A FORMAL SYSTEM
FOR DEFINING THE SYNTAX AND SEMANTICS
OF COMPUTER LANGUAGES

by

HENRY FRANCIS LEDGARD

B.S., Tufts University
(1964)
S.M., Massachusetts Institute of Technology
(1965)
E.E., Massachusetts Institute of Technology
(1967)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF
PHILOSOPHY
at the
MASSACHUSETTS INSTITUTE OF
TECHNOLOGY
February, 1969

Signature of Author Henry F Ledgard
Department of Electrical Engineering
February 24, 1969

Edward L. [illegible]

Certified by John M Wozencraft
Thesis Supervisor

Accepted by ____________________
Chairman, Departmental Committee
on Graduate Students

A FORMAL SYSTEM
FOR DEFINING THE SYNTAX AND SEMANTICS
OF COMPUTER LANGUAGES

by

Henry Francis Ledgard

Submitted to the Department of Electrical Engineering on
February 24, 1969 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

ABSTRACT

The thesis of this dissertation is that formal definitions of the syntax and semantics of computer languages are needed. This dissertation investigates two candidates for formally defining computer languages:

(1) the formalism of canonical systems for defining the syntax of a computer language and its translation into a target language, and

(2) the formalisms of the λ-calculus and extended Markov algorithms as a combined formalism used as the basis of a target language for defining the semantics of a computer language.

Formal definitions of the syntax and semantics of SNOBOL/1 and ALGOL/60 are included as examples of the approach.

Thesis Supervisor: Edward L. Glaser
Title: Associate Professor of Electrical Engineering, M.I.T.
(currently Chairman, Department of Information and
Computer Sciences, Case Western Reserve University)

ACKNOWLEDGEMENT

To Professor Edward Glaser, whose insight and imagination have sparked my enthusiasm and prompted many major developments throughout this dissertation;

To Professor John Wozencraft, whose warm guidance and penetrating criticisms have motivated a standard that this dissertation can only approximate;

To Professor Robert Graham, whose practical understanding of computer languages has helped initiate and direct this dissertation;

To Peter Landin, who patiently devoted hours teaching me his ideas on computer languages;

To Professor John Donovan, for his collaboration on canonic systems;

To Calvin Mooers, for many lively discussions on key issues;

To Leon Groisser, for his wise and thoughtful comments on my life as a student;

And to my parents, whose lifelong support has been invaluable.

"Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01). Reproduction in whole or in part is permitted for any purpose of the United States Government."

A Virtuoso Typist: Mrs. Lila S. Hartmann

STATEMENT OF ORIGIN

I gratefully acknowledge the following men, upon whose work this dissertation is heavily based. In particular:

a. The formalism of canonical systems is due to Emil Post and Raymond Smullyan.

b. The application of "canonic" systems to specify the syntax of a computer language was first made by John Donovan.

c. The notion of a defining canonical system and its use in formalizing derivations appeared earlier in works by Smullyan and Donovan.

d. The formalism of the λ-calculus is due to Alonzo Church.

e. The application of the λ-calculus to define a portion of the semantics of a computer language was first made by Peter Landin.

f. The characterizations of the semantics of ALGOL/60 and of the evaluator for the target language are based in part on similar characterizations by Landin.

g. The formalism of Markov algorithms is due to A. A. Markov.

h. The notion of adding string variables to Markov algorithms is due to A. Caracciolo.

The application and integration of the above work to define the syntax and semantics of computer languages is the principal contribution of this dissertation. In particular:

a. The application of canonical systems to define the translation of computer languages is due to the author.

b. The application of defining canonical systems to define notational abbreviations is new.

c. The notation for canonical systems and the uniform notation for defining canonical systems are for the most part new.

d. The application of the λ-calculus and (extended) Markov algorithms to define the primitive functions in a computer language is new.

e. The application of (extended) Markov algorithms to define the operation of an evaluator for the target language for characterizing semantics is new.

f. The definitions of the syntax and semantics of SNOBOL/1 and ALGOL/60 are new.

TABLE OF CONTENTS

Page

I. INTRODUCTION 9

II. CANONICAL SYSTEMS: A SELF-EXTENDING FORMALISM FOR SPECIFYING THE SYNTAX OF A COMPUTER LANGUAGE AND ITS TRANSLATION INTO A TARGET LANGUAGE . . . 12

1. Canonical Systems. 15
   a. The Basic Formalism. 15
   b. Application to Specify Syntax. 22
   c. Application to Specify Translation . . . 28
2. Defining Canonical Systems 32
   a. The Notion of a Defining Canonical System. 32
   b. Application to Derive Syntactically Legal Programs and Their Translations. . 34
   c. Application to Specify Notational Abbreviations 39
3. Discussion 44

III. EXTENDED MARKOV ALGORITHMS AND λ-CALCULUS: A COMBINED FORMALISM USED AS THE BASIS OF A LANGUAGE FOR DEFINING SEMANTICS. 49

1. The Target Language. 51
   a. Extended Markov Algorithms 51
   b. The λ-Calculus 60
   c. The Marriage of Extended Markov Algorithms to the λ-Calculus 63
   d. The Target Language. 74
2. An Evaluator for the Target Language 84
3. Discussion 100

IV. A SPECIFICATION OF THE SYNTAX AND SEMANTICS OF SNOBOL/1. 103

V. A SPECIFICATION OF THE SYNTAX AND SEMANTICS OF ALGOL/60. 122

VI. DISCUSSION 161

REFERENCES 197

BIOGRAPHICAL NOTE. 203

APPENDICES

Page

1. CANONICAL SYSTEMS
   1.1 Canonical System Specifying the Syntax of a Subset of ALGOL/60. 168
   1.2 Canonical System Specifying the Translation of the Subset into Assembler Language 169
   1.3 A Defining Canonical System for the Subset. . 170
   1.4 Derivation of a Syntactically Legal Program and its Translation into Assembler Language . 172

2. THE TARGET LANGUAGE
   2.1 Canonical System Specifying the Translation of the ALGOL/60 Subset into the Target Language. 173
   2.2 Definition of the Primitive Functions in the Subset 173
   2.3 Definition of an Evaluator for the Target Language 174

3. SNOBOL/1
   3.1 Canonical System Specifying Syntax. 176
   3.2 Canonical System Specifying Translation into the Target Language. 177
   3.3 Definition of Primitive Functions 178

4. ALGOL/60
   4.1 Canonical System Specifying Syntax. 180
   4.2 Canonical System Specifying Translation into the Target Language. 183
   4.3 Definition of Primitive Functions 185

5. THEORETICAL BACKGROUND OF CANONICAL SYSTEMS. . . . 189

ILLUSTRATIONS

Cartoon Based on "Machines Should Work, People Should Think," Slogan from IBM Television and Magazine Advertisements 8

Vending Machine of the Future. 204

DEFINITIONS

The following words are used like household words in this dissertation:

Symbol: A character or any indivisible sequence of characters.

Alphabet: A set of symbols.

String: A sequence of symbols on an alphabet.

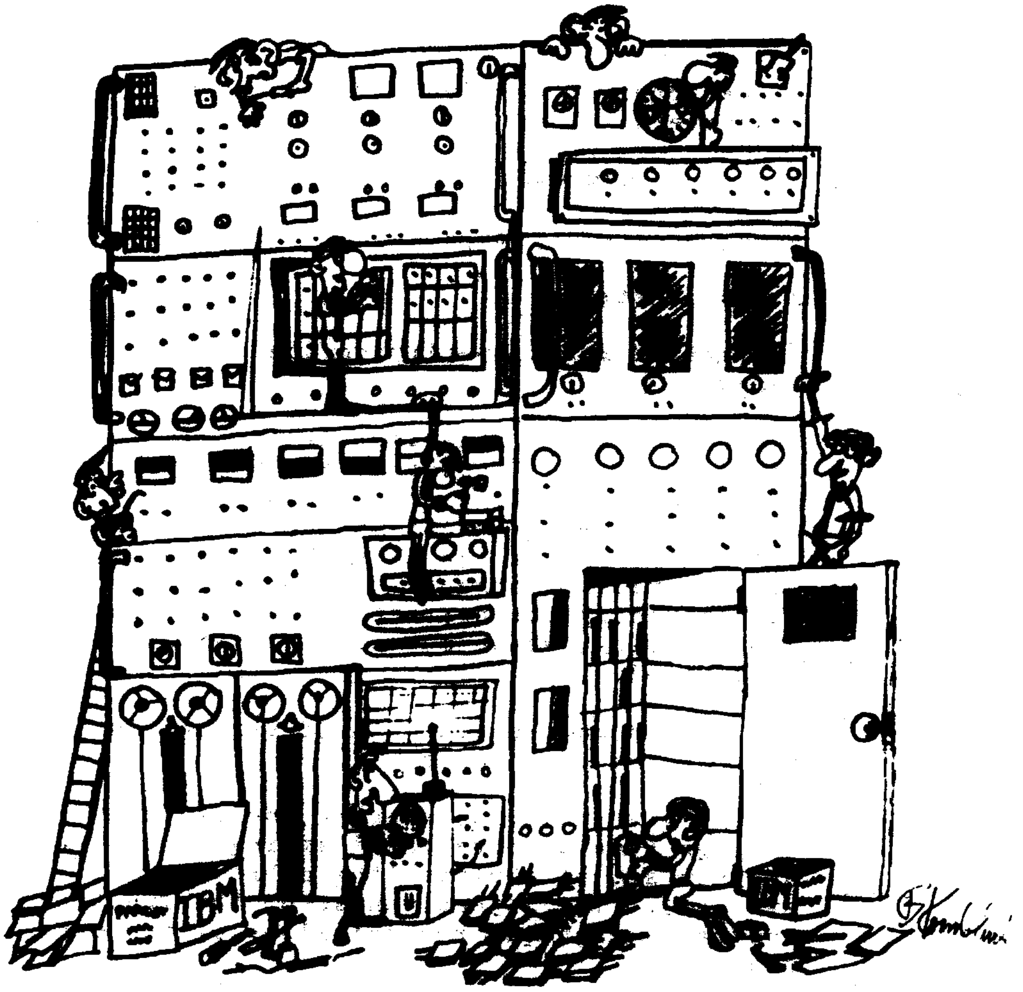
Language: A set of strings.

Syntax: The set of rules specifying the strings in a language.

Semantics: The set of rules relating the strings in a language to the "behavior" or "objects" that the strings denote. For a computer language implemented by translating the strings in the language into strings in a target language, the behavior or objects that a string denotes is defined by the corresponding target language string, whose meaning is presumably understood.

Translation: A function mapping one set of strings into another set of strings.

Abbreviation: A bijective function mapping one set of strings (the unabbreviated strings) into another set of strings (the abbreviated strings). The bijectiveness of the function insures the unique reversibility of the mapping.



Machines should work, people should think.*

**slogan from IBM television and magazine advertisements*

CHAPTER I

INTRODUCTION

This dissertation has a thesis: that formal definitions of the syntax and semantics of computer languages are needed. The formal system presented here was developed as a step towards meeting this objective.

There already exist formalisms, languages, and techniques for defining syntax and semantics. To be successful, a defining mechanism (or for that matter a computer language) should be simple, do clever things, and at the same time display fundamental principles about the objects being defined. Most methods for defining computer languages do not satisfy these criteria. The objective of this dissertation was to attempt to meet these criteria, to develop a lucid and uniform method for defining computer languages. A formal approach to language definition was taken in the hope that this approach would gain a degree of precision, simplicity and theoretical power. Although these virtues are not completely satisfied in this dissertation, I believe the formal system presented here excels existing methods for defining the syntax and semantics of a computer language. The shortcomings of this approach to language definition and recommendations for future research in removing these shortcomings are discussed in the conclusions of Chapters II and III and in Chapter VI.

Research generally progresses in two directions: in the development of new theories, and in the application and simplification of existing theories. This research is a study in the second direction. In particular, an attempt has been made to keep the notation and terminology of the formal system as simple as possible. It is natural for the author of a work to introduce notation, terminology, and conventions that became convenient for him to use, but which often obscure the work and its contributions to others. This author has tried to avoid this temptation.

The formal system for defining syntax and semantics will be given in two parts. First, Chapter II presents the formalism of canonical systems, which will be used to define the syntax of a computer language and its translation into an arbitrary target language. Second, Chapter III presents the formalisms of extended Markov algorithms and the λ-calculus, which will be used as the basis for a particular target language for defining the semantics of a computer language. The semantics of the target language are specified, in turn, by giving an extended Markov algorithm definition of a function for mapping a string in the target language into a string denoting its value.

Chapters IV and V illustrate the formal system by defining the syntax and semantics of the computer languages SNOBOL/1 and ALGOL/60. In particular, Chapter IV describes SNOBOL/1 in the spirit of providing a reference manual for

SNOBOL/1, and is directed to the reader who wishes a detailed knowledge of the language. Chapter V not only explicates the formal definition of ALGOL/60 but also relates the formal definition to other languages and other methods of language definition. Finally, Chapter VI contains a discussion of the utility of the formal system in defining computer languages.

CHAPTER II

CANONICAL SYSTEMS: A SELF-EXTENDING FORMALISM FOR SPECIFYING THE SYNTAX OF A COMPUTER LANGUAGE AND ITS TRANSLATION INTO A TARGET LANGUAGE

This chapter presents the formalism of canonical systems and its application to define the syntax of a computer language and its translation into a target language.

The mathematical underpinnings of canonical systems are due to Emil Post[1] and Raymond Smullyan.[2] Canonical 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 recursvely enumerable sets. Presumably, canonical 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[30,31] and Kleene.[32] In these works the notion of functions computable by a Turing machine were asserted[30] to comprise every function or algorithm that is intuitively computable by machine, and the functions computable by a Turing machine were shown equivalent[31,32] to the set of all "general recursive" sets, which are encompassed by canonicalsystems.

The application of a logically modified variant of the formal systems of Post,[1] Smullyan,[2] and Trenchard More[38] to specify completely the syntax of a computer language was first made by John Donovan.[3,5] 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" (in recognition of Post's work[1]) to describe his formal system. Although Donovan's formal system is not used here, many ideas and techniques presented here have stemmed from Donovan's work. The name "canonical systems" is used to distinguish the formal system presented in this dissertation from the formal systems of Post, Smullyan and Donovan. A discussion of the theoretical background for canonical systems (as presented here) is given in Appendix 5. The terminology for canonical systems presented here is due to both Post[1] and Smullyan.[2] The notation for canonical systems presented here is due in part to Post,[1] Smullyan[2] and Donovan,[3] and is in large part new. Many hours were spent in developing the notation presented here in the hope that the notation would be well-suited to computer languages. Discussions with Calvin Mocers have had a major effect on the notation.

To illustrate by example the techniques used in specifying the syntax and translation of a computer language with canonical systems, a small and rather useless subset of subset of ALGOL/60[28] will be taken as a source language, while IBM

System/360 assembler language[42] will be taken as a target language. The Backus-Naur form specification of the ALGOL/60 subset is given below:

```
<DIGIT>        ::=  1|2|3
<VAR>          ::=  A|B

<PRIMARY>      ::=  <DIGIT> | <VAR>
<ARITH EXP>    ::=  <PRIMARY> | <ARITH EXP> + <PRIMARY>
<STM>          ::=  <VAR>:=<ARITH EXP>

<TYPE LIST>    ::=  A | B | A,B
<DEC>          ::=  INTEGER<TYPE LIST>

<PROGRAM>      ::=  BEGIN  <DEC> ; <STM>  END
```

This subset allows programs containing only one declaration and one limited type of arithmetic assignment statement.

The rules for constructing a canonical system definition of a computer language, the rules for abbreviating a canonical system, and the rules for deriving strings defined by a canonical system will be presented informally in Section 2.1 of this chapter using the English language. In Section 2.2 these rules will be formally stated using the notion of a defining canonical system. In particular, each underlined expression in the next section will be defined formally in Section 2.2 with a defining canonical system. I now proceed to the informal definition of canonical systems and the application of this formalism to specify the syntax and translation of a computer language.

2.1 Canonical Systems

2.1a The Basic Formalism

A canonical system consists of a collection of the following items:

(1) An alphabet A, called the object alphabet.

(2) An alphabet P, called the predicate alphabet. Each predicate in the predicate alphabet is assigned a unique positive integer called its degree.

(3) An alphabet V, called the variable alphabet.

(4) Another alphabet, which consists of six punctuation symbols, the implication sign, conjunction sign, tuple sign, delimiter sign, left bracket sign, and right bracket sign.

(5) A finite sequence of strings that are well-formed productions, 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) lower case English letters (possibly subscripted or superscripted) for variable alphabet symbols (b) strings of capital English letters, digits, and spaces, each separated by a tuple sign, for predicate alphabet symbols (c) the symbols

→	implication sign
,	conjunction sign
:	tuple sign
;	delimiter sign
<	left bracket sign
>	right bracket sign

for punctuation symbols, and (d) symbols not in alphabets (2), (3) and (4) for object alphabet symbols.

A well-formed term consists of a sequence of variable and object alphabet symbols (e.g., "a+p" and "uv"). A

well-formed term tuple consists of a sequence of terms each separated by a tuple sign and enclosed by a left and right bracket sign (e.g., "<a+p:uv>"). A *well-formed atomic formula* consists of a predicate alphabet symbol followed by a term tuple (e.g., "ARITH EXP:VARS<a+p:uv>"). A *well-formed production* consists of (a) an atomic formula followed by the delimiter sign (e.g., "ARITH OP<+>;") or (b) a sequence of atomic formulas each separated by the conjunction sign and followed by the implication sign, another atomic formula, and the delimiter sign (e.g., "PRIMARY : VARS<p:v>,
ARITH EXP:VARS<a:u> → ARITH EXP:VARS<a+p:uv>;"). An atomic formula occurring before the implication sign is called a *premise*. An atomic formula following the implication sign or occurring alone is called a *conclusion*. 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 English letters, digits, spaces, and the punctuation symbols as members of the object alphabet. Since predicate alphabet characters, the implication sign, conjunction sign, and delimiter sign cannot occur within the brackets of a term tuple, I adopt the convention that these symbols can be used in a term tuple as object alphabet symbols. Furthermore, let the quotation marks "ˋ" and "ˊ" be symbols not contained in the object alphabet. Strings containing variable alphabet symbols, the tuple sign, left bracket sign and right bracket sign can also be used as members of the object alphabet provided that the strings are enclosed by the quotation marks when used within a production. For example, consider the following productions:

```
VAR<A>;
VAR<`x´>;
VAR<v>                     → ARITH EXP:VARS<v:v,>;
VAR<v>,  ARITH:VARS<a:u> → ARITH EXP:VARS<a+v:uv,>;
```

Here, the symbols {A x + ,} enclosed in angle brackets are object alphabet symbols. The symbols {a v u} are variable alphabet symbols.

A <u>derivation</u> is a string that can be obtained from a canonical system using the following two rules:

(1) If c; is a production containing no premises, then the string c can be derived from the canonical system.

(2) If p→c; is a production with premises p, and q→d; is an instance of this production with each variable in the production replaced by some object string, and each premise in q has been previously derived, then the string d can be derived from the canonic system.

These rules can be applied to the previously given production to derive the strings

```
              VAR<A>              VAR<x>
ARITH EXP:VARS<A:A;>    ARITH EXP:VARS<A+x+A:A,x,A,>;
```

The strings derivable from a canonical system will be interpreted in the following way. A predicate will be interpreted as the name of a set; the term tuple following a predicate will be interpreted as a string that is a member of the named set. In the above case, the set "VAR" contains two members, the strings "A" and "x". The set "ARITH EXP:VARS" contains an infinite number of members, some of which are "A:A," and "A+x+A:A,x,A,". Furthermore, I will follow the convention that each string of predicate characters separated by a tuple sign will be called a predicate part, and that predicates of degree k will consist of either one or k predicate parts. In the case where a predicate of degree k consists of k predicate parts (e.g.,"ARITH EXP:VARS"), each predicate part of the predicate will be some mnemonic describing the intended interpretation of the corresponding term in the associated term tuple (e.g., in the atomic production "ARITH EXP:VARS <a+p:uv>" the string "a+p" is interpreted as an arithmetic expression and the string "uv" is interpreted as the list of variables used in the arithmetic expression). The predicate parts and terms occurring after the tuple sign in an atomic production will be called "auxiliary" predicate parts and "auxiliary" terms (in the above case the term "uv" is the auxiliary term for the auxiliary predicate part "VARS").

For example, next consider the following canonical system specifying a set named "ARITH EXP:VARS", consisting of all pairs of strings such that the first element of each pair is an arithmetic expression in the subset of ALGOL/60, and the second element of each pair is a list of the variables occurring in the arithmetic expression:*

```
1.1 DIGIT<1>;
1.2 DIGIT<2>;
1.3 DIGIT<3>;
2.1 VAR<A>;
2.2 VAR<B>;

3.1 DIGIT<d> → PRIMARY:VARS<d:Λ>;
3.2 VAR<v>   → PRIMARY:VARS<v:v,>;
3.3 PRIMARY:VARS<p:v>                         → ARITH EXP:VARS<p:v>;
3.4 PRIMARY:VARS<p:v>, ARITH EXP:VARS<a:u> → ARITH EXP:VARS
         <a+p:uv>;
```

These productions can be interpreted:

1.1 The symbol "1" is a member of the set named "DIGIT".
1.2 The symbol "2" is a member of the set named "DIGIT".
1.3 The symbol "3" is a member of the set named "DIGIT".
2.1 The symbol "A" is a member of the set named "VAR".
2.2 The symbol "B" is a member of the set named "VAR".

3.1 If "d" represents a member of the set named "DIGIT",
then the pair of strings denoted by "d:Λ" is a member of the set named "PRIMARY:VARS".
3.2 If "v" represents a member of the set named "VAR",
then the pair of strings denoted by "v:v," is a member of the set named "PRIMARY:VARS".
3.3 If the pair "p:v" represents a member of the set named "PRIMARY:VARS",
then the pair of strings denoted by "p:v" is a member of the set named "ARITH EXP:VARS".
3.4 If the pair "p:v" represents a member of the set named "PRIMARY:VARS",
and the pair "a:u" represents a member of the set named "ARITH EXP:VARS",
then the pair of strings denoted by "a+p:uv"
is a member of the set named "ARITH EXP:VARS".

or more informally:

*The symbol "Λ" denotes the null string, i.e., if P is a string then

$$P\Lambda = P = \Lambda P$$

1. The symbols "1", "2" and "3" are digits.
2. The symbols "A" and "B" are variables.

3.1 If "d" is a digit,
then "d" is a primary with a null list of variables.

3.2 If "v" is a variable,
then "v" is a primary with a list "v," of variables.

3.3 If "p" is a primary with a list of variables "v",
then "p" is an arithmetic expression with the same list of variables "v".

3.4 If "p" is a primary with a list of variables"v",
and "a" is an arithmetic expression with a list of variables "u",
then "a+p" is an arithmetic expression with a list of variables "uv".

The rules for deriving strings specified by a canonical system can be applied to these productions to conclude that (a) the set named "DIGIT" consists of three members, the symbols "1", "2" and "3", (b) the set named "PRIMARY:VARS" consists of five members, the pairs of string "1:Λ", "2:Λ", "3:Λ", "A:A,", and "B:B,", and (c) the set named "ARITH EXP:VARS" contains an infinite number of members, some of which are "A:A,", "1+2:Λ", "A+B:A,B,", and "A+1+2+A+B:A,A,B,".

Abbreviations to the Basic Notation:

Using only the basic notation for a canonical system, a specification for a computer language often becomes lengthy. It will be convenient during the course of this dissertation to abbreviate some canonical system constructions. Here, I introduce four simple and useful abbreviations, the first two of which are due to Donovan.[3,5] The ability of canonical systems to define abbreviations formally will be discussed in Section 2.2c.

1.a If c_1, c_2, ... and c_n are conclusions with identical premises p, the productions

$p{\rightarrow}c_1$; $p{\rightarrow}c_2$; ... $p{\rightarrow}c_n$;

can be abbreviated

$p{\rightarrow}c_1$, c_2, ... c_n;

1.b If c_1, c_2, ... and c_n are conclusions with no premises, the productions

c_1; c_2; ... c_n;

can be abbreviated

c_1, c_2, ... c_n;

2. If $<t_1>$,$<t_2>$, ... and $<t_n>$ are term tuples denoting members of the same set S, the atomic **formulas**

$S<t_1>$, $S<t_2>$, ... , $S<t_n>$

can be abbreviated

$S<t_1>,<t_2>$, ... $,<t_n>$

3. If p_1, p_2, ... and p_n are premises with the same conclusion c, the productions

$p_1{\rightarrow}c$; $p_2{\rightarrow}c$; ... $p_n{\rightarrow}c$;

can be abbreviated

$p_1 \mid p_2 \mid \ldots \mid p_n \rightarrow c$;

4. If a and b are different variables, and P and R are predicates, the productions

```
P<a> → R<a>;    P<a>, R<b> → R<ba>;
```

can be abbreviated

```
P<a>  →  R<SEQ(a)>;
```

Thus, the productions*

```
(a) DIGIT<1>; DIGIT<2>;  DIGIT<3>;
(b) DIGIT<p> → CHAR<p>; LETTER<p> → CHAR<p>;
       MARK<p> → CHAR<p>;
(c) DIGIT<d> → DIGIT STR<d>; DIGIT<d>, DIGIT STR<s>
       → DIGIT STR<sd>;
```

can be abbreviated

```
(a) DIGIT<1>,<2>,<3>;
(b) DIGIT<p> | LETTER<p> | MARK<p> → CHAR<p>;
(c) DIGIT<d> → DIGIT STR<SEQ(d)>;
```

The abbreviated productions may informally be read:

(a) The symbols "1", "2", and "3" are digits.
(b) If p is a digit, or p is a letter, or p is a mark, then p is a character.
(c) If d is a digit, then a sequence of digits is a digit string.

2.1b Application to Specify Syntax

I define the syntax of a language as the set of rules specifying the strings in a language. The syntax of ALGOL/60 has the requirement that the type of each variable used in program must be declared. This requirement is not handled by the Backus-Naur form specification of the ALGOL/60 subset

*Productions (b) and (c) are from the canonical system defining the syntax of ALGOL/60.

given previously. For example, the syntactically illegal string

BEGIN INTEGER B; A:=1 END

can be derived using this specification. This requirement can readily be handled with a canonical system definition of the subset by

(a) specifying with each statement an auxiliary term specifying the list of variables used in the statement,

(b) specifying with each declaration an auxiliary term specifying the list of variables declared, and

(c) adding a premise to the production for a legal program specifying that each variable occurring in the list in (a) must be contained in the list in (b).

The canonical system for the subset of ALGOL/60 is given in Appendix 1.1a. There the second element in the term tuple for a primary, arithmetic expression, statement, and declation specify the list of variables used or declared in the corresponding source language string. The restrictive premise "IN<u:v>" (production 5) insures that each of the variables in the list "u" is contained in the list of declared variables "v". For example, the following pairs of lists are members of the set named "IN" (productions 6)

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

Thus the string

```
BEGIN   INTEGER A;  A:=1   END
```

is specified by this canonical system, whereas the illegal string

```
BEGIN   INTEGER B;  A:=1   END
```

is not specified by this canonical system because the pair <A,:B,> is not a member of the set named "IN".

An Abbreviation for Specifying Syntax:

In the specification of computer languages, it will be frequently necessary to write productions that specify auxiliary lists with a given source language construction. For example, consider the productions from Appendix 1.1a

```
3.1 DIGIT<d>  →  PRIMARY:VARS<d:Λ>;
3.4 PRIMARY:VARS<p:v>, ARITH EXP:VARS<a:u>
      → ARITH EXP:VARS<a+p:uv>;
```

Here the auxiliary terms corresponding to the predicate part "VARS" specify the list of variables used in each construction. Productions like these, in which

(a) an auxiliary term for an auxiliary predicate part in a conclusion is given as "Λ", and the auxiliary predicate part does not occur in a premise (e.g., the auxiliary term "Λ" for the predicate part "VARS" in production 3.1), or

(b) an auxiliary term for an auxiliary predicate part in a premise is a variable, and the auxiliary term for the same predicate part in a conclusion contains one occurrence of the variable (e.g., the variables "u" and "v" for the predicate part "VARS" in production 3.4).

occur frequently in canonical systems for computer languages. It is convenient not to have to specify explicitly the auxiliary terms and their predicate parts in these cases. I therefore introduce the following abbreviation:

(a) If p is an auxiliary predicate part occurring only in the conclusion of a production,
and the term t corresponding to p is given as null,
then ":p" and ":t" can be deleted from the production.

(b) If p is an auxiliary predicate part occurring in a premise and a conclusion,
and the term t corresponding to the occurrence of p in the premise is given as a variable,
and the term u corresponding to the occurrence of p in the conclusion contains one occurrence of the variable,
and the variable does not occur elsewhere in the production,
then the occurrence of ":p" and ":t" in the premise and the occurrence of the variable in the conclusion can be deleted.

Thus production 3.1 above can be abbreviated

```
3.1  DIGIT<d>   →  PRIMARY:VARS<d:Λ>;
3.1' DIGIT<d>   →  PRIMARY<d>;                      (use abr a)
```

and production 3.4 above can be abbreviated

```
3.4   PRIMARY:VARS<p:v>, ARITH EXP:VARS<a:u>
         → ARITH EXP:VARS<a+p:uv>;
3.4'  PRIMARY<p>, ARITH EXP:VARS<a:u> → ARITH EXP:VARS<a+p:u>;
                                                    (use abr b)
3.4"  PRIMARY<p>, ARITH EXP<a> → ARITH EXP:VARS<a+p:Λ>;
                                                    (use abr b)
3.4"' PRIMARY<p>, ARITH EXP<a> → ARITH EXP<a+p>;    (use abr a)
```

To obtain the unabbreviated equivalent of a production to which this abbreviation has been applied, one can

(a) Write down the abbreviated production.

(b) Write down the corresponding unabbreviated predicates used in the production.

(c) Specify for each predicate part occurring only in the conclusion a corresponding null term.

(d) Specify for each predicate part occurring both in a premise and in a conclusion a term that consists of a variable that does not occur elsewhere in the production.

Using rule (c), the production corresponding to

```
(prod 3.1')    DIGIT<d>  →  PRIMARY<d>;
(predicates)   DIGIT        PRIMARY:VARS
```

can be unabbreviated

```
3.1  DIGIT<d>  →  PRIMARY:VARS<d:Λ>;
```

Using rule (d), the production corresponding to

```
(prod 3.4''')  PRIMARY<p>,    ARITH EXP<a>  → ARITH EXP<a+p>;
(predicates)   PRIMARY:VARS   ARITH EXP:VARS  ARITH EXP:VARS
```

can be unabbreviated*

```
PRIMARY:VARS<p:v>, ARITH EXP:VARS<a:u> → ARITH EXP:VARS<a+p:uv>;
```

To insure the unique reversibility of this abbreviation, the first predicate part of each different predicate must be different, and the order in which added variables occur within the conclusion must be immaterial.

*The variables "u" and "v" added to production 3.4"' need not be identical to those given in production 3.4. A production with different variables is equivalent[2] in that each defines the same set of strings.

Using this and the previously given abbreviations, the canonical system of Appendix 1.1a has been abbreviated into the canonical system of Appendix 1.1b. The abbreviated canonical system can be viewed quite differently from its unabbreviated equivalent. For example, consider the abbreviated productions

```
3.2' VAR<v>      → PRIMARY:VARS<v:v,>;
3.3' PRIMARY<p> → ARITH EXP<p>;
```

and their unabbreviated equivalents

```
3.2  VAR<v>                → PRIMARY:VARS<v:v,>;
3.3  PRIMARY:VARS<p:v> → ARITH EXP:VARS<p:v>;
```

In production 3.2, a new auxiliary term "v," is specified for the auxiliary predicate part "VARS" and this auxiliary predicate and term are specified in the abbreviated production 3.2'. In production 3.3, however, the auxiliary list of variables is carried unchanged from the premise to the conclusion, and this list is <u>not</u> specified in the abbreviated production 3.3'.

Furthermore, consider the production

```
5.   STM:VARS<s:u>,  DEC:DEC VARS<d:v>,  IN<u:v>
         → PROGRAM<BEGIN d; s END>;
```

Here the auxiliary lists of variables "u" and "v" are constrained by the premise "IN<u:v>", and hence the auxiliary predicate parts and terms for these lists occur in both the abbreviated and unabbreviated productions.

Thus the auxiliary terms referring to the lists of variables and their associated auxiliary predicate parts are explicitly specified only when a new variable is added to the list (productions 3.2, 3.5 and 4.2) or when the list is required to have certain properties (production 5.). In languages like SNOBOL/1 and ALGOL/60, where the number of auxiliary terms is large, the abbreviation just given markedly reduced the size of their canonical systems specifying syntax.

2.1c Application to Specify Translation

I define the translation of a language as the function mapping the strings in the language into strings in some other language. This function can be specified by a canonical system specifying a set of 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 canonical systems by example. The specification of the syntax of the ALGOL/60 subset has been modified to specify not only the legal strings in the subset but also their translation into IBM System/360 assembler language. This specification is given in Appendix 1.2a. There the term to the left of each ".." specifies some string in the ALGOL/60 subset, the term to the right of each ".." specifies the representation of the string in the target language. For example, the following pair of strings is a member of the set named "PROGRAM":

```
BEGIN  INTEGER A; A:=1  END..*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 canonical system includes the specification of the comment entries in the assembler statements so that (hopefully) the reader will not have to be familiar with the assembler language to understand the translation.

An Abbreviation for Specifying Translation:

Except for the specification of strings in assembler language, the canonical system defining the translation of the subset is identical to the canonical system defining the syntax of the subset. In general, since a definition of the syntax of a language specifies the legal strings in a language and a definition of the translation of a language specifies the legal strings as well as their representation in some other language, the definition of the translation of a language will encompass the definition of the syntax of a language. This similarity leads to the following abbreviation.

Let numbers be placed on the productions of the canonical systems for the syntax and translation so that a production specifying the translation of a string is given the same number as the corresponding production specifying the syntax of the string. Let p_s and p_t be identically numbered productions from the canonical systems specifying respectively the syntax and translation.

(a) If p_s and p_t are identical, then p_t can be omitted.

(b) If a premise in p_s and p_t are identical, then the premise in p_t can be omitted.

(c) If an auxiliary predicate part and corresponding term of atomic formulas with identical first predicate parts in p_s and p_t are identical, then the auxiliary predicate part and term in p_t can be omitted.

For example consider the production from the syntax of the ALGOL/60 subset

```
5.    STM:VARS<s:u>,  DEC:DEC VARS<d:v>,  IN<u:v>
           → PROGRAM<BEGIN d; s END>;
```

and the corresponding production from the translation of the subset

```
5.'   STM:VARS<s..s':u>,  DEC:DEC VARS<d..d':v>,  IN<u:v>
           → PROGRAM<BEGIN d; s END..α>;
```

where α represents the string that specifies the translation of the program. Here, using rule (b), the premise "IN<u:v>" can be omitted from the translation production, and using rule (c) the auxiliary predicate parts and terms for the lists "u" and "v" of variables can be omitted to yield the abbreviated production for the translation

```
5."  STM<s..s'>, DEC<d..d'> → PROGRAM<BEGIN d; s END..α>;
```

To obtain the unabbreviated equivalent of an abbreviated canonical system defining translation, one must add to the canonical system defining translation (a) the numbered productions that occur in the canonical system for the syntax but do not occur in the canonical system for translation (b) the premises that occur in a production for syntax but do not occur in the identically numbered productions for translation, and (c) for atomic formulas with identical first predicate parts, the auxiliary predicate parts and corresponding terms that occur in a production for syntax but do not occur in the identically numbered production for the translation.

For example, consider the abbreviated translation production just given

```
5.''  STM<s..s'>, DEC<d..d'> → PROGRAM<BEGIN d; s END..α>;
```

and the corresponding production for the syntax

```
5.    STM:VARS<s:u>,  DEC:DEC VARS<d:v>,  IN<u:v>
          → PROGRAM<BEGIN d; s END>;
```

Here, the premise "IN<u:v>" occurs in the production for the syntax but not in the production for the translation, and the auxiliary predicate parts and corresponding terms for the predicate parts "VARS" and "DEC VARS" occur in the production for the syntax but not in the production for the translation. Adding this premise and these auxiliary predicate parts and their terms to the abbreviated production 5." for the translation, we obtain the unabbreviated production

```
5.'  STM:VARS<s..s':u>,  DEC:DEC VARS<d..d':v>,  IN<u:v>
         → PROGRAM<BEGIN d; s END..α>;
```

The abbreviated canonical system specifying the translation of the ALGOL/60 subset is given in Appendix 2.1b. The abbreviated canonical system of Appendix 2.1b can be viewed quite differently from its unabbreviated equivalent. The abbreviated canonical need specify only the new terms that must be <u>added</u> to the canonical system specifying the syntax in order to convert the canonical system specifying syntax into the canonical system specifying translation. In writing the abbreviated canonical system specifying translation, the requirements needed to insure the syntactic legality of a string whose translation is being specified can be omitted. These requirements are assumed to have been specified in the canonical system for the syntax. In languages like SNOBOL/1 and ALGOL/60, where the number of syntactic requirements is large, this abbreviation greatly reduced the size of the canonical systems defining the translations of the languages into the target language.

2.2 Defining Canonical Systems

2.2a The Notion of a Defining Canonical System

The previous sections have been devoted to developing canonical systems specifying sets of strings. The strings represented syntactically legal programs in a subset of ALGOL/60 and their counterparts in assembler language. The rules for forming and using the canonical systems for these sets were described informally in the text in English. The string representing a canonical system and the rules for using the canonical system can, in turn, be specified formally by another canonical system. In cases where a conflict would arise in distinguishing the strings of the first canonical system in the productions of the defining canonical system, the strings of the first canonical system can be enclosed by the quotation marks "`" and "´".

The productions specifying the rules for constructing another canonical system are given in Appendix 1.3a. These productions specify the alphabets of object symbols, predicate symbols, and variable symbols, and the rules for constructing well-formed terms, term tuples, atomic formulas, premises, conclusions, productions, and finally, canonical systems.*

The logical notion of using a second canonical system to formalize the rules for constructing a canonical system

*In the productions of Appendix 1.3, the quotation marks have been omitted for matching pairs of left and right brackets that occur as object symbols. For example, in the atomic formula "WF TERM TUPLE<<t>>", quotation marks have been omitted from the second and third brackets. In atomic formulas of this type, the scope of the left bracket sign extends to the matching right bracket sign, and all brackets thus enclosed are considered as object symbols.

was first presented by Smullyan[2] and later by Donavan.[3] In the works presented by Smullyan and Donavan, a notation different from the basic notation is used in a defining canonical system. The advantages of using quotation marks to distinguish symbols in the defined canonical system from symbols in the defining canonical system are that (a) the same notation is used for all canonical systems, and (b) definitions and rules formalized in one canonical system can be copied and applied to other canonical systems independently of their position in a series of defined and defining canonical systems (this point will be discussed in section 2.2c).

2.2b Application to Derive Syntactically Legal Programs

The rules for deriving strings specified by a canonical system can also be formalized with a defining canonical system. These rules are given in Appendix 1.3b. By adding a production of the form "CANONICAL SYSTEM STR<c>;", where c is some well-formed canonical system, these productions define the rules for deriving strings in the canonical system c.

In particular, productions 9 specify the rules for extracting productions from the member of the set "CANONICAL SYSTEM STR". Production 10 specifies the rule for substituting strings in the object alphabet in place of the variables in the productions to obtain instances of the productions. Productions 11 specify the rules for deriving strings specified by the production instances.

Productions 10 and 11 can be viewed as a formalization of the two logical rules of inference "substitution" and "modus ponens" for deriving strings specified by a canonical system. The substitution of object strings for variables in a production occurs through the predicate "SUBST". The predicate "SUBST" define a set of 4-tuples, where the first element of each 4-tuple is a production, the second element is a variable, the third element some string of object alphabet symbols, and the fourth element the production with each occurrence of the variable replaced by the object string. For example, using the canonical system of the syntax of the ALGOL/60 subset as a member of the set "CANONICAL SYSTEM STR", the following 4-tuple can be generated as a member of the set "SUBST"

```
<DIGIT<d>→PRIMARY:VARS<d:Λ>  :  d  :  1  :  DIGIT<1>→PRIMARY:VARS<1:Λ>>
```

The application of modus ponens to the production instances of a canonical system occurs in production 11.1.

