CS48

AN APPROACH TO COMPUTER LANGUAGE DESIGN

BY

W. M. McKEEMAN

TECHNICAL REPORT NO. CS48

AUGUST 31, 1966

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

AN APPROACH TO COMPUTER LANGUAGE DESIGN

BY

W. M. McKEEMAN

TECHNICAL REPORT NO. CS48

AUGUST 31, 1966

PREPARED UNDER CONTRACT Nonr-225(37) (NR-044-211)
OFFICE OF NAVAL RESEARCH

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

ACKNOWLEDGMENTS

I am well aware that I, as the first student to enter the Computer Science curriculum for the Ph.D. at Stanford, have received more than my share of help and advice. I wish to express my gratitude to Professor Nicklaus Wirth, upon whose work the major part of this paper is based. Professor Wirth has been a patient and gentle critic as well as a constant source of ideas. Professor George Forsythe, who first introduced me to the intricacies of automatic computation, has been extremely generous with his time. Without his counsel, inspiration, and tangible help, I could not have succeeded at Stanford; my debt is immense.

To Professors John McCarthy, William Miller, and Joyce Friedman, for their encouragement and their efforts in reading this thesis, and to my fellow student Rajagopal Reddy who has steadily prodded and encouraged me through the various stages of the Ph.D. program, sharing the tribulations of our pioneer status, I wish to express my thanks.

Among the pleasures of studying at Stanford is the seemingly inexhaustible stream of interesting people to meet with and talk to. I can remember conversations directly applicable to this thesis with at least the following friends: Robert Barton, Larry Breed, Ken Colby, Dave Dahm, Horace Enea, Robert Floyd, Ken Iverson, Cleve Moler, Glen Oliver, John Reynolds, Steve Russell and Harold van Zoeren; I am sure there are some I have forgotten to list.

The Office of Naval Research, under contract Nonr-225(37) (NR-044-211), the National Science Grant GP4053 and the Stanford University Computation Center contributed significant financial support.

And a final note of appreciation for my wife [redacted] whose patient encouragement has been very important.

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

The Goals of Computer Language Design

The universe and its reflection in the ideas of man have wonderfully complex structures. Our ability to comprehend this complexity and perceive an underlying simplicity is intimately bound with our ability to symbolize and communicate our experience. The scientist has been free to extend and invent language whenever old forms became unwieldy or inadequate to express his ideas. His readers however have faced the double task of learning his new language and the new structures he described. There has therefore arisen a natural control: a work of elaborate linguistic inventiveness and meager results will not be widely read.

As the computer scientist represents and manipulates information within a machine, he is simulating to some extent his own mental processes. He must, if he is to make substantial progress, have linguistic constructs capable of communicating arbitrarily complicated information structures and processes to his machine. One might expect the balance between linguistic elaboration and achieved results to be operable. Unfortunately, the computer scientist, before he can obtain his results, must successfully teach his language to one particularly recalcitrant reader: the computer itself. This teaching task, called compiler writing, has been formidable.

Consequently, the computing community has assembled, under the banner of standarization, a considerable movement for the acceptance of a few committee-defined languages for the statement of *all* computer processes. The twin ideals of a common language for programmers and the immediate interchangibility of programs among machines have largely failed to materialize. The main reason for the failure is that programmers, like all scientists before them, have never been wholly satisfied with their heritage of linguistic constructs. We hold that the demand for a fixed standard programming language is the antithesis of a desire for progress in computer science. *That the major responsibility for computer language design should rest with the language user will be our central theme*.

The reduction of compiler writing to a task that a language user might reasonably wish to undertake is the major technical obstacle. We are not alone in our desire to simplify compiler writing [4, 7, 17, 22, 25 and we must justify our particular approach in some detail.

We postulate the existence of a set of basic concepts common to all computing tasks. A language which includes just the basic concepts we will call a *kernel language*. The implementation of a compiler for a kernel language we will call an *extendable compiler*. We do not expect agreement on what constitutes the set of basic concepts or on the best kernel language to represent them. We do hope that our kernel language will be noncontroversial enough that the user will not be seriously hampered in building a language to suit his needs.

Our first claim is that modifying an extendable compiler is easier than building a compiler from first principles. The primary reason for this is that the user of an extendable compiler can largely ignore the details of such mechanisms as text scanning, syntactic analysis and program loading while concentrating on translating his forms (syntax) into his meaning (semantics). In many compiler systems the mechanisms for syntactic and semantic analysis, scanning, building tables and code production are inextricably entwined, making a change to any one of them hazardous, even for the expert. In our extendable compiler such functions are cleanly separated, both conceptually and physically in the text of the compiler program.

Our second claim involves the syntactic description of the user's language. We demand a phrase structure grammar (BNF, Backus-Naur Form, Chomsky type II, context free, etc.) from which a syntax preprocessor generates syntactic recognition tables for physical insertion into the compiler. We can show that if the syntax preprocessor accepts the phrase structure grammar without complaint, then the syntactic analyzer in the compiler will always function correctly. In short, we can prevent even the naive user from blundering into an ambiguous or otherwise ill-defined grammar.

Finally, we claim that the kernel language is a powerful and concise base upon which to build.

Review of the Literature and Summary

We assume (for the moment) the reader is familiar with the notion of a context-free grammar. The central problem in writing a compiler for a language described by a context-free grammar is the construction of an algorithm which will efficiently discover the grammatical structure of an arbitrary input text. And the basic step in a parsing algorithm is the identification of a substring in the text which, when replaced by application of a rewriting rule, brings us closer to goal of an analyzed text.

A string is a candidate for rewriting if it is identical to the right-hand side of a rewriting rule. If two or more candidates for rewriting overlap, then at most one of the rewritings can lead to a correct analysis. In Bounded Context Syntactic Analysis Floyd explores the possibility of making the decision by examining a fixed number of characters to the left and right of the candidate. A grammar for which such a decision is always possible is called of bounded context. Floyd shows that, if we chose the left and right bounds, we can determine if a given grammar is of bounded context, for the chosen bounds. The construction of a parsing algorithm then simply demands the construction of tables for the relevant contexts.

We immediately discover two difficulties. First, straightforward application of the ideas for a practical language results in tables of impractical size. Floyd points out several simplifications based on particular algorithms (such as a left-to-right scan of the text). But the main difficulty is that the amount of table required for the hardest decision is required for all decisions. Second, there are three decisions involved: where is the left end of the candidate, where is the right end, and what may we substitute for it. As might be expected, the bounds for the individual decisions are usually smaller than those of Floyd, resulting in a reduction of the table size.

In Syntactic Analysis and Operator Precedence Floyd presents a particular algorithm for making the parsing decisions. The algorithm is not properly a parsing algorithm since it skips some steps in the analysis thus failing to give the complete structure of the text under consideration. It is on the other hand more efficient for skipping them. The compiler writer must in each case decide whether the analysis provided is sufficiently complete. We also come immediately to face the problem that for some purposes the class of grammars acceptable to the algorithm is too restricted.

In _Euler: A Generalization of Algol 60, and its Formal Definition_, Wirth and Weber modify Floyd's algorithm to remove some of the restrictiveness on the acceptable grammer and also expand it into a proper analysis algorithm. No progress is made in reducing the size of the tables demanded by expanding the context.

In this paper we explore the implications of splitting the parsing decision into its three components. For context bounds of (1,1) the allowed grammars turn out to be identical to those of Wirth and Weber. For bounds of (2,1) for finding the left boundary, bounds of (1,2) for finding the right boundary and (0,0) for choosing the result of the rewriting we find a substantial improvement in the table size but they are still impractically large.

Also in _Euler_ ... we find that not only the form of the language but also the sequence of parsing steps is significant in the design of a compiler. The sequence of steps proceeding strictly from left to right in the text is called the _canonical parse_. The canonical parse turns out to be a natural vehicle for describing the sequence of execution in the compiled program as well as for proving a given class of grammars unambiguous.

In language design we attempt two goals: to present a language simpler and more powerful than Euler, and to make the defining mechanism sufficiently simple so that the language user can change the language to suit his needs.

Our first action is to equate those constructs in other languages that are conceptually similiar but take different forms (switches, procedures and name parameters) (lists, blocks, compound statements, parameter lists, iteration lists). Our second step is to integrate the concept of a list-valued constant into the language structure itself.

We describe the resulting language and compiler in some detail.

SECTION 2

COMPUTER LANGUAGE DEFINITION

Production Grammars

As can be seen by examining Table 1, there is little unanimity among authors regarding the formalisms for the description of production grammars. While our notation adheres closely to the consensus, our readers may wish to refer to the table for a more familiar terminology.

We define three primitive entities: (1) the vocabulary $\underline{V}$, a finite set of elements called symbols, (2) a null string of symbols, Λ and (3) the operation of catenation between strings and symbols denoted by juxtaposition. In terms of the primitive entities we make the following definitions:

$$\underline{V}^* = \{x \mid x = \Lambda \text{ or } (\exists y)(\exists Y), \quad y \in \underline{V}^*, \quad Y \in \underline{V}, \quad x = yY\}$$

is the set of all strings that can be formed from the elements of set $\underline{V}$. Note that we have used lower case latin letters to denote members of $\underline{V}^*$ and upper case latin letters to denote members of $\underline{V}$. This convention is extremely useful and we will adhere to it henceforth, usually without explicit reminder.

Author	vocabulary or alphabet	distinguished symbol	set of terminal symbols	set of nonterminal symbols	strings of symbols	empty string	production arrow	directly produces: one step	produces: zero or more steps	produces: one or more steps	set of productions
Chomsky [3]	V	S	V_T	V_N	Σ	$\mathcal{e}$	$\rightarrow$		$\Rightarrow$		
Eickel & Paul [5,6]	$\mathfrak{A}$	Z		$\mathfrak{T}(\mathfrak{A})$		Λ	::=	$\underset{\circ}{\rightarrow}$ †	$\underset{=}{\Rightarrow}$ †	$\Rightarrow$ †	$\prod$
Ginsburg [11]	V		Σ		$\theta(V)$	$\in$	$\rightarrow$	$\Rightarrow$	$\overset{*}{\Rightarrow}$		
Greibach [12]	$I \cup T$	X	T	I		Λ	$\rightarrow$	$\rightarrow$		$\overset{*}{\rightarrow}$	$\mathcal{P}$
Floyd [7]	V	S	T	NTC	W_V	Λ	$\rightarrow$	$\rightarrow$	$\Rightarrow$		P
Knuth [16]	$I \cup T$	S	T	I	$(I \cup T)^*$	$\in$	$\rightarrow$	$\rightarrow$	$\Rightarrow$	$\Rightarrow$	$\mathcal{D}$
Wirth & Weber [25]	$\mathfrak{V}$	A	$\mathfrak{B}$	$\mathfrak{V}-\mathfrak{B}$	$\mathfrak{V}^*$	Λ	$\rightarrow$	$\dot{\rightarrow}$		$\overset{*}{\Rightarrow}$	Φ
McKeeman	V	G	V_T	V_N	V^*	Λ	$\rightarrow$	$\rightarrow$	$\Rightarrow$	$\rightarrow \Rightarrow$	P

Table 1

A resume of notations used in recent papers on production grammars.

---

† The arrows of Eickel and Paul, like those of Gilbert [10] have the sense of reduction as opposed to the more standard sense of production.

$\ell \to r$, a production, is an ordered pair with both $\ell, r \in \underline{V}^*$. We call ℓ the left of the production, r the right of the production and read the production as ℓ produces r.

$\underline{P}$ is a finite set of productions.

$\underline{V}_L = \{U \mid (\exists x)(\exists y)(\exists z)$ with $yUz \to x$ in $\underline{P}\}$ is the set of symbols on the left in $\underline{P}$.

$\underline{V}_R = \{U \mid (\exists x)(\exists y)(\exists z)$ with $x \to yUz$ in $\underline{P}\}$ is the set of symbols on the right in $\underline{P}$.

$\underline{V}_T = \underline{V}_R - \underline{V}_L$ is the set of terminal symbols.

$\underline{V}_N = \underline{V} - \underline{V}_T$, the complement of $\underline{V}_T$, is the set of nonterminal symbols.

$\underline{V}_G = \underline{V}_L - \underline{V}_R$ is the set of symbols appearing only on the left in productions. We call $\underline{V}_G$ the set of goal symbols. If $\ell \to r$ is in $\underline{P}$, then for any x and y we may write $x\ell y \to xry$ and read $x\ell y$ directly produces xry, or xry directly reduces to $x\ell y$. We immediately note that for every production, the left of the production directly produces the right of the production. We regard each production as a rewriting rule allowing the substitution of the right of the production for any occurrence of the left of the production in any string. If a string is in $\underline{V}_T^*$ then there can be no applicable production and the process of production must halt, hence the name terminal symbols is applied to $\underline{V}_T$.

One may also regard a production as a rewriting rule in the direction opposite to the arrow. In that case the rule would be called a reduction. In simplest terms, we would think of speaking as involving actions of production and listening as involving actions of reduction. It will be convenient to phrase our theorems in terms of productions while our programs are capable only of reductions.

If $y = x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \ldots \rightarrow x_n = z$ for $n \geq 1$, then we write $y \Rightarrow z$ and read y produces z or z reduces to y.

If we write $y \rightarrow\Rightarrow z$ we mean $y \Rightarrow z$ with $n > 1$.

The set $\underline{DS}(\underline{P}) = \{x \mid (\exists G), G \in \underline{V}_G, G \rightarrow\Rightarrow x\}$ is the set of strings derivable in $\underline{P}$.

$\underline{L}(\underline{P}) = \underline{DS}(\underline{P}) \cap \underline{V}_T^*$ is called the language defined by $\underline{P}$. The members of $\underline{L}(\underline{P})$ are called the sentences of the language. Note that it is the sentences that can be written as text and we need be concerned only with the analysis of sentences.

Since a language is fully determined by the set of its productions $\underline{P}$, we will refer to the set of productions as the grammar $\underline{P}$. We lose the generality of being able to select a single member of $\underline{V}_G$ as the distinguished symbol, but the loss does not affect our considerations since we have other reasons to restrict $\underline{V}_G$ to a single unique member.

For example: Let

$\underline{P} = \{G \rightarrow X, X \rightarrow XX, X \rightarrow Y\}$.

Then:

$\underline{V} = \{G, X, Y\}$,

$\underline{V}_T = \{Y\}$,

$\underline{V}_N = \{G, X\}$,

$\underline{V}_G = \{G\}$,

$\underline{V}^* = \{\Lambda, G, X, Y, GG, GX, GY, XG, \cdots \text{etc.}\}$,

$\underline{V}_T^* = \{\Lambda, Y, YY, YYY, \ldots \text{etc.}\}$,

$\underline{L}(\underline{P}) = \{Y, YY, YYY, \ldots \text{etc.}\}$,

$$G \rightarrow X \rightarrow XX \rightarrow XXX \rightarrow XXY \rightarrow XYY \rightarrow YYY$$

is an explicit demonstration of the fact that G produces the string YYY $(G \Rightarrow YYY)$.

We now direct our attention to a subset of production grammars, called <u>phrase structure grammars</u>, in which the form of the productions is restricted to $L \rightarrow r$.[†]

We will also assume two additional restrictions:

(1) $\underline{V}_G$ has a single element; we will designate it by G .

(2) $(\forall X)(\exists t)$ such that $X \in \underline{V}_N$, $t \in \underline{V}_T^*$ and $X \Rightarrow t$.

The alternative to restriction (1) is to distinguish one member of $\underline{V}_G$ explicitly in the description of the language. We reject this for two reasons: First, the productions describing the other members of $\underline{V}_G$ can be discarded since they can never be used in an analysis; Second, we like to be able to test the productions for the existence of a unique goal as a check against programmer errors.

Restriction (2) excludes grammars that give rise to derivations that can never terminate in a sentence. It happens that condition (2) is also required to prove the equivalence of simple precedence grammars and symbol pair grammars (see page 27).

<u>The Canonical Parse</u>

If $xYz \rightarrow xyz$ and $z \in \underline{V}_T^*$, then we call the ordered pair (xYz, xyz) a canonical parsing step (abbreviated CPS). Note that it is the rightmost nonterminal symbol (RNS) that is replaced in a CPS. If every step in $s \Rightarrow t$ is a CPS, we call the sequence of steps a canonical parse. A CPS induces a partition (xyz) on the unreduced text.

[†]Note that we imply $L \in V$ and $r \in V^*$ by our conventions on upper and lower case.

Knuth calls the segment y the handle [16] which unfortunately conflicts with Greibach's term handle [12]. Wirth and Weber [25] call y the leftmost reducible substring which implies a relation that we do not wish to pursue. We will give y the name canonically reducible string and abbreviate it CRS. For a particular CPS, the CRS is well defined.

If we view the CPS in the sense of production, we see that zero or more symbols are added to the terminal string to the right of the rightmost nonterminal symbol. Therefore the length of the string z of terminal symbols is a monotonic function of the number of canonical parsing steps. Now viewed as a reduction, we see that the canonical parse inforces exactly the same order of productions as required by a left-to-right scan of the sentence.

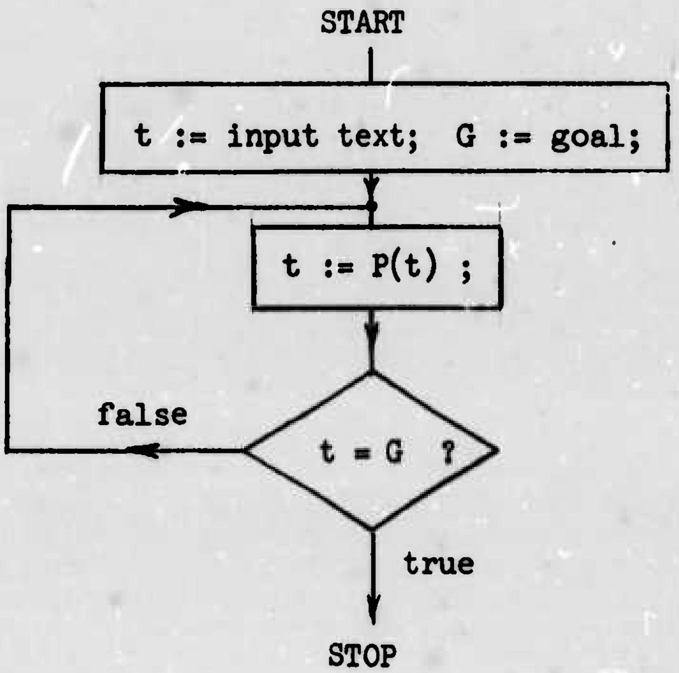
Because of its relation to left-to-right parsing algorithms, the concept of a canonical parse has appeared in many forms. It was first explicitly named in [5] and [25] independently.

A sentence which has two essentially different structures is called ambiguous. Formally, a sentence is unambiguous if and only if it has a unique canonical parse. Furthermore; a language containing an ambiguous sentence is ambiguous; a grammar defining an ambiguous language is ambiguous.

The reader should verify that the grammar, language and sentence in the preceding example are formally ambiguous according to our definition.

The Parsing Function

The problem of parsing a text t reduces to finding, at each stage, the string t_i so that $t_i \to t_{i-1}$ is a CPS. If a sentence t is unambiguous then we see immediately that each intermediate stage of its derivation is unambiguous. In particular, we note that for all i, t_i is uniquely determined by t_{i-1} alone.† We can therefore infer the existence of a uniquely valued parsing function P such that $P(t_{i-1}) = t_i$. The following algorithm is the complete solution to the problem of parsing an unambiguous sentence.



The assured existence of the function P is, however, of little use in constructing a translator. The only way to compute its values in general is to parse the sentence t and record the results in a table (which rather begs the question).

† For otherwise we would have two canonical parses of t .

It is surprising to find that for a restricted set of phrase structure grammars, we can find economical ways of computing the parsing function. Two [7,25] have been previously published. A third way, and some steps toward a fourth are presented below. Except that Floyd's algorithm skips some CPS, all are special cases of the following detailed breakdown of an algorithm to compute the function P .

P1, P2, and P3 are functions of three string-valued variables $\underline{x}$, $\underline{y}$ and $\underline{z}$. For the moment we will underline program variables to distinguish them from values with the same name but derived from the canonical parse. If the catenation $\underline{xyz}$ is in $\underline{DS}(\underline{P})$ and $\underline{L}(\underline{P})$ is unambiguous then there is a unique partition $xyz = \underline{xyz}$ of the catenation of strings in the program variables $\underline{x}$, $\underline{y}$ and $\underline{z}$ and a unique production $Y \rightarrow y$ in $\underline{P}$ such that $G \Rightarrow xYz \rightarrow xyz$ is canonical. We give an Algol-like definition of the functions in terms of the partition and production as follows:

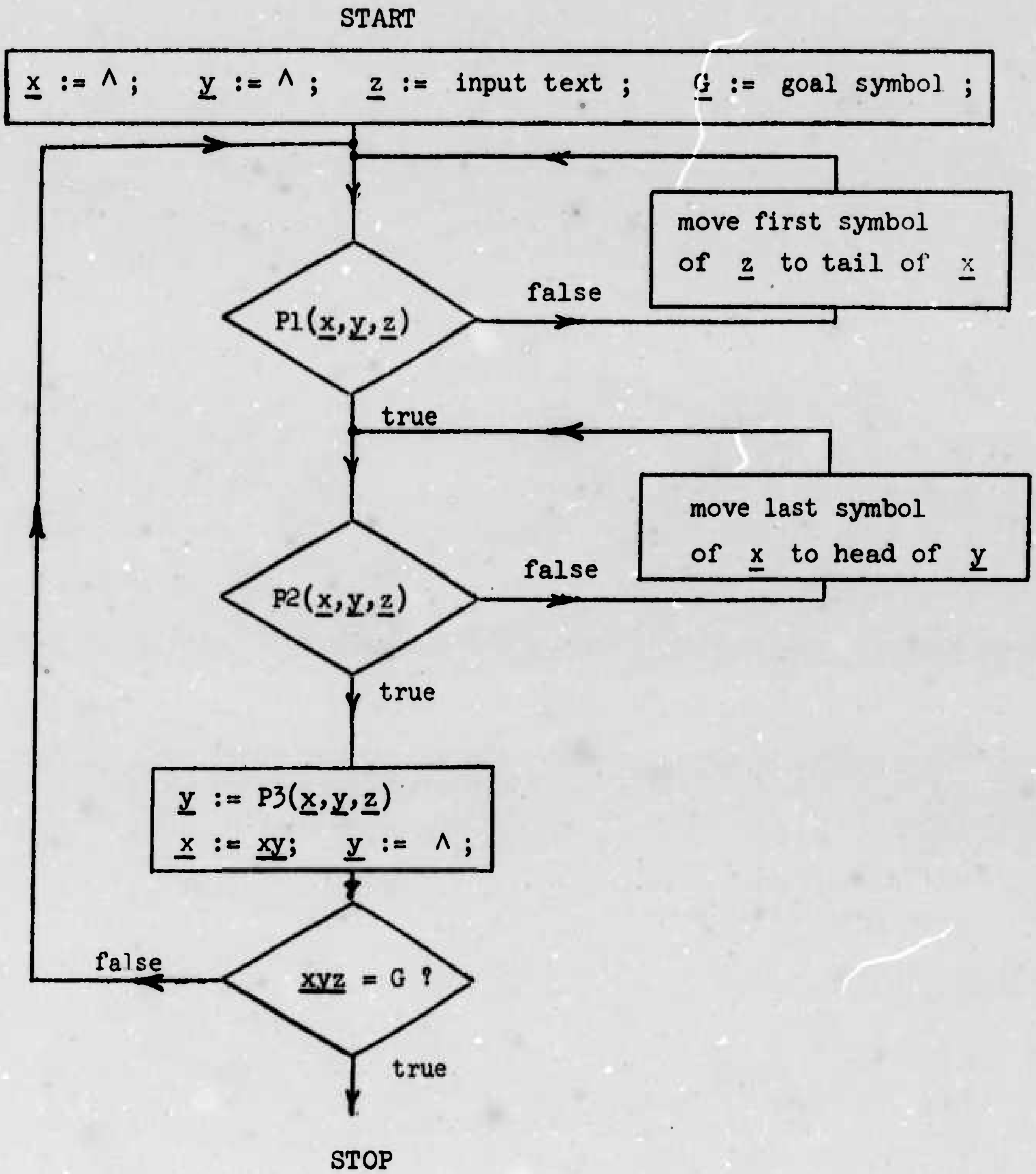
P1($\underline{x}$,$\underline{y}$,$\underline{z}$): $\underline{\text{If}}$ $G \rightarrow\Rightarrow \underline{xyz}$ $\underline{\text{then}}$
($\underline{x} = xy$ $\underline{\text{and}}$ $\underline{y} = \Lambda$ $\underline{\text{and}}$ $\underline{x} = z$) $\underline{\text{else}}$ undefined;

P2($\underline{x}$,$\underline{y}$,$\underline{z}$): $\underline{\text{If}}$ $G \rightarrow\Rightarrow \underline{xyz}$ $\underline{\text{then}}$
($\underline{x} = x$ $\underline{\text{and}}$ $\underline{y} = y$ $\underline{\text{and}}$ $\underline{z} = z$) $\underline{\text{else}}$ undefined;

P3($\underline{x}$,$\underline{y}$,$\underline{z}$): $\underline{\text{If}}$ P2($\underline{x}$,$\underline{y}$,$\underline{z}$) $\underline{\text{then}}$ Y $\underline{\text{else}}$ undefined.

The general parsing algorithm.

thm



In terms of a syntactic analysis algorithm, we would assign the following individual responsibilities to the functions:

P1: read the input tape..

P2: locate the CRS y to be replaced.

P3: perform the reduction.

Due to the monotonicity of the length of z, we must decide before each CPS whether to shorten z. At the termination of the loop on P1, we have assured ourselves that all of the CRS is on the tail of x. We have located one boundary of y. The left boundary is found in the loop on P2. At the termination of the larger loop, we substitute Y for y, leaving the nonterminal symbol Y on the tail of x. If we have reduced the entire string to the goal we are through. Otherwise, we return to the loop on P1.

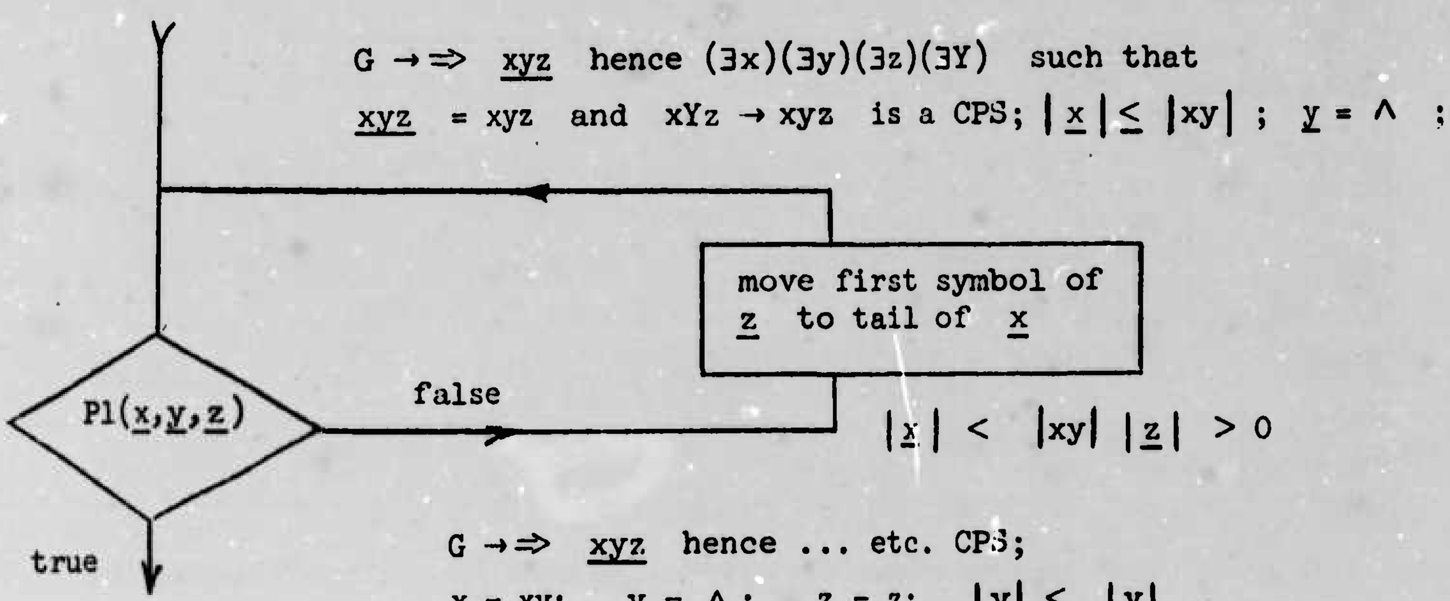
A cycle through the functions P1, P2, and P3 is equivalent to a single step on the function P. The string xyz is always identical, at the end of the main cycle, to the value of P(xyz). The main reason for introducing the function P1, P2, and P3 is that their values can be handled as reasonable computational entities. The parameters of the functions are still unwieldy which reflects the fact that the function values may depend upon an examination of the entire text.

