PRODUCTION SYSTEMS: A FORMALISM FOR SPECIFYING THE SYNTAX AND TRANSLATION OF COMPUTER LANGUAGES

by

Henry F. Ledgard

Oxford University Computing Laboratory

Programming Research Group

8169 COPY 2

PRODUCTION SYSTEMS: A FORMALISM FOR SPECIFYING

THE SYNTAX AND TRANSLATION OF COMPUTER LANGUAGES

BY

Henry F. Ledgard

Technical Monograph PRG-1

March 1970.

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford.

ABSTRACT

This paper investigates the application of a formalism called production systems to specify 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 same formalism can be used to specify the translation of a language into another.

(c) The specification of the context-free portions of syntax, the context-sensitive portions of syntax, and the translation can to a large extent be isolated.

(d) The formalism can be used to specify the "abstract" syntax of a language and its translation into "abstract" entities of a target language.

The following example applications of production systems are given:

(a) A specification of the syntax of a limited subset of ALGOL 60 and its translation into IBM System 360 assembler language.

(b) A specification of the abstract syntax of a small functional language and its translation into expressions in Church's λ-calculus.

<u>CONTENTS</u>

1. INTRODUCTION

1.1 Motivation

This paper presents the formalism of production systems and investigates its application to define the 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 author's interest in production systems stems partly from happenstance, and partly from a conviction that there are certain properties of the formalism that appear valuable in defining syntax and translation. First, production systems are based on solid mathematical foundations [1,2] and have, theoretically at least, the power to define the class of computable functions. This theoretical power, while desirable, can be misleading. There exist other formal notions like Turing machines and Markov algorithms with equivalent theoretical power, but it certainly appears hopeless to define the syntax and translation of computer languages with a Turing machine or a Markov algorithm.

The elusive but essential notion we must face in choosing any formal system is its 'acceptability' [7] 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. While production systems have the added power to define context-sensitive requirements on syntax and to define translation, production systems in their strict form do not possess the conciseness of Backus-Naur form. Owing to the more complex nature of context-sensitive requirements on languages and the specification of translation, some additional complexity must be expected. On the other hand, when viewed as a generative grammar, production systems provide some 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.

Conventionally, when a language is specified, the context-free portions of syntax are specified by productions in a context-free grammar, the context-sensitive requirements are separately specified using English text, and the semantics are usually specified by relating constructs in the language to concepts assumed understood in English or existing mathematics. A formalization of this intuitive approach to language definition is taken here, using only the definitional apparatus of production systems. Most productions in a production system specification of syntax define context-free requirements on strings. Context-sensitive requirements are specified by inserting certain restrictive premises, whose definitions are given separately. The semantics are specified by a separate production system defining the translation of a syntactically legal program into a target language*, whose meaning is presumably understood. The

*No target language for defining semantics is presented here.

resulting specifications are moderately concise, although admittedly not optimal.

Perspicuity of definition appears more important than conciseness of definition. Three factors seem paramount in determining perspicuity: segmentation of the parts of a definition, notation and the conceptual framework within which the definition is given. The segmentation of a production system specification discussed above certainly adds to the perspicuity of a production system's definition. Furthermore, the basic notation for production systems appears satisfactory. 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.

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. While the conceptual framework of sets appears unnatural for certain definitions (e.g. the definition of arithmetic functions), it appears convenient to view a language as a set of strings and the translation of one language into another as a set of ordered pairs of strings.

Superimposed on the basic notation for production systems is a notation for defining functions. Via the function-like notation portions of a production 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 (for example, the list of statement labels occurring in a block) needed to complete the specification are defined via the function-like notation, i.e. as functions that given a basic constituent (for example, a block) as an argument yields the auxiliary constituent (for example, its list of statement labels) as a result.

One deficiency as regards perspicuity of definition still remains in the application of production systems presented here. As mentioned above, in the specification of context-sensitive requirements, several functions are defined. For example, to define the requirement that all statement labels in a block are different, a function mapping a block into a list of its statement labels is defined. Functions like this, while intuitively simple, become somewhat complicated when defined in production systems. Whether functions like this ought to be defined by other methods is a subject I have not investigated.

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 complete language, ALGOL 60, has been defined with a production system, and a separate paper discussing this production system is being prepared. Syntactically, few computer languages are more complex than ALGOL 60, and it seems fair to say that a judgement (good or bad) on the merits of the production system of ALGOL 60 is a good test of the acceptability of production systems to define the syntax of most computing languages. No production systems specifying the translation of complete languages have been attempted. Hence the acceptability of production systems to adapt to various types of translation is largely untested.

Much other research on formal definition of computer languages has been pursued. A comprehensive review of existing methods has been written by de Bakker [14]. Several devices employed by others are used here, notably the work of McCarthy [12] and the IBM Vienna laboratory [13] on the definition of the abstract syntax and the use of Church's λ-calculus to define semantics by Landin [11]. With the thought that production systems may find a useful place in meeting the need for formal methods of language definition, the research presented here is offered.

1.2 Background of the Formalism

The mathematical underpinnings of production systems are due to Emil Post [1] and Raymond Smullyan [2]. A discussion of the theoretical background for production systems has been given [4] by this author. With suitable syntactic changes, production systems are equivalent to Smullyan's 'elementary formal systems'[2]. Production systems can be used to specify any 'recursively enumerable' set [2]. 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. Presumably, production systems can specify any translation or algorithm that a machine can perform. Heuristic evidence that this statement is true is due to the works of Turing [16,17] and Kleene [18]. In these works the notion of functions computable by a Turing machine were asserted [16] to comprise every function or algorithm that is intuitively computable by machine, and the functions computable by a Turing machine were shown equivalent [17,18] to the set of all 'general recursive' sets, which are encompassed by production systems.

The application of a logically modified variant of the formal systems of Post [1] Smullyan[2] and Trenchard More [19] to specify completely the syntax of a computer language was first made by John Donovan [3]. Donovan applied his formal system to specify the set of legal programs in a computer language, including the specification of allowable character spacing, and more importantly, the specification of context-sensitive requirements on the set of legal programs, like the requirement that all statement labels in a program be different. Donovan introduced the term 'canonic systems' to describe his formal system. 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 terminology for production systems presented here is due to both Post and Smullyan. The notation for production systems presented here is due in part to Post, Smullyan and Donovan, but for the most part is new.

1.3 An Informal Example

Before discussing the formalism of production systems in Section 2.1, this section informally presents an example production system, which hopefully will motivate the discussion of Section 2.1. A small and rather useless subset of ALGOL 60 will be taken as an example source language. The Backus-Naur form specification of the ALGOL 60 subset is given in Table 1.*

```
1.    <NUMBER>      ::=   1|2|3
2.    <ID>          ::=   A|B

3.1   <PRIMARY>     ::=   <NUMBER> | <ID>
3.2   <ARITH EXP>   ::=   <PRIMARY> | <ARITH EXP>+<PRIMARY>
3.3   <STM>         ::=   <ID>:=<ARITH EXP>

4.1   <TYPE LIST>   ::=   A|B|A.B
4.2   <DEC>         ::=   integer<TYPE LIST>

5.    <PROGRAM>     ::=   begin<DEC>;<STM>  end
```

Table 1. Backus-Naur form specification of ALGOL 60 subset.

This subset allows programs containing only one declaration and one limited type of arithmetic assignment statement. The syntax of ALGOL 60 has the requirement that the type of each identifier used in a program must be declared. This requirement is not handled by

*Underlined lower case letters are used here to represent reserved words in a computer language.

the Backus-Naur specification above. For example, the syntactically illegal program

begin integer B; A:=1 end

can be derived using this specification.

The production system specification of the ALGOL 60 subset is given in Table 2.

```
        begin NUMBER<n>, ID<i>, PRIMARY<p>, ARITH EXP<a>,
              STM<s>, TYPE LIST<ℓ>, DEC<d>;

1.      NUMBER<1>,<2>,<3>.
2.      ID<A>,<B>.

3.1     PRIMARY<n>,<i>.
3.2     ARITH EXP<p>,<a+p>.
3.3     STM<i:=a>.

4.1     TYPE LIST<A>,<B>,<A,B>.
4.2     DEC<integer ℓ>.

5.      PROGRAM<begin d;s end>  ←  IN<IDS<s>:IDS<d>>.


6.1     IN<A:A>,<B:B>,<A:A,B>,<B:A,B>.
6.2     IN<xy:ℓ>                ←  IN<x:ℓ>,<y:ℓ>.

7.      NON ID<+>,<:=>,<,>,<integer>,<n>.

        begin φ=IDS, NON ID<r>;
8.1     φ<i> = <i>.
8.2     φ<xiy> = <i,φ<xy>>.
8.3     φ<xry> = <φ<xy>>.
        end

        end
```

Table 2. Production system specification of ALGOL 60 subset.