```
11.1 DERIVATION<Λ>;
11.2 DERIVATION<d>,  PROD INSTANCE<c;>,  WF CONCLUSION<c>
       → DERIVATION<d c>;
11.3 DERIVATION<d>,  PROD INSTANCE<p→c;>,
         PREMS:DERIV CONT PREMS<p:d>  →  DERIVATION<d c>;
```

These productions can be read:

```
11.1 From no premises, the null string can be derived.
11.2 If the string d has been derived,
     and c; is an instance of a production that contains no
         premises,
     then the string c can be added to the string d.
```

11.3 If the string d has been derived,
and p→c; is an instance* of a production with premises p,
and the premises p are contained in the string d,
then the string c can be added to the string d.

For example, by successively using the following production instances

```
DIGIT<1>;
DIGIT<1>   →   PRIMARY:VARS<1:Λ>;
PRIMARY:VARS<1:Λ>   →   ARITH EXP:VARS<1:Λ>;
```

the following member of the set "DERIVATION" can be generated

```
DIGIT<1> PRIMARY:VARS<1:Λ> ARITH EXP:VARS<1:Λ>
```

Another example of a member of the set "DERIVATION" is generated in the right-hand column of Appendix 1.4a. By simply asserting that the canonical system defining the syntax of the ALGOL/60 subset is a member of the set "CANONICAL SYSTEM STR" (i.e., by simply adding the production "CANONICAL SYSTEM STR <`DIGIT<1>; ... IN<y:ℓ> → IN<xy:ℓ>;´>;" to the productions of Appendices 1.3a and 1.3b), Appendix 1.3 defines the rules for deriving syntactically legal programs in the ALGOL/60 subset. The derivation of Appendix 1.4a specifies that the string   BEGIN   INTEGER A; A:=1   END

is a member of the set "PROGRAM".

Yet another example of a member of the set "DERIVATION" is generated in the right-hand column of Appendix 1.4b. By

*An instance of a production P is the production P' obtained from P by applying substitution to <u>all</u> of the variables in a production.

asserting that the canonical system defining the translation of the ALGOL/60 subset is a member of the set "CANONICAL SYSTEM STR", Appendix 1.3 defines the rules for deriving syntactically legal programs and their translation. The derivation of Appendix 1.4b specifies that the string

```
BEGIN INTEGER A; A:=1 END..*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 SUPERVISOI
                           *STORAGE FOR VARIABLES
                           A DS     F
                             END
```

is a member of the set "PROGRAM".

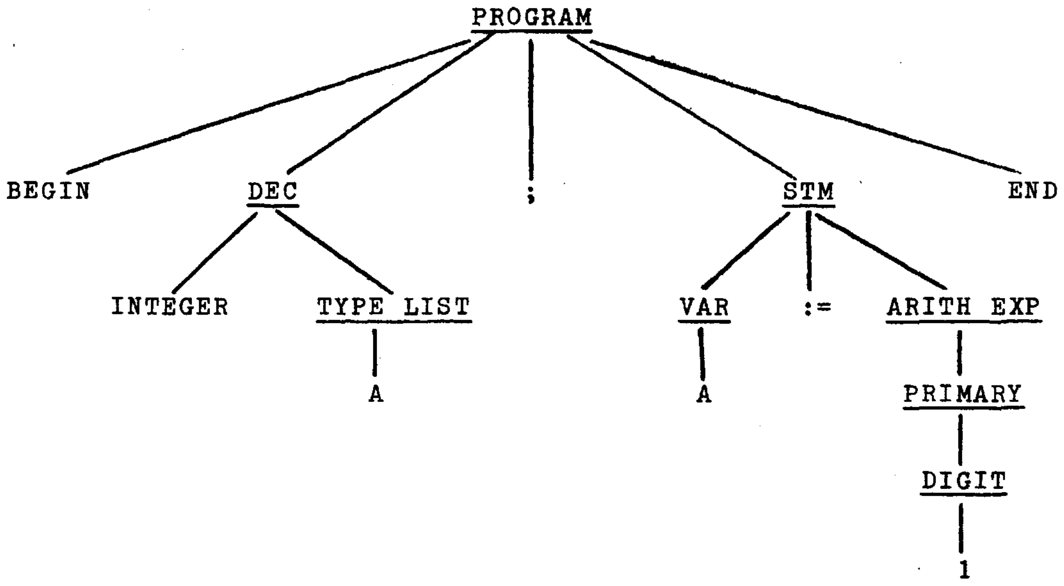
Thus by simply adding a production asserting that some well-formed canonical system is a member of the set "CANONICAL SYSTEM STR", the productions of Appendix 1.3 can be used to generate all strings defined by the canonical system.

Structural Description of Derived Strings:*

A derivation provides a "structural description" of a derived string. By a structural description[35] of a string, I mean the sequence of rules (here the sequence of productions) used in generating the string. The sequence of rules used in generating a string provides information about the structure of the string.

*This application is not used in the other sections of this dissertation.

For example, consider the derivation of Appendix 1.4a. If we consider only the first term of each derived term tuple, the derivation provides a structural description for the string "BEGIN INTEGER A; A:=1 END" that may be represented in the form of a syntactic tree:



The tree can be constructed by scanning the derivation from bottom to top and constructing the corresponding tree from the top down. The leaves of the tree are symbols from the object alphabet. The nodes of the tree are the partial predicate names occurring in derived conclusions. The branches joining a node are determined by the basic symbols and the previously derived conclusions used to construct the newly derived conclusion.

Using a canonical system for the translation of a language, a derivation can be used to construct a structural description of a target language string. The System/360 assembler language is not a "structured" language and hence the derivation of an assembler language program is not of concern. However, canonical systems have been used[4] to obtain structural descriptions of strings in a target language where knowledge of a string's tree-like structure is important for its analysis.*

2.2c Application to Specify Notational Abbreviations

I define an abbreviation as a bijective (one-to-one and onto) function mapping one set of strings (the unabbreviated strings) into another set of strings (the abbreviated strings). The bijectiveness of the function insures that we can recover the unabbreviated equivalent of each abbreviated string. I have introduced six abbreviations to the notation for canonical systems, four to the basic notation, one for a canonical system specifying syntax, and another for a canonical system specifying translation. Each of these abbreviations can be specified by a defining canonical system specifying a set of ordered pairs, where the first element of each pair is an abbreviated canonical system, and the second element is the corresponding unabbreviated canonical system.

*A canonical system derivation can lead to much more complicated structural descriptions than those that can be represented in tree-like form. I have not studied this issue.

The productions specifying the six abbreviations introduced to canonical systems are given in Appendix 1.3c. For example, productions 15.1 and 15.2 in

```
15.1 WF PROD<p→c;>                        → ABR1 P:P<p→c;:p→c;>;
15.2 WF PROD<p→c;>, ABR1 P:P<p→s;:t> → ABR1 P:P<p→c,s;:p→c;t>;
15.3 WF ATOM PROD<c;>                     → ABR1 AP:AP<c;:c;>;
15.4 WF ATOM PROD<c;>,  ABR1 AP:AP<s;:t;>
         → ABR1 AP:AP<s,c;:t;c;>;
15.5 ABR1 CS:CS<Λ;Λ>;
15.6 ABR1 CS:CS<c:d>, ABR1 P:P<p:q>    → ABR1 CS:CS<cp:dq>;
15.7 ABR1 CS:CS<c:d>, ABR1 AP:AP<p:q> → ABR1 CS:CS<cp:dq>;
```

specify a set of ordered pairs "ABR1 P:P", where the first element is a production of the form "p→c_1, c_2, ... , c_n;" and the second element is the corresponding unabbreviated productions "p→c_1; p→c_2; ... p→c_n;". Productions 15.3 and 15.4 augment this set to include atomic productions, and productions 15.5 through 15.7 specify the abbreviation for an entire canonical system.

Similarly, productions 16 through 20 specify the other five abbreviations to canonical systems.* Productions 21 and

*To apply abbreviation 20, the abbreviation for a canonical system specifying syntax, a production of the form "CS PREDICATES<p_1,p_2,... , p_n>" where the p_i, $1 \leq i \leq n$, are the unabbreviated predicate for the canonical system, must be added to productions 20.

To apply abbreviation 21, the abbreviation for a canonical system specifying translation, (a) the productions and premises occurring in the canonical system for syntax but not in the canonical system for translation must be added to the canonical system for translation, and (b) atomic formulas with identical first predicate parts from identically numbered productions from the canonical systems for the syntax and translation must be written together in the canonical system for translation and separated by "//".

22 specify abbreviations used in defining ALGOL/60 and will be discussed in the chapter on ALGOL/60. Finally, production 23 specifies the rule for converting some string (presumably a well-formed abbreviated canonical system) that is asserted to be a member of the set "ABR CANONICAL SYSTEM STR" into the corresponding member of the set "CANONICAL SYSTEM STR" (the unabbreviated equivalent of the abbreviated canonical system).*

For example, by asserting that the abbreviated canonical system of Appendix 1.1b is an abbreviated canonical system (i.e., by adding the production asserting that the canonical system of Appendix 1.1b is a member of the set "ABR CANONICAL SYSTEM STR"), the productions of Appendix 1.3c can be used to derive the conclusion that the canonical system of Appendix 1.1a is its corresponding unabbreviated equivalent (i.e., the canonical system of Appendix 1.1a is a member of the set "CANONICAL SYSTEM STR"). Similarly, by asserting that the canonical system of Appendix 1.2b is a member of the set "ABR CANONICAL SYSTEM STR", production 24. can be used to derive the conclusion that the canonical system of Appendix 1.2a is its unabbreviated equivalent.** In general, by

*The order in which abbreviations are removed from an abbreviated canonical system will generally depend on the abbreviations introduced. Production 23. defines one order in which the abbreviations introduced in this dissertation can be removed. Furthermore, any premise in production 23 that refers to an abbreviation not used in a particular abbreviated canonical system can be removed.

**As mentioned previously, an atomic production specifying the unabbreviated predicates of an abbreviated canonical system specifying syntax must be added to the defining canonical system to generate the correct unabbreviated (cont. next page)

(a) specifying the sets of ordered pairs defining some abbreviations, and

(b) adding a production like production 23 defining the rule for converting an abbreviated canonical system into its unabbreviated equivalent.

a defining canonical system can be used to generate the unabbreviated equivalent of any abbreviated canonical system. Moreover, having generated the equivalent unabbreviated canonical system, the productions of Appendix 1.3a and 1.3b can then be used to derive strings specified by the canonical system.

The productions of Appendix 1.3 are written using only the first two abbreviations to the basic notation. To define Appendix 1.3 using only the basic notation, the user could write a third canonical system, which would consist of simply (a) a production asserting that the canonical system of Appendix 1.3 is a member of the set "ABR CANONICAL SYSTEM STR", (b) productions 15 and 16 of Appendix 1.3 (these productions contain no abbreviations), and (c) the production "ABR CANONICAL SYSTEM STR<a>, ABR2 CS:CS<a:b>, ABR1 CS:CS<b:c> → CANONICAL SYSTEM STR<c>;". The user would then have a series of three canonical systems. The first (abbreviated) canonical system (e.g., Appendices 1.1b or 1.2b) would define the allowable strings in some source language. The

**(Cont. from p. 41) canonical system, and the productions of the abbreviated canonical systems specifying syntax and translation must be combined (according to the rules given earlier) to generate the complete unabbreviated canonical system specifying translation.

second canonical system would define the rules for forming the first canonical system, the rules for deriving strings specified by the first canonical system, and the rules for converting the first canonical system into the basic notation. The third canonical system would define the rules for converting the second canonical system into the basic notation. Thus, the series of canonical systems would ultimately be defined using only the basic notation. In general, a user may write a series of canonical systems to define the rules for constructing and using other canonical systems; in order for the series to be defined using only the basic canonical system notation , only the last member of the series need be written in the basic notation.

Note that productions 15 and 16 of Appendix 1.3 could be copied unchanged in the third canonical system. These productions formalize rules that are applicable to two canonical systems independently of their relative positions in a series of canonical systems. In fact, these productions can be copied and applied to the canonical system in which they themselves are given.

User-Coined Abbreviations:

Defining canonical systems provides a writer of a canonical system with a formal mechanism for introducing his own abbreviations to the notation. For example, consider the productions (from the canonical system of ALGOL/60):

```
PRIMARY<p>                                →  TERM<p>;
PRIMARY<p>,  MULT OP<m>,  TERM<t>  →  TERM<tmp>;
```

The user may wish to abbreviate these productions:

```
PRIMARY<p>,  MULT OP<m>  →  TERM<ALTSEQ(p m)>;
```

Productions 21 of Appendix 1.3c specify this abbreviation (as well as other variants of this abbreviation). Thus by simply adding new productions to the canonical system defining the conversion of a abbreviated canonical system to unabbreviated form, the notation for canonical systems can be tailored to fit a particular application.

2.3 Discussion

Canonical systems have placed under a single framework the complete definition of the syntax and translation of a language. The formalism was used to specify all legal programs, their translations into assembler language, the rules for deriving legal programs and their translations, and the rules for removing abbreviations from the specifications. Not once was it necessary to introduce concepts outside canonical systems;although some complexity was added to the formalism by introducing abbreviations to the basic notation, even the abbreviations were ultimately defined in terms of the basic formalism.

It is important to develop languages whose descriptions are concise. The Backus-Naur form specification of the ALGOL/60 subset and the English sentence describing the context-sensitive requirement provide one very concise and easily understandable description of the syntax of the subset. The canonical system of Appendix 1.1 has, in fact, been modeled after this description. Productions 1 through 5 correspond (except for the auxiliary elements generating the lists of used and declared variables) to the Backus-Naur form productions; the premise "IN<u:v>" in production 5 and the definition of the predicate "IN" formalize the context sensitive restriction stated in English.

The canonical system of Appendix 1.1 is not much more lengthy than the Backus-Naur form definition of the subset and the associated English sentence describing the context-sensitive restriction. Like Backus-Naur form, the language of canonical systems is readable. On the other hand, canonical systems have the added power to characterize completely both the syntax of a language and its translation into a target language, without resorting to the English Language. Moreover, the notation for canonicalsystems is not fixed. By changing or adding productions to a defining canonical system, the user can alter or abbreviate the notation for a defined canonical system to fit a particular language.

I wish to point out two additional features of the canonical systems of Appendices 1.1 and 1.2. First, barring any inadvertent errors, the canonical systems describe a set of ALGOL/60 programs and assembler language programs that will run on a computer when translated by an ALGOL/60 compiler or System/360 assembler. Second, the specification of the comments entries in the assembler language statements was provided not only to aid the reader. The comments are meaningful context-sensitive strings in the English language. The specification of these strings was handled as easily as the specification of the strings in assembler language. The specification of the strings in the English language illustrates the use of canonical systems to specify the entire operation of a translator, including the specification of meaningful comments. Moreover, it suggests the capacity of canonical systems to define string transformations in languages other than computer programming languages.

One use of canonical systems is in the development of a generalized translator for computer languages, i.e., a translator that is independent of both source and target languages. Canonical systems define a set by specifying rules for generating its members. To use a canonical system as a language for writing translators, an algorithm to recognize strings specified by a canonical system and output associated strings is needed. No algorithm for recognizing and constructing strings specified by a canonical system is presented in this dissertation. However, one algorithm for canonical systems has been devised and implemented by Alsop.[36]

Several important issues for using canonical systems in a generalized translator have not been studied. One critical issue is the development of a restriction on canonical systems to define only recursive sets rather than recursively enumerable sets. Theoretically, an algorithm for recognizing a string defined by a canonical system exists only if the set of strings defined by the canonical system is recursive. Other critical issues include speed of translation, recovery in case of an error in a source language program, and code optimization of target language programs. I expect that modifications to the basic formalism presented here will be necessary to use canonical systems in a generalized translator.

The notion of defining canonical systems unfolds several possibilities for using canonical system as a tool for working with computer languages. Just as a canonical system allows a user to change a source or target language construction by simply changing the productions specifying the construction, a defining canonical system allows the user to change the definition or use of a defined canonical system by simply changing productions of the defining canonical system. Although only rules for removing abbreviations from a canonical system and rules for deriving strings specified by a canonical system have been defined here, defining canonical systems may provide a flexible mechanism for embedding many _other_ rules for defining and manipulating computer languages.

As mentioned earlier, the results of this chapter apply to any recursively enumerable set. Any function or relation that is recursively enumerable can be specified by a canonical system. Canonical systems can be used to express algorithms and string transformations of a much different nature from those given here. The notion of defining canonical systems adds to the basic formalism a facility for allowing a user to formalize his own rules for defining and manipulating strings and their canonical systems. The modifications to the basic formalism presented here have been directed towards the application of canonical systems to define the syntax and translation of a language. But more importantly, canonical systems provides a definitional facility that the user has the freedom to tailor according to his own application and style.

CHAPTER III

EXTENDED MARKOV ALGORITHMS AND λ-CALCULUS: A COMBINED FORMALISM USED AS THE BASIS FOR A TARGET LANGUAGE FOR DEFINING SEMANTICS

This chapter presents a formal language (henceforth referred to as the target language) quite different from conventional machine or assembler language for defining the semantics of a computer language.

The semantics of a language can be defined as the set of rules relating the strings in a language to the behavior or objects that the strings denote. The behavior or object that a string denotes can be described by a string in some other language whose meaning is presumably understood. This approach to defining the semantics of computer languages will be taken in this chapter, namely, the presentation of a single language (whose meaning is presumably understood) for defining the semantics of multiple other languages. The semantics of a given source language will be specified by defining the translation of the language into the target language.

The semantics of the target language, however, will not be left to an English language explanation in the text. The semantics of the target language will be further explicated in Section 3.2 by giving a formal definition of a machine* that performs the computation indicated by a target language

*"Machine" in the sense of a set of logical rules.

string and produces the string denoted by the target language string. (In defining the semantics of a computer language, the word computation can be considered synonymous with the word "behavior" and all "objects" in a computer language can be considered as strings.) Thus the appeal to understanding the semantics of a computer language will be ultimately reduced to understanding the formalism in which the operation of the target language evaluating mechanism is expressed.

Generally, the semantics of different languages will be specified by giving different translations into the target language while leaving the definition of the target language evaluating mechanism unchanged. On the other hand, the definition of the evaluating mechanism can be changed to define source language constructs that appear difficult to define in the target language.*

The target language presented here is based on the formalism of Markov algorithms,[9] an extension to Markov algorithms due to Caracciolo,[10,11,12] and the formalism of the λ-calculus of Alonzo Church.[17,18] Extended Markov algorithms are used to define the primitive functions in a computer language, the λ-calculus is used to define new functions from the primitive functions. In a sense, the target language draws upon the best of each formalism. Markov algorithms explicate the notion of an algorithm operating on a string

*This was done to define indirect addressing in SNOBOL/1.

and are especially well-suited to the definition of primitive functions transforming strings into new strings. The λ-calculus explicates the notion of a function and is especially well-suited to the definition of new functions from the primitive functions.

The target language has several important properties. The language is formally based, and theorems regarding the completeness of the formalisms to define the set of all "computable" function exist.[31,32] The language is independent of the characteristics of existing computers. The basic notation for the target language is simple. Probably most importantly, the correspondence between many computer languages and the target language is somewhat simpler than the correspondence between computer languages and conventional machine or assembler languages.

3.1 The Target Language

3.1a Extended Markov Algorithms

Markov Algorithms:

Let A be an alphabet of characters, called the object alphabet, and let "→", "·" and "Λ" be characters not in A. A Markov algorithm is a finite list of substitution rules of the form

$$\begin{array}{l} s_1 \rightarrow(\cdot)\ t_1 \\ s_2 \rightarrow(\cdot)\ t_2 \\ \quad \vdots \\ s_n \rightarrow(\cdot)\ t_n \end{array}$$

where the s_i and t_i, $1 \le i \le n$, are either "Λ" or strings of object alphabet characters, and "(·)" indicates the possible occurrence of a "·" after the "→". The symbol "Λ" denotes the null string.

A Markov algorithm of the above form when applied to an object string X is taken to mean:

(a) Look down among the substitution rules for the first rule such that s_i occurs in X.

(b) If such a rule is found, replace the leftmost occurrence of s_i in X by the string t_i. If a "·" occurs after the "→" in the substitution rule, terminate the algorithm. Otherwise repeat the application of the algorithm to the newly formed string.

(c) If no such rule is found, terminate the algorithm.

For example, the Markov algorithm

$$\begin{array}{lll} B & \to & D \\ C & \to & F \\ O & \to & I \end{array}$$

transforms the string "COBBLER" into the string "FIDDLER", whereas the Markov algorithm

$$\begin{array}{lll} B & \to & D \\ C & \to\cdot & T \\ O & \to & I \end{array}$$

transforms the string "COBBLER" into the string "TODDLER".

Consider the following Markov algorithm for taking a parenthesized string of letters from the alphabet {I,O,N,X} and producing a string where the initial letters are reversed. (Here the character "*" is used as a marker, and the object alphabet consists of the characters {I O N X ( ) *}.)

```
II*    →    I*I
IO*    →    O*I
IN*    →    N*I
IX*    →    X*I

OI*    →    I*O
OO*    →    O*O
ON*    →    N*O
OX*    →    X*O

NI*    →    I*N
NO*    →    O*N
NN*    →    N*N
NX*    →    X*N

XI*    →    I*X
XO*    →    O*X
XN*    →    N*X
XX*    →    X*X

(I*    →    I(
(O*    →    O(
(N*    →    N(
(X*    →    X(

 ()    →.   Λ
  )    →    *)
```

A Markov algorithm for reversing a paranthesized string of letters {I O N X}

This algorithm when applied to the string "(NOXIN)" successively transforms it into the following strings

```
(NOXIN) → (NOXIN*) → (NOXN*I) → (NON*XI) → (NN*OXI)
        → (N*NOXI) → N(NOXI) → N(NOXI*) → N(NOI*X)
        → N(NI*OX) → N(I*NOX) → NI(NOX) → NI(NOX*)
        → NI(NX*O) → NI(X*NO) → NIX(NO)→NIX(NO*)
        → NIX(O*N) → NIXO(N) → NIXO(N*) → NIXON()
        →·NIXON
```

Even quite simple algorithms like the above become exceedingly lengthy when expressed in the Markov formalism. If the alphabet above included all 26 letters in the English alphabet, the Markov algorithm for reversing the letters in a string would require 704 substitution rules. To alleviate this growth, Caracciolo di Forino[10,11,12] in developing a Markov algorithm based language called PANON introduced the notion of a "string variable" as an extension to Markov algorithms.

Extended Markov Algorithms:

Let A and V be disjoint alphabets of characters, called respectively the object alphabet and variable alphabet, and let "→", "·" and "Λ" be characters not in A or V. Let each variable in V represent some pre-specified (possibly infinite) set of object alphabet strings. The case where different variables can represent different sets of object alphabet strings is not excluded. An extended Markov algorithm is a finite sequence of substitution rules of the

$$\begin{array}{lll} s_1 & \rightarrow(\cdot) & t_1 \\ s_2 & \rightarrow(\cdot) & t_2 \\ & \vdots & \\ s_n & \rightarrow(\cdot) & t_n \end{array}$$

where the s_i and t_i, $1 \leq i \leq n$, are either "Λ" or strings of object alphabet and variable alphabet characters such that each variable in t_i occurs also in s_i.

A string s_i represents the set of object alphabet strings computed by concatenating in order from left to right each of the object alphabet characters in s_i with any object alphabet string represented by a variable in s_i. The set represented by s_i is constrained in that each occurrence of the same variable in s_i must be set to the same object alphabet string in computing the set of concatenated object strings that s_i represents. For example, if ℓ is a string variable representing any member of the set {V W} and m is a string variable representing any member of the set {Y ZZ} the string "ℓAmAℓ" represents any member of the set {VAYAV VAZZAV WAYAW WAZZAW}.

A string s_i is said to occur within an object string X if one or more of the strings represented by s_i occurs within X. The "leftmost" occurrence of s_i in X is the string such that first, (of the occurrences of s_i in X) the occurrence begins with the leftmost object alphabet character, and second, the occurrence is as short as possible.

An extended Markov algorithm of the above form when applied to an object string X is taken to mean:

(a) Look down among the substitution rules for the first rule in which s_i occurs in X.

(b) If such a rule is found, replace the leftmost occurrence of s_i in X by the string obtained from t_i by replacing each variable in t_i by the string used in place of the variable in s_i. If a "." occurs after the "→" in the substitution rule, terminate the algorithm. Otherwise repeat the application to the newly formed string.

(c) If no such rule is found, terminate the algorithm.*

It will be convenient to introduce a special symbol after the s_i to mean that the string matched to s_i must extend to the last character of the object string. I will use the symbol "." for this purpose.**

For example, let s and s' be string variables representing any string of English letters. The extended Markov algorithm

(1) sI → sO

transforms the string "BINGO" into the string "BONGO", the extended Markov algorithm

(2) XsXs'X → ss'

*The transformation specified by a substitution rule of an extended Markov algorithm is computable only if the string variables represent recursive sets. This requirement is discussed in detail by Caracciolo (Chap. 5, ref. 11). In this dissertation all sets defined for string variables are recursive.

**This convention can be viewed solely within the framework of extended Markov algorithms by (a) replacing each "." after the s_i by a special character not in the object alphabet (b) replacing each corresponding t_i with t_i followed by the special character (c) appending to each object string X the special character, and (d) applying to the transformed object string an algorithm that simply removes the special character.

transforms the string "XABXCDX" into the string "ABCD", the extended Markov algorithm

$$(3) \qquad sXs \;\rightarrow\; X$$

transforms the string "QABXAB" into the string "QX", and the extended Markov algorithm

$$(4) \qquad \begin{array}{lll} Xs. & \rightarrow & \Lambda \\ sX & \rightarrow\cdot & X \end{array}$$

transforms the string "?VWXX?XBC" into the string "?XX?".*

More precisely, an extended Markov algorithm will be specified in three parts:

(a) A statement listing some string variables and the names of the sets whose members the variables represent.

(b) A formal definition of the sets named in (a).

(c) A list of extended Markov algorithm substitution rules including possible occurrences of the defined string variables.

I will use statements of the form " $| \, a_1,a_2,\ldots a_\ell \varepsilon A \, | \, b_1,b_2,\ldots b_m \varepsilon B \, | \, \ldots \, | \, p_1,p_2,\ldots p_n \varepsilon P \, |$ ", where the a_i, b_i, ... , and p_i are variables and the A, B, ... , and P are the names of the sets, to denote that a_1 represents members of the set named A, a_2 represents members of the set named A, etc. I will use canonical systems to define the named sets. Using this notation the above extended Markov algorithms are more precisely

*Note that the character "?" is not an English letter.

stated

```
        | s,s' ε LETTER STR |

        LETTER STR<A>,<B>, ... ,<Z>;
        LETTER STR<a>,<b>  →  LETTER STR<ab>;

    (1) sI       →  sO

    (2) XsXs'X   →  ss'

    (3) sXs      →  X

    (4) Xs.      →  X
        sX       →· X
```

Consider again the algorithm for reversing any parenthesized string of letters from the alphabet {I O X N}. Using the following variable and set definitions

```
        | c,d ε LETTER |

        LETTER<I>,<O>,<N>,<X>;
```

the extended Markov algorithm for this string transformation can now be simply given

```
          cd*  →   d*c
          (c*  →   c(
           ()  →·  Λ
            )  →   *)
```

Note that by simply augmenting the set named "LETTER" (and the object alphabet) to include all the letters of the English alphabet, the same four extended Markov algorithm substitution rules define the algorithm for reversing a string containing all English letters, whereas 704 substitution rules are required to define this transformation with a Markov algorithm.

Even with the extension to Markov algorithms given above, algorithms expressed in the extended Markov formalism often become exceedingly lengthy. One frequently occurring source of this lengthening is a requirement to construct the functional composition of two or more algorithms. Although Markov's monograph defines the additional substitution rules for taking two Markov algorithms and constructing the Markov algorithms defining their functional composition, the number of resulting substitution rules can be enormous. For example, for 2 Markov algorithms over an object alphabet consisting of all English letters, 1,457 substitution rules (Section 3.3, ref. 9) must be added to the algorithms to produce the algorithm representing their functional composition. Although by using the extension to Markov algorithms the number of additional rules could be reduced to 7, an algorithm composed by several functional compositions would quickly require many substitution rules and would be correspondingly difficult to understand.

On the other hand, Church's λ-calculus,[17,18] a formalism that makes precise the notion of a function and its properties, is ideally suited to handle the concept of functional composition. The next section presents the formalism of the λ-calculus, and the subsequent section discusses the embedding of the formalism of extended Markov algorithms within the formalism of the λ-calculus. This combined formalism will provide the heart of this dissertation's target language for defining semantics.

3.1b The λ-Calculus*

The λ-calculus is a formalism for writing certain classes of expressions. One interpretation (the interpretation taken here) of the formalism is as an explication of ideas about the specification and application of functions. Let C and V be disjoint sets of symbols, not including the symbols {λ . ( ) □}, where "□" denotes a string of one or more blank spaces. The set C will be called the set of constants. The set V will be called the set of variables. A well-formed expression in the λ-calculus is any string defined (recursively) by the following rules:

(a) If p is a variable, or p is a constant, then p is a well-formed expression.

(b) If E and F are well-formed expressions, then (E F) is a well-formed expression.

(c) If v is a variable and E is a well-formed expression, then λv.E is a well-formed expression.

For example, if C comprises the symbols {3 SQ} and V comprises the symbol {X}, some example expressions are "3", "(SQ 3)" and "λX.(SQ X)". An expression of the form (E F) is called a combination, and the expressions E and F in (E F) are called respectively the operator and operand of the combination. An expression of the form λv.E is called a λ-expression, and the

*The terminology in this chapter is due mostly to Church and Landin.

expression E in λv.E is called the body of the λ-expression. Here, a λ-expression of the form λv.E will be interpreted as a representation of the function mapping the variable v into the expression E.

An occurrence of a variable in a well-formed expression is distinguished as "free" or "bound" according to the following rules:

(a) If E is an expression consisting only of a variable, the occurrence of the variable in E is free.

(b) If E and F are expressions, an occurrence of a variable in (E F) is free or bound according as it is free or bound in E or F.

(c) If v is a variable and E is an expression, all occurrences of v in λv.E are bound while an occurrence of a variable different from v in λv.E is free or bound according as it is free or bound in E.

For example, in the expression "λX.(F X)", where "F" and "X" are variables, the occurrence of "F" is free and the occurrences of "X" are bound.

Church introduces rules for transforming expressions. Using these rules, some expressions can be transformed into a "principal normal form." The principal normal form of an expression may be viewed as a "canonical" or standard representation of the value of the expression. Because of the introduction of assignment and goto expressions into the target language to be presented later, the rules for transforming a target language expression into normal form will not always hold. Instead, the value of a target language expression will be defined in this dissertation by an extended Markov algorithm specification of a machine that mechanically converts an expression into a canonical representation of the value of the expression.

This machine will be defined formally in section 2 of this chapter. The operation of this machine for evaluating λ-calculus expressions will be presented informally in this section.

In general, the value of a constant or free variable is the object denoted by the constant or variable. A list of the values of the constants and free variables is called an "environment." The value of a λ-expression is called a "λ-closure" and consists of two parts: (a) the expression itself, and (b) the environment in which the λ-expression occurs, i.e., the list of the values of the constants and free variables in the expression.

The value of a combination is the object computed by evaluating its operand, evaluating its operator (using the values of constants and free variables given by the environment of the combination), and then applying the value of the operator to the value of the operand. If the operator of a combination is a λ-expression, the result of applying the λ-expression to its operand is computed by (a) coupling the bound variable of the λ-expression with the value of the operand to which the λ-expression is being applied (b) adding this couple to the environment of the λ-expression, and (c) evaluating the body of the λ-expression using this new environment.

Some example λ-calculus expression are the following:

3	λX.3	(λX.3 2)
(SQ 3)	λX.(SQ X)	(λX.(SQ X) 3)
X	λX.X	(λX.X 3)

If "2", "3" and "SQ" are constants denoting respectively the integer two, the integer three, and the function mapping an integer into its square, the nine expressions above denote

the integer three	the function mapping X into the integer three	the integer three
the integer nine	the function mapping X (presumably one integer) into its square	the integer nine
some object X	the identity function	the integer three

3.1c The Marriage of Extended Markov Algorithms to the λ-Calculus.

This section combines the formalism of extended Markov algorithms within the formalism of the λ-calculus. The wedding of these two formalisms will form the basis for the target language that will be presented in Section 3.1d.

Let E be a set of strings representing extended Markov algorithms, where the characters {[,],|, and "} do not occur in E. Let L be another set of strings, called the set of literals, where the character ' does not occur in L. Let C be a set of basic symbols, called the set of constants, where each constant is either a string from E enclosed by the brackets [and] or a string from L enclosed by the quotation marks ' and '. Let V be another set of basic symbols, called the set of variables, where each variable contains no occurrence of {[,], or '}. (Thus the sets C and V are disjoint.) An expression in the combined formalism will consist of any expression M such each occurrence of a variable in M is bound in M.

The extended Markov algorithms will be interpreted as definitions of primitive functions, the literals will be interpreted as representations of the objects upon which the primitive functions operate, and the variables will be interpreted as names of primitive functions, literals, or functions of the primitive functions and literals. In the examples in the text, the quotation marks will often be omitted from constants that represent integers.

Expressions in the λ-calculus are strings of basic symbols, and hence to include an extended Markov algorithm in the λ-calculus, it is necessary to have a linear representation of an extended Markov algorithm. An extended Markov algorithm of the form

$$\begin{array}{lll}
X & & \\
D & & \\
s_1 & \rightarrow(\cdot) & t_1 \\
s_2 & \rightarrow(\cdot) & t_2 \\
 & \vdots & \\
s_n & \rightarrow(\cdot) & t_n
\end{array}$$

where X is the statement listing the string variables in the algorithm, and D is the definition of the sets named in X, will therefore be represented

$$[X\ D\ s_1 \rightarrow(\cdot)\ t_1 \mid s_2 \rightarrow(\cdot)\ t_2 \mid \ldots \mid s_n \rightarrow(\cdot)\ t_n]$$

For convenience, however, the statement X and the definition D will generally be given separately from the list of substitution rules in the algorithm. For example, consider the following expression:

$$\lambda\alpha.([B\rightarrow D \mid C\rightarrow F \mid O\rightarrow I]\ \alpha)$$

This expression can be used in combination with other expressions to transform strings. For example the expression

$$(\lambda\alpha.([B\rightarrow D \mid C\rightarrow F \mid O\rightarrow I]\ \alpha)\ \text{'COBBLER'})$$

successively takes on the values

$$([B\rightarrow D \mid C\rightarrow F \mid O\rightarrow I]\ \text{'COBBLER'})$$

and finally

FIDDLER

In defining the semantics of computer languages, it will be convenient to consider the symbols $\{\rightarrow\ \cdot\ \Lambda\ [\ ]\ |\}$ as object alphabet symbols in an extended Markov algorithm. I therefore adopt the conventions that any string (not including the symbol ") enclosed by the quotation marks " and " in an extended Markov algorithm is to be considered as an object alphabet string. This use of quotation marks allows us to consider extended Markov algorithms whose object strings are *themselves* extended Markov algorithms. This point will be discussed in the definition of the primitive function "CAT", to be presented shortly.

The basic notation for the combined formalism is not especially suited to digestion by humans. To make the notation more palatable, I will introduce a series of alternate notations for writing expressions in the combined formalism. The alternate notations will be given for *convenience* and *conciseness* in communicating the expressions to humans. The alternate notations for the λ-calculus, and the λ-calculus definitions for conditional expressions and recursive functions are for the most part due to Landin.

Alternate Notations for Extended Markov Algorithms:

The linear representation of an extended Markov algorithm is difficult to visualize. Accordingly, I will generally use the notation

$$\begin{bmatrix} s_1 & \rightarrow(\cdot) & t_1 \\ s_2 & \rightarrow(\cdot) & t_2 \\ & \vdots & \\ s_n & \rightarrow(\cdot) & t_n \end{bmatrix}$$

(where the variable and set definitions for the algorithm will be given separately) in place of the strict linear representation of an extended Markov algorithm in the λ-calculus. For example, the expression

$$\lambda\alpha.([B\to D|C\to F|O\to I]\ \alpha)$$

will be written

$$\lambda\alpha.(\begin{bmatrix} B & \to & D \\ C & \to & F \\ O & \to & I \end{bmatrix}\ \alpha)$$

The Function CAT:

Let s be a string variable representing any string of characters and consider the following expression

$$\lambda\alpha.([s.\ \to\cdot\ \text{"}[\Lambda\to\cdot\text{"}\ s\ \text{"}]\text{"}]\ \alpha)$$

This expression defines a function mapping the value of the variable α into the extended Markov algorithm $[\Lambda \to\cdot\ \alpha]$, where "$\alpha$" here denotes the value of the variable α. This extended Markov algorithm when applied to an object string concatenates the string value of α to the object string. The function above will be called "CAT". For example, the expression ((CAT 'HELLO') ' THERE') successively takes on the values:

```
((λα.([s. →· "[Λ→·" s" ]" ] α) 'HELLO ') 'THERE')

(([s. →· "[Λ→·" s "]" ] 'HELLO ') 'THERE')

([Λ →· HELLO ] 'THERE')

HELLO THERE
```

Similarly, the expression ((CAT ((CAT 'HOW ') 'ARE ')) 'YOU') takes on the value "HOW ARE YOU". Note that the extended Markov algorithm [s. →· "[Λ→·" s "]"] maps its object string into another extended Markov algorithm, and thus extended Markov algorithms have the ability to define functionals, i.e., functions mapping an argument into a new function.

In defining the semantics of a computer language, it will frequently be necessary to concatenate strings to produce a string that represents an extended Markov algorithm or a string to which an extended Markov algorithm is applied. It will be convenient not to state explicitly the concatenation of strings in these cases, and I therefore introduce the following alternate solution.

> Let "CAT" be the function as defined above, let X_i $1 \le i \le n$ be expressions, and let ((CAT...((CAT((CAT X_1) X_2)) X_3)) ... X_n) be an expression whose value is an extended Markov algorithm or a string to which an extended Markov algorithm is applied. The X_i can be written directly in the form of the extended Markov algorithm or the concatenated string to which an extended Markov algorithm is applied.

Thus, for example, the expressions

$$\lambda\pi.\lambda\alpha.\lambda\beta.(((\text{CAT}((\text{CAT}((\text{CAT}((\text{CAT}\ \text{'[TRUE} \rightarrow\cdot\text{'})\ \alpha))\ \text{' FALSE} \rightarrow\cdot\text{'}\ ))\ \beta))\ \text{']'})\ \pi)$$

$$\begin{array}{l}\lambda\alpha.\lambda\beta.([\text{TRUE/TRUE} \rightarrow\cdot \text{TRUE} \mid \text{TRUE/FALSE} \rightarrow\cdot \text{FALSE} \mid \\ \qquad\quad \text{FALSE/TRUE} \rightarrow\cdot \text{FALSE} \mid \text{FALSE/FALSE} \rightarrow\cdot \text{FALSE}] \\ \qquad\quad ((\text{CAT}\ ((\text{CAT}\ \alpha)\ \text{'/'}))\ \beta))\end{array}$$

can be written

$$\lambda\pi.\lambda\alpha.\lambda\beta.([\text{TRUE} \rightarrow\cdot \alpha \mid \text{FALSE} \rightarrow\cdot \beta]\ \pi)$$

$$\begin{array}{l}\lambda\alpha.\lambda\beta.([\text{TRUE/TRUE} \rightarrow\cdot \text{TRUE} \mid \text{TRUE/FALSE} \rightarrow\cdot \text{FALSE} \mid \\ \qquad\quad \text{FALSE/TRUE} \rightarrow\cdot \text{FALSE} \mid \text{FALSE/FALSE} \rightarrow\cdot \text{FALSE}]\ \alpha/\beta)\end{array}$$

or further rewritten using the previously given alternate notation

$$\lambda\pi.\lambda\alpha.\lambda\beta.(\begin{bmatrix}\text{TRUE} & \rightarrow\cdot & \alpha \\ \text{FALSE} & \rightarrow\cdot & \beta\end{bmatrix}\pi)$$

$$\lambda\alpha.\lambda\beta.(\begin{bmatrix}\text{TRUE/TRUE} & \rightarrow\cdot & \text{TRUE} \\ \text{TRUE/FALSE} & \rightarrow\cdot & \text{FALSE} \\ \text{FALSE/TRUE} & \rightarrow\cdot & \text{FALSE} \\ \text{FALSE/FALSE} & \rightarrow\cdot & \text{FALSE}\end{bmatrix}\ \alpha/\beta)$$

The first expression defines a function* that when successively

*Greek letters will generally not occur as object strings for extended Markov algorithms. I will therefore use Greek letters in an extended Markov algorithm or the string to which it is applied to denote the symbols that are bound variables. Thus, in writing the strict representation of the algorithm or its object string in terms of λ-calculus expressions, strings not containing Greek letters are to be quoted and the Greek letters are not to be quoted.

applied to three arguments produces the value of the variable α if the value of the variable π is "TRUE" and produces the value of the variable β if the value of the variable π is "FALSE". The second expression defines a boolean-valued function that when successively applied to two boolean valued arguments produces the value "TRUE" if both arguments have the value "TRUE" and produces the value "FALSE" if either argument has the value "FALSE". The first expression will later be used to define conditional expressions. The second expression will later be used to define the function for producing the logical "and" of two arguments.

Note that the first expression above constructs an extended Markov algorithm from literal strings and _bound_ variables. The notion of a bound variable lends itself immediately to extended Markov algorithms embedded within the λ-calculus and allows the construction of extended Markov algorithms that depend on the _values_ of the variables to which the algorithms are applied. This compatibility between the married formalisms _greatly_ simplified the definitions of the primitive functions for SNOBOL/1 and ALGOL/60.

Alternate notations for the λ-calculus:

The basic notation for defining and applying functions in the λ-calculus is somewhat awkward for those accustomed to writing functions in the conventional mathematical notation. I thus introduce the following alternate notations.

Let F, V_1, V_2, ... , V_n be variables and M, Q, E_1, E_2, ... , E_n be expressions. Expressions of the form

(a) $(\lambda V_1.(\lambda V_2 \ldots (\lambda V_n.M\ E_n)\ \ldots\ E_2)\ E_1)$

(b) $(\lambda F.M\ \lambda V_1.\lambda V_2 \ldots \lambda V_n.Q)$

(c) $(\ldots((F\ E_1)\ E_2)\ \ldots\ E_n)$

can be written

(a) LET $V_1, V_2,\ \ldots\ , V_n = E_1, E_2,\ \ldots\ , E_n$
IN M

(b) LET $F(V_1, V_2,\ \ldots\ , V_n) = Q$
IN M

(c) $F(E_1, E_2,\ \ldots\ , E_n)$

where if $M, Q, E_1, E_2,\ \ldots\ ,$ or E_n are enclosed in parentheses, the parentheses can be dropped. Thus, for example, the expressions

$$(\lambda X.(\text{'SQ'}\ X)\ 3)$$

$$((\lambda X.\lambda Y.((\text{'CAT'}\ X)\ Y)\ \text{'HELLO '})\ \text{'THERE'})$$

$$(\lambda \text{COND}.(((\text{COND}\ \text{'TRUE'})\ 0)\ 1)\ \ \lambda\pi.\lambda\alpha.\lambda\beta.(\begin{bmatrix}\text{TRUE} \rightarrow\cdot\ \alpha \\ \text{FALSE} \rightarrow\cdot\ \beta\end{bmatrix}\ \pi))$$

can be written

```
LET X = 3
IN  'SQ' X

LET X,Y='HELLO ', 'THERE'
IN  (('CAT' X) Y)
```

$$\begin{aligned}&\text{LET } \text{COND}(\pi,\alpha,\beta) = \left(\begin{bmatrix}\text{TRUE} & \to\cdot & \alpha\\ \text{FALSE} & \to\cdot & \beta\end{bmatrix}\ \pi\right)\\ &\text{IN } \ \text{COND}(\text{'TRUE'},0,1)\end{aligned}$$

Conditional Expressions:

Consider the function COND defined previously

$$\text{COND}(\pi,\alpha,\beta) = \left(\begin{bmatrix}\text{TRUE} & \to\cdot & \alpha\\ \text{FALSE} & \to\cdot & \beta\end{bmatrix}\ \pi\right)$$

This function selects the value of α if the value of π is "TRUE" and the value of β if the value of π is "FALSE". For example, the value of COND('TRUE',0,1) is the string "0". Next consider the following expression from ALGOL/60