Theorem. If the input text is a sentence and the grammar is unambiguous, the general parsing algorithm will reduce the input text to the goal symbol via the canonical parse.

Before attempting the proof we must describe our general method for proving the correctness of algorithms. The basic mechanism of inductive closure for program loops is described by Floyd [9] as one technique of a verifying compiler. We state an initial set of relations that we know to be true upon entry to the loop. We then show that they are invariant with respect to execution of the loop, hence they are always true. We finally deduce some relations that are true at the completion of the loop, either as a final result, or as a component in a proof on a larger enclosing loop. In our deductions we must insure that all actions are defined and all loops terminate. Relations true upon exit from the algorithm are then correct descriptions of the final state of the algorithm.

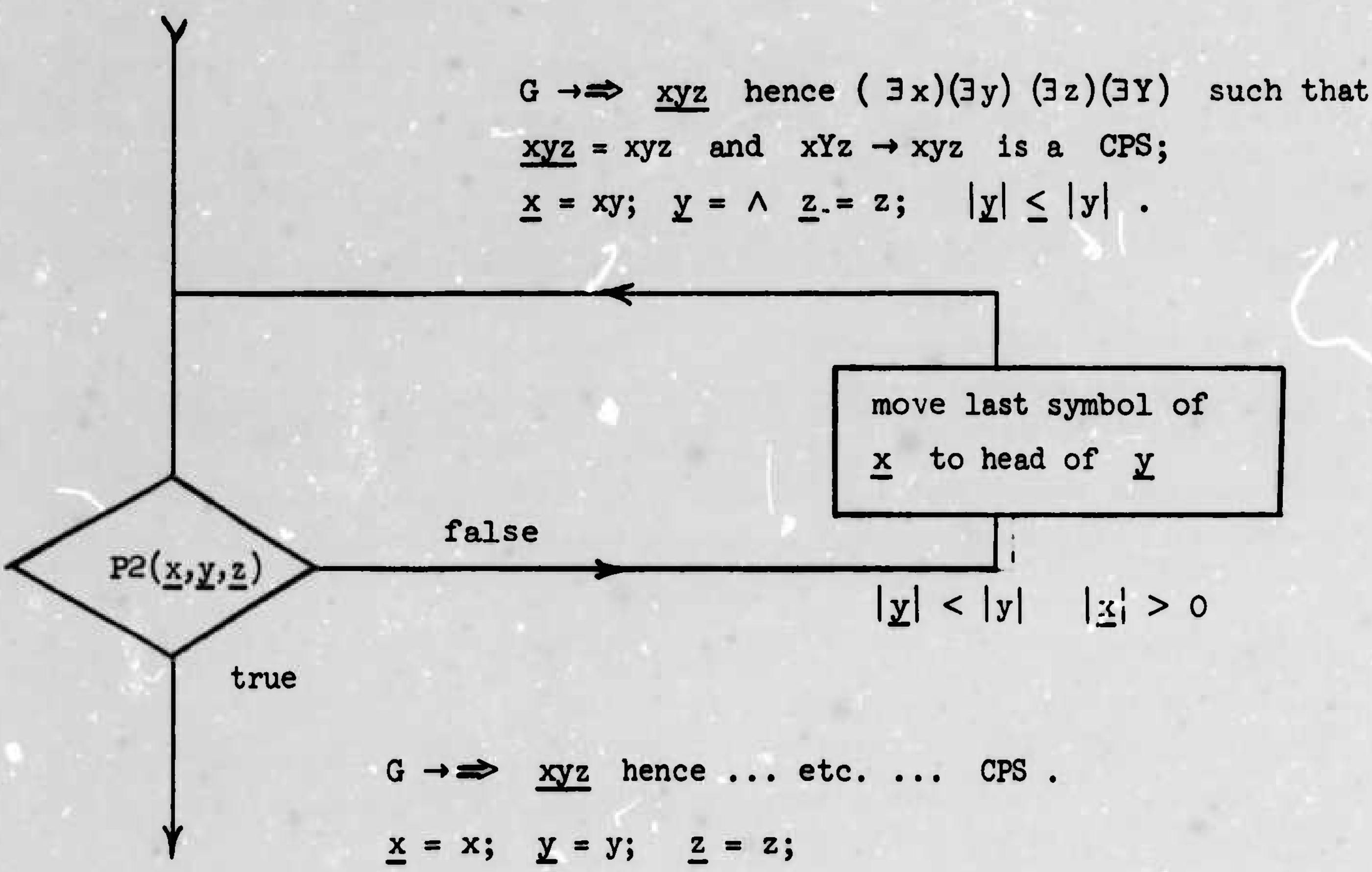
<u>Proof</u>. If the grammar is ambiguous, the parsing functions are not uniquely defined and it is meaningless to state the parsing algorithm. Similiarly, if the input text is not a sentence, all of P1, P2, and P3 are immediately undefined.

We need to show that we complete one CPS each time through the outer loop and that the process terminates in a finite number of steps. We cannot analyze the outer loop until we understand the inner loops. We consider the loop on P1 first.



We assume the truth of the relations listed at the top of the loop and derive those at the bottom. Since $G \rightarrow\!\!\Rightarrow \underline{xyz}$, P1 is defined. If it is false, $\underline{x} \neq xy$. But we have $|\underline{x}| \leq |xy|$ hence we derive $|\underline{x}| < |xy|$. From $|\underline{y}| = 0$ we get $|\underline{xy}| < |xy| \leq |xyz|$ hence $|\underline{xy}| < |xyz|$. But $\underline{xyz} = xyz$ thus $|\underline{z}| > 0$. There is, therefore, at least one character in $\underline{z}$ and the action in the box is defined. Furthermore, all of the assumptions are unaffected by the action, hence are invariants of the loop. The loop must terminate because $\underline{z}$ is of finite length. When P1 becomes true, the conditions on exit from the loop are consequences of the definition of P1.

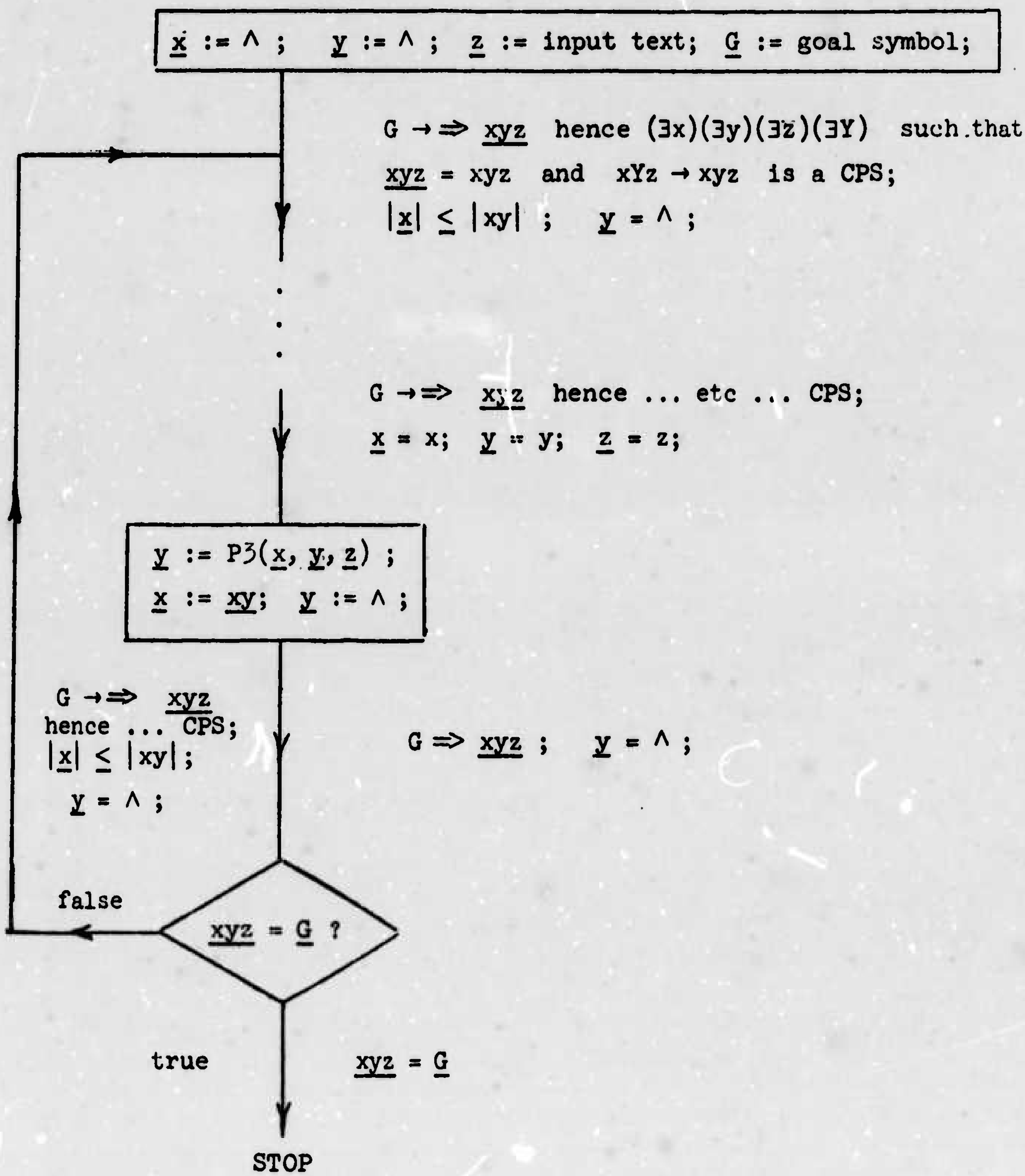
Now consider the loop on P2 with the results of P1 as assumptions.



P2 is initially defined and will remain so. Since $\underline{z}$ is never affected, we have $\underline{z} = z$ everywhere. If P2 is false we have either $\underline{x} \neq x$ or $\underline{y} \neq y$. But either inequality implies the other, so we have both.

From $|\underline{y}| \geq |y|$ we derive $|\underline{y}| < |y|$, hence $|\underline{x}| > 0$. Therefore the action in the box is defined. All the assumptions are preserved in the loop. The loop must terminate because $\underline{x}$ is of finite length yielding the stated relations as consequence of the definition of P2.

For the entire algorithm we can now write



By our assumptions, the input text is a sentence and we have $G \rightarrow\Rightarrow \underline{xyz}$ and its ramifications. Since $|\underline{x}| = 0$ initially, $|\underline{x}| \leq |xy|$ is vacuously true. P3 is defined and has value Y. $\underline{x}Y\underline{z} \rightarrow \underline{xyz}$ is a CPS by definition hence we have new $\underline{x} = xY$, $\underline{y} = \Lambda$, and $\underline{z} = z$ with $G \Rightarrow \underline{xyz}$. If $G = \underline{xyz}$, we are done. Otherwise, we may write again $G \rightarrow\Rightarrow \underline{xyz}$ and define new x, y, and z. Since $z \in \underline{V}_T^*$, xy must contain all of the nonterminal symbols. The last symbol of the new $\underline{x}$ is nonterminal, giving the required $|\underline{x}| \leq |xy|$. We find our assumptions invariant and also a consequence of the initial conditions. The loop must terminate since there are a finite number of steps in a canonical parse. QED.

Symbol Pair Parsing Functions

If we wish to find a reasonably efficient method for computing the parsing functions, we must renounce the privilege of examining the entire text at each stage. We will see that the effect of narrowing the view of the parsing functions will be to reduce the class of grammars for which we can build mechanical translators.

We first postulate that the parsing functions depend only upon a few symbols in the region of the CRS. We will be able to verify our postulate mechanically; if it is false then the grammar in question lies outside the range of that particular analysis.

Our approach will be to examine the grammar (mechanically, as it is very tedious) to discover all the sequences of symbols that can possibly occur in the region of the next CRS. For each possible sequence we will record the required value of the parsing functions. When the resulting functions are well defined the grammar is unambiguous and the syntactic analysis algorithm in the compiler always functions correctly. The function values are inserted into the compiler in a condensed tabulated form.

Consider the three new functions P1', P2', and P3' defined in terms of P1, P2, and P3.

If P1($\underline{x}$,$\underline{y}$,$\underline{z}$) is defined, X is the last symbol of $\underline{x}$ and Z is the first symbol of $\underline{z}$, then we define P1'(X,Z) to be identical to P1($\underline{x}$,$\underline{y}$,$\underline{z}$). Similiarly, P2'(X,Z) must be identical to P2($\underline{x}$,$\underline{y}$,$\underline{z}$) when P2 is defined, X is the last symbol of $\underline{x}$ and Z is the first symbol of the catenation $\underline{yz}$. P3'($\underline{y}$) must be identical to P3($\underline{x}$,$\underline{y}$,$\underline{z}$) when P3 is defined. We will call a grammar for which the functions P1', P2', and P3' are well defined a <u>symbol pair grammar</u> (or more generally, as we will see, a (1,1)(1,1) canonical parse grammar). We will be able to show that under restrictions (1) and (2) (page 11), symbol pair grammars are equivalent to simple precedence grammars [25].

The number of arguments for which P1, P2, and P3 are defined is in general, infinite. On the other hand, if NSY is the number of symbols in V, P1' and P2' need be defined for at most NSY squared possible arguments. P3' is defined only when y is the right part of a production and thus also has a finite number of possible arguments. It is immediately clear that we must apply a new restriction in order to make P3' well defined:

Restriction (3): No two productions may have equal right parts.

We may, as has been pointed out to the author by N. Wirth, lift restriction (3) if we have any way of distinguishing equal right parts. A particular case in point is the Algol 60 <identifier> which we might wish to reduce to <array identifier>, or to <variable>, etc., where the decision can be made due to other non-grammatical information. We will call the number of productions, (and, under restriction (3), the number of CRS) NPR.

We see that the boundaries between x and y and between y and z in the general parsing algorithm always lie immediately to the left of, within, or immediately to the right of the next CRS. The parameters X and Z of P1' and P2' always lie on opposite sides of one of the boundaries; the values of P1' and P2' depend upon where the boundaries lie with respect to the CRS. We will be able to compute the position of the boundaries with the help of the three following set definitions:

$TS(X)$, the set of <u>tail symbols</u> of X, is given by
$\{Y | (\exists y), X \rightarrow\Rightarrow yY\}$.

$HS(X)$, the set of <u>head symbols</u> of X is given by
$\{Y | (\exists y), X \rightarrow\Rightarrow Yy\}$.

$HS_T(X)$, the set of <u>terminal head symbols</u> is given by
$(HS(X) \cup \{X\}) \cap \underline{V}_T$.

Note that if X is terminal, the first two sets are null but the third is not.

When X is a tail symbol of the rightmost symbol in a CRS and Z is a head symbol of anything that might follow that CRS in a sentence, $Pl'(X,Z)$ must be true and never otherwise. Similiarly, whenever X lies within a CRS and Z is a head symbol of the next symbol needed toward the completion of that CRS, $Pl'(X,Z)$ must be false so that the needed symbol is moved onto $\underline{x}$. In terms of a production:

$W \rightarrow uUVv$,

we cannot start building V if U has not yet been fully formed. Since we have narrowed our view to one symbol on either side of the boundary, we <u>must never</u> move any symbol in the head of V from $\underline{z}$ to $\underline{x}$ if the last symbol of $\underline{x}$ is a tail symbol of U. If U has been formed and is the last symbol of $\underline{x}$, we <u>must</u> move any head symbol of V onto $\underline{x}$ to start building toward V and finally $uUVv$. We may very well find conflicting demands, a symbol that must be moved on account of one production and must not be moved on account of another production. Conflicts are common in practice and constitute a serious nuisance. The compiler writer can usually modify his grammar in a trivial way to remove the conflict. A more general solution would be to extend the view of the parsing functions, an approach which is discussed later in this section.

The function of the loop on P2' is to march down across a given CRS and locate its left boundary. In terms of the sample production, it is clear that P2'(U,V) must be false for every pair U,V that are contiguous within a CRS. P2'(X,Z) must be true whenever we cross the left boundary of a CRS--a condition that is true when X lies within a CRS and Z a head symbol of the next item to be formed within that CRS. We can summarize these relations with a mnemonic table:

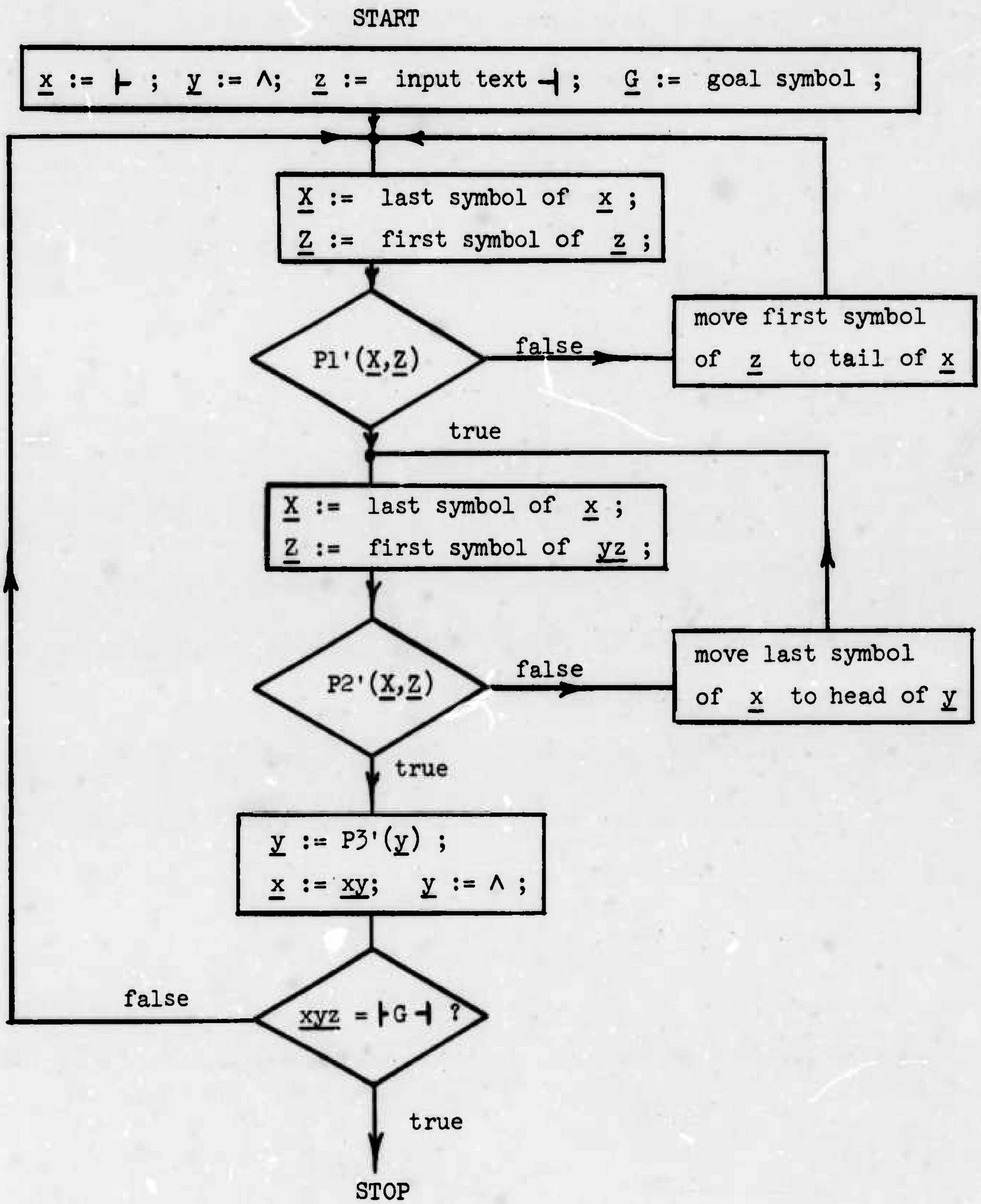
<table>
<tr><td colspan="2" align="center">W → uUVv</td></tr>
<tr><td>$P1'(U, HS_T(V))$ = false
P2'(U,V) = false</td><td>$P1'(TS(U), HS_T(V))$ = true
P2'(U,HS(V)) = true</td></tr>
<tr><td colspan="2" align="center">P3'(uUVv) = W</td></tr>
</table>

We need only consider terminal symbols for P1' since we know that z contains only terminal symbols. We are also implicitly assuming some strings to be non-empty. We avoid this last problem by adding a production leading to the goal symbol,

$G' \rightarrow \vdash G \dashv$, where $\vdash$ and $\dashv$ are end-of-file symbols

that we may use to initialize x and append to z. As modified the parsing algorithm becomes:

The Symbol Pair Parsing Algorithm



We state some consequences of the definition of symbol pair grammars.

<u>Theorem</u>. If the symbol pair parsing algorithm terminates normally, it has produced the canonical parse for the input text.

<u>Proof</u>. The only transformation allowed on the text is the substitution of the leftpart of a production for the rightpart, thus it is immediately obvious that if the algorithm functions at all, it produces a parse. After each substitution we see that the newly formed reduced symbol is the rightmost nonterminal symbol in the text, hence that step was a CPS. QED.

<u>Theorem</u>. A symbol pair grammar is unambiguous. ([25] p. 26).

<u>Proof</u>. Assume the contrary. Then there is a sentence for which there exist two canonical parses. We first show that the existence of two different overlapping CRS implies a conflict in the parsing functions. Assume that our text is

$$xL_1L_2\ldots L_kM_1M_2\ldots M_mR_1R_2\ldots R_nz$$

and both of $L_1\ldots M_m$ and $M_1\ldots R_n$ are CRS with $m > 0$, and one of k or $n > 0$.

We treat the case $k > 0$ in detail. From the fact that $L_1\ldots M_m$ is a CRS we immediately derive $P_1'(L_k, HS_T(M_1)) = \underline{false}$ and $P_2'(L_k, M_1) = \underline{false}$.

(We substitute the set as an argument of P_1' meaning the relation is true for all members of that set). Now perform the rightmost reduction and our text becomes

$$xL_1L_2\ldots L_kMz$$

where M was the leftpart of the production. Either L_k and M are next to each other in a production or further reduction brings us to the text

$$x'L'M'z'$$

where L' and M' are next to each other in a production,

$L' \Rightarrow uL_k$, $M' \Rightarrow M_l v$.

In the first case we have $P2'(L_k, M_l) = \underline{true}$ and in the second, $P_1'(L_k, HS_T(M')) = \underline{true}$. Either implies a conflict since $M_l \in HS(M')$ and HS_T is never empty.

The situation is entirely similiar for $n > 0$. Thus we find our only choice during reduction is which of several disjoint CRS to pick. Let us assume that we pick other than the leftmost, substituting for it the nonterminal symbol in the leftpart of its production. There is a CRS to the left which must always be disjoint from all other CRS, hence will eventually be reduced to its leftpart. But such a step is not a CPS because we have already formed a nonterminal symbol to its right. In order to form the canonical parse, we must always pick the leftmost CRS and it is unique, thus the canonical parse is unique and the grammar is unambiguous.

We will now define the simple precedence grammars of Wirth and Weber and show their equivalence, under restriction (2), to symbol pair grammars. We define three relations, $\lessdot$, $\doteq$, $\gtrdot$, between symbol pairs as follows:

For every production of the form $W \rightarrow uUVv$

$U \doteq V$,

$Z \in HS(V)$ implies $U \lessdot Z$

$X \in TS(U)$ implies $X \gtrdot V$

$X \in TS(U)$ and $Z \in HS(V)$ imply $X \gtrdot Z$.

If for each pair of symbols in $\underline{V}$ at most one of the above relations holds, the grammar is a simple precedence grammar.

__Theorem__. If $\underline{P}$ is a simple precedence grammar, then $\underline{P}$ is a symbol pair grammar. If $\underline{P}$ is a symbol pair grammar and restriction (2) holds, then $\underline{P}$ is a precedence grammar. We immediately exhibit a symbol pair grammar that violates restriction (2) and thus fails to be a simple precedence grammar.

$$\underline{P} = \{G \to AB,\ \ A \to X,\ \ A \to XB,\ \ B \to C,\ \ C \to CY\} \ .$$

The reader may find it instructive to build the six by six matrix of precedence relations implied by the definition and find the two conflicts, one of which is $X \lessdot C$ and $X \gtrdot C$.

__Proof__. We show that if $\underline{P}$ is not a simple precedence grammar then it is not a symbol pair grammar and the converse.

Assume that $\underline{P}$ is not a simple precedence grammar. Then there exist at least two symbols related by at least two of the three relations $\lessdot$, $\doteq$, $\gtrdot$. We treat each case separately.

(a). $U \doteq V$ implies $(\exists W)(\exists u)(\exists v)$ such that $W \to uUVv$.

(b). $U \lessdot V$ implies $(\exists W)(\exists u)(\exists S)(\exists v)$ such that $W \to uUSv$ and $V \in HS(S)$.

(c). $U \gtrdot V$ implies $(\exists W)(\exists u)(\exists R)(\exists v)$ such that $W \to uRVv$ with $U \in TS(R)$, or

$(\exists W)(\exists u)(\exists R)(\exists S)(\exists v)$ such that

$W \to uRSv$ with $U \in TS(R)$ and $V \in HS(S)$.

From the existence of a relation between two symbols we have been able to infer the existence of the production from which the relation was derived. Now from the productions we can derive some values for the functions P1' and P2'.

(a) implies $P2'(U,V)$ is false and $(\forall X)$, $X \in HS_T(V)$ gives $P1'(U,X)$ is false.

(b) implies $P2'(U,V)$ is true and $(\forall X)$, $X \in HS_T(S)$ gives $P1'(U,X)$ is false, Also $HS_T(V) \subset HS_T(S)$.

(c) implies $(\forall X)\ X \in HS_T(V)$ gives $P1'(U,X) = \text{true}$ since $U \in TS(R)$ and $HS_T(V) \subset HS_T(S)$.

On account of restriction (2), we see that HS_T is always nonempty. Therefore if any two of (a), (b) or (c) hold simultaneously, we have a conflict in P1' or P2'; hence <u>P</u> is not a symbol pair grammar.

<u>Converse</u>. Assume that <u>P</u> is not a symbol pair grammar. Then there exist symbols U and V for which either P1' or P2' is double valued.

(d). $P1'(U,V)$ is true implies $(\exists W)(\exists u)(\exists R)(\exists S)(\exists v)$ such that $W \rightarrow uRSv$ with $U \in TS(R)$ and $V \in HS_T(S)$.

(e). $P1'(U,V)$ is false implies $(\exists W)(\exists u)(\exists S)(\exists v)$ such that $W \rightarrow uUSv$ with $V \in HS_T(S)$.

Now $V \in HS_T(S)$ implies $V \in HS(S)$ or $V = S$, thus (e) implies $U \doteq V$ or $U \lessdot V$ and (g) implies $U \gtrdot V$, conflict.

(f). $P2'(U,V)$ is true implies $(\exists W)(\exists u)(\exists S)(\exists v)$ such that $W \rightarrow uUSv$ with $V \in HS(S)$.

(g). $P2'(U,V)$ is false implies $(\exists W)(\exists u)(\exists v)$ such that $W \rightarrow uUVv$.

But (g) implies $U \doteq V$ and (f) implies $U \lessdot V$. Conflict, QED.

In terms of the general parsing algorithm, the precedence relations can be thought of as a three valued function P12'(X,Z) which is used for both analysis loops. Replacing P1', it is false if it has value $<\!\cdot$ or $\doteq$ and true if $\cdot\!>$. Replacing P2', it is false if $\doteq$ and true otherwise. ([25] p. 20). It is surprising to find that even though the defining matrix for P12' is twice as dense as corresponding matrices for P1' and P2' and also contains spurious relations due to the over-restrictive fourth defining rule for simple precedence grammars, that no extra conflicts are introduced.

In either case, the matrices defining the parsing functions turn out to be rather sparse, and rather large. In the process of building the parsing functions, we tabulate the symbols of $\underline{V}$, and manipulate instead the integer corresponding to their symbol table location. As suggested by Floyd ([7] p. 323) we can frequently find functions f1 and g1 such that if P1'(U,V) is true, f1(U) > g1(V) and if P1'(U,V) is false, f1(U) < g1(V).

