~~Bob Welham~~

LOGIC FOR PROBLEM SOLVING

by

Robert Kowalski

Memo No 75

DEPARTMENT OF COMPUTATIONAL LOGIC
SCHOOL OF ARTIFICIAL INTELLIGENCE
UNIVERSITY OF EDINBURGH

9 HOPE PARK SQUARE EDINBURGH EH8 9NW

LOGIC FOR
PROBLEM SOLVING

by

Robert Kowalski

Memo No 75

March 1974.

INTRODUCTION.

Our thesis is that predicate logic is a useful language for representing knowledge. It is useful for stating problems and it is useful for representing the pragmatic information necessary for effective problem-solving. We shall support our thesis by investigating the application of predicate logic to problems of syntactic analysis and robot plan-formation. We shall investigate the utility of employing predicate logic both as a programming language and as a problem-solving language for reducing the solution of problems to subproblems.

We shall argue that the distinction between top-down and bottom-up problem-solving, which arises in syntactic analysis, applies more generally in predicate logic. It characterises the main difference between different proof procedures. Top-down proof procedures are goal-oriented. They reduce problems to subproblems with the objective of eventually reducing the original problem to a set of solved subproblems. Bottom-up proof procedures are consequence-oriented. They derive new solutions (or assertions of fact) from old ones with the objective of eventually deriving a solution of the original problem.

The top-down, bottom-up distinction is useful in the robot-plan formation problem. The frame axiom asserts that a fact which holds in a given state of the world continues to hold in the new state obtained by performing an action. Certain facts, which are affected by the action, are exceptions to the rule. The frame problem is the problem of the combinatorial explosion caused by interpreting the frame axiom bottom-up. Bottom-up interpretation of the frame axiom involves copying facts which hold in old states, re-asserting that they continue to hold in new states. Top-down interpretation of the frame axiom involves reducing the problem of determining that a fact holds in a new state to the problem of determining that the same fact previously held in an earlier state. It is generally considered that the frame problem arises as a result of applying predicate logic to the representation of knowledge about a changing world, and that the problem cannot be solved within the constraints of predicate logic. In our formulation, the frame problem is solved in first-order predicate logic by the top-down interpretation of the frame axiom.

Top-down analysis is the key also to the problem-reduction and procedural interpretations of predicate logic. In the problem-reduction interpretation, a sentence of the form

$$B \leftarrow A_1, \ldots, A_n$$

(which is read "B if A_1 and...and A_n") is interpreted as a problem-solving method which reduces problems of the form B to the set of subproblems $\{A_1, \ldots, A_n\}$. In the procedural interpretation, the same sentence is interpreted as a procedure declaration whose name B identifies the form of the procedure calls to which it can respond and whose body $\{A_1, \ldots, A_n\}$ is an unordered set of procedure calls A_i.

In the problem-reduction interpretation, predicate logic is a more satisfactory model of problem-solving than the problem-reduction model used in artificial intelligence. Predicate logic deals with the reduction of problems to dependent subproblems, which need to be solved compatibly. Problems consist of hypotheses and conclusions. Effective problem-solving involves a combination of top-down analysis starting from the conclusion of the problem and bottom-up analysis starting from the hypotheses. In general several problem-solving methods might need to co-operate in order to solve a given problem. Individual methods might work in separate cases. Between them the several methods might work in all the possible cases. In contrast the usual problem-reduction model deals with the reduction of problems to independent subproblems. Problems and subproblems have no hypotheses. Problem-solving methods need to be self-sufficient and to work independently.

In the procedural interpretation, predicate logic is a more satisfactory programming language than the machine-oriented languages which are used to program computers today. Unique among programming languages, predicate logic has been designed for the formalisation of human thought. It has a simple machine-independent semantics which makes predicate logic programs easier to modify and adapt to other purposes, and easier to integrate into more complicated programs. Predicate logic programs have no explicit input-output distinction. A procedure, written with the intention of constructing as output the result of appending one list to another, can be used to divide an input list into two parts which are returned as output. Since procedure bodies are sets (rather than sequences) of procedure calls, scheduling of procedure calls can be made sensitive to the context in which the procedure is called. Typically the appropriate scheduling of procedure calls depends on the input-output pattern of the procedure call which invokes the given procedure. Sets of assertions function as data structures. Top-down execution of procedures interrogates data structures, whereas bottom-up execution manipulates them.

The ability to combine flexibly top-down and bottom-up analysis is provided by the connection graph theorem-proving system. Sentences are stored in a graph whose arcs connect procedure calls with matching procedure names. Accessing relevant procedures and intersecting bottom-up with top-down analyses is facilitated by the connections in the graph.

These topics are treated in the following six chapters:

<u>Chapter 1</u>	deals with the machine-independent syntax and semantics of predicate logic.
<u>Chapter 2</u>	investigates the parsing problem, its predicate logic formulation, and the top-down and bottom-up strategies of solution.
<u>Chapter 3</u>	investigates the robot plan-formation problem, the frame problem, and conditional plans.
<u>Chapter 4</u>	studies the problem-reduction interpretation of predicate logic.
<u>Chapter 5</u>	studies the procedural interpretation.
<u>Chapter 6</u>	introduces the connection graph theorem-proving system.

CHAPTER 1. MACHINE-INDEPENDENT SYNTAX AND SEMANTICS.

The most outstanding characteristic of predicate logic regarded as a programming language is that programs have a simple machine-independent semantics. Both the syntax and semantics of predicate logic are further simplified by dealing with sentences in clausal form. Before the formal definitions, clausal form and such notions as atomic formulas and inconsistency will be illustrated in the following example.

Example of the Fallible Greek.

(1) Human (Turing) $\leftarrow$

(2) Human (Socrates) $\leftarrow$

(3) Greek (Socrates) $\leftarrow$

(4) Fallible (x) $\leftarrow$ Human (x)

(5) $\leftarrow$ Fallible (x), Greek (x)

In these five _clauses_, "Human", "Greek" and "Fallible" are _predicate symbols_, "Turing" and "Socrates" are _constant symbols_ and "x" is a _variable_. A predicate symbol P applied to a constant or variable t, i.e. P(t), is an _atomic formula_, read

"t is P"

Clauses (1), (2) and (3) unconditionally assert that Turing is human, Socrates is human, and Socrates is Greek, respectively. Clause (4) states that all humans are fallible by stating literally that x is fallible if x is human, no matter what x is. Clause (5) states that no x is both fallible and Greek. Clearly (5) contradicts what is implicit in (1)-(4), namely that Socrates is both fallible and Greek. We say that (1)-(5) are _inconsistent_.

The example of the fallible Greek has been used often to explain the behaviour of PLANNER programs {18}. Our intention in using this example is just the opposite: to show that predicate logic programs can be understood without understanding the behaviour they invoke inside a machine.

The Syntax of Sentences in Clausal Form.

A _sentence_ (in clausal form) is a set $\{C_1,\dots,C_n\}$ of clauses C_i. A _clause_ is a pair of sets of atomic formulas

$$B_1,\dots,B_m \leftarrow A_1,\dots,A_n$$

The two sets are written without the surrounding curly brackets and are separated by a backward arrow. The set $\{B_1,\ldots,B_m\}$ is the <u>conclusion</u> of the clause and the set $\{A_1,\ldots,A_n\}$ is its <u>hypothesis</u>. The <u>null clause</u>, n = 0 and m = 0, is written $\square$.

An <u>atomic formula</u> (or <u>atom</u>) is an expression

$P(t_1,\ldots t_k)$

where P is a k-ary predicate symbol and $t_1,\ldots,t_k$ are terms.

A <u>term</u> is a variable, a constant symbol, or an expression

$f(t_1,\ldots,t_k)$

where f is a k-ary function symbol and $t_1,\ldots,t_k$ are terms.

The sets of <u>variables</u>, <u>function symbols</u> and <u>predicate symbols</u> are any three disjoint sets of objects. Associated with every function symbol and predicate symbol is a unique natural number which is its <u>arity</u>. We assume there is an unlimited supply of variables and of function symbols and predicate symbols of every arity. <u>Constant symbols</u> are function symbols of zero-arity. (Thus "Socrates ( )" is a term, whereas "Socrates" is only a constant symbol.)

Because of the different positions they occupy in atomic formulas, it is always possible to distinguish between predicate symbols, function symbols and variables. It is convenient however to treat constant symbols, standing alone, as terms, allowing an expression such as Human (Socrates) to count as an atom. This convention introduces ambiguities: it is impossible to distinguish between constant symbols and variables only by means of the positions they occupy in atoms. The ambiguity is removed by employing the convention that the lower case letters

u,v,w,x,y,z,

possibly adorned with subscripts or other decorations, are used exclusively for variables. Thus in the atom

Adm(cons(x,nil), cons(y,nil)),

Adm is a predicate symbol of arity 2, cons a function symbol of arity 2, nil a constant symbol, x and y variables.

In addition to the syntactic form of sentences, syntax includes <u>proof theory</u>, which deals with axioms of logic, rules of inference and proof procedures. Proof theory can be used to assign an operational, behaviouristic meaning to sentences. Such a use of proof theory corresponds to the operational semantics of programming languages: the meaning of a program is determined by the behaviour of a machine which executes it. In this chapter we are concerned exclusively with the machine-independent semantics of predicate logic.

The Informal Semantics of Predicate Logic.

Read a sentence $\{C_1,\ldots,C_N\}$ as the conjunction of its clauses:

C_1 and...and C_N.

Read a clause $B_1,\ldots,B_m \leftarrow A_1,\ldots,A_n$ containing variables $x_1,\ldots,x_k$ as stating that

for all $x_1,\ldots,x_k$,
B_1 or...or B_m if A_1 and...and A_n.

In the special case m = 0 read

for no $x_1,\ldots,x_k$, A_1 and...and A_n.

For m = 0 and n = 0 read $\square$ as a contradiction. (We use the backward arrow $B \leftarrow A$, B if A, instead of the more usual forward arrow $A \rightarrow B$, if A then B, because it is more convenient for the problem-solving and procedural interpretations of predicate logic.)

A sentence S is inconsistent if every way of interpreting the predicate symbols and function symbols in S makes S false. If S is inconsistent then it is also said to be unsatisfiable, since no interpretation satisfies S in the sense of making it true. Any sentence containing the null clause or such implicit contradictions as $P \leftarrow$ and $\leftarrow P$ or $P(t) \leftarrow$ and $\leftarrow P(x)$ is obviously unsatisfiable.

Before defining the semantics of predicate logic more formally, we shall illustrate some of the expressive capabilities of predicate logic by means of some examples.

The Factorial example.

(1) Fact(0,s(0)) ←
(2) Fact(s(x),u) ← Fact(x,v), Times(s(x),v,u)
(3) ← Fact(s(s(0)),x)

Here read Fact(a,b) as stating that the factorial of a is b and Times(a,b,c) as stating that a times b is c. Regard the terms O,s(O),s(s(O)),...,s^n(O),... as numerals denoting the natural numbers: s^n(O) denotes the number n. The term s(a) denotes a+1, the successor of a.

