Technical Report TR-199 September 1972
NGL-21-002-270

REPRESENTATIONS OF THE LANGUAGE RECOGNITION PROBLEM FOR A THEOREM PROVER

by

Jack Minker
Gordon J. VanderBrug

UNIVERSITY OF MARYLAND
COMPUTER SCIENCE CENTER
COLLEGE PARK, MARYLAND

Technical Report TR-199 September 1972
NGL-21-002-270

REPRESENTATIONS OF THE LANGUAGE RECOGNITION PROBLEM FOR A THEOREM PROVER

by

Jack Minker
Gordon J. VanderBrug

September 1972

This work was supported by the National Aeronautics and Space Administration Grant NGR-21-002-270.

The authors acknowledge Mr. James Albus and Mr. Sam Rosenfeld of the National Aeronautics and Space Administration for their support of this work. We express our appreciation to Mr. Daniel Fishman of the University of Maryland for participating in the early efforts and for valuable discussions. We wish to thank Dr. Cordell Green of Stanford University for reading an early draft of the work and for providing many suggestions for improvements.

Page Intentionally Left Blank

TABLE OF CONTENTS

ABSTRACT

Two representations of the language recognition problem for a theorem prover in first-order logic are presented and contrasted. One of the representations is based on the familiar method of generating sentential forms of the language, and the other is based on the Cocke parsing algorithm. An augmented theorem prover is described which permits recognition of recursive languages. The state-transformation method developed by Cordell Green to construct problem solutions in resolution-based systems can be used to obtain the parse tree. In particular, the end-order traversal of the parse tree is derived in one of the representations. The paper defines an inference system, termed the cycle inference system, which makes it possible for the theorem prover to model the method on which the representation is based. The general applicability of the cycle inference system to state-space problems is discussed. Given an unsatisfiable set S, where each clause has at most one positive literal, it is shown that there exists an input proof. The clauses for the two representations satisfy these conditions, as do many state-space problems.

1. Introduction

In this paper we are concerned with the application of mechanical theorem proving in first-order logic to the problem domain of language recognition. The languages under consideration are those generated from grammars of the Chomsky hierarchy consisting of unrestricted, context-sensitive, context-free, and regular grammars.

Selecting this problem domain for study was based on the realization that a great deal is known about recognizing languages in the Chomsky hierarchy. Hence, the results that one obtains can be compared with an existing body of knowledge. The Resolution Principle is a rather powerful tool, and hence cumbersome and too general for many problem domains. Our hope was to shed some light on this particular problem. That is, assuming that mechanical theorem proving can be applied to the language recognition problem, can space and time bounds be obtained that compare favorably to those of existing techniques?

We show how to represent the problem of language recognition in the first-order predicate calculus and thereby permit the use of a mechanical theorem prover based on the Robinson Resolution Principle. We further show that several different representations of the language recognition may be formulated and, depending on the particular representation, comparable results may be obtained for context-free languages. A representation modeled after the Cocke recognizer (Cocke and Schwartz, 1970; Robinson and Marks, 1965) led to comparable results for context-free language recognition considering both time and space. The representations developed for recursive languages are complete and sound. An augmented theorem prover is described that permits determination of whether or not given strings are in a recursive language. A resolution-based inference system, called the cycle inference system, was designed to efficiently generate deductions from the representations presented in this paper and from similar types of representations. Given an unsatisfiable set S, where each clause has at most one positive literal, it is shown that there exists an input proof. The clauses for the two representations satisfy these conditions, as do many state-space problems.

In Section 2 of this report we provide background for those unfamiliar with problem representation and search, language recognition, and theorem proving. Section 3 presents two representations of the language recognition problem, the derivation sequence representation and the derivation tree representation. How the cycle inference system and an augmented theorem prover can recognize languages using these representations is described in Section 4. Section 5 shows how to extract a parse tree. Finally, Section 6 discusses the conclusions, and in particular, the general applicability of the cycle inference system.

2. Background

This section provides the reader with some background material in the areas of problem representation and search, language recognition, and theorem proving.

2.1 Problem Representation and Search

Problem-solving may be considered to consist of two parts. First, the problem must be formulated for the problem-solver. The result of this "setting-up" process is called a representation of the problem. The second part is the process of determining the solution to the problem from the representation, and is termed search. Beyond simply representing a problem, one is interested in finding a good representation. Developing a good representation of a problem involves finding a way of looking at the problem which simplifies the process of finding a solution. As noted by Amarel (Amarel, 1968), the problem of representation is concerned with the relationship between different ways of formulating a problem to a problem-solving system, and the efficiency with which the system can be expected to find a solution. Some representations lend themselves to fast search techniques and hence can be thought of as good representations. However, a good representation combined with inefficient search techniques will not result in efficient solutions. On the other hand, one could have a very efficient search which operates inefficiently because of the choice of a poor representation. Thus, the process of finding a solution to a problem could be inefficient because of a poor representation, a poor search technique, or both.

When the problem-solving system is a resolution-based theorem prover, the problem of representation becomes one of formulating the problem in the first-order predicate calculus. The formulation of the problem must have the property that solutions to the problem, if there are any, logically follow from it. The problem of finding efficient search techniques becomes one of directing the theorem prover towards a refutation (see Section 2.3).

Some problems can be represented in what has been called a state-space problem representation (Nilsson, 1971). In this approach, a problem is represented by a set of operators which transform one state description into another. The definition of a state-space representation of a problem consists of:

(1) the specification of the form of a state description and the initial state description.

(2) the specification of the set of operators and the transformations which they make on the state descriptions.

(3) the specification of the properties of goal state descriptors.

The space which is defined by a state-space problem representation can be thought of as a graph where the nodes represent states and the arcs represent operators. The representation defines the graph implicitly. Search can then be viewed as making explicit part (or possibly all of the space if it is finite) of an implicit graph.

There are many algorithms for searching the space of a state-space problem representation (Nilsson, 1971). These include the breadth-first and uniform-cost algorithms. The uniform-cost algorithm evaluates the merit of a node on the basis of the cost it took to reach the node, and chooses from among all the unexpanded nodes the one with minimal cost. Other search algorithms rank the unexpanded nodes on the basis of a heuristic judgment on how close each node is to a goal node (Michie, 1970). Hart, Nilsson, and Raphael (Hart, et al, 1968) have developed an algorithm which uses a merit ordering that has both a cost component and heuristic component.

2.2 Language Recognition

We denote, as is usual, a grammar to be a quadruplet (V_N, V_T, P, S) where the symbols V_N, V_T, P and S are, respectively, the nonterminals, terminals, productions, and start symbol. V_N, V_T, and P are finite sets, and $V_N \cap V_T$ is the empty set, while $V = V_N \cup V_T$ is termed the vocabulary of G.

The set of productions P consist of expressions of the form $\alpha \rightarrow \beta$ where α is a string in V^+ and β is a string in V^*. By V^* we mean the set of all strings composed of symbols of V, including the empty sentence, and $V^+ = V^* - \{\varepsilon\}$, where ε is the empty sentence.

To describe the language generated by a grammar we first define the relations $\overset{*}{\underset{G}{\Rightarrow}}$ and $\overset{*}{\underset{G}{\Rightarrow}}$ between strings in V^+. If $\alpha \rightarrow \beta$ is a production of P, and γ and δ are strings in V^+, then $\gamma\alpha\delta \underset{G}{\Rightarrow} \gamma\beta\delta$. That is, the production applied to the strings $\gamma\alpha\delta$ results in a new string $\gamma\beta\delta$. Now if $\alpha_1,\ldots,\alpha_n$ are strings in V^+, and $\alpha_1 \underset{G}{\Rightarrow} \alpha_2$, $\alpha_2 \underset{G}{\Rightarrow} \alpha_3$, $\ldots,\alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m$, then we say the α_1 derives α_m and write $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$. The sequence $\alpha_1,\ldots,\alpha_m$ is called a derivation sequence. If $\alpha_1 = S$, the start symbol of G, then the α_i are called sentential forms of G.

A language generated by G, denoted by L(G), is defined to be $\{w | S \overset{*}{\underset{G}{\Rightarrow}} w$ and w is in $V_T^+\}$. Thus a string is in L(G) if it is a sentential form of G which consists solely of terminal symbols. Since it will be clear as to what grammar is being referred to we shall drop the G that appears below the double arrow.

The Chomsky hierarchy of grammars consists of four different types of grammars distinguished by restrictions made on the nature of the productions used in the grammar. The grammar defined above, which makes no restrictions on the productions is termed unrestricted. If every production

$\alpha \to \beta$ of P has the property that $|\beta| \geq |\alpha|$, where $|x|$ denotes the number of symbols of the string x, the grammar is called context-sensitive. If every production $\alpha \to \beta$ of P is such that α is a single nonterminal and β is any non-empty string, then the grammar is termed context-free. Finally, if every production $\alpha \to \beta$ of P is either of the form $A \to aB$ or $A \to a$, where A and B are nonterminals and a is a terminal, then the grammar is called a regular grammar.

A language generated by an unrestricted, context-sensitive, context-free, or regular grammar is termed an unrestricted, context-sensitive, context-free, or regular language respectively.

We say that a grammar G is recursive if there is an algorithm which will determine for any string w, whether w is generated by G. Context-sensitive, context-free, and regular grammars are recursive, because the length preserving property of the productions makes it possible to successively generate all of the sentential forms in increasing lengths. A straightforward algorithm to determine whether or not w is in L(G) is to generate all of the sentential forms of length less than or equal to the length of w. If w is among them then w is in L(G); otherwise w is not in L(G).

A derivation of a sentence in a context-free language can be represented by a tree, called a derivation tree. The root of a derivation tree of a sentence is labeled with the start symbol of the grammar, the non-leaf nodes are labeled with nonterminals in a way which represents the applications of the productions to the sentential forms, and the leaves are labeled from left to right with the terminal symbols of the sentence.

Algorithms for recognizing context-free languages have been developed by Cocke (Cocke and Schwartz, 1970; Robinson and Marks, 1965) and by

Younger (Younger, 1967). They require that the grammar be in Chomsky Normal Form (Hopcroft and Ullman, 1969). A context-free grammar is in Chomsky Normal Form if all of its productions are of the form $A \rightarrow a$ or $A \rightarrow BC$, where A, B, and C are nonterminals and a is a terminal. Chomsky (Chomsky, 1959) has shown that every context-free grammar can be placed in Chomsky Normal Form. The recognizers of Cocke and Younger require a time proportional to n^3, where n is the length of the input string.

The basis of the algorithms of Cocke and Younger is the construction of a triangular array whose entries are sets of nonterminals. All of the nonterminals in a given set derive a substring of the input string which is determined by the location of the set in the array. The only difference between the two algorithms is the parameters which are used to define the mapping from the location of a set to the derived substring. Since the logic of the Cocke algorithm is the method on which one of the representations of the paper (the derivation tree representation) is based, the algorithm is now discussed in some detail. A representation similar to the derivation tree representation which is based on the Younger algorithm can easily be constructed.

Let $[d_{i,j}]$ be the triangular array, called the span triangle. We define elements of the set $d_{i,j}$ to be the nonterminals which derive the substring which begins at the ith position of the input string and ends at the (j-1)st position. That is, $A \varepsilon d_{i,j}$ iff $A \overset{*}{\Rightarrow} a_i \ldots a_{j-1}$, where $a_1 \ldots a_n$ is the input string. An entry $d_{i,j}$ in the span triangle is equal to the empty set whenever there is no nonterminal which derives the substring which begins at the ith position of the input string and ends at the (j-1)st position.

The algorithm is to compute the span triangle to determine whether or not the start symbol is in $d_{1,n+1}$. If $S \varepsilon d_{1,n+1}$ then the input string is accepted, since $S \overset{*}{\Rightarrow} a_1 \ldots a_n$ iff $S \varepsilon d_{1,n+1}$.