Productions 1 through 5 of this production system may be informally read.

Let n be a number, i be an identifier, p be a primary, a be an arithmetic expression, s be a statement, ℓ be a type list, and d be a declaration (all of which are to be defined below):

1. The symbols '1', '2' and '3' are numbers.

2. The symbols 'A' and 'B' are identifiers.

3. If n is a number, then n is a primary.
   If i is an identifier, then i is a primary.

3.2 If p is a primary, then p is an arithmetic expression.
If p is a primary, and a is an arithmetic expression, then a'+'p is an arithmetic expression.

3.3 If i is an identifier and a is an arithmetic expression, then i':='a is a statement.

4.1 The strings 'A', 'B' and 'A,B' are type lists.

4.2 If ℓ is a type list, then 'integer'ℓ is a declaration.

5. If d is a declaration, s is a statement, and each member of the list of identifiers for s is contained in the list of identifiers for d, then 'begin' d ';' s 'end' is a program.

The restrictive premise

IN<IDS<s>:IDS<d>>

is the essential one needed to insure that all identifiers must be declared. The function named 'IDS' maps a string in the ALGOL 60 subset into a list of identifiers occurring in the string. This function is defined in productions 8, where 'ϕ' is used in place of the name 'IDS' and r denotes a member of the class of non-identifier symbols, defined in productions 7. For example,

IDS<integer B>	=	<B>	IDS<A:=A+B>	=	<A,A,B>
IDS<A:=1>	=	<A>	IDS<A+B:=A+1+B>	=	<A,B,A,B>

Productions 6 define a set of ordered pairs named 'IN', where the first element is a list of identifiers and the second element is a list of identifiers containing each identifier given in the first list. For example the following ordered pairs are members of the set named 'IN'

<A:A> <B:A,B> <A,B:A,B> <A,B,A,B:A,B>

Jointly, the restrictive premise in production 5 and the definitions of productions 6 through 8 specify that the list of identifiers for a statement s be contained in the list of identifiers for a declaration d. Thus the string

begin integer A; A:=1 end

is specified by this production system, whereas the illegal string

begin integer B; A:=1 end

is not specified by this production system because the pair <A:B> where 'A' is the list of identifiers for the statement 'A:=1' and 'B' is the list of identifiers for the declaration 'integer B', is not a member of the set named 'IN'.

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 predicate in the predicate alphabet is assigned a unique positive integer called its degree.
3. An alphabet called the *variable alphabet.*
4. Another alphabet called the *punctuation alphabet*, which consists of eight symbols: the implication sign, conjunction sign, tuple sign, delimiter sign, left quote sign, right quote sign, left bracket sign, and right bracket sign.
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. Accordingly I will use (a) strings of capital letters, possibly interlated with digits, spaces and tuple signs, for predicate alphabet symbols (b) lower case letters (possibly subscripted or superscripted) for variable alphabet symbols (c) the symbols

←	implication sign
,	conjunction sign
:	tuple sign
.	delimiter sign
\ /	left and right quote signs
< >	left and right bracket signs

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

A *well-formed term* consists of a concatenated sequence of variable and object alphabet symbols (e.g. 'i', 'a', 'a+p' and 'i:=a').* 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. '<i:=a>' and '<x:ℓ>'). 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 term tuple of degree n (e.g. 'STM<v:=a>' and 'IN<x:ℓ>', where 'STM' and 'IN' are predicates of degrees 1 and 2 respectively). A *well-formed production* consists of

(a) an atomic formula followed by the delimiter sign (e.g., 'NUMBER<1>.') or

(b) an atomic formula followed by the implication sign, a sequence of atomic formulas each separated by the conjunction sign, and the delimiter sign (e.g. 'STM<i:=a> ← 10<i>, ARITH EXP<a>.').

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 letters, digits, spaces, and punctuation symbols as members of the object alphabet. Since capital letters, digits, spaces, the implication sign, conjunction sign, and delimiter sign cannot occur within the brackets of a term tuple as predicate, variable, or punctuation alphabet symbols, I adopt the convention that 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

---

* ':=' is considered a *single object alphabet* symbol, not the concatenation of the symbols ':' and '='.

** The use of the quote and bracket signs are not necessary to a strict definition of a production system. In essence, quote signs enable the free use of symbols in the object alphabet, and the bracket signs enable the omission of quote signs around symbols that occur frequently. Both these syntactic devices are reminiscent of Quine's notion of quotations and quasi-quotations.[15 productions:

LETTER<`a´>

NUMBER<1>

NUMBER<2>

NUMBER<3>

IN<A:A,B>

IN<B:A,B>

IN<xy:ℓ> ← IN<x:ℓ>, IN<y:ℓ>

Here, the symbols {a 1 2 3 A B} enclosed in angle brackets are object alphabet symbols. The symbols {x y ℓ} are variable alphabet symbols.

<u>Deductive Rules</u>:

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

NUMBER<1>

IN<A:A,B>

IN<B:A,B>

IN<A,B:A,B>

IN<A,B,A:A,B>

<u>Interpretation</u>:

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 'NUMBER' contains three members,

{1 2 3}

and the set 'IN' contains an infinite number of ordered pairs, some of which are denoted by

{<A:A,B> <B:A,B> <A,B:A,B> ... }.

2.2 Abbreviations and Modifications 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. In this section four notational conventions are given, the second of which is due to Donovan [3].

Abbreviations:

The two abbreviations are motivated by conciseness of definition. The first or 'block structure' abbreviation allows one to 'factor out' premises that are common to one or more productions. The second allows one to eliminate repeated occurrences of the same predicate name.

1. If $P_1, P_2, \ldots, P_n$ are predicates, $v_1, v_2, \ldots, v_n$ are variables, and C is a collection of productions such that any production containing v_i, $1 \le i \le n$, in the conclusion also contains the premise $P_i<v_i>$, then

   C

   can be abbreviated

   begin $P_1<v_1>$, $P_2<v_2>$, ... $P_n<v_n>$;

   C'

   end

   where C' is obtained from C by deleting any or all occurrences of the premises $P_1<v_1>$, $P_2<v_2>$, ... , and $P_n<v_n>$ and their associated punctuation signs.*

---

* If a premise is deleted from a production containing other premises, the conjunction sign preceeding or following the premise is deleted. If a premise is deleted from a production containing no other premises, the implication sign is deleted.

Thus, for example

```
ARITH EXP<p>        ←    PRIMARY<p>.
ARITH EXP<a+p>      ←    PRIMARY<p>, ARITH EXP<a>.
STM<i:=a>           ←    ID<i>, ARITH EXP<a>.
```

may be abbreviated

```
begin PRIMARY<p>, ARITH EXP<a>, ID<i>;
ARITH EXP<p>.
ARITH EXP<a+p>.
STM<i:=a>.
end
```

This abbreviation is extended to include nested _begin_ - _end_ bracketed productions with new 'declarations' of variables. For example

```
begin P<a>, Q<b>;
C<a+b>.
        begin R<b>;
        D<a+b>.
        end
end
```

is an abbreviation for

```
C<a+b>  ←  P<a>, Q<b>.
D<a+b>  ←  P<a>, R<b>.
```

2.a. If $<t_1>, <t_2>$, ... and $<t_n>$ are term tuples and P is a predicate, the atomic productions

$$P<t_1>.$$
$$P<t_2>.$$
$$\vdots$$
$$P<t_n>.$$

can be abbreviated

$$P<t_1>,<t_2>,\ldots<t_n>.$$

2.b. If $<t_1>$, $<t_2>$, ... and $<t_n>$ are term tuples and P is a predicate, the premises

$$P<t_1>,\ P<t_2>,\ \ldots\ P<t_n>$$

can be abbreviated

$$P<t_1>,<t_2>,\ldots<t_n>$$

For example, the productions

IN<A:A>.

IN<B:B>.

IN<A:A,B>.

IN<B:A,B>.

IN<xy:ℓ> ← IN<x:ℓ>, IN<y:ℓ>.

can be abbreviated

IN <A:A>,<B:B>,<A:A,B>,<B:A,B>.

IN<xy:ℓ> ← IN<x:ℓ>,<y:ℓ>.

<u>Notation for Functions:</u>

As mentioned in the introduction, 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 third and fourth notational conventions reflect this predisposition.