Clause (1) asserts that the factorial of O is 1. Clause (2) states that the factorial of x+1 is x+1 times v where v is the factorial of x. Clause (3) states that no x is the factorial of 2.

We assume that Times is interpreted as the Times relation over the natural numbers. Therefore clauses (1)-(5) are inconsistent where

(4) Times(s(O),s(O),s(O)) ←

(5) Times(s(s(O)),s(O),s(s(O))) ← .

Notice that, regarded as a program for computing the factorial relation, (1)-(3) can be understood without reference to an execution mechanism which interprets and executes the program.

Append example.

(1) Append(nil,x,x) ←

(2) Append(cons(x,y),z,cons(x,u)) ← Append(y,z,u)

(3) ← Append(cons(a,nil),cons(b,cons(c,nil)),x)

Read Append(a,b,c) as stating that the result of appending the list of objects b to the list a is the list c. Regard a term cons(a,b) as a list. Its first element is a and b is the rest of the list. The constant symbol nil denotes the empty list. cons(a,nil) represents the list containing the single element a and is abbreviated {a}. cons(b,cons(c,nil)) represents the list containing the 2 elements b and c, in that order, and is abbreviated {b,c}.

Clause (1) asserts that appending any list x to the empty list results in the same list x. Clause (2) states that appending z to a non-empty list, whose first element is x and remainder is y, results in a list with the same first element x and remainder u which results from appending z to y. Clause (3) states that no list results from appending {b,c} to {a} and is inconsistent with (1) and (2) which imply that {a,b,c} results from appending {b,c} to {a}.

The use of terms cons(a,b) to represent lists is common in list-processing programming languages. Foster's book {15} is a readable introduction to list-processing.

Admissible Pairs example.

(1) $Adm(x,y) \leftarrow Double(x,y),\ Triple(x,y)$

(2) $Double(nil,nil) \leftarrow$

(3) $Double(cons(x,y),cons(u,v)) \leftarrow Times(s(s(0)),x,u),\ Double(y,v)$

(4) $Triple(nil,nil) \leftarrow$

(5) $Triple(cons(x,nil),cons(u,nil)) \leftarrow$

(6) $Triple(cons(x,cons(y,z)),cons(u,v)) \leftarrow Times(s(s(s(0))),u,y)$
$Triple(cons(y,z),v).$

(7) $\leftarrow Adm(cons(s(0),u),v)$

Clauses (1)-(6) describe the relation Adm(a,b) which holds between two lists a and b when

$$b_i = 2a_i \quad \text{and}$$
$$a_{i+1} = 3b_i, \quad \text{for all } i < n \ ,$$

where a_i and b_i are the i^{th} elements of the lists a and b respectively. Clause (7) states that no pair of lists a and b is admissible if the first element of a is 1. But (7) is inconsistent with (1)-(6) which together with the correct interpretation of the Times relation imply that each of the pairs

$\{1\}$ and $\{2\}$, $\{1,6\}$ and $\{2,12\}$,
$\{1,6,36\}$ and $\{2,12,72\}$,...

is admissible.

Horn Clauses.

The preceding examples have used only Horn clauses

$$B_1,\ldots,B_m \leftarrow A_1,\ldots,A_n, \quad \text{where } m \leq 1,$$

which contain at most one atom in the conclusion. Horn clauses are adequate for many applications of predicate logic.

It is convenient to distinguish and have separate names for four kinds of Horn clauses:

(1) $m = 0,\ n = 0,\ \square$,
the null clause

(2) $m = 1,\ n = 0,\ B \leftarrow$
A Horn clause with no hypothesis is an assertion.

(3) $m = 0, n \neq 0, \leftarrow A_1,\ldots,A_n$.
A Horn clause with no conclusion is called a <u>goal statement</u>.

(4) $m = 1, n \neq 0, B \leftarrow A_1,\ldots,A_n$.
Every other Horn clause is called an <u>operator</u>.

The motivation for the terminology, "goal statement" and "operator" will be explained in the chapter on the problem-reduction interpretation of predicate logic.

The following two examples illustrate the use of non Horn clauses.

<u>Robert is always working</u>.

(1) At(Robert,work), At(Robert,home) ←
(2) Working(x) ← At(x,work)
(3) Working(Robert) ← At(Robert,home)
(4) ← Working(Robert)

Robert is either at home or at work. Everyone who is at work is working. But Robert is working even if he is at home. To accuse Robert of not working is to be inconsistent.

<u>Subset example</u>.

(1) Sub(x,y), Memb(arb(x,y),x) ←
(2) Sub(x,y) ← Memb(arb(x,y),y)
(3) ← Sub(A,A)

Clauses (1) and (2) result from rendering in clausal form the definition:

x is a subset of y if, for all z,
z is a member of y if z is a member of x.

Clause (3) asserts that A is not a subset of itself. The inconsistency of (1)-(3) is equivalent to the validity of the proposition that every set is a subset of itself.

This example can be used as an argument against the suitability of clausal form for the representation of knowledge. It can be argued that the rendering into clausal form destroys the intelligibility of the definition of the subset relation.

Alternatively it can be argued that the example shows the limitations of machine-independent semantics for understanding sentences in clausal form.

Clauses (1) and (2) have a natural, machine-oriented, procedural interpretation:

> In order to show that x is a subset of y,
>
> (1) assert that arb(x,y) is some arbitrary member of x and
>
> (2) show that arb(x,y) is a member of y.

The arbitrary element of x has parameters x and y in order to indicate that the element arb(A,B) chosen for showing that A is a subset of B is different from the one arb(C,D) chosen for showing that C is a subset of D or the one arb(B,A) for showing that B is a subset of A. The procedural interpretation of (1) and (2) will be explained more fully in later chapters.

Formal Semantics of Predicate Logic.

The following definition, like the definition of sentence in clausal form, is presented in a top-down manner. The first definition (of inconsistency) explains the goal concept to be defined in terms of other concepts. These concepts become the new goal concepts and are themselves defined in terms of lower-level concepts. Eventually the definitions terminate with a set of primitive, undefined concepts. In contrast, definitions presented in a bottom-up manner begin with the primitive concepts and define new concepts in terms of ones previously encountered. The definitions terminate when the goal concept has been defined.

Top-down presentation of definitions has the advantage that it is goal-directed. Each definition, as it is presented, is motivated in terms of the rôle it plays in defining the original goal concept. The disadvantage is that, since concepts are explained in terms of other undefined concepts, definitions cannot be completely understood as they are presented. Just the opposite holds for bottom-up presentation. Definitions can be understood as soon as they are given. But the motivations for the definitions cannot be understood until all the definitions have been completed.

The distinction between top-down and bottom-up applies in many places. It is the difference between analysis (top-down) and synthesis (bottom-up), between teleology (top-down) and determinism (bottom-up). It applies both to the writing and execution of computer programs and to the discovery and justification of theorems and proofs. The distinction between top-down and bottom-up and the application of predicate logic to the representation of knowledge are the dominating, unifying themes of these lecture notes.

A set $S = \{C_1,\ldots,C_N\}$ of clauses is <u>inconsistent</u> iff it is false in every interpretation I of S. S is <u>false in I</u> iff one of $C_1,\ldots,C_N$ is false in I. Otherwise S is <u>true in I</u>.

A clause C is <u>false in I</u> iff for some substitution σ of variable-free terms for variables in C, the clause $C\sigma$ which results from applying σ to C is false in I. Otherwise C is <u>true in I</u>.

A variable-free clause $B_1,\ldots,B_m \leftarrow A_1,\ldots,A_n$ is <u>false in I</u> iff all the atoms $A_1,\ldots,A_n$ in the hypothesis are true in I and all the atoms $B_1,\ldots,B_m$ in the conclusion are false in I. Otherwise the clause is <u>true in I</u>.

An <u>interpretation I</u> of S is an assignment of one of <u>true</u> or <u>false</u> to all variable-free atomic formulas constructible from the atomic formulas which occur in S. With a given interpretation I of S is associated a <u>domain</u> of objects which we assume to be the set of variable-free terms constructible from the function symbols (and constants) occurring in S. The variable-free terms which can be substituted for variables, in order to construct variable-free atomic formulas from atoms occurring in S, are chosen from the domain of objects associated with I.

For the set of clauses (1)-(4) of the example of the fallible Greek, the assignment of

<u>true</u> to Human(Socrates)
<u>true</u> to Human(Turing)
<u>true</u> to Greek(Socrates)
<u>true</u> to Fallible(Socrates)
<u>true</u> to Fallible(Turing)

is an interpretation I of (1)-(4). Notice that there are only two different variable-free instances of clause (4),

Fallible(x) $\leftarrow$ Human(x) .

Both instances are true in I. Notice that I is the only interpretation of (1)-(4) in which (1)-(4) is true.

But clause (5), $\leftarrow$ Fallible(x), Greek(x), is false in I. Therefore (1)-(5) is inconsistent, since no interpretation of (1)-(5) makes all of (1)-(5) true.

CHAPTER 2. TOP-DOWN AND BOTTOM-UP INTERPRETATIONS OF PREDICATE LOGIC.

The parsing problem, of showing that a string of words forms a sentence according to given rules of grammar, can be represented as a problem of demonstrating the inconsistency of a set of clauses in predicate logic. Different parsing procedures for determining that a string is a sentence correspond to different proof procedures for determining the inconsistency of a set of clauses. Top-down parsing procedures correspond to goal-directed proof procedures which work backwards from the conclusion of the theorem to be proved, reducing problems to subproblems, with the objective of eventually reducing the original problem to a set of initially solved subproblems. Bottom-up parsing procedures correspond to proof procedures which work forwards from the initial set of solved problems, deriving new solved problems from old ones, with the objective of eventually deriving a solution of the original problem.

Top-down and bottom-up proof procedures apply generally to the task of demonstrating the inconsistency of sets of clauses.

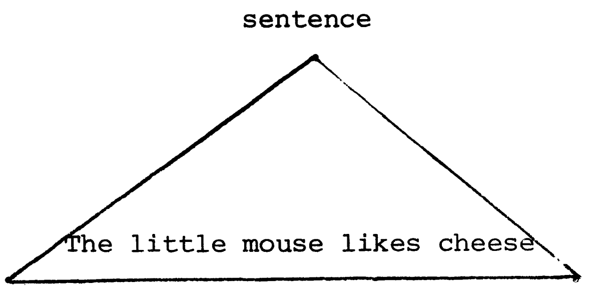
The Parsing problem.

The following informal description of the parsing problem and parsing procedures is based on Amarel's treatment described by Foster {16}.