The elements of the first row of the triangle are determined directly from the terminal productions of the grammar by the rule: $A \varepsilon d_{i,i+1}$, $1 \leq i \leq n$ iff $A \to a_i$ is a production in the grammar. To determine the remaining elements of the span triangle, the following rule is used. The nonterminal $A \varepsilon d_{i,j}$ iff there is a production $A \to BC$ and an integer ℓ, such that $i < \ell < j$, $B \varepsilon d_{i,\ell}$, and $C \varepsilon d_{\ell,j}$. The rule is based on the fact that $A \overset{*}{\Rightarrow} a_i \ldots a_{j-1}$ iff there is a production $A \to BC$ and an integer ℓ, such that $i < \ell < j$, $B \overset{*}{\Rightarrow} a_i \ldots a_{\ell-1}$, and $C \overset{*}{\Rightarrow} a_\ell \ldots a_{j-1}$. That is, A derives a substring iff there is a production $A \to BC$ in the grammar and the substring can be split into two parts in such a way that B derives the first part of the substring and C derives the second part. A simple induction proof will show that $S \varepsilon d_{1,n+1}$ iff $S \overset{*}{\Rightarrow} a_1 \ldots a_n$, that is, the algorithm does recognize a context-free language.

2.3 Theorem Proving Fundamentals

In dealing with theorem proving in this paper, we consider only the case where we are proving theorems within the first-order predicate calculus without equality. We assume that the reader is familiar with the first-order predicate calculus, and provide only that background in theorem proving dealing with the Robinson Resolution Principle.

By a clause we mean a disjunction of literals. A literal is either an atom or the negation of an atom. An atom is a predicate letter with terms as arguments of the predicate. We shall use $|\overline{\underline{\ }}|$ as notation for the null clause, the clause which is always false. Given a well-formed-formula (wff) within the first-order predicate calculus, there exists an algorithm that will transform the wff into clause form (Davis, 1963). The transformation is model preserving. That is, if the original wff is valid under some interpretation, then the clause form is valid under that interpretation. Clause form has been referred to as the Language of Davis and Putnam (Davis and Putnam, 1960).

The result of applying a substitution σ to an expression E is denoted by $E\sigma$. That is, a substitution is a replacement of a variable within the expression by a term. If $E\sigma = F$ for some σ, then F is said to be an <u>instance</u> of E. We shall say that E has been instantiated when we have made a substitution in E. In the event that F has no variables, then F is a <u>ground expression</u> and a <u>ground instance</u> of E.

If expressions E and F have a common instance G, than E and F are unifiable and there is a most general common instance $E\sigma = F\sigma$, where σ is the <u>most general unifier</u> of E and F. The most general unifier of E and F is such that if μ is any unifier of E and F, then there is a λ such that $\mu = \sigma\lambda$. Robinson (Robinson, 1965) has developed an algorithm for unifying two expressions.

A <u>factor</u> of a clause C is a clause $C\sigma$, where σ is the most general unifier of a subset of literals of C. Let C_1 and C_2 be two clauses with the literal L_1 in C_1 and the negation of the literal L_2 in C_2. If the set $\{L_1, \sim L_2\}$ is unifiable with most general unifier σ, then we define the <u>resolvent</u> of C_1, and C_2 to be the clause $C_3 = \{(C_1 - L_1) \cup (C_2 - L_2)\}\sigma$. The clauses C_1 and C_2 are called <u>resolvends</u>. It can be shown (Kowalski and Hayes, 1969) that if the two resolvends are valid, then the resolvent is valid; and that if a clause C is valid, then so is a factor of C.

Given a set S of clauses that are unsatisfiable, then if we form the set $R(S)$ which contains the set S and the resolvents of all pairs of clauses in S or factors of clauses in S, and if we define $R^0(S) = S$, and $R^{n+1}(S) = R(R^n(S))$, then for some finite n, the null clause will be in $R^n(S)$. The process of forming the sets $R^n(S)$ is termed the Resolution Principle. Robinson (Robinson, 1965) has shown that the Resolution Principle is complete and sound. It is termed complete because when given a set S of unsatisfiable clauses, by applying the Resolution Principle the null clause will result. It is called sound because when given a set S of clauses and given that resolution applied to this set of clauses results in the null clause, the set S of clauses is unsatisfiable.

3. The Representations

This section presents the derivation sequence representation for recognizing a language within the Chomsky hierarchy, and the derivation tree representation of the problem of recognizing a context-free language, and illustrates each representation for a particular language. The former requires an axiom schema to represent the problem, while the latter uses a fixed number of axioms.

3.1 The Derivation Sequence Representation

Consider the problem of recognizing $L(G)$ where $G = (V_N,V_T,P,S)$. The constants used in the representation are $V_N \cup V_T$. No functions are used. There are two predicate schemas used in the representation. They are $\mathrm{PROD}_{j,k}(a_1,\ldots,a_j,b_1,\ldots,b_k)$ and $\mathrm{SENTFORM}_k(x_1,\ldots,x_k)$, where j and k are positive integers. The first predicate may be interpreted to mean that $a_1\ldots a_j \to b_1\ldots b_k$ is a production in G, and the second may be interpreted to mean that $x_1\ldots x_k$ is a sentential form in $L(G)$. The axioms of the representation are:

$$\mathrm{DS1} = \{\mathrm{PROD}_{j,k}(a_1,\ldots,a_j,b_1,\ldots,b_k) \mid j\ ,\ k \text{ are positive integers and } a_1\ldots a_j \to b_1\ldots b_k \text{ is in } P\}$$

$$\mathrm{DS2} = \{\mathrm{SENTFORM}_1(S) \mid S \text{ is the start symbol of } G\}$$

$$\mathrm{DS3} = \{(\forall x_1,\ldots,x_m,y_1,\ldots,y_k)\ [\mathrm{SENTFORM}_m(x_1,\ldots,x_m) \wedge \mathrm{PROD}_{j,k}(x_{i+1},\ldots,x_{i+j},y_1,\ldots,y_k) \to \mathrm{SENTFORM}_{m+k-j}(x_1,\ldots,x_i,y_1,\ldots,y_k,x_{i+j+1},\ldots,x_m)] \mid m\ ,\ j\ ,\ i\ ,\ k \text{ are integers such that } 1 \le m\ ,\ 1 \le j \le m\ ,\ 0 \le i \le m-1\ ,\ 1 \le k\}\ .$$

The set of axioms DS1 and the axiom DS2 represent the productions of G and the start symbol of G respectively. The set of axioms DS3 represents to the problem-solver the fact that productions specify which substrings of sentential forms can be transformed, and what substrings they can be transformed into. The representation of the problem of recognizing $L(G_1)$, where the grammar G_1 (Hopcroft and Ullman, 1969) is shown in Figure 3.1, is given in Figure 3.2.

The representation of the language recognition problem depends on the type of the grammar G. Since the lengths of sentential forms for unrestricted grammars are unbounded, the set of axioms DS3 which would be required to represent the language recognition problem for unrestricted languages is infinite. Indeed, the derivation sequence representation does not represent the language recognition problem for unrestricted languages, because the procedure of generating sentential forms to perform the recognition process does not always halt for unrestricted languages.

For recursive languages the sentential forms that can be used to recognize a sentence of length n cannot exceed n in length. This places the bound $m + j - k \leq n$ on the set of axioms DS3. Thus for any particular recursive language only finitely many axioms are required, and the derivation sequence representation is indeed a representation. The left hand side of productions in a context-free grammar have length one, thus the integer j in DS1 and DS3 is one for the problem of recognizing a context-free language. For right-recursive regular languages the representation becomes (b_1 is a terminal symbol):

$$DS1' = \{PROD_{1,2}(a_1,b_1,b_2) \mid a_1 \rightarrow b_1 b_2 \text{ is in } P\} \cup \{PROD_{1,1}(a_1,b_1) \mid a_1 \rightarrow b_1 \text{ is in } P\}$$

$$DS2' = \{SENTFORM_1(S) \mid S \text{ is the start symbol of } G\}$$

$$G_1 = (V_N, V_T, P, S)$$

where

$$V_N = \{S, B, C\}$$

$$V_T = \{a, b, c\}$$

$$P = \left\{ \begin{array}{l} S \rightarrow aSBC \\ S \rightarrow aBC \\ CB \rightarrow BC \\ aB \rightarrow ab \\ bB \rightarrow bb \\ bC \rightarrow bc \\ cC \rightarrow CC \end{array} \right\}$$

FIGURE 3.1 A context-sensitive grammar, G_1.

$$DS1 = \left\{ \begin{array}{l} PROD_{1,4}(S,a,S,B,C) \\ PROD_{1,3}(S,a,B,C) \\ PROD_{2,2}(C,B,B,C) \\ PROD_{2,2}(a,B,a,b) \\ PROD_{2,2}(b,B,b,b) \\ PROD_{2,2}(b,C,b,c) \\ PROD_{2,2}(c,C,c,c) \end{array} \right\}$$

$$DS2 = \{SENTFORM_1(S)\}$$

$$DS3 = \{ (\forall x_1,\ldots,x_m,y_1,\ldots,y_k)\ [SENTFORM_m(x_1,\ldots,x_m) \wedge PROD_{j,k}(x_{i+1},\ldots,x_{i+j},y_1,\ldots,y_k) \rightarrow SENTFORM_{m+k-j}(x_1,\ldots,x_i, y_1,\ldots,y_k,x_{i+j+1},\ldots,x_m)] \mid (j,k) \text{ is } (1,4),\ (1,3),\ \text{or } (2,2)\}$$

FIGURE 3.2 The derivation sequence representation for the grammar G_1.

$$DS3' = \{(\forall x_1,\ldots,x_m,y_1,y_2)\ [SENTFORM_m(x_1,\ldots,x_m) \wedge PROD_{1,2}(x_m,y_1,y_2) \rightarrow SENTFORM_{m+1}(x_1,\ldots,x_{m-1},y_1,y_2)\ |\ 1 \leq m]\} \cup \{(\forall x_1,\ldots,x_m,y_1)\ [SENTFORM_m(x_1,\ldots,x_m) \wedge PROD_{1,1}(x_m,y_1) \rightarrow SENTFORM_m(x_1,\ldots,x_{m-1},y_1)\ |\ 1 \leq m]\}.$$

Not only does the representation depend on the type of the grammar which generates the language which is to be recognized, it also depends on the particular grammar. Thus, the set DS3 for the example of Figure 3.2 does not include any axioms which contain the predicate symbol $PROD_{j,k}$ except for $PROD_{1,4}$, $PROD_{1,3}$, and $PROD_{2,2}$.

3.2 The Derivation Tree Representation

Unlike the derivation sequence representation which is applicable to all languages in the Chomsky hierarchy, the derivation tree representation is only applicable to context-free languages whose grammars are in Chomsky Normal Form. Let $G = (V_N,V_T,P,S)$ be a context-free grammar which is in Chomsky Normal Form. The derivation tree representation of the problem of recognizing L(G) uses the constants $V_N \cup V_T$, and the following functions and predicates.

add(u,v)	the addition function
$PROD_{1,1}(a,b)$	$a \rightarrow b$ is in P
$PROD_{1,2}(a,b,c)$	$a \rightarrow bc$ is in P
INPUT(x,u)	x is the uth symbol in the input string
DERIVE(x,u,v)	x derives the substring of the input string which begins at the uth position and ends at the (v-1)st position.

The axioms of the representation are:

DT1 = $\{PROD_{1,1}(a,b) \mid a \rightarrow b$ in in P$\} \cup \{PROD_{1,2}(a,b,c) \mid a \rightarrow bc$ is in P$\}$

DT2 = $\{(\forall y,u)\cdot[INPUT(y,u) \wedge PROD_{1,1}(x,y) \rightarrow DERIVE(x,u,add(u,1))]\}$

DT3 = $\{(\forall x,y,z,u,v,w)\ [DERIVE(y,u,w) \wedge DERIVE(z,w,v) \wedge PROD(x,y,z) \rightarrow DERIVE(x,u,v]\}$.