```
IF A=0 THEN B*A ELSE B/A
```

and the (loosely written) expression in the combined formalism

```
COND(A=0,B*A,B/A)
```

where COND is defined as above. This expression does not correctly mirror the ALGOL/60 expression. In ALGOL/60 the expression B*A is evaluated *only* if the value of A is equal to zero, and the expression B/A is evaluated *only* if the value of A is not equal to zero. This order of evaluation insures that B/A is not evaluated if the value of A is zero. Now consider the following (loosely written) target language expression

$$(\mathrm{COND}(A=0,\lambda\pi.B*A,\lambda\pi.B/A)\ '\Lambda')$$

where π is a dummy variable. In evaluating this expression, the function COND will be applied to its arguments, one of the λ-expressions $\lambda\pi.B*A$ or $\lambda\pi.B/A$, will be selected and then the selected λ-expression will be applied to the operand $'\Lambda'$. Thus *only* the body of the *selected* λ-expression will be evaluated.* The use of the dummy variable serves as a delaying mechanism in evaluating expressions.

Conditional expressions of the above form will be used repeatedly in defining the semantics of computer languages. I therefore introduce the following alternate notation.

Let s_1, s_2, t_1, t_2, and t_3 be expressions. Expressions of the form

$$(\mathrm{COND}(s_1,\lambda\pi.t_1,\lambda\pi.t_2)\ '\Lambda')$$

and

$$(\mathrm{COND}(s_1,\lambda\pi.t_1,\lambda\pi.(\mathrm{COND}(s_2,\lambda\pi.t_2,\lambda\pi.t_3)\ '\Lambda'))\ '\Lambda')$$

can be written

$$\begin{array}{lcl} s_1 & \Rightarrow & t_1 \\ \mathrm{ELSE} & \Rightarrow & t_2 \end{array}$$

*Note, in forming a λ-closure, the body of the λ-expression is *not* evaluated.

and

$$\begin{array}{lll} s_1 & \Rightarrow & t_1 \\ s_2 & \Rightarrow & t_2 \\ \text{ELSE} & \Rightarrow & t_3 \end{array}$$

Similarly, this alternate notation can be extended to include an arbitrary number of nested conditional expressions.

For example, the expression

$$(\text{COND}(A=0,\lambda\pi.B*A,\lambda\pi.B/A)\ \text{'A'})$$

can be written

$$\begin{array}{lll} A=0 & \Rightarrow & B*A \\ \text{ELSE} & \Rightarrow & B/A \end{array}$$

3.1d The Target Language

The combined formalism of extended Markov algorithms and the λ-calculus presented in the previous section appears sufficient to define fairly concisely many constructions in computer languages. However, two common features of many computer languages, that for assigning new values to variables and that for transferring control to another statement in a program, have evaded characterization in the combined formalism. To handle this circumstance, the combined formalism will be augmented with new expressions to mirror directly the assignment of new values to variables and the transfer of evaluation from one expression to another. The augmented version of the combined formalism will comprise the target language of this dissertation.

Sequences of Expressions:

Before discussing the rules for forming well-formed expressions in the target language, let us consider a mechanism for defining a sequence of expressions, where each expression $E_1, E_2, \ldots, E_n$ in the sequence is to be evaluated in the numerical order indicated by its numerical subscript. Using the rule for evaluating the operand of a combination <u>before</u> the operator of a combination, the target language provides a device for handling a sequence of expressions.

Let $X, E, E_1, E_2, \ldots$, and E_n be expressions, and consider the following λ-expression, called T

$$\lambda\alpha.\lambda\beta.(\beta\ \alpha)$$

When evaluated, the combination (T E) results in first evaluating the expression E and then returning the value of the λ-closure for $\lambda\beta.(\beta\ \alpha)$, where α is coupled with the value of E. Next consider the combination

$$[(T\ E)\ \lambda\pi.X]$$

where square brackets have been used here (for convenience) in place of parentheses.[*] This combination is evaluated as follows:

1. The λ-closure for $\lambda\pi.X$ is computed

[*] Square brackets will be used frequently in this section. Strictly speaking, all square brackets should be replaced by parentheses.

2. The combination (T E) is computed, resulting in first evaluating E and then returning the λ-closure for $\lambda\beta.(\beta\ \alpha)$, where α is coupled with the value of E.

3. The value of the expression in 2 is applied to the value of the expression in 1, resulting in applying $\lambda\pi.X$ to E, which returns the value of X.

In particular, if X is the expression "π", this combination results in returning the value of E.

Next consider the expression

$$[(T\ E_1)\ \lambda\pi.[(T\ E_2)\ \lambda\pi.\pi]]$$

This combination is evaluated as follows:

1. The λ-closure for $\lambda\pi.[(T\ E_2)\ \lambda\pi.\pi]$ is computed. Note that the value of E_2 is not computed in forming the λ-closure.

2. The combination $(T\ E_1)$ is computed, resulting in first evaluating E_1 and then returning the λ-closure for $\lambda\beta.(\beta\ \alpha)$

3. The value of the expression in 2 is applied to the value of the expression in 1, resulting in returning the value of $[(T\ E_2)\ \lambda\pi.\pi]$. This evaluation results in first computing the value of E_2 and then returning the value of E_2.

Thus the evaluation of this expression results in first evaluating E_1, then evaluating E_2, and finally returning the value of E_2.

Similarly, consider the expression

$$\underset{\overset{\uparrow}{1}}{[}(T\ E_1)\ \lambda\pi.\underset{\overset{\uparrow}{2}}{[}(T\ E_2)\ \lambda\pi.\underset{\overset{\uparrow}{3}}{[}(T\ E_3)\ \lambda\pi.\pi]]]$$

When evaluated, this expression results in successively evaluating E_1, E_2, and E_3 and then returning the value of E_3. This expression, however, has the following important property, which will be used in the definition of the transfer of control to some labeled expression in a sequence of expressions. Let C_1, C_2, and C_3 be the combinations that are given by the matching paris of square brackets indicated by the numbers 1, 2, and 3 above. The evaluation of C_1 results in successively evaluating E_1, E_2, and E_3 and returning the value of E_3; the evaluation of C_2 results in successively evaluating E_2 and E_3 and returning the value of E_3; the evaluation of C_3 results in evaluating E_3 and returning the value of E_3.

More generally, an expression of the form

$$\underset{\substack{\uparrow\\1}}{[}(T\ E_1)\ \lambda\pi.\underset{\substack{\uparrow\\2}}{[}(T\ E_2)\ \ldots\ \lambda\pi.\underset{\substack{\uparrow\\n}}{[}(T\ E_n)\ \lambda\pi.\pi]\ldots]]$$

when evaluated, results in successively evaluating E_1, E_2, ... , and E_n and returning the value of E_n. Moreover, the evaluation of any combination C_i beginning with the square bracket denoted by the integer i results in successively evaluating the expressions E_i, E_{i+1}, ... , and E_n and returning the value of E_n. This later effect leads us to the notion of a "labeled" expression.

Labels and Label References:

Let V be the set of variables (as described earlier) and let L be the set obtained from V by affixing a ":" to each variable in V. The set L will be called the set of labels. Consider an expression of the form

$$\ell_1[(T\ E_1)\ \lambda\pi.\ell_2[(T\ E_2)\ \ldots\ \lambda\pi.\ell_n[(T\ E_n)\ \lambda\pi.\pi]\ldots]]$$

where the ℓ_i, $1\leq i\leq n$ indicates the possible occurrences of labels, each of which must be different. An expression of this form will be called a "sequence" of the expressions E_1, E_2, ..., and E_n. If we ignore the labels in an evaluation, the evaluation of any combination C_i following some label ℓ_i, $1\leq i\leq n$, results in successively evaluating E_i, E_{i+1}, ... , and E_n and returning the value of E_n.

A sequence of the above form may occur within the body of some λ-expression, which in turn may occur within a sequence in the body of some encompassing λ-expression, and so on for further encompassing λ-expressions. In the target language the transfer of control to some labeled expression will be designated by expressions of the form (GOTO. E), where E is an expression referring to some label. A _label reference_ will be a string of the form .ℓ , where ℓ: is a label. The _value_ of a label reference .ℓ will consist of two parts: (a) the combination in the _innermost_ encompassing λ-expression such that the combination is prefixed by the label ℓ: , and (b) the environment within which the combination is to be evaluated. The evaluation of a label reference will be called a "label-closure".

I now proceed to a presentation of the target language of the dissertation.

Target Language Expressions:

An expression in the target language is defined as follows. Let C, V, and L be sets of symbols, called the sets of constants, variables, and labels, as described earlier.

(a) If p is a variable or p is a constant, then p is an expression.

(b) If E and F are expressions, then (E F) is an expression.

(c) If v is a variable and E is an expression, then λv.E is an expression.

(d) If v is a variable and E is an expression, then (v ASSIGN. E) is an expression.

(e) If S is a sequence, then S is an expression.

(f) If E is an expression, then (GOTO. E) is an expression.

Expressions of type (a), (b), and (c) are expressions in the combined formalism as introduced previously. Expressions of type (d), (e), and (f) are new. The evaluation of an expression of the form (v ASSIGN. E) will result in first changing the value of the variable v to the value of the expression E and then returning the null string as the value of the expression (v ASSIGN. E). If the labels in an expression of type (e) are ignored, the evaluation of a sequence results in successively evaluating each of the component expressions E_1, E_2, and E_n in the sequence and returning the value E_n. If E is an expression of the form .ℓ , where ℓ: is a label, the evaluation of E will result in forming the label-closure for .ℓ and the evaluation of an expression of the form (GOTO. E) within some sequence will result in (a) stopping the evaluation of the expression in which E occurs and (b) continuing by evaluating the combination designated by the label-closure for $.\ell$ within the environment specified by the label-closure. Note that this mechanism allows transfer of control only to expressions within the same sequence or expressions in a sequence in some encompassing λ-expression. The previously given notation for defining a sequence of expressions is awkward. I thus introduce the following alternate notation in place of the strict representation of a sequence. Let E be a sequence of the form

$$\ell_1[(T\ E_1)\ \lambda\pi.\ell_2[(T\ E_2)\ \ldots\ \lambda\pi.\ell_n[(T\ E_n)\ \lambda\pi.\pi]\ldots]]$$

where the ℓ_i, $1\leq i\leq n$, indicate the possible occurrences of labels. A sequence of this form will be alternately written

$$\ell_1E_1;\ \ell_2E_2;\ \ldots\ \ \ell_nE_n$$

The addition of expressions of type (d), (e), and (f) take effect when it is desired to construct a sequence of expressions to be evaluated one after another or to interrupt the evaluation of a sequence and to continue the evaluation at some other labeled expression.

For example, consider the expression

```
LET  A= 5
IN   (A ASSIGN. (+(A,1)));
     (GOTO. .P);
     (A ASSIGN. 1);
   P:A
```

where "+" is a free variable whose value is the function for computing the arithmetic sum of two integers. The evaluation of this expression is as follows:

(1) The value of the bound variable A will be set to five and the body of the λ-expression evaluated.

(2) Since the body of the λ-expression is a sequence of expressions, each of the component expressions will be evaluated in order.

(3) The first expression in the sequence results in updating the value of A to six.

(4) The second expression results in transferring the evaluation to the expression labeled P.

(5) The evaluation of the expression labeled P results in returning the value of A, which has been set to six.

Recursive Definitions:

Consider the following (loosely written) expression defining the factorial function and its application to the integer five:

```
LET FACT(N) = EQ(N,0) ⇒ 0
              ELSE    ⇒ N*FACT(N-1)
IN  FACT(5)
```

where EQ is a boolean valued function for testing the equality of two integers. The function "FACT" when applied to the argument "5" will not evaluate to five factorial. The difficulty here arises in the definition of the function "FACT" where the variable "FACT" itself occurs as a free variable. This incorrect rendering of a recursive function can be corrected through the notion of a "fixed-point operator."[20,25] One fixed-point operator for target language expressions is the expression

```
Y  =  λF. LET π='Λ'
          IN  (π ASSIGN. (F π)); π
```

If M is an expression and F=E is a recursive definition of the function F, an expression of the form

```
LET F = E
IN  M
```

where E contains free occurrences of the variable F, can be correctly written

```
LET F = (Y λF.E)
IN  M
```

To avoid this somewhat awkward method for writing recursive functions, the following alternate notation is introduced.

> If F is a variable and E and M are expressions, an expression of the form
>
> ```
> LET F = (Y λF.E)    IN M
> ```
>
> where Y is the fixed-point operator given above, can alternately be written
>
> ```
> LET REC F=E    IN M
> ```

Thus the definition of the factorial function can be correctly written

```
LET REC FACT(N) = EQ(N,0) ⟹ 0
                  ELSE    ⟹ N*FACT(N-1)
IN  FACT(5)
```

The above fixed-point operator is sufficient to handle recursive definitions of single functions but not simultaneous recursive definition of two or more functions. In this dissertation simultaneous recursive definitions will not be needed until the semantics of ALGOL/60 procedure declarations is defined, and the presentation of a fixed-point operator to handle simultaneous recursive definitions will be deferred until the chapter on ALGOL/60. A detailed discussion of fixed-point operators is given by Wozencraft.[25]

A Definition of the Semantics of the ALGOL/60 Subset:

The definition of the semantics of the ALGOL/60 subset in terms of the target language is given in Appendices 2.1 and 2.2. The specification of the corresponding target language expression for a program in the subset has been broken into two parts. Appendix 2.1 defines the translation of a program into the target language assuming that the primitive "+" is a free variable. Appendix 2.2 defines the primitive "+". To form the complete target language expression, one must take the target language string specified in Appendix 2.1 and add to it the primitive function definitions of Appendix 2.2 in the form

```
LET    CAT α=[s. →• "[Λ→•" s "]" ]α
IN LET EQ(α,β) = ...                                          (a)
 •
 •
 •
IN LET REC +(X,Y) = EQ(Y,0) ⇒ 0 ELSE ⇒ SUM(SUCC X,PRED X)
IN LET d' IN s'
```

where "LET d' IN s'" is the target language string specified by Appendix 2.1.* For example, Appendix 2.1 specifies the following pair of strings

```
BEGIN  INTEGER A; A:=1+2  END .. LET A = 'Λ'
                                 IN  (A ASSIGN. (+('1','2'))
```

The string "LET A = 'Λ' IN (A ASSIGN. (+('1','2'))" when used in place of "LET d' IN s'" in expression (a) above specifies the complete target language expression for the program "BEGIN INTEGER A; A:=1+2 END".**

3.2 An Evaluator for the Target Language

To explain the semantics of the target language in the previous sections, an appeal was made through the English language. This section reduces that appeal to an appeal for understanding only the formalism of extended Markov algorithms.

*This division of the specification of the semantics of a computer language into a specification of a target language string and a separate specification of the primitive functions used in the target language string will be followed in the definitions of SNOBOL/1 and ALGOL/60. Also, the definitions of the string variables for the extended Markov algorithm primitives are given at the beginning of Appendix 2.2. These definitions must be added to each extended Markov algorithm using the string variables.

**It may happen that the use of identifiers in a source language program will conflict with the use of identifiers used to define the primitive functions in the target language. To avoid this conflict, the identifiers for the target language primitives strictly speaking should be given as identifiers that are different from the source language identifiers. This conflict can be avoided by appending to each target language identifier a symbol (e.g., the symbol "#") not allowed in source language identifiers.

The "value" of a target language expression will be defined in this section by an extended Markov algorithm definition of a machine that mechanically converts an expression into another expression, the value of the initial expression. The machine may be viewed as a hypothetical computer for the target language, and extended Markov algorithms may be viewed as the machine language for the computer. The definition of the target language evaluator is based on a similar definition given by Landin,[20,24] and Wozencraft.[25]

The extended Markov algorithm definition of the target language evaluator is given in Appendix 2.3. Before applying the algorithm to a target language expression, it is necessary to provide a unique index for each "λ" and "(" in the expression. Thus the expression

$$(\lambda X.('SQ'\ X)\ '3')$$

will be indexed

$$({}_1\lambda_2 X.({}_3'SQ'\ X)\ '3')$$

The indices allow unique identification of a λ-expression or combination.

The evaluation of an expression begins with a substitution rule transforming the expression to be evaluated into five strings: the "control" string, the "result" string, the "environment" string, the "store" string, and the expression itself. Subsequent substitution rules define transformations on the control, result, environment, and store strings until the value of the target language expression is computed. The final substitution rule returns the value of the expression.

Generally, the control string is a string of the form

$$a_k\ a_{k-1}\ \cdots\ a_1$$

where each a_i, $1 \leq i \leq k$, is an atomic part of an expression (e.g., a constant, variable, indexed lambda symbol, or indexed left parenthesis). The control string is used to hold the atomic parts of an expression <u>before</u> they are evaluated.

When the parts of the control string are evaluated, their values are placed on the store string. The store string is a string of the form

$$(111\ldots1,\ r_n)\ \ldots\ (111,r_3)(11,r_2)(1,r_1)$$

where each r_i, $1 \leq i \leq n$, is a string denoting the value of a constant, a variable, or a λ-expression, and the string of ones before each string value provides a unique pointer to the string value. A new store component for a string r_{n+1} is obtained by (a) obtaining the string of ones representing the pointer p to r_n and (b) prefixing the string "$(1p,r_{n+1})$" to the left of the store string.

The result string is used to store pointers to intermediate calculated values formed in the evaluation of a target language expression. The result string is a string of the form

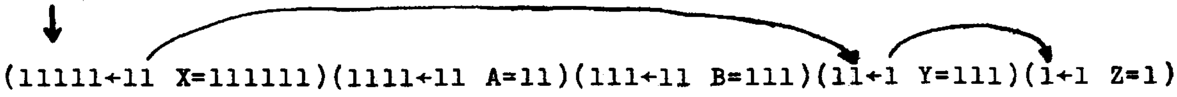
$$p_m \ \cdots \ p_2 \, p_1$$

where each p_i, $i \leq l \leq m$, is a pointer to some string value in the store.

Let $N_1, M_1, N_2, M_2, \ldots, N_k, M_k$ denote strings of ones, let $v_1, v_2, \ldots, v_k$ denote variables, and let $p_1, p_2, \ldots, p_k$ denote pointers to the store. The environment string is a string of the form

$$(N_k{\leftarrow}M_k \ v_k{=}p_k) \ \ldots \ (N_2{\leftarrow}M_2 \ v_2{=}p_2)(N_1{\leftarrow}M_1 \ v_1{=}p_1)$$

where each component $(N_i{\leftarrow}M_i \ v_i{=}p_i)$ is a string such that N_i, $1 \leq i \leq k$, identifies the environment for some λ-expression λ_j, v_i identifies the bound variable v of λ_j, p_j is a store pointer to the current value v, and M_i identifies the environment of the encompassing λ-expression. The environment M_i is said to be "linked" to the environment N_i. In general, the environment components linked to N_i provide pointers to the current values each of the bound variables in the λ-expression λ_j and its encompassing λ-expressions. The list of environment components linked to N_i will be called the environment N_i. For example, consider the environment "11111" in the environment



(11111←11 X=111111)(1111←11 A=11)(111←11 B=111)(11←1 Y=111)(1←1 Z=1)

The environment components linked to "11111" provide store pointers to the current values of the variables X,Y, and Z in the λ-expression whose environment is identified by "11111".

A new component is prefixed to the environment string each time a new λ-expression is applied. Thus each N_i at the left of each environment component identifies an environment for some applied λ-expression, and the environment components linked to N_i provide pointers to the values of the free variables in the body of the λ-expression whose environment is given by N_i. Since constants in the target language are treated as literal strings whose values are the strings themselves, the values of the constants in an expression are not placed on the environment string.

The set definitions for the string variables used in the extended Markov algorithm definition of the evaluator are given in Appendix 2.3a. The set "STR" defines the set of all strings that might occur within a target language expression. The sets "CONSTANT" and "VARIABLE" define the sets of constants and variables. The sets "PTR" and "INDEX" define respectively the set of pointers to the store string and the set of indices used in marking an expression. The set "EXP" defines the set of target language expressions, the set "EXP HD" defines the set of strings that can occur at the head of an expression, and the set "EXP TL" defines the set of strings that can occur at the tail of an expression. For example, in the expression "$(_1\lambda_2$X.$(_3$'SQ' X) '3')" the string "$(_1$" is the head of the expression and the string "λ_2X.$(_3$'SQ' X) '3')" is the tail of the expression, and in the expression "X" the variable "X" is the head of the expression and the tail of the expression is null.

The substitution rules for the extended Markov algorithm definition of the target language evaluator are given in Appendix 2.3b. Three alternate notations were used in writing these rules:

(1) Let x_i and y_i, $1\leq i\leq 5$, be string variables representing arbitrary strings used in an extended Markov algorithm. Generally, each substitution rule is of the form*

$$<cy_1-x_2ry_2-x_3ey_3-x_4sy_4-x_5py_5> \rightarrow <c'y_1-x_2r'y_2-x_3e'y_3-x_4s'y_4-x_5p'y_5>$$

where the c, r, e, s, and p are string referring to portions of the control, result, environment, store, and expression strings and the c', r', e', s', and p' are the transformed portions of these strings. Since the x_i and y_i occur in each substitution rule, a substitution rule of the above form will be written in the form

$$\left.\begin{matrix}c\\r\\e\\s\\p\end{matrix}\right] \rightarrow \left[\begin{matrix}c'\\r'\\e'\\s'\\p'\end{matrix}\right.$$

(2) If one of the five strings c, r, e, s, or p is given as null on both sides of the substitution rule, the symbol "_" can be used in place of the null string symbol "Λ".

(3) If one of the five components c, r, e, s, or p occurs unchanged in the right-hand side of the substitution rule, the symbol "I" can be used in place of the string in the right-hand side of the rule.

*The hyphen "-" is used to separate the control, result, environment, store, and expression strings.

Thus the substitution rule

$$<(_i y_1 - x_2 \Lambda y_2 - x_3 \Lambda y_3 - x_4 \Lambda y_4 - x_5 (_i ht\ h't')y_5>$$
$$\rightarrow <h'\ h\ APPLY.\ y_1 - x_2 \Lambda y_2 - x_3 \Lambda y_3 - x_4 \Lambda y_4 - x_5 (_i ht\ h't')y_5>$$

can be written using notation (1)

$$\left.\begin{array}{r} (_i \\ \Lambda \\ \Lambda \\ \Lambda \\ (_i ht\ h't') \end{array}\right] \rightarrow \left[\begin{array}{l} h'\ h\ APPLY. \\ \Lambda \\ \Lambda \\ \Lambda \\ (_i ht\ h't') \end{array}\right.$$

and further written using notations (2) and (3)

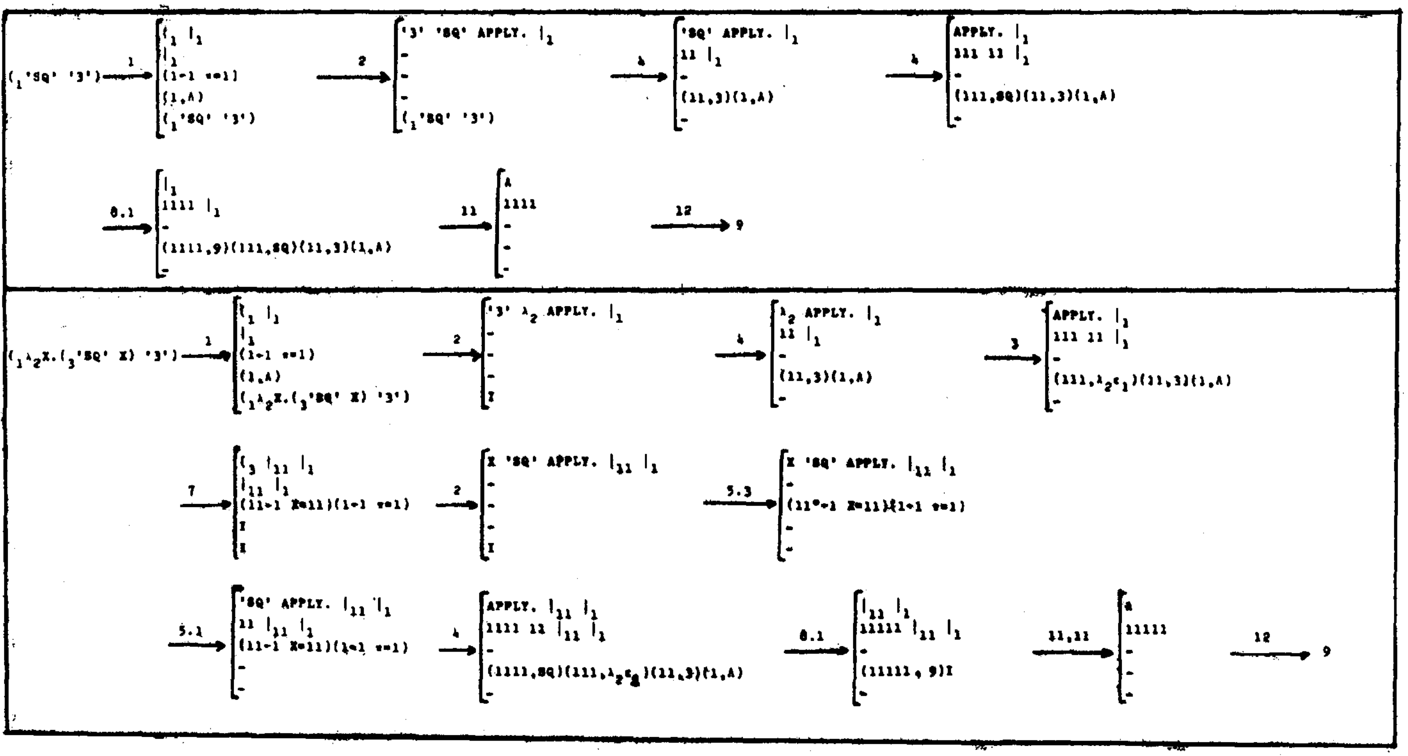
$$\left.\begin{array}{r} (_i \\ - \\ - \\ - \\ (_i ht\ h't') \end{array}\right] \rightarrow \left[\begin{array}{l} h'\ h\ APPLY. \\ - \\ - \\ - \\ I \end{array}\right.$$

Three example evaluations of target language expressions are given on the adjacent pages. Each of these evaluations shows the successive transformations on one of the initial expressions:*

```
('SQ' '3')  LET X='3'      LET X='3'
            IN  ('SQ' X)   IN  (X ASSIGN. '4');
                               (GOTO. .L);
                               (X ASSIGN. '5');
                            L: X
```

*The constant 'SQ' in the first two expressions represents the primitive function for squaring an integer. Strictly speaking, all primitive functions in the target language must be defined by constants that are extended Markov algorithms.

THREE EXAMPLE EXPRESSIONS AND THEIR EVALUATIONS



Note: In this example, decimal digits will be used in place of the corresponding strings of ones denoting store pointers and environment names.

$$({}_1\lambda_2 X.[{}_3({}_4 T{\downarrow}\ ({}_5 X\ \text{ASSIGN.}\ \text{'4'}))\quad \lambda_6\pi.[{}_7({}_8 T{\downarrow}\ ({}_9\text{GOTO.}\ \text{.L}))\quad \lambda_{10}\pi.[{}_{11}({}_{12}T{\downarrow}\ ({}_{13}X\ \text{ASSIGN.}\ \text{'5'}))\quad \lambda_{14}.L{:}[{}_{15}({}_{16}T{\downarrow}\ X)\ \lambda_{17}\pi.\pi]]]\quad \text{'3'})$$

$$\lambda_{18}\alpha.\lambda_{19}\beta.({}_{20}\beta\ \alpha)\qquad \lambda_{21}\alpha.\lambda_{22}\beta.({}_{23}\beta\ \alpha)\qquad \lambda_{24}\alpha.\lambda_{25}\beta.({}_{26}\beta\ \alpha)\qquad \lambda_{27}\alpha.\lambda_{28}\beta.({}_{29}\beta\ \alpha)$$

$$\xrightarrow{1}\left[\begin{array}{l}({}_1\ |_1\\ |_1\\ (1\leftarrow 1\ \pi=1)\\ (1,\Lambda)\\ \text{the above expression}\end{array}\right.\quad\xrightarrow{2,4,3,7}\left[\begin{array}{l}({}_3\ |_2\ |_1\\ |_2\ |_1\\ (2\leftarrow 1\ X=2)I\\ (3,\lambda_2\epsilon_1)(2,3)(1,\Lambda)\\ I\end{array}\right.\quad\xrightarrow{2,3,2}\left[\begin{array}{l}({}_5\ \lambda_{18}\ \text{APPLY. APPLY.}\ |_2\ |_1\\ 4\ |_2\ |_1\\ I\\ (4,\lambda_6\epsilon_2)I\\ I\end{array}\right.\quad\xrightarrow{2,5}\left[\begin{array}{l}X\ \text{ASSIGN. APPLY.}\ \lambda_{18}\ \text{APPLY. APPLY.}\ |_2\ |_1\\ 5\ 4\ |_2\ |_1\\ I\\ (5,4)I\\ I\end{array}\right.$$

$$\xrightarrow{5.3,5.1}\left[\begin{array}{l}\text{ASSIGN. APPLY}\ \lambda_{18}\ \text{APPLY. APPLY.}\ |_2\ |_1\\ 2\ 5\ 4\ |_2\ |_1\\ I\\ I\\ I\end{array}\right.\quad\xrightarrow{10.2}\left[\begin{array}{l}\lambda_{18}\ \text{APPLY. APPLY.}\ |_2\ |_1\\ 1\ 4\ |_2\ |_1\\ I\\ (5,4)(4,\lambda_6\epsilon_2)(3,\lambda_2\epsilon_1)(2,\overset{\checkmark}{4})(1,\Lambda)\\ I\end{array}\right.\quad\xrightarrow{3.7}\left[\begin{array}{l}\lambda_{19}\ |_3\ \text{APPLY.}\ |_2\ |_1\\ |_3\ 4\ |_2\ |_1\\ (3\leftarrow 2\ \alpha=1)(2\leftarrow 1\ X=2)(1\leftarrow 1\ \pi=1)\\ (6,\lambda_{18}\epsilon_2)I\\ -\end{array}\right.$$

$$\xrightarrow{3}\left[\begin{array}{l}|_3\ \text{APPLY.}\ |_2\ |_1\\ 7\ |_3\ 4\ |_2\ |_1\\ I\\ (7,\lambda_{19}\epsilon_3)I\\ I\end{array}\right.\quad\xrightarrow{11,7}\left[\begin{array}{l}({}_{20}\ |_4\ |_2\ |_1\\ |_4\ |_2\ |_1\\ (4\leftarrow 3\ \beta=4)(3\leftarrow 2\ \alpha=1)(2\leftarrow 1\ X=2)(1\leftarrow 1\ \pi=1)\\ I\\ I\end{array}\right.\quad\xrightarrow{2,5.3,5.2,5.1,5.3,5.1}\left[\begin{array}{l}\text{APPLY.}\ |_4\ |_2\ |_1\\ 4\ 1\ |_4\ |_2\ |_1\\ I\\ I\\ I\end{array}\right.$$

$$\xrightarrow{7}\left[\begin{array}{l}({}_7\ |_5\ |_4\ |_2\ |_1\\ |_5\ |_4\ |_2\ |_1\\ (5\leftarrow 2\ \pi=1)I\\ I\\ I\end{array}\right.\quad\xrightarrow{2,3,2,2}\left[\begin{array}{l}.L\ \text{GOTO. APPLY.}\ \lambda_{21}\ \text{APPLY.}\ |_5\ |_4\ |_2\ |_1\\ 8\ |_5\ |_4\ |_2\ |_1\\ (5\leftarrow 2\ \pi=1)(4\leftarrow 3\ \beta=4)(3\leftarrow 2\ \alpha=1)(2\leftarrow 1\ X=2)(1\leftarrow 1\ \pi=1)\\ (8,\lambda_{10}\epsilon_5)(7,\lambda_{19}\epsilon_3)(6,\lambda_{18}\epsilon_2)(5,4)(4,\lambda_6\epsilon_2)(3,\lambda_2\epsilon_1)(2,4)(1,\Lambda)\\ I\end{array}\right.\quad\xrightarrow{6.3,6.2,6.1}\left[\begin{array}{l}\text{GOTO. APPLY}\ \lambda_{21}\ \text{APPLY.}\ |_5\ |_4\ |_2\ |_1\\ 9\ 8\ |_5\ |_4\ |_2\ |_1\\ I\\ (9,({}_{15}\epsilon_2)I\\ I\end{array}\right.$$

$$\xrightarrow{9}\left[\begin{array}{l}({}_{15}\ |_2\ |_1\\ |_2\ |_1\\ I\\ I\\ I\end{array}\right.\quad\xrightarrow[\substack{\text{(these rules simply result in placing a pointer to the}\\ \text{current value of X on the result string)}}]{2,3,2,5.3,5.1,3,7,3,11,7,2,5.3,5.2,5.1,5.3,5.1,7,5.3,5.1}\left[\begin{array}{l}|_8\ |_7\ |_2\ |_1\\ 2\ |_8\ |_7\ |_2\ |_1\\ (8\leftarrow 7\ \pi=2)(7\leftarrow 6\ \beta=9)(6\leftarrow 2\ \alpha=2)I\\ (11,\lambda_{28}\epsilon_6)(10,\lambda_{27}\epsilon_2)(9,\lambda_{17}\epsilon_2)\\ I\end{array}\right.$$

$$\xrightarrow{11,11,11,11}\left[\begin{array}{l}\Lambda\\ 2\\ I\\ I\\ I\end{array}\right.\qquad\xrightarrow{12}4$$

Initialization and Termination of Evaluation (rules 1 and 12)*

The evaluation of an expression begins (rule 1) by initializing the control string with the head of the expression to be evaluated and the marker "$|_1$", initializing the result string with the marker "$|_1$", initializing the environment string with the string "$(1\leftarrow 1\ \pi=1)$", initializing the store string with the string "$(1,\Lambda)$", and initializing the expression string with the expression to be evaluated. Since the initial environment will generally contain the values of <u>no</u> free variables, the initial environment string contains the dummy variable π whose value is a pointer to the null string in the store. The marker "$|_1$" is placed on the control and result string to denote that the head of the expression is to be evaluated within the initial environment 1. In general, the subscript j of the leftmost $|_j$ in the control string denotes that the control string variables to the left of the $|_j$ are to be evaluated using the environment j, i.e., using the environment components linked to the component $(N_i\leftarrow M_i\ v_i=p_i)$ where $N_i=j$.

The evaluation terminates (rule 12) when the control string is null. When the control string is null, the result

*Rules 1 and 12 do not exactly follow the alternate notation for the evaluator given earlier. These rules are strictly given as

$$ht \quad \xrightarrow{1} \quad \langle h|_1 - |_1 - (\lambda_1 \leftarrow \lambda_1\ \pi=1) - (1,\Lambda) - ht\rangle$$

$$\langle \Lambda - p - x_3y_3 - x_4(p,r)y_4 - x_5y_5\rangle \quad \xrightarrow{12} \quad r$$

string will contain a pointer to some string value in the store. The string in the store is returned as the result of the evaluation. In general, the result of an evaluation is either a constant or a λ-closure. Strictly speaking, if the result of the evaluation is a λ-closure, the λ-expression and the values of its free variables should be returned as the result of the evaluation. If the result of the evaluation is a λ-closure, the λ-expression and the values of its free variables can be obtained from the environment, store, and expression strings specified prior to the termination of evaluation.

If a user were evaluating target language expressions with input-output facilities, (a) the initial values of the input and output strings (presumably those given on some device like a teletype or card reader) could be placed in the initial store string and (b) two system variables and pointers to their initial values could be placed on the initial environment string. The addition or removal of strings on the input or output device could then be defined by updating the values of the system variables to their new values. This is the mechanism used to define input-output in SNOBOL/1 (see Chapter IV).

Evaluation of Combinations (rule 2):

If a left parenthesis of a combination is at the left of the control string, the left parenthesis is removed from the control string,* and the head of its operand and operator are prefixed to the control string and the string "APPLY." is placed to the right of these two strings. Subsequent rules will evaluate the operand and operator, and then apply the value of the operator to the value of the operand to produce the value of the combination.

Evaluation and Application of λ-expressions (rules 3, 8, and 11):

If the name λ_i of a λ-expression is at the left of the control string (rule 3), the current environment j (initially the dummy environment 1) is obtained, the string "$\lambda_i \epsilon_j$" is placed in a new component at the left of the store string, and a pointer to the new store component is prefixed to the result string. The string "$\lambda_i \epsilon_j$" represents the λ-closure for λ_i in that (a) λ_i provides a name uniquely identifying the λ-expression λ_i contained in the expression string and (b) the environment component j provides the (linked) list of the pointers to the current values of the free variables of the λ-expression λ_i.

If the string "APPLY." is at the left of the control string, a pointer p to a λ-closure $\lambda_i \epsilon_j$ is at the left of the result string, and k is the index of the most recently added environment component (rule 8):

*In the discussion to follow, unless explicitly stated otherwise, the elements referred to at the left of the control string are assumed to be deleted from the control string after being evaluated.

(a) a new component ($1k \leftarrow j$ v=p'), where v is the bound variable of the λ-expression λ_i and p' is a pointer to the operand to which the λ-expression λ_i has been applied, is prefixed to the environment string. (This action results in setting the proper environment for evaluating the body of the λ-expression λ_i.)

(b) The head of the body of the λ-expression λ_i and a marker $|_{1k}$ are prefixed to the control string, and

(c) the pointers p and p' to the λ-closure and its operand are deleted from the result string and the marker $|_{1k}$ is prefixed to the result string.

If a marker $|_j$ is at the left of the control string and a pointer p and marker $|_j$ are at the left of the result string, the markers are deleted and the pointer p is left on the result string. The pointer will point to the value of applying the λ-expression to its operand.

Evaluation of Variables and Constants (rules 4 and 6):

If a variable is at the left of the control string, a pointer to the current value of the variable is prefixed to the result string (rule 4.1). The pointer is obtained by (a) obtaining the index j of the current environment and marking the environment component j with the symbol "°" (rule 4.3), and (b) then searching (rules 4.1 and 4.2) through the environment components linked to j for the occurrence of the variable.

If a constant is at the left of the control string (rule 6), a new store component containing the constant is prefixed to the store string, and the pointer to the new store component is prefixed to the result string.*

Evaluation of Label References (rules 5):

If a label reference $.\ell$ is at the left of the control string (rules 5), each environment component linked to the current environment component is searched for the occurrence of a component such that the λ-expression whose environment is specified by the component contains a body that is a sequence containing the label. If the label is found, a new store component $h\epsilon j$ containing the head of the expression following the label and the index j of the environment component is prefixed to the store, and a pointer to the new store component is placed on the result string. The head of the labeled expression and the environment index j provide a representation of the label-closure for $.\ell$ in that the head of the labeled expression uniquely identifies the labeled combination and the index j uniquely identifies the current environment of the sequence within which the combination occurs.

Transfer of Control (rule 10):

If the string "GOTO. APPLY." is at the left of the control string and a pointer p to a label closure $h\varepsilon_j$, where

*In the evaluator, all constants that are extended Markov algorithms must be enclosed by the quotation marks ' and h is the head of a labeled expression and j is the environment within which the labeled expression is to be evaluated, is at the left of the result string

(a) all portions of the control and result strings to the left of the markers $|_j$ are deleted, and

(b) the head of the expression following the label is prefixed to the control string.

This mechanism results in interrupting the evaluation of the current expression and continuing with the evaluation at the labeled expression using the environment j specified while evaluating the label-closure.

Application of Constants (rules 9.1 and 9.2):

If the string "APPLY." is at the left of the control string, and two store pointers p and p' to the strings s and s' are at the left of the result string, the string s is applied to the string s' (presumably s is an extended Markov algorithm and s' is the object string to which the algorithm is to be applied). The resulting string value is placed in a new store component, and the pointer to the new component is prefixed to the result string.

Assignment (rules 7.1 and 7.2):

If the string "ASSIGN. APPLY." is at the left of the control string and two store pointers p and p' are at the left of the result string, the string value in the store associated with p is changed to the string value associated with p'.

Addition of New Rules to the Evaluator:

It may happen that certain source language constructions are awkward to define solely within the target language and that these constructions can be more easily defined by adding new expressions to the target language and new evaluator rules to evaluate these expressions.

The rule applied to evaluate target language expressions is specified by the numerically first rule that is applicable to the current string values of the control, result, environment, store, and expression strings. By adding a rule to the evaluator whose left part specifies a configuration of the control, result, environment, store, and expression strings that, for the given configuration, provides a different transformation from the initial evaluator rules, the evaluator can be extended to define new types of target language expressions.

Generally, the rule applied by the evaluator is determined by the element at the left of the control string. For example, in the definition of indirect addressing in SNOBOL/1, it was desired to add a rule to the evaluator that would take some string value given in store and prefix the string value to the control string. The string value prefixed to the control string would then be evaluated in subsequent transformations as if the string value were itself a variable. By (a) allowing expressions of the form "(LOOKUP. X)", where X is a variable, in the target language translation of SNOBOL/1, and (b) adding the rule

$$\left.\begin{array}{r}\text{LOOKUP. APPLY.}\\ \text{p}\\ -\\ \text{(p,s)}\\ -\end{array}\right] \rightarrow \left[\begin{array}{l}\text{s}\\ \Lambda\\ -\\ \text{I}\\ -\end{array}\right.$$

to the evaluator, the extended evaluator defines indirect addressing. None of the initial evaluator rules are applicable to a configuration where the string "LOOKUP." is at the left of the control string; hence the rule can be placed in any numerical position within the initial sequence of rules.

3. Discussion

This chapter has presented a formally based target language in which the semantics of a computer language can be defined. The semantics of the target language was, in turn, defined in terms of the formalism of extended Markov algorithms by giving an extended Markov algorithm definition of a machine for evaluating target language expressions.

If used as a target language for the implementation* of

*Extended Markov algorithms have been implemented in the source language PANON-1B.[11,12]

a computer language, the target language allows the simple addition of built-in machine primitives. For example, if a computer has a built-in primitive for computing the sum of two integers, there is no need to define this primitive in the target language. This primitive can be used as a constant in the target language and in applying the primitive to its arguments the machine algorithm can be used. The point of using only extended Markov algorithms to define primitive functions is that for implementation of the target language the only necessary machine capability is that for implementing extended Markov algorithms. The fact that a given machine has certain built-in primitives simply relieves the person defining the semantics of a source language of defining the semantics of the built-in primitives in terms of extended Markov algorithms.

The target language is undesirable in one important sense. The computer language constructions for defining the assignment of new values to variables and for defining the transfer of control within a program required the addition of new expressions to the combined formalisms of extended Markov algorithms and the λ-calculus. The new expressions add to the complexity of the target language and place restrictions on the applicability of any theorems developed for λ-calculus expressions. This undesirable feature of the target language is, in part, redeemed in that the evaluator for the target language was completely defined within the

In the sixth century B.C. written language was continuous. There was no concept of breaking up units of expressions with punctuations marks. Kohmar Pehriad, a leading Macedonian literary figure, had the insightful idea of using a small round dot to indicate the end of a thought unit. Convinced of the utility of his invention, he spent almost thirty years of his life traveling through ancient Greece, Rome, and North Africa attempting to gain local acceptance of that small round dot. His effort was well-rewarded. The stark simplicity of his brilliant idea became popular so quickly that almost every written language used today uses the little round dot at the end of a unit of expression.

Pehriad's efforts did not stop with the dot. Recognizing the need for another mark to indicate pauses in the middle of thought units, he began using a dot with a curved descending tail in an expression to indicate a pause in the thought. This mark is, of course, quite familiar in our own language, and both the comma (Kohmar) and the period (Pehriad) have been named after their distinguished inventor.

CHAPTER IV

A DEFINITION OF THE SYNTAX AND SEMANTICS OF SNOBOL/1

In this chapter I attempt to demonstrate the thesis of this dissertation, that there should be formal definitions of the syntax and semantics of computer languages. As an example computer language, I have chosen SNOBOL/1, as initially defined by Farber, Griswold and Polonsky.[27] SNOBOL/1 was chosen as an example because (a) the language is simple enough to describe conveniently in a single chapter of this dissertation and (b) the language is fairly well-known. No knowledge of SNOBOL/1 will be assumed in this chapter. Rather, it is the intent of this chapter to define every construct (except character spacing) in the language. The definition of SNOBOL/1 will be in two parts: (a) an informal description of the language and of the techniques used in the formal definition in this chapter using the English language and (b) a formal description of the language in Appendix 3 using the formal system.

This chapter and the formal description of Appendix 3 may be viewed as a reference manual for SNOBOL/1. It is intended for a user who wishes a detailed description of the language.

The formal definition of SNOBOL/1 is divided into three parts. Appendix 3.1 gives the canonical system defining the syntax of SNOBOL/1, Appendix 3.2 gives the canonical system defining the translation of SNOBOL/1 into the target language, and Appendix 3.3 gives the definition of the primitive functions used in the target language. In writing the formal definition of the SNOBOL/1, it was necessary to resolve a few issues that were ambiguously or incompletely defined by the English language definition of the language given by Farber, Griswold and Polonsky.*

Introduction to SNOBOL/1

SNOBOL/1 is a language for defining transformations on strings of symbols. Programs in SNOBOL/1 are comprised of a linear sequence of rules of which there are four varieties: "input"rules for obtaining strings of symbols from some external input device (like a teletype or card reader), "assignment" rules for assigning names to strings, "pattern matching" rules for transforming strings into new strings, and "output" rules for writing strings on some external output device (like a teletype or card reader). In general, the behavior defined by each rule is executed in linear order. However, rules can be labeled with names and the

*For example, it was not clear whether the authors meant to permit or prohibit the use of the same variable name to denote different types of variables in a single pattern matching rule or whether to permit or prohibit the use of a name both as a string name and a label in the same program. I decide to prohibit the first of these constructions and to permit the second of these constructions.

ordinary sequence of execution interrupted and continued at some other labeled rule.

Introduction to the Techniques Used in Describing SNOBOL/1

The parts of this chapter will each describe some construct in the SNOBOL/1, e.g., a string, an arithmetic expression, a rule, or a statement. Each of these parts will consist of (a) portions of the productions from the canonical system of the translation (Appendix 3.2) of SNOBOL/1, (b) examples of the SNOBOL/1 constructs and their corresponding target language translations, and (c) an English language explanation of these constructs and their semantics as defined in the target language.

Theoretically, the (abbreviated) canonical system of the translation of SNOBOL/1 must be combined with the canonical system of the syntax of SNOBOL/1 to obtain the complete canonical system defining the set of legal programs and their target language translations. Nevertheless, except for the context-sensitive requirements on SNOBOL/1, the abbreviated canonical system of the translation of SNOBOL/1 provides a synopsis of a context-free specification of the language and its semantics in terms of the target language. Accordingly, the productions from the (abbreviated) canonical system of the translation will be used in the text to define the syntax and semantics of SNOBOL/1, and the specification of the context-sensitive requirements on syntax will be discussed at the end of the chapter.

As mentioned in the previous chapter, the first term of each term tuple in the specification of the translation of a language is generally of the form "s..t" where "s" represents some string in the source language and "t" represents the corresponding target language translation. The example SNOBOL/1 strings and their target language translations given in the text follow this notation.

Strings

