLOCATE STM <*ALLOCATE* id;> & LEGAL<* : ρ>

→ BASED ∈ ρ(id).

[*id must be declared as BASED*]

[11] stm ALLOCATE STM <*ALLOCATE* id$_1$ *SET* (id$_2$);> & LEGAL<* : ρ>

→ BASED ∈ ρ(id$_1$) & POINTER ∈ ρ(id$_2$).

[*id$_1$ must be declared as BASED and id$_2$ as POINTER*]

[12] stm FREE STM <*FREE* id;> & LEGAL <* : ρ>

→ BASED ∈ ρ(id).

[*id must be declared as BASED*]

[13] stm FREE STM <*FREE* id$_1$ → id$_2$;> & LEGAL<* : ρ>

→ POINTER ∈ ρ(id$_1$) & BASED ∈ ρ(id$_2$).

[14] stm RETURN STM <*RETURN;*> & LEGAL<* : ρ>.

[15] stm BLOCK <*BEGIN;* unit$_1$ unit$_2$ ... unit$_n$ *END*> & LEGAL<* : ρ>

→ unit-seq ≡ unit$_1$ unit$_2$ ... unit$_n$ &

ρ$_{local}$ ≡ <u>EXPLICIT ENV</u>(unit-seq) &

[*Get env for local declarations*]

DIFF IDLIST<<u>DOMAIN</u>(ρ$_{local}$)> &

[*The locally declared identifiers must each be different*]

ρ' ≡ <u>OVERRIDE</u>(ρ$_{local}$:ρ) &

[*Local declarations override those in the outer block*]

LEGAL<unit$_1$:ρ'> & ... & LEGAL<unit$_n$:ρ>.

[*Each local unit must be legal in the new environment*]

[c. *Declarations*]

[16] dcl ELEMENT DECLARATION <*DECLARE* id elem-atr-list;>.

[17] dcl STRUCTURE DECLARATION <*DECLARE* 1 id struc-atr-list, minor-struc-list;>.

[18] elem-atr-list ELEMENT ATTRIBUTE LIST <type-atr ∘ scope-atr ∘ storage-atr>

→ (scope-atr ≠ EXTERNAL | storage-atr = STATIC).

[*EXTERNAL identifiers cannot be AUTOMATIC or BASED*]

[19] struc-atr-list STRUCTURE ATTRIBUTE LIST <scope-atr ∘ storage-atr>

→ (scope-atr ≠ EXTERNAL | storage-atr = STATIC).

[20] minor-struc-list MINOR STRUCTURE LIST <n$_1$ id$_1$ succ$_1$, n$_2$ id$_2$ succ$_2$ ... , n$_k$ id$_k$ succ$_k$>

→ DIFF IDLIST <id$_1$, id$_2$ ... ,id$_k$> &

[*All minor structures at a given level must have different names*]

$n_1 \ge n_2$ & $n_2 \ge n_3$ & ... & $n_{k-1} \ge n_k$.

[*Level numbers at a given level must be equal or increasing*]

[21] succ STRUCTURE SUCCESSOR <type-atr>.

[22] succ STRUCTURE SUCCESSOR <n id minor-struc-list>

→ n < <u>LEVEL NUM</u>(minor-struc-list).

[*Major structure level numbers must be less than each component level number*]

[23] type-atr DATA TYPE ATR <Λ | *FIXED* | *FLOAT* | *BIT(n)* | *CHAR(n)*>.

[24] scope-atr SCOPE ATR <Λ | *EXTERNAL* | *EXTERNAL*>.

[25] storage-atr STORAGE ATR <Λ | *STATIC* | *AUTOMATIC* | *BASED(id)*>.

[d. *Expressions, Atomic Components, and Declared Types*]

[26] exp EXPRESSION <n> & DECLARED TYPE(* : ρ) ≡ ARITH.

[27] exp EXPRESSION <"c_1 c_2 ... c_n"> & DECLARED TYPE(* : ρ) ≡ STRING.

[28] exp EXPRESSION <*NULL*> & DECLARED TYPE(* : ρ) ≡ POINTER.

[29] exp EXPRESSION <*ADDR(ref)*> & DECLARED TYPE(* : ρ) ≡ POINTER.
← LABEL ≠ DECLARED TYPE(ref : ρ).

[30] exp EXPRESSION <ref>.