The axiom DT2 represents to the problem-solver the fact that if y is the u^{th} input symbol and $x \rightarrow y$ is a production, then x derives the substring of the input string which begins at the u^{th} position and ends at the u^{th} position. Thus if $a_1 \ldots a_n$ is the input string, then $a \overset{*}{=>} a_u$. The axiom DT3 represents the fact that if $y \overset{*}{=>} a_u \ldots a_{w-1}$, $z \overset{*}{=>} a_w \ldots a_{v-1}$, and $x \rightarrow yz$ are true, then $x \overset{*}{=>} a_u \ldots a_{v-1}$. DT2 will be used by the problem-solver to deduce the syntactic categories of the terminal symbols of the input string, and DT3 will be used to deduce which nonterminals derive which substrings of the input strings.

The representation of the problem of recognizing $L(G_2)$, where the grammar G_2 is given in Figure 3.3, is shown in Figure 3.4 for the well-known string, "they are flying planes."

$G_2 = (V_N, V_T, P, SENT)$

$V_N = \{SENT, V, VP, NP, PRN, COP, AUX, ADJ, VPROG, N\}$

$V_T = \{\text{they}, \text{are}, \text{flying}, \text{planes}\}$

$$P = \left\{ \begin{array}{llll} \text{SENT} & \rightarrow & \text{PRN} & \text{VP} \\ \text{VP} & \rightarrow & \text{V} & \text{N} \\ \text{VP} & \rightarrow & \text{COP} & \text{NP} \\ \text{SENT} & \rightarrow & \text{PRN} & \text{V} \\ \text{V} & \rightarrow & \text{AUX} & \text{VPROG} \\ \text{NP} & \rightarrow & \text{ADJ} & \text{N} \\ \text{PRN} & \rightarrow & \text{they} & \\ \text{COP} & \rightarrow & \text{are} & \\ \text{AUX} & \rightarrow & \text{are} & \\ \text{ADJ} & \rightarrow & \text{flying} & \\ \text{VPROG} & \rightarrow & \text{flying} & \\ \text{N} & \rightarrow & \text{planes} & \end{array} \right\}$$

FIGURE 3.3 A context-free grammar, G_2.

$$DT1 = \left\{ \begin{array}{l} PROD_{1,2}(SENT,PRN,VP) \\ PROD_{1,2}(VP,V,N) \\ PROD_{1,2}(VP,COP,NP) \\ PROD_{1,2}(SENT,PRN,V) \\ PROD_{1,2}(V,AUX,VPROG) \\ PROD_{1,2}(NP,ADJ,N) \\ PROD_{1,1}(PRN,they) \\ PROD_{1,1}(COP,are) \\ PROD_{1,1}(AUX,are) \\ PROD_{1,1}(ADJ,flying) \\ PROD_{1,1}(VPROG,flying) \\ PROD_{1,1}(N,planes) \end{array} \right\}$$

$$DT2 = (\forall y,u)[INPUT(y,u) \wedge PROD_{1,1}(x,y) \rightarrow DERIVE(x,u,add(u,1))]$$

$$DT3 = (\forall x,y,z,u,v,w)[DERIVE(y,u,w) \wedge DERIVE(z,w,v) \wedge PROD(x,y,z) \rightarrow DERIVE(x,u,v)]$$

FIGURE 3.4 The derivation tree representation for the grammar G_2.

4. Language Recognition in the Representation

In Section 3 two representations of the language recognition problem and examples of each were presented. In this section we show how the representations are used to perform recognition. To prove that a string is in a language $L(G)$ the string is represented as a theorem and the set of first-order predicate calculus statements consisting of the axioms of the representation of G and the negation of the theorem are shown to be unsatisfiable. Section 4.3 explains the use of what we have termed the cycle inference system, which reduces the number of clauses necessary to perform recognition. Recognizing a string in a language includes being able to decide that the string is not in the language as well as that it is in the language. Deciding that a string is not in a recursive language can be accomplished by an augmented theorem prover (described in Section 4.4).

4.1 Recognition in the Derivation Sequence Representation

The representation of the input string $a_1 \ldots a_n$ as a theorem is $\mathrm{SENTFORM}_n(a_1,\ldots,a_n)$. Figure 4.1 illustrates a proof in the representation given in Figure 3.2 that $\mathrm{SENTFORM}_3(a,b,c)$ is a theorem. The proof begins by resolving on the negation of the theorem and a clause from DS3. The resolvent is then resolved with the clause which represents the production $bC \to bc$ to give the clause which represents the sentential form abC. This two-step deduction of $\sim\mathrm{SENTFORM}_3(a,b,C)$ from $\sim\mathrm{SENTFORM}_3(a,b,c)$ is an example of a cycle. It represents the step $abC \Rightarrow abc$ in the derivation sequence of abc. Each of the three cycles in Figure 4.1 transforms one clause of the form $\sim\mathrm{SENTFORM}_k(a_1,\ldots,a_k)$ into another clause of the same form, and represents one step in the derivation sequence. The proof shown corresponds to the bottom-up analysis of the derivation sequence

$$S \Rightarrow aBC \Rightarrow abC \Rightarrow abc$$

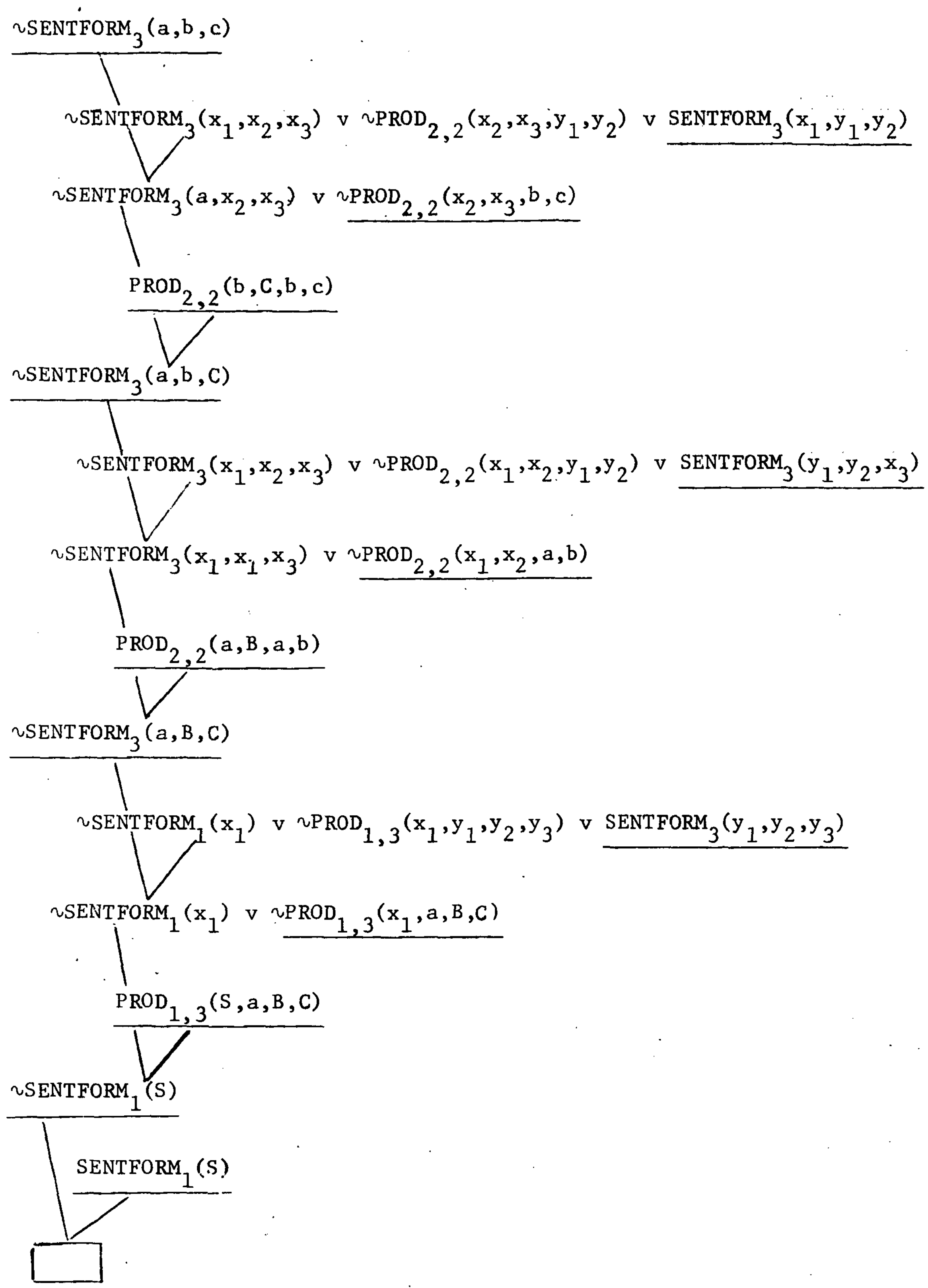

$\sim$SENTFORM$_3$(a,b,c)
$\sim$SENTFORM$_3$(x_1,x_2,x_3) v $\sim$PROD$_{2,2}$(x_2,x_3,y_1,y_2) v SENTFORM$_3$(x_1,y_1,y_2)
$\sim$SENTFORM$_3$(a,x_2,x_3) v $\sim$PROD$_{2,2}$(x_2,x_3,b,c)
PROD$_{2,2}$(b,C,b,c)
$\sim$SENTFORM$_3$(a,b,C)
$\sim$SENTFORM$_3$(x_1,x_2,x_3) v $\sim$PROD$_{2,2}$(x_1,x_2,y_1,y_2) v SENTFORM$_3$(y_1,y_2,x_3)
$\sim$SENTFORM$_3$(x_1,x_1,x_3) v $\sim$PROD$_{2,2}$(x_1,x_2,a,b)
PROD$_{2,2}$(a,B,a,b)
$\sim$SENTFORM$_3$(a,B,C)
$\sim$SENTFORM$_1$(x_1) v $\sim$PROD$_{1,3}$(x_1,y_1,y_2,y_3) v SENTFORM$_3$(y_1,y_2,y_3)
$\sim$SENTFORM$_1$(x_1) v $\sim$PROD$_{1,3}$(x_1,a,B,C)
PROD$_{1,3}$(S,a,B,C)
$\sim$SENTFORM$_1$(S)
SENTFORM$_1$(S)

FIGURE 4.1 A proof that abc ε L(G_1).

The derivation sequence representation can also perform a top-down analysis of a string. This is accomplished by giving sypport to the clause $SENTFORM_1(S)$ instead of the clause $\sim SENTFORM_3(a,b,c)$, as was done in the proof of Figure 4.1. In the proofs that correspond to a top-down analysis, cycles are two-step deductions of a clause of the form $SENTFORM_k(y_1,\ldots,y_k)$ to another clause of the same form. The negation of the theorem is not used until the next to the last deduction.

The derivation sequence representation models the derivation sequence of a string of symbols, and hence one could prove a theorem $SENTFORMn(a_1, \ldots,a_n)$ where some a_i are nonterminal symbols. A modification to the representation which eliminates this possibility can easily be obtained by the addition of axioms to indicate which symbols are terminals, and axioms which differentiate between those sentential forms which contain terminals from those which do not.

4.2 Recognition in the Derivation Tree Representation

To show that the string $a_1\ldots a_n$ is in the language L(G) we prove the theorem $INPUT(a_1,1) \wedge\ldots\wedge INPUT(a_n,n) \rightarrow DERIVE(S,1,n+1)$, where S is the start symbol of G. The interpretation of the theorem is that if a_i is the i^{th} input symbol for $1 \leq i \leq n$, then S derives the subset of the input string beginning at the 1^{st} position and ending at the n^{th} position, that is, S derives the entire input string. To prove this theorem, the negation of the above formula in clause form,

$$INPUT(a_1,1)$$

$$\vdots$$

$$INPUT(a_n,n)$$

$$\sim DERIVE(S,1,n+1)$$

is added to the set of clauses which is the derivation tree representation for the grammar G , and the resulting set of clauses is then shown to be unsatisfiable.