```
DIGIT<0>,<1> ... ,<9>;
LETTER<A>,<B> ... ,<Z>;
MARK<%>,<.>,<=>, ... ,</>;
DIGIT<p> | LETTER<p> | MARK<p>  →  BASIC SYMBOL<p>;
BASIC SYMBOL<b>  →  STRING<SEQ(ℓ)>;
```

Example Strings:

```
ABC123%                      A ROSE IS A ROSE
HESSE,KAFKA,MANN             ALPHA
```

The basic symbols in SNOBOL/1 are the decimal digits, the capital English letters, and a variety of other symbols like "%", "." and "=". A string, the basic data type, consists of any linear sequence of basic symbols.

Names

```
DIGIT<p> | LETTER<p>  →  NAME<p>;
NAME<m>,<n>           →  NAME<mn>,<m.n>;
NAME<n>               →  STR NAME<n..n>,<$n..(LOOKUP. n)>;
NAME<n>               →  VAR NAME<n>;
NAME<n>               →  BACK REF NAME<n>;
```

Example Names:

ALPHA
ABC.EFG
$BETA
1234
12.3
$1234

A string can be assigned a name and the name used in place of the string. A name consists of a sequence of decimal digits and English letters, possibly including medial periods.

Besides designating a string, a name can be used in two other contexts, that of a string "variable" and that of a string "back reference." These three uses of names shall be distinguished by calling a name that designates a string a "string name," a name that designates a variable a "variable name," and a name that designates a back reference a "back reference name." A string name is treated as a variable in the target language.

A string name can be indirectly referenced by prefixing a string name with a dollar sign. The string value of a string name prefixed by a dollar sign is the string whose *name* is the string *value* of the name prefixed by the dollar sign. For example, if the string value of the name "BETA" is the string "A ROSE IS A ROSE" and if the string value of the name "A" is the string "BETA", the string value of "$A" is the string "A ROSE IS A ROSE". The primitive function "LOOKUP." is used to handle indirect addressing in the target language. "LOOKUP." is defined by an extended Markov algorithm substitution rule (Appendix 3.3d) that must be added to the target language evaluator.* When evaluated, this substitution rule inserts the string value of a name at the left of the control string. Thus the string is treated as if itself were a variable to be evaluated in subsequent steps taken by the evaluator.

Arithmetic Expressions

```
DIGIT<d>                          → DIGIT STR<SEQ(d)>;
DIGIT STR<s>                      → INT<s>,<-s>;
INT<i>                            → ARITH OPERAND<“i”..'i'>;
STR NAME<n..n'>                   → ARITH OPERAND<n..n'>;
ARITH OPERAND<a..a'>,<b..b'>      → ARITH EXP<a+b..(+(a',b'))>,
                                              <a-b..(-(a',b'))>,
                                              <a*b..(*(a',b'))>,
                                              <a/b..(/(a',b'))>;
```

```
Example Arith Operands:        Example Arith Expressions:

     “65”..'65'                  A+B .. (+(A,B))
    “-65”..'-65'                 A+“65”..(+(A, '65'))
       A..A                      A*“-65”..(*(A, '-65'))
```

SNOBOL/1 allows a limited type of arithmetic on strings whose contents are integers. An integer can be used directly as an arithmetic operand by enclosing the integer in the quotation marks “ and ”. A name whose string value is an integer can also be used as an arithmetic operand. An

As mentioned in the chapter describing the target language evaluator, it may occasionally be convenient to define some source language constructs by adding rules to the evaluator rather than by defining the constructs solely within the target language. To define indirect addressing in the target language would require complicated additions to the canonical system of the translation of SNOBOL/1 arithmetic expression consists of an arithmetic operand followed by one of the arithmetic operators "+", "-", "", and "/" (defined in Appendix 3.3b) followed by another arithmetic operand. The string value of an arithmetic expression is the string computed by applying the arithmetic operator to the integer value of the two operands.

String Expressions

```
STRING EXP<Λ..'Λ'>;
STRING<s>                    → STRING EXP< s ..'s'>;
STR NAME<n..n'>              → STRING EXP<n..n'>;
ARITH EXP<a..a'>             → STRING EXP<a..a'>;
STRING EXP<s..s'>,<t..t'> → STRING EXP<s□t..((CAT s') t')>;
```

Example String Expressions:

```
Λ..'Λ'                    NAME REVERSE..((CAT NAME) REVERSE)
“ABC123%”..'ABC123% '     “ABC” A..((CAT 'ABC') A)
A..A                      X Y Z..((CAT ((CAT X) Y)) Z)
$A..(LOOKUP. A)
```

A string expression in SNOBOL/1 is an expression whose value is a string. A string can be used directly in an arithmetic expression by enclosing the string in the quotation marks “ and ”. A string name or arithmetic expression can also be used in a string expression. A sequence of string expressions each separated by one or more spaces* comprises a complete string expression. The value of a string expression is the string computed by concatenating the string values of each of the component string expressions.

*The symbol "□" denotes one or more spaces.

Patterns*

```
STRING<s>                    → PAT EXP<“s”..'s'>;
STR NAME<n..n'>              → PAT EXP<n..n'>;
VAR NAME<n>                  → PAT EXP:SPECS<*n*..'n' : nεSTR|>;*
VAR NAME<n>                  → PAT EXP:SPECS<*(n)*..'n' : nεBAL STR|>;
VAR NAME<n>, DIGIT STR<d>    → PAT EXP:SPECS<*n/d*..'n' :
                                 (n,d)εFIX LN STR|>;
BACK REF NAME<n>             → PAT EXP<n..'n'>;
PAT EXP<p..p'>,<q..q'>       → PAT EXP<p q..((CAT p') q')>;
PAT EXP<p..p'>               → PATTERN<p..p'>;
```

Example Patterns:

```
“ABC”..'ABC'
X Y..((CAT X) Y)
*NAME*..'NAME' : NAMEεSTR |
*NAME* “,”..((CAT 'NAME') ',') : NAMEεSTR |
*X* “ABC” *(Y)*..((CAT((CAT 'X') 'ABC')) 'Y') : XεSTR | YεBAL STR |
*X* Y X..((CAT((CAT 'X') Y) 'X') : XεSTR |
```

A pattern in SNOBOL/1 is the basic unit through which string transformations are accomplished. A pattern can be viewed as an expression representing a set of strings.

A string enclosed by quotation marks is a pattern expression representing the set of strings containing one member, the string itself. A string name is a pattern representing the set of strings containing one member, the string value of the string name. A variable name enclosed by asterisks is a pattern expression representing the set of all strings of basic symbols. A variable name enclosed by parentheses and further enclosed by asterisks is a pattern expression representing the set of all strings containing balanced pairs of

*The use of the auxiliary term for the predicate part "SPECS" will be discussed shortly.

parentheses. A variable name followed by a slash and a positive integer and enclosed by asterisks is a pattern expression representing the set of all strings whose number of basic symbols is given by the integer following the slash. A name that occurs elsewhere in a pattern as a variable name is a pattern expression representing the same set of strings represented by the variable name. A name used in this context is called a back-referenced name.

A sequence of patterns of pattern expressions each separated by one or more spaces comprises a complete pattern. A sequence of pattern expressions represents the set of all strings composed by concatenating representative strings from each of the sets represented by the component pattern expressions. This set is restricted in that a string used in place of a back reference name must be identical to the string used in place of the corresponding variable name.

A pattern is used to scan a given object string for the existence of one of the strings represented by the pattern. If more than one string represented by the pattern occurs within the object string, the member M such that (a) each of the strings (except the last) concatenated to form M is, from left to right, as short as possible and (b) the last string concatenated to form M is as long as possible is taken as the occurrence of the pattern in the object string.

Pattern Matching Rules

```
STR NAME<n..n'>, STR EXP<s..s'>, PATTERN:SPECS:VAR REFS
   <p..p':c:v>  →  PAT MATCH RULE<nØp=s..
   (MATCH_AND_ASSIGN(n',p',λπ.s','c','(v)'>;
```

Example Pattern Matching Rules:

```
X “ABC”=..(MATCH_AND_ASSIGN(X, 'ABC', λπ.'Λ','', '()'))
X *NAME* “,”=..(MATCH_AND_ASSIGN(X, ((CAT 'NAME') ',')
                ,λπ.'Λ', 'NAMEεSTR |', '(NAME,)'))
X ALPHA = BETA..(MATCH_AND_ASSIGN(X, ALPHA, λπ.BETA,'', '()'))
```

A pattern matching rule consists of a string name followed by pattern, an equal sign, and a string expression. The execution of a pattern matching rule results in the following sequence of actions:

(a) The string value of the string name is scanned for the occurrence of the pattern.

(b) If the occurrence of the pattern is found

  (i) each string variable in the pattern is assigned the value of the substring used in matching the variable to the object string,

  (ii) the string expression is evaluated (using the new values of the string variables), and

  (iii) the occurrence of the pattern in the object string is replaced by the string value of the string expression and the string name is assigned the value of this newly formed string.

(c) If the occurrence of the pattern is not found, no action is taken.

The pattern matching capability of SNOBOL/1 is handled in the target language through the function "MATCH_AND_ASSIGN", (see Appendix 3.3c) which essentially forms an extended Markov algorithm that reflects the same transformation defined by the pattern. In the formation of the extended Markov algorithm, the variable and back reference names are treated as extended Markov algorithm string variables. Hence the translation of a variable or back reference name is given as a constant (see definition of patterns given previously), the variable names are specified as extended Markov algorithm string variables representing members of one of the sets "STR", "BAL STR", and "FIX LN STR" (see the auxiliary term for the predicate part "SPECS" in the definition of a pattern) defined in Appendix 3.1a, and the lists of variable names* and their set specifications are passed as arguments to the function "MATCH_AND_ASSIGN". The evaluation of the function "MATCH_AND_ASSIGN" results in the following actions:

(a) An attempt is made to match the pattern to the object string.

(b) If a match is found, the values of the variables are updated, the value of the string expression is computed, the name to which the pattern has been applied is updated to its new value, and the string "TRUE" is returned.

(c) If no match is found, the string "FALSE" is returned.

*The list of variable names is given by the auxiliary term for the auxiliary predicate part "VAR REFS" generated in the canonicalsystem for the syntax of SNOBOL/1. This auxiliary term is also generated in the complete (unabbreviated) canonicalsystem of the translation of SNOBOL/1 and is used to specify the translation of SNOBOL/1 as indicated above.

Input Rules and Output Rules

```
PATTERN:SPECS:VAR REFS<p..p':c:v>
  → INPUT RULE<SYS .READ p..(MATCH_AND_ASSIGN
    (READER#,p',λπ.'Λ','c',(v),'v'))>:
STRING EXP<s..s'>  →  OUTPUT RULE<SYS .PRINT s..
                      (PRINTER# ASSIGN.((CAT PRINTER#) s'))>;
```

Example Input and Output Rules:

```
SYS .READ *X* ..(MATCH_AND_ASSIGN(READER#, 'X', λπ.'Λ',
                                 'XεSTR |','(X,)', 'X,')) ;
SYS .PRINT REVERSE..(PRINTER# ASSIGN. ((CAT PRINTER#) REVERSE))
```

An input rule consists of the string "SYS .READ" followed by a pattern. An output rule consists of the string "SYS .PRINT" followed by a string expression.

The input and output of strings from some external input device is defined in the target language by assuming that there are two system variables "READER#" and "PRINTER#" that contain the initial values of the input and output strings.* When a string is input into a program, the value of the system variable "READER#" is changed to the string computed from the current value by deleting the string to be read in, and the values of the string variables in the pattern are updated. The pattern matching and updating of variables are handled through the function "MATCH_AND_ASSIGN" described previously.

*The initial values of these variables can be added to the initial environment named λ_1 in the target language evaluator.

When a string is output from a program, the value of the system variable "PRINTER#" is updated by appending the string value of the string expression.

Assignment Rules

```
STR NAME<n..n'>, STR EXP<s..s'> → ASSIGN RULE
     <n=s..(n' ASSIGN. s')>;
```

Example Assignment Statement:

```
  REVERSE = X REVERSE..(REVERSE ASSIGN. ((CAT X) REVERSE))
```

An assignment rule consists of a string name followed by an equal sign and a string expression. The execution of an assignment rule results in assigning the string value of the string expression to the string name.

Rules

```
PAT MATCH RULE<r..r'> | INPUT RULE<r..r'> | OUTPUT RULE<r..r'> |
    ASSIGN RULE<r..r'>          → UNLABELED RULE<r..r'>;
UNLABELED RULE<r..r'>           → RULE<@r..r'>;
UNLABELED RULE<r..r'>, NAME<n> → RULE<n@r..0n0r'>;
```

Example Rules:

```
   NAME = NAME REVERSE..(REVERSE ASSIGN. ((CAT NAME) REVERSE)
L4 NAME = NAME REVERSE.. L4; (REVERSE ASSIGN. ((CAT NAME) REVERSE)
```

A rule must be prefixed by a sequence of blank spaces or a name. A name prefixing a rule is called a label and is used to identify a rule when the normal order of evaluation is to be interrupted and to be continued at the labeled rule.

Statements

```
NAME<n>        →  LABEL EXP<n.. .n>;
STR NAME<n>    →  LABEL EXP<$n..(LOOKUP. ((CAT '.') n))>
RULE<r..r'>,  LABEL EXP<ℓ..ℓ'>,<m..m'>
  →  STM<r..r'>,<r/(ℓ)..r';(GOTO. ℓ')>;
        <r/S(ℓ)..r' ⇒ (GOTO. ℓ') ELSE ⇒ 'Λ'>,
        <r/F(m)..r' ⇒ 'Λ' ELSE ⇒ (GOTO. m')>,
        <r/S(ℓ)F(m)..r'  ⇒ (GOTO. ℓ') ELSE ⇒ (GOTO. m')>,
        <r/F(m)S(ℓ)..r'  ⇒ (GOTO. ℓ') ELSE ⇒ (GOTO. m')>;
```

Example Statement:

```
L3 REVERSE = "," NAME REVERSE  /(L2) ..
    L3:  (REVERSE ASSIGN. ((CAT((CAT ',') NAME)) REVERSE));
         (GOTO. .L2)
```

A label expression in SNOBOL/1 is an expression whose string value is a label. A label can be referenced directly by giving the name of a label or by giving a string name whose value is a label and prefixing the string name by a dollar sign.

A statement consists of one of the strings "r", "r/(ℓ)", "r/S(ℓ)", "r/F(m)", "r/S(ℓ)F(m)", or "r/F(m)S(ℓ)", where r is a rule and ℓ and m are label expressions. The execution of a statement of the form "r/(ℓ)" results in executing rule r and then transferring control to the statement designated by the label expression ℓ. The execution of a rule of the form "r/S(ℓ)" results in evaluating rule r and then transferring control to the statement designated by the label expression ℓ if the rule (presumably a pattern matching rule or input rule) succeeded in matching the pattern in the rule to its object string. Similarly, a statement of the form r/F(m) results in transferring control to the statement designated by m if the execution of rule r *failed* to match the pattern in the rule to its object string. Finally, statements of the form "r/S(ℓ)F(m)" or "r/F(m)S(ℓ)" result in transferring control to one of the statements designated by ℓ or m if the execution of rule r succeeded or failed in matching its pattern to its object string.

Statement Sequences*

```
STM<s..s'>                      → STM SEQ<s..s'>;
STM SEQ<q..q'>, STM<s..s'> → STM SEQ<q↘s..q';s'>;
STM SEQ<q..q'>, STRING<s>  → STM SEQ<q↘*s..q'>,<*s↘q..q'>;
```

Example Statement Sequence:

```
L4 REVERSE = X REVERSE      L4:(REVERSE ASSIGN.((CAT X)
                          ..       REFERSE));
   SYS .PRINT REVERSE          (PRINTER# ASSIGN.((CAT
                                   PRINTER#) REVERSE));
```

A statement sequence consists of a list of statements each on a new line. The statements are executed in order unless a statement explicitly specifies a transfer of control. Arbitrary character strings prefixed by an asterisk can be inserted among statements. The character strings provide comments for the programmer and are not evaluated.

*The symbol "↘" denotes a new line.

SNOBOL/1 Programs*

STM SEQ:STR REFS<q..q':s_r>, NAME<n>, LIST:BVS:CORR NULL LIST <s_r:v_b:ℓ> → SNOBOL PROGRAM<q END n..LET v_b=ℓ IN (GOTO. 'n'); q'>

Example Program:

```
     L1    SYS       .READ *X*
     L2    X         *NAME* “,” =           /s(L3)F(L4)
     L3    REVERSE   = “,” NAME REVERSE  /(L2)
     L4    REVERSE   = X REVERSE
           SYS       .PRINT REVERSE
     END   L1
```

Translation:

```
LET X,NAME,REVERSE = 'Λ','Λ','Λ'
IN         (GOTO. .L1);
   L1:     (MATCH_AND_ASSIGN(READER#,'X',λπ.'Λ',XεSTR |', '(X,)'));
   L2:     (MATCH_AND_ASSIGN(X,((CAT 'NAME) ','),λπ.'Λ',
                                      'NAMEεSTR |','(NAME,)'))
                ⟹ (GOTO. .L3)  ELSE ⟹ (GOTO. .L4);
   L3:     (REVERSE ASSIGN. ((CAT ((CAT ',') NAME)) REVERSE));
           (GOTO. .L2);
   L4:     (REVERSE ASSIGN. ((CAT X) REVERSE));
           (PRINTER# ASSIGN. ((CAT PRINTER#) REVERSE));
```

*Like the list of variable names, the list of string names used in a SNOBOL/1 is generated in the canonical system for syntax and is used in the canonical system for the translation to form the list of bound variables for the target language translation of a program.

The predicate "LIST:BVS:CORR NULL LIST" names a set of ordered triples, where the first element of each triple is a list of names (e.g., X,Y,X,ALPHA,Y,), the second element is a name list containing one occurrence of each name in the first list (e.g., X,Y,ALPHA), and the third element is a list of null strings with the same number of elements as the second list (e.g., "Λ","Λ","Λ"). This predicate is used to set the list of string names in a program to bound variables each with the initial value of a null string.

A SNOBOL/1 program consists of a statement sequence followed by a statement of the form "END n", where "END" is a label and "n" designates the label of some statement in the statement sequence. The execution of a program begins by initializing the string values of the string names in the program to null and then executing the statements in the program beginning with the statement labeled by "n".

The example program above reads in a string from the input device and outputs the string computed from the input string by reversing the order of each substring separated by a comma. For example, if the string "HESSE, KAFKA, MANN" is on the input device, the string "MANN, KAFKA, HESSE" is printed on the output device.

Context-Sensitive Requirements on the Syntax of SNOBOL/1

There are a few context-sensitive requirements on the syntax of SNOBOL/1:

(a) The variable names in a pattern must each be different.

(b) The back-reference names in a pattern must be identical to the variable names and different from the string names.

(c) The labels in a program must each be different and each reference to a label in a label expression must refer to a name that actually occurs as a label.

These requirements are specified in the canonical system for the syntax of SNOBOL/1 by specifying with each construct.

(a) the lists of names used as string names, variable names, and back reference names (productions 3 of Appendix 3.1),

(b) the lists of names used as labels (production 11.3) and names used to refer to labels (production 12.1),

and specifying

(a) that the list "r_v" of variable names in a pattern must contain names each of which is different (the premise "DIFF NAME LIST<r_v>" in production 6.8),

(b) that the list "r_b" of back reference names in a pattern must be contained within the list "r_v" of variable names and that the list "r_s" of string names in a pattern must be disjoint from the list "r_v" of variable names (the premise "L1:L2:INTERSEC <r_b:r_v:r_b>,<r_s:r_v:Λ>" in production 6.8), and

(c) that the list of labels in a program must contain names each of which is different and that each label reference must be contained in the list of labels (production 14).

The addition predicates "DIFF NAME LIST" and "L1:L2:INTERSEC" are defined at the end of Appendix 3.1.

This chapter has attempted to describe in detail the syntax and semantics of SNOBOL/1. It is intended that a reader, having digested this chapter, would have sufficient knowledge of SNOBOL/1 and its formal definition to be able to use the compact, formal definition to answer further questions concerning the syntactic legality or meaning of a given SNOBOL/1 construct. It is hoped that this chapter has served that objective.

CHAPTER V

A SPECIFICATION OF THE SYNTAX AND SEMANTICS OF ALGOL/60

This chapter exercises the formal system presented in this dissertation to specify the syntax and semantics of ALGOL/60, as defined in the official ALGOL/60 report edited by Peter Naur.[28] The intent of this chapter is not only to explicate the formal specification of ALGOL/60, but also to relate the techniques used in the formal specification of ALGOL/60 to other languages and to compare the formal system presented here to other methods of language specification. A knowledge of ALGOL/60 is assumed in this chapter.

It is surprising that, although ALGOL/60 is the official publication language of the Association for Computing Machinery and is accordingly widely-publicized, the author knows of no implementation of the complete language. Probably the most important factor in this circumstance is the complexity of ALGOL/60. Indeed, in writing this chapter I frequently found myself in the difficult situation of first attempting to understand ALGOL/60 and then attempting to characterize the language with the formal system. There are many interrelated program constructions and a complicated variety of restrictions on programs that make the language difficult to understand and define. Nevertheless, as an example of the formal system, applied to a somewhat complex computer language, a specification of the syntax and semantics of ALGOL/60 is presented in Appendix 4.*

Previous Work by Peter Landin:

In his paper[21] "A Correspondence Between ALGOL/60 and Church's Lambda Notation," Peter Landin described the semantics of ALGOL/60 in terms of a modified form of Church's λ-calculus, called "imperative applicative expressions" or "IAEs". The target language presented here is similar to Landin's imperative applicative expressions in that the λ-calculus was augmented to directly handle assignment and transfer of control features of ALGOL/60. The target language differs from imperative applicative expressions in that (a) the mechanism to handle transfer of control here is different from that of Landin, and (b) Landin's (SECD) machine to evaluate imperative applicative expressions is specified by a λ-calculus expression, whereas the machine to evaluate target language expressions here is specified by an extended Markov algorithm.

The specification of the semantics of ALGOL/60 given here is heavily based on Landin's definition. On the other hand, the dissertation here not only includes a specification of the semantics of ALGOL/60, but also a specification of syntax and a definition of the primitive functions used in

*The specification of character spacing and of the use of exponents in numbers is not included.

specifying the semantics. The primitive functions used to specify the semantics of ALGOL/60 are defined only by example in Landin's paper.

The Syntax of ALGOL/60

The canonical system specifying the syntax of ALGOL/60 is specified in Appendix 4.1. The first term in each specified term tuple describes some string in ALGOL/60. If the auxiliary predicate parts and terms are deleted from this specification, Appendix 4.1 can be viewed as a partial (context-free) specification of the syntax. A context-free specification of ALGOL/60's syntax exists in the ALGOL/60 report and the specification of Appendix 4.1 closely parallels the specification in this report. Although it does not completely specify the syntax of the language, the context-free specification of ALGOL/60 is fairly straight-forward and the presentation of the canonical system of ALGOL/60 will therefore focus on the context-sensitive requirements.

Context-Sensitive Requirements on the Syntax of ALGOL/60

There are myriad context-sensitive requirements on the syntax of ALGOL/60. Among these requirements are

(a) The type of each identifier in a program must be declared.

(b) An identifier cannot be used in conflicting contexts in the same block. There are <u>many</u> variants of this requirement. For example, an identifier used as a real variable in a block cannot be used as a boolean variable, an array identifier, a procedure identifier, or a switch identifier.

(c) Any use of an array identifier must occur with a subscript list of the same dimension as that of the bound pair list in the array declaration.

(d) The bound pair list in an array declaration can depend only on variables that are non-local to the block in which the array declaration is given.

(e) All statement labels in a block must be different.

(f) The uses of actual parameters in a function designator must be compatible with the uses of the corresponding formal parameters in the procedure declaration. There are <u>many</u>, <u>many</u> variants of this requirement. For example, an actual parameter that is declared to be a real variable cannot correspond to a formal parameter that is used as a boolean variable, an actual parameter that is a procedure identifier must correspond to a formal parameter that is used with arguments that are consistent with the procedure declaration, and an actual parameter that is an arithmetic expression cannot correspond to a formal parameter that is called by name and assigned a value in the procedure declaration.

The context-sensitive requirements on the syntax of ALGOL/60 occur in many other computer languages besides ALGOL/60. The restriction (a) that the type of each identifier must be declared occurs in many computer languages. For example, in PL/1 each occurrence of an identifier used to name an object must be declared, either explicitly, contextually, or implicitly. An explicit declaration of an identifier is given through a DECLARE statement, whereby an identifier is given an attribute restricting the use of the identifier to statements operating on certain classes of data, e.g., fixed point numbers, character strings, or files. A contextual declaration of an identifier is given when an identifier occurs in a context where only one class of data objects can occur, e.g., in the statement "GET FILE (X) DATA" the identifier "X" is contextually declared as a member of the class file in that only a file name can occur after the string "GET FILE" in a GET statement. An implicit declaration of an identifier is given when an identifier is associated with other declared identifiers (e.g., in the statement "T = A * B", if "A" and "B" are declared as fixed point numbers, the identifier T may be implicitly declared as a fixed-point number). Programs not specifying a unique declaration for each identifier are illegal.

The restriction (b) that identifiers cannot be used in conflicting contextx occurs in almost every language where different classes of data objects are distinguished. For example, although PL/1 allows some identifiers to be used in different contexts, many contexts of declared identifiers are considered illegal, e.g., if "X" is explicitly declared as a bit string, the statement "GET FILE (X) DATA" is illegal since the GET statement contextually declares "X" as a file.

The restriction (e) that all statement labels in a block must be different occurs in almost every language allowing statements to be labeled and control to be passed to a labeled statement. The labels must be different in order for the destination of the transfer of control to be unique. For example, in Fortran IV no two statements may be labeled with the same statement number.

The restriction (f) that corresponding actual and formal parameters must be compatible likewise occurs in many languages and can become complicated, especially in languages allowing nested procedure definitions and applications like ALGOL/60.

The author knows of only one major computer language where a complete formal specification of its syntax has been given. In particular, the simulation language GPSS has been specified completely by Donovan,[3] using canonic systems. Otherwise, the syntax of many computer languages has been specified either informally or has been partially formalized, usually with a context-free grammar.

Before discussing the specification of the context-sensitive requirements on the syntax of ALGOL/60, the reader is reminded that the auxiliary predicate parts and terms in a production generally specify the lists of identifiers, labels, variables, etc., that are used within the source language string specified by the first term in the production. These lists will be referred to repeatedly in the productions to follow.

Specification of the Requirement that the Type of Each Variable Must be Declared:

Consider the (abbreviated) production* from the canonical

*The productions given in the text will generally be only portions of the corresponding productions given in Appendix 4. Portions of productions are given in the text to illuminate better the particular construction under discussion. An explication of the complete canonical system for ALGOL/60 will be given later in the chapter.

system of the syntax of ALGOL/60:

ID<i> → REAL VAR:R VARS<i:i,>;

If "i" designates a string that is an identifier, the term tuple "<i:i,>" designates a pair where the first element is an identifier used as real variable, and the second element designates the addition of the identifier to the list of identifiers *used* as real variables in a program. Consider also the production

IDLIST<ℓ> → TYPE DEC:DEC R VARS<REAL ℓ:ℓ,> ;

If "ℓ" designates a string that is a list of identifiers, the term tuple "<REAL ℓ:ℓ,>" designates a pair where the first element is an ALGOL/60 declaration of a list of identifiers as real variables, and the second element designates the addition of the list of identifiers to the list of identifiers *declared* as real variables.

Next consider the production

STM SEQ:R VARS<s:v_r>, DEC SEQ:DEC R VARS<d:v_{rd}>,
L1:L2:REL COMP<v_r:v_{rd}:v'_r>
→ BLOCK:R VARS<BEGIN d;s END:v'_r>;

Here, if

(a) "s" is a statement sequence with a list "v_r" of identifiers *used* as real variables

(b) "d" is a declaration sequence with a list "v_{rd}" of identifiers *declared* as real variables

(c) "v'_r" is the list computed from "v_r" and "v_{rd}" by forming their relative complement (i.e., "$v_r - v_{rd}$")

then

(d) "BEGIN d;s END" is a block with a list "v'_r" of identifiers that are used as real variables in the block but not declared within the block

Finally, consider the production

PROGRAM STR:R VARS<p:Λ> → ALGOL PROGRAM<p>;

Here, if (a) "p" is a string that is in the form of a program and (b) the list "R VARS" of identifiers that are used in the program as real variables but are not yet declared is given as null, then the string "p" is specified as a bone fide legal ALGOL program.

In this manner (a) each identifier in a program <u>used</u> as a real variable is added to the list of used real variables, (b) each identifier <u>declared</u> as a real variable is added to the list of declared real variables, (c) each identifier declared in a block as a real variable is removed from the list of identifiers used as real variables, and (d) a string is specified as a legal program only if the list of used (but as yet undeclared) real variables is given as null.

Specification That Identifiers Cannot be Used in Conflicting Contexts:

Consider the following production

```
STM SEQ:R VARS:B VARS<s:$v_r$:$v_b$>,  DEC SEQ<d>,
   DISJ ENTRY LISTS<($v_r$)($v_b$)>  →  BLOCK<BEGIN d;s END>;
```

where the predicate "DISJ ENTRY LISTS" specifies a set consisting of one or more identifier lists each enclosed in parentheses such that each list is disjoint from the others. If "v_r" and "v_b" specify the lists of identifiers used respectively as real variables and boolean variables, in a statement sequence, the premise "DISJ ENTRY LISTS<(v_r)(v_b)>" insures that the string "BEGIN d; s END" is a legal block only if the lists "v_r" and "v_b" are disjoint, i.e., not used in conflicting contexts.

Specification That Actual and Formal Parameters Must Be Compatible:

The requirements on the uses of actual and formal parameters of ALGOL/60 procedures is complicated. For example, let "P(X,A)" be a declared procedure with two formal parameters "X" and "A", where in the declaration of "P", "X" is used as a real variable and "A" is used as an integer array of dimension three. The function disignator "P(3.1,Q)", where "Q" is a declared integer array of dimension three would constitute a legal activation of the procedure "P", whereas the function designator "P(TRUE,Q)" would not be legal since the type "REAL" of "X" and the type "BOOLEAN" of "TRUE" are not compatible.

To specify the context-sensitive requirements on procedures, a number of additional predicates are defined. For simplicity, in the discussion to follow I will assume that ALGOL/60 has only three data types: real variables, boolean variables, and integer arrays. Consider the following productions:

```
DIMM<1>;
DIMM<m>   →  DIMM<m1>;
SPEC<REAL>,<BOOLEAN>;
DIMM<m>                      →  SPEC<INTEGER ARRAY(m)>;
SPEC<s>                      →  SPEC LIST<s>;
SPEC<s>,  SPEC LIST<ℓ>       →  SPEC LIST<ℓ,s>;
```

Here the predicate "SPEC" specifies a set comprising the strings {REAL BOOLEAN INTEGER ARRAY(1) INTEGER ARRAY(11) INTEGER ARRAY(111) ...}, where each string specifies the use of some formal parameter in a procedure declaration. The predicate "SPEC LIST" specifies a set where each member is a string of parameter specifications each separated by a comma.

For example, if "P" is a procedure declared as above, the specification list for the formal parameters of "P" would be "REAL,INTEGER ARRAY(111)". Similarly, if "P(3.1,Q)" and "P(TRUE,Q)" are function designators where "Q" is declared as an integer array of dimension three, the specification list for "P(3.1,Q)" would be "ARITH EXP,INTEGER ARRAY(111)" and the specification list for "P(TRUE,Q)" would be "BOOL EXP,INTEGER ARRAY(111)". In the specification of the syntax of ALGOL/60, a predicate "SPEC MATCH" is defined. The ordered pair "<ARITH EXP,INTEGER ARRAY(111):REAL,INTEGER ARRAY(111)>" is a member of this predicate, and thus, by using this predicate as a premise in the canonical system for ALGOL/60, the function designator "P(3.1,Q)" is allowed as a compatible function designator with the above indicated declaration of "P". On the other hand, the ordered pair "<BOOL EXP,INTEGER ARRAY(111):REAL,INTEGER ARRAY(111)>" is not a member of this predicate, and thus the function designator "P(TRUE,Q)" is not allowed as a compatible function designator for "P".

Since the number of data types in ALGOL/60 is much greater than the number of types assumed in the examples just given, the actual specification of the context-sensitive requirements is much more complicated than indicated in the previous paragraphs. A detailed discussion of the complete canonical system specification of the context-sensitive requirements on ALGOL/60 procedures is given at the end of this chapter.

The Semantics of ALGOL/60

It seems that much less work in computer science has been directed to formalizing semantics than in formalizing syntax. While many methods for characterizing (at least in part) the syntax of computer languages have been successfully developed, few methods for characterizing semantics have reached a development where entire languages have been characterized. An application of the λ-calculus has been used by Peter Landin[21] and John Wozencraft[25] to characterize respectively the semantics of ALGOL/60 and the classroom language PAL. The characterization of semantics given in this dissertation is in part based on these efforts.

A quite different approach to characterizing semantics has been taken by the IBM Vienna laboratory, which has undertaken the formidable task of characterizing the semantics of PL/1. This group has used portions of LISP, the predicate calculus, set theory, and other constructs of their own invention to characterize the semantics of PL/1. Their work has been described in several lengthy IBM technical reports. A judgment of the utility of their approach awaits a more digestible presentation of the formal system and the techniques used within the formal system.

The specification of the semantics of ALGOL/60 in terms of the target language presented here is given in Appendix 4.2. Much of the semantics of ALGOL/60, e.g., arithmetic expressions, boolean expressions, designational expressions, conditional statements and statement sequences, are straightforwardly defined in the target language and in part have been discussed in previous chapters. I will therefore focus the discussion of this chapter on some constructs in ALGOL/60 whose semantics are not quite as obviously expressed in terms of the target language.

The table on the following pages lists several example ALGOL/60 expressions and their translations into the target language. In the discussion to follow, the reader may find it helpful to refer to these examples.

EXAMPLE ALGOL/60 EXPRESSIONS AND THEIR TRANSLATIONS
INTO THE TARGET LANGUAGE

Syntactic Type	ALGOL/60 Expression	Translation into the Target Language
IDS	A1	—
NUM	65	'65'
NUM	-65	(NEGATE '65')
NUM	65.32	(+(TRANS_INT '65',TRANS_FRAC '32'))
ID	A	A
ID	X	(X 'Λ') --if X is a formal parameter called by name
VAR	A	A
VAR	B[1,X]	(GET_EL(⟦CONV_TO_INT '1'),(CONV_TO_INT X⟧,B))
FCN DES	P	(P 'Λ')
FCN DES	Q(X,Y,Z*Z)	(Q(λπ.X,λπ.Y,λπ.(*(Z,Z))))
ARITH EXP	A+B*C	(+(A,*(B,C)))
ARITH EXP	IF B THEN 0 ELSE 1	B ⇒ '0' ELSE ⇒ '1'
DES EXP	ALPHA	.ALPHA
DES EXP	009	.9
DES EXP	S[X]	((GET_EL(CONV_TO_INT X, S)) 'Λ')
COMMENT STM	COMMENT THIS IS A COMMENT	'Λ'
GOTO STM	GO TO 009	(GOTO. .9)
ASGT STM	P := X	LET π=(CONV_TO_INT X) IN (P# ASSIGN. π) If P is an integer procedure identifier
ASGT STM	A := B := X	LET π=(CONV_TO_INT X) IN LET α=B IN (α ASSIGN. π); LET α=A IN (α ASSIGN. π) If A and B are integer vars
FOR LIST EL	X STEP 1 UNTIL 5	λπ.STEP(λπ.X,λπ.'1',λπ.'5')
FOR STM	FOR V=1,2 DO V:=V+1	(FOR(V,DELAY_CAT⟦λπ.'1',λπ.'2'⟧,LET π=(CONV_TO_INT(+(V,1))) IN LET α=V IN (α ASSIGN. π)
UNCOND STM	ALPHA: GO TO 009	ALPHA: (GOTO. .9)

COND STM	IF B=TRUE
	THEN GO TO ALPHA
		(=(B, 'TRUE')) ⇒ (GOTO. .ALPHA) ELSE ⇒ 'Λ'

TYPE DEC	REAL X,Y,Z	X,Y,Z = 'Λ','Λ','Λ'
TYPE DEC	OWN REAL X,Y,Z	X,Y,Z = X#1,Y#1,Z#1
ARRAY DEC	REAL ARRAY A[1:10, 1:10]	A = (MAKE_LIST(⟦'1','1'⟧,⟦'1','1'⟧⟦'10','10'⟧
ARRAY DEC	OWN REAL ARRAY A[1:10,1:10]	A = (RESET_LIST(A#1⟦'1','1'⟧⟦'1','1'⟧⟦'10','10'⟧)
SW DEC	SWITCH S:=ALPHA,009	S = (INDEX_LIST('1'⟦ALPHA,.9⟧)
PROC DEC	REAL PROCEDURE P(X,Y) VALUE X: P:=X+Y	P(X,Y) = LET P#,X = 'Λ',(UNSHARE (X 'Λ')) IN LET π=(CONV_TO_REAL (+(X,(Y 'Λ'))) IN (P# ASSIGN. π); P#

BLOCK	BEGIN REAL X,Y;
	X := Y := 3;
	A := X*Y
	END
		LET REC X,Y='Λ','Λ'
		IN LET π=(CONV_TO_REAL '3') IN LET α=Y IN (α ASSIGN. π);
		LET α=X IN (α ASSIGN. π);
		LET π=(CONV_TO_REAL (*(X,Y)) IN LET α=A IN (α ASSIGN. π)

ALGOL PROGRAM	BEGIN REAL A,B;
	REAL PROCEDURE F(X,Y);
	VALUE X;
	F := X+Y/2;
	A := 3;
	B := A+F(4,A);
	END
		LET REC A,B,F(X,Y)='Λ','Λ',LET F#,X='Λ',(UNSHARE (X 'Λ'))
		IN LET π=(CONV_TO_REAL(+(X,(/((Y 'Λ'),'2'))))
		IN (F# ASSIGN. π); F#
		IN LET π=(CONV_TO_REAL '3') IN LET α=A IN (α ASSIGN. π);
		LET π=(CONV_TO_REAL (+(A,F(λπ.'4',λπ.A))))
		IN LET α=B IN (α ASSIGN. π)

Primitive Functions Used to Define the Semantics of ALGOL/60:

Appendix 4.3 defines the primitive functions used in defining the semantics of ALGOL/60. Appendices 4.3a and 4.3b define miscellaneous primitives, like the function "NEQ" for negating a boolean value, the function "HD" for computing the head of a list, and the function "ABS" for computing the absolute value of a number. Real numbers in ALGOL/60 are represented in the target language by their fractional equivalent. A fraction in the target language is a string of the form "xDy", where x and y represent respectively the numerator and denominator of the fraction. For example, the real number "1.5" in ALGOL/60 is translated into the target-language string "3D2" denoting the traction three-halves (3 Divided by 2). Appendix 4.3c defines the primitives "TRANS_INT" and "TRANS_FRAC" for converting real numbers to their fractional representation and the primitives "CONV_TO_REAL" and "CONV_TO_INT" for converting integer numbers to real numbers and real numbers to integer numbers. Appendices 4.3d and 4.3e define the arithmetic and boolean primitives.

Appendices 4.3f and 4.3g define the primitives used in defining the semantics of for statements and arrays and will be discussed later in the text.

Primitive functions similar to those given for ALGOL/60 can be used to define the semantics of many languages used for numerical processes. For example, in FORTRAN IV, the arithmetic and boolean primitives almost exactly parallel those for ALGOL/60. Although FORTRAN IV allows the user to (a) specify one of two precisions for real number arithmetic and (b) specify arithmetic for complex numbers, these facilities can be readily specified in the target language by (a) defining a primitive that converts target language fractions to the desired precision as real numbers and (b) defining the arithmetic operators for complex numbers in terms of those given for real numbers. Similarly, the FORTRAN IV facilities for arrays and DO statements closely parallel the ALGOL/60 facilities for arrays and for statements.

Assignment of Values to Variables and Procedures:

Consider the following ALGOL/60 assignment statements:

```
A := X
F := X
A := F := X
```

where "X" is an integer variable, "A" is a real variable, and "F" is a real procedure identifier. The corresponding target language expressions for these statements are:

```
LET π = (CONV_TO_REAL X) IN LET α = A IN (α ASSIGN. π)
LET π = (CONV_TO_REAL X) IN (F# ASSIGN. π)
LET π = (CONV_TO_REAL X) IN (F# ASSIGN. π);
                            LET α = A IN (α ASSIGN. π)
```

The expression on the right side of an assignment statement must be evaluated only once. Therefore, the translation of the right-hand expression is evaluated once and is linked with the dummy variable "π" and the value of π is used in each target language assignment expression. The primitive "CONV_TO_REAL" is applied to "π" before the assignment to convert the value of "π" to a real number.

Assignments in the target language can only be made to target language variables. The ALGOL/60 variables in the left side of the assignment statement are linked with the dummy target language variable "α" to handle the case where the ALGOL/60 variable is a formal parameter called by name and the ALGOL/60 variable must be translated into a target language expression that is not a variable. (This point will be discussed shortly.) By linking the dummy variable α with the translation of expression representing the ALGOL/60 variable, an assignment to α will also result in an assignment to the corresponding ALGOL/60 variable.

The assignment of a value to a procedure in a procedure declaration is handled by affixing the mark "#" to the procedure identifier and assigning the value of the right-hand expression to this newly formed identifier. The "#" is affixed to the identifier to avoid conflicts with the use of the procedure identifier in a recursive call to the procedure. In the translation of the entire procedure declaration, the translation of the last statement in the declaration is followed by the statement "F#", where F is the procedure identifier. Thus the evaluation of the procedure will return the value currently assigned to the procedure identifier.

Parameters Called by Name and Called by Value:

Consider the following ALGOL/60 procedure declaration:

```
PROCEDURE F(X,Y); VALUE Y;
   BEGIN
       Y := Y+Y;
       X := Y*Y;
   END
```

In this procedure declaration the formal parameter "X" is called by name and the formal parameter "Y" is called by value. If "A" and "B" are real numbers whose current values are "1" and "2", the evaluation of the procedure statement

```
F(A,B);
```

results in changing the value of "A" to "4" while leaving the value of "B" unchanged.

Next consider the following target language translations of the procedure declaration given above and procedure statement "F(A,B)":

```
LET F(X,Y) = LET Y = (UNSHARE (Y 'Λ'))
             IN  LET π = (CONV_TO_REAL (+(Y,Y)))
                         IN LET α = Y       IN (α ASSIGN. π);
                 LET π = (CONV_TO_REAL (*(Y,Y)))
   and                   IN LET α = (X 'Λ') IN (α ASSIGN. π)
F(λπ.A, λπ.B)
```

Here, the translations of the actual parameters "A" and "B" are given as functions mapping the dummy variable "π" into the variables of "A" and "B". In the evaluation of the procedure statement "F(A,B)", the function "$\lambda\pi$.B" will be applied to the null string (causing the evaluation of "B") and the function "UNSHARE" (Appendix 4.3a) will be applied to this value (causing the formation of a new cell in the store for the value of "B". Thus subsequent assignments to the formal parameter "Y" will not result in changing the value of "B". On the other hand, the function "UNSHARE" is not applied to "X" and the assignment of a value to "X" will result in changing the value of the corresponding actual parameter "A".

Lists in ALGOL/60:

In defining the semantics of ALGOL/60, it will be convenient to define primitive functions operating on lists of strings. I will use the notation

$$s_{1+}\ s_{2+}\ \cdots\ {}_{+}s_n$$

where the s_i, $1\leq i\leq n$, are strings, to denote a list. If X_1, X_2, ... ,X_n are expressions whose values are the strings s_1, s_2, ... ,s_n, the expression

(1) $((\text{CAT} \ldots ((\text{CAT}\ ((\text{CAT}\ ((\text{CAT}\ X_1)\ '_+'))\ X_2))\ '_+'))\ \ldots\ X_n)$

will result in forming the list

$$s_{1+}\ \ s_{2+}\ \ \cdots\ \ {}_{+}s_n$$

The concatenation of expressions to form lists will occur frequently in the formal definition of ALGOL/60. For convenience, I will generally omit the explicit specification of the concatenation of the component expressions of a list and write list expressions of the form (1) in the alternate notation

$$[x_{1+}\ \ x_{2+}\ \ \cdots\ \ {}_{+}x_n]$$

Arrays and Switches:

An array in ALGOL/60 is treated in the target language as an indexed linear list, where the number of elements in the list equals the number of elements in the array. For example, an array with a bound pair list

[1:2,1:3]

is translated into the string

$$(1_{+}1,\Lambda)_{+}(1_{+}2,\Lambda)_{+}(1_{+}3,\Lambda)_{+}(2_{+}1,\Lambda)_{+}(2_{+}2,\Lambda)_{+}(2_{+}3,\Lambda)$$

where the symbol "Λ" specifies an initial null value for each element of the array. The translation of arrays into lists is handled through the function "MAKE_LIST" (Appendix 4.3g), which converts the bound pair list of the array into a linear list of array elements each with an initial null value. An element of an array is obtained through the function "GET_EL", (Appendix 4.3g), which, given a subscript list and an array identifier, obtains the appropriate array element. The elements of an array are updated with new values through the function "RESET_LIST", which resets the value of one of the array elements in the array list.

Switches are also treated as linear lists. For example, a switch with a switch list "L,M,N" is translated into the target language string "$[(1,\lambda\pi.\ .L)_{+}(2,\lambda\pi.\ .M)_{+}(3,\lambda\pi.\ .N)]$ The elements of the target language list are given as dummy variable functions so that an element of a switch list is not evaluated unless the element is selected by a designational expression. The translation of switches into lists is handled through the primitive function "INDEX_LIST" (Appendix 4.3g), which forms an indexed list of switch elements. An element of a switch list is obtained by applying function "GET_EL" to the switch list and then applying the selected element to the null string. This application results in forming the proper label-closure for the label.

Own Variables:

Consider the following outlined ALGOL/60 program:

```
BEGIN
   REAL X,Y,Z;
   PROCEDURE F(A);  BEGIN OWN X;  ... END;
   .
   .
   .
END
```

and its target language translation

```
     LET   X#1 = 'Λ'
     IN    LET REC X,Y,Z,F(A) = 'Λ','Λ','Λ',LET X =X#1 IN ...
           IN
           .
           .
           .
```

The variable "X" in the ALGOL/60 procedure "F" is an own variable, and hence on successive calls to the procedure "F" the value of "X" is not re-initialized to a null value but maintains the value last assigned to "X" on the previous call. In the target language translation of the program, a new global identifier "X#1" is created, and on each call to "F" the value of "X" is set to the value of "X#1". In this manner an assignment to the value of "X" will also result in an assignment to "X#1". Since "X#1" is global to the entire target language expression, "X#1" will maintain the value last assigned to "X" and subsequent calls to "F" will result in resetting "X" to its last assigned value.

The mark "#" and positive integer are affixed to the global own identifiers so that these identifiers will not conflict with other identifiers in the target language expression.

Own arrays are treated similarly to own variables in that the own array identifiers are coupled with corresponding global identifiers. The global array identifiers are initialized with null values. Upon each entry to a block with an own array,

(a) the value of the global array identifier is updated to the value computed from the current value of the global identifier by (1) retaining the values of the array elements whose indices, as specified by the current value of the bound pair list, occur in the array list for the global identifier, and (2) setting to null the values of the array elements whose indices do not occur in the array list for the global identifier, and

(b) coupling the value of the own array identifier with the value of the corresponding global array identifier.

Thus, upon the first entry to the block, each element of the own array will be given as null. Since updating the value of the local own array identifier will also result in updating the value of the corresponding global array identifier, subsequent entry to the block will result in resetting the values of the previously given elements of the own array identifier to their previous values and setting the value of each array element not included in the previous bound pair list to null.

Own variables and own arrays have generally caused problems for those implementing languages with own variables in that special programs and storage areas have been needed to properly implement own variables. The above mechanism for handling own variables in the target language is quite straightforward and avoids the complexity generally associated with own variables

Goto Statements:

A statement of the form "GO TO L" in ALGOL/60, where L is a label reference, will result in interrupting the normal order of evaluation and continuing by evaluating the statement labeled by L in the same sequence or in the first encompassing block containing a statement with a label L. The mechanism for transferring control to a target language expression in the same or an encompassing sequence has been discussed in the chapter III.

On the other hand, a more complicated situation for transferring control occurs when a label is passed as an argument to a procedure.* For example, consider the procedure statement

```
F(L)
```

and the procedure declaration

```
PROCEDURE F(X);  LABEL X;
    BEGIN
    .
    .
    .
    GO TO X;
    .
    .
    .
    END
```

Since in the target language, the procedure statement is translated as

```
F(λπ. .L)
```

where the λ-closure for "λπ. .L" is evaluated relative to the

*Formal parameters that are labels called by *value* are excluded according to the ALGOL/60 report.

environment within which the *procedure* statement occurs and the GO TO statement is translated as

(GOTO. (X 'Λ'))

the label-closure for X will refer to the labeled statement in the block in which *procedure statement* occurs (or to a labeled statement in an encompassing block) and the environment given by the label closure will refer to the environment of the block specified at the time when the procedure statement was evaluated.

Furthermore, consider the ALGOL/60 program:

```
BEGIN INTEGER A,B;
      PROCEDURE F(I,X);  LABEL X;  VALUE I
         BEGIN    M:    B := B+1;
                        I := I+1;
                        IF  B=4  THEN GO TO L1;
                        IF  B=3  THEN GO TO X;
                        IF  B=2  THEN F(I,X);
                        IF  B=1  THEN F(I,M)   END F;

     A := B := 0;
     F(A,L1);
  L1: A := A*A
END
```

Here F is a *recursive* procedure that is called three times. On the second call to F the local label M is passed as an argument; the label-closure for M will specify an environment within which the value of I is 1. On the third call to F the GO TO statement "GO TO X" will result in resetting the environment within which the value of I is 1, and upon exiting from the procedure the value of I will be 2, and *not* 3.

Recursive Definitions:

ALGOL/60 allows the declaration of variables, arrays, switches, and procedures that can depend on each other. For example, the following declaration sequence can occur within a block

```
REAL PROCEDURE H1(X1); IF X1=0   THEN  1
                       ELSE X1•H2(X1-1);
REAL PROCEDURE H2(X2); IF X2=0   THEN  1
                       ELSE X2•H1(X2-1)
```

These declarations constitute a simultaneous recursive definition of the factorial function (e.g., the value of the function designator "H1(4)" is "24").

If E1, E2, and S are statements, and H1 and H2 are procedure identifiers that are (possibly) defined simultaneously recursive, the ALGOL/60 block

```
BEGIN
REAL PROCEDURE H1(X1);  E1;
REAL PROCEDURE H2(X1);  E2;
S
END
```

can be correctly defined by the target language translation

(1) (λπ.(λH1.(λH2.s (HD π)) (TL π)) (Y^2 λH1.λH2.[λX2.e2$_+$λX1.e1]))

where e1, e2, and s are the target language expressions for the ALGOL/60 statements E1, E2, and S and the fixed point operator Y^2 is

```
λF. LET π1,π2='Λ','Λ'
    IN  LET Z=((F π1) π2)
        IN  (π1 ASSIGN. HD Z);
            (π2 ASSIGN. TL Z);
            Z
```

Extending the alternate notation for recursive definitions given earlier, an expression of type (1) will be alternately written

```
LET REC H1,H2=λX1,e1,λX2.e2
    IN  s
```

and further rewritten

```
LET REC H1(X1),H2(X2)=e1,e2
    IN  s
```

More generally, if H1, H2, ... , Hk are declared variables, arrays, switches, or procedure identifiers whose target language translations are the expressions t1, t2, ... , tk, and s is the target language translation of the a statement, an expression of the form

```
(2)  (λπ.(λH1.(λH2...(Hk.s (1st π)) (2nd π)) ... (kth π))
          (Y^k λH1.λH2...λHk.[tk+...+t2+t1]))
```

where

```
1st π = (HD π)
2nd π = (HD (TL π))
  .
  .
  .
kth π = (HD (TL (TL ... π)...))
Y^k   = λF. LET π1,π2,...,πk='Λ','Λ',...,'Λ'
          IN  LET Z=(...((F π1) π2) ... πk)
              IN  (π1 ASSIGN. (HD Z));
                  (π2 ASSIGN. (HD (TL Z)));
                   .
                   .
                   .
                  (πk ASSIGN. (HD (TL (TL .. π)..));
                  Z
```

and

if Hi, $1 \le i \le k$, is a procedure definition of j variables
XI,X2, ... , Xj
then the expression ti is given as λX1.λX2...λXk.ei, where
ei is the target language translation of the procedure
body,

will correctly define the (possibly simultaneous recursive) definitions in s.

Further extending the alternate notation for k simultaneous recursive definitions, an expression in the target language of form (2) will alternately be written

```
LET REC H1,H2,...,Hk=t1,t2,...,tk
IN  s
```

Furthermore, if H_i , $1 \leq i \leq k$, is a procedure definition of j variables X1,X2,...,Xj, then Hi and ti will be given as Hi(X1,X2,...,Xj) and ei, where ei is the target language translation of the procedure body.

For Statements:

Consider the following ALGOL/60 for statement:

```
 (1)  FOR  X:=1, 2 STEP 2 UNTIL 7  DO  X:=X+1
```

Here, since the control variable is itself updated in the statement "X:=X+1", the statement "X:=X+1" is evaluated only three times, for the values of the control variable "X" equal to "1", "2" and "5". The critical point in this evaluation is that the increment for the control variable "X" is *delayed* until the statement following the "DO" is executed, possibly changing the current value of the control variable. Similarly, the evaluation of a for statement of the form

```
 (2)  FOR  X:=Q, U STEP V UNTIL W  DO  s;
```

where "s" is some statement, can result in changing the values of "X", "U", "V", or "W" before each iteration of the statement. The delay in the evaluation of for list elements is handled through the use of dummy variable functions. For example, consider the following function definitions:

```
REC STEP(A,B,C) = LET A',B',C' = (A 'Λ'),(B 'Λ'),(C 'Λ')
                  IN  (B'≥0)Λ(C'<A')  ⇒  'Λ'
                      (B'≤0)Λ(A'<C')  ⇒  'Λ'
                      ELSE            ⇒ [A' λπ.(STEP(λπ.
                                          +   (+(A',B')), B,C))]
```

```
REC DELAY_CAT L = LET     H,T = HD L, TL L
                  IN LET H'  = (H 'Λ')
                     IN  (T = 'Λ')     ⇒ H'
                         (H' = 'Λ')    ⇒ (DELAY_CAT T)
                         ELSE          ⇒ [H'+T]
```

```
REC FOR(V,L,S) = LET H,T = HD L, TL L
                 IN  (L = 'Λ')    ⇒  'Λ'
                     ELSE         ⇒  V := H; (S 'Λ');
                                     FOR(V, (DELAY_CAT T),S)
```

and the following target language translation of the for statement (2)

```
FOR(X,(DELAY_CAT [λπ.Q+λπ.(STEP(λπ.U,λπ.V,λπ.W))], s') *
```

Here the function "DELAY_CAT", when applied to the list of dummy variable functions in a for list, produces (a) the null string or (b) the evaluation of the next element in the for list followed by the dummy variable functions representing the remaining elements in the for list. The function "FOR" successively evaluates the statement within the for statement for each of the successively computed elements in the for list.

The semantic constructs in ALGOL/60 are similar to those in many other computer languages for performing numerical calculations, e.g., FORTRAN, MAD, AED and portions of PL/1. The semantic constructs in SNOBOL/1, defined in the previous chapter, appear in part in several languages for string manipulation, e.g., PANON/1B, TRAC and CONVERT. The characterization of certain important linguistic features, like

*s' represents the target language translation of the source language statement s.

structures in PL/1 and AMBIT/G and real-time operations in PL/1, has not yet been attempted with the target language presented in this dissertation. I suspect that the delay feature in evaluating target language expressions will prove useful in defining real-time operations and that modifications to the target language will be needed to characterize conveniently operations on structured data. Nevertheless, the characterization of SNOBOL/1 and ALGOL/60 have provided significant tests of the target language in defining semantics, and it is expected that future research will yield modifications and extensions of the concepts presented here to define more varied computer languages.

Since the discussion in this chapter has focused on a simplified exposition of certain constructs in ALGOL/60, the remainder of this chapter will be devoted to a detailed explanation of the complete formal definition of ALGOL/60, as given in Appendix 4.

Two Abbreviations for the Canonical Systems of ALGOL/60:*

Besides the abbreviations introduced earlier, two abbreviations have been added to the notation for canonical systems in writing the canonical systems for ALGOL/60. The first of these abbreviations allows the user to abbreviate constructions defining an alternating sequence of two other

*The remaining portions of this chapter are for those who wish to study in detail the formal definition of ALGOL/60 given in Appendix 4.

constructions (for example, defining a "for list," which consists of a sequence of for list elements each separated by a comma). Examples of the variants of this abbreviation are given in examples 7 in the table on the following page. The formal definition of this abbreviation is given in productions 21 of Appendix 1.3.

The second of these abbreviations generally allows the user to use a slash to abbreviate productions that are repeated for each of the constructions defining real, integer, and boolean quantities in ALGOL/60. An example of the use of this abbreviation is given in example 8 in the table on the following page. The formal definition of this abbreviation is given in productions 22 of Appendix 1.3.

Notes on the Canonical System Defining the Syntax of ALGOL/60:

Predicates Needed to Specify Context-Sensitive Requirements:

To specify the context-sensitive requirements on the syntax of ALGOL/60, a number of additional predicates (S31 through S41) are used. The predicate "TYPE" (S31.1) defines a set of three members, the strings "REAL", "INTEGER", and "BOOLEAN". The predicate "DIMM" defines a set consisting of strings of ones, where the number of ones in a string gives the dimension of an array. The predicate "SPEC" defines a set of strings, where each string specifies the use of some formal parameter in a procedure declaration. The predicate

EXAMPLES OF ABBREVIATIONS USED IN THE CANONIC SYSTEMS OF ALGOL/60

7a.	UNABR PRODS	FOR LIST EL<e>	→	FOR LIST<e>;
		FOR LIST EL<e>, FOR LIST<ℓ>	→	FOR LIST<ℓ,e>;
	ABR PRODS	FOR LIST EL<e>	→	FOR LIST<ALTSEQ(e ,)>;
7b.	UNABR PRODS	PRIM<p>	→	TERM<p>;
		PRIM<p>, MULT OP<m>, TERM<t>	→	TERM<tmp>;
	ABR PRODS	PRIM<p>, MULT OP<m>	→	TERM<ALTSEQ(p m)>;
7c.	UNABR PRODS	FOR LIST EL<e..e'>	→	FOR LIST<e..e'>;
		FOR LIST EL<e..e'>, FOR LIST<ℓ..ℓ'>	→	FOR LIST<ℓ,e..ℓ,e'>;
	ABR PRODS	FOR LIST EL<e..e'>	→	FOR LIST<ALTSEQ(e ,)..ALTSEQ(e' ,)>;
7d.	UNABR PRODS	PRIM<p..p'>	→	TERM<p..p'>;
		PRIM<p..p'>, MULT OP<m>, TERM<t..t'>	→	TERM<tmp..(m(t',p'))>;
	ABR PRODS	PRIM<p..p'>, MULT OP<m>	→	TERM<ALTSEQ(p m)..APPLIC(p' m)>;
7e.	UNABR PRODS	BOOL SEC<s..s'>	→	BOOL FAC<s..s'>;
		BOOL SEC<s..s'>, BOOL FAC<f..f'>	→	BOOL FAC<fAs..(A(f',s'))>;
	ABR PRODS	BOOL SEC<s..s'>	→	BOOL FAC<ALTSEQ(s A)..APPLIC(s' A)>;
8.	UNABR PRODS	ID<i>	→	REAL VAR:R VARS<i:i,>;
		ID<i>	→	INT VAR:I VARS<i:i,>;
		ID<i>	→	BOOL VAR:B VARS<i:i,>;
	ABR PRODS	ID<i>	→	REAL/INT/BOOL VAR:R/I/B VARS<i:i,>;

"SPEC LIST" defines a set where each member is a string of parameter specifications each separated by a comma. For example, if "P" is a declared procedure with two formal parameters "X" and "A", and "X" is used as a real variable and "A" is used as an integer array of dimension three, the specification list for the occurrence of the procedure declaration is "REAL,INTEGER ARRAY(111)".

The predicate "SPEC1:SPEC2:COMB" (S33) defines a set of triples, where the first element is a parameter specification designating some use of a formal parameter, the second element is a parameter specification designating some other compatible use of the parameter, and the third element the parameter specification designating their combined use. For example, if the formal parameter "X" were used in three contexts, as a real variable in an arithmetic expression, as a real variable in a subscript list, and as a real variable that is assigned a value in an assignment statement, the following triples could be generated

<Λ:REAL:REAL>   <REAL:REAL:REAL>   <REAL:ASGNED:REAL ASGNED>

designating the combined use of "X" as a "REAL ASGNED" variable. Note that if X is used both as a real and a boolean variable, there is no way to combine the specifications "REAL" and "BOOLEAN" to obtain the specification of the combined use of "X". In the generation of legal programs, the use of this predicate prevents the generation of illegal procedure declarations containing such incompatible uses of formal parameters.

The predicate "SPEC MATCH" (S34) defines a set of ordered pairs, where the first element is the parameter specification of an actual parameter, and the second element is a compatible parameter specification of the corresponding formal parameter. The predicate "SPEC LIST MATCH" augments this set to include lists of parameter specifications. For example, if "P" is a procedure as defined above and "Q" is a declared integer array of dimension three, the function designators "P(3.1,Q)" and "P(TRUE,Q)" would have specification lists "ARITH EXP, INTEGER ARRAY(111)" and "BOOLEAN EXP,INTEGER ARRAY(111)". The specification list "REAL,INTEGER ARRAY(111)" would <u>match</u> the specification list "ARITH EXP,INTEGER ARRAY(111)" but would <u>not</u> <u>match</u> the specification list "BOOL EXP,INTEGER ARRAY(111)". Thus the use of this predicate prevents the use of incompatible formal and actual parameters.

The predicate "USES:PARS WITH SPECS" (S35) defines a set of ordered pairs, where the first element of each pair contains several lists of formal parameters with each list followed by a parameter specification enclosed in parentheses* (e.g., "X,Y,Z,(REAL) A(111),B(1111),(BOOLEAN ARRAY))", and

*If the formal parameter is an array identifier, the identifier may be followed by the dimension of its subscript list; if the formal parameter is a procedure identifier, the identifier may be followed by the specification list for its actual parameters.

the second element contains the list of formal parameters with each formal parameter followed by its parameter specification (e.g., "X REAL,Y REAL,A BOOLEAN ARRAY(111),B BOOLEAN ARRAY(1111)"). The predicate "PARS:USES:SPECS" defines a set of triples, where the first element is a list of formal parameters (e.g., "X,Y,A,B"), the second element is a list of the uses of the parameters (e.g., "X REAL,Y REAL,A BOOLEAN ARRAY(111),B BOOLEAN ARRAY(1111)"), and the third element the parameter specification list for the parameters (e.g., "REAL,REAL,BOOLEAN ARRAY(111),BOOLEAN ARRAY(1111)"). This predicate is used to generate the specification list for the formal parameters in a procedure declaration.

The predicate "ENTRY" (S36) defines the set of elements that can occur as auxiliary lists in the canonic system for ALGOL/60. An entry is either an identifier, or an array identifier followed by the dimension of the subscript list given with the array identifier, or a procedure identifier followed by the specification list of the actual parameters given with the procedure identifier. The predicates "DIFF CHAR", "DIFF STR ", "DIFF ENTRY", "IN", "NOT IN", "NOT CONT", "DIFF ENTRY LIST", "DISJ ENTRY LIST", "L1:L2:INTERSEC" and "L1:L2:REL COMP" are similar to those given for SNOBOL/1. One important exception in the similarity for the ALGOL/60 predicates and the SNOBOL/1 predicates occurs in the definition of the predicate "IN" (S38.1). An entry is considered to be contained <u>in</u> a list of other entries only if the dimension of an array identifier or the specification list of a procedure identifier *matches* each of the dimensions of other identical array identifiers or the specification lists of other identical procedure identifiers.

Specification of the Context-Sensitive Requirements:

In general, the context-sensitive requirements on the syntax of ALGOL/60 are specified by specifying a number of auxiliary lists with each syntactic unit and later specifying that each of these lists has certain properties. The lists specify (a) the identifiers *declared* as real, integer, boolean, or switch variables (S24 and S26.2), (b) the identifiers *used* as real, integer, boolean, or switch variables (S8.3, S9.1 and S12.2), (c) the identifiers *declared* as real, integer, or boolean arrays (S25.9 and S25.10), (d) the identifiers *used* as real, integer, or boolean arrays (S8.4 and S9.3) (e) the identifiers *declared* as real, integer, boolean, or non-valued procedures (S27.12) (f) the identifiers *used* as real, integer, boolean, and non-valued procedures (S9.2, S9.9 and S9.10) (g) the labels* (S20.2 and S21.3) and label references (S12.1), (h) the procedure identifiers and variables

*Leading zeros in a numeric label do not effect the value of the label. For example, the strings "00149", "0149", and "149" each denote the label with value "149". Thus, a label is defined (S4) in the canonical system by a set of ordered pairs, where the first element is a label and the second element is its value. The auxiliary lists of labels and label references contain the *values* of each label string.

that are assigned a value in an assignment statement (S18.1 and S18.2), and (i) the variables used in the arithmetic expressions in an array declaration (S25.1).

The specification of the restrictions on each of these lists is complicated. The lists of formal parameters, parameters called by value, and labels in a procedure declaration must contain identifiers each of which a different (predicate "DIFF ENTRY LIST" in S27.12). The lists of formal parameters used as real, integer, boolean and switch variables, the lists of formal parameters used as real, integer, and boolean arrays, the lists of formal parameters used as real, integer, boolean and non-valued procedures, the lists of formal parameters used to reference labels, and the lists of assigned procedure identifiers must each be disjoint (predicate "DISJ ENTRY LISTS" in S27.12). The lists of declared identifiers and labels in a block must each contain different identifiers (predicate "DIFF ENTRY LIST" in S29). The lists of identifiers used as variables, arrays, procedures, and labels must each be disjoint (predicate "DISJ ENTRY LISTS" in S29).

The lists of identifiers used in a procedure declaration but not specified as formal parameters (the primed variables in S27.12), the lists of identifiers used in a block but not declared in the block (the double primed variables in S29), and the lists of identifiers used in the bound pair list of an array declaration (the variables with a subscript "m" in S29) must be obtained and specified as used identifiers in the procedure declaration or block. Furthermore, with each declaration (S25.4) or use (S8.4 and S9.3) of an array identifier, the dimension m of the associated bound pair list or subscript list is kept with the identifier in the auxiliary lists of declared and used arrays. Similarly, with each procedure declaration (S27.12) and function designator (S9.2, S9.9 and S9.10), the specification list x of the formal or actual parameters is kept with the identifier in the auxiliary lists of declared and used procedures. The specification list for a procedure declaration is obtained through the predicate "PARS:USES:SPECS" discussed earlier. The restrictions that the dimension of each use of an array identifier must match its declared dimension and that the actual and formal parameter lists must be compatible are specified through the predicates "PARS:SPECS:USES", "L1:L2:REL COMP" and "L1:L2 :INTERSEC" as discussed earlier.

Finally, a string is defined as a syntactically legal program only if the lists of used but not declared variables, arrays, procedures, labels, label references, and assigned procedure identifiers are each given as null (S30.3).

Notes on the Canonical System Specifying the Translation of ALGOL/60

Three additional predicates (T42) are used in the specification of the translation of ALGOL/60 into the target language. The predicates "LIST:CORR NULL LIST", "LIST:CORR UNSHARE LIST", and "LIST:CORR INDEXED LIST" define sets of ordered pairs where the first element of each pair is a list of identifiers (e.g., "X,Y,Z,") and the second element of each pair is respectively (a) the corresponding list of null strings (e.g., "'Λ','Λ','Λ',")* (b) the corresponding list of expressions applying the function "UNSHARE" to each identifier (e.g., "(UNSHARE (X 'Λ')),(UNSHARE (Y 'Λ')),(UNSHARE (Z (Y 'Λ'),", and (c) the corresponding list of identifiers each followed by a "#" and a positive integer (e.g., "X#1,Y#1,Z#1,").

*In the target language these lists are used in expressions like "LET X,Y,Z, = 'Λ','Λ','Λ', IN ...". Strictly speaking, the last comma in each list should be removed.

CHAPTER VI

DISCUSSION

This thesis describes a formal system for defining the rules for writing programs in a computer language and for defining what these programs mean. The author strove for simplicity of the formal system, and then applied the formal system to define two complete computer languages, ALGOL/60 and SNOBOL/1.

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

Canonical systems were used here to define the syntactically legal strings in a computer language and the translation of the legal strings into strings in some other language. Not once was it necessary to step outside the formalism to define the syntax or translation of a language. Although some complexity was added to the formalism by introducing abbreviations to the basic notation, even the abbreviations were ultimately defined in terms of the basic formalism.

Extended Markov algorithms and the λ-calculus were used as a basis for defining semantics. Prior to this effort, work has been done by others in using formalisms like recursive function theory, Markov algorithms, formal graph theory, and the λ-calculus to characterize computational processes. However, the marriage of extended Markov algorithms to the λ-calculus is to my knowledge the first attempt where two formalisms have been intimately combined to characterize computational processes. Almost every construction in SNOBOL/1 and ALGOL/60 was solely within the combined formalism. The introduction of new expressions to the combined formalism to mirror the assignment and transfer of control constructions in SNOBOL/1 and ALGOL/60 appeared unavoidable. Nevertheless, these additions accomplished complete definitions of the semantics of both languages. Moreover, the entire target language was eventually defined by an extended Markov algorithm defining a machine for evaluating strings in the target language.

The extended Markov algorithm definition of the target language evaluator not only reduced the definitions of semantics to a single formalism, but also demonstrated that a computer possessing only the characteristics needed to evaluate an extended Markov algorithm is sufficient to execute source language programs translated into the target language. The conventional machine facilities existing in most computers, like those for performing arithmetic and logical operations and those for transferring control within a program, are not needed to evaluate target language programs, although they may be convenient. On the other hand, such horribly detailed machine facilities, like those for shifting bits or branching on the setting of a mask, appear to be useless in evaluating target language programs. The ability to use extended Markov algorithms as the basic evaluating mechanism for computational processes suggests that machine languages quite different from those conventionally used might be more effective for defining computational processes. However, this subject is, at least, worth another doctoral dissertation.

One may well ask: Why was one formalism, canonical systems, used to define the syntax and translation of a language? Why was another pair of formalisms, extended Markov algorithms and the λ-calculus, used to define the semantics of a language? And why was just extended Markov algorithms used to define the target language evaluator? The following are my answers. First, it appears convenient to define the syntax and translation of a language with a generative grammar (which canonical systems provide) that frees the language designer from the details of specifying a scanning algorithm for determining whether a source language string is acceptable. Second, a computer language generally specifies some well-defined algorithm for performing a computation, and hence it seems somewhat natural to define the semantics of a computer language with some simpler algorithmic formalisms (like extended Markov algorithms and the λ-calculus). Third, extended Markov algorithms alone were sufficient to define the target language evaluator. Fourth, the considerations of naturalness and perspecuity arise again. The formalism of canonical systems seemed well-suited to define the syntax and translation of a language, the combined formalism of extended Markov algorithms and the λ-calculus readily lent themselves to defining what a language means, and extended Markov algorithms provided the desired concise definition for the target language evaluator. In short, different formalisms model different processes with different degrees of complexity.

I have attempted to separate the specification of the syntax and semantics of a language into three parts: (1) the specification of the legal strings in a language, (2) the specification of the translation of the legal strings into the target language, and (3) the specification of the primitive functions used in the target language. Although each of these specifications must depend on the others for their correctness, the specification of the primitive functions in the target language were written for the most part after the specification of the translation of the source language into the target language and resulted in few changes to the definition of translation. On the other hand, it is unfortunate that the specifications of the syntax and translation depended heavily on each other. A change in the specification of the syntax often required a change in the specification of the translation, and vice versa. It would certainly be valuable to develop a convention that would better isolate the specification of the syntax and translation.

Although the semantics of a source language was formally defined here by the target language, and although canonical systems specify only the syntax of a language, a large portion of the semantics of the source language was somewhat imperceptively defined in the canonical system defining only the syntax of the language. By using descriptive predicate names like "ARITH EXP", "COND STM", and "LABEL", a correspondence with the English language was made to aid the reader's understanding of what was being talked about, i.e., the semantics of the constructions being defined. A similar use of the English language occurs in a Backus-Naur form specification of a computer language. The use of metalinquistic variables, like "ARITH EXP", "DIGIT", and "PRIMARY" in productions like "<ARITH EXP> :: = <DIGIT> | <PRIMARY>", does convey some idea of what the specified strings mean, although strictly speaking the productions define only certain legal strings in a language. In this way both canonical systems and Backus-Naur form make good uses of one of the most popular meta-languages, the English language.

There are several immediate uses of the formal system presented here. First, when developing a language, it would be desirable to have a formal definition specifying precisely what strings are allowed in the language and what the strings mean. Such a formal definition could be given to others for their analysis and would sharpen the debate over whether the convenience of each construction in the language would be worth the difficulty in explaining or implementing the construction. Second, after the designers agreed upon the constructions in the language, the formal definition would be valuable to those implementing the language or those preparing the language manuals in that they would know unambiguously what was intended by the language designer.

The formal system presented here opens several avenues for future research. As previously mentioned, since canonical systems can define precisely both the syntax and translation of a language, canonical systems might be used as the basis for automatic translation between computer languages. If an efficient algorithm could be developed to recognize strings specified by a canonical system and generate their translation, a canonical system definition of a language could be immediately used to translate legal programs in the language into another language. Another use of the formal system might be in the implementation of "extensible" computer languages. By simply adding or changing the productions defining the syntax and semantics of a language, the new productions could be given to the algorithm for translating strings specified by a canonical system, thereby implementing the extended language.

The author has attempted to integrate and adapt three known formalisms to define computer languages. These formalisms have been blended into a formal system for defining computer languages rigorously and somewhat concisely. The most significant portions of the attempt here are the application of canonical systems, the marriage of extended Markov algorithms with the λ-calculus, and the application of extended Markov algorithms to define an evaluator for the target language. It is hoped that this work is a progressive step in achieving the thesis of this dissertation, to meet the need for formal methods for completely defining computer languages.

Appendix 1.1 CANONICAL SYSTEM SPECIFYING THE SYNTAX OF A SUBSET OF ALGOL/60

(a) Basic notation only

1.1	DIGIT	DIGIT<1>;
1.2		DIGIT<2>;
1.3		DIGIT<3>;
2.1	VAR	VAR<A>;
2.2		VAR<B>;
3.1	PRIMARY	DIGIT<d> → PRIMARY:VARS<d:A>;
3.2		VAR<v> → PRIMARY:VARS<v:v,>;
3.3	ARITH EXP	PRIMARY:VARS<p:v> → ARITH EXP:VARS<p:v>;
3.4		PRIMARY:VARS<p:v>, ARITH EXP:VARS<a:u> → ARITH EXP:VARS<a+p:uv>;
3.5	STM	ARITH EXP:VARS<a:u>, VAR<v> → STM:VARS<v:=a : v,u>;
4.1	TYPE LIST	TYPE LIST<A>;
4.2		TYPE LIST<B>;
4.3		TYPE LIST<A,B>;
4.4	DEC	TYPE LIST<t> → DEC:DEC VARS<INTEGER t:t,>;
5.	PROGRAM	STM:VARS<s:u>, DEC:DEC VARS<d:v>, IN<u:v> → PROGRAM<BEGIN d;s END>;
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<x:t>, IN<y:t> → IN<xy:t>;

(b) with abbreviations

1.	DIGIT	DIGIT<1>,<2>,<3>;
2.	VAR	VAR<A>,<B>;
3.1	PRIMARY	DIGIT<d>; → PRIMARY<d>;
3.2		VAR<v> → PRIMARY:VARS<v:v,>;
3.3	ARITH EXP	PRIMARY<p> → ARITH EXP<p>;
3.4		PRIMARY<p>, ARITH EXP<u> → ARITH EXP<a+p>;
3.5	STM	ARITH EXP<a>, VAR<v> → STM:VARS<v := a:v,>;
4.1	TYPE LIST	TYPE LIST<A>,<B>,<A,B>;
4.2	DEC	TYPE LIST<t> → DEC:DEC VARS<INTEGER t:t,>;
5.	PROGRAM	STM:VARS<s:u>, DEC:DEC VARS<d:v>, IN<u:v> → PROGRAM<BEGIN d; s END>
6.1	IN	IN<A,:A,>, <A,:A,B,>, <B,:B,>, <B,:A,B,>;
6.2		IN<x:t>,<y:t> → IN<xy:t>;

Appendix 1.2 CANONICAL SYSTEM SPECIFYING THE TRANSLATION OF THE SUBSET INTO ASSEMBLER LANGUAGE*

(a) Basic notation only

1.1	DIGIT	DIGIT<1>;
1.2		DIGIT<2>;
1.3		DIGIT<3>;
2.1	VAR	VAR<A>;
2.2		VAR<B>;
3.1	PRIMARY	DIGIT<d> → PRIMARY:VARS<d..=F'd':A>;
3.2		VAR<v> → PRIMARY:VARS<v..v:v,>;
3.3	ARITH EXP	PRIMARY:VARS<p..p':v> → ARITH EXP:VARS<p.. L 1,p' *LOAD p:v>;
3.4		PRIMARY:VARS<p..p':v>, ARITH EXP:VARS<a..a':u> → ARITH EXP:VARS<a+p..a'↲ A 1,p' *ADD p:uv>;
3.5	STM	ARITH EXP:VARS<a..a':u>, VAR<v> → STM:VARS<v := a..a'↲ ST 1,v *STORE RESULT IN v:v,u>;
4.1	TYPE LIST	TYPE LIST<A..A DS F>;
4.2		TYPE LIST<B..B DS F>;
4.3		TYPE LIST<A,B..A DS F↲B DS F>;
4.4	DEC	TYPE LIST<i..i'> → DEC:DEC VARS<INTEGER i..i':i,>;

5.	PROGRAM	STM:VARS<s..s':u>, DEC:DEC VARS<d..d':v>, IN<u;v>
		→ PROGRAM<BEGIN d; s END..*ASSEMBLER LANGUAGE PROGRAM↲ BALR 15,0
		*SET BASE REGISTER↲ USING *,15 *INFORM ASSEMBLER↲s'↲ SVC 0
		*RETURN TO SUPERVISOR↲*STORAGE FOR VARIABLES d'↲ END>;

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<x:i>, IN<y:i> → IN<xy:i>;

(b). with abbreviations

3.1	PRIMARY	DIGIT<d> → PRIMARY<d..=F'd'>;
3.2		VAR<v> → PRIMARY<v..v>;
3.3	ARITH EXP	PRIMARY<p..p'> → ARITH EXP<p.. L 1,p' *LOAD p>;
3.4		PRIMARY<p..p'>, ARITH EXP<a..a'> → ARITH EXP<a+p..a'↲ A 1,p' *ADD p>;
3.5	STM	ARITH EXP<a..a'>, VAR<v> → STM<v := a..a'↲ ST 1,v *STORE RESULT IN v>;
4.1	TYPE LIST	TYPE LIST<A..A DS F>,<B..B DS F>,<A,B..A DS F↲B DS F>;
4.2	DEC	TYPE LIST<i..i'> → DEC<INTEGER i..i'>;
5.	PROGRAM	STM<s..s'>, DEC<d..d'> → PROGRAM<BEGIN d; s END..*ASSEMBLER LANGUAGE PROGRAM↲ BALR 15,0 *SET BASE REGISTER↲ USING *,15 *INFORM ASSEMBLER↲s'↲ SVC 0 *RETURN TO SUPERVISOR↲ *STORAGE FOR VARIABLES d'↲ END>;

*The symbol "↲" denotes a new line.

Appendix 1.3 CANONICAL SYSTEM SPECIFYING THE RULES FOR CONSTRUCTING AND USING ANOTHER CANONICAL SYSTEM

(a) Productions defining the rules for constructing a canonical system

1.	OBJ ALPHA	OBJ ALPHA<0>,<1>, ...,<9>,<A>,<B>, ...,<Z>,<:=>,<;>, ... <,>;
2.1	PRED CHAR	PRED CHAR<0>,<1>, ...,<9>,<A>,<B>, ...,<Z>;
2.2	PRED PART	PRED CHAR<a> → PRED PART<a>;
2.3		PRED CHAR<a>, PRED PART<p> → PRED PART<pa>;
2.4	PRED ALPHA	PRED PART<p> → PRED ALPHA<p>;
2.5		PRED PART<p>, WF PRED<q> → PRED ALPHA<p`:´q>;
3.1	VAR ALPHA	VAR ALPHA<`a´>,<`b´>, ...,<`z´>;
3.2		SUB OR SUPERSCRIPT<$_0$>,<$_1$>, ...,<$_9$>,<$_a$>,<$_b$>, ...,<$_z$>,<'>;
3.3		VAR ALPHA<v>, SUB OR SUPERSCRIPT<s> → VAR ALPHA<vs>;
4.1	WF TERM	WF TERM<A>;
4.2		OBJ ALPHA<a> → WF TERM<a>;
4.3		VAR ALPHA<a> → WF TERM<a>;
4.4		WF TERM<t>,<r> → WF TERM<tr>;
4.5	WF TERM TUPLE	WF TERM<t> → WF TERM TUPLE<<t>>;
4.6		WF TERM<t>, WF TERM TUPLE<<r>> → WF TERM TUPLE<<r`:´t>>;
5.	WF ATOM FORM	PRED ALPHA<p>, WF TERM TUPLE<t> → WF ATOM FORM<pt>;
6.1	WF PREMISE	WF ATOM PROD<p> → WF PREMISE<p>;
6.2	WF CONCLUSION	WF ATOM PROD<p> → WF CONCLUSION<p>;
7.1	WF ATOM PROD	WF CONCLUSION<c> → WF ATOM PROD<c;>;
7.2	WF PROD	WF ATOM PROD<p> → WF PROD<p>;
7.3		WF PREMISE<p>, WF CONCLUSION<c> → WF PROD<p→c;>;
7.4		WF PROD<s→c;>, WF PREMISE<p> → WF PROD<s,p→c;>;
8.1	WF CANONICAL SYSTEM	WF PROD<p> → WF CANONICAL SYSTEM<p>;
8.2		WF PROD<p>, WF CANONICAL SYSTEM<c> → WF CANONICAL SYSTEM<cp>;

(b) Productions defining the rules for deriving strings specified by a canonical system

9.1	PROD	CANONICAL SYSTEM STR<s>, WF CANONICAL SYSTEM<s> → CANONICAL SYSTEM<s>;
9.2		CANONICAL SYSTEM<spt>, WF PROD<p> → PROD<p>;
10.	PROD INSTANCE	PROD<p>, SUBST<p:v:s:q>, STR WITH NO VARS<q> → PROD INSTANCE<q>;
11.1	DERIVATION	DERIVATION<A>;
11.2		DERIVATION<d>, PROD INSTANCE<c;>, WF CONCLUSION<c> → DERIVATION<d c>;
11.3		DERIVATION<d>, PROD INSTANCE<p→c;>, PREMS:DERIV CONT PREMS<p:d> → DERIVATION<d c>;
12.1	OBJ STR	OBJ ALPHA<a> → OBJ STR<a>;
12.2		OBJ ALPHA<a>, OBJ STR<s> → OBJ STR<sa>;
12.3	STR WITH NO VARS	STR WITH NO VARS<,>,<`:´>,<`<´>,<`>´>;
12.4		OBJ ALPHA<a> → STR WITH NO VARS<a>;
12.5		PRED ALPHA<a> → STR WITH NO VARS<a>;
12.6		STR WITH NO VARS<s>,<t> → STR WITH NO VARS<st>;
12.7	VAR:DIFF VAR	VAR CHAR:DIFF VAR CHAR<a:b>,<a:c>, ...,<x:y>,<$_0$:$_1$>,<$_0$:$_2$>, ..., <':z>;
12.8		VAR ALPHA<scx>,<sdy>, VAR CHAR:DIFF VAR CHAR<c:d> → VAR:DIFF VAR<scx:sdy>;
13.1	SUBST	VAR ALPHA<v>, OBJ STR<s> → SUBST<v:s:v:s>;
13.2		VAR ALPHA<v>, OBJ STR<s>, STR WITH NO VARS<t> → SUBST<v:s:t:t>;
13.3		VAR ALPHA<v>, OBJ STR<s>, VAR:DIFF VAR<v:w> → SUBST<v:s:w:w>;
13.4		SUBST<v:s:x:y>,<v:s:x':y'> → SUBST<v:s:xx':yy'>;
14.1	PREMS:DERIV CONT PREMS	WF ATOM FORM<p> → PREMS:DERIV CONT PREMS<p:p>;
14.2		WF ATOM FORM<p>, PREMS:DERIV CONT PREMS<s:t> → PREMS:DERIV CONT PREMS<s:t p>, <s p:t p>,

(e) Productions defining the rules for converting an abbreviated canonical system into unabbreviated form. (The following mnemonics are used here: P = Production, AP = Atomic Production, CS = Canonical System.)

15.1	ABR1 CS:CS	WF PROD<p→c;> → ABR1 P:P<p→c;:p→c;>;
15.2		WF PROD<p→c;>, ABR1 P:P<p→s;:t;> → ABR1 P:P<p→c,s;:p→c;t;>;
15.3		WF ATOM PROD<c;> → ABR1 AP:AP<c;:c;>;
15.4		WF ATOM PROD<c;>, ABR1 AP:AP<s;:t;> → ABR1 AP:AP<s,c;:t;c;>;
15.5		ABR1 CS:CS<A:A>;
15.6		ABR1 CS:CS<c:d>, ABR1 P:P<p:q> → ABR1 CS:CS<cp:dq>;
15.7		ABR1 CS:CS<c:d>, ABR1 AP:AP<p:q> → ABR1 CS:CS<cp:dq>;
16.1	ABR2 CS:CS	CS DELIMITER<,>;
16.2		CS DELIMITER<→>;
16.3		CS DELIMITER<;>;
16.4		WF ATOM PROD<p<t>> → ABR2 STR:STR<p<t>:p<t>>;
16.5		WF ATOM PROD<p<t>>, ABR2 STR:STR<p<r>:s> → ABR2 STR:STR<p<r>,<t>:s,p<t>>;
16.6		ABR2 CS:CS<A:A>;
16.7		ABR2 CS:CS<c:d>, ABR2 STR:STR<s:t>, CS DELIMITER<m> → ABR2 CS:CS<csm:dtm>;
17.1	ABR3 CS:CS	WF ATOM PROD<p>,<c> → ABR3 P:P<p→c;:p→c;>;
17.2		WF ATOM PROD<p>, ABR3 P:P<s→c;t> → ABR3 P:P<p|s→c;p→c;t>;
17.3		ABR3 CS:CS<A:A>;
17.4		ABR3 CS:CS<c:d>, ABR3 P:P<p:q> → ABR3 CS:CS<cp:dq>;
17.5		ABR3 CS:CS<c:d>, WF PROD<p>, NOT IN<|:p> → ABR3 CS:CS<cp:dq>;
18.1	ABR4 CS:CS	PRED PART<p>,<q>, VAR ALPHA<a>,<b>, DIFF STR<a:b> → ABR4 P:P<p<a>→q<SEQ(a)>; : p<a>→q<a>; p<a>, q<b>→q<ab>;>;
18.2		ABR4 CS:CS<A:A>;
18.3		ABR4 CS:CS<c:d>, ABR4 P:P<p:q> → ABR4 CS:CS<cp:dq>;
18.4		ABR4 CS:CS<c:d>, WF PROD<p>, NOT CONT<SEQ(:p> → ABR4 CS:CS<cp:dq>;
19.1	ABR5 CS:CS	PRED PART<x>,<p>, WF TERM<t_x>, AUX PRED:TERMS<p_1:t_1>,<p_2:t_2>, STR<s> → ABR5 P:P<s→xp_1p_2<$t_xt_1t_2$>; : s→xp_1`:´p$p_2$<$t_xt_1$`:´At_2>;>;
19.2		PRED PART<x>,<y>, WF TERM<t_x>,<t_y>, AUX PRED:TERMS<p:t>,<q_1:r_1>, <q_2:r_2>,<p_1:t_1>,<p_2:t_2> STR<s>,<s'>, VAR ALPHA<v>, NOT CONT <v:ss'$t_xt_yr_1r_2t_1t_2$> → ABR5 P:P<s yq_1q_2<$t_yr_1r_2$>s'→xp_1pp_2<t_xt_1tt_2>; :s yq_1pq_2<t_xr_1`:´vr_2>s'→xp_1pp_2<$t_xt_1$tvt_2>;>;
19.3		ABR5 P:P<p:q>,<q:r> → ABR3 P:P<p:r>;
19.4		ABR5 CS:CS<A:A>;
19.5		ABR5 CS:CS<c:d>, WF PROD<p>, PREDICATES MATCH<p> →ABR5 CS:CS<cp:dp>;
19.6		ABR5 CS:CS<c:d>, WF PROD<p>, ABR5 P:P<p:q>, PREDICATES MATCH<q> → ABR5 CS:CS<cp:dq>;
20.1	ABR6 CS:CS	PRED PART<p>, WF TERM<s>,<t> → AP SYN:AP TR:COMB<p<s>:p<t>:p<t>>;
20.2		AP SYN:AP TR:COMB<p<s>:q<t>:r<u>>, AUX PRED:TERMS<d:m> → AP SYN:AP TR:COMB<pd<sm>:qd<tm>:rd<um>>;
20.3		AP SYN:AP TR:COMB<p<s>:q<t>:r<u>>, AUX PRED:TERMS<d:m>, NOT CONT<d:p> → AP SYN:AP TR:COMB<pd<sm>:q<t>:rd<um>>;
20.4		AP SYN:AP TR:COMB<p<s>:q<t>:r<u>>, AUX PRED:TERMS<d:m>, NOT CONT<d:p> → AP SYN:AP TR:COMB<p<s>:qd<tm>:rd<um>>;
20.5		ABR6 CS:CS<A:A>;
20.6		ABR6 CS:CS<c:d>, WF ATOM PROD<p>, CS DELIMITER<m> → ABR6 CS:CS<cpm:dpm>;
20.7		ABR6 CS:CS<c:d>, AP SYN:AP TR:COMB<s:t:b>, CS DELIMITER<m> → ABR6 CS:CS<cs//tm:dbm>;
21.1	ABR7 CS:CS	PRED PART<p>,<q>,<r>, VAR ALPHA<u>,<v>,<w>,<u'>,<v'>,<w'>, OBJ ALPHA<s> DIFF STR<v:u>,<w:v>,<w:u'>,<w:v'>,<w:w'>,<w':u>,<w':v>, <w':u'>,<w':v'> → ABR7 P:P<p<u> → q<ALTSEQ(u s)>; : p<u>→q<u>;p<u>, q<v> → q<vsu>;>,<p<u>,r<w>→q<ALTSEQ(u w)>; : p<u>→q<u>;p<u>, r<w>, q<v>→q<vwu>;>,<p<u..u'>→q<ALTSEQ(u s)..ALTSEQ(u' s)>; :<p<u..u'> →q<u..u'>;p<u..u'>,q<v..v'>→q<vsu..v'su'>;>,<p<u..u'>→q<ALTSEQ(u s) ..APPLIC(u' s)>; : p<u..u'>→q<u..u'>;p<u..u'>,q<v..v'> →q<vsu..((sr')u')>;>,<p<u..u'>,r<w..w'>→q<ALTSEQ(u w)..APPLIC (u' w')>; : p<u..u'>→q<u..u'>;p<u..u'>,r<w..w'>,q<v..v'> →q<vwu..((w' v')u')>;>;
21.2		ABR7 CS:CS<A:A>;
21.3		ABR7 CS:CS<c:d>, ABR7 P:P<p:q> → ABR7 CS:CS<cp:dq>;
21.4		ABR7 CS:CS<c:d>, WF PROD<p>, NOT CONT<ALTSEQ:p> → ABR7 CS:CS<cp:dp>;
22.1	ABR8 CS:CS	ABR8 P:Pl:REST<A:A:A>;
22.2		ABR8 P:Pl:REST<p:q:r>, STR<s>, NOT CONT</:s> → ABR8 P:Pl:REST<ps:qs:rs>;
22.3		ABR8 P:Pl:REST<p:q:r>, IDSTR<i>, STR<i/s>, NOT CONT<□:s> → ABR8 P:Pl:REST<pi/s:qi:rs>;
22.4		ABR8 P:Pl:REST<p:q:r> → ABR8 P:PS:REST<p:q:r>;
22.5		ABR8 P:PS:REST<p:q:r>, CONT</:r>, ABR8 P:PS:REST<r:q':r'> → ABR8 P:PS:REST<p:qq':r'>;
22.6		ABR8 P:PS:REST<p:q:r>, NOT CONT</:r> → ABR8 P:P<p:qr>
22.7		ABR8 CS:CS<A:A>;
22.8		ABR8 CS:CS<c:d>, WF PROD<p>, NOT CONT</:p> → ABR8 CS:CS<cp:dp>;
22.9		ABR8 CS:CS<c:d>, WF PROD<p>, CONT</:p>, ABR7 P:P<p:q> → ABR8 CS:CS<cp:dq>;
23.	CANONICAL SYSTEM STR	ABR CANONICAL SYSTEM STR<a>, ABR6 CS:CS<a:b>, ABR8 CS:CS<b:c>, ABR7 CS:CS<c:d>, ABR4 CS:CS<d:e>, ABR3 CS:CR<e:f>, ABR5 CS:CS<f:g>, ABR2 CS:CS<g:h>, ABR1 CS:CS<h:i> → CANONICAL SYSTEM STR<i>;
24.1	CHAR	CHAR<A>,<B>, ...,<Z>,<`a´>,<`b´>, ...,<`z´>,<0>,<1>, ...,<9>,<,>,<→>, ...,<→>;
24.2	STR	STR<A>;
24.3		STR<s>, CHAR<c> → STR<sc>;
24.4	DIFF STR	DIFF CHAR<A:B>,<A:C>, ... <→:/>;
24.5		STR<axs>,<ayt>, DIFF CHAR x:y> → DIFF STR<axs>,<ayt>;
24.6	CONT	CHAR<c>, STR<sct> → CONT<c:sct>;
24.7	NOT CONT	STR<s> → NOT CONT<s:A>;
24.8		NOT CONT<sx:t>, DIFF CHAR<x:y> → NOT CONT<sx:ty>;
24.9		NOT CONT<sxa:ty>, DIFF CHAR<x:y> → NOT CONT<sxa:tya>;
24.10	CS DELIMITER	CS DELIMITER<,>,<→>,<;>;
24.11	AUX PRED	AUX PRED:TERMS<A:A>;
24.12	:TERMS	AUX PRED:TERMS<p:t>, PRED PART<q>, WF TERM<r> → AUX PRED:TERMS<p`:´q:t`:´r>;
24.13		PREDICATES MATCH<A>
24.14		PREDICATES MATCH<c>, WF ATOM PROD<p<t>>, CS DELIMITER<m>, CS PREDICATES<q>, CONT<p,:q> → PREDICATES MATCH<cp<t>m>;

Appendix 1.4 DERIVATION OF A LEGAL PROGRAM AND ITS TRANSLATION INTO ASSEMBLER LANGUAGE