[31] exp EXPRESSION <exp_1 op exp_2> & DECLARED TYPE(* : ρ) ≡ $type_r$
← $type_1$ ≡ DECLARED TYPE(exp_1 : ρ) &
$type_2$ ≡ DECLARED TYPE(exp_2 : ρ) &
$type_r$ ≡ RESULT TYPE($type_1$: op : $type_2$).

[32] ref REFERENCE <id> & DECLARED TYPE(* : ρ) ≡ $type_1$
← $type_1$ ε ρ(id).

[33] ref REFERENCE <id_1 → id_2> & DECLARED TYPE(* : ρ) ≡ $type_2$
← POINTER ε ρ(id_1) & BASED ε ρ(id_2)
$type_2$ ≡ DECLARED TYPE(id_2 : ρ).

[34] ref REFERENCE <id_1.id_2 ... .id_n> & DECLARED TYPE(* : ρ) ≡ $type_n$
← ρ_1 ≡ ρ(id_1) & ρ_2 ≡ ρ_1(id_2) & ... & $type_n$ ≡ ρ_{n-1}(id_n).

[35] op OPERATOR <op>
← (ARITH OPERATOR<op> | COMPARISON OPERATOR<op>).

[36] op ARITH OPERATOR < + >.

[37] op COMPARISON OPERATOR < ≤ | = | ≥ >.

[38] id IDENTIFIER <$\ell_1 \ell_2 \ldots \ell_n$>.

[39] ℓ LETTER <*A* | *B* | ... | *Z*>.

[40] n NUMBER <$d_1 d_2$ ... d_n>.

[41] d DIGIT <*0* | *1* | ... | *9*>.

[42] c CHARACTER <ℓ | d>.

[e. *Computations of Environments*]

[*Environment from Explicit Declarations*]

[43] EXPLICIT ENV(unit$_1$ ... unit$_n$) ≡ EXPLICIT ENV(unit$_1$) + ... + EXPLICIT ENV(unit$_n$).

[44] EXPLICIT ENV(id: exec-unit) ≡ {id → LABEL}.

[45] EXPLICIT ENV(exec-unit) ≡ Λ.

[46] EXPLICIT ENV(id: dcl-unit) ≡ {id → LABEL} + EXPLICIT ENV(dcl-unit).

[47] EXPLICIT ENV(DECLARE id elem-atr-list;) ≡ {id → DECLARED ATTRIBUTES(elem-atr-list)}.

[48] EXPLICIT ENV(DECLARE 1 id struc-atr-list, minor-struc-list;)
≡ {id → DECLARED ATTRIBUTES(struc-atr-list), EXPLICIT ENV(minor-struc-list)}.

[49] EXPLICIT ENV(n_1 id_1 $succ_1$, ... ,n_k id_k $succ_k$)
≡ EXPLICIT ENV(n_1 id_1 $succ_1$) + ... + EXPLICIT ENV(n_k id_k $succ_k$).

[50] EXPLICIT ENV(n id type-atr)
≡ {id → DECLARED TYPE ATR(type-atr)}.

[51] EXPLICIT ENV(n id minor-struc-list)
≡ {id → EXPLICIT ENV(minor-struc-list)}.

[52] DECLARED ATTRIBUTES(type-atr ∘ scope-atr ∘ storage-atr)
≡ DECLARED TYPE ATR(type-atr), DECLARED SCOPE ATR(scope-atr), DECLARED STORAGE ATR(storage-atr).

[53] DECLARED TYPE ATR(Λ) ≡ ARITH.

[54] DECLARED TYPE ATR(*FIXED* | *FLOAT*) ≡ ARITH.

[55] DECLARED TYPE ATR(*BIT*(n) | *CHAR*(n)) ≡ STRING.

[56] DECLARED TYPE ATR(*POINTER*) ≡ POINTER.

[57] DECLARED SCOPE ATR(Λ) ≡ INTERNAL.

[58] DECLARED SCOPE ATR(*INTERNAL*) ≡ INTERNAL.

[59] DECLARED SCOPE ATR(*EXTERNAL*) ≡ EXTERNAL.

[60] DECLARED STORAGE ATR(Λ) ≡ AUTOMATIC.

[61] DECLARED STORAGE ATR(*AUTOMATIC*) ≡ AUTOMATIC.

[62] DECLARED STORAGE ATR(*STATIC*) ≡ STATIC.

[63] DECLARED STORAGE ATR(*BASED*(id)) ≡ BASED.