3. If v_1, v_2, ... and v_n, $n \geq 2$, are variables and $R<v_1:v_2: \ldots v_n>$ is a premise occurring in a production P containing exactly one other occurrence c of v_n, then the premise

$$R<v_1:v_2: \ldots v_n>$$

can be deleted from P if c is replaced by the string

$$\underline{R}<v_1:v_2: \ldots v_{n-1}>$$

4. If $t_1, t_2, \ldots$, and t_n, $n \geq 2$, are terms and $R<t_1:t_2:\ldots t_n>$ is an atomic formula occurring in a production P, then

$$R<t_1:t_2: \ldots t_n>$$

may be alternately written

$$\underline{R}<t_1:t_2: \ldots t_{n-1}> = <t_n>$$

Thus the productions

```
PROGRAM<begin d;s end> ← DEC<d>, STM<s>, IDS<s:i_s>,
                                    IDS<d:i_d>, IN<i_s:i_d>.
IDS<i:i>  ←  ID<i>.
IDS<xiy:i,z>  ←  ID<i>, IDS<xy:z>
IDS<xry:z>    ←  NON ID<r>, IDS<xy:z>.
```

can be written

```
PROGRAM<begin d;s end>   ←  DEC<d>, STM<s>, IN<IDS<s>:IDS<d>>.
IDS<i>   =  <i>           ←  ID<i>
IDS<xiy> =  <i, IDS<xy>> ←  ID<i>.
IDS<xry> =  <IDS<xy>>    ←  NON ID<r>.
```

Writing

```
IN<IDS<s>:IDS<d>>
```

instead of

```
IDS<s:i_s>, IDS<d:i_d>, IN<i_s:i_d>
```

not only reduces the length of the production, but suggests a conceptual view of 'IDS' as a function *mapping* an object (here a well-formed ALGOL 60 statement or declataion) into another object (here a list of identifiers). The use of this function-like notation *strongly* governed the manner in which the production system specifications presented here were written.*

Finally, since the predicate name of a function often occurs repeatedly in the productions defining the function, I extend abbreviation 1 in that an underlined predicate name P may be replaced by a Greek letter ℓ provided the 'declaration'

ℓ=P

is given for the productions. For example the above productions defining the function 'IDS' may be written

```
begin φ = IDS,ID<i>, NON ID<r>;
φ<i>    =  <i>.
φ<xiy>  =  <i,φ<xy>>.
φ<xry>  =  <φ<xy>>.
end
```

* The notation for functions allows one to define functions over object strings and variables. An extension to allow definition of functions over *predicates* was attempted, but owing to a lack of suitable generalization, will not be discussed further.

3. APPLICATION TO SPECIFY SYNTAX AND TRANSLATION

3.1 Application to Specify (concrete) Syntax

The syntax of a language may be defined as the set of well-formed strings in a language. In this section I will be concerned with the specification of 'concrete' syntax, i.e. a specification of strings that are given a concrete or definite representation. Later in Section 4, I shall turn to the specification of 'abstract' syntax, i.e. a specification of syntax for which no particular string representation is given.

A production system specifying the syntax of the ALGOL 60 subset is given in Appendices 1a and 1b, where Appendix 1a uses only the basic notation and Appendix 1b employs the modifications and abbreviations to the notation. There the predicate 'PROGRAM' names a set of 1-tuples where each member is a syntactically legal program. An intuitive presentation of the abbreviated production system has been given in the introduction and will not be discussed further.

3.2 Application to Specify Translation

The translation of a language may be defined as the function (or relation) between the well-formed strings in the language and well-formed strings in another language. This function or relation can be specified by a production system specifying a set of ordered pairs of strings, where the first element in each pair is a legal string in the source language, and the second element is a corresponding string in the target language.

As in the previous section, I will illustrate this use of production systems by example. The specification of the syntax of the ALGOL 60 subset in Appendix 1b has been augmented to specify not only the legal strings in the subset but also their translation into IBM System 360 assembler language [21]. The additional productions are given in Appendix 1c. There a function 'TRANSLATE' mapping strings in the ALGOL 60 subset into strings in assembler language is defined. A pair <x:y> is defined as a member of the set 'PROGRAM:TRANSLATION' if x is a legal program as specified in the definition of syntax and y is the mapping of x as specified by the function 'TRANSLATE'. For example, the following pair of strings is a member of the set named 'PROGRAM:TRANSLATION'

<u>begin</u> <u>integer</u> A; A:=1 <u>end</u> :

```
*ASSEMBLER LANGUAGE PROGRAM
     BALR    15,0      *SET BASE REGISTER
     USING   *,15      *INFORM ASSEMBLER
     L       1,=F'1'   *LOAD 1
     ST      1,A       *STORE RESULT IN A
     SVC     0         *RETURN TO SUPERVISOR
*STORAGE FOR VARIABLES
A    DS      F
     END
```

Note that this production system includes the specification of the comment entries in the assembler statements to that (hopefully) the reader will not have to be familiar with the assembler language to understand the translation.

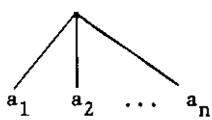
4. APPLICATION TO SPECIFY ABSTRACT SYNTAX

A definition of a class of 'abstract' objects is a definition for which no representation of objects is specified. Following the lines of McCarthy [12] and the IBM Vienna Laboratory [13], a definition of a class of abstract objects 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. For definitions of abstract objects, one needs some symbols to denote primitive objects, and some symbols to denote how composite objects are built up from primitive objects. Accordingly, arbitrary primitive symbols will be used to denote primitive objects, and composite objects containing n components will be denoted by the conventional notation for n-tuples, i.e.

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

or trees, i.e.



where a_1, a_2, ... and a_n denote objects. In general, the tree representation of an n-tuple may be designated by a node with n branches pointing to the n components of the object, where the leaves of the tree denote primitive objects. A defnintion will be considered 'abstract' in that all objects will be presented using only the primitive symbols and the notation for n-tuples (or equivalently, trees).

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

(a) Productions specifying the representation of primitive objects will be omitted in a production system specification. We shall say only what properties the primitive objects must possess and that any productions defining their representation must reflect these properties.

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

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

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

(c) Predicate names of a production system will be interpreted as predicates over the class of abstract objects in that $P<b>$, where P is a production system predicate name and b denotes an abstract object, 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 by production system predicates of degree 2 as follows. Let $(a_1,a_2,\ldots,a_n)$ denote an object in a class C and S_1, S_2, ... and S_n, $S_i \neq S_j$ for $i \neq j$, be the names of the selector functions over objects in C. Then

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

if and only if the conclusion

$$S_i<(a_1,a_2,\ldots,a_n):a_i>$$

or equivalently (using the function-like notation)

$$S_i<(a_1,a_2,\ldots,a_n)> = <a_i>$$

is derivable from the production system.

The notion of the definition of the translation of one class of abstract objects into another class of abstract objects may be couched in production systems by specifying a set of ordered pairs of abstract objects. The constructors, predicates, and selectors for objects in the target language can be defined analagously to the constructors, predicates and selectors of the source language. To illustrate the techniques for defining abstract syntax and translation, this section presents a small source language for defining functions and its translation into Church's λ-calculus. Owing to the more transparent notation for concrete representation of expressions in the λ-calculus, abstract objects in the source language will be translated into *concrete* representations of expressions in the λ-calculus. With relatively straight-forward extensions of the techniques presented in this section, the production systems may be made completely abstract in that both source and target language programs may be specified as abstract objects.

4.1 Mini-Language F

As an example source language for illustrating abstract definitions of computer languages, a small language called Mini-language F has been devised. Mini-language F is based on the ISWIM language of Peter Landin [10]. We first give an informal description of Mini-language F, using concrete representations of objects to indicate its syntax and an appeal to intuitive concepts expressed in English to indicate its semantics.

Primitive Objects: The primitive objects in Mini-language F include (a) the natural numbers, (b) a binary function that when applied to two natural numbers produces the natural number that is their numerical sum, and (c) a quarternary function that when applied to four objects, of which the first two are natural numbers, produces the third object if the first natural number is greater than or equal to the second and other wise produces the fourth object. The natural numbers will be represented by the symbols {0 1 2 ...}. The functions described by (b) and (c) above will be represented respectively by the symbols '+' and 'IF'.

Identifiers: The identifiers comprise the symbols {A B ... Z}.

Expression Lists: An expression list is a string of the form $e_1,e_2,\ldots,e_n$ where the e_i, $1 \leq i \leq n$, are expressions (defined below). The value of a list expression is the list of objects $a_1,a_2,\ldots,a_n$ obtained by successively evaluating each of the component expressions e_1,e_2, ..., and e_n.

Unit Expressions: A unit expression is either one of the primitive symbols {+ IF 0 1 2 ...} or an identifier. The value of a primitive symbol is the primitive object represented by the symbol. The value of an identifier is the object currently linked with the identifier (for linking of identifiers to objects, see definition and evaluation of let expressions).