We can, of course, do the same for P2'. The advantage accrues in requiring only 4 NSY memory locations for the tables defining the functions f1, g1, f2, and g2 instead of $2\ NSY^2$ locations required for the matrices explicitly defining P1' and P2'. This is somewhat offset by the fact that the Boolean matrices defining P1' and P2' could be packed in digital memory. At present, all syntax checking is done by the function P3' and the only error indication is that the CRS found is not in the production table. If we retained the functions P1' and P2' including the undefined values, we would have an additional (redundant) method of error checking.

Let P be an arbitrary Boolean matrix (values 0 and 1). For all X and Y, define

$$f(X) = \sum_{Y=1}^{NSY} 2^{(Y-1)} P(X,Y), \quad g(Y) = 2^{Y} \quad .$$

Then $P(X,Y) = 1$ if and only if $(f(X) \bmod g(Y)) \geq g(Y)/2$. Thus we can state that a relation always exists with which we can record the content of a Boolean matrix P in two linear arrays. The relations "$\leq$" and "$>$" are adequate in practice.

We present the symbol pair syntax preprocessor in two forms. The first is written in the kernel language presented in Section 3, the second is the listing of the Burroughs B5500 Algol program actually used to generate tables for the extendable compiler of Section 4. We find it informative to compare the programs for conciseness and readability. While the two programs accomplish essentially the same actions, the kernel language version is approximately one half as long as the Algol version. A detailed inspection of the program text reveals that the major savings are in implicit table lookups ($\in$, $\notin$, <u>index</u>) and the generalized <u>for</u> loop. In particular, there are 26 occurences of the symbol <u>for</u> in the kernel language version while the Algol version contains 35. Furthermore, we find ten labels in the Algol version of which perhaps one half are essential and none of which contribute to the reader's ability to understand the program.

Since the kernel language is discussed in detail in Section 3, we will say nothing further about it here. Burroughs B5500 Algol is in most respects exactly Algol 60. The input and output conventions are relatively standard except for the following features:

(1) On line 7 of the program we see a file declaration for the card punch. Its function, setting aside buffer areas for the card punch, is not important to an understanding of the program.

(2) Three lines below we find a WRITE statement in the form of a procedure call. The first parameter to WRITE is a format which is indicated to the Algol compiler by enclosing the format in the brackets < and >. All the remaining parameters are values to be written.

In the middle of the third page we see two STREAM procedures. They are an interface with the character mode machine instructions of the B5500 used to set and interrogate two-bit fields within the 48 bit B5500 word. Since we may have upwards of 100 symbols and have two matrices with that number squared of elements, packing the values is unavoidable in Stanford's 16 thousand word B5500 memory. Packing would be somewhat more convenient in the kernel language since we can use subscripts to access bit strings directly.

Finally, we use the machine clock to obtain execution time information for the user. One of our objectives is the accumulation of precise timing information for the behavior of the preprocessor as a function of the number of productions and number of symbols. Preliminary data gives the surprising conclusion that execution time is a linear function of the number of productions (about 2 seconds per production).

We now give a narrative of the kernel language version of the program. Our first action is to name all the identifiers local to the main block and initialize P to the null set. We examine the first character from the input medium and continue to read productions until an end-of-file symbol is encountered. Our productions are character strings whose length is a multiple of 12. The first 12 characters are the leftpart of the production and the remaining fields are the symbols of the rightpart. A carriage return delimits the production. Internally, a production is an ordered set of strings, each element representing one symbol in the production. We make special provision (if (<u>length</u> t)$\neq$ 0 <u>then</u> ...) for blank lines which can be used to increase the readability of the production tables. We also print the productions to supply the user with a record of his input.

If the leftpart of two successive productions is the same, we allow the user to substitute a field of twelve blanks for the second leftpart, again to increase readability. At the completion of input we immediately repair the omission.

Then, in three lines, we use the generalized for loop, set union and set difference to build all the symbol tables that we will need. Four more lines of program records them on the output medium.

After excluding the possibilities of empty and repeated rightparts, it becomes advantageous to replace the production table with a new table "PR" of identical format except that its elements are the indices of the production symbols in the vocabulary V. We then complete our grammar checks by excluding the possibility of a grammar with nonterminating phrases (restriction 2).

We define procedures to compute head and tail symbols. Note that we recompute the head and tail symbols repeatedly within the analysis loop. In the processor for the (2,1)(1,2) grammars we adopt a suggestion of N. Wirth to compute an "occurence" matrix which need not be re-evaluated. The latter is probably a superior approach.

We then initialize the matrices P1 and P2 to NSY squared values undefined, and proceed to evaluate the functions P1' and P2' according to the directions of the theory. For every pair of adjacent symbols in the grammar (j and k in the program) we evaluate the tail and head symbols. We record P2'(j,k) true and all of P2'(j,HS(k)) false. Then we modify heads to become the set HS_T and evaluate P1' in the same manner.

Our final task is the computation of Floyd's linearization functions f and g. Our algorithm is modeled on that of N. Wirth [26] but is simpler since our matrices are two valued instead of three valued. Our algorithm proceeds to satisfy the requirements of the decision function starting in the upper left corner of its defining matrix. We add a row to the satisfied area (null to begin with) and call uprow to assure that (1) f is large enough to satisfy all the requirements given by the value false and (2) g is large enough to satisfy all the requirements given by the value true. If we must change g we call upcol to readjust that entire column.

It is possible to have functions P1' and P2' but still not have a linearization for the relation pair $\leq$ and $>$. At any given stage of the operation of the algorithm above, we know that the submatrix in the upper left corner has been correctly linearized. Thus if we are going to fail, the failure must involve one of the last relations added to consideration. We can check within the adjusting procedures to see that we never return to adjust one of the last relations added. If we do, we have failed and print a diagnostic error trace indicating the exact reason for that particular failure.

```
' Kernel language version of (1,1)(1,1) syntax preprocessor '
{ new j k kl P PR NPR V NSY VL VR VT VN VG P1 P2 f g t heads tails
    fail beenatrowk beenatcolk HS TS upcol uprow change,
    P ← {}, ' the null set of productions '
    while in[1] ≠ eof do
    { t ← {}, ' the input loop, build a production '
        while in[1] ≠ cr do
        { t ← t ⊕ { in[1 to 12] }, ' fixed field, 12 characters '
            in ← in[13 to length in]
        },
        in ← in[2 to length in],
        if (length t) ≠ 0 then P ← P ⊕ {t}, ' add another production '
        out ← out ⊕ (⊕/t) ⊕ cr ' print the production '
    },
    NPR ← length P,  VL ← VR ← set {},
    for all i from 1 to NPR do  'replace omitted left parts'
        (if P[i][1] = "             " then P[i][1] ← P[i-1][1]),
    for all t from P do
        { VL ← VL ∪ {t[1]},  VR ← VR ∪ t[2 to length t]},
    V ← VL ∪ VR,  VT ← VR ⊖ VL,  VN ← V ⊖ VT, VG ← VL ⊖ VR,
    NSY ← length V,
    out ← out ⊕ (if (length VG) ≠ 1 then "no " else "") ⊕
        "Unique leftmost symbol: " ⊕ (⊕/VG) ⊕ cr ⊕
        "Terminal symbols: " ⊕ (⊕/VT) ⊕ cr ⊕
        "Non terminal symbols: " ⊕ (⊕/VN) ⊕ cr,
    for all t from P do (if (length t) = 1 then
        out ← out ⊕ t[1] ⊕ " has an empty right part" ⊕ cr),
    for all i from 1 to NPR do for all j from i+1 to NPR do
        (if P[i][2 to ∞] = P[j][2 to ∞] then
            out ← out ⊕ "Productions " ⊕ cr ⊕
            (⊕/P[i]) ⊕ "      and" ⊕ cr ⊕ (⊕/P[j]) ⊕ cr ⊕
            "have equal right parts" ⊕ cr),
    PR ← P, 'convert productions strings to symbol table location'
    for all i from 1 to NPR do for all j from 1 to length P[i] do
        PR[i][j] ← P[i][j] index V,
```

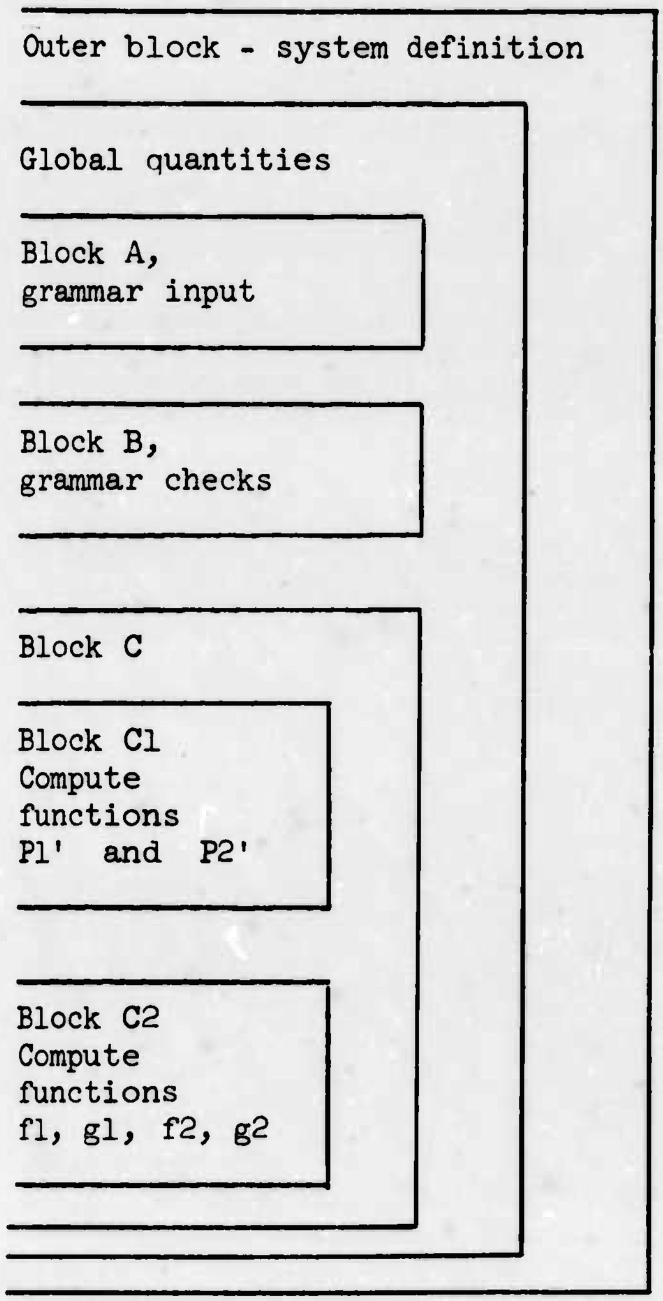
```
' The final grammar check--for nonterminating phrases'
for all i from 1 to NPR do P[i] ← P[i] ⊖ VT,
change ← true
while change do 'now try to collapse grammar'
{  change ← false,
   for all t from P do (if (length t) = 1 then
      {for all i from 1 to NPR do
         for all j from 2 to length P[i] do
         (if P[i][j] = t[1] then
            P[i] ← P[i] ⊖ {t[1]}),
         P ← P ⊖ {t}, change ← true
      })
},
if P ≠ {} then out ← out ⊕ "grammar includes a
non terminating phrase" ⊕ (⊕/⊕/P)⊕cr,
HS ← (P)
{ new s,
   for all t from PR do
      (if t[1] = s then if t[2] ∉ heads then
         { head ← heads ⊕ {t[2]}, HS(t[2]) })
},
TS ← (P)
{ new s,
   for all t from PR do
      (if t[1] = s then if t[-1] ∉ tails then
         { tails ← tails ⊕ {t[-1]}, TS(t[-1]) })
},
P1 ← P2 ← NSY list (NSY list Ω)
for all t from PR do for all i from 2 to (length t) - 1 do
{ j ← t[i], k ← t[i+1], heads ← tails ← {},
   TS(j),  HS(k),
   if P2[j][k] = Ω then
   { P2[j][k] ← 1,
      for all h from heads do
```

```
        if P2[j][h] = Ω then P2[j][k] ← 0 else
        (if P2[j][h] = 1 then out ← out ⊕
        "Conflict,  P2[" ⊕ V[j] ⊕ "][" ⊕ V[h] ⊕ "]" ⊕ cr),
    } else (if P2[j][k] = 0 then out ← out ⊕
    "Conflict, P2[" ⊕ V[j] ⊕ "][" ⊕ V[k] ⊕ "]" ⊕ cr),
    if V[k] ∈ VT then heads ← {k}, 'Now HST in heads'
    for all h from heads do (if V[k] ∈ VT then
    { if P1[j][h] = Ω then P1[j][h] ← 1 else
      (if P1[j][h] = 0 then out ← out ⊕
      "Conflict, P1[" ⊕ V[j] ⊕ "][" ⊕ V[h] ⊕ "]" ⊕ cr),
      for all g from tails do if P1[g][h] = Ω then P1[g][h] ← 0 else
      (if P1[g][h] = 1 then out ← out ⊕
      "Conflict, P1[" ⊕ V[g] ⊕ "][" ⊕ V[h] ⊕ "]" ⊕ cr),
    }
 },
 uprow ← (P)
 { new i p,
    if beenatrowk  ∧  i = k then fail ← true,
    beenatrowk ← beenatrowk  ∨ i = k,
    for all j from 1 to k1 do
    (if f[i] ≤ g[j] then if p[i][j] = 0 then f[i] ← g[j] + 1),
    for all j from 1 to k1 do
    (if  ¬ fail then if f[i] > g[j] then if p[i][j] = 1 then
      upcol(j, (P) p) ),

    if fail then out = out ⊕ "row= " ⊕ V[i] ⊕ cr

 },
 upcol ← (P)
 { new j p,
    if beenatcolk ∧ j = k then fail ← true,
    beenatcolk ← beenatcolk ∨ j = k,
    for all i from 1 to k do
    (if f[i] > g[j] then if p[i][j] = 1 then g[j] ← f[i]),
    for all i from 1 to k do
    (if ¬ fail then if f[i] ≤ g[j] then if p[i][j] = 0 then
      uprow(i, (P) p) ),
    if fail then out ← out ⊕ "col= " ⊕ V[j] ⊕ cr
},
```

```
fail ← false, kl ← 0
f ← g ← NSY list 0, 'Allocate storage to f and g'
for all k from 1 to NSY do if ¬ fail then
{ beenatrowk ← false,  f[k] ← g[k] ← 1,
   uprow(k, P P2),
   kl ← k, beenatcolk ← beenatrowk ← false,
   upcol(k, P P2)
},
out ← out ● "Linearized functions for P2:" ● cr ●
(for all i from 1 to NSY do (i base 10) ● tab ● V[i] ●
(f[i] base 10) ● tab ● (g[i] base 10) ● cr),
fail ← false, kl ← 0
for all k from 1 to NSY do if ¬ fail then
{ beenatrowk ← false, f[k] ← g[k] ← 1,
   uprow(k, P P1),
   kl ← k, beenatcolk ← beenatrowk ← false,
   upcol(k, P P1)
},
out ← out ● "Linearized functions for P1:" ● cr ●
(for all i from 1 to NSY do (i base 10) ● tab ● V[i] ●
(f[i] base 10) ● tab ● (g[i] base 10) ● cr)
}  'end of program'
```

The Algol version follows the kernel language version closely. We have taken especial care to minimize conflict in memory use in the Algol version. We provide three globals quantities, MAXNPR, MAXNSY, and MAXLPR which determine the size of the tables in the program. Within the system definition block (see the following diagram) we define the global arrays. Our first action block is A, where the data cards are read and the various tables built. In block B we check that the tables represent a grammar according to the restrictions of the theory. In block C1 the recognition functions are computed and in C2 the linearization is completed.

Block structure of the symbol pair analysis program



```
BEGIN  COMMENT  SYNTAX PROCESSOR, W. M. MCKEEMAN OCT, 1965;
INTEGER MAXNSY;                  COMMENT  MAX NUMBER OF SYMBOLS;
INTEGER MAXNPR;                  COMMENT  MAX NUMBER OF PRODUCTIONS;
INTEGER MAXLPR;                  COMMENT  MAX LENGTH OF A PRODUCTION;
INTEGER EITHER, YES, NO, LE, GT;
INTEGER TI, OT;                  COMMENT  TIMING INFORMATION;
FILE OUT CP O(2,10);

PROCEDURE TIMER;
BEGIN INTEGER T;  T ← TIME(1);
   WRITE(<"TIME =", F7.2," TOTAL ELAPSED = ", F7.2, " MIN.">,
   (T-OT)/3600, (T-TI)/3600);
   OT ← T;
END TIMER;

MAXNSY ← 300;  MAXNPR ← 300;  MAXLPR ← 5;
EITHER ← 0;  YES ← 1;  NO ← 2;  LE ← 1;  GT ← 2;
OT ← TI ← TIME(1);

BEGIN  COMMENT  SET UP GLOBAL TABLES;
   INTEGER ARRAY V0, V1[0:MAXNSY];     COMMENT  12 SIG. CHARS;
   INTEGER ARRAY PR[0:MAXNPR,0:MAXLPR];COMMENT  PRODUCTIONS;
   BOOLEAN ARRAY UNRIGHT[0:MAXNSY];
   INTEGER NPR;  COMMENT  ACTUAL NUMBER OF PRODUCTIONS READ;
   INTEGER NSYN, NSY;  COMMENT  ACTUAL NUMBER OF SYMBOLS READ;

   BEGIN  COMMENT  BLOCK A;     COMMENT  CARD INPUT BLOCK;
      INTEGER I, J, K, L;
      LABEL INPUTLOOP, EOF, FOUND;
      INTEGER ARRAY P0, P1[0:MAXNPR, 0:MAXLPR];
      INTEGER ARRAY MTBL[0:MAXNSY];   COMMENT  MASTER TABLE;
      INTEGER ARRAY PRTB[0:1022];  COMMENT  PRODUCTION TABLE;
      WRITE(<"PRODUCTIONS:"//>);
      NPR ← 0;
      INPUTLOOP:
      READ(<12A6>, FOR K ← 0 STEP 1 UNTIL MAXLPR DO
         [P0[NPR+1, K], P1[NPR+1, K]])[EOF];
      IF P0[NPR+1, 1] = "      " THEN WRITE(<" ">) ELSE
      BEGIN  NPR ← NPR + 1;
         WRITE(<I8,X8,2A6," ←    ",10A6>, NPR,
         FOR K ← 0 STEP 1 UNTIL MAXLPR DO [P0[NPR,K],P1[NPR,K]]);
      END;
      GO TO INPUTLOOP;

      EOF:
      NSY ← 0;  V0[0] ← V1[0] ← "      ";
      FOR K ← 0 STEP 1 UNTIL MAXLPR DO
      BEGIN  FOR I ← 1 STEP 1 UNTIL NPR DO
         BEGIN  FOR J ← 0 STEP 1 UNTIL NSY DO
            IF P0[I,K] = V0[J] AND P1[I,K] = V1[J] THEN
               GO TO FOUND;
            J ← NSY ← NSY + 1;  V0[J] ← P0[I,K];  V1[J] ← P1[I,K];
            FOUND:
            PR[I,K] ← J;
            IF K ≠ 0 THEN UNRIGHT[J] ← TRUE;
```

```
         END I;
         IF K = 0 THEN NSYN ← NSY;
      END K;

      FOR I ← 2 STEP 1 UNTIL NPR DO IF PR[I,0] = 0 THEN
         PR[I,0] ← PR[I-1,0];

      WRITE([PAGE]);  WRITE(<"INTERMEDIATE SYMBOLS:">);
      WRITE(<5(I8,X3,2A6)>, FOR I ← 1 STEP 1 UNTIL NSYN DO
         [I, V0[I], V1[I]]);
      WRITE(<//"TERMINAL SYMBOLS:">);
      WRITE(<5(I8,X3,2A6)>, FOR I ← NSYN+1 STEP 1 UNTIL NSY DO
         [I, V0[I], V1[I]]);
      WRITE(CP,<"FILL V0[*] WITH 0,", 6("""",A6,"""",",")/
      (8("""",A6,"""",","))>,FOR I ← 1 STEP 1 UNTIL NSY DO V0[I]);
      WRITE(CP,<"FILL V1[*] WITH 0,", 6("""",A6,"""",",")/
      (8("""",A6,"""",","))>,FOR I ← 1 STEP 1 UNTIL NSY DO V1[I]);

      L ← 0;
      FOR I ← 1 STEP 1 UNTIL NSY DO
      BEGIN  MTB[I] ← L+1;
         FOR J ← 1 STEP 1 UNTIL NPR DO IF PR[J,1] = I THEN
         BEGIN  FOR K ← 2 STEP 1 UNTIL MAXLPR DO
            IF PR[J,K] ≠ 0 THEN PRTB[L←L+1]←PR[J,K];
            PRTB[L←L+1] ← -J;  PRTB[L←L+1] ← PR[J,0];
         END J;
         PRTB[L←L+1] ← 0;
      END I;

      WRITE(CP,<"FILL PRTB[*] WITH  0,",10(I4,",")/
      ("  ",14(I4,","))>, FOR I ← 1 STEP 1 UNTIL L DO PRTB[I]);
      WRITE(CP,<"FILL MTB[*] WITH ", 13(I3,",")/
      ("  ",17(I3,","))>,0,FOR I ← 1 STEP 1 UNTIL NSY DO MTB[I]);
      WRITE(CP,<"NSY ← ",I3, ";  NSYN ← ", I3, ";  NPRTB ← ", I3,";">,
      NSY, NSYN, L);
   END BLOCK A;


   BEGIN  COMMENT  BLOCK B;                COMMENT  GRAMMAR CHECKS;
      INTEGER I, J, K;
      LABEL OK;
      J ← 0;
      FOR I ← 1 STEP 1 UNTIL NSYN DO IF NOT ONRIGHT[I] THEN
      BEGIN  J ← J + 1;
         WRITE(</"THE UNIQUE TARGET SYMBOL IS:  ", 2A6>, V0[I],V1[I]);
      END I;
      IF J ≠ 1 THEN WRITE(<"THERE IS NO UNIQUE LEFTMOST SYMBOL">);

      FOR I ← 1 STEP 1 UNTIL NPR DO
      BEGIN  COMMENT  CHECK FOR EMPTY LEFT AND RIGHT PARTS;
         IF PR[I,0] = 0 THEN
            WRITE(<"PRODUCTION ", J, " HAS AN EMPTY LEFT PART">,I);
         IF PR[I,1] = 0 THEN
            WRITE(<"PRODUCTION ", J, " HAS AN EMPTY RIGHT PART">,I);
         FOR J ← I+1 STEP 1 UNTIL NPR DO
```

```
      BEGIN  COMMENT  CHECK FOR IDENTICAL RIGHT PARTS;
         FOR K ← 1 STEP 1 UNTIL MAXLPR DO IF PR[I,K] ≠ PR[J,K] THEN
            GO TO OK;
         WRITE(<"PRODUCTIONS ",J," AND ", J,
         " MUST BE DISTINGUISHED BY THE INTERPRETATION RULES">,I,J);
      OK:
      END J;
   END I;
   TIMER;
END BLOCK B;


BEGIN  COMMENT  BLOCK C;           COMMENT  SYNTAX ANALYSIS;
   ALPHA ARRAY P1, P2[0:NSY, 0:NSY DIV 24];

   COMMENT  PACKING AND UNPACKING PROCEDURES;
   STREAM PROCEDURE SET2BITS(W, I, V);  VALUE I;
   BEGIN  DI ← W;  2(SKIP I DB); SI ← V; SKIP 46 SB;
      2(IF SB THEN DS ← SET ELSE DS ← RESET; SKIP SB));
   END SET2BITS;

   INTEGER STREAM PROCEDURE GET2BITS(W, I);  VALUE I;
   BEGIN  DI ← LOC GET2BITS;  SKIP 46 DB; SI ← W; 2(SKIP I SB));
      2(IF SB THEN DS ← SET ELSE DS ← RESET; SKIP SB));
   END GET2BITS;

   BEGIN  COMMENT  BLOCK C 1;   COMMENT COMPUTE PRECEDENCE RELATIONS;
      INTEGER ARRAY HEADS, TAILS[0:NSY];
      INTEGER G, H, I, J, K, L, LC, RC, T, DIV24, MOD24;
      BOOLEAN FAIL;
      LABEL SKIPP1, SKIPP2, DONE;

      PROCEDURE HS(S);  VALUE S;  INTEGER S;
      BEGIN  COMMENT  FIND THE LEFTMOST SYMBOLS OF S;
         INTEGER I, J, K;
         LABEL GOTITALREADY;
         FOR I ← 1 STEP 1 UNTIL NPR DO IF PR[I,0] = S THEN
         BEGIN  K ← PR[I,1];
            FOR J ← 1 STEP 1 UNTIL LC DO IF HEADS[J] = K THEN
               GO TO GOTITALREADY;
            LC ← LC + 1; HEADS[LC] ← K;  HS(K);
            GOTITALREADY:
         END I;
      END HS;

      PROCEDURE TS(S);  VALUE S;  INTEGER S;
      BEGIN  COMMENT  FIND THE RIGHTMOST SYMBOLS OF S;
         INTEGER I, J, K;
         LABEL GOTITALREADY, R;
         FOR I ← 1 STEP 1 UNTIL NPR DO IF PR[I,0] = S THEN
         BEGIN  FOR J ← MAXLPR STEP -1 UNTIL 1 DO IF PR[I,J] ≠ 0 THEN
            GO TO R;
            R:  K ← PR[I,J];
            FOR J ← 1 STEP 1 UNTIL RC DO IF TAILS[J] = K THEN
               GO TO GOTITALREADY;
```

```
      RC ← RC + 1;  TAILS[RC] ← K;  TS(K);
      GOTITALREADY:
   END I;
END TS;

PROCEDURE CONFLICT(I,J,M); INTEGER I,J,M;
BEGIN  INTEGER C;
   FAIL ← TRUE;
   WRITE([NO],<X29,"/">);
   WRITE(<"CONFLICT, ", 2A6, " ", A1, " AND " A1,X2,2A6>,
   V0[I],V1[I], M,M, V0[J],V1[J]);
END CONFLICT;

FAIL ← FALSE;
FOR I ← 1 STEP 1 UNTIL NPR DO FOR L ← 2 STEP 1 UNTIL MAXLPR DO
BEGIN
   J ← PR[I,L-1];  K ← PR[I,L];
   IF K = 0 THEN GO TO DONE;
   DIV24 ← K DIV 24;  MOD24 ← K MOD 24;
   LC ← RC ← 0;
   TS(J);  HS(K);
   T ← GET2BITS(P2[J,DIV24],MOD24);
   IF T = YES THEN GO TO SKIPP2;
   IF T = NO THEN CONFLICT(J,K,"N") ELSE
   SET2BITS(P2[J,DIV24],MOD24,YES);
   FOR H ← 1 STEP 1 UNTIL LC DO
   BEGIN  DIV24 ← HEADS[H] DIV 24; MOD24 ← HEADS[H] MOD 24;
      IF GET2BITS(P2[J,DIV24],MOD24) = YES THEN
      CONFLICT(J,HEADS[H],"N") ELSE
      SET2BITS(P2[J,DIV24],MOD24,NO);
   END H;
   SKIPP2:

   IF K > NSYN THEN
   BEGIN  COMMENT IF "K" IS TERMINAL WE MUST TABULATE IT;
      LC ← LC + 1;
      HEADS[LC] ← K;
   END;
   FOR H ← 1 STEP 1 UNTIL LC DO
   BEGIN  COMMENT  ONLY TERMINAL SYMBOLS ARE INVOLVED;
      IF HEADS[H] ≤ NSYN THEN GO TO SKIPP1;
      DIV24 ← HEADS[H] DIV 24;  MOD24 ← HEADS[H] MOD 24;
      IF GET2BITS(P1[J,DIV24],MOD24) = NO THEN
         CONFLICT(J,HEADS[H],"S") ELSE
         SET2BITS(P1[J,DIV24],MOD24,YES);
      FOR G ← 1 STEP 1 UNTIL RC DO
         IF GET2BITS(P1[TAILS[G],DIV24],MOD24) = YES THEN
            CONFLICT(TAILS[G],HEADS[H],"S") ELSE
            SET2BITS(P1[TAILS[G],DIV24],MOD24,NO);
      SKIPP1:
   END H;
   DONE:
END L I;
IF NOT FAIL THEN WRITE(</"NO CONFLICTS WERE FOUND">);
TIMER;
```

```
END BLOCK C 1;

BEGIN  COMMENT  BLOCK C 2;     COMMENT  LINEARIZE MATRICES;
   INTEGER ARRAY F, G[0:NSY];
   INTEGER K, K1;
   BOOLEAN FAIL, BEENATROWK, BEENATCOLK;

   PROCEDURE UPCOL(J,P); VALUE J; INTEGER J; ALPHA ARRAY P[0,0];
      FORWARD;

   PROCEDURE UPROW(I,P); VALUE I; INTEGER I; ALPHA ARRAY P[0,0];
   BEGIN INTEGER J;
      IF BEENATROWK AND I = K THEN FAIL ← TRUE;
      BEENATROWK ← BEENATROWK OR I = K;
      FOR J ← 1 STEP 1 UNTIL K1 DO IF F[I] ≤ G[J] THEN
         IF GET2BITS(P[I,J DIV 24],ENTIER(J MOD 24)) = GT THEN
            F[I] ← G[J] + 1;
      FOR J ← 1 STEP 1 UNTIL K1 DO IF NOT FAIL THEN
         IF F[I] > G[J] THEN
            IF GET2BITS(P[I,J DIV 24],ENTIER(J MOD 24)) = LE THEN
               UPCOL(J, P);
      IF FAIL THEN WRITE(<"ROW = ",I3, "  ", 2A6>,I,V0[I],V1[I]);
   END UPROW;

   PROCEDURE UPCOL(J,P); VALUE J; INTEGER J; ALPHA ARRAY P[0,0];
   BEGIN INTEGER I, JDIV24, JMOD24;
      IF BEENATCOLK AND J = K THEN FAIL ← TRUE;
      BEENATCOLK ← BEENATCOLK OR J = K;
      JDIV24 ← J DIV 24;  JMOD24 ← J MOD 24;
      FOR I ← 1 STEP 1 UNTIL K DO IF F[I] > G[J] THEN
         IF GET2BITS(P[I,JDIV24],JMOD24) = LE THEN G[J] ← F[I];
      FOR I ← 1 STEP 1 UNTIL K DO IF NOT FAIL THEN
         IF F[I] ≤ G[J] THEN
            IF GET2BITS(P[I,JDIV24],JMOD24) = GT THEN
               UPROW(I, P);
      IF FAIL THEN WRITE(<"COL = ",I3, "  ", 2A6>,J,V0[J],V1[J]);
   END UPCOL;

   FAIL ← FALSE;  K1 ← 0;  WRITE([PAGE]);
   FOR K ← 1 STEP 1 UNTIL NSY DO IF NOT FAIL THEN
   BEGIN  BEENATROWK ← FALSE; F[K] ← G[K] ← 1;
      UPROW(K,P2);
      K1 ← K;  BEENATCOLK ← BEENATROWK ← FALSE;
      UPCOL(K,P2);
   END K;
   IF FAIL THEN
      WRITE(<"LINEARIZATION FAILURE FOR FUNCTIONS BELOW">);
   WRITE(<"LINEARIZED PRODUCTION RECOGNITION MATRIX"/
   X7,"NO,",X9,"SYMBOL",X10,"F",X7,"G"/(I10,X6,2A6,2I8)>,
   FOR K ← 1 STEP 1 UNTIL NSY DO [K,V0[K],V1[K],F[K],G[K]]);
   WRITE(CP,<"FILL F2[*] WITH 0,", 18(I2,",")/(24(I2,","))>,
   FOR K ← 1 STEP 1 UNTIL NSY DO F[K]);
   WRITE(CP,<"FILL G2[*] WITH 0,", 18(I2,",")/(24(I2,","))>,
   FOR K ← 1 STEP 1 UNTIL NSY DO G[K]);
```

```
        FAIL ← FALSE;  K1 ← 0;  TIMER;  WRITE([PAGE]);
        FOR K ← 1 STEP 1 UNTIL NSY DO IF NOT FAIL THEN
        BEGIN  BEENATROWK ← FALSE; F[K] ← G[K] ← 1;
            UPROW(K,P1);
            K1 ← K;  BEENATCOLK ← BEENATROWK ← FALSE;
            UPCOL(K,P1);
        END K;
        IF FAIL THEN
            WRITE(<"LINEARIZATION FAILURE FOR FUNCTIONS BELOW">);
        WRITE(<"LINEARIZED HIERARCHY ANALYSIS MATRIX:"/
        X7,"NO.",X9,"SYMBOL",X10,"F",X7,"G"/(I10,X6,2A6,2I8)>,
        FOR K ← 1 STEP 1 UNTIL NSY DO [K,V0[K],V1[K],F[K],G[K]]);
        WRITE(CP,<"FILL F1[*] WITH 0,", 18(I2,","),/(24(I2,",")))>,
        FOR K ← 1 STEP 1 UNTIL NSY DO F[K]);
        WRITE(CP,<"FILL G1[*] WITH 0,", 18(I2,","),/(24(I2,",")))>,
        FOR K ← 1 STEP 1 UNTIL NSY DO G[K]);
    END BLOCK C 2;
  END BLOCK C;
END;
TIMER;
END.
```