[Environment from contextual declarations]

[64] $\underline{\text{CONTEXTUAL ENV}}(\text{unit}_1 \ldots \text{unit}_n : \rho) \equiv \rho_1 + \ldots + \rho_n$
$\leftarrow \rho_1 \equiv \underline{\text{CONTEXTUAL ENV}}(\text{unit}_1 : \rho)$ &
$\ldots$ & $\rho_n \equiv \underline{\text{CONTEXTUAL ENV}}(\text{unit}_n : \rho_{n-1})$.

[65] $\underline{\text{CONTEXTUAL ENV}}(\text{id}\text{: unit} : \rho) \equiv \underline{\text{CONTEXTUAL ENV}}(\text{unit} : \rho)$

[66] CONTEXTUAL ENV(*GOTO* id; | *RETURN;* | *ALLOCATE* id; | *FREE ID;* : ρ) ≡ Λ.

[67] CONTEXTUAL ENV(*ALLOCATE* id_1 *SET*(id_2)*;* : ρ) ≡ Λ
← $\text{id}_2 \in$ DOMAIN(ρ).

[68] CONTEXTUAL ENV(*ALLOCATE* id_1 *SET*(id_2)*;* : ρ) ≡ {id_2 → POINTER,INTERNAL,AUTOMATIC}
← $\text{id}_2 \notin$ DOMAIN(ρ).

[69] CONTEXTUAL ENV(*FREE* $\text{id}_1 \rightarrow \text{id}_2$*;* : ρ) ≡ Λ
← $\text{id}_2 \in$ DOMAIN(ρ).

[70] CONTEXTUAL ENV(*FREE* $\text{id}_1 \rightarrow \text{id}_2$*;* : ρ) ≡ {$\text{id}_2$ → POINTER,INTERNAL,AUTOMATIC}
← $\text{id}_2 \notin$ DOMAIN(ρ).

[71] CONTEXTUAL ENV(*IF* exp *THEN* exec-unit : ρ) ≡ CONTEXTUAL ENV(exp : ρ)
+ CONTEXTUAL ENV(exec-unit : ρ).

[72] CONTEXTUAL ENV(ref = exp*;* : ρ) ≡ CONTEXTUAL ENV(ref : ρ)
+ CONTEXTUAL ENV(exp : ρ).

[73] CONTEXTUAL ENV(*BEGIN;* $\text{unit}_1 \ldots \text{unit}_n$ *END;* : ρ)
≡ CONTEXTUAL ENV(unit-seq : ρ')
← unit-seq ≡ $\text{unit}_1 \ldots \text{unit}_n$ &
ρ_{local} ≡ EXPLICIT ENV(unit-seq) &
ρ' ≡ OVERRIDE(ρ_{local} : ρ).
[Explicit local declarations override declarations of the outer block]

[74] CONTEXTUAL ENV(*DECLARE* id elem-atr-list*;* : ρ)
≡ CONTEXTUAL ENV(elem-atr-list : ρ).

[75] CONTEXTUAL ENV(*DECLARE 1* id struc-atr-list, minor-struc-list*;* : ρ)
≡ CONTEXTUAL ENV(struc-atr-list : ρ).

[76] CONTEXTUAL ENV(type-atr ∘ scope-atr ∘ storage-atr : ρ)
≡ {id → POINTER,INTERNAL,AUTOMATIC}
← scope-atr = BASED(id) & id ∉ DOMAIN(ρ).

[77] CONTEXTUAL ENV(type-atr ∘ scope-atr ∘ storage-atr : ρ)
≡ Λ
← scope-atr ≠ BASED(id).

[78] CONTEXTUAL ENV(exp_1 op exp_2 : ρ)
≡ CONTEXTUAL ENV(exp_1 : ρ) + CONTEXTUAL ENV(exp_2 : ρ).

[79] CONTEXTUAL ENV(n | "$c_1 c_2 \ldots c_n$" | *NULL* : ρ)
≡ Λ.

[80] CONTEXTUAL ENV(id | $\text{id}_1.\text{id}_2 \ldots .\text{id}_n$: ρ) ≡ Λ.

[81] CONTEXTUAL ENV(*ADDR(*ref*)* : ρ) ≡ CONTEXTUAL ENV(ref : ρ).

[82] CONTEXTUAL ENV($\text{id}_1 \rightarrow \text{id}_2$: ρ) ≡ Λ
← $\text{id}_1 \in$ DOMAIN(ρ).