```
Rule 1:  DERIVATION<A>;
Rule 2:  DERIVATION<d>, PROD INSTANCE<c;>, WF CONCLUSION<e>                 → DERIVATION<d e>;
Rule 3:  DERIVATION<d>, PROD INSTANCE<p→c;>, PREMS:DERIV CONT PREMS<p:d>   → DERIVATION<d e>;
```

(a) Derivation of a syntactically legal program

	Premises	Production from App. 1.1a	Conclusion added to derivation
c_1		1.1	DIGIT<1>
c_2		2.1	VAR<A>
c_3	c_2	3.1	PRIMARY:VARS<1:A>
c_4	c_3	3.3	ARITH EXP:VARS<1:A>
c_5	c_1,c_4	3.5	STM:VARS<A:=1 : A,>
c_6		4.1	TYPE LIST<A>
c_7	c_6	4.4	DEC:DEC VARS<INTEGER A:A,>
c_8		6.1	IN<A,:A,>
c_9	c_5,c_7,c_8	5.	PROGRAM<BEGIN INTEGER A; A:=1 END>

(b) Derivation of a syntactically legal program and its translation into assembler language.

	Premises	Production from App. 2.1a	Conclusion added to derivation
c_1		1.1	DIGIT<1>
c_2		2.1	VAR<A>
c_3	c_2	3.1	PRIMARY:VARS<1..=F'1':A>
c_4	c_3	3.3	ARITH EXP:VARS<1.. L 1,=F'1' *LOAD 1:A>
c_5	c_1,c_4	3.5	STM:VARS<A:=1.. L 1,=F'1' *LOAD 1 ST 1,A *STORE RESULT IN A:A,>
c_6		4.1	TYPE LIST<A..A DS F>
c_7	c_6	4.4	DEC:DEC VARS<INTEGER A..A DS F:A,>
c_8		6.1	IN<A,:A,>
c_9	c_5,c_7,c_8	5.	PROGRAM<BEGIN INTEGER A; A:=1 END.. *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>

Appendix 2.1 CANONICAL SYSTEM SPECIFYING THE TRANSLATION OF THE ALGOL/60 SUBSET INTO THE TARGET LANGUAGE

3.1	PRIMARY	DIGIT<d> → PRIMARY<d..'d'>;
3.2		VAR<v> → PRIMARY<v..v>;
3.3	ARITH EXP	PRIMARY<p..p'> → ARITH EXP<p..p'>;
3.4		PRIMARY<p..p'>, ARITH EXP<a..a'> → ARITH EXP<a+p:(+(a',p'))>;
3.5	STM	ARITH EXP<a..a'>, VAR<v> → STM<v:=a..(v ASSIGN. a')>;
4.1	TYPE LIST	TYPE LIST<A..'A'>,<B..'A'>,<A,B..'A','A'>;
4.2		TYPE LIST<ℓ..ℓ'> → DEC<INTEGER ℓ..ℓ=ℓ'>;
5.	PROGRAM	STM<s..s'>, DEC<d..d'> → PROGRAM<BEGIN d;s END..LET d' IN s'>;

Appendix 2.2 DEFINITION OF PRIMITIVE FUNCTIONS FOR SUBSET

Set definitions for string variables | r,s ε STR |

```
CHAR  DIGIT<0>,<1>, ... ,<9>;
      LETTER<A>,<B>, ... ,<Z>;
      MARK<,>,<°>, ... ,<⊔>;
      DIGIT<p> | LETTER<p> | MARK<p> → CHAR<p>;
STR   STR<Λ>;
      STR<s>, CHAR<c> → STR<sc>;
```

Definition of primitive functions

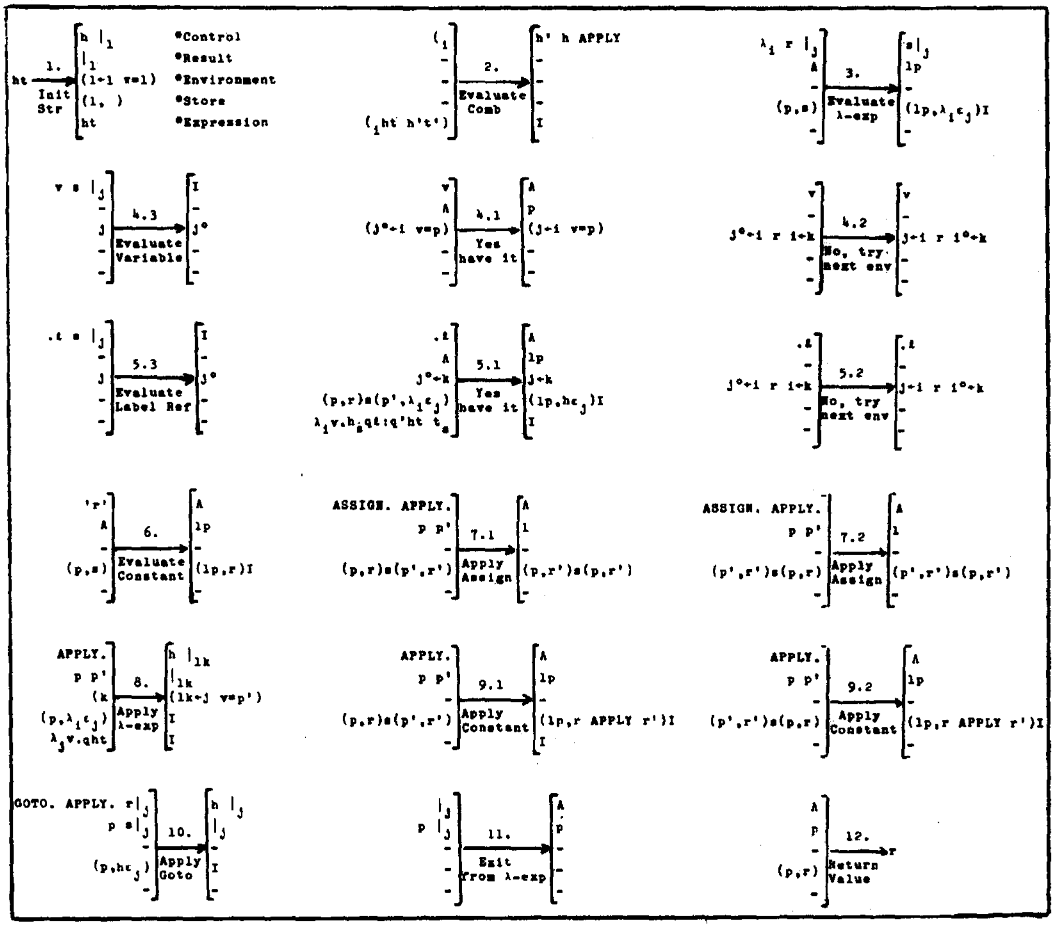
```
CAT α  =          [  s.    →·   "[Λ→·"s"]" ]  α

EQ(α,β)  =        [  α/α.  →·   TRUE       ]  α/β
                  [  α/β.  →·   FALSE      ]

COND(π,α,β)  =    [ TRUE   →·   α          ]  π
                  [ FALSE  →·   β          ]

                  [ /s0/r. →·   s1r        ]
                  [ /s1/r. →·   s2r        ]
SUCC α  =         [          :             ]  /α/
                  [ /s8/r. →·   s9r        ]
                  [ /s9/r. →    /s/0r      ]
                  [ //r.   →·   1r         ]

                  [ /0/.   →·   0          ]
                  [ /1/9r. →·   9r         ]
PRED α  =         [ /s0/r. →    /s/9r      ]  /α/
                  [ /s1/r. →·   s0r        ]
                  [ /s2/r. →·   s1r        ]
                  [          :             ]
                  [ /s9/r. →·   s8r        ]

REC +(X,Y)  =  EQ(Y,'0') ⇒ X
               ELSE      ⇒ SUM(SUCC X, PRED Y)
```

Appendix 2.3 DEFINITION OF AN EVALUATOR FOR THE TARGET LANGUAGE

(a) Set definitions for string variables: | r,r',s,s',x_1,y_1,...,x_5,y_5 ε STR | ℓ,v, VARIABLE | p,p' ε PTR | i,j,k ε INDEX | h,h' ε EXP ED | t,t' ε EXP TL | h_s ε SEQ ED | t_s ε SEQ TL | q,q' ε LABEL STR |

DIGIT LETTER MARK CHAR	DIGIT<0>,<1>, ... ,<9>; LETTER<A>,<B>, ... ,<Z>,<α>,<β>,<π>; MARK<$>,<+>, ... ,<]>; DIGIT<p> | LETTER<p> | MARK<p> → CHAR<p>;
STR	CHAR<c> → STR CHAR<c>,<λ>,<.>,<(>,<)>,<[illegible]>,<`:´>; STR<Λ>; STR<s>, STR CHAR<c> → STR CHAR<sc>;
CONSTANT VARIABLE	STR<s> → CONSTANT<'s'>; CHAR<c> → VARIABLE<SEQ(c)>;
PTR INDEX	PTR<1>; PTR<p> → PTR<1p>; DIGIT<d> → INDEX<SEQ(d)>;
LABEL STR	LABEL STR<Λ>; LABEL STR<s>, VARIABLE<ℓ> → LABEL STR<sℓ[illegible]>;
EXP	CONSTANT<p> | VARIABLE<p> → EXP<p>; EXP<e>,<f>, INDEX<i> → EXP<($_i$e f)>; VARIABLE<v>, EXP<e>, INDEX<i> → EXP<λ_iv.e>; VARIABLE<v>, EXP<e>, INDEX<i> → EXP<($_i$v ASSIGN. e)>; SEQ<s> → EXP<s>, EXP<e>, INDEX<i> → EXP<($_i$GOTO. e)>;
SEQ	INDEX<i>,<j>,<k> → T<λ_iα.λ_jβ.($_k$β α)>; EXP<e>, T<t>, INDEX<i>,<j>,<k> → SEQ<($_i$($_j$t e) λ_kπ.π)>; SEQ<s>, VARIABLE<ℓ> → SEQ<fℓh>; SEQ<s>, T<t>, EXP<e>, INDEX<i>,<j>,<k> → SEQ<($_i$($_j$t e) λ_kπ.s)>;
EXP ED EXP TL SEQ ED, SEQ TL	CONSTANT<c>, VARIABLE<v>, INDEX<i> → EXP ED<c>,<v>,<($_i$>,<λ_i>, <v ASSIGN.>,<GOTO.>; EXP<ht>, EXP ED<h> → EXP TL<t>; VARIABLE<ℓ>, EXP<e>, SEQ<h_sfℓh_t> → SEQ ED<h_s>, SEQ TL<h_t>;

(b) Substitution rules



Appendix 3.1 **CANONICAL SYSTEM SPECIFYING THE SYNTAX OF SNOBOL/1**

1.1	DIGIT	DIGIT<0>,<1>, ... ,<9>;
1.2	LETTER	LETTER<A>,<B>, ... ,<Z>;
1.3	MARK	MARK<$>,<.>,<=>, ... ,</>;
1.4	BASIC SYMBOL	DIGIT<p> | LETTER<p> | MARK<p> → BASIC SYMBOL<p>;
2.	STRING	BASIC SYMBOL<b> → STRING<SEQ(b)>;
3.1	NAME	DIGIT<p> | LETTER<p> → NAME<p>;
3.2		NAME<m>,<n> → NAME<mn>,<m.n>;
3.3	STR NAME	NAME<n> → STR NAME:STR REFS<n:n,>,<$n:n,>;
3.4	VAR NAME	NAME<n> → VAR NAME:VAR REFS<n:n,>;
3.5	BACK REF NAME	NAME<n> → BACK REF NAME:BACK REFS<n:n,>;
4.1	DIGIT STR	DIGIT<d> → DIGIT STR<SEQ(d)>;
4.2	INT	DIGIT STR<s> → INT<s>,<-s>;
4.3	ARITH EXP	INT<i> → ARITH OPERAND<"i">;
4.4		STR NAME<n> → ARITH OPERAND<n>,<$n>
4.5		ARITH OPERAND<a>,<b> → ARITH EXP<a+b>,<a-b>,<a*b>,<a/b>
5.1	STRING EXP	STRING EXP<A>;
5.2		STRING<s> → STRING EXP<"s">;
5.3		STR NAME<n> → STRING EXP<n>;
5.4		ARITH EXP<a> → STRING EXP<a>;
5.5		STRING EXP<s>,<t> → STRING EXP<s@t>;
6.1	PATTERN	STRING<s> → PAT EXP<"s">;
6.2		STR NAME<n> → PAT EXP<n>;
6.3		VAR NAME<n> → PAT EXP<*n*>;
6.4		VAR NAME<n> → PAT EXP<*(n)*>
6.5		VAR NAME<n>, DIGIT STR<d> → PAT EXP<*n/d*>;
6.6		BACK REF NAME<n> → PAT EXP<n>;
6.7		PAT EXP<p>,<q> → PAT EXP<p@q>;
6.8		PAT EXP:STR REFS:VAR REFS:BACK REFS<p:r_s:r_v:r_b>, DIFF NAME LIST<r_v>, L1:L2:INTERSEC<r_b:r_v:r_b>,<r_s:r_v:A> → PATTERN:STR REFS:VAR REFS<p:$r_s$$r_v$:$r_v$>;
7.	ASSIGN RULE	STR NAME<n>, STRING EXP<s> → ASSIGN RULE<n=s>;
8.	PAT MATCH RULE	STR NAME<n>, STRING EXP<s>, PATTERN<p> → PAT MATCH RULE<n@p=s>;
9.	INPUT RULE	PATTERN<p> → INPUT RULE<SYS .READ p>;
10.	OUTPUT RULE	STRING EXP<s> → OUTPUT RULE<SYS .PRINT s>;
11.1	RULE	ASSIGN RULE<r> | PAT MATCH RULE<r> | INPUT RULE<r> | OUTPUT RULE<r> → UNLABELED RULE<r>;
11.2		UNLABELED RULE<r> → RULE<@r>;
11.3		UNLABELED RULE<r>, NAME<n> → RULE:LABELS<n@r:n,>;
12.1	LABEL EXP	NAME<n> → LABEL EXP:LABEL REFS<n:n,>;
12.2		STR NAME<n> → LABEL EXP<$n>
12.3	STM	RULE<r>, LABEL EXP<ℓ>,<m> → STM<r>,<r/(ℓ)>,<r/S(ℓ)>,<r/S(ℓ)F(m)>,<r/F(m)>,<r/F(m)S(ℓ)>;
13.1	STM SEQ	STM<s> → STM SEQ<s>;
13.2		STM SEQ<q>, STM<s> → STM SEQ<qλs>;
13.3		STM SEQ<q>, STRING<s> → STM SEQ<qλ*s>,<*sλq>;
14.	SNOBOL PROGRAM	STM SEQ:LABELS:LABEL REFS<q:ℓ:ℓ_r>, NAME<n>, DIFF NAME LIST<END,ℓ>; L1:L2:INTERSEC<END,ℓ:n,ℓ_r:END,ℓ> → SNOBOL PROGRAM<qλEND n>;
15.1	NAME LIST	NAME LIST<A>;
15.2		NAME LIST<ℓ>, NAME<n> → NAME LIST<n,ℓ>;
16.1	DIFF CHAR	DIFF CHAR<A:B>,<A:C>, ... ,<*:=>;
16.2	DIFF STR	DIFF CHAR<x:y>, CHAR STR<axs$_1$>,<ays$_2$> → DIFF STR<axs$_1$:ays$_2$>;
16.3	DIFF NAME	NAME<n>,<m>, DIFF STR<n:m> → DIFF NAME<n:m>;
17.1	IN	NAME<n> → IN<n:n,>;
17.2		IN<n:ℓ>, NAME<m> → IN<n:m,ℓ>,<n:ℓm,>;
17.3	NOT IN	NAME<n> → NOT IN<n:A>;
17.4		NOT IN<n:ℓ>, DIFF NAME<n:m> → NOT IN<n:m,ℓ>;
18.1	NOT CONT	CHAR<c> → NOT CONT<c:A>;
18.2		NOT CONT<c:s>, DIFF CHAR<c:d> → NOT CONT<c:sd>;
19.1	DIFF NAME LIST	DIFF NAME LIST<A>;
19.2		DIFF NAME LIST<ℓ>, NAME<n>, NOT IN<n:ℓ> → DIFF NAME LIST<n,ℓ>;
20.1	L1:L2:INTERSEC	NAME LIST<ℓ> → L1:L2:INTERSEC<A:ℓ:A>;
20.2		L1:L2:INTERSEC<ℓ_1:ℓ_2:i>, NAME<n>, IN<n:ℓ_2> → L1:L2:INTERSEC<n,ℓ_1:ℓ_2:n,i>;
20.3		L1:L2:INTERSEC<ℓ_1:ℓ_2:i>, NAME<n>, NOT IN<n:ℓ_2> → L1:L2:INTERSEC<n,ℓ_1:ℓ_2:i>;

Appendix 3.2 <u>CANONICAL SYSTEM SPECIFYING THE TRANSLATION OF SNOBOL/1 INTO THE TARGET LANGUAGE</u>

3.3	STR NAME	NAME<n> → STR NAME<n..n>,<$n..(LOOKUP. n)>;
4.3	ARITH EXP	INT<i> → ARITH OPERAND<"i"..'i'>;
4.4		STR NAME<n..n'> → ARITH OPERAND<n..n'>;
4.5		ARITH OPERAND<a..a'>,<b..b'> → ARITH EXP<a+b..(+(a',b'))>, <a-b..(-(a',b'))>,<a*b..(*(a',b'))>,<a/b..(/(a',b'))>;
5.1	STRING EXP	STRING EXP<Λ..'Λ'>;
5.2		STRING<s> → STRING EXP<"s"..'s'>;
5.3		STR NAME<n..n'> → STRING EXP<n..n'>;
5.4		ARITH EXP<a..a'> → STRING EXP<a..a'>;
5.5		STRING EXP<s..s'>,<t..t'> → STRING EXP<s@t..((CAT s') t')>;
6.1	PATTERN	STRING<s> → PAT EXP<"s"..'s'>;
6.2		STR NAME<n..n'> → PAT EXP<n..n'>;
6.3		VAR NAME<n> → PAT EXP:SPECS<*n*..'n' : n∈STR |>;
6.4		VAR NAME<n> → PAT EXP:SPECS<*(n)*..'n' : n∈BAL STR |>;
6.5		VAR NAME<n>, DIGIT STR<d> → PAT EXP:SPECS<*n/d*..'n' : (n,d)∈FIX LN STR |>;
6.6		BACK REF NAME<n> → PAT EXP<n..'n'>;
6.7		PAT EXP<p..p'>,<q..q'> → PAT EXP<p@q..((CAT p') q')>;
6.8		PAT EXP<p..p'> → PATTERN<p..p'>;
7.	ASSIGN RULE	STR NAME<n..n'>, STR EXP<s..s'> → ASSIGN RULE<n=s..(n ASSIGN. 's')>;
8.	PAT MATCH RULE	STR NAME<n..n'>, STR EXP<s..s'>, PATTERN:SPECS:VAR REFS<p..p':c:v> → PAT MATCH RULE<n@p=s..(MATCH_AND_ASSIGN(n', p', λx.s', 'c', '(v)')>;
9.	INPUT RULE	PATTERN:SPECS:VAR REFS<p..p':c:v> → INPUT RULE<SYS .READ p..(MATCH_AND_ASSIGN(READER#, p', λx.'Λ', 'c', '(v)')>;
10.	OUTPUT RULE	STRING EXP<s..s'> → OUTPUT RULE(SYS .PRINT s..(PRINTER# ASSIGN. ((CAT PRINTER#) s'))>;
11.1	RULE	ASSIGN RULE<r..r'> | PAT MATCH RULE<r..r'> | INPUT RULE<r..r'> | OUTPUT RULE<r..r'> → UNLABELED RULE<r..r'>;
11.2		UNLABELED RULE<r..r'> → RULE<@r..r'>;
11.3		UNLABELED RULE<r..r'>, NAME<n> → RULE<n@r.. n :r'>;
12.1	LABEL EXP	NAME<n> → LABEL EXP<n.. .n>;
12.2		STR NAME<n..n'> → LABEL EXP<$n..(LOOKUP. ((CAT '.') n));
12.3	STM	RULE<r..r'>, LABEL EXP<ℓ..ℓ'>,<m..m'> → STM<r..r'>,<r/(ℓ)..r'; <r/s(ℓ)..r' ⇒ (GOTO. ℓ') ELSE ⇒ 'Λ'>,<r/s(ℓ)F(m)..r' ⇒(GOTO. ℓ') ELSE ⇒(GOTO. m')>, <r/F(m)..r' ⇒ 'Λ' ELSE ⇒ (GOTO. ℓ')>,<r/F(m)s(ℓ)..r' ⇒(GOTO. ℓ') ELSE ⇒(GOTO. m')>;
13.1	STM SEQ	STM<s..s'> → STM SEQ<s..s'>;
13.2		STM SEQ<q..q'>, STM<s..s'> → STM SEQ<q$s..q';s'>;
13.3		STM SEQ<q..q'>, STRING<s> → STM SEQ<q>*s..q'>,<*s>q..q'>;
14.	SNOBOL PROGRAM	STM SEQ:STR REFS<q..q':s_r>, NAME<n>, LIST:BVS:CORR NULL LIST<s_r:v_b:ℓ> → SNOBOL PROGRAM<q>END n..LET v_b=ℓ IN (GOTO. 'n'); q'>;
22.1	LIST:BVS:CORR NULL LIST	NAME<n> → LIST:BVS:CORR NULL LIST<n:n:'Λ'>;
22.2		LIST:BVS:CORR NULL LIST<ℓ:b:x>, NAME<n>, IN<n:ℓ> → LIST:BVS:CORR NULL LIST<ℓ,n:b:x>;
22.3		LIST:BVS:CORR NULL LIST<ℓ:b:x>, NAME<n>, NOT IN<n:ℓ> → LIST:BVS:CORR NULL LIST<ℓ,n:b:x,'Λ'>;

Appendix 3.3 DEFINITION OF PRIMITIVE FUNCTIONS FOR SNOBOL/1

Set definitions for string variables: | r,s STR | b,c BAL STR |

CHAR	DIGIT<0>,<1>, ... ,<9>; LETTER<A>,<B>, ... ,<Z>; MARK<+>,<->, ... ,<?>; DIGIT<p> | LETTER<p> | MARK<p> → CHAR<p>;
STR	STR<A>; STR<s>, CHAR<c> → STR<sc>;
BAL STR	STR<s>, NOT CONT<(:s>,<):s> → BAL STR<s>; BAL STR<s>,<t> → BAL STR<(s)>,<st>;
FIX LN STR	FIX LN STR<A:0>; FIX LN STR<s:n>, SUCC<m:n>, CHAR<c> → FIX LN STR<sc:n>; †
NOT CONT	DIFF CHAR<A:B>,<A:C>, ... ,<?:">; CHAR<c> → NOT CONT<c:A>; NOT CONT<c:s>, DIFF CHAR<c:d> → NOT CONT<c:sd>;
SUCC	STR OF NINES:ZEROS<9:0>; STR OF NINES:ZEROS<n:y> → STR OF NINES:ZEROS<n9:y0>; STR<s> → SUCC<s0:s1>,<s1:s2>, ... ,<s8:s9>; STR<s>, STR OF NINES:ZEROS<n:y> → SUCC<n:1y>,<s0n:s1y>,<s1n:s2y>, ... ,<s8n:s9y>;

(a) Miscellaneous basic primitives

```
CAT α =          [ s.            →·  "[Λ→·" s "]" ]  α

EQ(α,β) =        [ α/β.          →·  TRUE          ]  α/β
                 [ α/β.          →·  FALSE         ]

NEQ(α,β) =       [ α/β.          →·  FALSE         ]  α/β
                 [ α/β.          →·  TRUE          ]

COND(γ,α,β) =    [ TRUE          →·  α             ]  γ
                 [ FALSE         →·  β             ]

                 [ TRUE/TRUE     →·  TRUE          ]
AND(α,β) =       [ TRUE/FALSE    →·  FALSE         ]  α/β
                 [ FALSE/TRUE    →·  FALSE         ]
                 [ FALSE/FALSE   →·  FALSE         ]

HD α =           [ (b,c)         →·  b             ]  α
                 [  ()           →·  Λ             ]

TL α =           [ (b,c)         →·  c             ]  α
                 [  ()           →·  ()            ]
```

(b) Arithmetic primitives

```
ABS α        [ -s.     →·    s    ]  α
             [  s.     →·    s    ]

NEGATE α =   [ -s.     →·    s    ]  α
             [  s.     →·   -s    ]

IS_POS α =   [ -s.     →·  FALSE  ]  α
             [  s.     →·  TRUE   ]

IS_NEG α =   [ -s.     →·  TRUE   ]  α
             [  s.     →·  FALSE  ]

             [ /s0/r.  →·  s1r    ]
             [ /s1/r.  →·  s2r    ]
             [          :         ]
SUCC α =     [          :         ]  /α/
             [ /s8/r.  →·  s9r    ]
             [ /s9/r.  →   /s/0r  ]
             [  //r.   →·  1r     ]

             [  /0/.   →·    0    ]
             [ /1/9r.  →·   9r    ]
             [ /s0/r.  →   /s/9r  ]
PRED α =     [ /s1/r.  →·  s0r    ]  /α/
             [ /s2/r.  →·  s1r    ]
             [          :         ]
             [ /s9/r.  →·  s8r    ]
```

```
REC ∸(X,Y) =        EQ(Y,'0')   ⇒  X
                    ELSE        ⇒  ∸(PRED X, PRED Y)

REC SUM(X,Y) =      EQ(Y,'0')   ⇒  X
                    ELSE        ⇒  SUM(SUCC X, PRED X)