A proof that the sentence 'they are flying planes' is in the language generated by the grammar G_2 is shown in Figure 4.2. There are two types of cycles in the proof. Those similar to the one labeled (A) in the figure have two resolutions and use the axioms DT2 along with the production axioms of the form $PROD_{1,1}(a,b)$. These cycles determine the syntactic categories of the terminal symbols in the input string. The second type of cycle, labeled (B) in the figure, has three resolutions and uses the axiom DT3 along with the production axioms of the form $PROD_{1,2}(a,b,c)$. These cycles deduce that nonterminals derive certain substrings of the input string from previously known facts of this type and from the productions. The single dotted lines in the figure indicate cycles similar to the first type, the pairs of dotted lines indicate cycles of the second type. The clause DERIVE(SENT,1,5) is deduced and resolves with its negation (which comes from the negation of the theorem) to give a refutation.

4.3 The Cycle Inference System

The proofs that were used in the previous sections to illustrate how the two representations can determine that a string is in a language were the simplest possible proofs. If the inference system that was used allowed resolutions between any two clauses many unnecessary clauses would be generated.

Consider first the derivation sequence representation. The clauses in the set DS3 (see Section 3.1) are used to model one step in a derivation sequence. It is possible for two clauses in DS3 to resolve with each other to produce a clause which models two steps in a derivation sequence. This

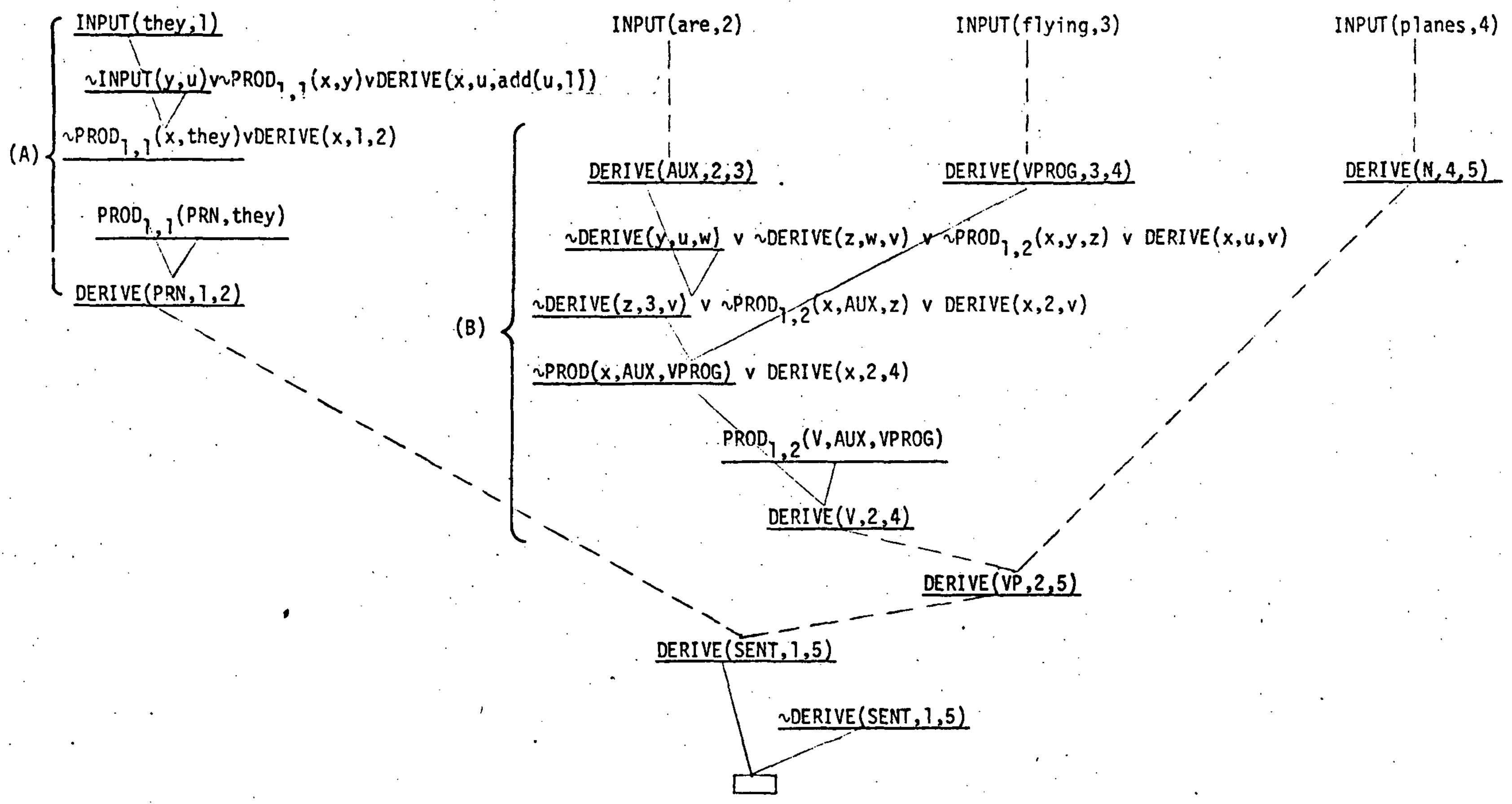
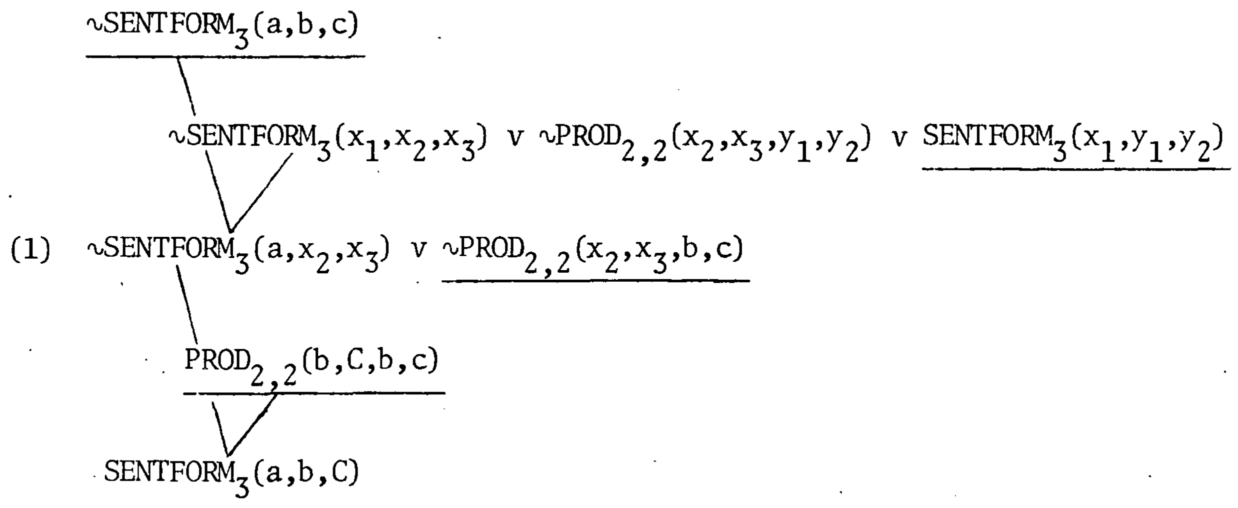

INPUT(they,1)
∼INPUT(y,u)v∼PROD1,1(x,y)vDERIVE(x,u,add(u,1))
∼PROD1,1(x,they)vDERIVE(x,1,2)
PROD1,1(PRN,they)
DERIVE(PRN,1,2)
(A)
INPUT(are,2)
INPUT(flying,3)
INPUT(planes,4)
DERIVE(AUX,2,3)
DERIVE(VPROG,3,4)
DERIVE(N,4,5)
∼DERIVE(y,u,w) v ∼DERIVE(z,w,v) v ∼PROD1,2(x,y,z) v DERIVE(x,u,v)
∼DERIVE(z,3,v) v ∼PROD1,2(x,AUX,z) v DERIVE(x,2,v)
∼PROD(x,AUX,VPROG) v DERIVE(x,2,4)
PROD1,2(V,AUX,VPROG)
DERIVE(V,2,4)
(B)
DERIVE(VP,2,5)
DERIVE(SENT,1,5)
∼DERIVE(SENT,1,5)

FIGURE 4.2 A proof that 'they are flying planes' is in $L(G_2)$. The single dotted lines indicate cycles similar to (A), the pairs of dotted lines indicate cycles similar to (B).

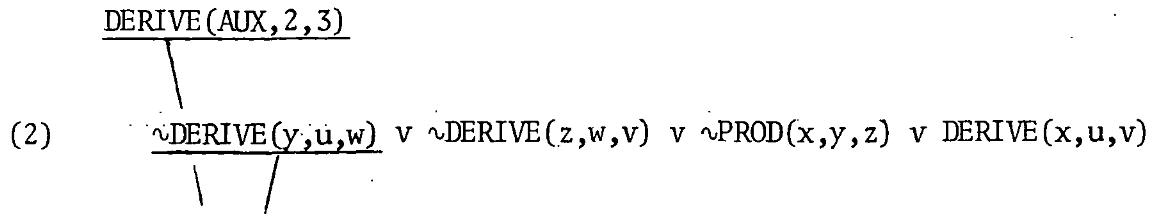
action is undesirable because in most cases the two steps of the derivation sequence which the above resolvent modeled will not be used in the proof that a string is in a language. It can be eliminated by employing an inference system which uses set-of-support with the negation of the theorem given support, since such an inference system does not allow axioms to resolve with each other. However, set-of-support does allow resolutions in the derivation sequence representation which are undesirable. Consider an example of a cycle which is taken from the proof given in Figure 4.1.



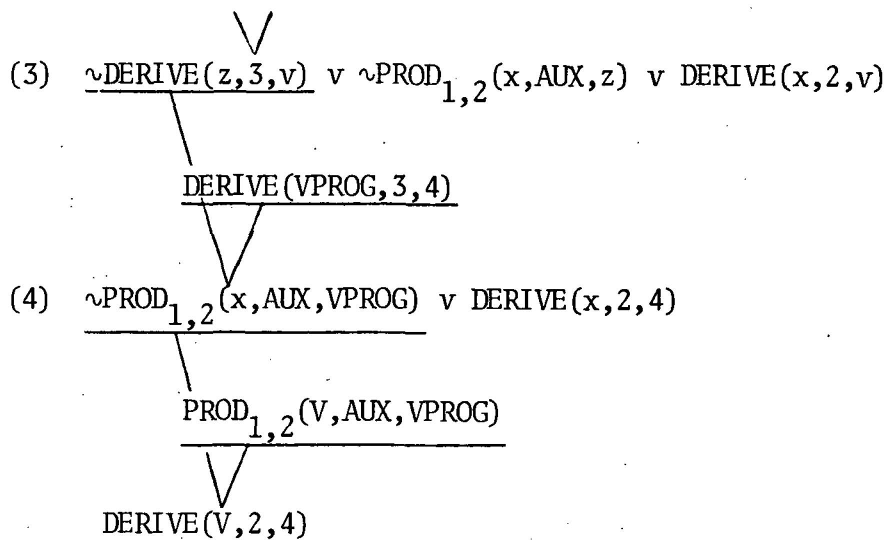
$\sim SENTFORM_3(a,b,c)$
$\sim SENTFORM_3(x_1,x_2,x_3) \vee \sim PROD_{2,2}(x_2,x_3,y_1,y_2) \vee SENTFORM_3(x_1,y_1,y_2)$
(1) $\sim SENTFORM_3(a,x_2,x_3) \vee \sim PROD_{2,2}(x_2,x_3,b,c)$
$PROD_{2,2}(b,C,b,c)$
$SENTFORM_3(a,b,C)$

The clause labeled (1) has support since it is a descendant of a clause with support, and the first literal of (1) will unify with many literals in clauses which are in DS3. These resolvents would be unnecessary, but set-of-support would allow them.