[83] CONTEXTUAL ENV($\text{id}_1 \rightarrow \text{id}_2$: ρ) ≡ {id_1 → POINTER,INTERNAL,AUTOMATIC}
← $\text{id}_1 \notin$ DOMAIN(ρ).

[*Environment from Implicit Declarations*]

[84] $\underline{\text{IMPLICIT ENV}}(\text{unit}_1 \ldots \text{unit}_n : \rho) \equiv \rho_1 + \ldots + \rho_n$
$\leftarrow \rho_1 \equiv \underline{\text{IMPLICIT ENV}}(\text{unit}_1 : \rho)$ &
$\ldots$ & $\rho_n \equiv \underline{\text{IMPLICIT ENV}}(\text{unit}_n : \rho_{n-1})$.

[85] $\underline{\text{IMPLICIT ENV}}(\text{id : unit} : \rho) \equiv \underline{\text{IMPLICIT ENV}}(\text{unit} : \rho)$.

[86] $\underline{\text{IMPLICIT ENV}}(\text{dcl} : \rho) \equiv \Lambda$.

[87] $\underline{\text{IMPLICIT ENV}}(\textit{RETURN;} : \rho) \equiv \Lambda$.

[88] $\underline{\text{IMPLICIT ENV}}(\textit{GOTO}\ \text{id;} \mid \textit{ALLOCATE}\ \text{id;} \mid \textit{FREE}\ \text{id;} : \rho)$
$\equiv \underline{\text{IMPLICIT ENV}}(\text{id}:\rho)$.

[89] $\underline{\text{IMPLICIT ENV}}(\textit{ALLOCATE}\ \text{id}_1\ \textit{SET}(\text{id}_2); \mid \textit{FREE}\ \text{id}_2 \rightarrow \text{id}_1; : \rho)$
$\equiv \underline{\text{IMPLICIT ENV}}(\text{id}_1:\rho)$.

[90] $\underline{\text{IMPLICIT ENV}}(\text{ref} = \text{exp}; : \rho)$
$\equiv \underline{\text{IMPLICIT ENV}}(\text{ref} : \rho) + \underline{\text{IMPLICIT ENV}}(\text{exp} : \rho)$.

[91] $\underline{\text{IMPLICIT ENV}}(\textit{BEGIN;}\ \text{unit}_1 \ldots \text{unit}_n\ \textit{END;} : \rho)$
$\equiv \underline{\text{IMPLICIT ENV}}(\text{unit seq} : \rho')$
$\leftarrow \text{unit-seq} \equiv \text{unit}_1 \ldots \text{unit}_n$ &
$\rho_{\text{local}} \equiv \underline{\text{EXPLICIT ENV}}(\text{unit-seq})$ &
$\rho' \equiv \underline{\text{OVERRIDE}}(\rho_{\text{local}} : \rho)$.

[92] $\underline{\text{IMPLICIT ENV}}(\text{exp}_1\ \text{op}\ \text{exp}_2 : \rho)$
$\equiv \underline{\text{IMPLICIT ENV}}(\text{exp}_1 : \rho) + \underline{\text{IMPLICIT ENV}}(\text{exp}_2 : \rho)$.

[93] $\underline{\text{IMPLICIT ENV}}(\text{n} \mid \text{"}c_1\ c_2 \ldots c_n\text{"} \mid \textit{NULL} \mid \text{id}_1. \ldots .\text{id}_n : \rho)$
$\equiv \Lambda$

[94] $\underline{\text{IMPLICIT ENV}}(\textit{ADDR}(\text{ref}) : \rho)$
$\equiv \underline{\text{IMPLICIT ENV}}(\text{ref} : \rho)$

[95] $\underline{\text{IMPLICIT ENV}}(\text{id}_1 \rightarrow \text{id}_2 : \rho)$
$\equiv \underline{\text{IMPLICIT ENV}}(\text{id}_2 : \rho)$

[96] $\underline{\text{IMPLICIT ENV}}(\text{id} : \rho)$
$\equiv \Lambda$
$\leftarrow \text{id} \in \underline{\text{DOMAIN}}(\rho)$

[97] $\underline{\text{IMPLICIT ENV}}(\text{id} : \rho)$
$\equiv \{\text{id} \rightarrow \text{ARITH, INTERNAL, AUTOMATIC}\}$
$\leftarrow \text{id} \notin \underline{\text{DOMAIN}}(\rho)$

[*Computation of Result Types*]

[*Symmetry Rule*]

[98] RESULT TYPE($type_1$: op : $type_2$) ≡ RESULT TYPE($type_2$: op : $type_1$).