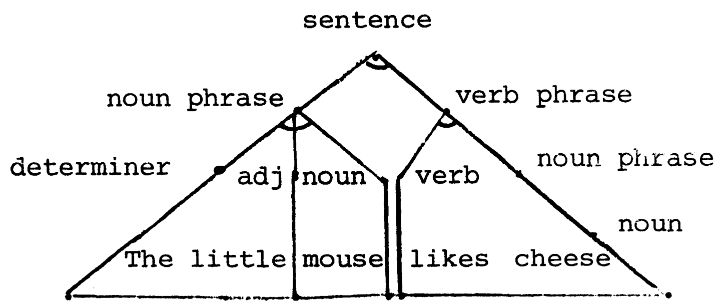
Given a grammar and an initial string of words such as

"The little mouse likes cheese"

the parsing problem is to demonstrate that the string is a sentence by filling in the triangle



with a parse



The parse tree is constructed in accordance with the rules of grammar. In this example, nine rules of grammar have been applied:

(1) A noun phrase followed by a verb phrase is a sentence.

(2) A determiner followed by an adjective followed by a noun is a noun phrase.

(3) A noun is a noun phrase.

(4) A verb followed by a noun phrase is a verb phrase.

(5) "The" is a determiner.

(6) "little" is an adjective.

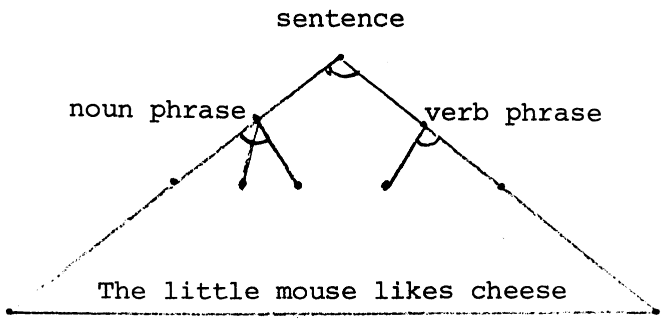
(7) "mouse" is a noun.

(8) "likes" is a verb.

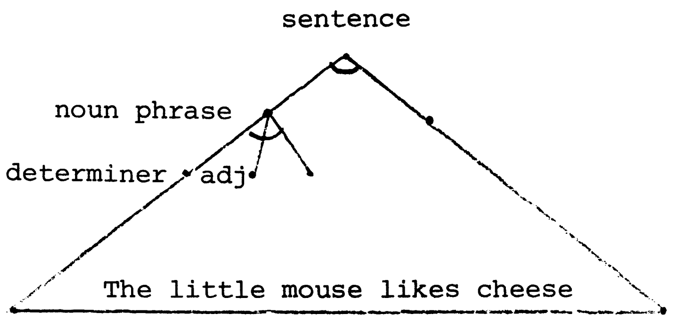
(9) "cheese" is a noun.

Different ways of filling in the triangle determine different parsing procedures. <u>Top-down procedures</u> are determined by filling in the triangle from the top downwards. <u>Bottom-up procedures</u> are determined by filling in the triangle from the bottom upwards.

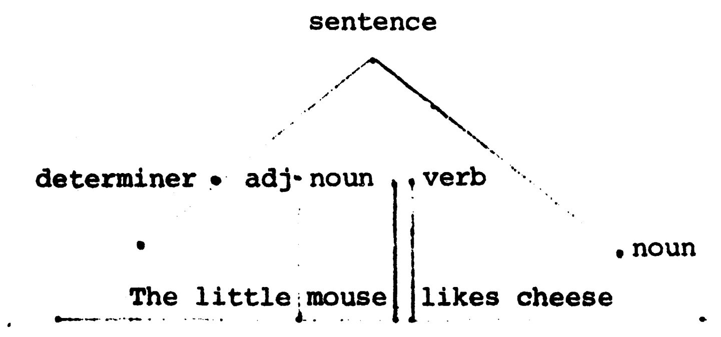
A top-down procedure might involve an unbiased generation of all branches in parallel:



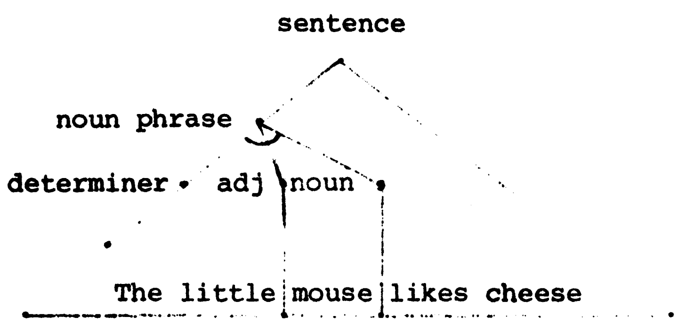
or it might be biased towards a left-to-right analysis of the sentence:



Similarly, a bottom-up procedure might generate the parse by an unbiased analysis of all components of the string in parallel:



Or it might be biased towards a left-to-right investigation:



The triangle can be filled in from right-to-left, bi-directionally top-down and bottom-up or by any other method. Every method of filling in the triangle determines a parsing procedure. For our purposes, it is important to distinguish at this time mainly the top-down and bottom-up procedures.

When the parsing problem is formulated in predicate logic, top-down and bottom-up parsing procedures correspond to different proof procedures. Later in this chapter, top-down and bottom-up proof procedures will be defined in detail for the general problem of determining the inconsistency of sets of Horn clauses.

A predicate logic representation of the parsing problem.

Regard the initial string of words as a graph, the arcs of which are labelled by the words occurring in the initial string. A node occurs between consecutive words and also at the beginning and end of the string:

^{1}The ^{2}little ^{3}mouse ^{4}likes ^{5}cheese^6.

The initial graph is represented by a set of assertions:

(1) The(1,2) $\leftarrow$

(2) little(2,3) $\leftarrow$

(3) mouse(3,4) $\leftarrow$

(4) likes(4,5) $\leftarrow$

(5) cheese(5,6) $\leftarrow$

The rules of grammar are formulated as operators:

(6)	S(x,y)	$\leftarrow$ Np(x,z),Vp(z,y)
(7)	Np(x,y)	$\leftarrow$ Noun(x,y)
(8)	Np(x,y)	$\leftarrow$ Det(x,u),Adj(u,v),Noun(v,y)
(9)	Vp(x,y)	$\leftarrow$ Verb(x,y)
(10)	Vp(x,y)	$\leftarrow$ Verb(x,z),Np(z,y)
(11)	Det(x,y)	$\leftarrow$ The(x,y)
(12)	Adj(x,y)	$\leftarrow$ little(x,y)
(13)	Noun(x,y)	$\leftarrow$ mouse(x,y)
(14)	Verb(x,y)	$\leftarrow$ likes(x,y)
(15)	Noun(x,y)	$\leftarrow$ cheese(x,y)

The goal of determining that the string of words is a sentence is formulated in the goal statement:

(16) $\leftarrow$ S(1,6)

Clauses (1)-(16) are inconsistent. A proof of their inconsistency involves generating a parse of the string as a sentence.

A more realistic example would include many more clauses of the kind (6)-(15) which define the rules of grammar and record the grammatical categories of individual words in the vocabulary. In this example only one clause (9) is unnecessary for a proof of inconsistency. In more realistic examples the number of unnecessary clauses generally exceeds that which is necessary for a proof.

Figure 1 illustrates a refutation of (1)-(16) corresponding to a top-down, left-to-right parse of the sentence. The refutation is a sequence of goal statements beginning with the initial goal statement (16) and ending with the empty one.

```
      o ← S(1,6)
      |
(6)   |
      |
      o ← Np(1,z),Vp(z,6)
      |
(8)   |
      |
      o ← Det(1,u),Adj(u,v),Noun(v,z),Vp(z,6)
      |
(11)  |
      |
      o ← The(1,u),Adj(u,v),Noun(v,z),Vp(z,6)
      |
(1)   |
      |
      o ← Adj(2,v),Noun(v,z),Vp(z,6)
      |
(12)  |
      |
      o ← little(2,v),Noun(v,z),Vp(z,6)
      |
(2)   |
      |
      o ← Noun(3,z),Vp(z,6)
      |
(13)  |
      |
      o ← mouse(3,z),Vp(z,6)
      |
(3)   |
      |
      o ← Vp(4,6)
      |
(10)  |
      |
      o ← Verb(4,z),Np(z,6)
      |
(14)  |
      |
      o ← likes(4,z),Np(z,6)
      |
(4)   |
      |
      o ← Np(5,6)
      |
(7)   |
      |
      o ← Noun(5,6)
      |
(15)  |
      |
      o ← cheese(5,6)
      |
(5)   |
      |
      o □
```

Figure 1. A top-down refutation of clauses (1)-(16).

A derived goal statement C_{i+1} is obtained from the proceding goal statement C_i in the sequence by

(1) matching, with some substitution θ of terms for variables, the underlined selected atom A in C_i with the atom A' in the conclusion of some clause C in the initial set (1)-(15) of assertions and operators ($A\theta = A'\theta$),

(2) deleting the selected atom in C_i and replacing it by the set of atoms constituting the hypothesis of C, and

(3) applying the matching substitution θ to the resulting clause.

In this example the selection of atoms in goal statements determines that the parse is executed in a left-to-right manner