Let Expressions: A let expression is a string of the form

(1) let $i = e_1$ in e_2

or (2) let $i(x_1,\ldots,x_n) = e_1$ in e_2

where i is an identifier, $x_1,\ldots,$ and x_n are identifiers each of which must be different, and e_1 and e_2 are expressions. In a let expression t of the above form, all occurrences of the identifier i except in e_1 are said to be 'bound in t', and all occurrences of $x_1,\ldots,x_n$ except in e_2 are said to be 'bound in t'. An occurrence of an identifier in an expression e is 'free in e' if it is not bound in e. The value of a let expression of the form (1) is computed by evaluating e_1, linking the free occurrences of i in e_2 with the value found, and then evaluating e_2. The value of a let expression of the form (2) is computed by forming the function mapping $x_1,\ldots,x_n$ into e_1 (where the free identifiers in e other than $x_1,\ldots,x_n$ in e_1 are linked with their current values), linking the free occurrences of i in e_2 with the function formed, and then evaluating e_2.

Combinations: A combination consists of a string of the form $e(\ell_e)$ where e is an expression and ℓ_e is an expression list. The value of a combination is obtained by evaluating e and ℓ_e and then applying the value of e to the value of ℓ_e. This evaluating is well-defined (i.e. not in violation) only if the value of e is a function and the value of ℓ_e is a list of objects such that the number of the components of the list is identical to the number of arguments of the function. Furthermore, in the case where the value of e is one of the primitive functions denoted by '+' or 'IF', the values of the first two components in the list ℓ_e must be natural numbers. The following alternate notations may be used for combinations

$[e_1+e_2]$	in place of	$+(e_1,e_2)$
$[e_1 \geq e_2 \Rightarrow e_3$ else $\Rightarrow e_4]$	in place of	$IF(e_1,e_2,e_3,e_4)$

Expressions: An expression is either a unit expression, a let expression, or a combination.

Programs: A program is an expression such that no identifiers occur free in the expression.

Example 1

```
let F(Y) = [Y+3]
in  [F(1)+F(2)]
```

Example 2

```
let F(X) = [X+X]
in  let G(P,X) = [P(X)+P(1)]
    in  G(F,2)
```

Example 3

```
let Y = 2
in  let F(X)=[X+Y]
    in  F
```

Example 4 (illegal)

```
let F(X) = [X>3 => X
            else => [X+F(X+1)]
in  F(2)
```

Example 5

```
let F(X) = 2
in  let F(X) = [X>3 => X
                else => [X+F(X+1)]]
    in  F(2)
```

The values of the example programs 1, 2 and 3 above are respectively the natural number nine, the natural number six, and the function mapping x into summation of x and the natural number two. The program of example 4 is syntactically illegal since the occurrence of 'F' in the conditional expression is free. The value of example program 5 is the natural number four.

A production system specifying the concrete syntax of mini-language F is given in Appendix 2a. Productions 1 through 4, aside from two premises, specify the class of programs ignoring context-sensitive requirements. The context-sensitive requirement that the parameters $x_1, \ldots,$ and x_n of a function definition each be different is specified in production 3.2 by a premise requiring that the list $\ell = x_1, \ldots, x_n$ be a member of the set named 'DIFF IDLIST'. The requirement that no identifiers in a program occur free is specified in production 4 by a premise requiring that the list of free identifiers of the program be null.

The auxiliary predicates needed to specify the two context-sensitive requirements are defined in productions 5 through 9. Some example strings defined by these productions are

```
FREE IDS<[A+B]>                    =      <A,B>
FREE IDS<let A = 1 in [A+B]>       =      <B>

LIST<∧:A,B>                        =      <A,B>
LIST<A,B:C,D,A>                    =      <A,B,C,D,A>

REL COMP<A,B:A>                    =      <B>
REL COMP<A,B,C,D:A,B,X,Y>          =      <C,D>

IN<A:A>,<A:A,B,C>
NOT IN<A:B>,<A:B,C,D>
DIFF IDLIST<A,B,C,D>
```

Here the function '<u>FREE IDS</u>' maps an expression into its list of free identifiers. The function '<u>LIST</u>' maps two identifier lists into a single list containing all occurrences of identifiers in the first two lists. The function '<u>REL COMP</u>' maps two identifier lists into a single list containing only the identifiers occurring in the first list but not in the second (similar to the relative complement of two sets). The predicate 'IN' defines a set of ordered pairs where the first element is an identifier and the second element is a list of identifiers containing an occurrence of the first identifier. The predicate 'NOT IN' defines a set of pairs where the first element is an identifier and the second element is a list of identifiers *not containing* an occurrence of the first identifier. The predicate 'DIFF IDLIST' define a set where each element is a list of different identifiers.

4.2 Definition of Abstract Syntax

In a definition of abstract syntax, we first assume that there are certain classes of primitive objects with certain primitive properties. For mini-language F these comprise the class of identifiers, the class of natural numbers, and two classes containing one member each, the addition function and the if function. These four classes have their conventional properties and will be denoted by the predicate names

ID   NAT NUM   ADD FCN   IF FCN

A list of identifiers in mini-language F may now be defined as a pair

$$(i,\wedge)$$

where i is an identifier and '$\wedge$' is a symbol denoting the null object, or as a pair

$$(i,\ell)$$

where i is an identifier and ℓ is itself a list of identifiers. Similarly, a let expression may be defined as a triple of the form

$$(i,e_1,e_2)$$

or

$$(i,f,e_2)$$

where i is an identifier, e_1 and e_2 are expressions, and f is a function. In either case, the first element of a triple denotes the bound identifier of the let expression, the second element denotes the definiens (i.e. the object to which the identifier is bound), and the third element denotes the expression within which the identifier is bound to the definiens.

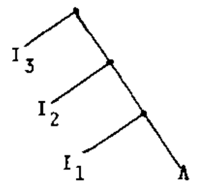
The class of identifier lists and class of let expressions may be given the predicate named 'IDLIST' and 'LET EXP' and may be defined by the productions

(1) $\text{IDLIST}\langle(i,\wedge)\rangle \leftarrow \text{ID}\langle i\rangle.$
(2) $\text{IDLIST}\langle(i,\ell)\rangle \leftarrow \text{ID}\langle i\rangle,\ \text{IDLIST}\langle\ell\rangle.$
(3) $\text{LET EXP}\langle(i,e_1,e_2)\rangle \leftarrow \text{ID}\langle i\rangle,\ \text{EXP}\langle e_1\rangle,\ \text{EXP}\langle e_2\rangle.$
(4) $\text{LET EXP}\langle(i,f,e_2)\rangle \leftarrow \text{ID}\langle i\rangle,\ \text{FCN}\langle f\rangle,\ \text{EXP}\langle e_2\rangle.$

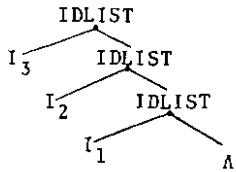
The n-tuples defined by these productions may be represented via the notation of trees. For example, the identifier list

$$(I_3,(I_2,(I_1,\wedge)))$$

where I_1, I_2 and I_3 are unspecified identifiers, may also be represented



Furthermore, each non-terminal node may be labelled with the predicate name of the class within which the n-tuple is a member. For the identifier list above, we have the tree with labelled nodes.



For each composite object we must define the selector functions for extracting the components of the object. In particular, for an identifier list we wish to select its head and tail, and for a let expression we wish to select its bound identifier, its definiens, and the expression within which the identifier is bound to the definiens. These five functions may be given the named '$\underline{\mathrm{HD}}$', '$\underline{\mathrm{TL}}$', '$\underline{\mathrm{BID}}$', '$\underline{\mathrm{DEF}}$', and '$\underline{\mathrm{BEXP}}$', defined as follows:

(1a)	$\underline{\mathrm{HD}}\langle(i,\Lambda)\rangle = \langle i\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle.$
(1b)	$\underline{\mathrm{TL}}\langle(i,\Lambda)\rangle = \langle\Lambda\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle.$
(2a)	$\underline{\mathrm{HD}}\langle(i,\ell)\rangle = \langle i\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle,\ \mathrm{IDLIST}\langle\ell\rangle.$
(2b)	$\underline{\mathrm{TL}}\langle(i,\ell)\rangle = \langle\ell\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle,\ \mathrm{IDLIST}\langle\ell\rangle.$
(3a)	$\underline{\mathrm{BID}}\langle(i,e_1,e_2)\rangle = \langle i\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle,\ \mathrm{EXP}\langle e_1\rangle,\ \mathrm{EXP}\langle e_2\rangle.$
(3b)	$\underline{\mathrm{DEF}}\langle(i,e_1,e_2)\rangle = \langle e_1\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle,\ \mathrm{EXP}\langle e_1\rangle,\ \mathrm{EXP}\langle e_2\rangle.$
(3c)	$\underline{\mathrm{BEXP}}\langle(i,e_1,e_2)\rangle = \langle e_2\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle,\ \mathrm{EXP}\langle e_1\rangle,\ \mathrm{EXP}\langle e_2\rangle.$
(4a)	$\underline{\mathrm{BID}}\langle(i,f,e_2)\rangle = \langle i\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle,\ \mathrm{FCN}\langle f\rangle,\ \mathrm{EXP}\langle e_2\rangle.$
(4b)	$\underline{\mathrm{DEF}}\langle(i,f,e_2)\rangle = \langle f\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle,\ \mathrm{FCN}\langle f\rangle,\ \mathrm{EXP}\langle e_2\rangle.$
(4c)	$\underline{\mathrm{BEXP}}\langle(i,f,e_2)\rangle = \langle e_2\rangle$	$\leftarrow$	$\mathrm{ID}\langle i\rangle,\ \mathrm{FCN}\langle f\rangle,\ \mathrm{EXP}\langle e_2\rangle.$