[*Operations on Scalars*]

[99] RESULT TYPE($type_1$: op : $type_2$) ≡ ARITH
← PLUS OP<op> & ($type_1$ = ARITH | $type_1$ = STRING)
& ($type_2$ = ARITH | $type_2$ = STRING).

[100] RESULT TYPE(POINTER : op : POINTER) ≡ STRING
←. EQUALITY OP<op>.

[101] RESULT TYPE($type_1$: op : $type_2$) ≡ STRING
← COMPARISON OP<op> & ($type_1$ ≡ ARITH | $type_1$ ≡ STRING)
& ($type_2$ ≡ ARITH | $type_2$ ≡ STRING).

[*Addition and Comparison of Structures*]

[102] RESULT TYPE({id_1 → $type_1$, ... ,id_n → $type_n$} : op : {id'_1 → $type'_1$, ... ,id'_n → $type'_n$})
≡ {id_1 → $type''_1$, ... ,id_n → $type''_n$}
← PLUS OP<op> & RESULT TYPE($type_1$: + : $type'_1$) ≡ $type''_1$
... & RESULT TYPE($type_n$: + : $type'_n$) ≡ $type''_n$.

[103] RESULT TYPE({id_1 → $type_1$, ... ,id_n → $type_n$} : op : {id'_1 → $type'_1$, ... ,id'_n → $type'_n$})
≡ STRING
← COMPARISON OP<op> RESULT TYPE($type_1$: op : $type'_1$) ≡ STRING
... RESULT TYPE($type_n$: op : $type'_n$) ≡ STRING.

[104] RESULT TYPE(type : op : {id_1 → $type_1$, ... ,id_n → $type_n$})
≡ RESULT TYPE({id_1 → type, ... ,id_n → type} : op : {id_1 → $type_1$, ... ,id_n → $type_n$}).

[*g. Miscellaneous Predicates*]

[105] type DERIVED TYPE ATR <ARITH | POINTER | STRING |
{id_1 → $type_1$, ... ,id_n → $type_n$}>.

[106] derived-atr DERIVED ATR <INTERNAL | EXTERNAL | AUTOMATIC | STATIC | BASED | type>.

[107] derived-atr-list DERIVED ATR LIST <derived-atr_1, ... ,derived-atr_n>.

[108] ENVIRONMENT <Λ | {id_1 → $\text{derived-atr-list}_1$, ... ,$id_n$ → $\text{derived-atr-list}_n$} >.

[109] DOMAIN <ρ> ≡ Λ
← ρ = Λ.

[110] $\underline{\text{DOMAIN}}\ \langle \rho \rangle \equiv \text{id}_1, \ldots, \text{id}_n$

$\leftarrow \rho . = \{ \text{id}_1 \rightarrow \text{derived-atr-list}_1, \ldots, \text{id}_n \rightarrow \text{derived-atr-list}_n \}.$

[111] $\underline{\text{APPLY}}\ \langle \rho : \text{id} \rangle \equiv \text{ERROR}$

$\leftarrow \text{id} \notin \text{DOMAIN}(\rho).$

[112] $\underline{\text{APPLY}}\ \langle \rho : \text{id} \rangle \equiv \text{derived-atr-list}$

$\leftarrow \rho = \{\ldots \text{id} \rightarrow \text{derived-atr-list} \ldots\}.$

[113] $\underline{\text{PLUS}}(\rho : \Lambda) \equiv \rho.$

[114] $\underline{\text{PLUS}}(\rho : \rho') \equiv \{x, y\}$

$\leftarrow \rho = \{x'\} \ \& \ \rho' = \{y\}.$

[115] $\underline{\text{DELETE}}(\rho : \Lambda) \equiv \Lambda.$

[116] $\underline{\text{DELETE}}(\rho : \text{id}) \equiv \rho_1 + \rho_2$

$\leftarrow \rho = \rho_1 + \{\text{id} \rightarrow \text{derived-atr-list}\} + \rho_2.$

[117] $\underline{\text{DELETE}}(\rho : \text{id}, \ell) \equiv \underline{\text{DELETE}}(\rho' : \ell)$