(2,1)(1,2) Parsing Functions

Consider the grammar

$\underline{P} = \{G \rightarrow AB,\ \ B \rightarrow BC,\ \ B \rightarrow C\}$.

The symbol B is left recursive, that is, $B \in HS(B)$. From the first production we can derive a conflict in P2'. Similiarly, if A had been right recursive, we would have had a conflict in P1' from the first production. We can sum up both situations by saying that an internal recursion will always cause a conflict. Note that the grammar

$\underline{P} = \{G \rightarrow AB',\ \ B' \rightarrow B,\ \ B \rightarrow BC,\ \ B \rightarrow C\}$

has no internal recursion and is a symbol pair grammar. While we must reject arbitrary grammar transformations on semantic grounds, the insertion of a dummy production does not affect the semantic interpretation of the language. The reader will note several such dummy productions in the grammar of our kernel language.

We would like to extend the range of our grammars without requiring additional work by the programmer. It is perfectly feasible to test for internal recursions and automatically insert dummy productions into the grammar prior to starting the analysis of the syntax

A perhaps more hopeful approach is to extend the view of the functions P1' and P2'. It happens that internal recursions are allowed if we look left one extra symbol for P1 and right one extra symbol for P2. Extending the notation of Wirth and Weber ([25] p. 32) we call the symbol pair grammars (1,1)(1,1) canonical parse grammars and the suggested extension (2,1)(1,2) canonical parse grammars.

A (2,1)(1,2) syntax preprocessor is considerably more complicated than that for a (1,1)(1,1) grammar. In particular, the defining matrices for P1"(X,Y,Z) and P2"(X,Y,Z) contain NSY cubed elements. Even though the density of defined entries is on the order of one percent, a moderately large grammar may require 10,000 entries. It is encouraging to note that no naturally occuring grammar has failed to be a (2,1)(1,2) grammar.

The rules for deriving the values of P1" and P2" are similar to those for deriving P1' and P2'. We will first tabulate the various set definitions required for the derivations and then state the rules derived from certain standard production formats.

Set definitions.

canonical parse tail two symbols T2S(P)	$\{\{X,Y\} \mid (\exists u)(\exists R)\ P \rightarrow uXR,\ R \Rightarrow Y\} \cup$ $\{\{X,Y\} \mid (\exists u)(\exists R)\ P \rightarrow uR,$ $\{X,Y\} \in T2S(R)\}$
Canonical parse head two symbols H2S(P)	$\{\{X,Y\} \mid (\exists u)(\exists R)\ P \rightarrow XRu,$ $(Y = R \text{ or } Y \in HS(R))\} \cup$ $\{\{X,Y\} \mid (\exists u)(\exists R)\ P \rightarrow Ru,$ $\{X,Y\} \in H2S(R)\}$
Allowed predecessors AP(P)	$\{X \mid (\exists Q)(\exists R)(\exists x)(\exists y)\ R \rightarrow xXQy,$ $(P = Q \text{ or } P \in HS(Q))\}$
Allowed successors AS(P)	$\{X \mid (\exists Q)(\exists R)(\exists x)(\exists y)\ R \rightarrow xQXy,$ $(P = Q \text{ or } P \in TS(Q))\}$

Derivation rules for the parsing function values.

P1"

$W \rightarrow UVv$	$(\exists X)(\exists Z)\ Z \in HS_T(V),\ X \in AP(W)$ implies $P1''(X,U,Z) = \underline{\text{false}}$.
$W \rightarrow UVv$	$(\exists X)(\exists Y)(\exists Z)\ Z \in HS_T(V),\ X \in AP(W)$, $U \rightarrow\Rightarrow Y$ implies $P1''(X,Y,Z) = \underline{\text{true}}$.
$W \rightarrow UVv$	$(\exists X)(\exists Y)(\exists Z)\ Z \in HS_T(V),\ \{X,Y\} \in T2S(U)$ implies $P1''(X,Y,Z) = \underline{\text{true}}$.
$W \rightarrow tTUVv$	$(\exists Z)\ Z \in HS_T(V)$ implies $P1''(T,U,Z) = \underline{\text{false}}$.
$W \rightarrow tTUVv$	$(\exists Y)(\exists Z)\ Z \in HS_T(V),\ U \rightarrow\Rightarrow Y$ implies $P1''(T,Y,Z) = \underline{\text{true}}$.
$W \rightarrow tTUVv$	$(\exists X)(\exists Y)(\exists Z)\ Z \in HS_T(V),\ \{X,Y\} \in T2S(U)$ implies $P1''(X,Y,Z) = \underline{\text{true}}$.

P2"

$W \rightarrow tTU$	$(\exists S)\ S \in AS(W),\ (\exists Z)\ Z \in HS_T(S)$ implies $P2''(T,U,Z) = \underline{\text{false}}$.
$W \rightarrow tTU$	$(\exists S)\ S \in AS(W),\ (\exists Y)(\exists Z)\ Z \in HS_T(S)$, $U \rightarrow\Rightarrow Y$ implies $P2''(T,Y,Z) = \underline{\text{true}}$.
$W \rightarrow tTU$	$(\exists Y)(\exists Z)\ \{Y,Z\} \in H2S(U)$ implies $P2''(T,Y,Z) = \underline{\text{true}}$.
$W \rightarrow tTUVv$	$P2''(T,U,V) = \underline{\text{false}}$.
$W \rightarrow tTUVv$	$(\exists Y)(\exists Z)\ Z \in HS_T(V),\ U \rightarrow\Rightarrow Y$ implies $P2''(T,Y,Z) = \underline{\text{true}}$.
$W \rightarrow tTUVv$	$(\exists Y)(\exists Z)\ \{Y,Z\} \in H2S(U)$ implies $P2''(T,Y,Z) = \underline{\text{true}}$.

The block structure of the (2,1)(1,2) preprocessor is similiar to that of the symbol pair preprocessor. We organize the set definitions for allowed predecessors, allowed successors, single character derivatives ($Y \rightarrow\Rightarrow Z$), head symbols and tail symbols as Boolean matrices. If, for example, AP[I,J] = <u>true</u> then symbol number J is an allowed predecessor of symbol number I. We gain by avoiding table look ups and loose by being forced to pack the matrices. The blocks C1, C2, and C3 contain relatively transparent algorithms for the computation of the five sets.

Block C4 delineates the algorithm for computing the function P1". Consider an arbitrary canonical derivation $Y \Rightarrow t$ where $t \in \underline{V}_T^*$. For every intermediate stage of the derivation (such that it has at least two symbols) the pair of rightmost two symbols of the produced string are an entry in the canonical parse tail two symbols of Y. The procedure T2S tabulates pairs of tail symbols over all possible derivations emanating from its argument. Storage requirements force us to abandon the Boolean matrix definition for these sets and we tabulate them in a linear array. It is also infeasible to record the values of P1" in a three dimensional matrix hence we record the values in four linear arrays, the first three giving the coordinates of the point and the fourth its value. At the innermost loop of the analysis (nested within four FOR's and five IF's) we find a call on procedure **ENTER** which records the computed value. Since the speed of execution of the algorithm is proportional to the speed of **ENTER**, we have attempted to code it efficiently. The first implication is the need for a binary table look up which itself demands that the three coordinate arrays be packed in a single word as the polynomial value $I \times N^2 + J \times N + K$ where $N > NSY$. Secondly we use even powers of two and Burroughs B5500 partial word operators instead of multiplies and divides as indicated in the comments.

In block C5 we find similiar algorithms to compute P2". As one immediately sees by inspecting the output from a trial run on the pages following the program, even a small grammar generates an enormous number of relations. The number is so large that we have been unable to test the program for large grammars. Yet we feel that the information in the tables is highly redundant leading us to conjecture the existence of some analogue to Floyd's f and g functions for condensing the information. To date we have not been able to find a reliable algorithm for this purpose.

Our inability to condense the definitions of P1" and P2" into reasonably compact tables is the only bar to their use in the syntactic analyzer of the compiler. It appears that (2,1)(1,2) grammars are sufficiently powerful to describe computer languages with no further generalization. There would be some advantage in generalizing the function P3' to allow repeated and empty right parts in the production tables.

The sample output has been slightly rearranged from the actual computer output. The first page contains listings of $\underline{P}$, $\underline{V}_N$, $\underline{V}_T$, and G. Then follow the definitions of the five sets. The left margin contains the symbol number and name; the top margin the least significant digit of the symbol number. A dot signifies that the symbol numbered in the top margin stands in the indicated relation to the symbol in the left margin. For example, EOF is in the head of <PROGRAM>.