Similar things happen in the derivation tree representation. Consider the cycle taken from Figure 4.2.



$DERIVE(AUX,2,3)$
(2) $\sim DERIVE(y,u,w) \vee \sim DERIVE(z,w,v) \vee \sim PROD(x,y,z) \vee DERIVE(x,u,v)$


(3) ∿DERIVE(z,3,v) v ∿PROD1,2(x,AUX,z) v DERIVE(x,2,v)
DERIVE(VPROG,3,4)
(4) ∿PROD1,2(x,AUX,VPROG) v DERIVE(x,2,4)
PROD1,2(V,AUX,VPROG)
DERIVE(V,2,4)

Clause (2) is the axiom that comes from the singleton set called DT3 in the definition of the representation, and it can resolve with itself. Again set-of-support would rule this out, but it would not rule out clauses (3) or (4) resolving with a clause from DT3.

The cycle inference is a rule which restricts the proof procedure to resolutions which perform cycles. We first give a general definition of the cycle inference and then define its application to the two representations.

DEFINITION:

An inference system is a cycle inference system iff there exists $n + 1$ not necessarily distinct sets of clauses, A , $B_1,\dots,B_n$, and an n-tuple $N = (i_1,\dots,i_n)$, where $i_j \in \{0,1\}$, such that one resolvent of each resolution is a clause in A or a descendant of a clause in A , while the other resolvent cycles among the sets $B_1,\dots,B_n$. If $i_j = 0$ the jth resolution of the cycle must use a clause in B_j ; if $i_j = 1$, the jth resolution of the cycle may use a clause in B_j or a descendant of a clause in B_j . The value n is called the length of the cycle.

The set A is similar to the set of clauses which are given support in the set-of-support inference system. The set A is like the support set in that every resolvent has a parent either from A itself, or a descendant of a clause in A. It is different from the support set because the cycle inference system demands that exactly one parent be either from A or a descendant of a clause in A, while the set-of-support inference system specifies that at least one parent (allowing for the possibility of both parents) be either from the support set or a descendant of a clause from the support set.

The sets $B_1,\ldots,B_n$ can almost be thought of as sets of side clauses. The set from which the side clauses must be chosen cycles among the B_i. If the n-tuple is all zeroes, then all the side clauses are input clauses. Since each resolvent has one parent which is an input clause, the proof is an input proof, (Chang, 1970). If the jth element of N is 1, then the jth side clause can be a descendant of a clause in B_j, which makes the proof no longer an input proof. As will be seen, the former is the case in the derivation sequence representation and the latter is the case in the derivation tree representation.

To apply the cycle inference to the derivation sequence representation take (see Section 3.1 for notation):

$$n = 2$$

$$A = \{\sim\mathrm{SENTFORM}_n(a_1,\ldots,a_n)\}$$

$$B_1 = \mathrm{DS2} \cup \mathrm{DS3}$$

$$B_2 = \mathrm{DS1}$$

$$N = (0,0)$$

This application of the cycle inference results in a bottom-up analysis similar to the proof shown in Figure 4.1. The inclusion of the set DS2, which is the singleton $\mathrm{SENTFORM}_1(S)$, in B_1 allows for the unit resolution which produces the null clauses. Setting A equal to DS2 and including $\sim\mathrm{SENTFORM}_n(a_1,\ldots,a_n)$ in B_1 gives a top-down analysis.

The proofs in the derivation tree representation contain cycles of length two which convert the terminal symbols of the input string into syntactic categories, and cycles of length three which take two clauses which represent the fact that a nonterminal derives a substring of the input string and produces a third clause of the same type (see Figure 4.2). Since there are cycles of varying lengths in these proofs, one could not expect to define a cycle inference system that could be used uniformly in the proofs. It is possible to make a small change in the representation to allow for the uniform application of a cycle inference system. The change makes the set DT2:

$$(\forall y,u)\ [\mathrm{INPUT}(y,u) \wedge \mathrm{INPUT}(\alpha,0) \wedge \mathrm{PROD}_{1,1}(x,y) \rightarrow \mathrm{DERIVE}(x,u,\mathrm{add}(u,1))],$$

and makes the theorem:

$$\mathrm{INPUT}(\alpha,0) \wedge \mathrm{INPUT}(a_1,1) \wedge\ldots\wedge \mathrm{INPUT}(a_n,n) \rightarrow \mathrm{DERIVE}(S,1,n+1),$$

where α is a constant not previously used in the representation. A cycle inference can now be applied during the entire proof by setting (see Section 3.2 for notation):

$$n = 3$$

$$A = \{\mathrm{INPUT}(y,u) \mid y \in V_T \cup \{\alpha\},\ 0 \le u \le n\}$$

$$B_1 = \mathrm{DT2} \cup \mathrm{DT3} \cup \{\sim\mathrm{DERIVE}(S,1,n+1)\}$$

$$B_2 = A$$

$$B_3 = \mathrm{DT1}$$

$$N = (0,1,0)$$

The first step of a cycle uses either the singleton in DT2 or DT3, the second step uses a clause in A or a descendant of a clause in A (these clauses will have the form DERIVE(x,u,v)), and the third step uses a production clause. The fact that the second step uses a clause which is a descendant of a base clause makes the proofs not an input proof. The clause ∼DERIVE(S,1,n+1) is included in B_1 so that the single resolvent which generates the empty clause can be made.

Since using the cycle inference system on either of the representations will result in a refutation whenever the input string is a sentence in the language, we say that the cycle inference system is complete for these representations. Conversely, since whenever the cycle inference system produces a refutation from either of the representations, the input string is a sentence in the language, we say that the cycle inference system is sound for these representations.

The use of the cycle inference system results in the problem-solver modeling a state-space representation of a problem. The basis for the derivation sequence representation is the state-space representation which specifies a sentential form as a state description with the start symbol the initial state description; the set of operators as the set of productions with the transformation on a state description a legal application of a production to a sentential form; and the goal state description as the input string. The use of the cycle inference system guarantees that the only deductions which the theorem prover makes will transform sentential form clauses into sentential form clauses. That is, the theorem prover is restricted to deductions which model the application of an operator.

In the derivation tree representation a state description is a statement that a nonterminal derives a substring of the input string and an operator is again a production. However, the transformation on the state descriptors maps two states into a single state. Again, the use of the cycle inference system results in the problem-solver modeling the search of the state-space of the problem. The general applicability of the cycle inference system will be discussed more completely in the conclusions.

4.4 The Augmented Theorem Prover

The examples given in the previous sections show how a theorem prover which employs a cycle inference system can determine that a string is a sentence in a language by proving that the statement which represents the string is a theorem of the representation. In order to represent the problem of language recognition the problem-solver must be able to determine that a string is not a sentence in the language as well as that it is a sentence in the language. The problem-solver (i.e., the theorem prover) can be given this capability for the two representations by augmenting it so that it can terminate either by achieving a refutation or by reaching a stage where no new clauses can be generated (Kowalski, 1970). In our application, stages occur at the end of cycles. We call a theorem prover with the above capability an augmented theorem prover. An augmented theorem prover, using the cycle inference system, can determine that a string is not a sentence using either representation because they both have finite state-spaces. A theorem prover using the cycle inference system can only generate finitely many clauses from the set of clauses consisting of either of the representations union the negation of the theorem. If one of these clauses is the empty clause, the string is a sentence, otherwise the string is not a sentence. We now discuss a space bound and a time bound for each representation.

Consider first the problem of deciding whether or not $a_1 \ldots a_n$ is in a recursive language L(G) using the derivation sequence representation. The number of clauses in the representation itself is finite because the cardinality of each of the sets of clauses (see Section 3.1 for notation) is:

$$|DS1| = p$$

$$|DS2| = 1$$

$$|DS3| \approx pn^2$$

where p is the number of productions in G. The cycle inference system results only in cycles which produce clauses which represent sentential forms. Since L(G) is recursive, the maximum number of these clauses is proportional to g^n where g is the number of symbols (terminal and nonterminal) in G (Hopcroft and Ullman, 1969). Since the number of clauses that can be generated using the cycle inference system is finite, an augmented theorem prover can determine that a string is not in a language using the derivation sequence representation. Like the space bound, the time bound for this representation is also exponential.

In the derivation tree representation the cycle inference system restricts the theorem prover to generating a clause which is a statement that a nonterminal derives a substring of the input string (i.e., a clause of the form DERIVE(x,u,v) from two other clauses of this form). The number of choices for x in this clause is h, where h is the number of nonterminals in G, while there are at most n choices each for u and v. This results in at most hn^2 clauses of this form. Hence the number of clauses that can be generated from this representation is also finite, and an augmented theorem prover can be used to determine that a string is not a sentence in a language.

To obtain a time bound for recognition in this representation we compute the number of cycles required to deduce the arbitrary but fixed clause DERIVE(A,u_0,v_0) . We assume that the "DERIVE clauses" for smaller substrings have already been deduced. To derive the clause DERIVE(A,u_0,v_0) the cycle inference system will attempt to generate cycles of the following form.

DERIVE(B,u_0,w_0)

$\sim$DERIVE(y,u,w) v $\sim$DERIVE(z,w,v) v $\sim$PROD$_{1,2}(x,y,z)$ v DERIVE(x,u,v)

$\sim$DERIVE(z,w_0,v) v $\sim$PROD$_{1,2}(x,B,z)$ v DERIVE(x,u_0,v)

DERIVE(C,w_0,v_0)

$\sim$PROD$_{1,2}(x,B,C)$ v DERIVE(x,u_0,v_0)

PROD$_{1,2}(A,B,C)$

DERIVE(A,u_0,v_0)

The number of possibilities for B and C depend on the grammar, and there are at most n possibilities for w . Hence the number of cycles required to deduce DERIVE(A,u_0,v_0) is $\sim n$. Since the number of clauses of this form is $\sim n^2$, $\sim n^3$ cycles are necessary to perform recognition.

In either of the representations, all of the solutions to the problem can be obtained by changing the structure of the theorem prover so that it does not stop the first time that the null clause is generated. The structure would be such that the theorem prover would terminate only when no new clauses can be generated. Each distinct derivation of the null clause represents a distinct solution to the problem. It goes without saying that solutions in the derivation sequence representation are derivation sequences, and solutions in the derivation tree representation are derivation trees.

4.5 The Existence of Input Proofs in the Two Representations

Theorem proving programs which restrict themselves to input proofs are particularly simple to implement, and the search for input proofs is very efficient. Thus it is important to know what types of unsatisfiable sets of clause have input proofs. In this section we show the existence of input proofs for a class of unsatisfiable sets of clauses which includes the two representations of the language recognition problem.

The clauses in the two representations can be divided into three categories as follows:

	Derivation Sequence Representation	Derivation Tree Representation
unit positive clauses	DS1, DS2	DT1, INPUT clauses coming from ∿theorem
clauses with exactly one positive literal and a nonzero number of negative literals	DS3	DT2, DT3
negative clauses	∿theorem	DERIVE clause coming from ∿theorem

The clauses in the sets DS3, DT2, and DT3 are derived from a wff of the form $(Q_1x_1)(Q_2x_2)\dots(Q_nx_n)[P_1(\ ) \wedge P_2(\ ) \wedge\dots\wedge P_n(\ ) \rightarrow P_{n+1}(\ )]$ where Q_i , $i = 1,\dots,n$ is either $\forall$ or $\exists$. All of the clauses in the two representations have at most one positive literal. Sets of clauses having this property have been called implications sets (Slagle and Koniver, 1971). The following theorem proves the existence of inputs proofs for this class of unsatisfiable sets of clauses. The proof of the theorem uses the concept of *linear resolutions and the fact that linear resolution is complete*, (Loveland, 1970; Luckham, 1970). A proof $(T = C_0, C_1, \ldots, C_n, \boxed{})$ is in linear form with top clause T if each C_k is the resolvent of C_{k-1} and either an input clause or some C_j where $j < k - 1$.