$\leftarrow \rho' = \underline{\text{DELETE}}\langle \rho : \text{id} \rangle.$

[118] $\underline{\text{OVERRIDE}}(\rho : \rho') \equiv \rho + \rho''$

$\leftarrow \rho'' \equiv \underline{\text{DELETE}}(\rho' : \underline{\text{DOMAIN}}(\rho))$

[119] $\underline{\text{LEVEL NUM}}(n_1\ \text{id}_1\ \text{succ}_1, \ldots, n_k\ \text{id}_k\ \text{succ}_k) \equiv n_k.$

[120] $\text{CONVERTIBLE}\ \langle \text{type}_1 : \text{type}_2 \rangle$

$\leftarrow \underline{\text{RESULT TYPE}}(\text{type}_1 : = : \text{type}_2) \equiv \text{STRING}.$

[121] $\text{DIFF IDLIST}\ \langle \Lambda \mid \text{id} \rangle.$

[122] $\text{DIFF IDLIST}\ \langle \ell, \text{id} \rangle$

$\leftarrow \text{id} \notin \ell.$

Appendix 2: TRANSLATION OF λ-CALCULUS EXPRESSIONS INTO NORMAL FORM

[a. *Primitive Predicates*]

id IDENTIFIER [*An ordered set of distinct atoms*]

DIFF ID [*The set of all pairs of different identifiers*]

exp EXPRESSION <exp> ← (IDENTIFIER<exp> | COMBINATION<exp> | LAMBDA EXP<exp>).

COMBINATION <exp$_1$(exp$_2$)>.

LAMBDA EXP <λid.exp>.

[b. *Free Identifiers*]

FREE IDS(id) ≡ id.
FREE IDS(exp$_1$ (exp$_2$)) ≡ UNION(FREE IDS(exp$_1$) : FREE IDS(exp$_2$)).
FREE IDS(λid.exp) ≡ REF COMP(FREE IDS(exp) : id).

[*The functions UNION and REL COMP are straightforward to define and are left as an exercise for the reader*]

[c. *λ-Calculus Substitution Rule*]

SUBST(id : exp : id) ≡ exp.
SUBST(id : exp : id$_1$) ≡ id$_1$ ← DIFF ID<id : id$_1$>.
SUBST(id : exp : exp$_1$(exp$_2$)) ≡ exp$_1$'(exp$_2$') ← exp$_1$' ≡ SUBST(id : exp : exp$_1$) &
exp$_2$' ≡ SUBST(id : exp : exp$_2$).

SUBST(id : exp : λid.exp$_1$) ≡ λid.exp$_1$.
SUBST(id : exp : λid$_1$.exp$_1$) ≡ λid$_1$.exp$_1$' ← DIFF ID<id : id$_1$> &
id$_1$ ∉ FREE IDS(exp) &
exp$_1$' ≡ SUBST(id : exp : exp$_1$).

SUBST(id : exp : λid$_1$.exp$_1$) ≡ λid$_2$.exp$_1$" ← DIFF ID<id : id$_1$> &
id$_1$ ∈ FREE IDS(exp) &
DIFF ID<id$_1$: id$_2$> &
id$_2$ ∉ FREE IDS(exp) &
id$_2$ ∉ FREE IDS(exp$_1$) &
exp$_1$' ≡ SUBST(id$_1$: id$_2$: exp$_1$) &
exp$_1$" ≡ SUBST(id : exp : exp$_1$').

[d. *Conversion to Normal Form*]

NORMAL FORM <id>.
NORMAL FORM <exp$_1$(exp$_2$)> ← NORMAL FORM<exp$_1$> & NORMAL FORM<exp$_2$> &
(IDENTIFIER<exp$_1$> | COMBINATION<exp$_1$>).

NORMAL FORM <λid.exp ← NORMAL FORM<exp>.

CONV(exp) ≡ exp ← NORMAL FORM<exp>.
CONV(λid.exp) ≡ λid'.exp' ← exp' ≡ SUBST(id : id' : exp').
CONV(λid.exp (exp$_1$)) ≡ exp$_2$ ← exp$_2$ ≡ SUBST(id : exp$_1$: exp).
CONV(exp$_1$(exp$_2$)) ≡ CONV(exp$_1$'(exp$_2$')) ← exp$_1$' ≡ CONV(exp$_1$) & exp$_2$' ≡ CONV(exp$_2$)

TRANS TO NORMAL FORM(exp) ≡ exp' ← CONV(exp) ≡ exp' & NORMAL FORM<exp'>.