The first tabulated value for P1" indicates that <EXPR> ELSE IF is an expected triplet and that IF is not to be moved from $\underline{z}$ to $\underline{x}$ in the general parsing algorithm (because <EXPR> ELSE must first form <TRUEPART>).

```
BEGIN  COMMENT  (2,1)(1,2) SYNTAX PROCESSOR  MCKEEMAN  JAN. 1966;
INTEGER MAXNSY;               COMMENT  MAX NUMBER OF SYMBOLS;
INTEGER MAXNPR;               COMMENT  MAX NUMBER OF PRODUCTIONS;
INTEGER MAXLPR;               COMMENT  MAX LENGTH OF A PRODUCTION;
INTEGER TI, OT, T;            COMMENT  TIMING INFORMATION;
INTEGER P2CSAVE, SI, I;       COMMENT  STATISTICS STORAGE;
REAL ARRAY RECORD[0:20];
DEFINE PACKED = ALPHA#;

PROCEDURE TIMER;
BEGIN  OT ← T;  T ← TIME(1);
   WRITE(<"TIME =", F7.2," TOTAL ELAPSED = ", F7.2, " MIN.">,
   (T-OT)/3600, (T-TI)/3600);
END TIMER;

PROCEDURE SAV(X);  VALUE X;  REAL X; RECORD[SI←SI+1] ← X;

MAXNSY ← 300;  MAXNPR ← 300;  MAXLPR ← 5;
P2CSAVE ← SI ← 0;
T ← TI ← TIME(1);
WRITE(<"(2,1)(1,2) SYNTAX PROCESSOR, MCKEEMAN, JAN. 1966"//>);

BEGIN  COMMENT  SET UP GLOBAL TABLES;
   INTEGER ARRAY V0, V1[0:MAXNSY];      COMMENT  12 SIG. CHARS;
   INTEGER ARRAY PR[0:MAXNPR, 0:MAXLPR];  COMMENT  PRODUCTIONS;
   BOOLEAN ARRAY ONRIGHT[0:MAXNSY];
   INTEGER NPR;  COMMENT  ACTUAL NUMBER OF PRODUCTIONS READ;
   INTEGER NSY, NSYN;  COMMENT  ACTUAL NUMBER OF SYMBOLS READ;

   BEGIN  COMMENT  BLOCK A;     COMMENT  CARD INPUT BLOCK;
      INTEGER I, J, K;
      LABEL INPUTLOOP, EOF, FOUND;
      INTEGER ARRAY P0, P1[0:MAXNPR, 0:MAXLPR];
      WRITE(<"PRODUCTIONS:"//>);
      NPR ← 0;
      INPUTLOOP:
      READ(<12A6>, FOR K ← 0 STEP 1 UNTIL MAXLPR DO
         [P0[NPR+1, K], P1[NPR+1, K]])[EOF];
      IF P0[NPR+1,1] = "      " THEN WRITE(<" ">) ELSE
      BEGIN  NPR ← NPR + 1;
         WRITE(<I8,X8,2A6," ←    ",10A6>, NPR,
          FOR K ← 0 STEP 1 UNTIL MAXLPR DO [P0[NPR,K], P1[NPR,K]]);
      END;
      GO TO INPUTLOOP;

      EOF:
      NSY ← 0;  V0[0] ← V1[0] ← "      ";
      FOR K ← 0 STEP 1 UNTIL MAXLPR DO
      BEGIN  FOR I ← 1 STEP 1 UNTIL NPR DO
         BEGIN  FOR J ← 0 STEP 1 UNTIL NSY DO
            IF P0[I,K] = V0[J] AND P1[I,K] = V1[J] THEN
               GO TO FOUND;
            J ← NSY ← NSY + 1;
            V0[NSY] ← P0[I,K];  V1[NSY] ← P1[I,K];
            FOUND:
```

```
         PR[I,K] ← J;
         IF K ≠ 0 THEN ONRIGHT[J] ← TRUE;
      END I;
      IF K = 0 THEN NSYN ← NSY;  COMMENT  STILL IN INTERMEDIATE SYM;
   END K;

   FOR I ← 2 STEP 1 UNTIL NPR DO IF PR[I,0] = 0 THEN
      PR[I,0] ← PR[I-1,0];

   WRITE([PAGE]);  WRITE(<"INTERMEDIATE SYMBOLS:">);
   WRITE(<3(I8,X3,2A6)>, FOR I ← 1 STEP 1 UNTIL NSYN DO
      [I, V0[I], V1[I]]);
   WRITE(<//"TERMINAL SYMBOLS:">);
   WRITE(<3(I8,X3,2A6)>, FOR I ← NSYN+1 STEP 1 UNTIL NSY DO
      [I, V0[I], V1[I]]);
   COMMENT  GATHER STATISTICS;
   SAV(NPR);  SAV(NSY);  SAV(NSYN);
END BLOCK A;


BEGIN  COMMENT  BLOCK B;              COMMENT  GRAMMAR CHECKS;
   INTEGER ARRAY TEST[0:NPR, 0:MAXLPR];
   BOOLEAN CHANGE, EMPTY;
   INTEGER I, J, K, Z;
   LABEL OK;

   J ← 0;
   FOR I ← 1 STEP 1 UNTIL NSYN DO IF NOT ONRIGHT[I] THEN
   BEGIN  J ← J + 1;
      WRITE(</"THE UNIQUE TARGET SYMBOL IS:  ", 2A6>, V0[I], V1[I]);
   END I;
   IF J ≠ 1 THEN WRITE(<"THERE IS NO UNIQUE LEFTMOST SYMBOL">);

   FOR I ←1 STEP 1 UNTIL NPR DO FOR J ← 0 STEP 1 UNTIL MAXLPR DO
      TEST[I,J] ← IF PR[I,J] > NSYN THEN 0 ELSE PR[I,J];
   CHANGE ← TRUE;
   WHILE CHANGE DO
   BEGIN  CHANGE ← FALSE;
      FOR I ← 1 STEP 1 UNTIL NPR DO
      BEGIN  Z ← TEST[I,0];
         IF Z ≠ 0 THEN
         BEGIN  EMPTY ← TRUE;
            FOR J ← 1 STEP 1 UNTIL MAXLPR DO
               EMPTY ← EMPTY AND TEST[I,J] = 0;
            IF EMPTY THEN FOR K ← 1 STEP 1 UNTIL NPR DO
               FOR J ← 0 STEP 1 UNTIL MAXLPR DO IF TEST[K,J] = Z THEN
                  TEST[K,J] ← 0;
            CHANGE ← CHANGE OR EMPTY;
         END;
      END;
   END CHANGE;
   FOR I ← 1 STEP 1 UNTIL NPR DO IF TEST[I,0] ≠ 0 THEN
      WRITE(<"PRODUCTION", I4, " LEADS TO A NON-TERMINATING PHRASE">,
      I);
```

```
   FOR I ← 1 STEP 1 UNTIL NPR DO
   BEGIN  COMMENT  CHECK FOR EMPTY LEFT AND RIGHT PARTS;
      IF PR[I,0] = 0 THEN
         WRITE(<"PRODUCTION ", J, " HAS AN EMPTY LEFT PART">,I);
      IF PR[I,1] = 0 THEN
         WRITE(<"PRODUCTION ", J, " HAS AN EMPTY RIGHT PART">,I);
      FOR J ← I+1 STEP 1 UNTIL NPR DO
      BEGIN  COMMENT  CHECK FOR IDENTICAL RIGHT PARTS;
         FOR K ← 1 STEP 1 UNTIL MAXLPR DO IF PR[I,K] ≠ PR[J,K] THEN
            GO TO OK;
         WRITE(<"PRODUCTIONS ",J," AND ", J,
         " MUST BE DISTINGUISHED BY THE INTERPRETATION RULES">,I,J);
      OK:
      END J;
   END I;
   TIMER;
   WRITE([PAGE]);
END BLOCK B;


BEGIN  COMMENT  BLOCK C;            COMMENT  SYNTAX ANALYSIS;
   PACKED ARRAY CR[0:1022];    COMMENT  COORDINATES;
   INTEGER ARRAY S1, S2[0:1022];  COMMENT  NSY*2;
   BOOLEAN ARRAY V[0:1022];
   PACKED ARRAY INHEAD, INTAIL[0:NSY, 0:NSY DIV 48];
   PACKED ARRAY SCD[0:NSY, 0:NSY DIV 48];
   PACKED ARRAY AP, AS[0:NSY, 0:NSY DIV 48];
   BOOLEAN ARRAY BEENTHERE[0:NSY];
   INTEGER NVAL, P2C, FINDS;

   BOOLEAN STREAM PROCEDURE GETBIT(A, I);  VALUE I;
   BEGIN  SI ← A;  SKIP I SB;  TALLY ← 1;
      IF SB THEN GETBIT ← TALLY;
   END GETBIT;

   STREAM PROCEDURE SETBIT(A, I);  VALUE I;
   BEGIN  DI ← A;  SKIP I DB;  DS ← SET;
   END SETBIT;

   PROCEDURE ENTER(I, J, K, X); VALUE I, J, K, X;
   INTEGER I, J, K;  BOOLEAN X;
   BEGIN  LABEL  BINARYLOOKUP, GOTITALREADY;
      INTEGER B, M, T, H, NH, L;
      IF NVAL = 1022 THEN
      BEGIN  WRITE(<"TOO MANY ANALYSIS FUNCTION VALUES">);
         NVAL ← 1;
         TIMER;
      END;
      COMMENT  WE PACK COORDINATES BOTH FOR STORAGE ECONOMY AND
      SPEED IN THE BINARY LOOKUP FOR INSERTION;
      B ← 0;  T ← NVAL;  H ← K&J[24:36:12]&I[12:36:12];
      COMMENT  H IS THE COORDINATES AS POWERS OF 2*10;
```

```
   BINARYLOOKUP:  M ← (B+T).[36:11];  COMMENT  DIV 2;
   NH ← CR[M];
   IF NH < H THEN B ← M ELSE
   IF NH > H THEN T ← M ELSE
   BEGIN  IF NOT (X EQV V[M]) THEN WRITE(<"CONFLICT,", 6A6>,
      V0[I],V1[I], V0[J],V1[J], V0[K],V1[K]);
      FINDS ← FINDS + 1;  COMMENT  FOR STATISTICS;
      GO TO GOTITALREADY;
   END;
   IF B+1 ≠ T THEN GO TO BINARYLOOKUP;
   FOR L ← NVAL STEP -1 UNTIL T DO
   BEGIN  CR[L+1] ← CR[L];
      V[L+1] ← V[L];
   END L;
   CR[T] ← H;  V[T] ← X;
   NVAL ← NVAL + 1;
   GOTITALREADY:
END ENTER;

PROCEDURE PUT(X, Y);  VALUE X, Y;  INTEGER X, Y;
BEGIN  COMMENT  ENTER A HEAD OR TAIL PAIR INTO LIST;
   INTEGER I;  LABEL GOTITALREADY;
   FOR I ← 1 STEP 1 UNTIL P2C DO
      IF S1[I] = X THEN
         IF S2[I] = Y THEN GO TO GOTITALREADY;
   P2C ← P2C + 1;
   IF P2C > 1022 THEN WRITE(<"TOO MANY PAIRS">);
   COMMENT  WE SAVE P2C FOR STATISTICAL ANALYSIS;
   IF P2C > P2CSAVE THEN P2CSAVE ← P2C;
   S1[P2C] ← X;  S2[P2C] ← Y;
   GOTITALREADY:
END PUT;

PROCEDURE PRINTMATRIX(TITLE, M);  FORMAT TITLE;
PACKED ARRAY M[0,0];
BEGIN  COMMENT  PRINT A BOOLEAN MATRIX;
   INTEGER I, J;
   WRITE(TITLE);
   WRITE(<X9, "SYMBOL", X5, 100I1>,
      FOR I ← 1 STEP 1 UNTIL NSY DO I MOD 10);
   FOR I ← 1 STEP 1 UNTIL NSY DO
      WRITE(<I3, X3, 2A6, X2, 100A1>, I, V0[I], V1[I],
      FOR J ← 1 STEP 1 UNTIL NSY DO
      IF GETBIT(M[I,J DIV 48],ENTIER(J MOD 48))THEN "." ELSE " ");
   TIMER;
   WRITE([PAGE]);
END PRINTMATRIX;

PROCEDURE TABULATE(N);  VALUE N;  ALPHA N;
BEGIN  COMMENT  PRINT VALUES PF P1"(X,Y,Z) AND P2"(X,Y,Z);
   INTEGER I, C1, C2, C3;
   FOR I ← 1 STEP 1 UNTIL NVAL DO
   BEGIN  C1 ← CR[I].[12:12];  C2 ← CR[I].[24:12];
      C3 ← CR[I].[36:12];
      WRITE(<A2,"""","( ",6A6,")  =  ",A2,"""","(",2(I3,","),I3,
```

```
      ") = ", L5>,
      N, VO[C1],V1[C1], VO[C2],V1[C2], VO[C3],V1[C3],
      N, C1, C2, C3, V[I]);
   END;
   WRITE(<I5, " FUNCTION VALUES, DENSITY =", F6.2, "%",
      ", ENTRIES/VALUE", F6.2>,
      NVAL, 100×NVAL/NSY*3, (FINDS+NVAL)/NVAL);
   TIMER;
   SAV((T-OT)/3600);  SAV(NVAL);
   WRITE([PAGE]);
END TABULATE;

CR[0] ← 0;   COMMENT  A SAFE "BOTTOM" FOR THE BINARY LOOKUP;

BEGIN  COMMENT  BLOCK C 1;   COMMENT  HEAD AND TAIL OCCURENCES;
   COMMENT  INHEAD[I,J] IMPLIES J IS IN THE HEAD OF I;
   INTEGER I, J;

   PROCEDURE HS(S);  VALUE S;  INTEGER S;
   BEGIN  COMMENT  FIND ALL THE HEADS OF S;
      INTEGER I, J, Z;
      IF NOT GETBIT(INHEAD[S,S DIV 48],ENTIER(S MOD 48)) THEN
      BEGIN SETBIT(INHEAD[S,S DIV 48],ENTIER(S MOD 48));
         FOR I ← 1 STEP 1 UNTIL NPR DO IF PR[I,0] = S THEN
         BEGIN  Z ← PR[I,1];
            HS(Z);
            FOR J ← 1 STEP 1 UNTIL NSY DO
               IF GETBIT(INHEAD[Z,J DIV 48],ENTIER(J MOD 48)) THEN
                  SETBIT(INHEAD[S,J DIV 48],ENTIER(J MOD 48));
         END I;
      END;
   END HS;

   PROCEDURE TS(S);  VALUE S;  INTEGER S;
   BEGIN  COMMENT  FIND ALL THE TAILS OF S;
      INTEGER I, J, Z;
      LABEL F;
      IF NOT GETBIT(INTAIL[S,S DIV 48],ENTIER(S MOD 48)) THEN
      BEGIN SETBIT(INTAIL[S,S DIV 48],ENTIER(S MOD 48));
         FOR I ← 1 STEP 1 UNTIL NPR DO IF PR[I,0] = S THEN
         BEGIN  FOR J ← MAXLPR STEP -1 UNTIL 1 DO
            IF PR[I,J] ≠ 0 THEN GO TO F;
            F:  Z ← PR[I,J];
            TS(Z);
            FOR J ← 1 STEP 1 UNTIL NSY DO
               IF GETBIT(INTAIL[Z,J DIV 48],ENTIER(J MOD 48)) THEN
                  SETBIT(INTAIL[S,J DIV 48],ENTIER(J MOD 48));
         END I;
      END;
   END TS;

   FOR I ← 0 STEP 1 UNTIL NSY DO FOR J ← 0 STEP 1 UNTIL NSY DIV 48
      DO INHEAD[I,J] ← INTAIL[I,J] ← 0;
   FOR I ← 1 STEP 1 UNTIL NSY DO
   BEGIN  HS(I);  TS(I);
```

```
      END;

      PRINTMATRIX(<//"INHEAD:"/>, INHEAD);
      PRINTMATRIX(<//"INTAIL:"/>, INTAIL);
   END BLOCK C1;


   BEGIN  COMMENT  BLOCK C 2;   COMMENT SINGLE CHARACTER DERIVATIVES;
      COMMENT  SCD[I,J] IMPLIES THAT J IS A SINGLE CHARACTER
      DERIVATIVE OF I;
      INTEGER I, J, K;
      BOOLEAN CHANGE;
      FOR I ← 0 STEP 1 UNTIL NSY DO
         FOR J ← 0 STEP 1 UNTIL NSY DIV 48 DO SCD[I,J] ← 0;
      FOR I ← 1 STEP 1 UNTIL NPR DO
         IF PR[I,2] = 0 THEN
            SETBIT(SCD[PR[I,0],PR[I,1] DIV 48],ENTIER(PR[I,1] MOD
            48));
      CHANGE ← TRUE;
      WHILE CHANGE DO
      BEGIN  CHANGE ← FALSE;
         FOR I ← 1 STEP 1 UNTIL NSYN DO
            FOR J ← 1 STEP 1 UNTIL NSYN DO
               IF GETBIT(SCD[I,J DIV 48],ENTIER(J MOD 48)) THEN
                  FOR K ← 1 STEP 1 UNTIL NSY DO
                  IF GETBIT(SCD[J,K DIV 48],ENTIER(K MOD 48)) THEN
                     IF NOT GETBIT(SCD[I,K DIV 48],ENTIER(K MOD 48))
                     THEN
                     BEGIN  CHANGE ← TRUE;
                        SETBIT(SCD[I,K DIV 48],ENTIER(K MOD 48));
                     END;
      END CHANGE;

      PRINTMATRIX(<//"SINGLE CHARACTER DERIVATIVES:"/>, SCD);
   END BLOCK C 2;


   BEGIN  COMMENT  BLOCK C 3;   COMMENT PREDECESSORS AND SUCCESSORS;
      COMMENT  AP[I,J] IMPLIES J IS AN ALLOWED PREDECESSOR OF I;
      INTEGER I, J, P;
      FOR I ← 0 STEP 1 UNTIL NSY DO FOR J ← 0 STEP 1 UNTIL NSY DIV 48
         DO AP[I,J] ← AS[I,J] ← 0;

      FOR P ← 1 STEP 1 UNTIL NSY DO
         FOR I ← 1 STEP 1 UNTIL NPR DO
         BEGIN  COMMENT  PREDECESSORS FIRST;
            FOR J ← 2 STEP 1 UNTIL MAXLPR DO
               IF PR[I,J] = 0 THEN ELSE
               IF GETBIT(INHEAD[PR[I,J],P DIV 48],ENTIER(P MOD 48))
               THEN
               SETBIT(AP[P,PR[I,J-1] DIV 48],ENTIER(PR[I,J-1] MOD
               48));
```

```
            FOR J ← 1 STEP 1 UNTIL MAXLPR-1 DO
               IF PR[I,J+1] = 0 THEN ELSE
               IF GETBIT(INTAIL[PR[I,J],P DIV 48],ENTIER(P MOD 48))
               THEN
               SETBIT(AS[P,PR[I,J+1] DIV 48],ENTIER(PR[I,J+1] MOD
               48));
         END I P;

      PRINTMATRIX(<//"ALLOWED PREDECESSORS:"/>, AP);
      PRINTMATRIX(<//"ALLOWED SUCCESSORS:"/>, AS);
   END BLOCK C3;


   BEGIN  COMMENT  BLOCK C 4;   COMMENT  HIERARCHY ANALYSIS;
      INTEGER A, B, C, P, X, Y, Z, I, I1, I2, I3;

      PROCEDURE T2S(P);  VALUE P;  INTEGER P;
      BEGIN  COMMENT  THE CANONICAL PARSE TAIL 2 SYMBOLS OF P;
         INTEGER I, J, R, X, Y;
         LABEL F;
         BEENTHERE[P] ← TRUE;
         FOR I ← 1 STEP 1 UNTIL NPR DO IF PR[I,0] = P THEN
         BEGIN FOR J ← MAXLPR STEP -1 UNTIL 1 DO IF PR[I,J] ≠ 0 THEN
            GO TO F;
            F:  R ← PR[I,J];
            IF J ≠ 1 THEN
            BEGIN  COMMENT  PRODUCTION LENGTH AT LEAST TWO;
               X ← PR[I, J-1];'
               PUT(X, R);
               FOR Y ← 1 STEP 1 UNTIL NSYN DO
                IF GETBIT(SCD[Y,R DIV 48],ENTIER(R MOD 48)) THEN
                    PUT(X, Y);
            END;
            IF NOT BEENTHERE[R] THEN T2S(R);
         END I;
      END T2S;

      NVAL ← 1;  FINDS ← 0;  CR[1] ← 1024*3;
      FOR I1 ← 1 STEP 1 UNTIL NPR DO IF PR[I1, 2] ≠ 0 THEN
      BEGIN  COMMENT  HIERARCHY ANALYSIS RELATIONS;
         B ← PR[I1, 1];  C ← PR[I1, 2];
         P ← PR[I1, 0];
         FOR I ← 1 STEP 1 UNTIL NSY DO BEENTHERE[I] ← FALSE;
         P2C ← 0;  T2S(B);
         FOR Z ← NSYN + 1 STEP 1 UNTIL NSY DO
         IF GETBIT(INHEAD[C,Z DIV 48],ENTIER(Z MOD 48)) THEN
         BEGIN  COMMENT  Z ARE IN HST(C);
            FOR X ← 1 STEP 1 UNTIL NSY DO
            IF GETBIT(AP[P,X DIV 48],ENTIER(X MOD 48)) THEN
            BEGIN  ENTER(X, B, Z, FALSE);
               IF B ≤ NSYN THEN
               FOR Y ← 1 STEP 1 UNTIL NSY DO
                  IF GETBIT(SCD[B,Y DIV 48],ENTIER(Y MOD 48)) THEN
                     ENTER(X, Y, Z, TRUE);
```

```
         END X;
         FOR I3 ← 1 STEP 1 UNTIL P2C DO
            ENTER(S1[I3], S2[I3], Z, TRUE);
      END Z;

      FOR I2 ← 3 STEP 1 UNTIL MAXLPR DO IF PR[I1, I2] ≠ 0 THEN
      BEGIN
         A ← PR[I1, I2-2];  B ← PR[I1, I2-1];  C ← PR[I1, I2];
         FOR I ← 1 STEP 1 UNTIL NSY DO BEENTHERE[I] ← FALSE;
         P2C ← 0;  T2S(B);
         FOR Z ← NSYN+1 STEP 1 UNTIL NSY DO
         IF GETBIT(INHEAD[C,Z DIV 48],ENTIER(Z MOD 48)) THEN
         BEGIN  ENTER(A, B, Z, FALSE);
            IF B ≤ NSYN THEN
            FOR Y ← 1 STEP 1 UNTIL NSY DO
            IF GETBIT(SCD[B,Y DIV 48],ENTIER(Y MOD 48)) THEN
               ENTER(A, Y, Z, TRUE);
            FOR I3 ← 1 STEP 1 UNTIL P2C DO
               ENTER(S1[I3], S2[I3], Z, TRUE);
         END Z;
      END I2;
   END I1;
   NVAL ← NVAL - 1;

   WRITE(<"HIERARCHY ANALYSIS FUNCTIONS:"/>);
   TABULATE("P1");
END BLOCK C 4;


BEGIN  COMMENT  BLOCK C 5;   COMMENT  PRODUCTION RECOGNITION;
   INTEGER A, B, C, R, P, Y, Z, I, I1, I2, I3;
   LABEL LASTONE;

   PROCEDURE H2S(P);  VALUE P;  INTEGER P;
   BEGIN  COMMENT  THE CANONICAL PARSE HEAD 2 SYMBOLS IN P;
      INTEGER I, R, X, Y, Z;
      BEENTHERE[P] ← TRUE;
      FOR I ← 1 STEP 1 UNTIL NPR DO IF PR[I,0] = P THEN
      BEGIN
         IF PR[I,2] ≠ 0 THEN
         BEGIN  COMMENT  PRODUCTION OF LENGTH AT LEAST TWO;
            X ← PR[I, 1];  R ← PR[I, 2];
            IF GETBIT(INHEAD[R,Y DIV 48],ENTIER(Y MOD 48)) THEN
               PUT(X, Y);
         END;
         R ← PR[I, 1];
         IF NOT BEENTHERE[R] THEN H2S(R);
      END I;
   END H2S;

   NVAL ← 1;  FINDS ← 0;  CR[1] ← 1024*3;
   FOR I1 ← 1 STEP 1 UNTIL NPR DO IF PR[I1, 2] ≠ 0 THEN
   BEGIN  COMMENT  PRODUCTION RECOGNITION RELATIONS;
      FOR I2 ← 2 STEP 1 UNTIL MAXLPR DO
```

```
            BEGIN  A ← PR[I1, I2-1];  B ← PR[I1, I2];
               IF I2 = MAXLPR THEN GO TO LASTONE;
               C ← PR[I1, I2+1];
               IF C = 0 THEN GO TO LASTONE;
               ENTER(A, B, C, FALSE);
               IF B ≤ NSYN THEN
               FOR Y ← 1 STEP 1 UNTIL NSY DO
               IF GETBIT(SCD[B,Y DIV 48],ENTIER(Y MOD 48)) THEN
                  FOR Z ← NSYN+1 STEP 1 UNTIL NSY DO
                     IF GETBIT(INHEAD[C,Z DIV 48],ENTIER(Z MOD 48)) THEN
                        ENTER(A, Y, Z, TRUE);
               FOR I ← 1 STEP 1 UNTIL NSY DO BEENTHERE[I] ← FALSE;
               P2C ← 0;    H2S(B);
               FOR I3 ← 1 STEP 1 UNTIL P2C DO
                  ENTER(A, S1[I3], S2[I3], TRUE);
            END I2;

            LASTONE:
            P ← PR[I1, 0];
            FOR R ← 1 STEP 1 UNTIL NSY DO
            IF GETBIT(AS[P,R DIV 48],ENTIER(R MOD 48)) THEN
               FOR Z ← NSYN+1 STEP 1 UNTIL NSY DO
               IF GETBIT(INHEAD[R,Z DIV 48],ENTIER(Z MOD 48)) THEN
               BEGIN  ENTER(A, B, Z, FALSE);
                  IF B ≤ NSYN THEN
                  FOR Y ← 1 STEP 1 UNTIL NSY DO
                  IF GETBIT(SCD[B,Y DIV 48],ENTIER(Y MOD 48)) THEN
                     ENTER(A, Y, Z, TRUE);
               END Z R;
            FOR I ← 1 STEP 1 UNTIL NSY DO BEENTHERE[I] ← FALSE;
            P2C ← 0;    H2S(B);
            FOR I3 ← 1 STEP 1 UNTIL P2C DO
               ENTER(A, S1[I3], S2[I3], TRUE);
         END I1;
         NVAL ← NVAL - 1;

         WRITE(<"PRODUCTION RECOGNITION FUNCTIONS:"/>);
         TABULATE("P2");
      END BLOCK C 5;
   END BLOCK C;
END;
SAV((T-TI)/3600);  SAV(P2CSAVE);
WRITE(PRINFIL, <9E8.2, "MCKEEMAN">, FOR I ← 1 STEP 1 UNTIL SI DO
   RECORD[I]);
END.
```

```
(2,1)(1,2) SYNTAX PROCESSOR, MCKEEMAN, JAN. 1966

PRODUCTIONS:

         1      <PROGRAM>     →    EOF          <EXPR>       EOF
         2      <EXPR>        →    <IF CLAUSE>  <TRUEPART>   <EXPR>
         3                    →    <SUM>

         4      <TRUEPART>    →    <EXPR>       ELSE
         5      <IF CLAUSE>   →    IF           <EXPR>       THEN

         6      <SUM>         →    <SUM>        +            <PRIMARY>
         7                    →    <SUM>        -            <PRIMARY>
         8                    →    +            <PRIMARY>
         9                    →    -            <PRIMARY>
        10                    →    <PRIMARY>
        11      <PRIMARY>     →    IDENT
        12                    →    INTEGER
        13                    →    (            <EXPR>       )


INTERMEDIATE SYMBOLS:
         1   <PROGRAM>            2    <EXPR>              3   <TRUEPART>
         4   <IF CLAUSE>          5    <SUM>               6   <PRIMARY>

TERMINAL SYMBOLS:
         7   EOF                  8    IF                  9   +
        10   -                   11    IDENT              12   INTEGER
        13   (                   14    ELSE               15   THEN
        16   )

THE UNIQUE TARGET SYMBOL IS:  <PROGRAM>
TIME =   0.12, TOTAL ELAPSED =    0.12 MIN.
```

```
INHEAD:

      SYMBOL          1234567890123456
  1   <PROGRAM>       .     .
  2   <EXPR>           . ... ......
  3   <TRUEPART>       ..... ......
  4   <IF CLAUSE>        .   .
  5   <SUM>               ..  .....
  6   <PRIMARY>            .    ...
  7   EOF                   .
  8   IF                     .
  9   +                       .
 10   -                        .
 11   IDENT                     .
 12   INTEGER                    .
 13   (                           .
 14   ELSE                         .
 15   THEN                          .
 16   )                              .
TIME =   0.04, TOTAL ELAPSED =    0.16 MIN.
```

```
INTAIL:

      SYMBOL          1234567890123456
  1   <PROGRAM>       .     .
  2   <EXPR>           .  ..    ..   .
  3   <TRUEPART>        .          .
  4   <IF CLAUSE>        .          .
  5   <SUM>               ..    ..   .
  6   <PRIMARY>            .    ..   .
  7   EOF                   .
  8   IF                     .
  9   +                       .
 10   -                        .
 11   IDENT                     .
 12   INTEGER                    .
 13   (                           .
 14   ELSE                         .
 15   THEN                          .
 16   )                              .
TIME =   0.04, TOTAL ELAPSED =    0.20 MIN.
```

```
SINGLE CHARACTER DERIVATIVES:

            SYMBOL       1234567890123456
  1    <PROGRAM>
  2    <EXPR>                ..    ..
  3    <TRUEPART>
  4    <IF CLAUSE>
  5    <SUM>                  .    ..
  6    <PRIMARY>                   ..
  7    EOF
  8    IF
  9    +
 10    =
 11    IDENT
 12    INTEGER
 13    (
 14    ELSE
 15    THEN
 16    )
TIME =   0.04, TOTAL ELAPSED =     0.24 MIN.
```

```
ALLOWED PREDECESSORS:

         SYMBOL      1234567890123456
   1   <PROGRAM>
   2   <EXPR>          ..  ..    .
   3   <TRUEPART>       .
   4   <IF CLAUSE>     ..  ..    .
   5   <SUM>           ..  ..    .
   6   <PRIMARY>       ..  ....  .
   7   EOF            .
   8   IF              ..  ..    .
   9   +               ... ..    .
  10   -               ... ..    .
  11   IDENT           ..  ....  .
  12   INTEGER         ..  ....  .
  13   (               ..  ....  .
  14   ELSE           .
  15   THEN           .
  16   )              .
TIME =    0.04, TOTAL ELAPSED =     0.28 MIN.
```

```
ALLOWED SUCCESSORS:

         SYMBOL      1234567890123456
   1   <PROGRAM>
   2   <EXPR>              .      ...
   3   <TRUEPART>     .
   4   <IF CLAUSE>     .
   5   <SUM>               . ..   ...
   6   <PRIMARY>           . ..   ...
   7   EOF            .
   8   IF             .
   9   +                  .
  10   -                  .
  11   IDENT               . ..   ...
  12   INTEGER             . ..   ...
  13   (              .
  14   ELSE           .
  15   THEN            .
  16   )                   . ..   ...
TIME =    0.04, TOTAL ELAPSED =     0.32 MIN.
```

HIERARCHY ANALYSIS FUNCTIONS:

```
P1"( <EXPR>       ELSE           IF        ) = P1"( 2, 14,  8) = TRUE
P1"( <EXPR>       ELSE           +         ) = P1"( 2, 14,  9) = TRUE
P1"( <EXPR>       ELSE           -         ) = P1"( 2, 14, 10) = TRUE
P1"( <EXPR>       ELSE           IDENT     ) = P1"( 2, 14, 11) = TRUE
P1"( <EXPR>       ELSE           INTEGER   ) = P1"( 2, 14, 12) = TRUE
P1"( <EXPR>       ELSE           (         ) = P1"( 2, 14, 13) = TRUE
P1"( <EXPR>       THEN           IF        ) = P1"( 2, 15,  8) = TRUE
P1"( <EXPR>       THEN           +         ) = P1"( 2, 15,  9) = TRUE
P1"( <EXPR>       THEN           -         ) = P1"( 2, 15, 10) = TRUE
P1"( <EXPR>       THEN           IDENT     ) = P1"( 2, 15, 11) = TRUE
P1"( <EXPR>       THEN           INTEGER   ) = P1"( 2, 15, 12) = TRUE
P1"( <EXPR>       THEN           (         ) = P1"( 2, 15, 13) = TRUE
P1"( <EXPR>       )              EOF       ) = P1"( 2, 16,  7) = TRUE
P1"( <EXPR>       )              +         ) = P1"( 2, 16,  9) = TRUE
P1"( <EXPR>       )              -         ) = P1"( 2, 16, 10) = TRUE
P1"( <EXPR>       )              ELSE      ) = P1"( 2, 16, 14) = TRUE
P1"( <EXPR>       )              THEN      ) = P1"( 2, 16, 15) = TRUE
P1"( <EXPR>       )              )         ) = P1"( 2, 16, 16) = TRUE
P1"( <TRUEPART>   <EXPR>         EOF       ) = P1"( 3,  2,  7) = TRUE
P1"( <TRUEPART>   <EXPR>         ELSE      ) = P1"( 3,  2, 14) = TRUE
P1"( <TRUEPART>   <EXPR>         THEN      ) = P1"( 3,  2, 15) = TRUE
P1"( <TRUEPART>   <EXPR>         )         ) = P1"( 3,  2, 16) = TRUE
P1"( <TRUEPART>   <IF CLAUSE>    IF        ) = P1"( 3,  4,  8) = FALSE
P1"( <TRUEPART>   <IF CLAUSE>    +         ) = P1"( 3,  4,  9) = FALSE
P1"( <TRUEPART>   <IF CLAUSE>    -         ) = P1"( 3,  4, 10) = FALSE
P1"( <TRUEPART>   <IF CLAUSE>    IDENT     ) = P1"( 3,  4, 11) = FALSE
P1"( <TRUEPART>   <IF CLAUSE>    INTEGER   ) = P1"( 3,  4, 12) = FALSE
P1"( <TRUEPART>   <IF CLAUSE>    (         ) = P1"( 3,  4, 13) = FALSE
P1"( <TRUEPART>   <SUM>          +         ) = P1"( 3,  5,  9) = FALSE
P1"( <TRUEPART>   <SUM>          -         ) = P1"( 3,  5, 10) = FALSE
P1"( <TRUEPART>   <PRIMARY>      +         ) = P1"( 3,  6,  9) = TRUE
P1"( <TRUEPART>   <PRIMARY>      -         ) = P1"( 3,  6, 10) = TRUE
P1"( <TRUEPART>   IF             IF        ) = P1"( 3,  8,  8) = FALSE
P1"( <TRUEPART>   IF             +         ) = P1"( 3,  8,  9) = FALSE
P1"( <TRUEPART>   IF             -         ) = P1"( 3,  8, 10) = FALSE
P1"( <TRUEPART>   IF             IDENT     ) = P1"( 3,  8, 11) = FALSE
P1"( <TRUEPART>   IF             INTEGER   ) = P1"( 3,  8, 12) = FALSE
P1"( <TRUEPART>   IF             (         ) = P1"( 3,  8, 13) = FALSE
P1"( <TRUEPART>   +              IDENT     ) = P1"( 3,  9, 11) = FALSE
P1"( <TRUEPART>   +              INTEGER   ) = P1"( 3,  9, 12) = FALSE
P1"( <TRUEPART>   +              (         ) = P1"( 3,  9, 13) = FALSE
P1"( <TRUEPART>   -              IDENT     ) = P1"( 3, 10, 11) = FALSE
P1"( <TRUEPART>   -              INTEGER   ) = P1"( 3, 10, 12) = FALSE
P1"( <TRUEPART>   -              (         ) = P1"( 3, 10, 13) = FALSE
P1"( <TRUEPART>   IDENT          +         ) = P1"( 3, 11,  9) = TRUE
P1"( <TRUEPART>   IDENT          -         ) = P1"( 3, 11, 10) = TRUE
P1"( <TRUEPART>   INTEGER        +         ) = P1"( 3, 12,  9) = TRUE
P1"( <TRUEPART>   INTEGER        -         ) = P1"( 3, 12, 10) = TRUE
P1"( <TRUEPART>   (              IF        ) = P1"( 3, 13,  8) = FALSE
P1"( <TRUEPART>   (              +         ) = P1"( 3, 13,  9) = FALSE
P1"( <TRUEPART>   (              -         ) = P1"( 3, 13, 10) = FALSE
P1"( <TRUEPART>   (              IDENT     ) = P1"( 3, 13, 11) = FALSE
P1"( <TRUEPART>   (              INTEGER   ) = P1"( 3, 13, 12) = FALSE
```

```
P1"( <TRUEPART>  (            (          ) = P1"(  3, 13, 13) = FALSE
P1"( <IF CLAUSE> <EXPR>       ELSE       ) = P1"(  4,  2, 14) = FALSE
P1"( <IF CLAUSE> <TRUEPART>   IF         ) = P1"(  4,  3,  8) = FALSE
P1"( <IF CLAUSE> <TRUEPART>   +          ) = P1"(  4,  3,  9) = FALSE
P1"( <IF CLAUSE> <TRUEPART>   -          ) = P1"(  4,  3, 10) = FALSE
P1"( <IF CLAUSE> <TRUEPART>   IDENT      ) = P1"(  4,  3, 11) = FALSE
P1"( <IF CLAUSE> <TRUEPART>   INTEGER    ) = P1"(  4,  3, 12) = FALSE
P1"( <IF CLAUSE> <TRUEPART>   (          ) = P1"(  4,  3, 13) = FALSE
P1"( <IF CLAUSE> <IF CLAUSE>  IF         ) = P1"(  4,  4,  8) = FALSE
P1"( <IF CLAUSE> <IF CLAUSE>  +          ) = P1"(  4,  4,  9) = FALSE
P1"( <IF CLAUSE> <IF CLAUSE>  -          ) = P1"(  4,  4, 10) = FALSE
P1"( <IF CLAUSE> <IF CLAUSE>  IDENT      ) = P1"(  4,  4, 11) = FALSE
P1"( <IF CLAUSE> <IF CLAUSE>  INTEGER    ) = P1"(  4,  4, 12) = FALSE
P1"( <IF CLAUSE> <IF CLAUSE>  (          ) = P1"(  4,  4, 13) = FALSE
P1"( <IF CLAUSE> <SUM>        +          ) = P1"(  4,  5,  9) = FALSE
P1"( <IF CLAUSE> <SUM>        -          ) = P1"(  4,  5, 10) = FALSE
P1"( <IF CLAUSE> <SUM>        ELSE       ) = P1"(  4,  5, 14) = TRUE
P1"( <IF CLAUSE> <PRIMARY>    +          ) = P1"(  4,  6,  9) = TRUE
P1"( <IF CLAUSE> <PRIMARY>    -          ) = P1"(  4,  6, 10) = TRUE
P1"( <IF CLAUSE> <PRIMARY>    ELSE       ) = P1"(  4,  6, 14) = TRUE
P1"( <IF CLAUSE> IF           IF         ) = P1"(  4,  8,  8) = FALSE
P1"( <IF CLAUSE> IF           +          ) = P1"(  4,  8,  9) = FALSE
P1"( <IF CLAUSE> IF           -          ) = P1"(  4,  8, 10) = FALSE
P1"( <IF CLAUSE> IF           IDENT      ) = P1"(  4,  8, 11) = FALSE
P1"( <IF CLAUSE> IF           INTEGER    ) = P1"(  4,  8, 12) = FALSE
P1"( <IF CLAUSE> IF           (          ) = P1"(  4,  8, 13) = FALSE
P1"( <IF CLAUSE> +            IDENT      ) = P1"(  4,  9, 11) = FALSE
P1"( <IF CLAUSE> +            INTEGER    ) = P1"(  4,  9, 12) = FALSE
P1"( <IF CLAUSE> +            (          ) = P1"(  4,  9, 13) = FALSE
P1"( <IF CLAUSE> -            IDENT      ) = P1"(  4, 10, 11) = FALSE
P1"( <IF CLAUSE> -            INTEGER    ) = P1"(  4, 10, 12) = FALSE
P1"( <IF CLAUSE> -            (          ) = P1"(  4, 10, 13) = FALSE
P1"( <IF CLAUSE> IDENT        +          ) = P1"(  4, 11,  9) = TRUE
P1"( <IF CLAUSE> IDENT        -          ) = P1"(  4, 11, 10) = TRUE
P1"( <IF CLAUSE> IDENT        ELSE       ) = P1"(  4, 11, 14) = TRUE
P1"( <IF CLAUSE> INTEGER      +          ) = P1"(  4, 12,  9) = TRUE
P1"( <IF CLAUSE> INTEGER      -          ) = P1"(  4, 12, 10) = TRUE
P1"( <IF CLAUSE> INTEGER      ELSE       ) = P1"(  4, 12, 14) = TRUE
P1"( <IF CLAUSE> (            IF         ) = P1"(  4, 13,  8) = FALSE
P1"( <IF CLAUSE> (            +          ) = P1"(  4, 13,  9) = FALSE
P1"( <IF CLAUSE> (            -          ) = P1"(  4, 13, 10) = FALSE
P1"( <IF CLAUSE> (            IDENT      ) = P1"(  4, 13, 11) = FALSE
P1"( <IF CLAUSE> (            INTEGER    ) = P1"(  4, 13, 12) = FALSE
P1"( <IF CLAUSE> (            (          ) = P1"(  4, 13, 13) = FALSE
P1"( <SUM>       +            IDENT      ) = P1"(  5,  9, 11) = FALSE
P1"( <SUM>       +            INTEGER    ) = P1"(  5,  9, 12) = FALSE
P1"( <SUM>       +            (          ) = P1"(  5,  9, 13) = FALSE
P1"( <SUM>       -            IDENT      ) = P1"(  5, 10, 11) = FALSE
P1"( <SUM>       -            INTEGER    ) = P1"(  5, 10, 12) = FALSE
P1"( <SUM>       -            (          ) = P1"(  5, 10, 13) = FALSE
P1"( EOF         <EXPR>       EOF        ) = P1"(  7,  2,  7) = FALSE
P1"( EOF         <IF CLAUSE>  IF         ) = P1"(  7,  4,  8) = FALSE
P1"( EOF         <IF CLAUSE>  +          ) = P1"(  7,  4,  9) = FALSE
P1"( EOF         <IF CLAUSE>  -          ) = P1"(  7,  4, 10) = FALSE
P1"( EOF         <IF CLAUSE>  IDENT      ) = P1"(  7,  4, 11) = FALSE
```