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 objects containing n components, $\underline{S_1},\underline{S_2},\ldots,$ and $\underline{S_n}$ be the function names of the n selector functions over the class of objects, and $t_1,t_2,\ldots,t_n$ be terms. Productions of the form

$$C\langle(t_1,t_2,\ldots,t_n)\rangle \rightarrow P.$$
$$\underline{S_1}\langle(t_1,t_2,\ldots,t_n)\rangle = \langle t_1\rangle \rightarrow P.$$
$$\underline{S_2}\langle(t_1,t_2,\ldots,t_n)\rangle = \langle t_2\rangle \rightarrow P.$$
$$\vdots$$
$$\underline{S_n}\langle(t_1,t_2,\ldots,t_n)\rangle = \langle t_n\rangle \rightarrow P.$$

may be combined into the single production

$$C\langle(\underline{S_1}\ t_1,\underline{S_2}\ t_2,\ \ldots,\ \underline{S_n}\ t_n)\rangle \rightarrow P.$$

Thus productions 1, 1a, 1b 2,2a,2b 3,3a,3b and 4,4a,4b can be combined

(1') $\mathrm{IDLIST}\langle(\underline{\mathrm{HD}}\ i,\ \underline{\mathrm{TL}}\ \Lambda)\rangle \rightarrow \mathrm{ID}\langle i\rangle.$

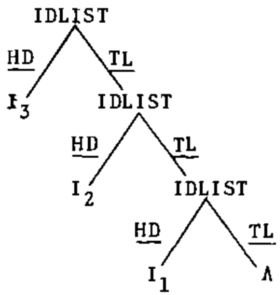
(2') $\mathrm{IDLIST}\langle(\underline{\mathrm{HD}}\ i,\ \underline{\mathrm{TL}}\ \ell)\rangle \rightarrow \mathrm{ID}\langle i\rangle,\ \mathrm{IDLIST}\langle\ell\rangle.$

(3') $\mathrm{LET\ EXP}\langle(\underline{\mathrm{BID}}\ i,\underline{\mathrm{DEF}}\ e_1,\underline{\mathrm{BEXP}}\ e_2)\rangle \rightarrow \mathrm{ID}\langle i\rangle,\ \mathrm{EXP}\langle e_1\rangle,\ \mathrm{EXP}\langle e_2\rangle.$

(4') $\mathrm{LET\ EXP}\langle(\underline{\mathrm{BID}}\ i,\underline{\mathrm{DEF}}\ f,\ \underline{\mathrm{BEXP}}\ e_2)\rangle \rightarrow \mathrm{ID}\langle i\rangle,\ \mathrm{FCN}\langle f\rangle,\ \mathrm{EXP}\langle e_2\rangle.$

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 abstract definition of the class of objects. This abbreviated notation will be used repeatedly in the sequel.

The selector functions defined over a class of objects may be added to the tree representation of an object by labelling the branches of a tree with the name of the selector function used to select the component of the object designated by the branch. For the identifier list above, we may construct the labelled tree



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 mini-language F that the identifiers given as parameters in a function definition must each be different. In terms of abstract syntax, the identifiers $I_1, I_2, \ldots,$ and I_n in the list

$$(I_n, \ldots (I_2, (I_1, \Lambda)) \ldots)$$

used in a function definition must each be different.

Next consider the productions

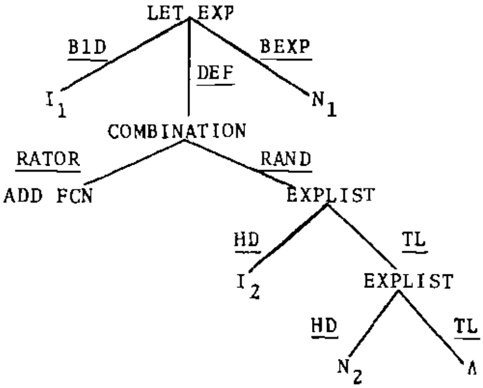
```
begin ID<i>,<j>, IDLIST<ℓ>;
NOT IN<i:(j,Λ)>      ←    DIFF ID<i:j>.
NOT IN<i:(j,ℓ)>      ←    DIFF ID<i:j>, NOT IN<i:ℓ>.
DIFF IDLIST<(i,Λ)>
DIFF IDLIST<(i,ℓ)>   ←    NOT IN<i:ℓ>, DIFF IDLIST<ℓ>.
end
```

As mentioned earlier, the predicate 'ID' specifying the class of identifiers is left unspecified in the definition of the abstract syntax of mini-language F. 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 mini-language F is given in Appendix 2b. There the intuitive role of the predicates parallel those given for the concrete syntax, except that no concrete representation of programs is specified and that the predicates

ID   DIFF ID   NAT NUM   ADD FCN   IF FCN

are left unspecified. For example, the following abstract program is defined by Appendix 2b.



where I_1 and I_2 are identifiers and N_1 and N_2 are natural numbers. This abstract program corresponds to any one of the concrete programs.

<u>let</u> A = 1 <u>in</u> +(A,2)
<u>let</u> X = 2 <u>in</u> [X+4]

and many others. Note that the program

<u>let</u> A = 1 <u>in</u> +(B,2)

is not derivable because the identifier B occurs free in the program. In terms of the abstract tree, the identifiers chosen for I_1 and I_2 must be identical in order for I_2 not to occur free.

4.3 Concrete Representations of Abstract Programs

To specify a concrete representation of a class of objects, given a definition of its abstract syntax, we may simply add to the definition of abstract syntax

(a) a definition of the predicates for the classes of primitive objects, and

(b) a definition of a function mapping abstract objects into concrete representations.

For mini-language F, we define

(a) the predicates 'ID', 'DIFF ID', 'NAT NUM', 'ADD FCN', and 'IF FCN', and

(b) a function named '<u>CONCRETIZE</u>' mapping an abstract program into its concrete representation as specified in the informal definition of mini-language F.

For example, the representation of let expressions whose definiens are expressions is defined by the production

$$\phi<t> \quad = \quad < \underline{\text{let}}\,\phi<\underline{BID}<t>> \;=\; \phi<\underline{DEF}<t>> \quad \underline{\text{in}}\ \phi<\underline{BEXP}<t>> > \qquad \leftarrow \quad EXP<\underline{DEF}<t>>.$$

where ϕ is used in place of the function name '<u>CONCRETIZE</u>' and t denotes a let expression.

Finally, a pair <p:q>is specified as a member of the set 'ABSTRACT PROGRAM:CONCRETIZATION' if p is a member of the set 'ABSTRACT PROGRAM' and q is the mapping of p into concrete form as specified by the function '<u>CONCRETIZE</u>'.

4.4 Translation of Mini-Language F into λ-Calculus

The semantics of mini-language F may be defined in terms of Church's λ-calculus [6,7], in particular the λ-Kδ-calculus. Albeit mini-language F can be viewed merely as a variant notation for a class of λ-calculus expressions. Nevertheless, to illustrate the specification of the translation of abstract programs with production systems, the translation of mini-language F into the λ-calculus is given. In particular, mini-language F is defined in terms of the λ-calculus where the only constants are

(a) The natural numbers, represented by {0 1 2 ... }

(b) A 'Curried' function '+' that when applied to two natural numbers N_1 and N_2 in an expression of the form

$$+\ N_1\ N_2$$

yields the natural number that is the sum of N_1 and N_2.

(c) A function '$\geq$' that when applied to two natural numbers N_1 and N_2 in an expression of the form

$$\geq\ N_1\ N_2$$

yields one of the expressions

$$\lambda\alpha.\lambda\beta.\alpha \qquad \text{or} \qquad \lambda\alpha.\lambda\beta.\beta$$

accordingly as the number N_1 <u>is</u> or <u>is not</u> greater than or equal to the number N_2.