Theorem:

If S is a satisfiable set of clauses such that each clause in S has at most one positive literal, T is a negative clause, and the set $\{S \cup T\}$ is unsatisfiable, then there exists an input proof for $\{S \cup T\}$.

Proof:

Since linear resolution is complete, there exists a derivation of the null clause in linear form with T as the top clause. Let $(T, C_1, \ldots, C_n, \boxed{})$ be a linear derivation of the null clause. We show by induction that all of the clauses in the linear derivation must have no positive literals, and hence that all of the side clauses must be input clauses. The top clause, T has no positive literals by hypothesis. The clause C_1 must have no positive literals since it is resolvent with the negative clause T and an input clause that has at most one positive literal. Assume that the first $k > 1$ clauses in the linear derivation are negative. Since the clauses $(T, C_1, \ldots, C_{k-1})$ are all negative, none of the clauses $(T, C_1, \ldots, C_{k-2})$ can be used as side clauses to resolve with C_{k-1} to yield C_k. Thus an input clause must be used. But, since the input clause has at most one positive literal, the clause C_k is a negative clause. Hence, all of the clauses in $(T,C_1,\ldots,C_n)$ are negative, and the proof is an input proof. This completes the proof.

We have seen that the proofs which the cycle inference system finds in the derivation sequence representation are input proofs, while those found in the derivation tree representation are not. Recall that the n-tuple used to define a cycle inference system for the derivation sequence representation was (0,0), and that (0,1,0) was used in the definition for the derivation tree representation. The cycle inference system finds input proofs iff the n-tuple $(i_1,\ldots,i_n)$ is all zeroes, because then and only then will each resolution of a cycle use a clause in set A or a descendant of a clause in A , and a clause in B_j for some j .

Input proofs in the derivation tree representation use $\sim$DERIVE(S,1,n+1) as the top clause, and correspond to a sequential representation of the derivation tree. Whereas the proofs found by the cycle inference system model the bottom-up construction of the span triangle (i.e., the Cocke algorithm), the input proofs model the top-down generation of the span triangle. One of the input proofs which corresponds to the proof shown in Figure 4.2 is shown in Figure 4.3.

∿DERIVE(SENT,1,5)

  ∿DERIVE(y,1,w) v ∿DERIVE(z,w,5) v ∿$\mathrm{PROD}_{1,2}$(SENT,y,z)

∿DERIVE(PRN,1,w) v ∿DERIVE(VP,w,5)

  ∿INPUT(y,1) v ∿$\mathrm{PROD}_{1,1}$(PRN,y) v ∿DERIVE(VP,2,5)

∿INPUT(they,1) v ∿DERIVE(VP,2,5)

∿DERIVE(VP,2,5)

  ∿DERIVE(y,2,w) v ∿DERIVE(z,w,5) v ∿$\mathrm{PROD}_{1,1}$(VP,y,z)

∿DERIVE(V,2,w) v ∿DERIVE(N,w,5)

  ∿DERIVE(V,2,4) v ∿INPUT(u,4) v ∿$\mathrm{PROD}_{1,1}$(N,y)

∿DERIVE(V,2,4) v INPUT(planes,4)

∿DERIVE(V,2,4)

  ∿DERIVE(y,2,w) v ∿DERIVE(z,w,4) v ∿$\mathrm{PROD}_{1,2}$(V,y,z)

∿DERIVE(AUX,2,w) v ∿DERIVE(VPROG,w,4)

  ∿INPUT(y,2) v ∿$\mathrm{PROD}_{1,2}$(AUX,y) v ∿DERIVE(VPROG,3,4)

∿INPUT(are,2) v ∿DERIVE(VPROG,3,4)

∿DERIVE(VPROG,3,4)

  ∿INPUT(y,3) v ∿$\mathrm{PROD}_{1,2}$(VPROG,y)

∿INPUT(y,3)

$\square$

Figure 4.3 An input proof in linear form in the derivation tree representation. The clauses which are not indented indicate how input proofs in this representation model the top-down generation of the span triangle.

5. Extracting Derivation Sequences and Derivation Trees

Obtaining the answer to a theorem proving problem involves converting the proof from a graph with the empty clause at the root to one with a clause at the root which represents the answer. The process is performed after the proof is completed, and is accomplished by converting each clause arising from the negation of the theorem into a tautology (Luckham and Nilsson, 1971). These added literals filter through the proof and become the representation of the answer.

In the derivation sequence representation the literal $\mathrm{SENTFORM}_n(a_1, \ldots, a_n)$ is added to the negation of the theorem and becomes the answer. Converting the clauses in the negation of the theorem for the derivation tree representation results in the answer:

$$\sim\mathrm{INPUT}(a_1,1) \text{ v}\ldots\text{v } \sim\mathrm{INPUT}(a_n,n) \text{ v } \mathrm{DERIVE}(S,1,n+1) \;.$$

The interpretation of the above clause is that if the input string is $a_1 \ldots a_n$ then the start symbol derives the entire input string. Thus in both of the representations the answer extraction process yields a yes/no type answer.

However, the answer in both of the representations is really more than yes or no. As the names imply, the answer in the derivation sequence representation is a derivation sequence of the input string, and in the derivation tree representation is a derivation tree of the input string. These more complete answers are presently embedded in the proofs. The concepts of states and state-transformation functions (Green, 1969) can be used to build-up and extract these more complete answers to the problem. In the next paragraphs we describe modifications to the representations which incorporate the state-transformation method.

In one formulation of the method, actions are represented as constants and the state-transformation function maps actions and states into states. The sequence of actions that represents the solution to the problem are built-up by composing the state transformation-function.

In the derivation sequence representation an action involves the application of a production to a sentential form, hence each production must be assigned a unique constant. We will include an additional term in each of the production axioms to store this constant. In the example of the modified derivation sequence representation for the grammar G_1, which is shown in Figure 5.1, the constants are circled numbers. The sentential form literal will now include a term to represent the state of the recognition process. The start state is represented as $\text{SENTFORM}_1(S,\text{nil})$ since no actions in the derivation sequence have taken place when the sentential form is the start symbol. The axioms in the set DS3 are modified to include a state-transformation mapping. An action is really more than the application of a production to a sentential form, it is the application of a production to a particular position in a sentential form. Sentential forms can contain duplicate nonterminal symbols, so unless the state-transformation recorded the position in the sentential form where the production was applied as well as the production which was applied, it would not be remembering the state of the recognition process.

The modified representation uses a 3-place state-transformation function called 'apply.' The first argument is the name of the production applied, the second the initial position of its application, and the third the state to which the action is applied. Figure 5.1 gives the modified representation for the grammar G_1 , and Figure 5.2 illustrates its use by showing a cycle of a top-down proof in the representation. The modified

$$DS1' = \left\{ \begin{array}{l} PROD_{1,4}(S,a,S,B,C,\textcircled{1}) \\ PROD_{1,3}(S,a,B,C,\textcircled{2}) \\ PROD_{2,2}(C,B,B,C,\textcircled{3}) \\ PROD_{2,2}(a,B,a,b,\textcircled{4}) \\ PROD_{2,2}(b,B,b,b,\textcircled{5}) \\ PROD_{2,2}(b,C,b,c,\textcircled{6}) \\ PROD_{2,2}(c,C,c,c,\textcircled{7}) \end{array} \right\}$$

$$DST' = [SENTFORM_1(S,nil)]$$

$$DS3' = \{(\forall x_1,\ldots,x_m,y_1,\ldots,y_k,s,t)[SENTFORM_m(x_1,\ldots,x_m,t)$$

$$\wedge\ PROD_{j,k}(x_{i+1},\ldots,x_{i+j},y_1,\ldots,y_k,s) \rightarrow SENTFORM_{m+k-j}(x_1,\ldots,x_i,$$

$$y_1,\ldots,y_k,x_{i+j+1},\ldots,x_m,\ \text{apply}\ (s,i+1,t))]\ \mid\ (j,k)\ \text{is}\ (1,4),\ (1,3),\ \text{or}\ (2,2)\}$$

Figure 5.1 The derivation sequence representation for $L(G_1)$ modified to include state-transformations.

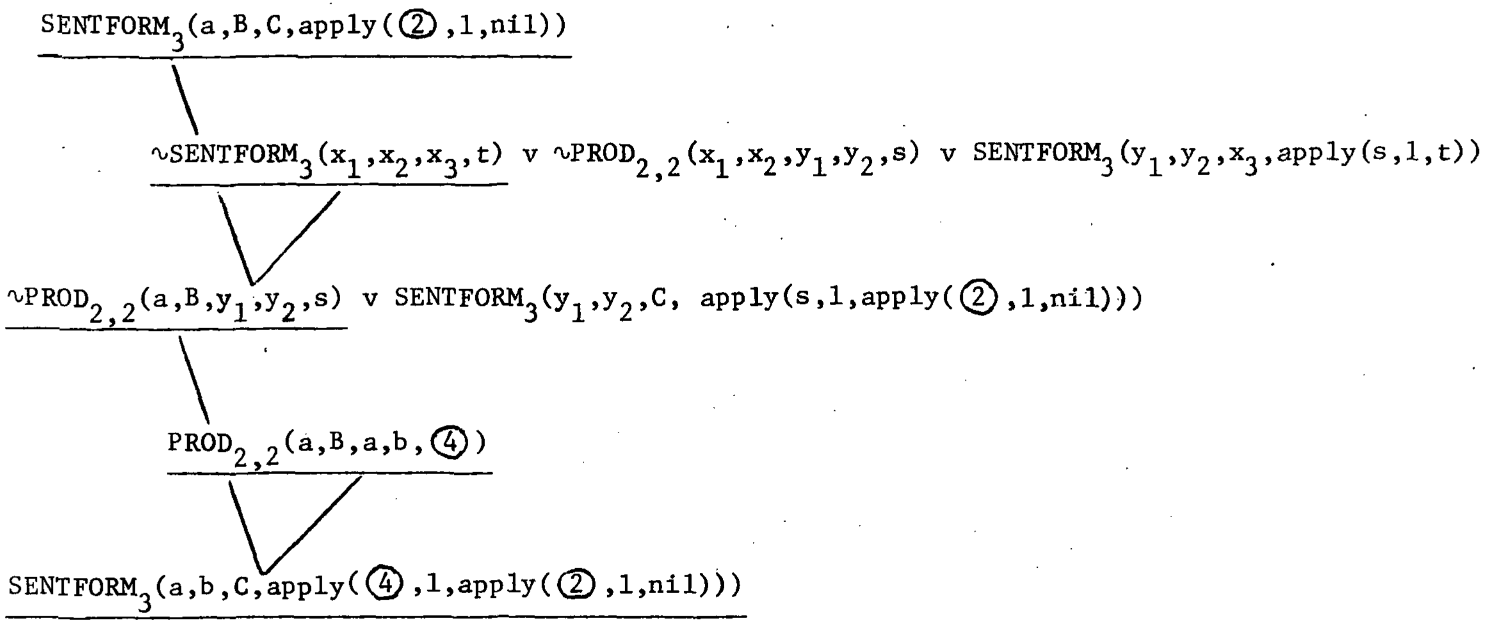

$\mathrm{SENTFORM}_3(a,B,C,\mathrm{apply}(②,1,\mathrm{nil}))$
$\sim\mathrm{SENTFORM}_3(x_1,x_2,x_3,t) \vee \sim\mathrm{PROD}_{2,2}(x_1,x_2,y_1,y_2,s) \vee \mathrm{SENTFORM}_3(y_1,y_2,x_3,\mathrm{apply}(s,1,t))$
$\sim\mathrm{PROD}_{2,2}(a,B,y_1,y_2,s) \vee \mathrm{SENTFORM}_3(y_1,y_2,C,\ \mathrm{apply}(s,1,\mathrm{apply}(②,1,\mathrm{nil})))$
$\mathrm{PROD}_{2,2}(a,B,a,b,④)$
$\mathrm{SENTFORM}_3(a,b,C,\mathrm{apply}(④,1,\mathrm{apply}(②,1,\mathrm{nil})))$

Figure 5.2 A cycle in a proof in the derivation sequence representation modified to include state-transformations.

representation also works for bottom-up proofs. In both cases the first production used in a top-down analysis (generation) is innermost in the 'apply' function, and the first production used in a bottom-up analysis (reduction) is outermost. The answer extraction process can be used to get at the final state.