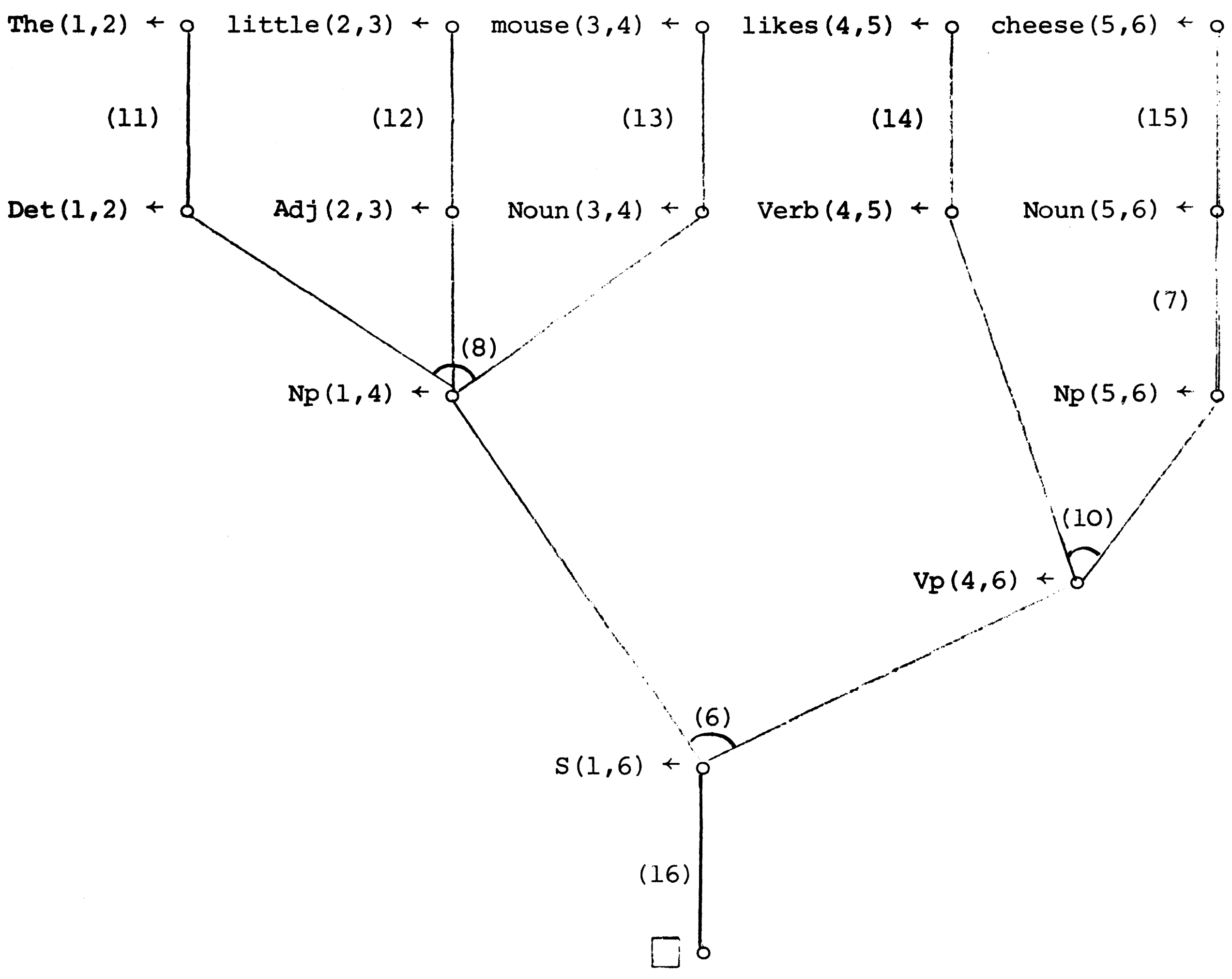


Figure 2. A bottom-up refutation of clauses (1)-(16).

The refutation in Figure 2 corresponds to a bottom-up parse which is unbiased towards left or right directions. The refutation is a tree of assertions beginning with the initial assertions at the tips of the tree and ending with the empty assertion at the root. A new assertion is derived from its immediate predecessors in the tree

$$A_1 \leftarrow, \ldots, A_n \leftarrow$$

by matching their atoms with all the atoms in the hypothesis of some initial operator

$$A \leftarrow A_1', \ldots, A_n'$$

The new assertion is

$$A\theta \leftarrow$$

where θ is the matching substitution,

$$A_i\theta = A_i'\theta, \quad \text{for all } i \leq n .$$

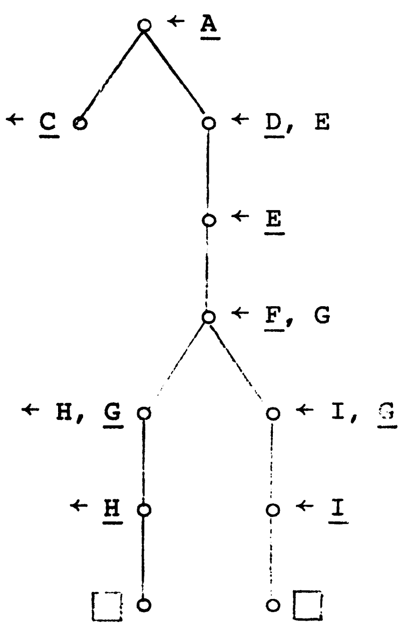
The operators in the Horn clause representation of the parsing problem can be regarded as a program for parsing strings of words as sentences. The set of initial assertions functions as a data base which records the individual words in the initial string. The initial goal statement functions as the top level procedure call. Interpreting operators top-down uses them as procedures for interrogating the data base. Interpreting operators bottom-up uses them to manipulate the data base.

Our formulation of the parsing problem was obtained jointly with Alain Colmerauer and results from representing his Q-system { 8 } in predicate logic. It is interesting that the Q-system is a bottom-up parsing procedure. The more abstract predicate logic formulation of Q-systems is neutral with respect to top-down and bottom-up directions.

Although our example deals only with context-free rules of grammar, it is easy to see how to extend the representation in order to deal with context-sensitive grammars and arbitrary re-writing systems.

Search spaces of derivations.

It is important to distinguish between individual derivations determined by a given inference system and whole spaces consisting of all the derivations which are determined by the inference system. Figure 3 shows the search space of all top-down derivations determined by a given initial set of clauses and by the selection procedure which selects the alphabetically earliest atom in every goal statement.



← A

A ← C

A ← D, E

D ← E

E ← F, G

F ← H

F ← I

G ←

H ←

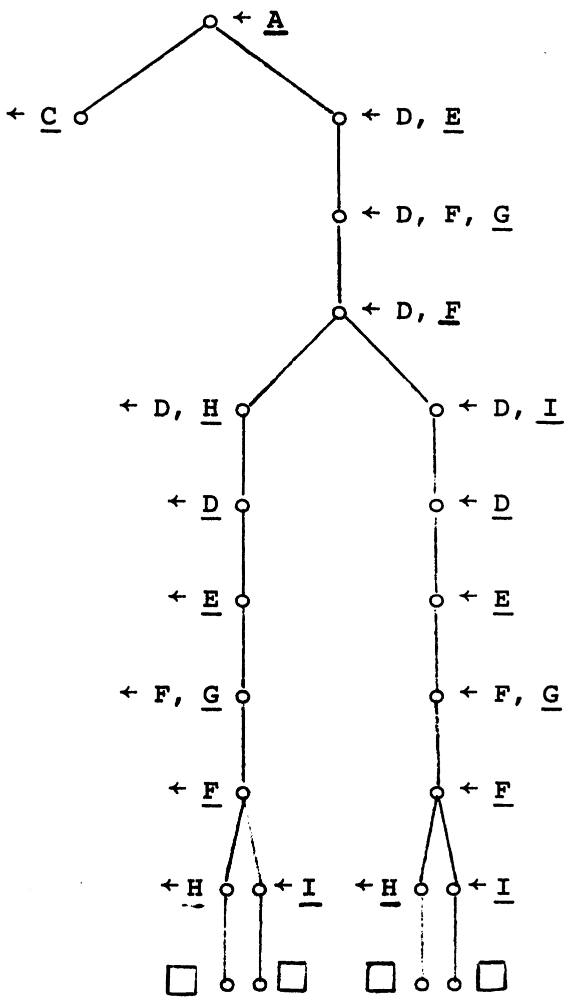
I ←

J ←

K ← J

Figure 3. A search space of all top-down derivations determined by the selection of alphabetically earliest atoms in goal statements. Notice that, because the hypothesis of a goal statement is a set of atoms, application of the operator D ← E to the goal statement ← D, E results in the goal statement ← E. No operator applies to the goal statement ← C. The operator K ← J, and the assertion J ←, on the other hand, apply to no goal statement in the search space.

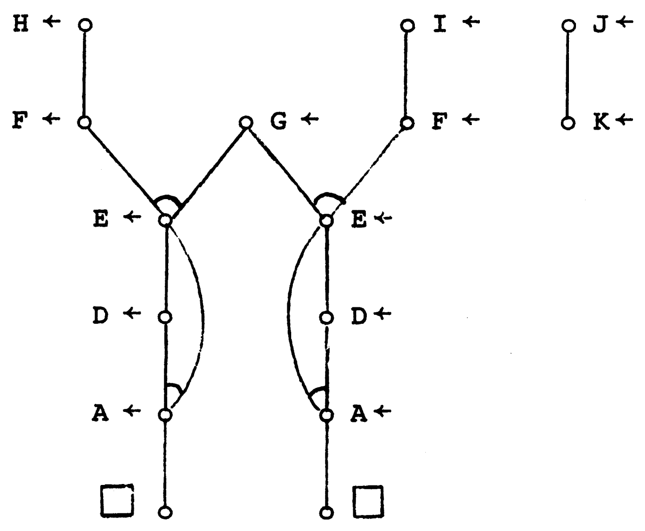
Figure 4 shows the different search space of top-down derivations determined by the same initial set of clauses and by the different selection procedure which always selects the alphabetically latest atom in goal statements. In both figures, derivations are individual paths in the search space structured as a tree. Paths beginning with the initial goal statement and ending with the empty clause are refutations.



← A
A ← C
A ← D, E
D ← E
E ← F, G
F ← H
F ← I
G ←
H ←
I ←
J ←
K ← J

Figure 4. A search space of derivations determined by the selection of alphabetically latest atoms in goal statements. Notice that this selection procedure results in a search space having more and longer derivations than the search space of Figure 3.

Figure 5 shows the search space of all bottom-up derivations for the same initial set of clauses. Every legal bottom-up derivation from the initial set of clauses is an appropriate subtree in the search space.



← A
A ← C
A ← D, E
D ← E
E ← F, G
F ← H
F ← I
G ←
H ←
I ←
J ←
K ← J

Figure 5. A search space of all bottom-up derivations. Notice that the operator A ← C, which interpreted top-down replaces the problem of solving A by the problem of solving C, is not applicable bottom-up because the assertion C ← is not attainable. On the other hand, the operator K ← J, which is not applicable top-down, applies when interpreted bottom-up to the initial assertion J ←.

It is worth distinguishing between tree-representations and graph-representations of derivations and search spaces of derivations. Tree-representations, such as those employed in Figures 3 - 5, contain distinct nodes for distinct ways of deriving the same clause. Graph-representations, such as those in Figures 6 and 7, contain a single node for every clause no matter how it is derived. Graph-representations suggest proof procedures which check, every time a clause is generated, whether the clause has been generated redundantly before. Tree-representations leave open the option of testing for redundancy. In the sequel, tree-representations will be used in preference to graph-representations.

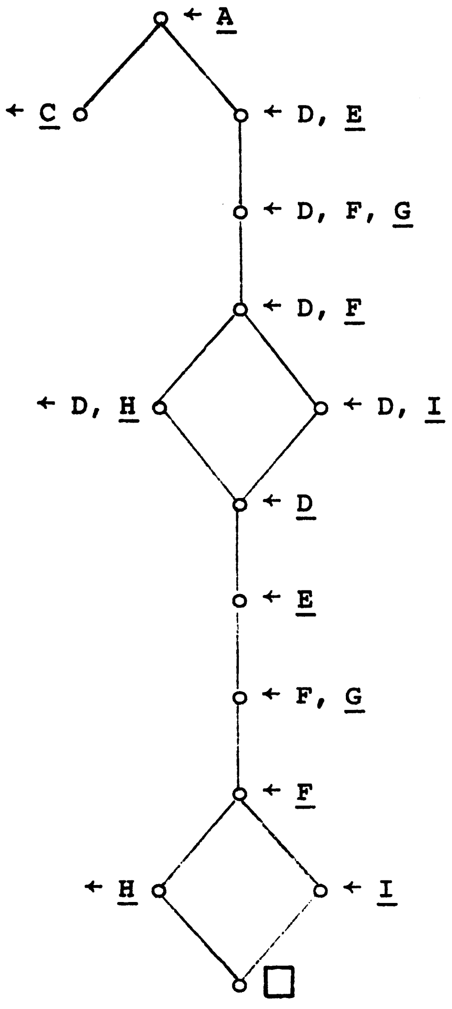


Figure 6. The graph-representation of the top-down search space of Figure 4.