P1"(	EOF	<IF CLAUSE>	INTEGER	)	=	P1"(	7,	4,	12)	=	FALSE
P1"(	EOF	<IF CLAUSE>	(	)	=	P1"(	7,	4,	13)	=	FALSE
P1"(	EOF	<SUM>	EOF	)	=	P1"(	7,	5,	7)	=	TRUE
P1"(	EOF	<SUM>	+	)	=	P1"(	7,	5,	9)	=	FALSE
P1"(	EOF	<SUM>	-	)	=	P1"(	7,	5,	10)	=	FALSE
P1"(	EOF	<PRIMARY>	EOF	)	=	P1"(	7,	6,	7)	=	TRUE
P1"(	EOF	<PRIMARY>	+	)	=	P1"(	7,	6,	9)	=	TRUE
P1"(	EOF	<PRIMARY>	-	)	=	P1"(	7,	6,	10)	=	TRUE
P1"(	EOF	IF	IF	)	=	P1"(	7,	8,	8)	=	FALSE
P1"(	EOF	IF	+	)	=	P1"(	7,	8,	9)	=	FALSE
P1"(	EOF	IF	-	)	=	P1"(	7,	8,	10)	=	FALSE
P1"(	EOF	IF	IDENT	)	=	P1"(	7,	8,	11)	=	FALSE
P1"(	EOF	IF	INTEGER	)	=	P1"(	7,	8,	12)	=	FALSE
P1"(	EOF	IF	(	)	=	P1"(	7,	8,	13)	=	FALSE
P1"(	EOF	+	IDENT	)	=	P1"(	7,	9,	11)	=	FALSE
P1"(	EOF	+	INTEGER	)	=	P1"(	7,	9,	12)	=	FALSE
P1"(	EOF	+	(	)	=	P1"(	7,	9,	13)	=	FALSE
P1"(	EOF	-	IDENT	)	=	P1"(	7,	10,	11)	=	FALSE
P1"(	EOF	-	INTEGER	)	=	P1"(	7,	10,	12)	=	FALSE
P1"(	EOF	-	(	)	=	P1"(	7,	10,	13)	=	FALSE
P1"(	EOF	IDENT	EOF	)	=	P1"(	7,	11,	7)	=	TRUE
P1"(	EOF	IDENT	+	)	=	P1"(	7,	11,	9)	=	TRUE
P1"(	EOF	IDENT	-	)	=	P1"(	7,	11,	10)	=	TRUE
P1"(	EOF	INTEGER	EOF	)	=	P1"(	7,	12,	7)	=	TRUE
P1"(	EOF	INTEGER	+	)	=	P1"(	7,	12,	9)	=	TRUE
P1"(	EOF	INTEGER	-	)	=	P1"(	7,	12,	10)	=	TRUE
P1"(	EOF	(	IF	)	=	P1"(	7,	13,	8)	=	FALSE
P1"(	EOF	(	+	)	=	P1"(	7,	13,	9)	=	FALSE
P1"(	EOF	(	-	)	=	P1"(	7,	13,	10)	=	FALSE
P1"(	EOF	(	IDENT	)	=	P1"(	7,	13,	11)	=	FALSE
P1"(	EOF	(	INTEGER	)	=	P1"(	7,	13,	12)	=	FALSE
P1"(	EOF	(	(	)	=	P1"(	7,	13,	13)	=	FALSE
P1"(	IF	<EXPR>	THEN	)	=	P1"(	8,	2,	15)	=	FALSE
P1"(	IF	<IF CLAUSE>	IF	)	=	P1"(	8,	4,	8)	=	FALSE
P1"(	IF	<IF CLAUSE>	+	)	=	P1"(	8,	4,	9)	=	FALSE
P1"(	IF	<IF CLAUSE>	-	)	=	P1"(	8,	4,	10)	=	FALSE
P1"(	IF	<IF CLAUSE>	IDENT	)	=	P1"(	8,	4,	11)	=	FALSE
P1"(	IF	<IF CLAUSE>	INTEGER	)	=	P1"(	8,	4,	12)	=	FALSE
P1"(	IF	<IF CLAUSE>	(	)	=	P1"(	8,	4,	13)	=	FALSE
P1"(	IF	<SUM>	+	)	=	P1"(	8,	5,	9)	=	FALSE
P1"(	IF	<SUM>	-	)	=	P1"(	8,	5,	10)	=	FALSE
P1"(	IF	<SUM>	THEN	)	=	P1"(	8,	5,	15)	=	TRUE
P1"(	IF	<PRIMARY>	+	)	=	P1"(	8,	6,	9)	=	TRUE
P1"(	IF	<PRIMARY>	-	)	=	P1"(	8,	6,	10)	=	TRUE
P1"(	IF	<PRIMARY>	THEN	)	=	P1"(	8,	6,	15)	=	TRUE
P1"(	IF	IF	IF	)	=	P1"(	8,	8,	8)	=	FALSE
P1"(	IF	IF	+	)	=	P1"(	8,	8,	9)	=	FALSE
P1"(	IF	IF	-	)	=	P1"(	8,	8,	10)	=	FALSE
P1"(	IF	IF	IDENT	)	=	P1"(	8,	8,	11)	=	FALSE
P1"(	IF	IF	INTEGER	)	=	P1"(	8,	8,	12)	=	FALSE
P1"(	IF	IF	(	)	=	P1"(	8,	8,	13)	=	FALSE
P1"(	IF	+	IDENT	)	=	P1"(	8,	9,	11)	=	FALSE
P1"(	IF	+	INTEGER	)	=	P1"(	8,	9,	12)	=	FALSE
P1"(	IF	+	(	)	=	P1"(	8,	9,	13)	=	FALSE
P1"(	IF	-	IDENT	)	=	P1"(	8,	10,	11)	=	FALSE

```
P1"( IF        -          INTEGER    ) = P1"(  8, 10, 12) = FALSE
P1"( IF        -          (          ) = P1"(  8, 10, 13) = FALSE
P1"( IF        IDENT      +          ) = P1"(  8, 11,  9) = TRUE
P1"( IF        IDENT      -          ) = P1"(  8, 11, 10) = TRUE
P1"( IF        IDENT      THEN       ) = P1"(  8, 11, 15) = TRUE
P1"( IF        INTEGER    +          ) = P1"(  8, 12,  9) = TRUE
P1"( IF        INTEGER    -          ) = P1"(  8, 12, 10) = TRUE
P1"( IF        INTEGER    THEN       ) = P1"(  8, 12, 15) = TRUE
P1"( IF        (          IF         ) = P1"(  8, 13,  8) = FALSE
P1"( IF        (          +          ) = P1"(  8, 13,  9) = FALSE
P1"( IF        (          -          ) = P1"(  8, 13, 10) = FALSE
P1"( IF        (          IDENT      ) = P1"(  8, 13, 11) = FALSE
P1"( IF        (          INTEGER    ) = P1"(  8, 13, 12) = FALSE
P1"( IF        (          (          ) = P1"(  8, 13, 13) = FALSE
P1"( +         <EXPR>     EOF        ) = P1"(  9,  2,  7) = TRUE
P1"( +         <EXPR>     +          ) = P1"(  9,  2,  9) = TRUE
P1"( +         <EXPR>     -          ) = P1"(  9,  2, 10) = TRUE
P1"( +         <EXPR>     ELSE       ) = P1"(  9,  2, 14) = TRUE
P1"( +         <EXPR>     THEN       ) = P1"(  9,  2, 15) = TRUE
P1"( +         <EXPR>     )          ) = P1"(  9,  2, 16) = TRUE
P1"( +         <SUM>      EOF        ) = P1"(  9,  5,  7) = TRUE
P1"( +         <SUM>      +          ) = P1"(  9,  5,  9) = TRUE
P1"( +         <SUM>      -          ) = P1"(  9,  5, 10) = TRUE
P1"( +         <SUM>      ELSE       ) = P1"(  9,  5, 14) = TRUE
P1"( +         <SUM>      THEN       ) = P1"(  9,  5, 15) = TRUE
P1"( +         <SUM>      )          ) = P1"(  9,  5, 16) = TRUE
P1"( +         <PRIMARY>  EOF        ) = P1"(  9,  6,  7) = TRUE
P1"( +         <PRIMARY>  +          ) = P1"(  9,  6,  9) = TRUE
P1"( +         <PRIMARY>  -          ) = P1"(  9,  6, 10) = TRUE
P1"( +         <PRIMARY>  ELSE       ) = P1"(  9,  6, 14) = TRUE
P1"( +         <PRIMARY>  THEN       ) = P1"(  9,  6, 15) = TRUE
P1"( +         <PRIMARY>  )          ) = P1"(  9,  6, 16) = TRUE
P1"( +         (          IF         ) = P1"(  9, 13,  8) = FALSE
P1"( +         (          +          ) = P1"(  9, 13,  9) = FALSE
P1"( +         (          -          ) = P1"(  9, 13, 10) = FALSE
P1"( +         (          IDENT      ) = P1"(  9, 13, 11) = FALSE
P1"( +         (          INTEGER    ) = P1"(  9, 13, 12) = FALSE
P1"( +         (          (          ) = P1"(  9, 13, 13) = FALSE
P1"( -         <EXPR>     EOF        ) = P1"( 10,  2,  7) = TRUE
P1"( -         <EXPR>     +          ) = P1"( 10,  2,  9) = TRUE
P1"( -         <EXPR>     -          ) = P1"( 10,  2, 10) = TRUE
P1"( -         <EXPR>     ELSE       ) = P1"( 10,  2, 14) = TRUE
P1"( -         <EXPR>     THEN       ) = P1"( 10,  2, 15) = TRUE
P1"( -         <EXPR>     )          ) = P1"( 10,  2, 16) = TRUE
P1"( -         <SUM>      EOF        ) = P1"( 10,  5,  7) = TRUE
P1"( -         <SUM>      +          ) = P1"( 10,  5,  9) = TRUE
P1"( -         <SUM>      -          ) = P1"( 10,  5, 10) = TRUE
P1"( -         <SUM>      ELSE       ) = P1"( 10,  5, 14) = TRUE
P1"( -         <SUM>      THEN       ) = P1"( 10,  5, 15) = TRUE
P1"( -         <SUM>      )          ) = P1"( 10,  5, 16) = TRUE
P1"( -         <PRIMARY>  EOF        ) = P1"( 10,  6,  7) = TRUE
P1"( -         <PRIMARY>  +          ) = P1"( 10,  6,  9) = TRUE
P1"( -         <PRIMARY>  -          ) = P1"( 10,  6, 10) = TRUE
P1"( -         <PRIMARY>  ELSE       ) = P1"( 10,  6, 14) = TRUE
P1"( -         <PRIMARY>  THEN       ) = P1"( 10,  6, 15) = TRUE
```

```
P1"( -          <PRIMARY>     )              ) =  P1"( 10,  6, 16)  =  TRUE
P1"( -          (             IF             ) =  P1"( 10, 13,  8)  =  FALSE
P1"( -          (             +              ) =  P1"( 10, 13,  9)  =  FALSE
P1"( -          (             -              ) =  P1"( 10, 13, 10)  =  FALSE
P1"( -          (             IDENT          ) =  P1"( 10, 13, 11)  =  FALSE
P1"( -          (             INTEGER        ) =  P1"( 10, 13, 12)  =  FALSE
P1"( -          (             (              ) =  P1"( 10, 13, 13)  =  FALSE
P1"( (          <EXPR>        )              ) =  P1"( 13,  2, 16)  =  FALSE
P1"( (          <IF CLAUSE>   IF             ) =  P1"( 13,  4,  8)  =  FALSE
P1"( (          <IF CLAUSE>   +              ) =  P1"( 13,  4,  9)  =  FALSE
P1"( (          <IF CLAUSE>   -              ) =  P1"( 13,  4, 10)  =  FALSE
P1"( (          <IF CLAUSE>   IDENT          ) =  P1"( 13,  4, 11)  =  FALSE
P1"( (          <IF CLAUSE>   INTEGER        ) =  P1"( 13,  4, 12)  =  FALSE
P1"( (          <IF CLAUSE>   (              ) =  P1"( 13,  4, 13)  =  FALSE
P1"( (          <SUM>         +              ) =  P1"( 13,  5,  9)  =  FALSE
P1"( (          <SUM>         -              ) =  P1"( 13,  5, 10)  =  FALSE
P1"( (          <SUM>         )              ) =  P1"( 13,  5, 16)  =  TRUE
P1"( (          <PRIMARY>     +              ) =  P1"( 13,  6,  9)  =  TRUE
P1"( (          <PRIMARY>     -              ) =  P1"( 13,  6, 10)  =  TRUE
P1"( (          <PRIMARY>     )              ) =  P1"( 13,  6, 16)  =  TRUE
P1"( (          IF            IF             ) =  P1"( 13,  8,  8)  =  FALSE
P1"( (          IF            +              ) =  P1"( 13,  8,  9)  =  FALSE
P1"( (          IF            -              ) =  P1"( 13,  8, 10)  =  FALSE
P1"( (          IF            IDENT          ) =  P1"( 13,  8, 11)  =  FALSE
P1"( (          IF            INTEGER        ) =  P1"( 13,  8, 12)  =  FALSE
P1"( (          IF            (              ) =  P1"( 13,  8, 13)  =  FALSE
P1"( (          +             IDENT          ) =  P1"( 13,  9, 11)  =  FALSE
P1"( (          +             INTEGER        ) =  P1"( 13,  9, 12)  =  FALSE
P1"( (          +             (              ) =  P1"( 13,  9, 13)  =  FALSE
P1"( (          -             IDENT          ) =  P1"( 13, 10, 11)  =  FALSE
P1"( (          -             INTEGER        ) =  P1"( 13, 10, 12)  =  FALSE
P1"( (          -             (              ) =  P1"( 13, 10, 13)  =  FALSE
P1"( (          IDENT         +              ) =  P1"( 13, 11,  9)  =  TRUE
P1"( (          IDENT         -              ) =  P1"( 13, 11, 10)  =  TRUE
P1"( (          IDENT         )              ) =  P1"( 13, 11, 16)  =  TRUE
P1"( (          INTEGER       +              ) =  P1"( 13, 12,  9)  =  TRUE
P1"( (          INTEGER       -              ) =  P1"( 13, 12, 10)  =  TRUE
P1"( (          INTEGER       )              ) =  P1"( 13, 12, 16)  =  TRUE
P1"( (          (             IF             ) =  P1"( 13, 13,  8)  =  FALSE
P1"( (          (             +              ) =  P1"( 13, 13,  9)  =  FALSE
P1"( (          (             -              ) =  P1"( 13, 13, 10)  =  FALSE
P1"( (          (             IDENT          ) =  P1"( 13, 13, 11)  =  FALSE
P1"( (          (             INTEGER        ) =  P1"( 13, 13, 12)  =  FALSE
P1"( (          (             (              ) =  P1"( 13, 13, 13)  =  FALSE
  262 FUNCTION VALUES, DENSITY =  6.40%, ENTRIES/VALUE  1.00
TIME =   0.18, TOTAL ELAPSED =    0.37 MIN.
```

PRODUCTION RECOGNITION FUNCTIONS:

```
P2"( <EXPR>        ELSE        IF          ) = P2"( 2, 14,  8) = FALSE
P2"( <EXPR>        ELSE        +           ) = P2"( 2, 14,  9) = FALSE
P2"( <EXPR>        ELSE        -           ) = P2"( 2, 14, 10) = FALSE
P2"( <EXPR>        ELSE        IDENT       ) = P2"( 2, 14, 11) = FALSE
P2"( <EXPR>        ELSE        INTEGER     ) = P2"( 2, 14, 12) = FALSE
P2"( <EXPR>        ELSE        (           ) = P2"( 2, 14, 13) = FALSE
P2"( <EXPR>        THEN        IF          ) = P2"( 2, 15,  8) = FALSE
P2"( <EXPR>        THEN        +           ) = P2"( 2, 15,  9) = FALSE
P2"( <EXPR>        THEN        -           ) = P2"( 2, 15, 10) = FALSE
P2"( <EXPR>        THEN        IDENT       ) = P2"( 2, 15, 11) = FALSE
P2"( <EXPR>        THEN        INTEGER     ) = P2"( 2, 15, 12) = FALSE
P2"( <EXPR>        THEN        (           ) = P2"( 2, 15, 13) = FALSE
P2"( <EXPR>        )           EOF         ) = P2"( 2, 16,  7) = FALSE
P2"( <EXPR>        )           +           ) = P2"( 2, 16,  9) = FALSE
P2"( <EXPR>        )           -           ) = P2"( 2, 16, 10) = FALSE
P2"( <EXPR>        )           ELSE        ) = P2"( 2, 16, 14) = FALSE
P2"( <EXPR>        )           THEN        ) = P2"( 2, 16, 15) = FALSE
P2"( <EXPR>        )           )           ) = P2"( 2, 16, 16) = FALSE
P2"( <TRUEPART>    <EXPR>      EOF         ) = P2"( 3,  2,  7) = FALSE
P2"( <TRUEPART>    <EXPR>      ELSE        ) = P2"( 3,  2, 14) = FALSE
P2"( <TRUEPART>    <EXPR>      THEN        ) = P2"( 3,  2, 15) = FALSE
P2"( <TRUEPART>    <EXPR>      )           ) = P2"( 3,  2, 16) = FALSE
P2"( <TRUEPART>    <SUM>       EOF         ) = P2"( 3,  5,  7) = TRUE
P2"( <TRUEPART>    <SUM>       ELSE        ) = P2"( 3,  5, 14) = TRUE
P2"( <TRUEPART>    <SUM>       THEN        ) = P2"( 3,  5, 15) = TRUE
P2"( <TRUEPART>    <SUM>       )           ) = P2"( 3,  5, 16) = TRUE
P2"( <TRUEPART>    <PRIMARY>   EOF         ) = P2"( 3,  6,  7) = TRUE
P2"( <TRUEPART>    <PRIMARY>   ELSE        ) = P2"( 3,  6, 14) = TRUE
P2"( <TRUEPART>    <PRIMARY>   THEN        ) = P2"( 3,  6, 15) = TRUE
P2"( <TRUEPART>    <PRIMARY>   )           ) = P2"( 3,  6, 16) = TRUE
P2"( <TRUEPART>    IDENT       EOF         ) = P2"( 3, 11,  7) = TRUE
P2"( <TRUEPART>    IDENT       ELSE        ) = P2"( 3, 11, 14) = TRUE
P2"( <TRUEPART>    IDENT       THEN        ) = P2"( 3, 11, 15) = TRUE
P2"( <TRUEPART>    IDENT       )           ) = P2"( 3, 11, 16) = TRUE
P2"( <TRUEPART>    INTEGER     EOF         ) = P2"( 3, 12,  7) = TRUE
P2"( <TRUEPART>    INTEGER     ELSE        ) = P2"( 3, 12, 14) = TRUE
P2"( <TRUEPART>    INTEGER     THEN        ) = P2"( 3, 12, 15) = TRUE
P2"( <TRUEPART>    INTEGER     )           ) = P2"( 3, 12, 16) = TRUE
P2"( <IF CLAUSE> <TRUEPART>    <EXPR>      ) = P2"( 4,  3,  2) = FALSE
P2"( <SUM>         +           <PRIMARY>   ) = P2"( 5,  9,  6) = FALSE
P2"( <SUM>         -           <PRIMARY>   ) = P2"( 5, 10,  6) = FALSE
P2"( EOF           <EXPR>      EOF         ) = P2"( 7,  2,  7) = FALSE
P2"( EOF           <SUM>       EOF         ) = P2"( 7,  5,  7) = TRUE
P2"( EOF           <PRIMARY>   EOF         ) = P2"( 7,  6,  7) = TRUE
P2"( EOF           IDENT       EOF         ) = P2"( 7, 11,  7) = TRUE
P2"( EOF           INTEGER     EOF         ) = P2"( 7, 12,  7) = TRUE
P2"( IF            <EXPR>      THEN        ) = P2"( 8,  2, 15) = FALSE
P2"( IF            <SUM>       THEN        ) = P2"( 8,  5, 15) = TRUE
P2"( IF            <PRIMARY>   THEN        ) = P2"( 8,  6, 15) = TRUE
P2"( IF            IDENT       THEN        ) = P2"( 8, 11, 15) = TRUE
P2"( IF            INTEGER     THEN        ) = P2"( 8, 12, 15) = TRUE
P2"( +             <PRIMARY>   EOF         ) = P2"( 9,  6,  7) = FALSE
P2"( +             <PRIMARY>   +           ) = P2"( 9,  6,  9) = FALSE
```

```
P2"( +          <PRIMARY>   -        ) = P2"(  9,  6, 10) = FALSE
P2"( +          <PRIMARY>   ELSE     ) = P2"(  9,  6, 14) = FALSE
P2"( +          <PRIMARY>   THEN     ) = P2"(  9,  6, 15) = FALSE
P2"( +          <PRIMARY>   )        ) = P2"(  9,  6, 16) = FALSE
P2"( +          IDENT       EOF      ) = P2"(  9, 11,  7) = TRUE
P2"( +          IDENT       +        ) = P2"(  9, 11,  9) = TRUE
P2"( +          IDENT       -        ) = P2"(  9, 11, 10) = TRUE
P2"( +          IDENT       ELSE     ) = P2"(  9, 11, 14) = TRUE
P2"( +          IDENT       THEN     ) = P2"(  9, 11, 15) = TRUE
P2"( +          IDENT       )        ) = P2"(  9, 11, 16) = TRUE
P2"( +          INTEGER     EOF      ) = P2"(  9, 12,  7) = TRUE
P2"( +          INTEGER     +        ) = P2"(  9, 12,  9) = TRUE
P2"( +          INTEGER     -        ) = P2"(  9, 12, 10) = TRUE
P2"( +          INTEGER     ELSE     ) = P2"(  9, 12, 14) = TRUE
P2"( +          INTEGER     THEN     ) = P2"(  9, 12, 15) = TRUE
P2"( +          INTEGER     )        ) = P2"(  9, 12, 16) = TRUE
P2"( -          <PRIMARY>   EOF      ) = P2"( 10,  6,  7) = FALSE
P2"( -          <PRIMARY>   +        ) = P2"( 10,  6,  9) = FALSE
P2"( -          <PRIMARY>   -        ) = P2"( 10,  6, 10) = FALSE
P2"( -          <PRIMARY>   ELSE     ) = P2"( 10,  6, 14) = FALSE
P2"( -          <PRIMARY>   THEN     ) = P2"( 10,  6, 15) = FALSE
P2"( -          <PRIMARY>   )        ) = P2"( 10,  6, 16) = FALSE
P2"( -          IDENT       EOF      ) = P2"( 10, 11,  7) = TRUE
P2"( -          IDENT       +        ) = P2"( 10, 11,  9) = TRUE
P2"( -          IDENT       -        ) = P2"( 10, 11, 10) = TRUE
P2"( -          IDENT       ELSE     ) = P2"( 10, 11, 14) = TRUE
P2"( -          IDENT       THEN     ) = P2"( 10, 11, 15) = TRUE
P2"( -          IDENT       )        ) = P2"( 10, 11, 16) = TRUE
P2"( -          INTEGER     EOF      ) = P2"( 10, 12,  7) = TRUE
P2"( -          INTEGER     +        ) = P2"( 10, 12,  9) = TRUE
P2"( -          INTEGER     -        ) = P2"( 10, 12, 10) = TRUE
P2"( -          INTEGER     ELSE     ) = P2"( 10, 12, 14) = TRUE
P2"( -          INTEGER     THEN     ) = P2"( 10, 12, 15) = TRUE
P2"( -          INTEGER     )        ) = P2"( 10, 12, 16) = TRUE
P2"( (          <EXPR>      )        ) = P2"( 13,  2, 16) = FALSE
P2"( (          <SUM>       )        ) = P2"( 13,  5, 16) = TRUE
P2"( (          <PRIMARY>   )        ) = P2"( 13,  6, 16) = TRUE
P2"( (          IDENT       )        ) = P2"( 13, 11, 16) = TRUE
P2"( (          INTEGER     )        ) = P2"( 13, 12, 16) = TRUE
   92 FUNCTION VALUES, DENSITY =  2.25%, ENTRIES/VALUE  1.39
TIME =    0.06, TOTAL ELAPSED =     0.42 MIN.
```

SECTION 3

THE KERNEL LANGUAGE

Principles of Design

The kernel language must above all provide the programmer with a convenient means for controlling an automatic digital computer. Our first task is to discuss several general principles of language design and the contribution each makes toward the final form of the kernel language.

We require that the language be minimal in that the forms of the language must be concise and that there be as few kinds of forms as necessary. The conciseness and mnemonic significance of expressions in program text will depend upon the available character set as well as the aesthetic suitability of the multicharacter symbols chosen to represent the various linguistic entities. We have exercised considerable care in choosing the forms for the kernel language, drawing from the notations of Algol, Euler [25], Iverson's language [15] and PL/I[14]. We nevertheless realize that our readers with different experience in language or with different hardware may take strong exception to our choices. Our interest is primarily in the organization behind the linguistic façade and we take refuge in the realization that the language user can choose his own forms with the aid of the mechanisms of the extendable compiler.

We minimize the number of different structural forms by requiring that the kernel language be <u>involuted</u>. Involution is achieved by avoiding constructs that are applicable only in local context; we give some examples of failures in existing computer languages.

In Algol 60 we find the following isolated features:

(1) A primitive list structure in the constructs <for list> and <actual parameter list> which is unavailable elsewhere (for instance, to be used in array initialization).

(2) General call by name is available only through actual parameters.

(3) Dynamic memory allocation is available only at block entry.

We also find that most compilers provide a separate language for input and output which includes only a fraction of the power of the complete language. In each case the power of the language can be increased, the number of primitive concepts reduced and the compiler simplified by bringing the action out into the main program on a level with other statements.

By choosing operators and data types to reflect closely the mental processes of the language user we can substantially add to his ability to write brief and lucid programs. With distressing frequency we find that existing computers are ill-suited to the tasks thus set. We will find that our goal of designing a mutable computer language frequently implies a more anthropoid machine.

A program can be viewed as a sequence of operations on a data structure. It is necessary to provide the programmer with forms designed to control the sequence conveniently. We find that with a sufficiently elaborate set of <u>sequence controlling forms</u>, we have no need for the more traditional labels and go-to statements. Lest we be misunderstood, the inclusion of labels would not appreciably complicate the translator. We would regard the appearance of the label definition as an instruction to initialize the corresponding local variable to an appropriate value of type label upon entry into the scope of the variable. The mechanics of implementing the go-to statement are given in Wirth and Weber ([25] p 52). We feel that labels are an anochronistic holdover from early computer languages and are not in the collection of basic concepts.

Whenever possible we defer actions to a later time. A deferred action implies an increased freedom since we have preserved our ability to choose what action, if any, to take.

In particular, we shall require that each value be marked with its type during execution. In this way we can make the machine operators dynamically data dependent.†

The extendable compiler is a translator from the kernel language into a machine language. That language will generally be a mixture of direct commands to the hardware and interpretable information to direct the hardware and other programs present at execution time. We will call the program structure present at execution time the interpretive system (or simply, the system) to distinguish it from the hardware.

† Consider, for instance, the effect of the arithmetic operators in Iverson's language ([15], p 13). Dynamic typing demands a memory organization substantially different than any known to the author. It can be avoided by adding typing information to the declaration structure of the kernel language.

The program, the system and the hardware form three levels of control over the action of the machines. It is possible to have even more levels of control than those described here. For instance, the microprogramming feature of the IBM System 360 line of machines [19] could be inserted between the interpreter and the hardware.† We may change the system at any level. As we progress down the levels, the changes become more difficult (more expensive) and the results are more general.

Example Programs in the Kernel Language

We have implemented an extendable compiler for the kernel language on the Burroughs B5500. The actual compiler and its description are given elsewhere [20] but we wish to present the results of the execution of selected kernel language programs as motivation for the sequel. (Note another extensive example on page 35.)

We give several trivial examples which are essentially self-explanatory and finish with a version of the extendable compiler written in the kernel language. The programs and output are given in typescript instead of actual computer listing because the B5500 character set is not sufficiently rich to produce readable listing. We present the B5500 listing of the first example for the purpose of reader comparison.

† This has in fact been done by H. Weber for Euler IV on IBM 360 model 30.

Example 1. The procedure assigned to the identifier "factorial" gives the usual recursive definition of the factorial function. The local variable "n" is initialized from the parameter list when the procedure is activated. Note the subscript "[1]". If it were omitted the procedure would return a value of type list with one member equal to the required factorial. The subscript, here analogous to the assignment to a procedure identifier in Algol 60, serves to select the desired value.