For example, the abstract program for the concrete mini-language F program

```
let F(X,Y)  =  [X+Y]
in  F(1,2)
```

is translated into the λ-calculus expression

$$(\lambda F.F\ 1\ 2)\quad(\lambda X.\lambda Y.\ +\ X\ Y)$$

which successively reduces to

$$(\lambda X.\lambda Y.\ +\ X\ Y)\ 1\ 2$$
$$+\ 1\ 2$$
$$3$$

The abstract program for the mini-language F concrete program

```
let X = 3
in  [X ≥ 1 => 4
     else  => 5]
```

is translated into the λ-calculus expression

$$(\lambda X.(\lambda\alpha.\lambda\beta.\lambda\pi_1.\lambda\pi_2.\ \geq\ \alpha\ \beta\ \pi_1\ \pi_2)\ X\ 1\ 4\ 5)\ 3$$

which successively reduces to

$$(\lambda\alpha.\ \beta.\lambda\pi_1.\lambda\pi_2.\ \geq\ \alpha\ \beta\ \pi_1\ \pi_2)3\ 1\ 4\ 5$$
$$\geq\ 3\ 1\ 4\ 5$$
$$(\lambda\alpha.\lambda\beta.\alpha)\ 4\ 5$$
$$4$$

The formal specification of the translation of mini-language F into the λ-calculus is given in Appendix 2d. There, the function '<u>TRANSLATE</u>' defines the mapping of abstract programs into the λ-calculus. A pair <p:q> is specified as a member of the set 'ABSTRACT PROGRAM:TRANSLATION' if p is a member of the set 'ABSTRACT PROGRAM' and q is the mapping of p into the λ-calculus as specified by the function '<u>TRANSLATE</u>'.

5. DISCUSSION

Production systems have placed under a single framework the complete definition of the syntax and translation of a computer language. Not once was it necessary to introduce concepts 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 computer languages. The notation, the abbreviations, and the conceptual view of using production systems have undergone several stages of evolution.

Besides simplicity, such attendant qualities like naturalness, perspicuity, and communicativeness have been accorded due allowance. Necessarily, I have used my personal discretion in weighing these qualities. It is inevitable 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.

As mentioned in the introduction, the syntax of one complete language, ALGOL 60, has been specified by a production system, and a paper discussing this production system is being prepared. When viewed in its most restrictive interpretation, the syntax of ALGOL

60 is complicated. The variety of predicates and functions needed to specify ALGOL 60, as well as the variety of other definitions attempted with production systems, have had a major effect on the notation, abbreviations, and conceptual view of production systems presented here. Although the examples in this paper were contrived mainly to illustrate the *formalism* of production, at least some experience exercising production systems to define more general cases of syntax and translation has been obtained. Nevertheless, the critical test of the acceptability of production systems to define the syntax and translation of complete computer languages awaits further exploration.

Production systems can be used to specify definitions and string transformations much different from those given here. For example, the ALGOL 60 specification mentioned above contains a formal definition of the reduction rules for the λ-calculus. Outside of this example 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 I have not investigated.

ACKNOWLEDGEMENT

To Edward Glaser, whose insight and imagination kindled much of this research.

To Dana Scott and John Wozencraft, whose mature views of computation significantly influenced the author.

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

Appendix I: PRODUCTION SYSTEM SPECIFYING THE SYNTAX OF A SUBSET OF ALGOL 60 AND ITS TRANSLATION INTO ASSEMBLER LANGUAGE

(a) Syntax: Basic Notation only

```
1.1  NUMBER      NUMBER<1>.
1.2              NUMBER<2>.
1.3              NUMBER<3>.
2.1  ID          ID<A>.
2.2              ID<B>.

3.1  PRIMARY     PRIMARY<n>          ←  NUMBER<n>.
3.2              PRIMARY<i>          ←  ID<i>.
3.3  ARITH EXP   ARITH EXP<p>        ←  PRIMARY<p>.
3.4              ARITH EXP<a+p>      ←  PRIMARY<p>,  ARITH EXP<a>.
3.5  STM         STM<i:=a>           ←  ID<i>,  ARITH EXP<a>.

4.1  DEC         TYPE LIST<A>.
4.2              TYPE LIST<B>.
4.3              TYPE LIST<A,B>.
4.4              DEC<integer>        ←  TYPE LIST<ℓ>.

5.   PROGRAM     PROGRAM<begin d;s end>   ←  DEC<d>, STM<s>, IDS<s:i_s>,
                                             IDS<d:i_d>, IN<i_s:i_d>.

6.1  IN          IN<A:A>.
6.2              IN<B:B>.
6.3              IN<A:A,B>.
6.4              IN<B:A,B>.
6.5              IN<xy:ℓ>  ←  IN<x:ℓ>, IN<y:ℓ>, TYPE LIST<ℓ>.

7.1  NON ID      NON ID<+>.
7.2              NON ID<:=>.
7.3              NON ID<,>.
7.4              NON ID<integer>.
7.5              NON ID<n>           ←  NUMBER<n>.

8.1  IDS         IDS<i:i>            ←  ID<i>.
8.2              IDS<xiy:i,z>        ←  ID<i>, IDS<xy:z>.
8.3              IDS<xry:z>          ←  NON ID<r>, IDS<xy:z>.
```

(b) Syntax : with additions to notation

```
                 begin NUMBER<n>, ID<i>, PRIMARY<p>, ARITH EXP<a>, STM<s>,
                       TYPE LIST<ℓ>, DEC<d>;

1.   NUMBER      NUMBER<1>,<2>,<3>.
2.   ID          ID<A>,<B>.

3.1  PRIMARY     PRIMARY<n>,<i>.
3.2  ARITH EXP   ARITH EXP<p>,<a+p>.
3.3  STM         STM<i:=a>.

4.1  DEC         TYPE LIST<A>,<B>,<A,B>.
4.2              DEC<integer ℓ>.

5.   PROGRAM     PROGRAM<begin d;s end>     ←     IN<IDS<s>:IDS<d>>.
```

```
6.1  IN           IN<A:A>,<B:B>,<A:A,B>,<B:A,B>.
6.2               IN<xy:ℓ>                          ←    IN<x:ℓ>,<y:ℓ>.

7.   NON ID       NON ID<+>,<:=>,<,>,<integer>,<n>.

                  begin φ = IDS,  NON ID<r>;
8.1  IDS          φ<i>     = <i>
8.2               φ<xiy>   = <i,φ<xy>.
8.3               φ<xry>   = <φ<xy>>
                  end
```

(c) Translation

```
                  begin φ = TRANSLATE  φa = TRANS ARITH EXP,  φp = TRANS PRIMARY;

9.1  (program)    φ<begin d; s end> = <*ASSEMBLER LANGUAGE PROGRAM
                                          BALR     15,0     *SET BASE REGISTER
                                          USING    *,15     *INFORM ASSEMBLER
                                     φ<s>

                                          SVC      0        *RETURN TO SUPERVISOR
                                     *STORAGE FOR IDENTIFIERS
                                     φ<d>
                                          END>.

9.2  (dec)        φ<integer A>       = <A    DS    F>.
9.3               φ<integer B>       = <B    DS    F>.
9.4               φ<integer A,B>     = <A    DS    F
                                        B    DS    F>.

9.5  (stm)        φ<i:= a>           = <φa<a>
                                             ST    1,i            *STORE RESULT IN i>.
9.6  (arith exp)  φa<a+p>            = <φa<a>
                                             A     1,φp<p>        *ADD p>.
9.7               φa<p>              = <     L     1,φp<p>        *LOAD p>.

9.8  (primary)    φp<n>              = <=F'n'>.
9.9               φp<i>              = <i>.
                  end


10.  PROG:TRANS   PROGRAM:TRANSLATION<x:y>  ←  PROGRAM<x>,  TRANSLATE<x>=<y>.

                  end
```

Appendix 2: PRODUCTION SYSTEM SPECIFYING SYNTAX OF MINI-LANGUAGE F ANO ITS TRANSLATION INIO THE λ-CALCULUS

(a) Concrete Syntax

$\underline{\text{begin}}$ $\text{DIGIT}\langle d\rangle$, $\text{NAT NUM}\langle n\rangle$, $\text{ID}\langle i\rangle,\langle j\rangle$, $\text{IDLIST}\langle \ell\rangle,\langle \ell_1\rangle,\langle \ell_2\rangle$,
$\text{EXP LIST}\langle \ell_e\rangle$, $\text{EXP}\langle e\rangle,\langle e_1\rangle,\langle e_2\rangle,\langle e_3\rangle,\langle e_4\rangle$, $\text{UNIT EXP}\langle u\rangle$,
$\text{LET EXP}\langle t\rangle$, $\text{COMBINATION}\langle c\rangle$;