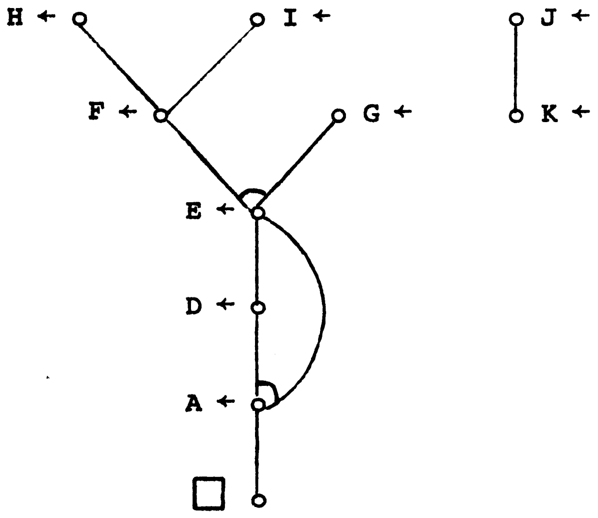


Figure 7. The graph-representation of the bottom-up search space of Figure 5.

Search strategies.

A proof procedure consists of an inference system and a search strategy. The inference system specifies, by means of axioms and rules of inference, the search space of all admissible derivations. The search strategy determines the sequence in which derivations in the search space are generated in the search for a refutation.

A search strategy can be depth-first, exhausting one line of argument before turning to another. It can be breadth-first, exploring all lines of argument simultaneously, in parallel. Or it can be merit-directed, generating at every stage a derivation of best merit, as determined by some procedure which partially orders derivations according to some notion of merit.

Search strategies can be autonomous procedures which generate derivations in a sequence determined by their own deliberations. Alternatively, they can execute domain-specific sequencing instructions formulated and conveyed to the proof procedure by the problem-poser.

The problem of designing effective search strategies will be investigated in more detail in later chapters.

Formal definitions of top-down and bottom-up derivations.

A detailed treatment of substitutions, matching and application of substitutions to expressions is contained in the next section.

Let S be a set of Horn clauses and let there be given a selection procedure which selects a unique atom from every occurrence of a goal statement. A sequence $C_1, \ldots, C_N$ of goal statements is a top-down derivation of C_N from S, with top-clause C_1 iff (1) and (2):

(1) $C_1 \in S$.

(2) For all $i < N$, the selected atom A_j in C_i

$\leftarrow A_1, \ldots, A_{j-1}, A_j, A_{j+1}, \ldots, A_n$

matches the atom A in the conclusion of some operator or assertion belonging to S,

$A \leftarrow B_1, \ldots, B_m$.

C_{i+1} is the new goal statement

$\leftarrow (A_1, \ldots, A_{j-1}, B_1, \ldots, B_m, A_{j+1}, \ldots, A_n)\ \theta$, where θ is the substitution which matches A_j and A. (The variables in $A \leftarrow B_1,\ldots,B_m$ are renamed so that they are distinct from all variables in C_i.)

A derivation of the null clause from S is a <u>top-down refutation</u> of S.

In general every atom in an occurrence of a goal statement is a candidate for selection. A selection procedure is of the <u>last-in-first-out</u> kind if in a derived goal statement only a most recently introduced atom is selected; i.e. in C_{i+1} the selected atom is one of

$B_1\theta,\ldots,B_m\theta$.

Let S be a set of Horn clauses. A set of assertions is a <u>bottom-up derivation</u> from S iff it is implied by (1) and (2):

(1) If $A \leftarrow$ belongs to S then

$\{A \leftarrow\}$

is a bottom-up derivation of $A \leftarrow$ from S.

(2) If, for $i \leq n$, D_i is a bottom-up derivation of $A_i \leftarrow$ from S and if

$A \leftarrow A_1',\ldots,A_n'$

is an operator in S such that A_1 matches A_1' and..

..and A_n matches A_n' simultaneously, with matching substitution θ, then

$D_1 \cup \ldots \cup D_n \cup \{A\theta \leftarrow\}$

is a bottom-up derivation of $A\theta \leftarrow$ from S. (The variables in all the clauses $A_1 \leftarrow,\ldots,A_n \leftarrow$ and $A \leftarrow A_1',\ldots,A_n'$ are renamed so that no clause shares variables with any of the others.)

In case (2) we allow the operator to be a goal statement $\leftarrow A_1',\ldots,A_n'$, in which case the new derivation $D_1 \cup \ldots \cup D_n \cup \{\square\}$ is a <u>bottom-up refutation</u> of S.

Notice that the set-theoretic representation of bottom-up derivations is closer to the graph-representation than it is to the tree-representation.

<u>Substitutions and matching</u>.

A <u>substitution</u> is a set of <u>substitution components</u> which are assignments of terms to variables:

$\{x_1 := t_1,\ x_2 := t_2,\ldots,x_n := t_n\}$.

No two components have the same variable. The <u>result</u> of applying a substitution θ to an expression X (term, atom, clause, set of expressions) is a new expression $X\theta$ which differs from X only in that it contains the term t_i wherever X contains x_i. If θ is the substitution

$$\{x_1 := t_1, \ldots, x_n := t_n\}$$

then the <u>product</u> $\theta\sigma$, where σ is also a substitution, is a new substitution:

$$\theta\sigma = \{x_1 := t_1\sigma, \ldots, x_n := t_n\sigma\} \cup \sigma'$$

where σ' is the subset of σ which affects variables different from those affected by θ.

A <u>unifier</u> of a set of expressions E is a substitution θ such that $E\theta$ contains exactly one element. A <u>most general unifier</u> of E is unifier of E such that, for every other unifier σ of E, there exists a substitution λ such that

$$\sigma = \theta\lambda \ .$$

If E contains two expressions, $E = \{A, A'\}$ and if θ is a most general unifier of E, then the two expressions are said to match and θ is a <u>matching</u> substitution. (Notice that all most general unifiers are essentially equivalent, in the sense that they differ from one another only in the different names for the variables they introduce into the expressions they are applied to.) There exist various unification algorithms which compute most general unifiers {49,51,58,2}.

A <u>simultaneous unifier</u> of a family $\varepsilon = \{E_1, \ldots, E_n\}$ of sets of expressions is a substitution θ which unifies each set E_i, i.e.

$$E_i\theta \text{ is a singleton, all } i \leq n.$$

A <u>most general simultaneous unifier</u> of ε is a unifier θ such that for all other unifiers σ there exists a substitution λ such that

$$\sigma = \theta\lambda$$

If each set in ε contains two expressions, $E_i = \{A_i, A_i'\}$ and if θ is a most general simultaneous unifier of ε, then the pairs A_i, A_i' are said to <u>match simultaneously</u> with <u>matching substitution</u> θ.

The most general simultaneous unifier θ of a family ε can be computed by successively unifying individual sets of expressions:

$\theta = \theta_1 \theta_2 \ldots \theta_n$, where

θ_1 is a most general unifier of E_1,

θ_{i+1} is a most general unifier of E_{i+1} [illegible] θ_i,

and $E_1, \ldots, E_n$ is an enumeration of the sets in ε.

Every enumeration of the members of ε gives rise to the same most general simultaneous unifier θ. This fact can usefully be applied in the generation of bottom-up derivations to obtain the new assertion

$$A\theta \leftarrow$$

from the old assertions $A_1 \leftarrow, \ldots, A_n \leftarrow$ using the operator $A \leftarrow A_1', \ldots, A_n'$. The most general sumultaneous unifier θ of the family

$$\{ \{A_1, A_1'\}, \ldots, \{A_n, A_n'\} \}$$

can be obtained by selecting an enumeration $A_1', \ldots, A_n'$ of the atoms in the operator and consecutively matching them with the assertions $A_i \leftarrow$. The new assertion of $A\theta \leftarrow$ is the last clause in the sequence of clauses

$$C_1, \ldots, C_N \; .$$

(1) C_1 is $A \leftarrow A_1', \ldots, A_n'$.

(2) For all $i < N$, C_i is of the form

$$B \leftarrow B_1, \ldots, B_{j-1}, \underline{B_j}, B_{j+1}, \ldots, B_m,$$

having a selected atom B_j. B_j matches the atom A_k in some assertion $A_k \leftarrow$.

C_{i+1} is

$$(B \leftarrow B_1, \ldots, B_{j-1}, B_{j+1}, \ldots, B_m)\theta_i$$

where θ_i is the substitution which matches B_j and A_k .

<u>Factorial example</u>.

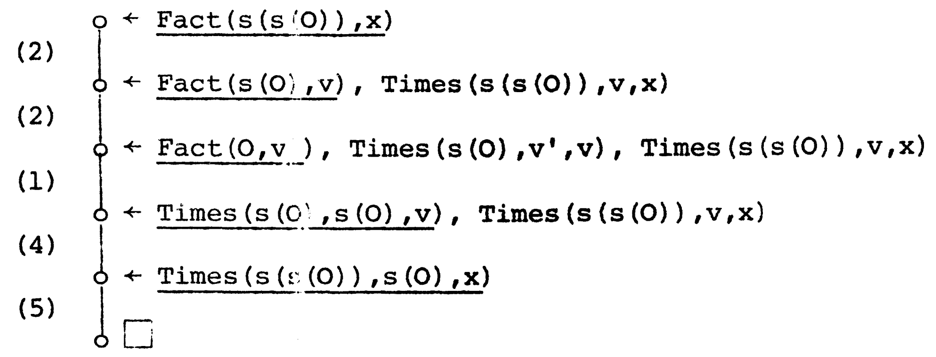
(1) Fact(O,s(O)) ←

(2) Fact(s(x),u) ← Fact(x,v), Times(s(x),v,u)

(3) ← Fact(s(s(O)),x)

(4) Times(s(O),s(O),s(O)) ←

(5) Times(s(s(O)),s(O),s(s(O))) ←



← Fact(s(s(O)),x)
(2)
← Fact(s(O),v), Times(s(s(O)),v,x)
(2)
← Fact(O,v'), Times(s(O),v',v), Times(s(s(O)),v,x)
(1)
← Times(s(O),s(O),v), Times(s(s(O)),v,x)
(4)
← Times(s(s(O)),s(O),x)
(5)
□

<u>Figure 8</u>. A search space of all top-down derivations for the factorial example.

Figure 8 illustrates the entire search space of all top-down derivations determined by a particular selection procedure for the clauses (1)-(5) of the factorial example. The single complete derivation in the search space can be regarded as a computation of the factorial of 2.

Figure 9 illustrates the entire search space of all bottom-up derivations determined by (1)-(5).