Note also that the identifier "k" does not appear in the declaration of its list. It is local to the scope of the iterative statement and is declared by its appearance as the control variable.

```
{new factorial,
   factorial ← Ⓟ
   {new n,  if n = 0 then 1 else n × factorial(n-1)}[1],
   for all k from 1 to 6 do
      out ← (k base 10) ⊕ " factorial =" ⊕
      (factorial(k) base 10) ⊕ cr
} eof
*** output ***
1 factorial = 1
2 factorial = 2
3 factorial = 6
4 factorial = 24
5 factorial = 120
6 factorial = 720
```

B5500 Version of Example 1.

```
BEGIN   %  TEST PROGRAM FOR RECURSIVE FACTORIAL  %
   NEW FACTORIAL  N,
   FACTORIAL ← #
   BEGIN  NEW N,
      IF N = 0 THEN 1 ELSE N×FACTORIAL BEGIN N-1 END
   END[1],
   FOR ALL N FROM 1 TO 6 DO
      OUT ← (N BASE 10) CAT " FACTORIAL = " CAT
      (FACTORIAL BEGIN N END  BASE 10) CAT CR
END  EOF
```

*** output ***

```
 1 FACTORIAL = 1
 2 FACTORIAL = 2
 3 FACTORIAL = 6
 4 FACTORIAL = 24
 5 FACTORIAL = 120
 6 FACTORIAL = 720
```

1110 INSTRUCTIONS EXECUTED

--BR--	CAT	UNION	INTER	DIFF	BASE	OR	AND
57	18	0	0	0	12	0	0
<	≤	=	≠	≥	>	MEM	INCL
0	0	27	0	0	0	0	0
CONTAI	EQV	MAX	MIN	+	-	×	
0	0	0	0	0	21	21	0
MOD	DIV	*	NOTMEM	INDEX	LIST		
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
	/	PD	STO	SET	BRB	FETCH	XCG
0	0	1	7	1	6	102	1
NAME	JOF	BRF	VAL	BEGIN	END	XEQ	CASXIT
121	27	6	249	56	56	0	0
SUBS	CALL	AP	RTN	EOS	FOR	FORXIT	EOP
27	27	175	27	56	7	1	1
							NOT
0	0	0	0	0	0	0	0
MINUS	ABS	TYPE	ROUND	LENGTH	CHOP		
0	0	0	0	0	0	0	0

Example 2. Inner product.

```
{ out ← (( + / {1,2,3} x {3,2,1}) base 10) ⊕ cr} eof
```

```
***  output  ***
```

10

Example 3. A simple sort procedure.

```
{ new sort,
   sort ← (P)
   { new x,
      for all i from 1 to length x do
         for all j from i+1 to length x do
            (if x[i] > x[j] then x[{i,j}] ← x[{j,i}]),
      x
   }[2],
   for all i from sort({6,5,4,3,2,1}) do
      out ← (i base 10) ⊕ cr
} eof
```

```
***  output  ***
```

```
1
2
3
4
5
6
```

Example 4. A procedure to generate all the permutations of an ordered set.

```
{ new perm,
   perm ← (P)
   { new x,
      if 0 = length x then {x} else
      ⊕/(for all i from 1 to length x do
         for all t from perm(x[1 to i-1] ⊕ x[i+1 to length x])
            do x[i to i] ⊕ t)
   }[1],
   for all test from {"", "a", "ab", "abc", "abcd"} do
      for all p from perm(test) do out ← p  cat  cr
} eof
```

```
***  output  ***

a
ab
ba
abc
acb
bac
bca
cab
cba
abcd
abdc
acbd
acdb
...
etc.
```

Example 5. The following program is a compiler-executor for a small language. The organization of the program is essentially that of the extendable compiler written for the Burroughs B5500. We will make comments on the kernel language constructs used, the organization of the compiler-executor and the implementation of the small language.

We find the following major sections: (1) The syntactic analysis tables, (2) The scanner, (3) The compile actions definition and the compiler, (4) The execute actions definition and the executor, (5) The test program and its output.

In the outermost list we find the declaration of all the global identifiers. To seven of them we find immediate assignments of syntactic analysis tables. The tables are best understood in reference to the Backus-Naur Form description of the small language contained in the comments in the compile action definitions. The table "reservedsymbols" is a list of strings which correspond to the nonterminal and terminal symbols in the grammar. The position of a symbol in the list is called its symbol number.

The table "productionrightparts" is a list of lists, each of the latter corresponding, in order, to the right part of a production (<program> is symbol 1, eof is symbol 15, {1,15} corresponds to <program>eof). "productionleftparts" contains the symbol number of the left part of the corresponding production.

The next four tables are linearized representations for the parsing functions P1' and P2' which locate the right and left ends of the next CRS. All seven tables could have been produced by a syntax preprocessor similiar to the symbol pair algorithm of Section 2.

The scanner must fetch the next terminal symbol from the input text each time it is called. In the case of the small language this means identifying digits (which are less than 10 in our character set), catenating identifiers (letters are less than 36), matching reserved identifiers with their syntactic symbol numbers, entering variables into the symbol table and matching special characters with their syntactic symbol numbers.

Now skip ahead to the procedure assigned to "compile". After some initializing we find "while compiling do" which controls a loop down to the end of the procedure. Within that loop we immediately find the syntactic analysis algorithm. In the first inner loop we are scanning ahead to the right under control of the linearized form of function P1'. Having located the right end of the CRS, we exit the loop and enter a loop scanning for the left of the CRS under control of the linearized version of function P2'. At the termination of the second loop, we may compare the CRS with the production right part table to find the production number "pn".

"pn" is used as a subscript to select the compile actions corresponding to that production from the preceding table. The prescript operator "[compileactions[pn]]" causes the execution, in order, of actions from the explicit list following the prescript. For instance, the discovery of production two would cause the integer 12 to be placed in the code array, the program pointer to be incremented and the variable "compiling" to be set to false, thus terminating compilation.

At the termination of each compilation step we find the substitution of the production left part for the CRS.

The compiler is considerably simplified by having the entire code array and execution memory available at compile time.

The translated code for the small language consists of a sequence of twelve operation codes. Within the procedure assigned to "excute" we find another prescript "[executeactions[code[pp]]]". The operation code in "code[pp]" is used to select a sequence of execution actions from the preceding table. Execution proceeds until the operation code 12 causes the execution action "executing ← false" whereupon execution terminates.

```
'Micro-mutant, a small version of the extendable compiler'
{new code pp memory variables mp text tp f1 g1 f2 g2
  reservedsymbols productionleftparts productionrightparts
  compile compileactions execute executeactions
  scan nextsymbol scanval,
  'Seven tables prepared by a syntax preprocessor'
  reservedsymbols ← 'syntactic vocabulary'
  {"<program>", "<stmt>", "<stmtl>", "<if clause>",
    "<label>", "<list head>", "<expr>", "<exprl>",
    "<arith exp>", "<term>", "<terml>", "<factor>",
    "<integer>", "<var>", "eof", "go", "output", "if",
    "<ident>", "begin", "(", "<digit>", "end", "to",
    ":", ",", "+", "-", "x", "/", "then", ")" },
  productionrightparts ←
  {{1,15}, {15,2,15}, {3}, {5,3}, {4,3}, {6,23},
    {16,24,7}, {17,7}, {7}, {18,7,32}, {5,25}, {20,2},
    {6,26,2}, {8}, {14,27,8}, {9}, {9,28,10}, {9,29,10},
    {10}, {11}, {11,30,12}, {11,31,12}, {12}, {14},
    {13}, {21,17,33}, {22}, {13,22}, {19}},
  productionleftparts ← {1,1,2,3,3,3,3,3,3,4,5,6,6,7,8,
    8,9,9,9,10,11,11,11,12,12,12,13,13,14},
  f1 ← {1,2,3,1,1,1,3,4,4,5,5,6,6,6,3,1,1,1,7,1,1,7,3,
    1,7,1,1,1,1,1,1,7,6},
  g1 ← {1,1,1,1,1,1,1,1,1,1,1,1,1,1,2,3,3,3,3,3,3,6,1,
    1,7,1,6,4,4,5,5,3,3},
  f2 ← {1,1,1,6,6,1,1,1,1,1,1,1,1,1,7,1,5,5,1,7,5,1,1,
    5,1,7,4,3,3,2,2,1,1},
  g2 ← {1,7,6,1,1,1,5,4,1,3,1,2,1,1,1,1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,1,1},
```

```
scan ← (P) 'fetch the next terminal symbol from the text'
{new t,  while text[tp] = " "[1] do tp ← tp + 1,
   if text[tp] < 10 then
   {nextsymbol ← "<digit>" index reservedsymbols,
      scanval ← text[tp],  tp ← tp + 1
   } else
   if text[tp] < 36 then
   {'catenate an identifier'
      t ← tp,  while text[tp] < 36 do tp ← tp + 1,
      t ← text[t to tp-1],
      nextsymbol ← t  index reservedsymbols,
      if nextsymbol = Ω  then
      {'a variable'
         nextsymbol ← "<ident>" index reservedsymbols,
         scanval ← t  index variables,
         if scanval = Ω  then variables[scanval ← mp ← mp + 1] ← t
      }
   } else 'must be a special character'
   {nextsymbol ← text[tp to tp] index reservedsymbols,
      tp ← tp + 1
   }
}, 'end of scanning algorithm'
```

```
compileactions ←
{  {},              '<program>      ::=  <program> eof          '
   {12,17,19},      '<program>      ::=   eof <stmt> eof        '
   {2,17},          '<stmt>         ::=  <stmtl>                '
   {},              '<stmtl>        ::=  <label> <stmtl>        '
   15,              '<stmtl>        ::=  <if clause> <stmtl>    '
   {},              '<stmtl>        ::=  <list head> end        '
   {1,17},          '<stmtl>        ::=   go to <expr>          '
   {7,17},          '<stmtl>        ::=   output <expr>         '
   {},              '<stmtl>        ::=  <expr>                 '
   {3,17,18,17},    '<if clause>    ::=   if <expr> then        '
   13,              '<label>        ::=  <ident> :              '
   {},              '<list head>    ::=   begin <stmt>          '
   {},              '<list head>    ::=  <list head> , <stmt>   '
   {},              '<expr>         ::=  <exprl>                '
   {6,17},          '<exprl>        ::=  <var> ← <exprl>        '
   {},              '<exprl>        ::=  <arith exp>            '
   {8,17},          '<arith exp>    ::=  <arith exp> + <term>   '
   {9,17},          '<arith exp>    ::=  <arith exp> - <term>   '
   {},              '<arith exp>    ::=  <term>                 '
   {},              '<term>         ::=  <terml>                '
   {10,17},         '<terml>        ::=  <terml> x <factor>     '
   {11,17},         '<terml>        ::=  <terml> / <factor>     '
   {},              '<terml>        ::=  <factor>               '
   {5,17},          '<factor>       ::=  <var>                  '
   {4,17,14,17},    '<factor>       ::=  <integer>              '
   {},              '<factor>       ::=  ( <expr> )             '
   {},              '<integer>      ::=  <digit>                '
   16,              '<integer>      ::=  <integer> <digit>      '
   {4,17,14,17}     '<var>          ::=  <ident>                '},
```

```
compile ← (P)
{new  x  xv  xp  lp  pn  compiling,
   pp ← tp ← 1,  mp ← 0,  compiling ← true,
   x ← xv ← 25 list 0,    xp ← 2,
   x[1] ← "(" index reservedsymbols,
   x[2] ← "eof" index reservedsymbols,
   memory ← variables ← 10 list Ω ,   code ← 100 list 0,
   scan, 'initialize nextsymbol and scanval'
   while compiling do
   {while f1[x[xp]] ≤ g1[nextsymbol] do
      {x[xp←xp+1] ← nextsymbol, xv[xp] ← scanval,
         scan 'the decision for function P1'},
      lp ← xp,
      while f2[x[lp-1]] ≤ g2[x[lp]] do lp ← lp-1,
      'the right part of the next CRS is between lp and xp'
      pn ← x[lp to xp] index productionrightparts,
      'the production number is used as an index to select
       a sequence of compile actions'
      [compileactions[pn]] 'a prescript on the following list'
      {'the first twelve compile actions correspond to
         execution macro-instruction operation codes'
         code[pp] ←  1, code[pp] ←  2, code[pp] ←  3,
         code[pp] ←  4, code[pp] ←  5, code[pp] ←  6,
         code[pp] ←  7, code[pp] ←  8, code[pp] ←  9,
         code[pp] ← 10, code[pp] ← 11, code[pp] ← 12,
         'the remaining 7 rules do fixups, label initialization,
          increment the program pointer, etc.'
```

```
        memory[xv[xp-1]] ← pp,                   '13'
        code[pp] ← xv[xp],                       '14'
        code[xv[xp-1]] ← pp,                     '15'
        xv[lp] ← (xv[xp-1] x 10) + xv[xp],       '16'
        pp ← pp + 1,                             '17'
        xv[lp] ← pp,                             '18'
        compiling ← false                        '19'
    },
    xp ← lp, 'making the left-for-right part subst.'
    x[xp] ← productionleftparts[pn]
  }
}, 'end of compilation procedure'
```

```
executeactions ←
({2,15},             'unconditional branch      1'
 {15,1},             'clear stack               2'
 {1,10,15},          'branch on zero            3'
 {4,1,6,1},          'load stack from code      4'
 {8,1},              'load stack from memory    5'
 {9,7,5,1},          'store stack to memory     6'
 {3,1},              'decimal output            7'
 {11,5,1},           'add                       8'
 {12,5,1},           'subtract                  9'
 {13,5,1},           'multiply                 10'
 {14,5,1},           'divide                   11'
 16                  'halt                     12'),
```

```
execute ← Ⓟ
{new executing stack sp,
   sp ← 0,  pp ← 1,  executing ← true,
   stack ← 100 list 0,
   while executing do
   [executeactions[code[pp]]]
   {  pp ← pp + 1,                                        ' 1'
      pp ← stack[sp],                                     ' 2'
      out ← (stack[sp] base 10) ● cr,                     ' 3'
      sp ← sp + 1,                                        ' 4'
      sp ← sp - 1,                                        ' 5'
      stack[sp] ← code[pp],                               ' 6'
      stack[sp-1] ← stack[sp],                            ' 7'
      stack[sp] ← memory[stack[sp]],                      ' 8'
      memory[stack[sp-1]] ← stack[sp],                    ' 9'
      pp ← if stack[sp] = 0 then code[pp] else pp + 1,    '10'
      stack[sp-1] ← stack[sp-1] + stack[sp],              '11'
      stack[sp-1] ← stack[sp-1] - stack[sp],              '12'
      stack[sp-1] ← stack[sp-1] x stack[sp],              '13'
      stack[sp-1] ← stack[sp-1] ÷ stack[sp],              '14'
      sp ← 0,                                             '15'
      executing ← false                                   '16'
   }
},  'end of execution procedure'
```

```
'test program for micro-mutant compiler'
text ←
"begin  n ← 1,
   k:  if  1024-n  then
   begin output n,
      n ← 2 X n,
      go to  k
   end
 end  eof eof ",
compile,
out ← "code dump:" ⊕ cr ⊕ "pp" ⊕ tab ⊕ "inst" ⊕ cr,
for all  i  from  1  to  pp-1  do
   out ← (i base 10)⊕ tab ⊕ (code[i] base 10) ⊕ cr,
   out ← cr,
   execute,
   out ← cr ⊕ "memory dump:" ⊕ cr,
   for all  i  from  1  to  mp  do out ← variables[i] ⊕ "=" ⊕
   (memory[i] base 10) ⊕ cr
eof
```

code dump:

pp	inst
1	4
2	1
3	4
4	1
5	6
6	2
7	4
8	1024
9	4
10	1
11	5
12	9
13	3
14	35
15	4
16	1
17	5
18	7
19	2
20	4
21	1
22	4
23	2
24	4
25	1
26	5
27	10
28	6
29	2
30	4
31	2
32	5
33	1
34	2
35	2
36	2
37	12

1
2
4
8
16
32
64
128
256
512

memory dump:
n = 1024
k = 7

Syntactic and Semantic Definition

The following table is the phrase structure grammar for the kernel language. We adopt the Backus-Naur Form of the Algol report, substitute the reduction symbol "::=" for the production arrow "→" of Section 2, enclose the members of $\underline{V}_N$ in the brackets "<" and ">" and underline the multicharacter representations for members of $\underline{V}_T$. The special symbols integer, identifier and string are discussed on page 93.

We remind our readers that the grammar obeys two restrictions that occasionally give it an artificial appearance. First, it is a symbol pair grammar. Second, the productions have been carefully selected to reflect the desired sequence of execution in the canonical parse to simplify the production of the machine code.

Symbol Pair Grammar for the Kernel Language

<program> ::= ⊢ <expression> ⊣

<expression> ::= <expression$_1$>

<expression$_1$> ::= <if clause> <expression$_1$> |
<expression$_2$>

<expression$_2$> ::= <expression$_3$>

<expression$_3$> ::= <if clause> <truepart> <expression$_3$> |
<primary$_1$> ← <expression$_3$> |
<procedure> <expression$_3$> |
<for clause> do <expression$_3$> |
<for clause> <while clause> do <expression$_3$> |
<while clause> do <expression$_3$> |
<step list>

<procedure> ::= P

<if clause> ::= if <expression> then

<truepart> ::= <expression$_2$> else

<for clause> ::= for all identifier from <step list>

<while clause> ::= <while> <step list>

<while> ::= while

<step list> ::= <simple expr> to <simple expr> |

<simple expr> by <simple expr> |

<simple expr> to <simple expr> by <simple expr> |

<simple expr> by <simple expr> to <simple expr> |

<simple expr>

<simple expr> ::= <simple expr$_1$>

<simple expr$_1$> ::= <primary> <infix> <simple expr$_1$> |

<prefix> <simple expr$_1$> |

<infix> / <simple expr$_1$> |

<primary>

<primary> ::= <primary$_1$>

<primary$_1$> ::= <primary$_1$> [<expression>] | <primary$_2$>

<primary$_2$> ::= <constant> | (<expression>) |

identifier <list> | identifier | <list> | <case>

<list> ::= <list head> }

<list head> ::= <list head> , <expression> |

<begin> <declaration> | <begin> <expression>

<begin> ::= {

<case> ::= <case head> }

<case head> ::= <case head> , <expression> |

<case begin> <declaration> |

<case begin> <expression>

<case begin> ::= [<expression>] {

<declaration> ::= <declaration$_1$>

<declaration$_1$> ::= new identifier |

<declaration$_1$> identifier

<constant> ::= true | false | integer | integer . | . integer |

integer . integer | universe | string | <begin> }

<infix> ::= ⊕ | ∪ | ∩ | ⊖ | base | ∨ | ∧ | < | ≤ | = | ≠ |

≥ | > | ∈ | ∉ | index | list | ⊂ | ⊃ | ≡ | max | min |

+ | - | × | mod | ÷ | ↑

<prefix> ::= ¬ | minus | abs | type | round | chop | length | set

We will now give our interpretations of each construct. The description of an involuted language involves the use of terms before they are defined. Paragraph numbers and cross references are used to ease the reader's task in following the description.

We must distinguish between text describing the form of a construct, text giving examples of a construct, text describing the meaning of a construct, text justifying the choice of a construct and text advocating a particular system organization or machine design. We distinguish them when possible with paragraph headings of Syntax, Examples, Semantic Description, Justification, and Implementation, respectively.

Implementation of Reserved Words, Identifiers, Strings and Integers. By an underlined word in the grammar we mean to reserve that word for exclusive use in the given grammatical context. We do not then need special character sets or escape symbols to write programs. One implication is that spaces are significant and that we cannot know whether an identifier is reserved or not until we have seen all of it. Thus we find that the process of catenating identifiers must take place outside of (and before) the syntactical analysis algorithm. We assign this task to a procedure called the scanner. It turns out to be convenient to recognize and convert both integers and strings there also. As a result we find the symbols integer, identifier and string terminal in the grammar but not underlined. The inclusion of natural language text within a program in the form of parenthetical comments to the reader is provided by choosing an otherwise unused character as a comment bracket. We reject the Algol 60 comment convention because it is neither concise nor independent of the program structure (since it involves the use of the semicolon).

Semantic Description of Values. Before we can discuss constants, we must introduce the values they represent. We specify in the language four unstructured types of values (*undefined*, *number*, *name* and *process*) and three structured types (*string*, *list*, and *set*). (3-1)

Syntax of Constants.

<constant> ::= Ω | ∞ | universe |

integer | integer . | . integer | integer . integer |

true | *false* |

<begin> } | string

<begin> ::= {

Examples of Constants.

Ω ∞ universe true false

2 2. .2 136.721

{} "ABCDEFGHIJ-32"

Semantic Description of Constants of Typed Undefined.

Ω , ∞, and *universe* have type undefined. The value of a variable before anything has been stored into it is Ω ; the result of dividing a positive number by zero is ∞; the intersection over the null collection of sets is *universe* (the universal set).

The operators = and $\neq$ are valid for all of the above; ∞ is a valid operand for all numeric operators; *universe* is a valid operand for all operators that accept sets as operands.

Implementation of Values of Type Undefined. The appearance of an undefined value is usually cause for alarm. An alarm should cause the system to originate a warning action to the programmer, but beyond that we make no particular recommendation as to the form of the warning or the means of suppressing it.

Justification of Values of Type Undefined. Undefined values can arise in a variety of ways. We might think of, for instance, the value of an uninitialized variable, the result of division by zero and the result of an invalid subscripting operation. We propose the introduction of a type undefined and a collection of values of type undefined corresponding to (usually) pathalogical situations such as those described above. For some we may wish explicit constants in the language. Thus we might write

$$\text{if} \quad X = \infty \quad \text{then} \ldots$$

to test for a division by zero.

The introduction of a type undefined provides a conceptually simple mechanism with which to warn the programmer of some of the wilder errors as well as providing a relatively noncontroversial system reaction to the errors. If the error is isolated, the system may proceed with execution of the program, leaving behind an indicative trail of undefined values.

Semantic Description of Numbers. A value of type number will be the computer representation of a real number. We have two reasons for not wishing to make our concept of a number precise.

First, the only reasonable choice for numbers in a given implementation of the kernel language will be those acceptable to the floating point hardware of the machine. For that implementation, the programmer's knowledge about values of type number will be a pragmatic mixture of his knowledge of numbers in the abstract and his study of the machine specifications.

Second, a study of the desirable properties of computer numbers is well beyond the scope of this paper. We hope to see some results in this direction in a study presently being conducted by W. Kahan, J. Welsch, and N. Wirth.

We do find it useful to distinguish three subsets of the class of computer numbers, the first of which is computer integers. The second is the set of characters which is the set of integers restricted to the range 0 to 255 inclusive. (3-2) Finally we have the logical values 0 and 1. (3-3)

Semantic Description of Strings. We consider a fixed input or output device. We assume a correspondence between the printing character of the device and the characters (see 3-2). Normally some of the characters are unused for printable characters and may be used for nonprinting or control functions. A string in the kernel language is an ordered set of printable characters delimited by the string quote ("). We adopt the PL/I convention that within the string, two contiguous string quotes signify a single string quote within the string.

Justification of Strings. The programmer communicates with his program via strings of characters; thus unrestricted ability to analyze, manipulate and produce character strings is a minimal requirement for any computer language. In much the same spirit that a compiler for numerical work provides certain standard functions such as square root and natural logarithm, we must provide primitive string manipulating functions.

Implementation of Input and Output. For the kernel language we assume that we have a single input medium and a single output medium.

If we view the program over the history of its execution, the input and output are each single contiguous strings of characters. We name two special variables (IN and OUT) and access them in the normal manner with our primitive string manipulating functions. The fact that in real time, the program may have to wait before an access to IN can be completed does not affect the program logic. On the other hand, the program must have control over when, in real time, the output appears. Thus we establish the convention that whenever a carriage return is catenated onto OUT, the string OUT is shortened past the carriage return and the excised characters appear on the output medium.

Semantic Description of the Null List. The construct {} represents the null set of values. We use the dummy production <begin> for technical reasons having to do with the emission of block entry code from the canonical parse.

Semantic Description of true and false. The constants true and false are synonymous with the characters 1 and 0.

Semantic Description of Variables. A variable is an object which can be named in the kernel language and to which any value can be assigned. The designation variable is given by either an identifier or a subscripted identifier (see 3-9).

Semantic Description of Values of Type Name. Corresponding to every valid name in the kernel language is a value of type name within the system. Names are created as intermediate results and are not accessible to the programmer.

Implementation of Variables and Names. A variable corresponds to a memory address. The type of the value stored in a variable must be preserved, thus we find that we allocate two words for a variable and use the second to store the type information. We would prefer a machine in which the type bits were automatically associated with each word but had special properties. In particular we would like to determine whether the variable contains a value of type address to effect indirect addressing but without accessing the whole variable to find out. We believe this implies that at least some of the type bits must be accessible in a fraction of the time to access a memory word.

Syntax of Declarations.

<declaration> ::= <declaration$_1$>

<declaration$_1$> ::= new identifier |

<declaration$_1$> identifier

Examples of Declarations.

new a b c d

new thisone thatone anyone

Semantic Description of Declarations. At most one declaration appears in the head of a list (see 3-5). The extent of the list defines the scope of the identifiers in the declaration. Every identifier in a program must either be reserved or lie within the scope of an identifier of the same name. Upon entry into the scope of an identifier, the system allocates a variable to it and gives it the value uninitialized. An identifier names the variable allocated to it. If an identifier appears in more than one declaration, the use of that identifier names the variable corresponding to the smallest containing scope.

Justification of Declarations. The use of declarations to define the scope of variables is well established. With dynamic typing of values, we find no particular advantage in binding the type of an identifier with a compile-time declaration. The involuted nature of the kernel language moves the structural implications of the conventional declaration out into the main program. Thus we find that declarations in the kernel language are reduced to the single action of delineating the scope of, and allocating variables for identifiers used in the program.

We regard this as the final step in the direction taken by Wirth and Weber ([25] p. 43).

Implementation of Declarations. From the viewpoint of variable addressing, the program consists of a nested collection of scopes. Thus from any point in the program we may assign a unique ordered pair of integers to each variable, namely, the depth of nesting of the scope of the identifier and the position of the identifier in the declaration. We call the integers the scope level and order number respectively. At compile time we can name the variables with the scope level and order number.

The form of the declaration suggests that we should allocate a list of variables corresponding to the declared identifiers upon entry into the scope of the identifiers. The order number of a variable is the index of that variable in the list of local variables. Thus we expect to use the scope level to find a particular list and the order number to find an element of that list. At execution time we convert the compile time name into a value of type name by locating the memory location assigned to the variable.

The designers of programming languages have traditionally (3-4)
indulged themselves in a semantic ambiguity: one cannot always tell from the form of an expression (a subscripted variable for instance) whether the name, or the value stored in the named location, is indicated. In the Algol 60 construct of general call by name the ambiguity is complete; the expression must yield both name and value, the choice depending upon its use at a remote location. One can remove the ambiguity by introducing explicit name and value operators into the language ([25] p. 45). Since the choice is always ultimately clear from the context in which the expression is found, we have chosen to dynamically defer the final fetch of the value in cases where there is doubt.

Syntax of Lists.

```
<list>          ::=  <list head> }
<list head>     ::=  <list head> , <expression> |
                     <begin> <declaration>  |
                     <begin> <expression>
<begin>         ::=  {
```

Examples of Lists.

```
{1, 2, 3, "ABC"}
{x ← 1,  y ← y-2,  if  x < y  then  z  else  z ← y }
{new  a b c,  a ← b ← 1.0,  c ← 5 }
{new a, {new a, a ← 2}, a ← 2 }
```

Semantic Description of Lists. A list is an ordered set of (3-5) expressions which are evaluated sequentially. The value of the evaluated list is the ordered set of evaluated expressions and is of type list. The declaration is not an expression and does not contribute to the value of the list.

Justification of Lists. Arrays, trees, iteration lists, parameter lists, strings, blocks and compound statements are ordered sets. The inclusion of arbitrary (even infinite) lists in the kernel language together with the principle of involution yields a drastic reduction in the number of primitive concepts.

Semantic Description of Values of Type List. A value of type list is an ordered collection of values with any admixture of the value types.

Implementation of Lists. We discover in the literature two alternatives for representing lists. The first, in LISP, demands a list structure where all elements are explicitly linked in storage. In Euler and Burroughs B5500 hardware we find that a value of type list is a descriptor which delineates the extent and locates the list. The list elements are stored in sequentially contiguous memory locations. The first comparison is in the amount of storage required to represent a given list. In LISP we must use memory for the linking information; in Euler we must use memory for dynamic typing. We estimate that implicit linking saves a factor of two in memory. The second comparison is in ease of access. In LISP we must explicitly trace the list structure to find an element near the end of a list; in Euler we may access any element of any list directly via a subscript. There is no reason to expect the implicit list structure organization to be less efficient than conventional index registors for array applications so long as descriptors do not have to be repeatedly fetched from memory. Even with the repeated fetching, the B5500 is able to subsume the extra core accesses under cover of the multiply operation time so as to be proportionately as fast as the 7090 for matrix problems. Our third comparison is in ease of modification. In LISP we must change a link to append or insert an element to a list where in Euler we must copy the entire contiguous block. Implicit linking is severely less efficient here. Fourth, we must consider storage reclamation. In both systems the majority of time is spent in searching out and identifying the valid list structure. In Euler we find that the percentage of execution time spent in storage reclamation is roughly the same as the percentage of storage occupied with valid list structure; we have no figures on LISP. In any case we do not expect the systems to be much different in this respect.

We do not know which represents the most efficient solution; we suspect that it is both problem dependent and hardware dependent. We have chosen implicit linkings so as to have array capability without introducing them into the language as a distinct form.

Semantic Description of Values of Type Set. A set differs from a list in two ways:

(1) A set cannot contain two equal values.

(2) The programmer cannot prescribe the order of the members of the set. Certain operations are allowed on sets and not on lists.

Justification of Values of Type Set. The set operations of membership, inclusion, equivalence, etc., require preorganization for efficient implementation. We choose to sort the values of a set by a machine determined order to facilitate table look ups (binary searches), union and intersection (merges), etc. The membership operation (for instance) takes

$\log_2 n$ operations in a sorted set and $n/2$ operations in an unsorted set (on the average).