An action in the derivation tree representation is also the application of a production, but unlike the derivation sequence representation the action in the derivation tree representation works on a pair of states. Hence the state-transformation function must map a pair of states and an action into a state. We call the function 'reduce.' Figure 5.3 gives the modified derivation tree representation for the grammar G_2. The 3-step cycle which is explicitly shown in the proof of Figure 4.2 appears for the modified representation in Figure 5.4. Figure 5.5 gives a deduction in the representation (explicitly showing only INPUT and DERIVE clauses). The order of the names of the productions in the final state is the end-order traversal (Knuth, 1968) of the derivation tree. The representation is capable of finding all of the derivation trees of a sentence.

$$
\mathrm{DT1'} = \left\{ \begin{array}{l}
\mathrm{PROD}_{1,2}(\mathrm{SENT},\mathrm{PRN},\mathrm{VP},\textcircled{1}) \\
\mathrm{PROD}_{1,2}(\mathrm{VP},\mathrm{V},\mathrm{N},\textcircled{2}) \\
\mathrm{PROD}_{1,2}(\mathrm{VP},\mathrm{COP},\mathrm{NP},\textcircled{3}) \\
\mathrm{PROD}_{1,2}(\mathrm{SENT},\mathrm{PRN},\mathrm{V},\textcircled{4}) \\
\mathrm{PROD}_{1,2}(\mathrm{V},\mathrm{AUX},\mathrm{VPROG},\textcircled{5}) \\
\mathrm{PROD}_{1,2}(\mathrm{NP},\mathrm{ADJ},\mathrm{N},\textcircled{6}) \\
\mathrm{PROD}_{1,1}(\mathrm{PRN},\mathrm{they},\textcircled{7}) \\
\mathrm{PROD}_{1,1}(\mathrm{COP},\mathrm{are},\textcircled{8}) \\
\mathrm{PROD}_{1,1}(\mathrm{AUX},\mathrm{are},\textcircled{9}) \\
\mathrm{PROD}_{1,1}(\mathrm{ADJ},\mathrm{flying},\textcircled{10}) \\
\mathrm{PROD}_{1,1}(\mathrm{VPROG},\mathrm{flying},\textcircled{11}) \\
\mathrm{PROD}_{1,1}(\mathrm{N},\mathrm{planes},\textcircled{12})
\end{array} \right\}
$$

$$
\mathrm{DT2'} = \{(\forall y,u,s)[\mathrm{INPUT}(y,u) \wedge \mathrm{PROD}_{1,1}(x,y,s) \rightarrow \mathrm{DERIVE}(x,u,\mathrm{add}(u,1),s)]\}
$$

$$
\begin{aligned}
\mathrm{DT3'} = \{(\forall x,y,z,u,v,w,s,t,r)[&\mathrm{DERIVE}(y,u,w,s) \wedge \mathrm{DERIVE}(a,w,v,t) \\
&\wedge \mathrm{PROD}_{1,2}(x,y,z,r) \rightarrow \mathrm{DERIVE}(x,u,v,\mathrm{reduce}(s,t,r))]\}
\end{aligned}
$$

Figure 5.3 The derivation tree representation for G_2 modified to include state-transformations.

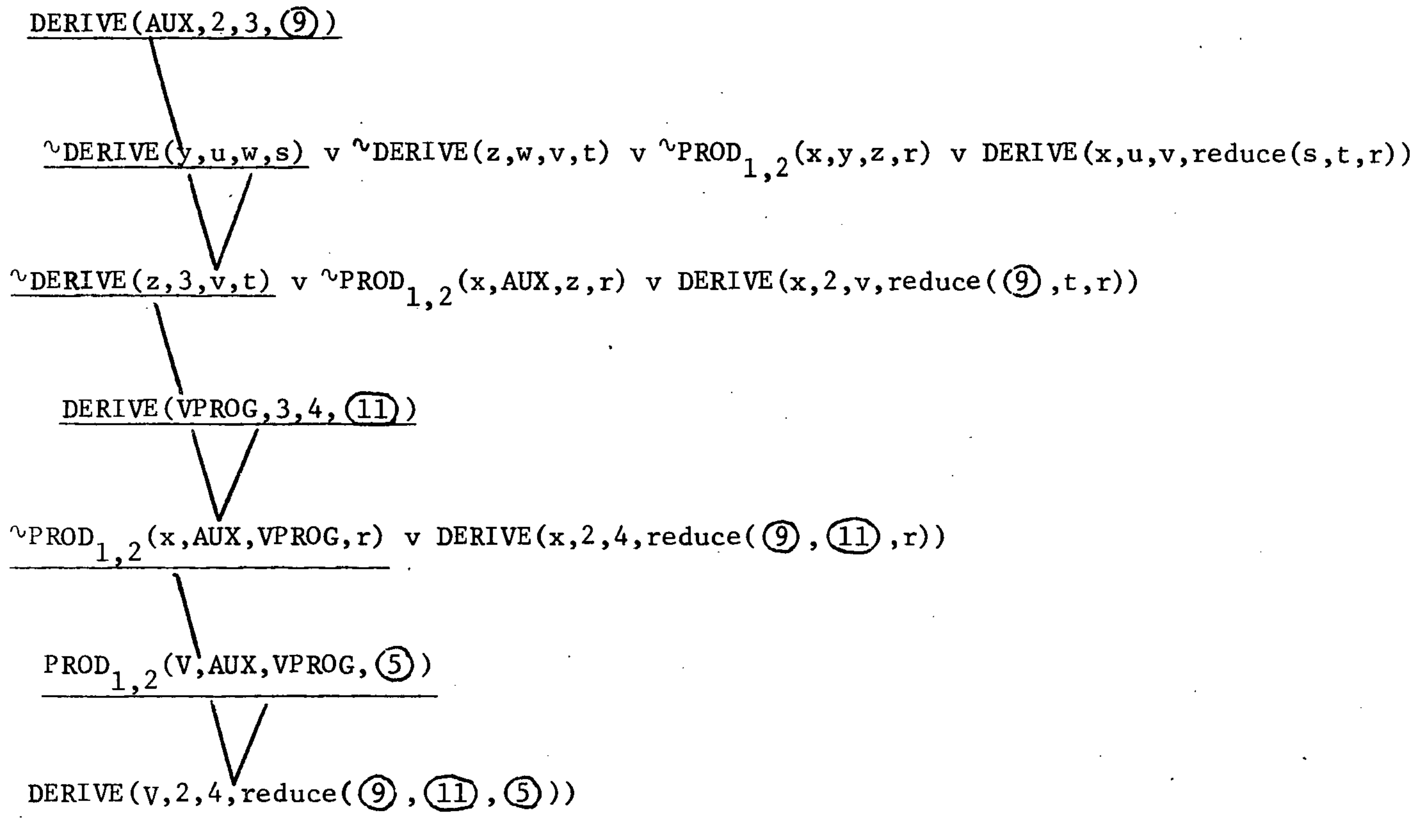

DERIVE(AUX,2,3,⑨)
∿DERIVE(y,u,w,s) v ∿DERIVE(z,w,v,t) v ∿PROD1,2(x,y,z,r) v DERIVE(x,u,v,reduce(s,t,r))
∿DERIVE(z,3,v,t) v ∿PROD1,2(x,AUX,z,r) v DERIVE(x,2,v,reduce(⑨,t,r))
DERIVE(VPROG,3,4,⑪)
∿PROD1,2(x,AUX,VPROG,r) v DERIVE(x,2,4,reduce(⑨,⑪,r))
PROD1,2(V,AUX,VPROG,⑤)
DERIVE(V,2,4,reduce(⑨,⑪,⑤))

Figure 5.4 A cycle in the derivation tree representation modified to include state-transformations.

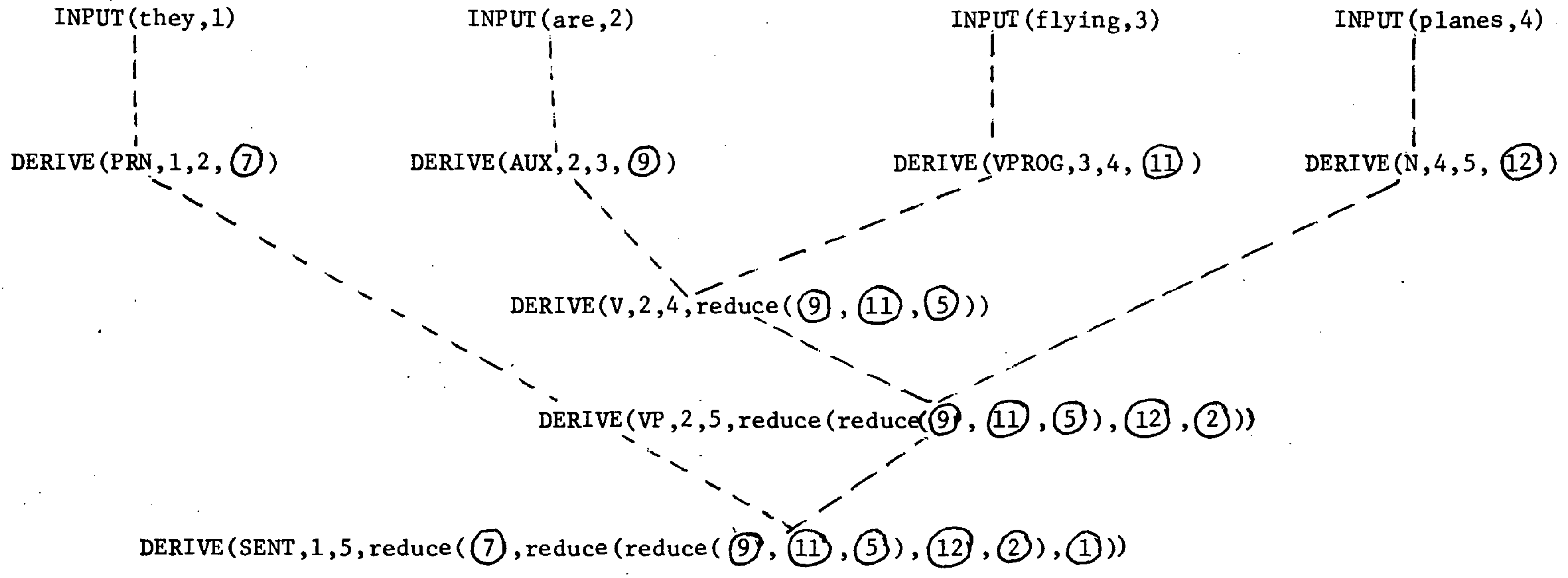


FIGURE 5.5 A deduction (explicitly showing only INPUT and DERIVE clauses) in the derivation tree representation which is modified to include state-transformations.

6. Conclusions

We have developed two representations in the first-order predicate calculus of the language recognition problem. Both of these representations are modeled after known algorithms for recognizing strings: the algorithm of generating sentential forms whose lengths do not exceed the length of the string, and the Cocke algorithm. Both algorithms can be thought of as state-space representations of the language recognition problem.

In the former algorithm, the states are sentential forms, with the start symbol the initial state, and the string the goal state. The set of operators is the set of productions, and the transformation of a state to a state is the legal application of a production to a sentential form. In this representation, the operators are single-input/single-output operators.