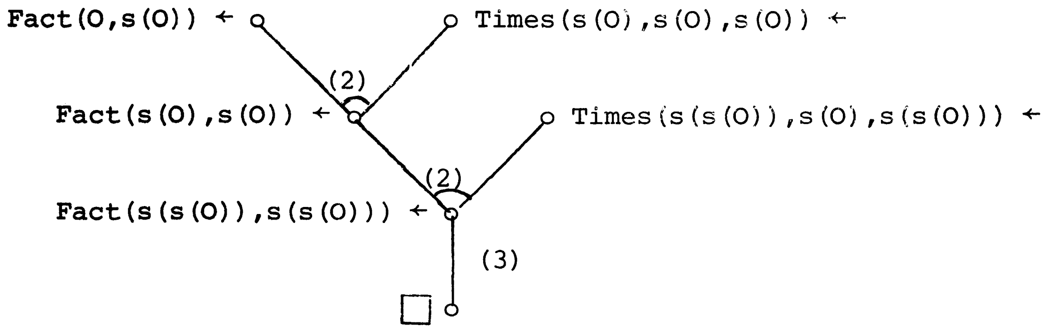


Figure 9. The search space of all bottom-up derivations for the factorial example.

Correctness and completeness.

An inference system is correct if every set of clauses which has a refutation is inconsistent. The inference system is complete if every inconsistent set of clauses has a refutation. The notions of correctness and completeness connect semantics with the proof theoretical part of syntax. A correct and complete inference system is one for which the notions of inconsistent set of clauses and refutable set coincide.

The correctness of both top-down and bottom-up inference systems is not difficult to verify.

The completeness of the bottom-up system has been proved by Robinson {50}. The system he proves complete is not limited to Horn clauses. The bottom-up system we consider is closer to the one investigated and proved complete in {21}.

Completeness for the top-down system has been proved, without the restriction to Horn clauses, but only for selection procedures which select on a last-in-first-out basis {30,26,48}. In addition, the systems proved complete all employ the additional factoring rule of inference which is defined in Chapter 6. It is not difficult to prove completeness for the top-down system as it has been defined in this Chapter: for Horn clauses, without factoring and with no constraints on the selection procedure.

Terminology.

The top-down interpretation of clauses is a version of modus tollens:

From $B \leftarrow A$ and $\leftarrow B$
infer $\leftarrow A$.

Bledsoe calls this backwards chaining{ 3}.

The bottom-up interpretation of clauses is a version of modus ponens:

From $B \leftarrow A$ and $A \leftarrow$
infer $B \leftarrow$.

Bledsoe calls this forwards chaining { 3} .

Both top-down and bottom-up inference are special cases of the resolution rule{ 49} defined in Chapter 6. Model elimination{ 28} , linear resolution { 29,32} , ordered linear resolution{ 48}, SL-resolution{ 26} and G-deduction{ 37} are top-down inference systems. Hyper-resolution{ 50} P_1-deduction{ 50} and M-clash resolution{ 53} are bottom-up systems. Kuehner's system{ 27} for Horn clauses combines top-down and bottom-up inference.

Among the top-down systems just mentioned, all except linear resolution employ a last-in-first-out selection procedure. The importance of relaxing this constraint on the selection procedure is illustrated by the admissible pairs example investigated in Chapter 5.

Linear resolution employs no selection procedure. Given a goal statement containing n atoms it potentially investigates the n! redundant sequences in which the atoms can be selected.

CHAPTER 3. ROBOT PLAN FORMATION AND THE FRAME PROBLEM.

The main problem in artificial intelligence today is to develop general languages and methods for representing knowledge satisfactorily within the computer. The problem of representation applies to

(1) the representation of factual knowledge about the world and general knowledge about laws governing physical relationships and change, and

(2) the representation of pragmatic knowledge necessary for effective problem-solving.

Adequate systems for the representation of knowledge are a prerequisite for problem-solving systems which combine knowledge about the world together with knowledge about problem-solving in order to solve problems. They are a prerequisite also for learning systems which construct their own representation of the world and develop their own problem-solving procedures. The failure of problem-solving and learning systems to perform satisfactorily can be attributed in large part to the inadequacy of the underlying representation system. The importance of representation has been argued by McCarthy {35} and by Minsky in the introduction to Semantic Information Processing {39}.

In this Chapter we investigate the application of predicate logic to the representation of factual knowledge and general knowledge about actions and change. For simplicity we deal with a one-agent universe. The robot plan-formation problem is to construct a sequence of actions which transforms an initial state into a goal state, given a description of the initial state of the world, of the goal state and of the set of actions which transform one state of the world into another.

In the next two chapters, dealing with the problem-reduction and procedural interpretations of predicate logic, we investigate the application of predicate logic to the representation of pragmatic knowledge about problem-solving procedures.

The use of predicate logic to represent the laws of change typically runs into the frame problem: how to state and to deal with the fact that almost all assertions which hold true of a given state of the world continue to hold true of the new state obtained by applying an action to the old state.

Failure to solve the frame problem has led many researchers to reject the use of predicate logic for robot plan-formation and to experiment with new systems (STRIPS {14}, PLANNER {18}). We shall argue that the first part of the frame problem is solved by the use of a suitable notation and the second part is solved by using the frame axiom top-down instead of bottom-up.

The robot plan-formation problem.

We assume that a description of the initial state of the world and of the properties desired of a goal state are given. The robot can perform various actions which transform one state of the world into another. Each action has associated

(1) preconditions which must hold true in a state in order for the action to be applicable to it,

(2) an add list of new assertions which hold true of the state obtained by applying the action, and

(3) a delete list of assertions which are the exceptions to the general rule that every assertion true in the old state remains true in the new state obtained by applying the action.

The problem is to find a sequence of actions which successively transforms the initial state through intermediate states into a goal state. (The explicit association of preconditions, add list and delete list with every action is due to STRIPS.)

The predicate logic representation of the robot plan-formation problem will be investigated for the simple example in Figure 10 {37}.

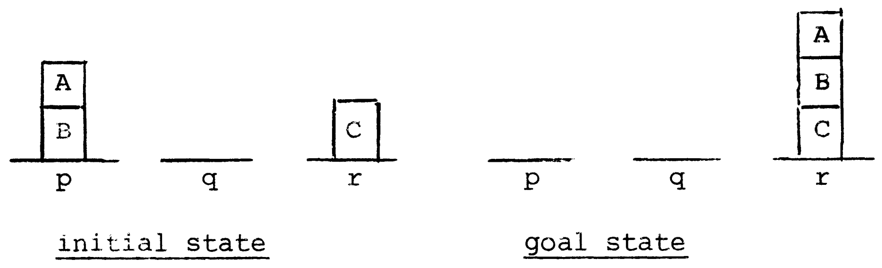


Figure 10. The initial and goal states for a robot plan-formation problem. There are three manipulatable objects A, B and C and three places (unmanipulatable objects) p, q and r. In the initial state, A is on B, B is on p and C is on r; A, q and C are clear. In the goal state, A is on B, B is on C and C is on r.

For each x,y and z there is an action, pickup(x,y,z), which allows the robot to pickup x from y and to put it down on z.

(1) The preconditions for the action
pickup(x,y,z) are that
x be manipulatable,
x be clear,
z be clear,
x be on y, and
x be different from z.

(2) The action adds the assertions that
x is on z, and
y is clear.

(3) The action deletes the assertions that
x is on y, and
z is clear.

The simplest solution of the problem is to

(1) pickup(A,B,q), then
(2) pickup(B,p,C), and finally
(3) pickup(A,q,B).

<u>**A predicate logic representation of the robot plan-formation problem.**</u>

Initial state O.	(1)	Poss(O) ←
	(2)	Holds(on(A,B),O) ←
	(3)	Holds(on(B,p),O) ←
	(4)	Holds(on(C,r),O) ←
	(5)	Holds(clear(A),O) ←
	(6)	Holds(clear(q),O) ←
	(7)	Holds(clear(C),O) ←
State-independent assertions.	(8)	Manip(A) ←
	(9)	Manip(B) ←
	(10)	Manip(C) ←
Goal state w.	(11)	← Holds(on(A,B),w), Holds(on(B,C),w), Holds(on(C,r),w), Poss(w)
State space.	(12)	Poss(do(x,w)) ← Poss(w), Pact(x,w)

<u>Preconditions</u>. (13) Pact(pickup(x,y,z),w) ← Manip(x)
Holds(clear(x),w),
Holds(clear(z),w),
Holds(on(x,y),w),
Diff(x,z)

<u>Add list</u>. (14) Holds(on(x,z), do(pickup(x,y,z),w)) ←

(15) Holds(clear(y), do(pickup(x,y,z),w)) ←

<u>Delete list to frame axiom</u>. (16) Holds(u, do(pickup(x,y,z),w)) ← Holds(u,w),
Diff(u,on(x,y)),
Diff(u,clear(z)).

The Diff relation holds between two variable-free terms s and t if s and t are syntactically distinct. It is useful to imagine that clauses (1)-(16) are supplemented by infinitely many clauses of the form

Diff(s,t) ←

for every pair of terms which are not unifiable. Equivalently we might imagine (1)-(16) supplemented by the finitely many clauses

$\mathrm{Diff}(f(x_1,\ldots,x_n),\ g(y_1,\ldots,y_m)) \leftarrow$

$\mathrm{Diff}(f(x_1,\ldots,x_n),\ f(y_1,\ldots,y_n)) \leftarrow \mathrm{Diff}(x_i,y_i)$

for every pair of distinct function symbols f and g in case of clauses of the first kind and for every n-ary function symbol f and every index $i \leq n$ in case of clauses of the second kind. In practice the clauses defining the Diff relation would be used exclusively in a top-down manner and would be compiled. In other words, every occurrence of an atom Diff(s,t) in the hypothesis of a clause would be treated as a procedure call to a procedure written in an ordinary programming language.

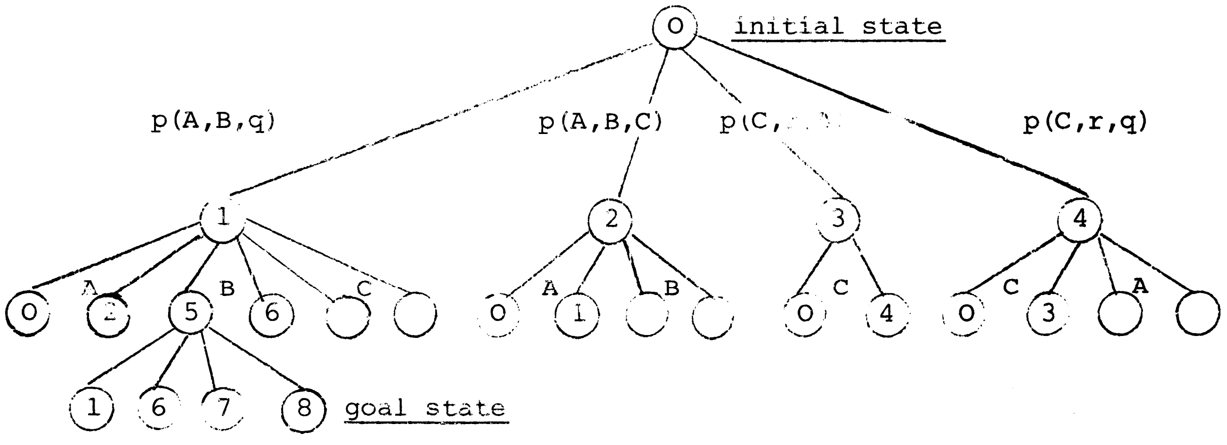
Instead of writing On(x,y,w) and Clear(x,w) we treat "On" and "Clear" as function symbols and write Holds(on(x,y),w) and Holds(clear(x),w). This notational device solves the first part of the frame problem by allowing the use of just a single general frame axiom instead of individual frame axioms for each assertion which is preserved by the application of an action.