I.1	NAT NUM	$\text{DIGIT}\langle 0\rangle,\langle 1\rangle,\ \dots\ ,\langle 9\rangle.$
I.2		$\text{NAT NUM}\langle d\rangle,\langle nd\rangle.$
2.1	ID	$\text{ID}\langle A\rangle,\langle B\rangle,\ \dots\ ,\langle Z\rangle.$
2.2	IDLIST	$\text{IDLIST}\langle i\rangle,\langle i,\ell\rangle.$
2.3	EXPLIST	$\text{EXPLIST}\langle e\rangle,\langle e,\ell_e\rangle.$
3.1	UNIT EXP	$\text{UNIT EXP}\langle +\rangle,\langle \text{IF}\rangle,\langle n\rangle,\langle i\rangle.$
3.2	LET EXP	$\text{LET EXP}\langle \underline{\text{let}}\ i = e_1\ \underline{\text{in}}\ e_2\rangle,\langle \underline{\text{let}}\ i(\ell) = e_1\ \underline{\text{in}}\ e_2\rangle \ \leftarrow\ \text{DIFF IDLIST}\langle \ell\rangle.$
3.4	COMBINATION	$\text{COMBINATION}\langle e(\ell_e)\rangle,\langle [e_1+e_2]\rangle,\langle [e_1 \geq e_2 \Rightarrow e_3\ \underline{\text{else}} \Rightarrow e_4]\rangle.$
3.5	EXP	$\text{EXP}\langle u\rangle,\langle t\rangle,\langle c\rangle.$
4.	PROGRAM	$\text{PROGRAM}\langle e\rangle \ \leftarrow\ \text{NULL LIST}\langle \underline{\text{FREE IDS}}\langle e\rangle\rangle.$

$\underline{\text{begin}}\ \phi = \underline{\text{FREE IDS}};$

5.1	FREE IDS	$\phi\langle +\rangle$	=	$\langle \wedge\rangle.$
5.2		$\phi\langle \text{SELECT}\rangle$	=	$\langle \wedge\rangle.$
5.3		$\phi\langle n\rangle$	=	$\langle \wedge\rangle.$
5.4		$\phi\langle i\rangle$	=	$\langle i\rangle.$
5.5		$\phi\langle e,\ell_e\rangle$	=	$\langle \underline{\text{LIST}}\langle \phi\langle e\rangle : \phi\langle \ell_e\rangle\rangle\rangle.$
5.6		$\phi\langle \underline{\text{let}}\ i{=}e_1\ \underline{\text{in}}\ e_2\rangle$	=	$\langle \underline{\text{LIST}}\langle \phi\langle e_1\rangle : \underline{\text{REL COMP}}\langle \phi\langle e_2\rangle : i\rangle\rangle\rangle.$
5.7		$\phi\langle \underline{\text{let}}\ i(\ell){=}e_1\ \underline{\text{in}}\ e_2\rangle$	=	$\langle \underline{\text{LTST}}\langle\ \underline{\text{REL COMP}}\langle \phi\langle e_1\rangle : \ell\rangle : \underline{\text{REL COMP}}\langle \phi\langle e_2\rangle : i\rangle\rangle\rangle.$
5.8		$\phi\langle e(\ell_e)\rangle$	=	$\langle \underline{\text{LIST}}\langle \phi\langle e\rangle : \phi\langle \ell_e\rangle\rangle\rangle.$
5.9		$\phi\langle [e_1+e_2]\rangle$	=	$\langle \phi\langle e_1,e_2\rangle\rangle.$
5.10		$\phi\langle [e_1 \geq e_2 \Rightarrow e_3\ \underline{\text{else}} \Rightarrow e_4]\rangle$	=	$\langle \phi\langle e_1,e_2,e_3,e_4\rangle\rangle.$

$\underline{\text{end}}$

$\underline{\text{begin}}\ \phi{=}\underline{\text{LIST}}$

6.1	LIST	$\phi\langle \wedge : \wedge\rangle$	=	$\langle \wedge\rangle$
6.2		$\phi\langle \wedge : \ell\rangle$	=	$\langle \ell\rangle$
6.3		$\phi\langle \ell : \wedge\rangle$	=	$\langle \ell\rangle$
6.4		$\phi\langle \ell_1 : \ell_2\rangle$	=	$\langle \ell_I,\ell_2\rangle$

$\underline{\text{end}}$

$\underline{\text{begin}}\ \phi{=}\underline{\text{REL COMP}}$

7.1	REL COMP	$\phi\langle \wedge : \wedge\rangle$	=	$\langle \wedge\rangle.$	
7.2		$\phi\langle \wedge : \ell\rangle$	=	$\langle \wedge\rangle.$	
7.3		$\phi\langle i : \wedge\rangle$	=	$\langle i\rangle.$	
7.4		$\phi\langle i : \ell\rangle$	=	$\langle \wedge\rangle$	$\leftarrow\ \text{IN}\langle i : \ell\rangle.$
7.5		$\phi\langle i : \ell\rangle$	=	$\langle i\rangle.$	$\leftarrow\ \text{NOT IN}\langle i : \ell\rangle.$
7.6		$\phi\langle i,\ell_1 : \ell\rangle$	=	$\langle \underline{\text{LIST}}\langle \phi\langle i : \ell\rangle : \phi\langle \ell_1 : \ell\rangle\rangle\rangle$	

$\underline{\text{end}}$

```
8.1  IN            IN<i:i>,<i:i,ℓ>.
8.2                IN<i:j,ℓ>            ←   IN<i:ℓ>.
8.3  NOT IN        NOT IN<i:j>          ←   DIFF ID<i:j>.
8.5                NOT IN<i:j,ℓ>        ←   DIFF ID<i:j>,  NOT IN<i:ℓ>.

9.1  NULL LIST     NULL LIST<∧>.
9.2  DIFF IDLIST   DIFF IDLIST<i>
9.3                DIFF IDLIST<i,ℓ>     ←   NOT IN<i:ℓ>, DIFF IDLIST<ℓ>.

10.  DIFF ID       DIFF ID<A:B>,<A:C>, ... ,<Z:Y>.
                   end
```

(b) Abstract Syntax

```
                   begin NAT NUM<n>, ID<i>,<j>, IDLIST<ℓ>,<ℓ_1>,<ℓ_2>, EXP LIST<ℓ_e>,
                         EXP<e>,<e_1>,<e_2>, UNIT EXP<u>, LET EXP<t>, COMBINATION<i>,
                         FCN<f>, ADD FCN<f_a>, IF FCN<f_i>;
1.1  IDLIST        IDLIST<(HD i, TL ∧)>.
1.2                IDLIST<(HD i, TL ℓ)>.
1.3  EXPLIST       EXPLIST<(HD e, TL ∧)>.
1.4                EXPLIST<(HD e, TL ℓ_e)>.

2.1  UNIT EXP      UNIT EXP<f_a>,<f_i>,<n>,<i>.
2.2  LET EXP       FCN<(BIDS ℓ,BODY e)>.              ←   DIFF IDLIST<ℓ>.
2.3                LET EXP<(BID i,DEF e_1,BEXP e_2)>.
2.4                LET EXP<(BID i,DEF f,BEXP e_2)>.
2.5  COMBINATION   COMBINATION<(RATOR e,RAND ℓ_e)>.
2.6  EXP           EXP<u>,<t>,<i>.

3.   PROGRAM       ABSTRACT PROGRAM<e>                ←   NULL LIST<FREE IDS<e>>.

                   begin ϕ=FREE IDS
4.1  FREE IDS      ϕ<f_a>  =  <∧>.
4.2                ϕ<f_i>  =  <∧> .
4.3                ϕ<n>    =  <∧>.
4.4                ϕ<i>    =  <(i,∧)>.
4.5                ϕ<ℓ_e>  =  <ϕ<HD<ℓ_e>>>.                  ←  NULL LIST<TL<ℓ_e>>.
4.6                ϕ<ℓ_e>  =  <LIST<ϕ<HD<ℓ_e>>:ϕ<TL<ℓ_e>>>>  ←  EXPLIST<TL<ℓ_e>>.
4.7                ϕ<f>    =  <REL COMP<ϕ<BODY<f>>:BIDS<f>>>.
4.8                ϕ<t>    =  <LIST<ϕ<DEF<t>>:REL COMP<ϕ<BEXP<t>>:BID<t>>>>.
4.9                ϕ<c>    =  <LIST<ϕ<RATOR<c>>:ϕ<RAND<c>>>>.
                   end

                   begin ϕ=LIST
5.1  LIST          ϕ<∧:∧> = <∧>.
5.2                ϕ<∧:ℓ> = <ℓ>.
5.3                ϕ<ℓ:∧> = <ℓ>.
5.4                ϕ<i:ℓ> = <(i,ℓ)>.
5.5                ϕ<ℓ_1:ℓ_2>  = <ϕ<HD<ℓ_1>:ϕ<TL<ℓ_1>:ℓ_2>>>.
                   end

                   begin ϕ=REL COMP
6.1  REL COMP      ϕ<∧:∧> = <∧>.
6.2                ϕ<∧:ℓ> = <∧>.
6.3                ϕ<i:∧> = <i>.
6.4                ϕ<i:ℓ> = <∧>                     ←  IN<i:ℓ>
6.5                ϕ<i:ℓ> = <i>                     ←  NOT IN<i:ℓ>
6.6                ϕ<ℓ_1:ℓ>= <LIST<ϕ<HD<ℓ_1>:ℓ>:ϕ<TL<ℓ_1>:ℓ>>>.
```

```
7.1  IN           IN<i:(i,∧)>,<i:(i,ℓ)>.
7.2               IN<i:(j,ℓ)>                          ← IN<i:ℓ>.
7.3  NOT IN       NOT IN<i:(j,∧)>                      ← DIFF ID<i:j>.
7.4               NOT IN<i:(j,ℓ)>                      ← DIFF ID<i:j>, NOT IN<i:ℓ>.

8.1  NULL LIST    NULL LIST<∧>.
8.2  DIFF IDLIST  DIFF IDLIST<(i,∧)>.
8.3               DIFF IDLIST<(i,ℓ)>                   ← NOT IN<i:ℓ>, DIFF IDLIST<ℓ>.
```