In the formulation of the Cocke algorithm as a state-space problem the states are either statements that a terminal symbol appears in a certain position of the string, or statements that a nonterminal symbol derives a substring. Instead of a single initial state, there is a set of initial states, the ones which collectively represent the string. The final state is the statement that the start symbol derives the entire string. A solution in this representation is not a path from one of the inital states to the final state, but a set of paths from all of the initial states to the final state. The productions are again the operators; however in this representation there are two types of operators. Operators of the form $A \rightarrow a$ map statements that a terminal appears in a certain position of the string into statements that a nonterminal derives a substring. They are single-input/single-output operators. However, operators of the form $A \rightarrow BC$ map pairs of statements that a nonterminal derives a substring into similar statements. They are multiple-input/single-output operators [Sandewall (Sandewall, 1971) refers to operators with more than one input and exactly one output as conporators]. Thus, in developing the derivation sequence and derivation tree representations, we have formulated two state-space problem representations in first-order logic.

The types of literals and axioms which are used to represent the states and operators in the two representations are summarized in Figure 6.1. Notice that the distinction is made between the axioms which represent the operators and those which represent the conditions necessary to apply an operator to a state. The sets of axioms DS3, DT2, and DT3 specify to the theorem prover the form of the input state(s), the form of an applicable operator, and the form of the output state. That is, they are really algorithms written in first-order logic for applying the operators to the states.

The use of the cycle inference system on these representations restricts the theorem prover to those resolutions which are necessary to apply an operator to a state. We believe that the cycle inference system is applicable to any state-space problem provided a representation in first-order logic of the states (initial, goal, and intermediate), the operators, and the conditions necessary to apply an operator to a state can be found. The cycle inference system makes it possible to use a theorem prover as the problem-solver for these state-space problems, since it results in the theorem prover modeling a special purpose state-space problem-solver. Each cycle represents the application of an operator, so that the number of steps used by the theorem prover and a special purpose problem-solver in searching for a solution are comparable (this assumes the use of the same strategy). Thus, the cycle inference system can be

		States	Initial State(s)	Goal State	Operators	Axioms which represent the application of an operator to a State
Derivation Sequence Representation	bottom-up analysis	$\sim$SENTFORM$_m(x_1,\ldots,x_m)$	$\sim$SENTFORM$_n(a_1,\ldots,a_n)$	$\sim$SENTFORM$_1$(S)	PROD$_{j,k}(x_1,\ldots,x_j, y_1,\ldots,y_k)$	Axioms in the set DS3
	top-down analysis	SENTFORM$_m(x_1,\ldots,x_m)$	SENTFORM$_1$(s)	SENTFORM$_n$ $(a_1,\ldots,a_n)$		
Derivation Tree Representation		INPUT(y,u) DERIVE(y,u,w)	INPUT$(a_1,1)$ ⋮ INPUT(a_n,n)	DERIVE(s,1,n+1)	PROD$_{1,1}(x_1,y_1)$ PROD$_{1,2}(x_1,y_1,y_2)$	Axioms in the sets DT2 and DT3

Figure 6.1 A summary of the types of literals and axioms which are used in the two representations.

characterized as a resolution-based method for applying an operator axiom to a state clause.

The cycle inference system is general enough to be applicable both to representations with multiple-input operators and to those with a set of initial states. If the operators are single-input/single-output, then the use of the cycle inference system will produce input proofs as with the derivation sequence representation. If the operators are multiple-input/single-output and there is a single initial state, the proofs will be linear (Loveland, 1970; Luckham, 1970). The derivation tree representation provides an example of the use of the cycle inference system on a representation with both multiple-input operators and a set of initial states. The proofs in this representation which are shown are not linear because there is more than one initial state.

There are certain considerations that can be made in implementing the cycle inference system. A cycle can be thought of as a macro-resolution. It is a sequence of resolutions necessary to apply an operator axiom to a state clause. The only resolvent in this sequence of resolutions which can be used in further deductions is the last one, the state clause. The intermediate resolvents can never be used again, and hence do not have to be stored. It is possible that the first parts of cycles necessary to apply two distinct operators to a state (or possibly a set of states) are the same. For example in the derivation tree representation, the two operators $A \rightarrow BC$ and $B \rightarrow BC$ will both be applicable to the same pairs of states. The first two resolutions in the cycles for applying these two operators will be the same. An efficient implementation of the cycle inference system will not perform these first two resolutions twice.

The cycle inference system is not a theorem proving search strategy because its use does not indicate which pair of clauses to use next in a resolution, but rather indicates which pairs of clauses can be used as resolvends. A search strategy can be used in conjunction with the cycle inference system in the same manner that it is used with any problem-solver. The search strategy specifies which state clause (sets of state clauses for the case of multiple-input operators (Kowalski, 1970)) are to be input to the next cycle. To generate the state clauses of the derivation tree representation in the same order as the Cocke algorithm, a breadth-first search of the space is made. Many types of search strategies, depth-first, breadth-first, or a combination of the two which uses some heuristic can be used with the derivation sequence representation. This representation is not good as the derivation tree representation in the sense that the number of states which must be expanded to find a solution in the former representation is greater than in the latter independent of the search strategy employed.

We have shown that the two representations fall into a larger class of unsatisfiable sets of clauses for which input proofs are known to exist, namely those sets where each clause has at most one positive literal. Thus, although input resolution in general is not complete, it is complete for sets of clauses of the above form. Many representations do satisfy this property, including those of state-space problems which use a single positive literal to represent a state.

The method which was used in Section 5 to build up, by functional composition, a representation of the path from the initial state(s) to the current state can be used in any representation of this type. The method is to include in the state clause a term for the composed functions and to build into the operator axioms the steps which compose the functions. Its use provides a concise representation of the particular solution which was found.

The method which we have illustrated of specifying a state-space problem in first-order logic and using a theorem prover employing the cycle inference system as a problem-solver can be used to experiment with state-space problems. The cycle inference system is a problem-solver for a class of state-space problems — namely, those problems where the information required to specify the states and the operators can be expressed in first-order logic. The structures of the states and the operators for the two representations presented in this paper were not extremely complicated, and therefore, it was not very difficult to represent them in first-order logic. Some state-space problems may have states and operators whose structures are so complex that even though it is possible to formulate them in first-order logic, it is unadvisable to do so. We do not claim that whenever it is possible to use the cycle inference system to "interpret" a state-space problem that this method should be employed. However, the cycle inference system does provide all of the structure of a resolution-based theorem prover (constants, variables and quantifiers, functions, predicates, logical operators). Of course the necessity of using axiom schemas in the derivation sequence representation reminds us that first-order logic does not allow quantification over predicates, and (more generally) that first-order logic is less than the ultimate language to use for problem representations. Nevertheless, first-order logic is well-defined and is rather general, and it appears as though it can be used to describe a variety of state-space problems. Finally it is worth pointing out that a cycle inference system based problem-solver is problem independent, and that to modify an existing representation all that is necessary is to change the axioms and respecify the nature of the cycles.

BIBLIOGRAPHY

Amarel, S. (1965): "Problem Solving Procedures for Efficient Syntactic Analysis," ACM 20th National Conference.

Amarel, S. (1968): "Representations of Problems of Reasoning About Actions," in B. Meltzer and D. Michie (eds.), Machine Intelligence 3, American Elsevier, New York, pp. 131-171.

Chang, C. L. (1970): "The Unit Proof and Input Proof in Theorem Proving," JACM, vol. 17, no. 4, pp. 698-707.

Chang, C. L. and Slagle, J. R. (1971): "Completeness of Linear Refutation for Theories with Equality," JACM, vol. 18, no. 1, pp. 126-136.

Chomsky, N. (1959): "On Certain Formal Properties of Grammars," Inf. and Control, vol. 2, no. 2, pp. 137-167.

Cocke, J. and Schwartz, J. T. (1970): Programming Languages and Their Compilers, Courant Institute of Mathematical Sciences, New York.

Davis, M. (1963): "Eliminating the Irrelevant From Mechanical Proofs," Proc. of Symp. in Appl. Math., vol. 15, pp. 15-30.

Davis, M. and Putnam, H. (1960): "A Computing Procedure for Quantification Theory," JACM, vol. 7, no. 3, pp. 201-215.

Green, C. C. (1969a): "Application of Theorem Proving to Problem Solving," In: D. E. Walker and L. M. Norton (eds.), Proc. Intern. Joint Conf. on Artificial Intelligence, Washington, D. C., pp. 219-239.

Green, C. C. (1969b): "The Application of Theorem Proving to Question-Answering Systems," Ph.D. Thesis, Electrical Engr. Dept., Stanford U.

Hart, P. N., Nilsson, N., and Raphael, B. (1968): "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," IEEE Trans. Sys. Sci. Cybernetics, vol. SSC-4, no. 2, pp. 100-107.

Hopcroft, J. E. and Ullman, J. D. (1969): Formal Languages and Their Relation to Automata, Addison-Wesley, Reading, Mass.

Knuth, D. E. (1968): Fundamental Algorithms, Addison-Wesley, Reading, Mass.

Kowalski, R. (1970): "Search Strategies for Theorem Proving," In: B. Meltzer and D. Michie (eds.), Machine Intelligence 5, American Elsevier, New York, pp. 181-200.

Kowalski, R. and Hayes, P. (1969): "Semantic Trees in Automatic Theorem Proving," In: B. Meltzer and D. Michie (eds.), Machine Intelligence 4, American Elsevier, New York, pp. 87-101.

Loveland, D. W. (1970): "A Linear Format for Resolution," In: Proc. IRIA 1968 Symp. Auto. Demonstration, Lecture Notes on Math., no. 125, Springer-Verlag, New York, 1970.

Luckham, D. (1970): "Refinement Theorems in Resolution Theory," In: Proc. IRIA 1968 Symp. Auto. Demonstration, Lecture Notes on Math., no. 125, Springer-Verlag, New York, 1970.

Luckham, D. and Nilsson, N. (1971): "Extracting Information from Resolution Proof Trees," Artificial Intelligence, vol. 2, no. 1, pp. 27-54.

Meltzer, B. (1971): "Prolegamena To a Theory of Efficiency of Proof Procedures," In: N. V. Findler and B. Meltzer (eds.), Artificial Intelligence and Heuristic Programming, American Elsevier, New York, pp. 15-36.

Michie, D. (1970): "Experiments With an Adaptive Graph Traverser," In: B. Meltzer and D. Michie (eds.), Machine Intelligence 5, American Elsevier, New York, pp. 301-320.

Nilsson, N. (1971): Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York.

Robinson, J. A. (1965): "A Machine-Oriented Logic Based on the Resolution Principle," JACM, vol. 12, no. 1, pp. 23-41.

Robinson, J. A. (1970): "An Overview of Mechanical Theorem Proving," In: R. Banerji and M. D. Mesarovic (eds.), Theoretical Approaches to Non-Numerical Problem Solving, Springer-Verlag, New York, pp. 2-20.

Robinson, J. I. and Marks, S. L. (1965): "A System for Automatic Analysis of English Text," RM-4654-PR, The RAND Corp., Sept. 1965.

Sandewall, E. (1971): "Heuristic Search: Concepts and Methods," In: N. V. Findler and B. Meltzer (eds.), Artificial Intelligence and Heuristic Programming, Springer-Verlag, New York, pp. 81-100.

Slagle, J. R. and Koniver, D. A. (1971): "Finding Resolution Proofs and Using Duplicate Goals in AND/OR Trees," Inf. Sciences, vol. 3, no. 4, pp. 315-342.

Wos, L. T., Robinson, G., and Carson, D. F. (1965): "Efficiency and Completeness of the Set-of-Support Strategy," JACM, vol. 12, no. 4, pp. 536-541.

Younger, D. H. (1967): "Recognition and Parsing of Context-Free Languages in Time n^3," Inf. and Control, vol. 10, pp. 189-208.