The ability to use variables which range over assertions and sentences is provided by higher-order logic. The syntactic device of replacing predicate symbols by function symbols gives first-order logic the ability to simulate this feature of higher-order logic.

<u>Top-down and bottom-up interpretations of the state space axiom (12).</u>

Clauses (1)-(16) are neutral with respect to the top-down and bottom-up interpretations. Not only may the entire set of clauses be interpreted top-down or bottom-up, but different clauses may be interpreted differently. One clause may be executed top-down and another bottom-up. Even within a single clause different atoms might be executed in opposite directions.

If clause (12) is executed bottom-up then it is used to derive that a new state do(x,w) is possible given assertions that the old state w is possible and that the action x can be applied in state w. Consistent bottom-up execution of (12) begins with the initial state, applies actions to produce new states from old states and terminates when it generates a state which satisfies the goal state description. Figure 11 illustrates part of the search space of all states generated by executing (12) bottom-up.



<u>Figure 11</u>. Part of the search space of all states obtained by executing clause (12) bottom-up. Here p(x,y,z) abbreviates pickup(x,y,z). The letter A, B or C between two arcs in the space indicates the object picked up in obtaining the two states at the bottom of the arcs. Distinct states are associated with distinct nodes. However states labelled by the same number are isomorphic in the sense that they are characterised by the same set of assertions.

If clause (12) is executed top-down then it is used to replace the problem of showing that a new state do(x,w) is possible by the subproblems of showing that the old state w is possible and that the action x can be applied in w. Consistent top-down execution of (12) begins with the goal state description and terminates when it derives a new goal state description which is satisfied by the initial state. In fact the precise behaviour effected by top-down execution of (12) depends on details about the selection procedure and about the direction of execution of other clauses. Figures 15 and 16 below illustrate part of a search space of all goal statements obtained by executing all of the clauses (1)-(16) top-down.

An interesting combination of execution strategies is obtained when clause (12) is activated by the bottom-up execution of the atom Poss(w) followed by the top-down execution of Pact(x,w). Interpreted in this manner, clause (12) is used, when given an assertion that a state w is possible, to derive that the new state do(x,w) is possible by testing first that the action x can be performed in w. Such an interpretation of clause (12) together with a top-down interpretation of all other clauses is illustrated in Figures 13 and 14 below.

<u>The frame problem and execution strategies for the frame axiom (16)</u>.

The second part of the frame problem arises when the frame axiom (16) is executed bottom-up in order to derive, from the assertion that u holds in state w, the new assertion that u continues to hold in the state do(x,w). By bottom-up interpretation of (16) we mean more precisely that (16) is activated by the bottom-up execution of the atom Holds(u,w) followed by top-down execution of the atoms Diff(u,on(x,y)) and Diff(u,clear(z)). Otherwise if all atoms in the hypothesis of (16) were executed bottom-up then the search space of all derivable clauses would include all assertions of the form Diff(s,t) ← for all pairs of terms s,t which do not unify.

In more realistic problems than that involved in our three-block example, states of the world have a complex structure which can be described only by means of a very large number of assertions. In such problems, bottom-up interpretation of the frame axiom leads to generation of an intolerable number of assertions about derived states of the world.

Both PLANNER and STRIPS attempt to solve the frame problem by abandoning the frame axiom and by using instead special procedures to determine whether a fact holds true in a given state:

To determine whether u holds in do(x,w):

(1) Check whether u belongs to the add list of x. If it does, return <u>success</u>.

(2) Otherwise, check whether u belongs to the delete list of x. If it does return <u>failure</u>.

(3) Otherwise, return the result of determining whether u holds in w.

But this sequence of steps is identical to that involved in running the frame axiom (16) and the add list (14),(15) axioms top-down, trying the add list before the frame axiom and selecting the atoms Diff(u,on(x,y)) and Diff(u,clear(z)) before the atom Holds(u,w) in the hypothesis of the frame axiom.

<u>Bottom-up execution of (1)-(16).</u>

Figure 12 illustrates a small part of the search space determined by bottom-up execution of clauses (1)-(16). (Even in this example all atoms having predicate symbol Diff are activated top-down.) Only those assertions are demonstrated which concern states belonging to the solution path. In general a search strategy would generate many assertions which refer to states not belonging to the solution path.

Pact(p(A,B,q),O) ←	Pact(p(B,p,C),1) ←	Pact(p(A,q,B),5) ←
Holds(on(B,p),1) ←	Holds(on(A,q),5) ←	Holds(on(B,C),8) ←
Holds(on(C,r),1) ←	Holds(on(C,r),5) ←	Holds(on(C,r),8) ←
Holds(clear(A),1) ←	Holds(clear(A),5) ←	Holds(clear(A),8) ←
Holds(clear(C),1) ←	Holds(clear(B),5) ←	Holds(clear(p),8) ←
Poss(1) ←	Poss(5) ←	Poss(8) ←

<u>Figure 12.</u> Part of the search space of assertions determined by executing (1)-(16) bottom-up. As in Figure 11, p(x,y,z) abbreviates pickup(x,y,z).
1 abbreviates do(pickup(A,B,q),O).
5 abbreviates do(pickup(B,p,C),1).
8 abbreviates do(pickup(A,q,B),5).

Assertions such as

Holds(on(A,q),1) ←	Holds(on(B,C),5) ←	Holds(on(A,B),8) ←
Holds(clear(B),1) ←	Holds(clear(p),5) ←	Holds(clear(q),8) ←

are not included in Figure 12 because they are special instances of axioms in the add list (14) and (15).

Bottom-up execution of (12) only.

Figures 13 and 14 illustrate a small part of the search space determined by top-down interpretation of all clauses except (12) which is used bottom-up to derive new states from old ones. The selection of atoms is determined by the objective of minimising the generation of alternative branches in the search space. This criterion of selection is elaborated upon and discussed in the next chapter, concerned with the problem-reduction interpretation of predicate logic.

Notice that the solution refutation of Figures 13 and 14 contains many subderivations consisting of consecutive steps which have no alternatives or else have alternatives which fail in a very few steps. The alternatives which do not fail correspond to genuinely alternative actions generating new states in the state space illustrated in Figure 11.

Notice that top-down execution of the frame axiom is more complicated than has been suggested in the earlier discussion of the frame problem. The complication is that the frame axiom can be used not only to determine whether a known fact u holds in a known state do(x,w), but it can also be used to generate facts which hold in a known state or to generate states in which a known fact holds. More generally, the frame axiom can usefully be applied in situations where u and do(x,w) are either partially known or totally unknown. This aspect of the behaviour of clauses is connected with the lack of input-output distinction in predicate logic programs. It is investigated in Chapter 5 which deals with the procedural interpretation of predicate logic.

The mixed top-down, bottom-up interpretation of clauses in this example can be simulated by a proof procedure which interprets all clauses top-down: rewrite clauses (1),(11) and (12). Use the fact that

C, not—A ← B is equivalent to
C ← A,B and
C ← not—A,B is equivalent to
C, A ← B.

Write NPoss(t) instead of not-Poss(t). Clauses (1),(11) and (12) become (1'),(11') and (12') respectively:

(1') ← NPoss(O)
(11') NPoss(w) ← Holds(on(A,B),w),Holds(on(B,C),w),Holds(on(C,r),w)
(12') NPoss(w) ← NPoss(do(x,w)),Pact(x,w).

Top-down interpretation of the new set of clauses is equivalent to the mixed interpretation illustrated in Figures 13 and 14. The kind of renaming involved in rewriting clauses (1),(11) and (12) has been investigated by Meltzer{ 36}.