(c) Function "CONCRETIZE" mapping abstract programs into a concrete representation

```
                  begin  = CONCRETIZE
9.1  CONCRETIZE   φ<u>   =  <u>

9.2               φ<ℓe>  =  <φ<HD<ℓe>>>                ← NULL LIST<TL<ℓe>>
9.3               φ<ℓe>  =  <φ<HD<ℓe>>,φ<TL<ℓe>>>      ← EXP LIST<TL<ℓe>>

9.4               φ<t>   =  <letφ<BID<t>> = φ<DEF<t>>
                             in  φ<BEXP<t>>>           ← EXP<DEF<t>>

9.5               φ<t>   =  <letφ<BID<t>>(φ<BIDS<DEF<t>>>) = φ<BODY<DEF<t>>>
                             in φ<BEXP<t>>>            ← FCN<DEF<t>>.

9.6               φ<c>   =  <φ<RATOR<c>>(φ<RAND<c>>)>

9.7               φ<c>   =  <[φ<HD<ℓe>>+φ<TL<ℓe>>]    ← RATOR<c>=<fa>, RAND<c>=<ℓe>.
9.8               φ<c>   =  <[φ<HD<ℓe>> ≥ φ<HD<TL<ℓe>> => φ<HD<TL<TL<ℓe>>>>
                                 else                 => φ<HD<TL<TL<TL<ℓe>>>>>]>
                                                             ← RATOR<c>=<fi>,
                                                               RAND<c>= <ℓe>.
                  end


10.1              ADD FCN<+>.
10.2 IF FCN       IF FCN<IF>.
10.3 NAT NUM      DIGIT<0>,<1>, ... ,<9>.
10.4              NAT NUM<d>,<nd>.
10.5 ID           ID<A>,<B>, ... ,<Z>.
10.6 DIFF ID      DIFF ID<A:B>,<A:C>, ... ,<Z:Y>.


11.  PROG:CONC    ABSTRACT PROGRAM: CONCRETIZATION<p:q>

                                  ← ABSTRACT PROGRAM<p>, CONCRETIZE<p>=<q>.
```

(d) Translation of Abstract Programs into λ-Calculus

```
                    begin φ = TRANSLATE;

12.1 TRANSLATE      φ<n>   = <n>.
12.2                φ<i>   = <i>.
12.3                φ<f_a> = <f_a>.
12.4                φ<f_i> = <(λα.λβ.λπ_1.λπ_2. ≥ α β π_1 π_2)>.
12.5                φ<ℓ_e> = <φ<HD<ℓ_e>>>.                          ← NULL LIST<TL<ℓ_e>>.
12.6                φ<ℓ_e> = <φ<HD<ℓ_e> φ<TL<ℓ_e>>>                 ← EXP LIST<TL<ℓ_e>>.
12.7                φ<f>   = < (CONS PREFIX<BIDS<f>>.φ<BODY<f>>)>.
12.8                φ<t>   = <(λφ<BID<t>>.φ<BEXP<t>>) φ<DEF<t>>>.
12.9                φ<i>   = <(φ<RATOR<c>>φ<RAND<c>>)>.
                    end

                    begin φ = CONS PREFIX

13.1 CONS PREFIX    φ<ℓ>   = <λHD<ℓ>>             ←   NULL LIST<TL<ℓ>>.
13.2                φ<ℓ>   = <λHD<ℓ>.φ<TL<ℓ>>>.   ←   IDLIST<TL<ℓ>>.
                    end


14.  PROGRAM:       ABSTRACT PROGRAM:TRANSLATION<p:q>  ←
        TRANS                          ABSTRACT PROGRAM<p>,  TRANSLATE<p>=<q>.
```

REFERENCES

The following works describe the theoretical foundations of production systems:

1. Emil L. Post
Formal Reductions of the General Combinatorial Decision Problem
American Journal of Mathematics, Volume 65, pp. 197-215, 1943.

2. Raymond M. Smullyan
Theory of Formal Systems
Annals of Mathematical Studies, Number 47, Princeton University Press, Princeton, New Jersey, 1961.

The following references describe work on applications of related formal systems:

3. John J. Donovan
Investigations in Simulation and Simulation Languages, Ph.D. dissertation, Yale University, New Haven, Connecticut, 1966.
This reference adapts Smullyan's formal system to specify the syntax of computer languages, and introduces the term 'canonic systems' to describe the resulting variant.

4. Henry F. Ledgard
A Formal System for Defining the Syntax and Semantics of Computer Languages.
MAC-TR-60 (Ph.D. dissertation) Project MAC, M.I.T., Cambridge, Massachusetts, 1969.
This reference applies production systems (here called 'canonical' systems) to define both the syntax of a computer language and its translation into a target language.

5. John J. Donovan and Henry F. Ledgard
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.
This reference also considers the use of canonic systems to define the syntax and translation of a computer language.

The following references describe the theory of the λ-calculus.

6. Alonzo Church
The Calculi of Lambda-Conversion
Annals of Mathematical Studies, Number 6, Princeton University Press, Princeton, New Jersey 1941.

7. Haskell B. Curry and Robert Feys
Combinatory Logic
Volume I, North-Holland Publishing Company, Amsterdam, 1958.

8. John M. Wozencraft
Class Notes for 'Programming Linguistics,'
Subject 6.231, M.I.T., Spring Term, 1968.

The following references have also been used.

9. Peter Naur (Editor)
Revised Report on the Algorithmic Language ALGOL 60
Communications of the ACM, Volume 6, Number 1, pp. 1-23, 1963.

10. Peter J. Landin
The Next 700 Programming Languages
Communications of the ACM, Volume 9, Number 3, 1966.

11. Peter J. Landin
A Correspondence Between ALGOL 60 and Church's Lambda-Notation.
Communications of the ACM, Volume 8, Numbers 2 and 3, February, 1965.

12. John M. McCarthy
A Formal Description of a Subset of ALGOL
in *Formal Language Description Languages for Computer Programming* (T. B. Steel - editor)
North Holland Publishing Company, Amsterdam 1966.

13. P. Lucas, P. Lauer and H. Stigleitner
Method and Notation for the Formal Definition of Programming Languages
IBM Technical Report 25.087, IBM Laboratory, Vienna 1968.

14. J. W. de Bakker
Semantics of Programming Languages
Mathematical Centre, Amsterdam

15. Willard V. Quine
Mathematical Logic
Harper and Row, New York, 1951.

16. A. M. Turing
On Computable Numbers with an Application to the Entscheidungsproblem
Proceedings of the London Mathematical Society, Volume 42, pp. 230-265, 1936.

17. A. M. Turing
Computability and Lambda-Definability
Journal of Symbolic Logic, Volume 4, pp. 153-160, 1937.

18. Stephen C. Kleene
Lambda-Definability and Recursiveness
Duke Mathematical Journal, Volume 2, pp. 340-353, 1936.

19. Trenchard More
Relations Between Simplicational Calculi
Ph.D. dissertation, M.I.T., Cambridge, Massachusetts, 1962.

20. Henry F. Ledgard
10 Mini-Languages in Need of Formal Definition
Informal Paper, Programming Research Group, Oxford University, 1969.

21. ----------
A Programmer's Introduction to the IBM System 1360 Architecture, Instructions, and Assembler Language,
International Business Machines Corporation, White Plains, New York, 1965.