SIGN(X,Y) =         AND(IS_POS X, IS_POS Y)  ⇒  'A'
                    AND(IS_POS X, IS_NEG Y)  ⇒  '-'
                    AND(IS_NEG X, IS_POS Y)  ⇒  '-'
                    ELSE                     ⇒  'A'

LESS(X,Y) =         NEG(∸(Y,X), '0')

DIFF(X,Y) =         LESS(X,Y)   ⇒  NEGATE(∸(Y,X))
                    EQ(X,Y)     ⇒  '0'
                    ELSE        ⇒  ∸(X,Y)

REC PROD(X,Y) =     EQ(Y,'0')   ⇒  '0'
                    ELSE        ⇒  SUM(X, PROD (X, PRED X))

REC QUOT(X,Y) =     LESS(X,Y)   ⇒  '0'
                    ELSE        ⇒  SUM('1', QUOT (∸(X,Y), Y))

+(X,Y) =            AND(IS_POS X, IS_POS Y)  ⇒  SUM(X,Y)
                    AND(IS_POS X, IS_NEG Y)  ⇒  DIFF(X, ABS Y)
                    AND(IS_NEG X, IS_POS Y)  ⇒  DIFF(Y, ABS X)
                    ELSE                     ⇒  NEGATE(SUM(ABS X, ABS Y))

-(X,Y) =            +(X, NEGATE Y)

*(X,Y) =            LET S=SIGN(X,Y)  IN  CAT(S, PROD (ABS X, ABS Y))

/(X,Y) =            LET S=SIGN(X,Y)  IN  CAT(S, QUOT (ABS X, ABS Y))
```

(c) Basic pattern matching function

```
REC ASGN_LIST(L,M)  =  LET H,T  =  HD L,TL L
                       IN EQ(H,'A')  ⇒  'A'
                          ELSE       ⇒  LET v1=(LOOKUP. H) IN (v1 ASSIGN. (HD M));
                                                               ASGN_LIST(T, (TL M))
MATCH_AND_ASSIGN(NAME, PAT, STR_EXP,SET_SPECS,VARS)

        LET v =  ( ⎡| SET_SPECS                          ⎤
                   ⎢s PAT t.      →• (VARS,(s),(t),)  ⎥ NAME)
                   ⎣  s.          →•       A          ⎦

        IN EQ (v,'A')  ⇒  'FALSE'
           ELSE        ⇒  LET v1,v2,v3 = HD v, HD (TL v), HD (TL (TL v))
                          IN  ASGN_LIST(VARS,v1);
                              LET  STR_EXP = (STR_EXP 'A')
                              IN   (NAME ASSIGN. T(CAT((CAT v2) STR_EXP)) v3);
                                   'TRUE'
```

(d) Definition of LOOKUP. to be added to evaluator

```
        LOOKUP. APPLY.⎤          ⎡s
                    P ⎥   T.1    ⎢A
                    - ⎥─────────→⎢-
                (p,s) ⎥          ⎢I
                    - ⎦          ⎣-
```

Appendix 4.1 **CANONICAL SYSTEM SPECIFYING THE SYNTAX OF ALGOL/60**

1.1	DIGIT	DIGIT<0>,<1>, ... ,<9>;
1.2	LETTER	LETTER<A>,<B>, ... ,<Z>,<\`a´>,<\`b´>, ... ,<\`z´>;
1.3	MARK	MARK<+>,<–>, ... ,<[>,<]>;
1.4	CHAR	DIGIT<p> | LETTER<p> | MARK<p> → CHAR<p>;
2.1	DIG STR	DIGIT<d> → DIG STR<SEQ(d)>;
2.2	LET STR	LETTER<ℓ> → LET STR<SEQ(ℓ)>;
2.3	ID STR	LETTER<ℓ> → ID STR<ℓ>;
2.4		ID<i>, LETTER<ℓ> → ID STR<iℓ>;
2.5		ID<i>, DIGIT<d> → ID STR<id>;
2.6	STR	STR<Λ>;
2.7		CHAR<c> → STR<SEQ(c)>;
3.1	PAR DELIM	PAR DELIM<,>;
3.2		LET STR<ℓ> → PAR DELIM<)ℓ\`:´(>;
4.1	LABEL:VAL	ID STR<s> → LABEL:VAL<s:s>;
4.2		LABEL:VAL<1:1>,<2:2>, ... ,<9:9>;
4.3		LABEL:VAL<ℓ:v>, DIGIT<d> → LABEL:VAL<ℓd:vd>;
4.4		LABEL:VAL<ℓ:v>, DIGIT STR<ℓ> → LABEL:VAL<0ℓ:v>;
5.1	ADD OP	ADD OP<+>,<–>;
5.2	MULT OP	MULT OP<×>,<\`/´>,<÷>;
5.3	REL OP	REL OP<\`<´>,<≤>,<=>,<≥>,<\`>´>,<≠>;
6.1	UNSIGN INT	DIGIT STR<s> → UNSIGN INT<s>;
6.2	UNSIGN NUM	DIGIT STR<s>,<t> → UNSIGN NUM<s>,<.t>,<s.t>;
6.3	INT	UNSIGN INT<i> → INT<i>,<+i>,<–i>;
6.4	NUM	UNSIGN NUM<n> → NUM<n>,<+n>,<–n>;
7.1	ID	ID STR<i> → ID<i>;
7.2	IDLIST	IDSTR<i> → IDLIST<ALTSEQ(i ,)>;
8.1	VAR	ARITH EXP<a> → SUBSCRIPT LIST:DIMN<a:1>;
8.2		ARITH EXP<a>, SUBSCRIPT LIST:DIMN<ℓ:m> → SUBSCRIPT LIST:DIMN<ℓ,a:m1>;
8.3		ID<i> → REAL/INT/BOOL VAR:R/I/B VARS<i:i,>;
8.4		ID<i>, SUBSCRIPT LIST:DIMN<ℓ:m> → REAL/INT/BOOL VAR:R/I/B ARRAYS<i[ℓ]:i(m),>;
9.1	FCN DES	ID<i> → ACT PAR:SPECS:S VARS<i:SWITCH,:i,>;
9.2		ID<i>, SPEC LIST<x> → ACT PAR:SPECS:R/I/B/N PROCS<i:REAL/INTEGER/BOOLEAN/NONVAL PROCEDURE(ℓ),:i(x),>;
9.3		ID<i>, DIMN<m> → ACT PAR:SPECS:R/I/B ARRAYS<i:REAL/INTEGER/BOOLEAN ARRAY(m),:i(m),>;
9.4		REAL/INT/BOOL VAR<v> → ACT PAR:SPECS<v:REAL,/INTEGER,/BOOLEAN,>;
9.5		ARITH EXP<a> → ACT PAR:SPECS<a:ARITH EXP,>;
9.6		BOOL EXP<b> → ACT PAR:SPECS<b:BOOL EXP,>;
9.7		DES EXP<d> → ACT PAR:SPECS<d:LABEL,>;
9.8		ACT PAR<p>, PAR DELIM<d> → ACT PAR PART<ALTSEQ(p d)>;
9.9		ID<i> → REAL/INT/BOOL/NONVAL FCN DES:R/I/B IN PROCS<i:i(),>;
9.10		ID<i>, ACT PAR PART:SPECS<p:x,> → REAL/INT/BOOL/NONVAL FCN DES:R/I/B IN PROCS<i(p):i(x),>;
9.11		REAL FCN DES<f> | INT FCN DES<f> | BOOL FCN DES<f> | NONVAL FCN DES<f> → FCN DES<f>;
10.1	ARITH EXP	UNSIGN NUM<p> | REAL VAR<p> | INT VAR<p> | REAL FCN DES<p> | INT FCN DES<p> → PRIMARY<p>;
10.2		ARITH EXP<a> → PRIMARY<(a)>;
10.3		PRIMARY<p>, MULT OP<m> → TERM<ALTSEQ(p m)>;
10.4		TERM<t>, ADD OP<a> → TERM SEQ<ALTSEQ(t a)>;
10.5		TERM SEQ<s> → SIMPLE ARITH EXP<s>,<+s>,<–s>;
10.6		SIMPLE ARITH EXP<s> → ARITH EXP<s>;
10.7		BOOL EXP<b>, SIMPLE ARITH EXP<s>, ARITH EXP<a> → ARITH EXP<IF b THEN s ELSE a>;
11.1	BOOL EXP	BOOL PRIM<TRUE>,<FALSE>;
11.2		SIMPLE ARITH EXP<a>,<b>, REL OP<r> → RELATION<arb>;
11.3		RELATION<p> | BOOL VAR<p> | BOOL FCN DES<p> → BOOL PRIM<p>;
11.4		BOOL EXP<b> → BOOL PRIM<(b)>;
11.5		BOOL PRIM<p> → BOOL SEC<p>,< p>;
11.6		BOOL SEC<s> → BOOL FAC<ALTSEQ(s ∧)>;
11.7		BOOL FAC<f> → BOOL TERM<ALTSEQ(f ∨)>;
11.8		BOOL TERM<t> → BOOL IMP<ALTSEQ(t ⊃)>;
11.9		BOOL IMP<i> → SIMPLE BOOL<ALTSEQ(i ≡)>;
11.10		SIMPLE BOOL<s> → BOOL EXP<s>;
11.11		BOOL EXP<a>,<b>, SIMPLE BOOL<s> → BOOL EXP<IF a THEN b ELSE s>;
12.1	DES EXP	LABEL:VAL<ℓ:v> → SIMPLE DES EXP:LABEL REPS<ℓ:v,>;
12.2		ID<i>, ARITH EXP<a> → SIMPLE DES EXP:S VARS<i[a]:i,>;
12.3		DES EXP<d> → SIMPLE DES EXP<(d)>;
12.4		SIMPLE DES EXP<s> → DES EXP<s>;
12.5		BOOL EXP<b>, SIMPLE DES EXP<s>, DES EXP<d> → DES EXP<IF b THEN s ELSE d>;
13.	EXP	ARITH EXP<e> | BOOL EXP<e> | DES EXP<e> → EXP<e>;
14.	DUMMY STM	DUMMY STM<Λ>;
15.	COMMENT STM	STR<s>, NOT CONT<;:s> → COMMENT STM<COMMENT s>;
16.	GOTO STM	DES EXP<d> → GOTO STM<GO TO d>;
17.	PROC STM	FCN DES<f> → PROC STM<f>;

```
18.1   ASGT STM       IDSTR<i>                                      → R/I/B LEFT PART:ASGNED PROC IDS<i:i,>;
18.2                  REAL/INT/BOOL VAR<i>, IDSTR<i>                → R/I/B LEFT PART:ASGNED VARS<i:i,>;
18.3                  REAL/INT/BOOL VAR<i[ℓ]>                       → R/I/B LEFT PART<i[ℓ]>;
18.4                  R/I/B LEFT PART<ℓ>, ARITH/ARITH/BOOL EXP<e>   → R/I/B ASGT STM<ℓ:=e>;
18.5                  R/I/B LEFT PART<ℓ>, R/I/B ASGT STM<s>         → R/I/B ASGT STM<ℓ:=s>;
18.6                  R/I/B ASGT STM<s>  →  ASGT STM<s>;

19.1   FOR STM        ARITH EXP<a>                  → FOR LIST EL<a>;
19.2                  ARITH EXP<a>,<b>,<c>          → FOR LIST EL<a STEP b UNTIL c>;
19.3                  ARITH EXP<a>, BOOL EXP<b>→ FOR LIST EL<a WHILE b>;
19.4                  FOR LIST EL<e>                → FOR LIST<ALTSEQ(e ,)>;
19.5                  REAL INT VAR<v>, FOR LIST<t>, STM:LABELS:LABEL REFS<s:ℓ:ℓ_r>, L1:L2:REL COMP<ℓ_r:ℓ:ℓ'_r>,
                        DIFF ENTRY LIST<ℓ> → FOR STM:LABELS:LABEL REFS<FOR v:=t DO s:Λ:ℓ'_r>;

20.1   UNCOND STM     DUMMY STM<s> | COMMENT STM<s> | GOTO STM<s> | PROC STM<s> | ASGT STM<s> | FOR STM<s> |
                        BLOCK<s> | COMPOUND STM<s>  →  UNCOND STM<s>;
20.2                  UNCOND STM<s>,  LABEL:VAL<ℓ:v>  →  UNCOND STM:LABELS<ℓ`:´s:v,>;

21.1   COND STM       BOOL EXP<b>, UNCOND STM<u>            → COND STM<IF b THEN u>;
21.2                  BOOL EXP<b>, UNCOND STM<u>, STM<s>    → COND STM<IF b THEN u ELSE s>;
21.3                  COND STM<s>, LABEL:VAL<ℓ:v>           → COND STM:LABELS<ℓ`:´s:v,>;

22.1   STM            UNCOND STM<s> | COND STM<s> → STM<s>;
22.2   STM SEQ        STM<s>                      → STM SEQ<s>;
22.3                  STM<s>,  STM SEQ<q>         → STM SEQ<q;s>;

23.    COMPOUND STM   STM SEQ<s>, STR<c>, NOT CONT<;:c>,<END:c>,<ELSE:c> → COMPOUND STM<BEGIN s END c>;

24.1   TYPE DEC       IDLIST<ℓ> → TYPE DEC:DEC R/I/B VARS<REAL/INTEGER/BOOLEAN ℓ:ℓ,>;
24.2                  IDLIST<ℓ> → TYPE DEC:DEC R/I/B VARS<OWN REAL/INTEGER/BOOLEAN ℓ:ℓ,>;

25.1   ARRAY DEC      ARITH EXP:R VARS:I VARS:B VARS:S VARS:R ARRAYS:I ARRAYS:B ARRAYS:R PROCS:I PROCS:B PROCS:N PROCS
                        <a:v_r:v_i:v_b:v_s:a_r:a_i:a_b:p_r:p_i:p_b:p_n>,<a':v'_r:v'_i:v'_b:v'_s:a'_r:a'_i:a'_b:p'_i:p'_b:p'_n>
                          BOUND PAIR:DIM R VARS:DIM I VARS:DIM B VARS:DIM S VARS:DIM R ARRAYS:DIM I ARRAYS
                          DIM B ARRAYS:DIM R PROCS:DIM I PROCS:DIM B PROCS:DIM N PROCS
                          <a`:´b:v_rv'_r:v_iv'_i:v_bv'_b:v_s:v'_s:a_ra'_r:a_ia'_i:a_ba'_b:p_rp'_r:p_ip'_i:p_bp'_b:p_np'_n>;
25.2                  BOUND PAIR<p>                       → BPLIST:DIMN<p:1>;
25.3                  BOUND PAIR<p>, BPLIST:DIMN<ℓ:m>     → BPLIST:DIMN<ℓ,p:m1>;
25.4                  BPLIST:DIMN<ℓ:m>, IDSTR<i>          → ARRAY:ARRAY VARS<i[ℓ]:i(m),>;
25.5                  ARRAY<i[ℓ]>                         → ARRAY SEG<i[ℓ]>;
25.6                  ARRAY<i[ℓ]>, ARRAY SEG<s[ℓ]>        → ARRAY SEG<i,s[ℓ]>;
25.7                  ARRAY SEG<s>                        → ARRAY LIST<s>;
25.8                  ARRAY SEG<s>, ARRAY LIST<ℓ>         → ARRAY LIST<ℓ,s>;
25.9                  ARRAY LIST:ARRAY VARS<ℓ:v> → ARRAY DEC:DEC R/I/B/A ARRAYS<REAL/INTEGER/BOOLEAN/A ARRAY ℓ:v>;
25.10                 ARRAY LIST:ARRAY VARS<ℓ:v> → ARRAY DEC:DEC R/I/B/A ARRAYS<OWN REAL/INTEGER/BOOLEAN/A ARRAY ℓ:v>;

26.1   SW DEC         DES EXP<d>              → SW LIST<ALTSEQ(d ,)>;
26.2                  IDSTR<i>, SW LIST<ℓ> → SW DEC:DEC S VARS<SWITCH i:=ℓ:i,>;

27.1   FORMAL PAR     IDSTR<i>                       → FORMAL PAR:PARS<i:i,>;
27.2      PART        FORMAL PAR<p>, PAR DELIM<d> → FORMAL PAR LIST<ALTSEQ(p d)>;
27.3                  FORMAL PAR PART<Λ>;
27.4                  FORMAL PAR LIST<ℓ> → FORMAL PAR PART<(ℓ)>;
27.5   VALUE PART     VALUE PART<Λ>;
27.6                  IDLIST<ℓ> → VALUE PART:PARS<VALUE ℓ; : ℓ>;
27.7   SPECIFIER      TYPE<REAL>,<INTEGER>,<BOOLEAN>;
27.8      PART        TYPE<t> → SPECIFIER<LABEL>,<SWITCH>,<t>,<ARRAY>,<t ARRAY>,<PROCEDURE>,<t PROCEDURE>;
27.9                  IDLIST<ℓ>, SPECIFIER<s> → SPECIFIER LIST:PARS<sℓ; : ℓ(s),>;
27.10                 SPECIFIER PART<Λ>;
27.11                 SPECIFIER LIST<ℓ> → SPECIFIER PART<SEQ(ℓ)>;
27.12  PROC DEC       IDSTR<i>, FORMAL PAR PART:PARS<f:f_p>, VALUE PART:PARS<u:u_p>,
                        SPECIFIER PART:PARS<e:e_p>, STM:R VARS:I VARS:B VARS:S VARS:R
                        ARRAYS:I ARRAYS:B ARRAYS:R PROCS:I PROCS:B PROCS:N PROCS:LABELS
                        :LABEL REFS: ASGNED VARS:ASGNED PROC IDS<s:v_r:v_i:v_b:v_s:a_r:a_i:a_b:p_r:p_i:p_b:p_n:ℓ:ℓ_r:i_as:i_pas>,
                        L1:L2:INTERSEC:REL COMP<v_r:f_p:v_rf:v'_r>,<v_i:f_p:v_if:v'_i>,<v_b:f_p:v_bf:v'_b>,<v_s:f_p:v_sf:v'_s>,
                                            <a_r:f_p:a_rf:a'_r>,<a_i:f_p:a_if:a'_i>,<a_b:f_p:a_bf:a'_b>,
                                            <p_r:f_p:p_rf:p'_r>,<p_i:f_p:p_if:p'_i>,<p_b:f_p:p_bf:p'_b>,<p_n:f_p:p_nf:p'_n>,
                                            <ℓ_r:f_p:ℓ_rf:ℓ'_r>,<ℓ'_r:ℓ_s:x:ℓ''_r>,<i_as:f_p:i_asf:i'_as>,<i_pas:i,:x:i'_pas>
                        DIFF ENTRY LIST<f_p>,<u_p>,<ℓ>,
                        DISJ ENTRY LISTS<(v_rf)(v_if)(v_bf)(v_sf)(a_rf)(a_if)(a_bf)(p_rf)(p_if)(p_bf)(p_nf)(ℓ_rf)(i,i_pas)>,
                        PARS:USES:SPECS<f_p:e_p v_rf(REAL)v_if(INTEGER)v_bf(BOOLEAN)v_sf(SWITCH)a_rf(REAL ARRAY)a_if(INTEGER ARRAY)
                                        a_bf(BOOLEAN ARRAY)p_rs(REAL PROCEDURE)p_if(INTEGER PROCEDURE)p_bf(BOOLEAN PROCEDURE)
                                        p_nf(NONVAL PROCEDURE) _rf(LABEL)u_p(VALUE)i_asf(ASGNED) :x>
                        → PROC DEC:DEC R/I/B/N PROCS:R VARS:I VARS:B VARS:S VARS:R ARRAYS:I ARRAYS:B ARRAYS:P PROCS
                          :I PROCS:B PROCS:N PROCS:LABELS:LABEL REFS:ASGNED VARS:ASGNED PROC IDS
                          <REAL/INTEGER/BOOLEAN/A PROCEDURE if;ues i(x):v'_r:v'_i:v'_b:v'_s:a'_r:a'_i:a'_b:
                          p'_r:p'_i:p'_b:p'_n:Λ:ℓ''_r:i'_as:i'_pas>;

28.1   DEC            TYPE DEC<d> | ARRAY DEC<d> | SW DEC<d> | PROC DEC<d> → DEC<d>;
28.2   DEC SEQ        DEC<d>                → DEC SEQ<d>;
28.3                  DEC<d>,  DEC SEQ<s>  →  DEC SEQ<s;d>;


29.    BLOCK          STM SEQ:R VARS:I VARS:B VARS:S VARS:R ARRAYS:I ARRAYS:B ARRAYS:R PROCS
                       :I PROCS:B PROCS:N PROCS:LABELS:LABEL REFS<s:v_r:v_i:v_b:v_s:a_r:a_i:a_b:p_r:p_i:p_b:p_n:ℓ:ℓ_r>,
                      DEC SEQ:R VARS:I VARS:B VARS:S VARS:R ARRAYS:I ARRAYS:B ARRAYS:R PROCS
                       :I PROCS:B PROCS:N PROCS:DEC R VARS:DEC I VARS:DEC B VARS:DEC S VARS
                       :DEC R ARRAYS:DEC I ARRAYS:DEC B ARRAYS:DEC R PROCS:DEC I PROCS:DEC B PROCS
                       :DEC N PROCS:DIM R VARS:DIM I VARS:DIM B VARS:DIM S VARS:DIM R ARRAYS:DIM I ARRAYS
                       :DIM B ARRAYS:DIM R PROCS:DIM I PROCS:DIM B PROCS:DIM N PROCS:LABEL REFS
                       <d:v'_r:v'_i:v'_b:v'_s:a'_r:a'_i:a'_b:p'_r:p'_i:p'_b:p'_n:v_rd:v_id:v_bd:v_sd:a_rd:a_id:a_bd:p_rd:p_id:p_bd:p_nd
                       :v_rm:v_im:v_bm:v_sm:a_rm:a_im:a_bm:p_rm:p_im:p_bm:p_nm:ℓ'_r>,
                      STR<c>,  NOT CONT<;:c>,<END:c>,<ELSE:c>,
```

		L1:L2:REL COMP$<v_rv'_r:v_{rd}:v''_r>,<v_iv'_i:v_{id}:v''_i>,<v_bv'_b:v_{bd}:v''_b>,<v_sv'_s:v_{sd}:v''_s>,$ $<a_ra'_r:a_{rd}:a''_r>,<a_ia'_i:a_{id}:a''_i>,<a_ba'_b:a_{bd}:a''_b>,$ $<p_rp'_r:p_{rd}:p''_r>,<p_ip'_i:p_{id}:p''_i>,<p_bp'_b:p_{bd}:p''_b>,<p_np'_n:p_{nd}:p''_n>,$ $<\ell_r\ell'_r:\ell:\ell''_r>,$ DIFF ENTRY LIST$<v_{rd}v_{id}v_{bd}v_{sd}a_{rd}a_{id}a_{bd}p_{rd}p_{id}p_{bd}p_{nd}\ell>,$ DISJ ENTRY LISTS$<(v_r)(v_i)(v_b)(v_s)(a_r)(a_i)(a_b)(p_r)(p_i)(p_b)(p_n)(\ell)(\ell''_r)>,$ $<(v_{rm})(v_{rd})(v_{im})(v_{id})(v_{sm})(v_{sd})(a_{rm})(a_{rd})(a_{im})(a_{id})(a_{bm})(a_{bd})$ $(p_{rm})(p_{rd})(p_{im})(p_{id})(p_{bm})(p_{bd})(p_{nm})(p_{nd})>$ →BLOCK:R VARS:I VARS:B VARS:S VARS:R ARRAYS:I ARRAYS:B ARRAYS:R PROCS :I PROCS:B PROCS:N PROCS:LABELS:LABEL REFS <BEGIN d;s END c:$v''_rv_{rm}:v''_iv_{im}:v''_bv_{bm}:v''_sv_{sm}:a''_ra_{rm}:a''_ia_{im}:a''_ba_{bm}$ $:p''_rp_{rm}:p''_ip_{im}:p''_bp_{bm}:p''_np_{nm}:\Lambda:\ell''_r>;$
30.1	ALGOL PROGRAM	BLOCK<p> COMPOUND STM<p> → PROGRAM STR<p>;
30.2		PROGRAM STR<s>, LABEL:VAL<ℓ:v> → PROGRAM STR<ℓ':'s>;
30.3		PROGRAM STR:R VARS:I VARS:B VARS:S VARS:R ARRAYS:I ARRAYS:B ARRAYS:R PROCS :I PROCS:B PROCS:N PROCS:LABELS:LABEL REFS:ASGNED PROC IDS <s:Λ:Λ:Λ:Λ:Λ:Λ:Λ:Λ:Λ:Λ:Λ:Λ:Λ:Λ> → ALGOL PROGRAM<s>;
31.1	TYPE	TYPE<REAL>,<INTEGER>,<BOOLEAN>;
31.2	DIMM	DIMM<1>;
31.3		DIMM<m> → DIMM<m1>;
32.1	SPEC	SPEC<Λ>,<LABEL>,<SWITCH>,<ARITH EXP>,<BOOL EXP>,<ASGNED>,<VALUE>;
32.2		TYPE<t> → SPEC<t>,<VALUE t>,<ASGNED t>,<ASGNED VALUE t>;
32.3		TYPE<t>, DIMM<m> → SPEC<ARRAY>,<t ARRAY>,<t ARRAY(m)>,<VALUE t ARRAY(m)>;
32.4		TYPE<t>, SPEC LIST<s> → SPEC<PROCEDURE>,<t PROCEDURE>,<t PROCEDURE(s)>,<NONVAL PROCEDURE(s)>;
32.5	SPEC LIST	SPEC<s> → SPEC LIST<ALTSEQ(s ,)>;
33.1	SPEC1:SPEC2 :COMB	SPEC<s> → SPEC1:SPEC2:COMB<Λ:s:s>,<s:s:s>;
33.2		TYPE<t>, DIMM<m> → SPEC1:SPEC2:COMB<ARRAY:REAL ARRAY(m):REAL ARRAY(m)>, <t ARRAY:t ARRAY(m):t ARRAY(m)>;<t:VALUE:VALUE t>,<t:ASGNED:ASGNED t>, <VALUE t:ASGNED VALUE t>,<t ARRAY(m):VALUE:VALUE t ARRAY(m)>;
33.3		TYPE<t>, SPEC LIST<s> → SPEC1:SPEC2:COMB<PROCEDURE:NONVAL PROCEDURE(s):NONVAL PROCEDURE(s)>, <t PROCEDURE:t PROCEDURE(s):t PROCEDURE(s)>;
34.1	SPEC MATCH	EXP SPEC<Λ>,<VALUE>,<ASGNED VALUE>;
34.2		SPEC1:SPEC2:COMB<s:t:c> → SPEC MATCH<s:t>;
34.3		EXP SPEC<s> → SPEC MATCH<ARITH EXP:s REAL>,<ARITH EXP:s INTEGER>, <BOOL EXP:s BOOLEAN>;
34.4	SPEC LIST MATCH	SPEC MATCH<s:t> → SPEC LIST MATCH<s:t>;
34.5		SPEC MATCH<s:t>, SPEC LIST MATCH<s':t'> → SPEC LIST MATCH<s',s:t',t>;
35.1	USES:PARS WITH SPECS	IDLIST<ℓ> → USES:PARS WITH SPECS<Λ:ℓ,>;
35.2		IDSTR<i>, SPEC1:SPEC2:COMB<s:t:c>, USES:PARS WITH SPECS<u:xis,y> → USES:PARS WITH SPECS<ui(t):xi c,y>;
35.3		IDSTR<i>, SPEC1:SPEC2:COMB<s:t:c>, USES:PARS WITH SPECS<u(t):xis,y> → USES:PARS WITH SPECS<u,i(t):xi c,y>;
35.4		ENTRY<i(p)>, SPEC1:SPEC2:COMB<s:t(p):c>, USES:PARS WITH SPECS<u:xis,y> → USES:PARS WITH SPECS<ui(p)(t):x ic,y>;
35.5		ENTRY<i(p)>, SPEC1:SPEC2:COMB<s:t(p):c>, USES:PARS WITH SPECS<u(t):xis,y> → USES:PARS WITH SPECS<u,i(p)(t):xi c,y>;
35.6	PARS:USES :SPECS	PARS:USES:SPECS<Λ:Λ:Λ>;
35.7		USES:PARS WITH SPECS<u:x> → PARS:USES:SPECS<Λ:u:x>;
35.8		IDSTR<i>, PARS:USES:SPECS<p:u:xi,y> → PARS:USES:SPECS<pi,:u:xy>;
36.1	ENTRY	ID<i>, SPEC LIST<s>, DIMM<m> → ENTRY<i>,<i(s)>,<i(m)>;
36.2	ENTRY LIST	ENTRY LIST<Λ>;
36.3		ENTRY LIST<ℓ>, ENTRY<e> → ENTRY LIST<e,ℓ>;
37.1	DIFF CHAR	DIFF CHAR<A:B>,<A:C>, ... ,<[:]>;
37.2	DIFF STR	CHAR STR OR NULL<axs>,<ayt>, DIFF CHAR<x:y> → DIFF STR<axs:ayt>;
37.3	DIFF ENTRY	ID STR<i>,<j>, DIFF STR<i:j>, SPEC LIST<s>,<t> → DIFF ENTRY<i:j>,<i(s):j>,<i:j(t)>,<i(s):j(t)>;
38.1	IN	ID STR<i>, SPEC LIST MATCH<s:t>, DIMM<m> → ENTRY MATCH<i:i>,<i(s):i(t)>,<i(m):i(m)>;
38.2		ENTRY MATCH<e:e'> → IN<e:e',>;
38.3		IN<e:ℓ>, ENTRY MATCH<e:e'> → IN<e:e',ℓ>,<e:ℓe',>;
38.4		IN<e:ℓ>, DIFF ENTRY<e:e'> → IN<e:e',ℓ>,<e:ℓe',>;
38.5	NOT IN	ENTRY<e> → NOT IN<e:Λ>;
38.6		NOT IN<e:ℓ>, DIFF ENTRY<e:e'> → NOT IN<e:e',ℓ>;
39.1	NOT CONT	CHAR STR OR NULL<s> → NOT CONT<s: >;
39.2		NOT CONT<sx:t>, DIFF CHAR<x:y> → NOT CONT<sx:ty>;
39.3		NOT CONT<sxa:ty>, DIFF CHAR<x:y> → NOT CONT<sxa:tya>;
40.1	DIFF ENTRY LIST	DIFF ENTRY LIST<Λ>;
40.2		DIFF ENTRY LIST<ℓ>, ENTRY<e>, NOT IN<e:ℓ> → DIFF ENTRY LIST<e,ℓ>;
40.3	DISJ ENTRY LISTS	ENTRY LIST<ℓ> → LIST OF LISTS:UNION<(ℓ):ℓ>;
40.4		LIST OF LISTS:UNION<ℓ:u>, ENTRY LIST<ℓ'> → LIST OF LISTS:UNION<(ℓ),(ℓ'):uℓ'>;
40.5		ENTRY LIST<ℓ> → DISJ PAIR OF LISTS<ℓ:Λ>;
40.6		DISJ PAIR OF LISTS<ℓ:ℓ'>, ENTRY<e>, NOT IN<e:ℓ> → DISJ PAIR OF LISTS<ℓ:e,ℓ'>;
40.7		ENTRY LIST<ℓ> → DISJ ENTRY LISTS<(ℓ)>;
40.8		DISJ ENTRY LISTS<ℓ>, LIST OF LISTS:UNION<ℓ:u>, DISJ PAIR OF LISTS<u:ℓ'> → DISJ ENTRY LISTS<ℓ(ℓ')>;
41.1	L1:L2 :INTERSEC	ENTRY LIST<ℓ> → L1:L2:INTERSEC:<ℓ:Λ:Λ>, L1:L2:REL COMP<ℓ:Λ:Λ>;
41.2		L1:L2:INTERSEC<ℓ:ℓ':i>, ENTRY<e>, IN<e:ℓ> → L1:L2:INTERSEC<ℓ:e,ℓ':e,i>;
41.3		L1:L2:INTERSEC<ℓ:ℓ':i>, ENTRY<e>, NOT IN<e:ℓ> → L1:L2:INTERSEC<ℓ:e,ℓ':i>;
41.4	L1:L2 :REL COMP	L1:L2:REL COMP<ℓ:ℓ':r>, ENTRY<e>, IN<e:ℓ> → L1:L2:REL COMP<ℓ:e,ℓ':r>;
41.5		L1:L2:REL COMP<ℓ:ℓ':r>, ENTRY<e>, NOT IN<e:ℓ> → L1:L2:REL COMP<ℓ:e,ℓ':e,r>;
41.6		L1:L2:INTERSEC<ℓ:ℓ':i>, L1:L2:REL COMP<ℓ:ℓ':r> → L1:L2:INTERSEC:REL COMP<ℓ:ℓ':i:r>;

Appendix 4.2 CANONICAL SYSTEM SPECIFYING THE TRANSLATION OF ALGOL/60 INTO THE TARGET LANGUAGE

6.2	UNSIGN NUM	DIGIT STR<s>,<t> → UNSIGN NUM<s..'s'>,<.t..(TRANS_FRAC 't')>, <s.t..(+(TRANS_INT 's', TRANS_FRAC 't'))>;
6.3	INT	UNSIGN INT<i> → INT<i..'i'>,<+i..'i'>,<-i..'-i'>;
6.4	NUM	UNSIGN NUM<n..n'> → NUM<n..n'>,<+n..n'>,<-n..(NEGATE n')>;
7.1	ID	IDSTR<i> → ID:NAME FORMALS:OWN VARS<i..i:Λ:Λ>,<i..(i 'Λ'):i,:Λ>, <i..i:Λ:i,>;
7.2	IDLIST	IDSTR<i> → IDLIST<ALTSEQ(i ,)>;
8.1	VAR	ARITH EXP<a..a'> → SUBSCRIPT LIST<a..(CONV_TO_INT a')>;
8.2		ARITH EXP<a..a'>, SUBSCRIPT LIST<ℓ..ℓ'> → SUBSCRIPT LIST<ℓ,a..[ℓ',(CONV_TO_INT a')]>;
8.3		ID<i..i'> → REAL/INT/BOOL VAR<i..i'>;
8.4		ID<i..i'>, SUBSCRIPT LIST<ℓ..ℓ'> → REAL/INT/BOOL VAR<i[ℓ]..(GET_EL (ℓ',i'))>;
9.1	FCN DES	ID<i..i'> → ACT PAR<i..λπ.i'>;
9.2		ID<i..i'> → ACT PAR<i..λπ.i'>;
9.3		ID<i..i'> → ACT PAR<i..λπ.i'>;
9.4		REAL/INT/BOOL VAR<v..v'> → ACT PAR<v..λπ.v'>;
9.5		ARITH EXP<a..a'> → ACT PAR<a..λπ.a'>;
9.6		BOOL EXP<b..b'> → ACT PAR<b..λπ.b'>;
9.7		DES EXP<d..d'> → ACT PAR<d..λπ.d'>;
9.8		ACT PAR<p..p'>, PAR DELIM<d> →ACT PAR PART<ALTSEQ(p d)..ALTSEQ(p' ,)>;
9.9		ID<i..i'> →REAL/INT/BOOL/NONVAL FCN DES<i..(i' 'Λ')>;
9.10		ID<i..i'>, ACT PAR PART<p..p'>→REAL/INT/BOOL/NONVAL FCN DES<i(p)..(i'(p',))>
9.11		REAL FCN DES<f..f'> | INT FCN DES<f..f'> | BOOL FCN DES<f..f'> | NONVAL FCN DES<f..f'> → FCN DES<f..f'>;
10.1	ARITH EXP	UNSIGN NUM<p..p'> | REAL VAR<p..p'> | INT VAR<p..p'> | REAL FCN DES<p..p'> | INT FCN DES<p..p'> → PRIM<p..p'>;
10.2		ARITH EXP<a..a'> → PRIM<(a)..a'>;
10.3		PRIM<p..p'>, MULT OP<m> → TERM<ALTSEQ(p m)..COMB(p' m)>;
10.4		TERM<t..t'>, ADD OP<a> → TERM SEQ<ALTSEQ(t a)..COMB(t' a)>;
10.5		TERM SEQ<s..s'> → SIMPLE ARITH EXP<s..s'>,<+s..s'>,<-s..(NEGATE s')>;
10.6		SIMPLE ARITH EXP<s..s'> → ARITH EXP<s..s'>;
10.7		BOOL EXP<b..b'>, SIMPLE ARITH EXP<s..s'>, ARITH EXP<a..a'> → ARITH EXP<IF b THEN s ELSE a'..b' ⇒ s' ELSE ⇒ a'>;
11.1	BOOL EXP	BOOL PRIM<TRUE..'TRUE'>,<FALSE..'FALSE'>;
11.2		SIMPLE ARITH EXP<a..a'>,<b..b'>, REL OP<r> → RELATION<arb..(r(a',b'))>;
11.3		RELATION<p..p'> | BOOL VAR<p..p'> | BOOL FCN DES<p..p'> → BOOL PRIM<p..p'>;
11.4		BOOL EXP<b..b'> → BOOL PRIM<(b)..b'>;
11.5		BOOL PRIM<p..p'> → BOOL SEC<p..p'>,< p..(p')>;
11.6		BOOL SEC<s..s'> → BOOL FAC<ALTSEQ(s ∧)..COMB(s' ∧)>;
11.7		BOOL FAC<f..f'> → BOOL TERM<ALTSEQ(f ∨)..COMB(f' ∨)>;
11.8		BOOL TERM<t..t'> → BOOL IMP<ALTSEQ(t ⊃)..COMB(t' ⊃)>;
11.9		BOOL IMP<i..i'> → SIMPLE BOOL<ALTSEQ(i ≡)..COMB(i' ≡)>;
11.10		SIMPLE BOOL<s..s'> → BOOL EXP<s..s'>;
11.11		BOOL EXP<b..b'>,<c..c'>, SIMPLE BOOL<s..s'> → BOOL EXP<IF b THEN s ELSE c..b' ⇒ s' ELSE ⇒ c'>;
12.1	DES EXP	LABEL:VAL<ℓ:v> → SIMPLE DES EXP<ℓ.. λπ. .v>;
12.2		ID<i..i'>, ARITH EXP<a..a'> → SIMPLE DES EXP<i[a]..((GET_EL(CONV_TO_INT a',i')) 'Λ')>;
12.3		DES EXP<d..d'> → SIMPLE DES EXP<(d)..d'>;
12.4		SIMPLE DES EXP<s..s'> → DES EXP<s..s'>;
12.5		BOOL EXP<b..b'>, SIMPLE DES EXP<s..s'>, DES EXP<d..d'> → DES EXP <IF b THEN s ELSE d..b' ⇒ s' ELSE ⇒ d'>;
13.	EXP	ARITH EXP<e..e'> | BOOL EXP<e..e'> | DES EXP<e..e'> → EXP<e..e'>
14.	DUMMY STM	DUMMY STM<Λ..'Λ'>;
15.	COMMENT STM	STR<s> → COMMENT STM COMMENT<s..'Λ'>;
16.	GOTO STM	DES EXP<d..d'> → GOTO STM<GO TO d..(GOTO. d')>;
17.	PROC STM	FCN DES<f..f'> → PROC STM<f..f'>;
18.1	ASGT STM	IDSTR<i> → R/I/B LEFT PART<i..(i# ASSIGN. π)>;
18.2		REAL/INT/BOOL VAR<i..i'>, IDSTR<i> → R/I/B LEFT PART<i..LET a=i' IN (a ASSIGN. π)>;
18.3		REAL/INT/BOOL VAR<i[ℓ]..(GET_EL(i',ℓ')> → R/I/B LEFT PART<i[ℓ]..LET a=i' IN (a ASSIGN. (RESET_EL(ℓ',i',π))>;
18.4		R/I/B LEFT PART<ℓ..ℓ'>, ARITH/ARITH/BOOL EXP<e..e'> → R/I/B ASGT STM<ℓ:=e.. LET π=(CONV_TO_REAL/CONV_TO_INT/IDEN π e') IN ℓ'>;
18.5		R/I/B LEFT PART<ℓ..ℓ'>, R/I/B ASGT STM<s..s'> → R/I/B ASGT STM<ℓ:=s..s';ℓ'>;
18.6		R/I/B ASGT STM<s..s'> → ASGT STM<s..s'>;
19.1	FOR STM	ARITH EXP<a..a'> → FOR LIST EL<a..λπ.a'>;
19.2		ARITH EXP<a..a'>,<b..b'>,<c..c'> → FOR LIST EL<a STEP b UNTL c..λπ.(STEP(λπ.a',λπ.b',λπ.c'))>;
19.3		ARITH EXP<a..a'>, BOOL EXP<b..b'> → FOR LIST EL<a WHILE b..λπ.(WHILE(λπ.a',λπ.b'))>;
19.4		FOR LIST EL<e..e'> → FOR LIST<ALTSEQ(e,)..ALTSEQ(e' ,)>;
19.5		REAL/INT VAR<v..v'>, FOR LIST<t..t'>, STM<s..s'> → FOR STM<FOR v:=t DO s..(FOR(v', DELAY_CAT [t',s']))>;

20.1	UNCOND STM	DUMMY STM<s..s'> | COMMENT STM<s..s'> | GOTO STM<s..s'> | PROC STM<s..s'> | ASGT STM<s..s'> FOR STM<s..s'> | BLOCK<s..s'> | COMPOUND STM<s..s'> → UNCOND STM<s..s'>;
20.2		UNCOND STM s..s' , LABEL:VAL :v UNCOND STM : s..v : s' ;
21.1	COND STM	BOOL EXP<b..b'>, UNCOND STM<u..u'> → LABEL:VAL<ℓ$_1$:v$_1$>,<ℓ$_2$:v$_2$> → COND STM:LABELS<IF t THEN u.. b' ⇒(GOTO. .v$_1$) ELSE ⇒(GOTO. .v$_2$);v$_1$`:`u';v$_2$`:`'A':v$_1$,ℓ$_2$,ℓ$_1$;
21.2		BOOL EXP<b..b'>, UNCOND STM<u..u'>, STM<s..s'> → LABEL:VAL<ℓ$_1$:v$_1$>,<ℓ$_2$:v$_2$>,<ℓ$_3$:v$_3$> → COND STM:LABELS<IF b THEN u.. b' ⇒ (GOTO. .v$_1$) ELSE ⇒ (GOTO. .v$_2$);v$_1$`:`u';(GOTO. .v$_3$);ℓ$_2$`:`s';ℓ$_3$`:`'A':v$_1$,v$_2$,v$_3$,`>;
21.3		COND STM<s..s'>, LABEL:VAL<ℓ:v> → COND STM<ℓ`:`s..v`:`s'>;
22.1	STM	UNCOND STM<s..s'> | COND STM<s..s'> → STM<s..s'>;
22.2	STM SEQ	STM<s..s'> → STM SEQ<s..s'>;
22.3		STM<s..s'> STM SEQ<q..q'> → STM SEQ<q;s..q';s'>;
23.	COMPOUND STM	STM SEQ<s>, STM<e> → COMPOUND STM<BEGIN s END e..s'>;
24.1	TYPE DEC	IDLIST<ℓ>, LIST:CORR NULL LIST<ℓ:ℓ$_n$> → TYPE DEC<REAL/INTEGER/BOOLEAN ℓ..ℓ=ℓ$_n$>;
24.2		IDLIST<ℓ>, LIST:CORR INDEXED LIST<ℓ:ℓ$_i$> → TYPE DEC:DEC OWN VARS<OWN REAL/INTEGER/BOOLEAN ℓ..ℓ=ℓ$_i$:ℓ$_i$>;
25.1	ARRAY DEC	ARITH EXP<a..a'>,<b..b'> → BOUND PAIR<a`:`b..a'|b'>;
25.2		BOUND PAIR<p..a|b> → BPLIST<p..a|b>;
25.3		BOUND PAIR<p..a|b>, BPLIST<ℓ..x|y> → BPLIST<ℓ,p..x,a|y,b>;
25.4a		BPLIST<ℓ..x|y>, IDSTR<i> → ARRAY:ARRAY IDS<i[ℓ]..i=(MAKE_LIST([illegible])):i,`>;
25.4b		BPLIST<ℓ..x|y>, IDSTR<i>, LIST:CORR INDEXED LIST<i,:j,> → ARRAY:OWN IDS<i[ℓ]..(i=(RESET_LIST(j,[illegible]))>;
25.5		ARRAY<i[ℓ]..i=x> → ARRAY SEG<i[ℓ]..i=x>;
25.6		ARRAY<i[ℓ]..i=x>, ARRAY_SEG<s[ℓ]..p=q> → ARRAY SEG<i,s[ℓ]..i,p=q,x>;
25.7		ARRAY SEG<s..p=q> → ARRAY LIST<s..p=q>;
25.8		ARRAY SEG<s..p=q>, ARRAY LIST<ℓ..x=y> → ARRAY LIST<ℓ,s..p,x=q,y>;
25.9		ARRAY LIST:ARRAY IDS:OWN IDS<ℓ..ℓ':i:A> → ARRAY DEC<REAL/INTEGER/BOOLEAN ARRAY ℓ..ℓ'>;
25.10		ARRAY LIST:ARRAY IDS:OWN IDS<ℓ..ℓ':A:i> → ARRAY DEC:DEC OWN ARRAYS<OWN REAL/INTEGER/BOOLEAN ARRAY ℓ..ℓ':i>;
26.1	SW DEC	DES EXP<d..d'> → SW LIST<ALTSEQ(d ,)..ALTSEQ(λ=.d' ,)>;
26.2		IDSTR<i>, SW LIST<ℓ..ℓ'> → SW DEC<SWITCH i:=ℓ..i=(INDEX_LIST('I',ℓ')>;
27.12	PPOC DEC	IDSTR<i>, FORMAL PAR PART:PARS<f:f$_p$>, VALUE PART:PARS<u:u$_p$>, SPECIFIER PART<c>, STM:NAME FORMALS<s..s':n>, L1:L2:REL COMP<f$_p$:u$_p$:n$_p$>,<a:n$_p$:n'>, L1:L2:INTERSEC<a:u$_p$:A>, LIST:CORR UNSHARE LIST<u$_p$:ℓ$_u$> → PROC STM:NAME FORMALS <REAL/INTEGER/BOOLEAN/A PROCEDURE i f;usc.. i(f$_p$)=LET iℓ,u$_p$='A',ℓ$_u$ IN s'; iℓ :n'>;
28.1	DEC	TYPE DEC<d..d'> | ARRAY DEC<d..d'> | SW DEC<d..d'> | PROC DEC<d..d'> → DEC<d..d'>
28.2	DEC SEQ	DEC<d..x=y> → DEC SEQ<d..REC x=y>;
28.3		DEC<d..x=y>, DEC SEQ<s..REC x'=y'> → DEC SEQ<s;d..REC x',x=y',y>;
29.	BLOCK	STM SEQ:OWN VARS:OWN ARRAYS<s..s':v$_0$:a$_0$>, DEC SEQ:OWN VARS:OWN ARRAYS:DEC OWN VARS :DEC OWN ARRAYS<d..d':v$_0'$:a$_0'$:v$_{0d}$:a$_{0d}$>, L1:L2:REL COMP<v$_0$v$_0'$:v$_{0d}$:v$_0''$>, <a$_0$a$_0'$:a$_{0d}$:a$_0''$>, STR<e> → BLOCK:OWN VARS:OWN ARRAYS:GLOBAL VARS:GLOBAL ARRAYS <BEGIN d;s END e.. LET d' IN s':v$_0''$:a$_0''$:v$_{0d}$:v$_{0d}$>;
30.1	ALGOL PROGRAM	BLOCK<p..p'> | COMPOUND STM<p..p'> → PROGRAM STR<p..p'>;
30.2		PROGRAM STR<s..s'>, LABEL:VAL<ℓ:v> → PROGRAM STR<ℓ`:`s..v`:`s'>;
30.3		PROGRAM STR:GLOBAL VARS:GLOBAL ARRAYS:OWN VARS:OWN ARRAYS:NAME FORMALS <s..s':v$_g$:a$_g$:A: :A> DIFF ENTRY LIST<v$_g$a$_g$>, LIST:CORR NULL LIST<v$_g$:ℓ$_n$>,<a$_g$:ℓ$_a$> → ALGOL PROGRAM<s..LET v$_g$a$_g$= ℓ$_n$ℓ$_a$ IN s'>;
42.1	LIST:CORR NULL LIST	LIST:CORR NULL LIST<Λ:Λ>;
42.2		LIST:CORR NULL LIST<ℓ:m>, IDSTR<i> → LIST:CORR NULL LIST<i,ℓ:'A',m>;
42.3	LIST:CORR UNSHARE LIST	LIST:CORR UNSHARE LIST<Λ:Λ>
42.4		LIST:CORR UNSHARE LIST<ℓ:m>, IDSTR<i> → LIST:CORR UNSHARE LIST<i,ℓ:(UNSHARE (i 'A')),m>;
42.5	LIST:CORR INDEXED LIST	LIST:CORR INDEXED LIST<Λ:Λ>;
42.6		LIST:CORR INDEXED LIST<ℓ:m>, IDSTR<i>, UNSIGN INT<j> → LIST:CORR INDEXED LIST<i,ℓ:i#j,m>;

Appendix 4.3 DEFINITION OF PRIMITIVE FUNCTIONS FOR ALGOL/60

Set definitions for string variables: | d∈DIGIT | r,s,t∈STR |