<u>Syntax of Subscripts</u>.

[<expression>]

<u>Examples of Subscripts</u>.

[i] [x-z] [{1,2,3}]

<u>Semantic Description of Subscripts</u>. We will distinguish between the subscript expression (the expression in the syntax above), the subscript operator (the result of applying certain standard transformations to the subscript expression), the subscript operand (the object in the kernel language to which the subscript operator is being applied) and the subscripted expression (the final result achieved by applying the subscript operator to the subscript operand). A subscript expression has meaning if (1) it has type number or (2) it has type list and all its members have type number. A subscripted expression has meaning if (1) the subscript has meaning and (2) the subscript operand is one of the structured types, string, list or set. If a subscripted expression does not have meaning, it yields a value of type undefined.

<u>Subscripts of Type Number</u>. If the value of the subscript expression is of type number, the value of the subscript operator is the nearest (rounded up) integer.

<u>Subscripts of Type List</u>. If the value of the subscript expression is of type list and each element of the list is of type number then the subscript operator is the list of nearest integers (rounded up) corresponding to the numbers in the subscript expression.

Justification of Subscripts. Various constructs in the kernel language have the form of ordered sets. Numerical subscripts will be used to select elements from the ordered sets and list valued subscripts will be used to select subsets from the ordered sets.

Examples of Subscripted Lists.

list	subscript	result
{10, 20, 30, 40}	[1]	= 10
{10, 20, 30, 40}	[minus 1]	= 40
{10, 20, 30, 40}	[{2,4}]	= {20, 40}
{10, 20, 30, 40}	[1 to 3]	= {10, 20, 30}
{10, {20, {30}, 40}}	[2]	= {20, {30}, 40}
{10, {20, {30}, 40}}	[2][2]	= {30}

Syntax of the Case Expression.

```
<case>         ::=  <case head> }
<case head>    ::=  <case head> , <expression> |
                    <case begin> <declaration> |
                    <case begin> <expression>
<case begin>   ::=  [<expression>] {
```

Examples of the Case Expression.

```
[n] {1,2,3,5,7,11,13,17,19}
[{x, minus 1}] { new a,
      a ← "Invalid type for subscript operator",
      a ← "Invalid type for subscript operand",
      a ← "Subscript out of range",
      out ← a ⊕ cr
}
```

Semantic Description of the Case Expression. [13] The case expression has the form of an explicit list preceded by a subscript. Upon execution the following occurs: (1) The subscript operator is evaluated. (2) The list is entered. (3) Storage is allocated for the local variables (if any). If the subscript operator is an integer then we have (4) If the value of the integer is zero or larger in magnitude than the number of expressions in the list, a value of type undefined results. If the subscript operator is positive then it is used as an index to select an expression counting from the front of the list; if it is negative it is used to select an expression counting from the rear of the list. (5) The selected expression is evaluated and the value of the case expression is the value of the selected expression. If the subscript operator is of type list then (4) Each number in the list is used sequentially to select an expression as done above for subscript operators of type number. (5) The value of the case expression is the list of values so computed.

Implementation of Case Expressions. The use of an index to select an expression out of a list of expressions suggests that the machine code itself should have the form of a list structure where the code for an expression occupies exactly one memory location.

Justification of Case Expressions. The case expression represents one of the more powerful sequence controlling features of the kernel language. If the subscript operator is a number, it resembles the Algol 60 switch without the nuisance of labels. The list valued subscript operator allows reordering and repetition of the expressions in a list.

Syntax of Primaries.

<primary> ::= <primary$_1$>

<primary$_1$> ::= <primary$_1$> [<expression>] | <primary$_2$>

<primary$_2$> ::= <constant> | (<expression>) | identifier <list> |
identifier | <list> | <case>

Examples of Primaries.

3.0 (x-z) X X[2][a-2] Y{1,2,3} {1,2,3} [n]{1,2,3}

Semantic Description of Primaries. Parentheses allow the programmer to reorder the evaluation of operators in the conventional manner. They have no other meaning in the kernel language.

An identifier followed by a list signifies a procedure activation. The list (of parameters) is evaluated and the name of the variable corresponding to the identifier is computed. If the variable contains a value of type process the process is activated, otherwise the value undefined is returned. (See 3-12). (3-6)

If an identifier appears alone, the name of the variable corresponding to the identifier is first computed. If that variable holds a value of type process, the process is activated and the name of the identifier is replaced with the value of the process. (See 3-11).

Semantic Description of Subscripted Primaries. If the (3-7)
subscript operand has type name, it is replaced by the value of the named variable. The effect of the subscript operand on types string and list follows.

Semantic Description of Subscripted Strings. A value of type string is an ordered set of characters. If the subscript operand is of type string the following remarks apply: (1) If the subscript operator is an integer and this integer is positive and less than or equal to the length of the string, the value of the subscripted expression is the character selected by counting from the front of the string; if the integer is negative and no larger, in magnitude, than the length of the string, the character is selected by counting from the rear of the string; otherwise the value undefined is returned. (2) If the subscript operator is a list of integers then the result is a (sub) string selected by applying each integer as a subscript operator in order of occurence. (3-8)

Implementation of Strings. If we view a string as a packed read-only data structure then the operation of forming a contiguous substring can be accomplished by constructing a new descriptor to point into the old string. An implication is that a scanning algorithm does not have to move characters, only locate them.

Semantic Description of Subscripted Lists. A value of type list is an ordered set of values. If the subscript operand is of type list the following remarks apply: (1) If the subscript operator is an integer then the value returned is the name of the variable selected according to the algorithm given in paragraph (3-8). (2) If the subscript operator is a list of integers then the result is the (sub) list selected by applying each integer as a subscript operator in their order of occurence in the subscript operator. (3-9)

Syntax of Prefix Operators. (3.10)

<prefix> ::= ¬ | minus | type | abs | round | chop | length | set

Semantic Description of Prefix Operators. A prefix operator is a single valued partial function of one operand. The action of the operator is defined when the function is given over the allowed range of the operand. All of the above operators, except type, length and set, are numeric prefix operators. Their behavior for numeric operands is obvious; their behavior for list valued operands is discussed presently.

Semantic Description of the Operator "type". The range of operands for type is the collection of all values. The function defined by the operator gives an integer corresponding to the type of the operand. We leave the actual integer to be implementation defined since it is convenient to have more than one system type corresponding to a given kernel language type. Normally we test for type with a construct like

```
if (type a)  =  (type " ") then ...
```

rather than attempting to remember the correspondence between integers and types.

Semantic Description of the Operator "length". The operand of length must have type set, list or string. The value of the function defined by the operator is the number of elements in the structured operand.

An application of the operator set is the only way to transform a value of type list into a value of type set. The resulting value will have no repeated elements and will have been reordered.

Syntax of Infix Operators.

<infix> ::= $\cap$ | $\cup$ | $\ominus$ | $\subset$ | $\supset$ | $\in$ | $\notin$ | index | $\oplus$ | list | base |
$\vee$ | $\wedge$ | $<$ | $\leq$ | $=$ | $\neq$ | $\geq$ | $>$ | max | min | $+$ | $-$ |
$\times$ | mod | $\div$ | $\uparrow$

Semantic Description of Infix Operators. An infix operator is a single valued partial function of two (right and left) operands. The action of the operator is defined when the function is given over the allowed range of the operands.

Semantic Description of $\cap$, $\cup$, $\ominus$, $\subset$, and $\supset$. The range of values for both operands is the collection of all values of type set. Their defining functions are, respectively, set intersection, union, difference, inclusion and containment.

Semantic Description of $\in$ and $\notin$. The left operand ranges over the collection of all values; the right operand must be of type set or list. The value of the function defining $\in$ is true if a value equal to the left operand is found in the right operand. The function defining $\notin$ is the complement of the above.

Justification of Set Operators. The concept of a set is a natural data type for many algorithms. Its simplicity makes the set a natural object for the kernel language.

Implementation of Set Operators. The elements of the set valued operands of the above operators are sorted to facilitate the construction of efficient algorithms for their execution (sort - merge, binary look up, etc.)

Semantic Description of index. Index is identical to $\in$ except that the resulting value is the index within the set of the value, if found, and of type undefined otherwise.

Semantic Description of base. The operands of base must be both integers. The result is a value of type string. The string is the legible representation of the left operand to the base specified by the right operand.

Semantic Description of list. The left operand of list must be a number and the right may have any value. The left operand is rounded to the nearest integer and the result is that many copies of the right operand (thus a value of type list).

Semantic Description of $\oplus$. The range of operands of $\oplus$ is the collection of values for which the types of the operands (left and right respectively) are string, string; set, list; set, set; list, set; list, list. In the first case the result is a string obtained by catenating the right operand onto the tail of the left operand. Otherwise the result is a list containing the members of the left operand followed by the members of the right operand.

Semantic Description of $=$ and $\neq$. The operands of $=$ and $\neq$ may range over all values. If the operands do not have the same type, they are unequal. If they have an unstructured type, they are equal if they are identical. If they have a structured type, they are equal if they have the same length and the corresponding elements are equal.

Semantic Description of Numeric Infix Operators. All of the remaining operators are numeric infix operators. If both operands are of type number, the function defining the operators is usually obvious.

We make the following comments. The operators $\vee$ and $\wedge$ (logical "or" and logical "and") accept as operands only logical values (See 3-3). The result of $s \div t$ is the (real valued) quotient. If we wish the integer quotient, we write $\text{chop}(s \div t)$. s <u>mod</u> t is defined to be the function $s - t \times \text{chop}(s \div t)$ for all numbers.

Syntax of Simple Expressions.

<simple expr> ::= <simple expr$_1$>

<simple expr$_1$> ::= <primary> <infix> <simple expr$_1$> |
<prefix> <simple expr$_1$> |
<infix> / <simple expr$_1$> | <primary>

Examples of Simple Expressions.

3-2-1 $a + b - c \times d \text{ mod } e \div f \uparrow g$

$\neg$(minus abs round chop a) = (b max c min d)

$+$ / 1 to n {1,2,3} - {2,3,4}

Semantic Description of Simple Expressions. From the grammar above we deduce that the operand of a prefix operator is the value of the (largest possible) simple expression to its right. The operands of an infix operator are the primary to its left and the (largest possible) simple expression to its right. We further deduce that all operators (excepting those reordered by parentheses) are evaluated right-to-left.

Justification of Right to Left. We have provided a fairly extensive catalog of operators in the kernel language while leaving room for further extensions. With so many operators it would be confusing at best to assign hierarchies to them. In search for a simple rule ordering the evaluations, we are left with either left-to-right or right-to-left order ([15] p. 8). The normal (and only reasonable) interpretation of prefix operators demands a right-to-left ordering among themselves. We choose the same order for infix operators as a concession to consistency.

<u>Semantic Description of List Valued Operands for Numeric Operators</u>. If a numeric prefix operator finds a list as operand, we will follow Iverson ([15] p. 13) in generalizing the operator to yield the list of values obtained by applying the prefix operator to the members of the operand in order. If a numeric infix operator finds a value of type list and a value of type number as operands, the result is the list obtained by applying the operator successively between the number and elements of the list. If the operator finds two lists as operands, the result is the list obtained by applying the operator between corresponding members of the lists. The operation terminates on the shorter of the two lists.

More formally, let s and t be numbers and S and T be lists. Then if ⊙ is a numeric prefix operator, the following are equivalent: (See 3-13).

```
⊙ S        for all  v  from  S  do  ⊙v   .
```

If ⊙ is a numeric infix operator then the following are equivalent:

```
s ⊙ T      for all  v  from  T  do  s ⊙ v
S ⊙ t      for all  v  from  S  do  v ⊙ t
S ⊙ T      for all  i  from  1  to (length S) min (length T)
           do  S[i] ⊙ T[i]  .
```

Semantic Description of Compression. If Θ is any infix operator then the following are equivalent:

Θ / T for all v from T do $u \leftarrow u \Theta v$.

The latter depends upon the initial value of u for which we specify the following:

$\oplus$, "" ;

$\cup$, { } ;

$\cap$, universe ;

$\ominus$, Ω ;

base, Ω ;

$\vee$, 0 ;

$\wedge$, 1 ;

$<$, $\leq$, $=$, $\neq$, $\geq$, $>$, all undefined ;

max , $-\infty$;

min , ∞ ;

$+$, 0 ;

$-$, 0 ;

$\times$, 1 ;

mod , 1 ;

$\div$, 1 ;

$\uparrow$, Ω .

Justification of Compression. Compression, as well as the other generalizations of the numeric operators in the paragraphs preceding, is a concise way of expressing common programming tasks. Furthermore, as pointed out by R. S. Barton, they provide a mnemonic notation for ignoring the order of execution so that, if parallelism is available, it can be utilized. For example, the inner product:

$$+ / u \times v$$

of vectors of length n can be performed in $\log_2 n + 2$ operation times if n multipliers and $n \div 2$ adders are available.

Syntax of Step Lists.

```
<step list>   ::= <simple expr> to <simple expr>  |
                  <simple expr> by <simple expr>  |
                  <simple expr> by <simple expr> to <simple expr>  |
                  <simple expr> to <simple expr> by <simple expr>  |
                  <simple expr>
```

Examples of Step Lists.

2 by minus 2 to minus 16   1 to n

x-z to X[n] by 2   1 by 1

Semantic Description of Step Lists. A step list is a list of values of type number. The value of the first expression above is called the *initial value*; the value following the *to* is called the *limiting value*; the value following the *by* is called the *step value*. The evaluation of the step list proceeds as follows:

(1) All the expressions are evaluated in the order of their occurence in the program.

(2) If the step value is missing it is replaced by 1.

(3) If the limiting value is missing, it is replaced by a value of type undefined.

(4) If all the values thus computed are of type number, the step list has for value all the numbers of the form (initial value) + (n) x (step value) lying between (inclusive) the initial and limiting values where n ranges over the integers from 0 to infinity. If the limiting value is undefined the set is infinite, otherwise, it is undefined.

Syntax of Assignments.

<expression$_3$> ::= <primary$_1$> ← <expression$_3$>

Examples of Assignments.

a ← 1,   (if x = y then z[1] else z[2]) ← 7 ,

b ← 1 to n,   c[2][x-z] ← "ave." ,

c ← { new x, if(length b) = 3 then out ← "3" ,

out ← out ⊕ cr, x ← out }

Semantic Description of Assignments. The primary on the left must have a value of type name. If it does, the value of the named variable is replaced by the value of the expression on the right. The value of the expression is also the value of the assignment.

Justification of Assignments. The assignment allows the saving of temporary intermediate values. We also provide some flexibility in the designated variable on the left of the arrow (i.e., subscripted or unsubscripted identifiers and the parenthesized expressions). Both of

(if a = b then c else d) ← 3,

a ← P c, a ← 3,

are meaningful, and, if a initially equals b, have the same effect. In the first case the principle of deferment demands delaying the fetch of the value of c until the end of the conditional expression at which point we discover that it is the name that we want. In the second, we delay until after return from the procedure. The latter case is exactly the Algol 60 call-by-name construct.

Syntax of Procedures.

<expression$_3$> ::= <procedure> <expression$_3$>
<procedure> ::= P
<primary$_2$> ::= identifier <list> | identifier

Examples of Procedures.

```
increase ← P a ← a + 1 ,
increase ← P {new a, a ← a + 1} ,
increase{P a} ,
factorial ← P
   {new n,
    if n = 0 then 1 else n x factorial{n-1}
   }[1]
```

Semantic Description of Procedure Definition. A procedure definition is denoted by the mark (P) followed by an expression called the procedure expression. The execution of a procedure definition produces a value of type process. If the procedure expression is not an explicit list (or an explicit list followed by subscripts) then it is called a parameterless procedure.

Semantic Description of Parameterless Procedure Activation. (3-11)
Whenever the name of a variable is computed, that variable is inspected to determine whether or not it contains a value of type process. If it does, the process is activated and the name of the variable is replaced with the resulting value. If that value is again of type name, the test is repeated, etc., until a value of some other type is returned. If, at the time of procedure activation, all of the variables valid at the place of procedure definition are defined, then the effect, and the resulting value are the same as would be obtained by executing the procedure expression in the same environment at the place of definition.

Semantic Description of a Procedure with Parameters. (3-12)
If the procedure expression is an explicit list, then it has a (perhaps null) list of identifiers local to the scope defined by the list. We call the variables allocated to these identifiers the first, second, third, etc., initializable variables of that procedure.

Semantic Description of the Activation of a Procedure with Parameters. If the procedure activation is signified by an identifier followed by an explicit list, we call the list the actual parameter list. If the variable allocated to the identifier does not contain a value of type process, a value of type undefined is returned. Otherwise the activation is identical to that for the parameterless procedure[†] except that the initializable variables of the procedure are given the values of the corresponding actual parameters.

Justification of Values of Type Process.[††] The ability to define a process that can be activated upon demand is present in some form in Algol 60 procedures, functions, switches and name parameters. We have in the kernel language a single process defining construct. The value of a process may be of any type and the value may depend upon where the process is activated. (For instance, if a process is activated to the left of a replacement arrow it may return a value of type name rather than the actual value of the named variable.) Process recursion, the programming analogue of mathematical induction, is frequently the most natural way of expressing an algorithm in the kernel language.

The second and third examples above show the kernel language equivalent of Algol 60 name parameters. The local variable a is initialized to the procedure to compute a, a non local variable. Each occurence of the identifier a in the list body causes the procedure

---

[†] Note that since every access to the procedure identifier causes a procedure activation, there is no equivalent to the Algol 60 procedure assignment statement. If the procedure has parameters it is necessarily list-valued unless, as in the factorial example, a subscript is used to select the desired value.

[††] Values of type process are similiar to the quotations of Euler. The difference is that Euler quotations behaved differently when passed as parameters and when stored in local variables. We have eliminated the distinction.

to be activated. The first activation yields the name of the non local a since it is called to the left of the assignment arrow; the second yields the value. The result is that the non local a is increased by 1

<u>Implementation of Primaries of Type Process</u>. The necessity of accessing a word to compute its address is a consequence of the general-call-by-name concept from Algol 60. The provision for a special fast-access bit associated with the word is required for efficient implementation[†]

<u>Syntax of While-Controlled Iterations</u>.

```
<expression₃>    ::=  <while clause> do <expression₃>
<while clause>   ::=  <while> <step list>
<while>          ::=  while
```

<u>Examples of While-Controlled Iterations</u>.

```
while  in[1] ≠ " "[1] do (a ← a ⊕ in[(1)],
                           in ← in[2 to length in]
                         )

while x ↑ 2 ≠ a  do  x ← (x + a ÷ x) ÷ 2
```

<u>Semantic Description of While-Controlled Iterations</u>. A while-controlled iteration consists of a while clause and a controlled expression.

[†]The Burroughs B5500 has the special bit (called the flag bit) but it can be examined by the hardware only by accessing the word. Thus even in the assignment

$$a \leftarrow a$$

three memory references are required.

The while clause is evaluated; if it has value true then the controlled expression is evaluated and we return to re-evaluate the while clause; if it has value false we terminate the iteration and the value of the while-controlled iteration is the list of values of the controlled expression; if it has any other value, the iteration is terminated with a value of type undefined.

Syntax of For-Controlled Iterations.

<expression$_3$>	::=	<for clause> do <expression$_3$> |
		<for clause> <while clause> do <expression$_3$>
<for clause>	::=	for all identifier from <step list>
<while clause>	::=	<while> <step list>
<while>	::=	while

Examples of For-Controlled Iterations.

for all I from 1 to n do S ← S + I ↑ 3,

+ / for all I from 1 to n do I ↑ 3,

for all t from table while looking do

    if t = object then looking ← false else emit(0)

Semantic Description of For-Controlled Iterations. (3-13)

The for-controlled iteration provides for the execution of the controlled expression of a fixed number of times or a fixed number of times with the possibility of an early termination. The step list of the for clause is evaluated once; if it is not list or set valued, the value of the for-controlled expression is of type undefined. The scope of the identifier of the for clause is the controlled expression. The variable allocated to the identifier assumes in order each value from the iteration set and the controlled expression is executed. If there is a while clause and its value is not true before the execution of the controlled expression, the iteration is terminated.

The value of the for-controlled expression is the list of values assumed by the controlled expression.

<u>Syntax</u> <u>of</u> <u>Conditional</u> <u>Expressions</u>.

<expression> ::= $\langle\text{expression}_1\rangle$

$\langle\text{expression}_1\rangle$ ::= <if clause> $\langle\text{expression}_1\rangle$ | $\langle\text{expression}_2\rangle$

$\langle\text{expression}_2\rangle$ ::= $\langle\text{expression}_3\rangle$

$\langle\text{expression}_3\rangle$ ::= <if clause> <truepart> $\langle\text{expression}_3\rangle$

<if clause> ::= <u>if</u> <expression> <u>then</u>

<truepart> ::= $\langle\text{expression}_2\rangle$ <u>else</u>

<u>Examples</u> <u>of</u> <u>Conditional</u> <u>Expressions</u>.

if $x = y$ then if $y \neq z$ then $x \leftarrow y$ max z ,

if test{7} then $\{x \leftarrow 1,\ y \leftarrow 2\}$ else $x \leftarrow 3$,

if if $A \subset B$ then true else $z \notin B$ then $B \leftarrow \{\}$

<u>Semantic</u> <u>Description</u> <u>of</u> <u>Conditional</u> <u>Expressions</u>. The first form of conditional expression is an if clause followed by an expression. The if clause is evaluated; if it has value true the expression is evaluated and the value obtained is the value of the conditional expression; otherwise the value of the conditional expression is of type undefined.

For the second form we evaluate the conditional expression; if it is true we evaluate the truepart expression; if it is false we evaluate the final expression; otherwise we create a value of type undefined.

Syntax of Programs.

<program> ::= ⊢ <expression> ⊣

Semantic Description of a Program. The value of a program is the value of the expression. Note that by the nature of the kernel language (identification of Algol 60 blocks and values of type list) the value of a program will be a list structure of the intermediate results.

Implementation of a Program. On account of the copious list structure generated by a program, we must have some form of remote storage and recall mechanism. The list structure of the program is well suited for segmentation and overlay.

BIBLIOGRAPHY

[1] R. S. Barton, "A new approach to the functional design of a digital computer", <u>Proc. WJCC</u> (1961), p. 393.

[2] Noam Chomsky, Syntactic Structures, Mouton & Co. (1964).

[3] _______, "Formal properties of grammars", <u>Handbook of Mathematical Psychology</u>, Vol. II, John Wiley & Sons (June 1963), pp. 323-418.

[4] Jay C. Early, "Generating a recognizer for a BNF grammar", Technical report, Carnegie Institute of Technology (August 1965).

[5] Jurgen Eickel and Manfred Paul, "The parsing and ambiguity problem for Chomsky-languages", Report 6409, Computation Center, Mathematics Institute, Technical High School, Munich (no date).

[6] Jurgen Eickel, "Generation of parsing algorithms for Chomsky type-2 languages", Report 6401, Computation Center, Mathematics Institute, Technical High School, Munich.

[7] Robert W. Floyd, "Syntactic analysis and operator precedence", <u>J. ACM</u>, vol. 10, no.3 (July 1963), pp. 316-333.

[8] _______, "Bounded Context Syntactic Analysis", <u>Comm. ACM</u> vol. 7, no 2 (Feb. 1964), p. 62.

[9] _______, private communication (Jan. 1966).

[10] Philip Gilbert, "On the syntax of algorithmic languages", <u>J. ACM</u>, vol. 13, no. 1 (June 1966), pp. 90-107.

[11] Seymour Ginsburg and Joeseph Ullian, "Ambiguity in context free languages", <u>J. ACM</u>, vol. 13, no. 1 (Jan. 1966), pp. 62-89.

[12] Sheila A. Greibach, "A new normal-form theorem for context free phrase structure grammars", <u>J. ACM</u>, vol. 12, no. 1 (Jan. 1965), pp. 42-52.

[13] C.A.R. Hoare, "Case Expressions", Algol Bulletin 15 (Oct. 1964) pp. 20-22.

[14] IBM Systems Reference Library, PL/I: Language Specifications, Form C28-6571 (no date).

[15] Kenneth Iverson, A Programming Language, John Wiley & Sons (1962).

[16] Donald Knuth, "On the translation of languages from left to right", Information and Control, vol. 8 (Dec. 1965), p. 607.

[17] F. E. J. Kruseman Aretz, "Algol 60 translation for everybody", Elektronische Datenverarbeitung, vol. 6 (1964), p. 233.

[18] Reino Kurki-Suonio, "On character set reduction", Technical report, Carnegie Institute of Technology, (August 1965).

[19] W. C. McGee and H. E. Petersen, "Microprogram control for experimental sciences", Proc. FJCC, (Sept. 1965), p. 77.

[20] W. M. McKeeman, "MUTANT, An Extendable Compiler on the Burroughs B5500", (to be published).

[21] Naur, et al, "Revised Report on the Algorithmic Language ALGOL 60", Comm. ACM, vol. 6 (Jan. 1963), pp. 1-17.

[22] John Reynolds, "COGENT Programming Manual", Report 7022, Argonne National Laboratory (March 1965).

[23] A. van Wijngaarden, "Recursive definition of syntax and semantics", IFIP working conference (Sept. 1964).

[24] _______, "Generalized Algol", Symbolic languages in data processing, (Rome 1962), 409 pp.

[25] N. Wirth and H. Weber, "Euler: A generalization of Algol, and its formal definition", Technical Report CS20, Computer Science Department, Stanford University (April 1965) (also in part - Comm. ACM, vol. 9, nos. 1 & 2 (Jan. 1966)).

[26] N. Wirth, "Find precedence functions", (Algorithm 265) Comm. ACM, vol. 8, no. 10 (October 1965).

[27] N. Wirth and C.A.R. Hoare, "A contribution to the development of ALGOL", Technical Report CS35, Computer Science Department, Stanford University (Feb. 1966).

UNCLASSIFIED
Security Classification

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) Computer Science Department Stanford University Stanford, Calif. 94305	2a. REPORT SECURITY CLASSIFICATION Unclass. 2b. GROUP ---

3. REPORT TITLE

AN APPROACH TO COMPUTER LANGUAGE DESIGN

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
Manuscript for Publication (Technical Report)

5. AUTHOR(S) (Last name, first name, initial)

McKEEMAN, William M.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
August 31, 1966	124	27

8a. CONTRACT OR GRANT NO. Nonr-225(37) b. PROJECT NO. NR-044-211 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) CS48 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) none

10. AVAILABILITY/LIMITATION NOTICES

Releasable without limitations on dissemination

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
---	Office of Naval Research Code 432 Washington, D. C, 20360

13. ABSTRACT

A kernel programming language includes those constructs universally applicable to the problem of computer control. The need for constructs outside the kernel leads to the concept of an extendable compiler. We approach this problem by attempting to simplify the methods of generating a compiler and by designing a basic language upon which to build.

The form of a language is determined by its grammar. In particular, we demand a context free grammar as the initial input to a syntax preprocessor which produces syntactic analysis tables for the extendable compiler. The methods are extensions of the precedence grammars of Floyd and Wirth-Weber.

A formal mathematical description of a class of analysis algorithms including the above is given and two new syntax preprocessor algorithms are presented. Some theorems concerning the behavior of the algorithms and the nature of the acceptable grammars are given.

(cont.)

DD FORM 1 JAN 64 1473

UNCLASSIFIED
Security Classification

Unclassified
Security Classification

d548

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
1. Compiler						
2. Computer-language						
3. Parsing						
4. Context-free grammar.						
5. Extendable compiler						

INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2a. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2b. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7a. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7b. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8a. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8b, 8c, & 8d. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9a. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9b. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through
_______________________________________."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through
_______________________________________."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through
_______________________________________."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS), (S), (C), or (U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, roles, and weights is optional.

DD FORM 1 JAN 64 1473 (BACK)

Unclassified
Security Classification

In language design, we attempt to carry the EULER development by Wirth and Weber to a more concise and powerful form. We advocate languages that are minimal and involuted. A minimal language combines into a single construct any two conceptually similar but notationally different constructs. An involuted language avoids constructs that are applicable only in local context. In the resulting language we find such previously diverse constructs as lists, parameter lists, blocks, compound statements, for lists, and arrays to be identical. After combining the features of the reduced EULER with some ideas from Iverson and PL/I we find that our control over the flow of execution within a program is sufficiently complete such that we can discard the traditional label and go-to statement as irrelevant.

As a final example of the kernel language, we present an extendable compiler written in the kernel language itself.

Our conclusions are that the precedence grammar techniques are quite efficient and useful. Further improvement could make them substantially superior to other methods of compiler generation. We believe that the computing community would be better served with a minimal common language which the user would routinely extend than by any large general purpose language. Finally we believe that the growing agreement on the constructs common to all programming task should have a much more significant effect upon machine design than is presently the case.