```
DIGIT          DIGIT<0>,<1>, ... ,<9>;
CHAR           LETTER<A>,<B>, ... ,<Z>,<"a">,<"b">, ... <"z">;
               MARK<+>,<->, ... ,<;>|
               DIGIT<p> | LETTER<p> | MARK<p>  →  CHAR<p>;

STR            STR<Λ>;
               STR<s>, CHAR<c>  →  STR<sc>;
```

(a) Miscellaneous basic primitives

```
CAT α =          [       s.        →.  "[Λ→." s "]" ] α

EQ(α,β) =        [      s/s        →.   TRUE         ] α/β
                 [      s/t        →.   FALSE        ]

NEQ(α,β) =       [      s/s        →.   FALSE        ] α/β
                 [      s/t s      →.   TRUE         ]

COND(r,α,β) =    [    TRUE         →.     α          ] r
                 [    FALSE        →.     β          ]

                 [   TRUE/TRUE     →.   TRUE         ]
AND(α,β) =       [   TRUE/FALSE    →.   FALSE        ] α/β
                 [   FALSE/TRUE    →.   FALSE        ]
                 [   FALSE/FALSE   →.   FALSE        ]

NOT α =          [    TRUE         →.   FALSE        ] α
                 [    FALSE        →.   TRUE         ]

HD α =           [    /r,s/        →.     r          ] /α/
                 [    /r/          →.     r          ]

TL α =               [ /r,s/       →.       s   ] α
                     [ /r/         →.       Λ   ]

UNSHARE α =          [  s.         →.       s   ] α

IDEN X =                X
```

(b) Basic arithmetic and boolean primitives

```
ABS α =              [ -s.     →.      s    ]  α
                     [  s.     →.      s    ]

NEGATE α =           [ -s.     →.      s    ]  α
                     [  s.     →.     -s    ]

IS_POS α =           [ -s.     →.    FALSE  ]  α
                     [  s.     →.    TRUE   ]

IS_NEG α =           [ -s.     →.    TRUE   ]  α
                     [  s.     →.    FALSE  ]

NUM α =              [ sDt.    →.      s    ]  α
                     [  s.     →.      s    ]

DEN α =              [ sDt.    →.      t    ]  α
                     [  s.     →.      1    ]

IS_INT α =           [ sDt.    →.    FALSE  ]  α
                     [  s.     →.    TRUE   ]

MAKE_REAL(α,β) =     [ s/t.    →.     sDt   ]  α/β
```

(c) Arithmetic conversion primitives (see arithmetic primitives for definitions of ÷ and /)

$$\texttt{TRANS\_INT } \alpha = \begin{bmatrix} \texttt{/0ds/} & \rightarrow & \texttt{/ds/} \\ \texttt{/s/} & \rightarrow\cdot & \texttt{sD1} \end{bmatrix} /\alpha/$$

$$\texttt{TRANS\_FRAC } \alpha = \begin{bmatrix} \texttt{/sd0/} & \rightarrow & \texttt{/sd/} \\ \texttt{/sd/tDr.} & \rightarrow & \texttt{/s/dtD0r} \\ \texttt{//rDr.} & \rightarrow\cdot & \texttt{tD1r} \\ \texttt{/s/} & \rightarrow & \texttt{/s/D} \end{bmatrix} /\alpha/$$

$$\texttt{CONV\_TO\_REAL } \alpha = \begin{bmatrix} \texttt{sDt.} & \rightarrow\cdot & \texttt{sDt} \\ \texttt{s.} & \rightarrow\cdot & \texttt{sD1} \end{bmatrix} \alpha$$

```
ENTIER X =           LET A,B = NUM X,DEN X
                     IN  /(A,B)

CONV_TO_INT X =      ENTIER(÷ (X, '1D2'))
```

(d) Arithmetic primitives

$$\texttt{SUCC } \alpha = \begin{bmatrix} \texttt{/s0/r.} & \rightarrow\cdot & \texttt{s1r} \\ \texttt{/s1/r.} & \rightarrow\cdot & \texttt{s2r} \\ & \vdots & \\ \texttt{/s8/r.} & \rightarrow\cdot & \texttt{s9r} \\ \texttt{/s9/r.} & \rightarrow & \texttt{/s/0r} \\ \texttt{//r.} & \rightarrow\cdot & \texttt{1r} \end{bmatrix} /\alpha/$$

$$\texttt{PRED } \alpha = \begin{bmatrix} \texttt{/0/.} & \rightarrow\cdot & \texttt{0} \\ \texttt{/1/9r.} & \rightarrow\cdot & \texttt{9r} \\ \texttt{/s0/r.} & \rightarrow & \texttt{/s/9r} \\ \texttt{/s1/r.} & \rightarrow\cdot & \texttt{s0r} \\ \texttt{/s2/r.} & \rightarrow\cdot & \texttt{s1r} \\ & \vdots & \\ \texttt{/s9/r.} & \rightarrow\cdot & \texttt{s8r} \end{bmatrix} /\alpha/$$

```
REC ∸(X,Y) =        EQ(Y, '0') ⇒ X
                    ELSE        ⇒ ∸ (PRED X, PRED Y)

REC SUM(X,Y) =      EQ(Y, '0') ⇒ X
                    ELSE        ⇒ SUM ( SUCC X, PRED Y)

LESS (X,Y) =        NEQ (∸(Y,X), '0')

REC PROD(X,Y) =     EQ(Y,'0') ⇒ '0'
                    ELSE       ⇒ SUM (X, PROD(X, PRED Y))

DIFF (X,Y) =        LESS(X,Y) ⇒ NEGATE(∸(Y,X))
                    EQ(X,Y)   ⇒ '0'
                    ELSE      ⇒ ∸(X,Y)

REC QUOT(X,Y) =     LESS(X,Y) ⇒ '0'
                    ELSE      ⇒ SUM('1',QUOT(∸(X,Y), Y))

PRI_SUM(X,Y) =      AND(IS_INT X, IS_INT Y) ⇒ SUM(X.Y)
                    ELSE                    ⇒ LET   N1,D1,N2,D2 = NUM X, DEN X, NUM Y,  DEN Y
                                              IN LET   N = DIFF(PROD(N1,D2), PROD(N2,D1 ))
                                                 IN    LET   D = PROD(D1,D2)
                                                       IN    MAKE_REAL(N,D)

PRI_DIFF(X,Y) =     AND(IS_INT X, IS_INT Y) ⇒ DIFF(X,Y)
                    ELSE                    ⇒ LET    N1,D1,N2,D2 = NUM X, DEN X, NUM Y,  DEN Y
                                              IN LET   N = DIFF(PROD(N1,D2),PROD(N2,D1))
                                                 IN    LET D = PROD(D1,D2)
                                                       IN  MAKE_REAL(N,D)

PRI_PROD(X,Y) =     AND(IS_INT X, IS_INT Y) ⇒ PROD(X,Y)
                    ELSE                    ⇒ LET    N1,D1,N2,D2 = NUM X, DEN X, NUM Y,  DEN Y
                                              IN LET   N = PROD(N1,N2)
                                                 IN    LET D = PROD(D1,D2)
                                                       IN MAKE_REAL(N,D)

PRI_QUOT(X,Y) =     AND(IS_INT X, IS_INT Y) ⇒ QUOT(X,Y)
                    ELSE                    ⇒ LET    N1,N2,N2,D2 = NUM X, DEN X, NUM Y, DEN Y
                                              IN LET   N = PROD(N1,D2)
                                                 IN    LET D = PROD(N2,D1)
                                                       IN  MAKE_REAL(N,D)
```

```
SIGN(X,Y) =         AND(IS_POS X, IS_POS Y) ⇒ 'Λ'
                    AND(IS_POS X, IS_NEG Y) ⇒ '-'
                    AND(IS_NEG X, IS_POS Y) ⇒ '-'
                    ELSE                    ⇒ 'Λ'

+(X,Y) =            AND(IS_POS X, IS_POS Y) ⇒ PRI_SUM(X,Y)
                    AND(IS_POS X, IS_NEG Y) ⇒ PRI_DIFF(X, ABS Y)
                    AND(IS_NEG X, IS_POS Y) ⇒ PRI_DIFF(Y, ABS X)
                    ELSE                    ⇒ NEGATE(PRI_SUM(ABS X, ABS Y))

×(X,Y) =            LET S = SIGN(X,Y)
                    IN  CAT(S, PRI_PROD(ABS X, ABS Y))

/(X,Y) =            LET S = SIGN(X,Y)
                    IN  CAT(S, PRI_QUOT(ABS X, ABS Y))

-(X,Y) =            + (X, NEGATE Y)

÷(X,Y) =            LET S = SIGN(X,Y)
                    IN  CAT(S, ENTIER(ABS (/(X,Y)))
```

(e) Boolean primitives

```
¬X =                NOT X

Λ(X,Y) =            AND (X,Y)

V(X,Y) =            NOT(AND(NOT X, NOT Y))

⊃(X,Y) =            NOT(AND(X, NOT Y))

≡(X,Y) =            EQ(X,Y)

PRI_LESS(X,Y) =     LET N1,D1,N2,D2 = NUM X, DEN X, NUM Y, DEN Y
                    IN  LESS(PROD(N1,D2), PROD(N2,D1))


<(X,Y) =            AND(IS_POS X, IS_POS Y) ⇒ PRI_LESS(X,Y)
                    AND(IS_POS X, IS_NEG Y) ⇒ FALSE
                    AND(IS_NEG X, IS_POS Y) ⇒ TRUE
                    ELSE                    ⇒ PRI_LESS(ABS Y, ABS X)

=(X,Y) =            EQ(X,Y)

≠(X,Y) =            NEQ(X,Y)

≤(X,Y) =            V(<(X,Y), = (X,Y) )

≥(X,Y) =            NOT(<(X,Y))

>(X,Y) =            NOT(≤(X,Y))
```

(f) For statement primitives

```
REC STEP(A,B,C) =   LET A¦B¦C¦ = (A 'Λ'),(B 'Λ'),(C 'Λ')
                    IN  AND(IS_POS B¦ LESS(C¦A')) ⇒ 'Λ'
                        AND(IS_NEG B¦ LESS(A¦C')) ⇒ 'Λ'
                        ELSE                      ⇒[A'+λπ.(STEP(λπ. (+(A¦B¦)),B,C)]]

REC WHILE(A,B) =    LET A¦B' = (A 'Λ'),(B 'Λ')
                    IN  NOT B' ⇒ 'Λ'
                        ELSE   ⇒[A'+λπ.(WHILE (A,B))]

REC DELAY_CAT L =   LET H,T = HD L, TL L
                    IN  LET  H' = (H 'Λ')
                        IN   EQ(T, 'Λ') ⇒ H'
                             EQ(H¦ 'Λ') ⇒ (DELAY_CAT T)
                             ELSE       ⇒[H'+T]

REC FOR(V,L,S) =    LET H,T = HD L, TL L
                    IN  EQ(L, 'Λ') ⇒ 'Λ'
                        ELSE       ⇒ (IS_INT V) ⇒ (V ASSIGN. (CONV_TO_INT H)) ELSE ⇒
                                        V ASSIGN. (CONV_TO_REAL H)];
                                     (S 'Λ');
                                     FOR (V, (DELAY_CAT T), S)
```

(g) Array and list primitives

```
GET_EL(I,L) =                [ r(I,s)t.  →·     s     ] L

RESET_EL(I,L,X) =            [ r(I,s)t.  →·  r(I,X)t  ] L

REC INDEX_LIST(I,L) =        LET H,T = HD L, TL L
                             IN  NULL T  ⇒(I,H)
                                 ELSE    ⇒[(I,H)+ INDEX_LIST(+(I,1), T)]

REC LAST L =                 LET H,T = HD L, TL L
                             IN  NULL T  ⇒H
                                 ELSE    ⇒ LAST T

REC TRUNC L =                LET H,T = HD L, TL L
                             IN  NULL(TL T)  ⇒HD T
                                 ELSE        ⇒[H+ TRUNC T]

REC ADD1(SUBSLIST,LB,UB) = LET S_1,S_2,S_3,T_1,T_2,T_3 = LAST SUBSLIST,LAST LB,LAST UB,TRUNC LB,TRUNC UB
                             IN  NEQ(S_1,S_3)  ⇒[T_1+ (+(S_1, '1'))]
                                 ELSE          ⇒[ADD1(T_1,T_2,T_3)+ S_2]

REC MAKE_LIST(I,LB,UB) =     EQ(I,UB) ⇒ (I, 'A')
                             ELSE     ⇒ [(I, 'A') + MAKE_LIST((ADD1(I,LB,UB)), LB,UB)]

REC RESET_LIST               EQ(J,UB) ⇒ (J, GET_EL(J, ARRAY))
  (ARRAY,J̄,LB,UB)       =    ELSE     ⇒ [(J, GET_EL(J, ARRAY))+ RESET_LIST((ADD1(J,LB,UB)),LB,UB)]
```

Appendix 5. THEORETICAL BACKGROUND FOR CANONICAL SYSTEMS

The intent of this appendix is (a) to describe and relate the formalisms of Post's formal systems[1] and Smullyan's "elementary formal" systems,[2] (b) to show that the formalism of "canonical" systems presented in this dissertation is equivalent (except for changes in notation) to Smullyan's elementary formal system, and (c) to show that the terminology and interpretation of canonical systems given here relate to the terminology and interpretation of the formal systems of Post and Smullyan.

A formal system will be described by giving

(a) A set A of primitive symbols: For example, this set may be the symbols {0 1 ... 9} or the set of characters in a computer language.

(b) A set C of auxiliary symbols:* For example, this set may include the symbols {SQ + =}.

(c) A set S of initial statements composed from the primitive and auxiliary symbols: The set S will be composed of strings from A∪C.**

(d) A set E of well-formed expressions: The set of well-formed expressions will generally incorporate symbols from A∪C and other symbols.

(e) A series of rules for using the well-formed expressions: The rules will be used to derive new statements containing the primitive symbols from the set S of initial statements.

*All sets of symbols in the systems of Post and Smullyan are assumed to be disjoint from each other.

**The symbol "∪" denotes the binary operation of set union.

(f) An *interpretation* of the formal system: Strictly speaking, an interpretation is not part of a formal system. An interpretation is placed on a formal system by a user, who wishes to draw conclusions about the objects that the symbols of the system represent.

POST'S SYSTEMS

(a) *Primitive Symbols*
Let A be a finite set of symbols $\{A_1\ A_2\ \ldots\ A_i\}$.

(b) *Auxiliary Symbols*
Let C be a finite set of symbols $\{C_1\ C_2\ \ldots\ C_j\}$.

Let L be the set $A \cup C$, the union of the sets A and C. Post calls the set L the set of "primitive letters" and does not distinguish the sets A or C. The sets A and C are distinguished here to clarify the distinction between a Post system and a Smullyan elementary formal system.

(c) *Initial Statements*
The initial statements S are a set $\{S_1\ S_2\ \ldots\ S_k\}$, where each S_i, $1 \le i \le k$, is a string of letters from L.

(d) *Well-formed Expressions*
Let V be a finite set of symbols $\{V_1\ V_2\ \ldots\ V_\ell\}$ called *variables*.
A *premise* is a string of symbols from $L \cup V$.
A *conclusion* is a string of symbols from $L \cup V$.
A well-formed expression is a string of the form "$Q_1, Q_2, \ldots, Q_m \xrightarrow{\text{produce}} C$" where the Q_i, $1 \le i \le m$, are premises and C is a conclusion such that each variable in C also occurs in at least one Q_i. A well-formed expression is called a *production*.

A set E is a *system in canonical form* if E is a finite set $\{P_1\ P_2\ \ldots\ P_n\}$, where each P_i, $1 \le i \le n$, is a production.

(e) Rules for *Using-Formed Expressions*

Rule 1: A string X is called an *instance** of a production P_i if X can be obtained from P_i by substituting for *each* variable in P_i some string (possibly null) of letters from L. The string substituted for each occurrence of the same variable must be the same.

*The word "instance" is not used by Post.

Rule 2: If each premise in an instance of a production has been derived, then the conclusion of the production can be derived.

The statements derivable from a Post system are

(a) The initial statements

(b) The statements that can be derived from the productions by first applying Rule 1 to obtain an instance of the production and then applying Rule 2 to the production instance.

(f) Interpretation

A production can be viewed as a rewriting rule for obtaining new statements from previously derived statements. The interpretation of the derived statements are subject to the interpretation of the initial letters.

Example 1: A Post System Defining the Set of Squares of Positive Integers

(a) Primitive Symbols A = {1}

(b) Auxiliary Symbols C = {SQ}

L = {1 SQ}

(c) Initial Statements S = {1SQ1}

(d) Well-formed Expressions V = {u v}
E = {uSQv→u1SQuuv1}

(e) Derived Statements {1SQ1 11SQ1111 111SQ111111111 ...}

(f) Interpretation

The string of ones occurring to the left of "SQ" represents the positive integer denoted by the number of ones.

The string of ones occurring to the right of "SQ" represents the positive integer that is the numerical square of the integer to the left of "SQ".

Example 2: Another Post System Defining the Set of Squares of the Positive Integers.

Note: The intent of this example is to illustrate that the "canonical systems" given in this dissertation fit the definition of a system in canonical form given by Post.

(a) Primitive Symbols   A = {1}

(b) Auxiliary Symbols   C = {N:SQ < > :}

L = A∪C = {1 N:SQ < > :}

(c) Initial Statements   S = {N:SQ<1>}

(d) Well-formed Expressions   V = {u v}
E = {N:SQ<u:v>→N:SQ<u1:uuv1>}

(e) Derived Statements
{N:SQ<1:1> N:SQ<11:1111> N:SQ<111:111111111> ...}

(f) Interpretation
The string "N:SQ" is the name of a set.
The string "<x:y>", where x and y are strings of ones, are members of the set "N:SQ".
The string of ones before the ":" represents a positive integer; the string of ones to the right of the ":" represents the square of the positive integer to the left of the ":".

SMULLYAN'S "ELEMENTARY FORMAL" SYSTEMS[2]

Smullyan's elementary formal systems are a descendant of Post's formal systems.

(a) Primitive Symbols
Let A be a finite set of symbols {A_1 A_2 ... A_i} called the object alphabet.

(b) Auxiliary Symbols
Let P be a set of symbols {P_1 P_2 ...} called the predicate alphabet. With each predicate alphabet symbol we associate a unique positive integer called its degree.
Let Z be the set {, →} . The symbol "→" is called the "implication sign and the symbol "," is called the "punctuation" sign.
The set C of auxiliary symbols is the set P∪Z.

(c) Initial Statements - None
Smullyan includes the initial statements as members of the set of well-formed expressions.

(d) Well-formed expressions
Let V be a set of symbols {V_1 V_2 ...} called the set of variables.
A term is a string from V∪A.

A well-formed atomic formula is a string of the form "$Pt_1, t_2, \ldots, t_k$" where t_i, $1 \leq i \leq k$, are terms and P is a predicate of degree k.

A well-formed expression is either an atomic formula or an expression of the form $X_1 \rightarrow X_2 \ldots \rightarrow X_m$ (assuming association to the right; e.g., "$X_1 \rightarrow X_2 \rightarrow X_3$" is to be read "$X_1$ implies (X_2 implies X_3)") where X_i, $1 \leq i \leq m$ are atomic formulas.* A well-formed expression is called a well-formed formula.

A set E is an elementary formal system if E is a finite set $\{F_1\ F_2\ \ldots\ F_n\}$ where the F_i, $1 \leq i \leq n$, are well-formed formulas, called axioms.

(e) Rules for Using Well-formed Expressions

Rule 1: (Substitution) A formula F' can be derived from a formula F by substitution if F' can be obtained from F by substituting a string in A for each occurrence of some variable in F.**

Rule 2: (Modus Ponens) A formula F' can be derived from a formula F by modus ponens if F is the form X → F' and X is some previously derived atomic formula. More generally, a formula X_n can be derived from a formula of the form $X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_{n-1} \rightarrow X_n$ if each X_i, $1 \leq i \leq n$, is an atomic formula and $X_1, X_2, \ldots, X_{n-1}$ have each been previously derived. In this case, we refer to the $X_1, X_2, \ldots$, and X_{n-1} as premises, X_n as a conclusion, and say that the conclusion X_n is derivable from the conjunction of the premises $X_1, X_2, \ldots$, and X_{n-1}.***

The "provable strings" of an elementary formal system E are

(i) the axioms of E

(ii) the strings that can be derived from the axioms by a finite number of applications of rules 1 and 2.

*Note that no restriction is placed on the use of a variable occurring in X_m but not in X_i, $1 \leq i \leq m-1$.

**In an elementary formal system, it is not necessary to substitute object strings for each variable in formula to derive strings from the well-formed formulas. Thus we can derive strings containing variables in an elementary formal system. In a Post system, we must substitute object strings for each variable in a production before we can derive strings.

***If each variable is replaced by an object string, this generalization of modus ponens is identical to rule 2 for deriving strings given by Post.

An *instance* of a well-formed formula F is a string obtained from F by applying rule 1 (substitution) to *all* variables in F. A formula so obtained is called a *sentence*.

The "provable sentences" of an elementary formal system E are the provable strings containing *no* variables.

(f) *Interpretation*
Let P be a predicate of degree k in an elementary formal system E, and let Y be a set of k-tuples of strings from A. We say that the predicate P *represents* the set Y if the following condition holds: $PX_1,X_2, \dots ,X_k$ is a provable sentence in E if and only if the k-tuple $(X_1, X_2, \dots ,X_k)$ is contained in Y.

Thus an elementary formal system can be viewed as a set of axioms used to enumerate the members of sets whose names are denoted by the predicates.

Example 3: An Elementary Formal System Defining the Set of Squares of the Positive Integers

(a) *Primitive Symbols* A = {1}

(b) *Auxiliary Symbols* P = {R} Z = {, →}

(d) *Well-formed Expressions* V = {u v}
E = {R1,1 Ru,v →Ru1,uuv1}

(e) *Derived Statements*
{R1,1 R11,1111 R111,111111111 ...}
The derived statements given above are (in the Smullyan sense) the *atomic sentences* derived from E.

(f) *Interpretation*
If R is the name of a set, the ordered pairs {(1,1) (11,1111) (111,111111111) ...} are the members of R. We interpret the set R as containing all ordered pairs such that the string to the left of the "," represents a positive integer and the string to the right of the "," represents the positive integer that is the square of the integer represented by the string of ones to the left of the ",".

CANONICAL SYSTEMS (as presented in this dissertation)

The formalism called "canonical systems", as presented in this dissertation, is equivalent (except for changes in notation) to Smullyan's elementary formal systems.

(a) Primitive Symbols In this dissertation the primitive or "object" alphabet is the set of characters used in some computer language.

(b) Auxiliary Symbols The predicate alphabet P here is a string of English letters or digits each separated by the tuple sign ":". Each string of English letters of digits is called a predicate part, and the number of predicate parts in a predicate is usually identical to the number of terms in a term tuple following the predicate. The separation of predicates into parts is made (a) to give some mnemonic describing the role of each term in a term tuple following the predicate, and (b) to provide a convenient notation for abbreviating a canonical system.

The set Z is given as {: →} rather than {, →} since the comma "," is a character occurring frequently in computer languages.

(d) Well-formed Expressions A well-formed formula "$X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_{n-1} \rightarrow X_n$" is written here as "$X_1, X_2, \ldots ,X_{n-1} \rightarrow X_n$" to connote the meaning that X_n is derivable from a canonical system if and only if each of the instances of the premises $X_1, X_2, \ldots ,X_{n-1}$ are derivable. This alternate formulation is in the spirit of Post.

The delimiter ";" is introduced here to separate the well-formed formulas of a canonical system. The well-formed formulas in a Smullyan system are separated by the use of appropriate spacing of formulas in a page of text.

Furthermore, the string of terms following a predicate is enclosed by the angle brackets "<" and ">" so that the characters "," , ";" and "→" can be used in the terms as object symbols without the use of quotation marks.

(e) Rules for Using Well-Formed Expressions The rules for using well-formed productions of a canonical system are identical to the rules used by Smullyan.

(f) Interpretation The interpretation given to a canonical system here is a hybrid of the interpretation of the systems of Post and Smullyan

(i) The productions of a canonical system are viewed as rewriting rules (Post).

(ii) The derived strings of a canonical system are viewed as statements about the membership of n-tuples of strings in sets whose names are given by the predicates (Smullyan).

REFERENCES

The following works describe the theoretical foundations of canonical 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 canonic systems to computer languages:

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

4. Henry F. Ledgard
   A Scheme for the Translation of Computer Languages, Ph.D. dissertation proposal, M.I.T., Cambridge, Massachusetts, 1967.
   This reference applies canonic 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.

6. Joseph W. Alsop
A Canonic Translator
MAC-TR-46, Project MAC, M.I.T., 1967
This reference describes an algorithm that uses a canonic system specification of a language as a data base to recognize strings specified by the canonic system and generate their translation.

7. James T. Doyle
Issues of Undecidability in Canonic Systems, S.M. dissertation, M.I.T., Cambridge, Massachusetts, 1968.

8. Joseph P. Haggerty
Complexity Measures for Language Recognition by Canonic Systems, S.M. dissertation, M.I.T., Cambridge, Massachusetts, 1969.

The following is the basic reference for Markov algorithms:

9. Andrei A. Markov
Theory of Algorithms
Acadamy of Sciences of the USSR, Moscow, 1954, English Translation by Israel Program for Scientific Translations.

The following describe the extension of Markov algorithms used in this dissertation.

10. A. Caracciolo di Forino
Generalized Markov Algorithms and Automata
Lecture delivered at the International Summer School of Physics Course on Automata Theory, Ravello, Italy, 1964.

11. A. Caracciolo di Forino and N. Wolkenstein
On a Class of Programming Languages for Symbol Manipulation based on Extended Markov Algorithms, Centro Sudi Calcolatrici Electroniche del C.N.R., Pisa Italy, 1963.

12. A. Caracciolo di Forino
String processes and generalized Markov algorithm in Symbol Manipulation Languages and Techniques, North-Holland Publishing Company, Amsterdam, 1968.

The following are other references on Markov algorithms:

13. Anton P. Zeleznikar
Some Algorithm Theory and its Applicability
American Mathematical Society Translations, Series 2, Volume 18, pp. 141-158, 1963. This reference describes a 2-dimensional variant of Markov algorithms.

14. V. K. Detlovs
The Equivalence of Normal Algorithms and Recursive Functions
American Mathematical Society Translations, Series 2, Volume 23, pp. 15-82, 1963.

15. V. S. Cernjavskii
On a Class of Normal Markov Algorithms
American Mathematical Society Translations, Series 2, Volume 48, pp. 1-35, 1965.

16. L. A. Kaluzhnin
Algorithmization of Mathematic Problems
Problems of Cybernetics, Volume 2, pp. 371-391, 1961.
This reference analyzes the advantages and shortcomings of Markov algorithms.

The following are the basic references on the λ-calculus:

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

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

The following references describe the theory and application of the λ-calculus:

19. Peter J. Landin
A Formal Description of ALGOL 60
Formal Language Description Languages for Computer Programming, North-Holland Publishing Company, Amsterdam, 1966.

20. Peter J. Landin
The λ-Calculus Approach
Advances in Programming and Non-Numerical Computation, Permagon Press, New York, 1966.

21. 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.

22. Christopher Strachey
Towards a Formal Semantics
Formal Language Description Languages for Computer Programming, North-Holland Publishing Company, Amsterdam, 1966.

23. C. Bohm
The CUCH as a Formal and Description Language
Formal Language Description Languages for Computer Programming, North-Holland Publishing Company, Amsterdan, 1966.

24. Arthur Evans, Jr.
Class notes for Linguistic Structures, Subject 6.688, M.I.T., Fall Term, 1966.
These notes are based on class lectures given by Peter Landin.

25. John M. Wozencraft
Class notes for "Programming Linguistics," Subject 6.231, M.I.T., Spring Term, 1968.

26. James H. Morris
Lamda Calculus Models of Programming Languages, Ph.D. dissertation, M.I.T., December 1968.

The following references describe the computer languages SNOBOL/1 and ALGOL/60.

27. David J. Farber, Ralph E. Griswold, and I. P. Polonsky
SNOBOL, A String Manipulating Language
Journal of the ACM, Volume 11, Number 2, pp. 21-30, 1964.

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

The following references have also been used:

29. Peter E. Lauer
The Formal Explicates of the Notion of An Algorithm, Technical Report 25.072, IBM Laboratory Vienna, February, 1967.
This reference explains and relates formalisms (including Post's systems, Markov algorithms, and the λ-calculus) related to the theory of computability.

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

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

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

33. E. V. Detlovs
The Equivalence of Normal Algorithms and Recursive Functions
American Mathematical Society Translations, Series 2, Volume 23, pp. 15-81, 1963.

34. Marvin L. Minsky
Computation: Finite and Infinite Machines, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1967.

35. Noam Chomsky
On Certain Formal Properties of Grammars
Information and Control, Volume 2, Number 4, pp. 393-395, 1959.

36. Alfred B. Manaster
Class notes for "Introduction to Mathematical Logic," Subject 18.886, M.I.T., Spring Term, 1967.

37. Thomas B. Steel, Jr. (Editor)
Formal Language Description Languages for Computer Programming, North-Holland Publishing Company, Amsterdam, 1966.

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

39. Calvin N. Mooers
How Some Fundamental Problems are Treated in the Design of the TRAC Language
Symbol Manipulation Languages Techniques, North-Holland Publishing Company, Amsterdam, 1968.

40. Joseph Weizenbaum
ELIZA - A Computer Program for the Study of Natural Language Communication between Man and Machine
Communications of the ACM, Volume 9, Number 1, pp. 36-45, 1966.

41. Jerome A. Feldman
A Formal Semantics for Computer Languages and its Application to a Compiler-Compiler
Communications of the ACM, Volume 9, Number 1, 1966.

42. -------
A Programmer's Introduction to the IBM System 1360 Architecture, Instructions, and Assembler Language, International Business Machines Corporation, White Plains, New York, 1965.

43. Francis J. Russo
A Heuristic Approach to Alternate Routing in a Job Shop
MAC-TR-19, Project MAC, M.I.T., 1965.

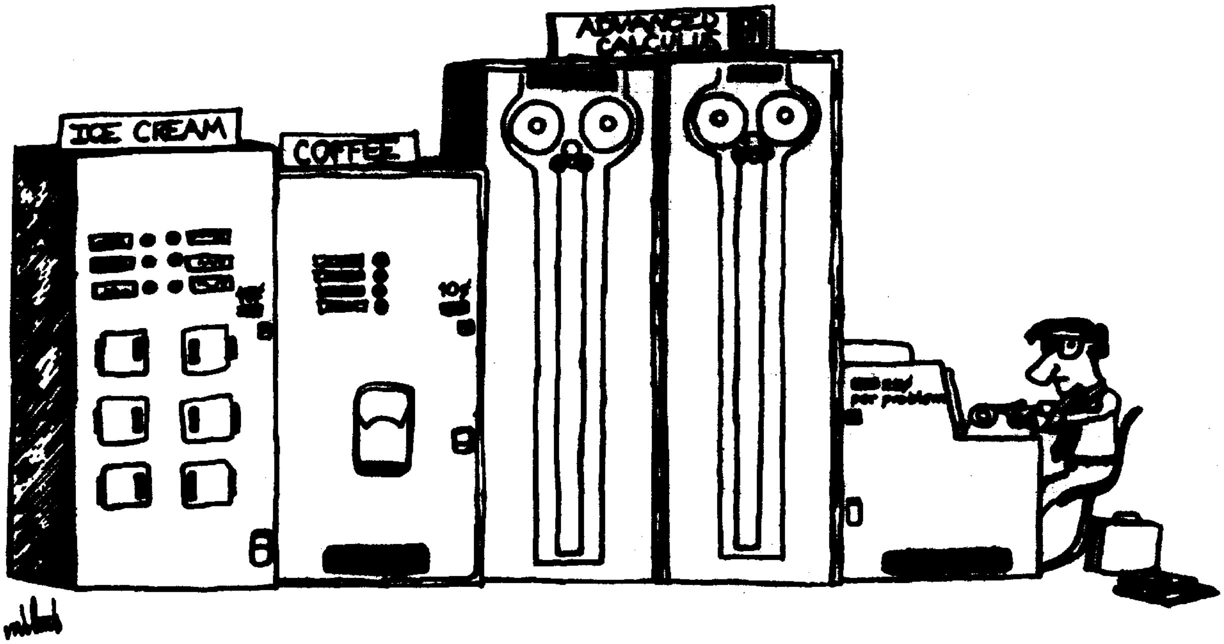
BIOGRAPHICAL NOTE

Henry Francis Ledgard greeted Lowell, Massachusetts, on February 22, 1943. He graduated from Keith Academy of Lowell in 1960 and received a Bachelor of Science degree (magna cum laude) from Tufts University in 1964. While at Tufts, he was elected president of the Tufts Tau Beta Pi chapter, which received the "Outstanding Chapter of the Year Award" in 1963. Honors during his matriculation included the "Amos E. Dolbear Award for Excellence in Electrical Engineering" and the "Award for Outstanding Service to the Tufts Community."

After graduating from Tufts, the author began studies in computer science at Massachusetts Institute of Technology, where he received the degree of Master of Science in 1965 and the degree of Electrical Engineer in 1967. While at M.I.T. the author was associated with Bell Laboratories and Massachusetts General Hospital. In 1965 he became a member of the staff of the Electrical Engineering Department, first as a teaching assistant, and later as a research assistant in which capacity he undertook the research presented in this dissertation.

The author likes northwest days, snow, music, cats, Monhegan Island, politics, working hard, and playing hard.

VENDING MACHINE OF THE FUTURE



DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
None

3. REPORT TITLE
A Formal System for Defining the Syntax and Semantics of Computer Languages

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Ph.D. Thesis, Department of Electrical Engineering, February 1969

5. AUTHOR(S) *(Last name, first name, initial)*
Ledgard, Henry F.

6. REPORT DATE
April 1969

7a. TOTAL NO. OF PAGES
204

7b. NO. OF REFS
43

8a. CONTRACT OR GRANT NO.
Office of Naval Research, Nonr-4102(01)
b. PROJECT NO.
NR-048-189
c.
RR 003-09-01
d.

9a. ORIGINATOR'S REPORT NUMBER(S)
MAC-TR-60 (THESIS)

9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES
This document has been approved for public release and sale;
its distribution is unlimited.

11. SUPPLEMENTARY NOTES
None

12. SPONSORING MILITARY ACTIVITY
Advanced Research Projects Agency
3D-200 Pentagon
Washington, D.C. 20301

13. ABSTRACT

The thesis of this dissertation is that formal definitions of the syntax and semantics of computer languages are needed. This dissertation investigates two candidates for formally defining computer languages:

(1) the formalism of canonical systems for defining the syntax of a computer language and its translation into a target language, and

(2) the formalisms of the λ-calculus and extended Markov algorithms as a combined formalism used as the basis of a target language for defining the semantics of a computer language.

Formal definitions of the syntax and semantics of SNOBOL/1 and ALGOL/60 are included as examples of the approach.

14. KEY WORDS

Computers
Computer languages
Machine-aided cognition
Multiple-access computers
On-line computer
Real-time computers
Syntax and semantics
Time-sharing
Time-shared computers