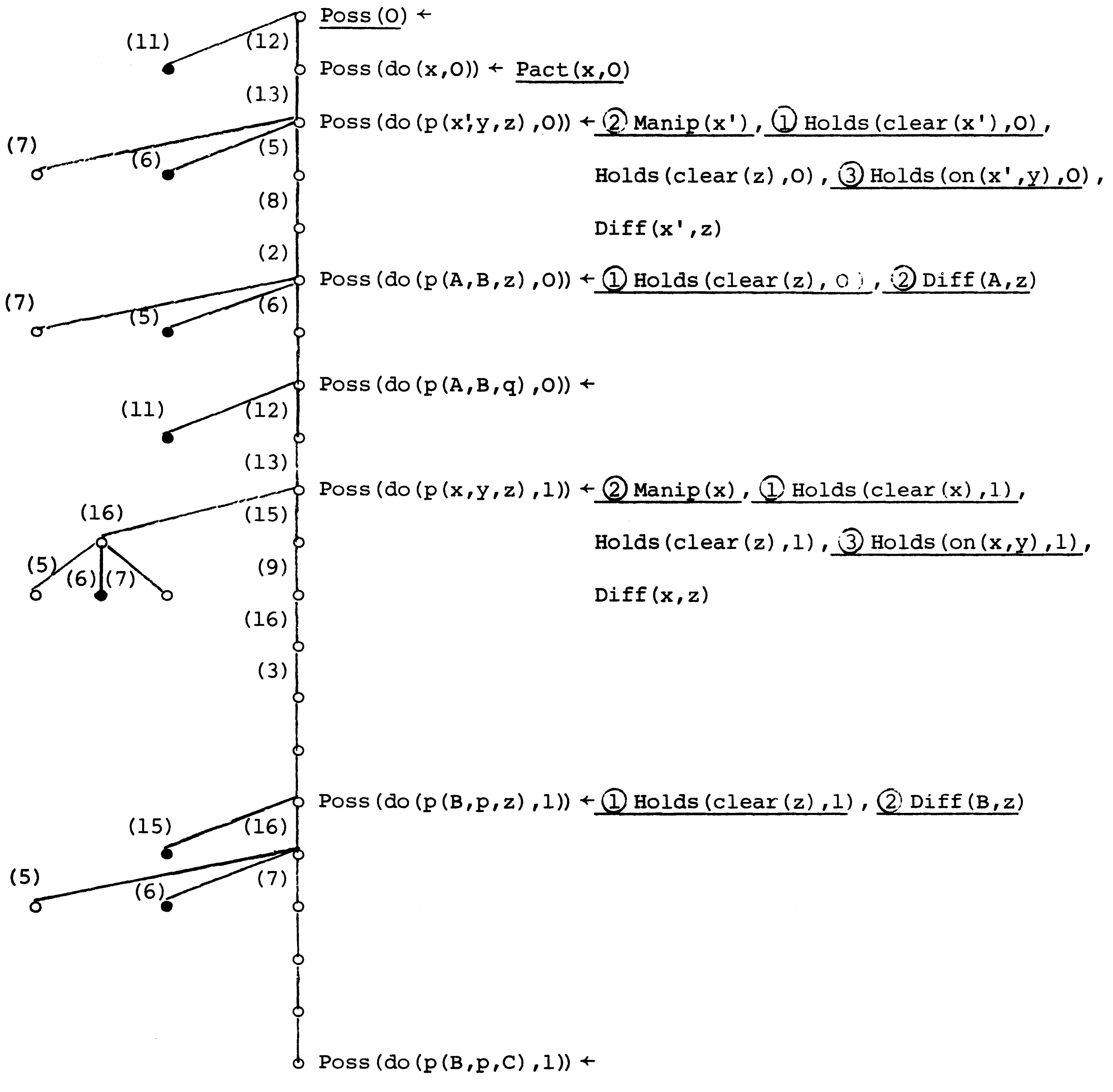
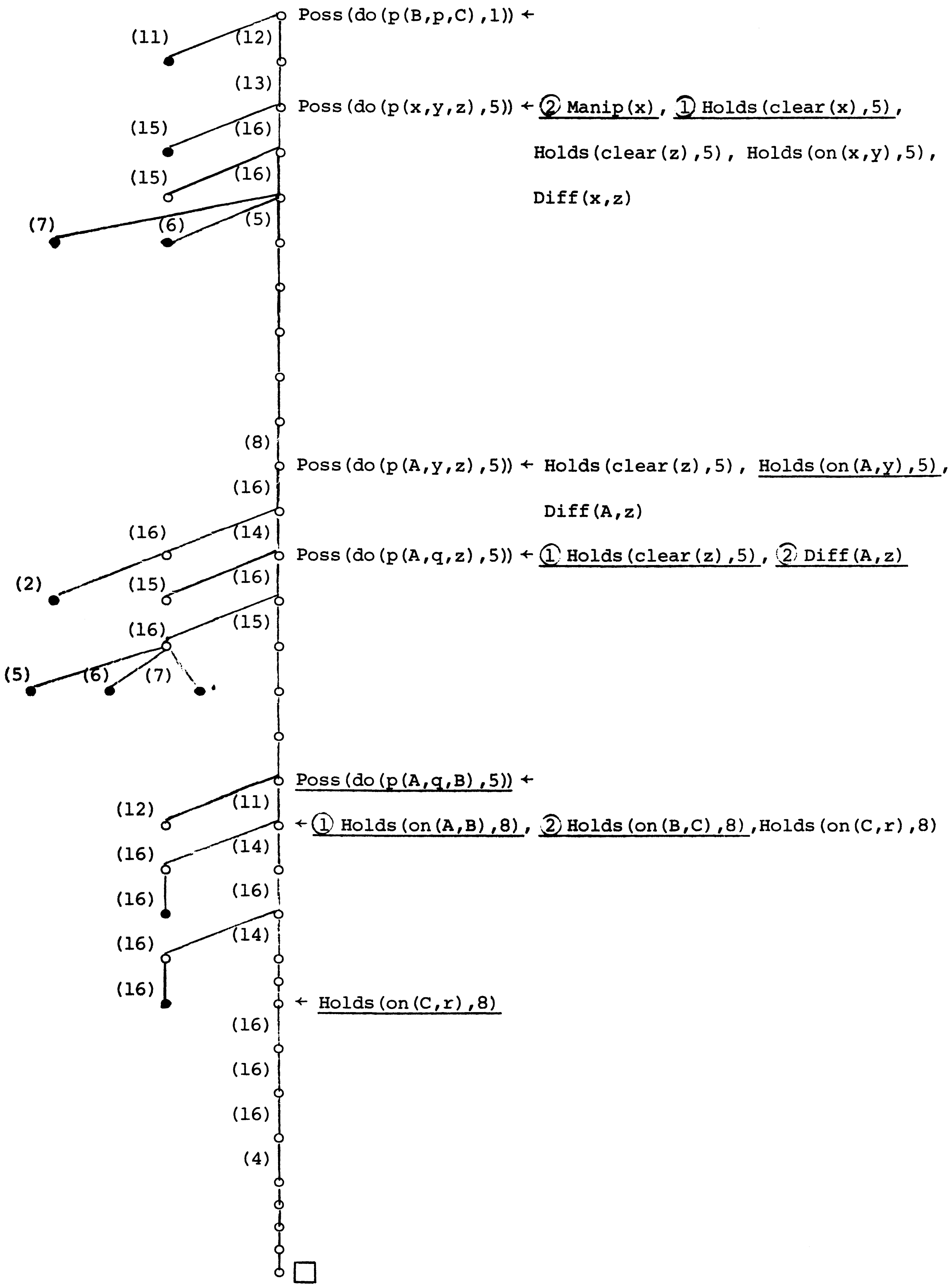

Poss(O) ←
(11)
(12)
Poss(do(x,O)) ← Pact(x,O)
(13)
Poss(do(p(x',y,z),O)) ← ② Manip(x'), ① Holds(clear(x'),O), Holds(clear(z),O), ③ Holds(on(x',y),O), Diff(x',z)
(7)
(6)
(5)
(8)
(2)
Poss(do(p(A,B,z),O)) ← ① Holds(clear(z), O), ② Diff(A,z)
(7)
(5)
(6)
Poss(do(p(A,B,q),O)) ←
(11)
(12)
(13)
Poss(do(p(x,y,z),1)) ← ② Manip(x), ① Holds(clear(x),1), Holds(clear(z),1), ③ Holds(on(x,y),1), Diff(x,z)
(16)
(15)
(5)
(6)
(7)
(9)
(16)
(3)
Poss(do(p(B,p,z),1)) ← ① Holds(clear(z),1), ② Diff(B,z)
(15)
(16)
(5)
(6)
(7)
Poss(do(p(B,p,C),1)) ←

Figure 13. The initial part of a refutation determined by bottom-up execution of clause (12). All branches away from the solution path are illustrated. Darkened nodes are terminal nodes containing a selected atom which matches no atom on the opposite side of an arrow in an input clause. The numbers preceding underlined atoms indicate the relative order in which they or their descendants are selected. Unlabelled arcs denote responses to activation of an atom containing the predicate symbol Diff. Nodes are unlabelled in order to suppress distracting details.

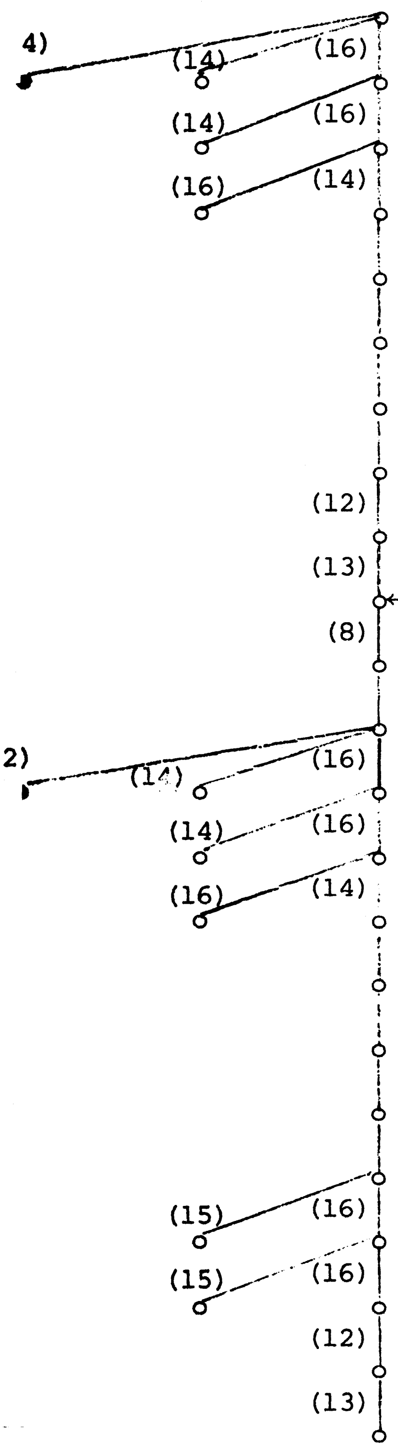


Poss(do(p(B,p,C),1)) ←
(11)
(12)
(13)
Poss(do(p(x,y,z),5)) ← ② Manip(x), ① Holds(clear(x),5), Holds(clear(z),5), Holds(on(x,y),5), Diff(x,z)
(15)
(16)
(15)
(16)
(7)
(6)
(5)
(8)
Poss(do(p(A,y,z),5)) ← Holds(clear(z),5), Holds(on(A,y),5), Diff(A,z)
(16)
(16)
(14)
Poss(do(p(A,q,z),5)) ← ① Holds(clear(z),5), ② Diff(A,z)
(2)
(15)
(16)
(16)
(15)
(5)
(6)
(7)
Poss(do(p(A,q,B),5)) ←
(12)
(11)
← ① Holds(on(A,B),8), ② Holds(on(B,C),8), Holds(on(C,r),8)
(16)
(14)
(16)
(16)
(16)
(14)
(16)
← Holds(on(C,r),8)
(16)
(16)
(16)
(4)

Figure 14. The remaining part of the refutation whose initial part is illustrated in Figure 13.

Top-down execution of (1)-(16).

Figures 15 and 16 illustrate a top-down refutation of clauses (1)-(16).



← ③ Holds(on(A,B),w), ② Holds(on(B,C),w), ① Holds(on(C,r),w), Poss(w)

← Holds(on(B,C),w'), ① Diff(on(B,C),on(A,y)), ② Diff(on(B,C),clear(B)),

Holds(on(C,r),w'), ③ Diff(on(C,r),on(A,y)), ④ Diff(on(C,r),clear(B)),

⑤ Poss(do(p(A,y,B),w'))

← ⑤ Holds(on(B,C),w'), ④ Holds(on(C,r),w'), ① Manip(A),

Holds(clear(A),w'), Holds(clear(B),w'),

③ Holds(on(A,y),w'), ② Diff(A,B), Poss(w')

← Holds(on(C,r),w"), ① Diff(on(C,r),on(B,y')), ② Diff(on(C,r),clear(C)),

⑤ Holds(clear(A), do(p(B,y',C),w")),

⑥ Holds(clear(B), do(p(B,y',C),w")),

Holds(on(A,y),w"), ③ Diff(on(A,y),on(B,y')), ④ Diff(on(A,y),clear(C)),

⑦ Poss(do(p(B,y',C),w"))

← Holds(on(C,r),w"), Holds(clear(A),w"), ① Diff(clear(A),on(B,y')),

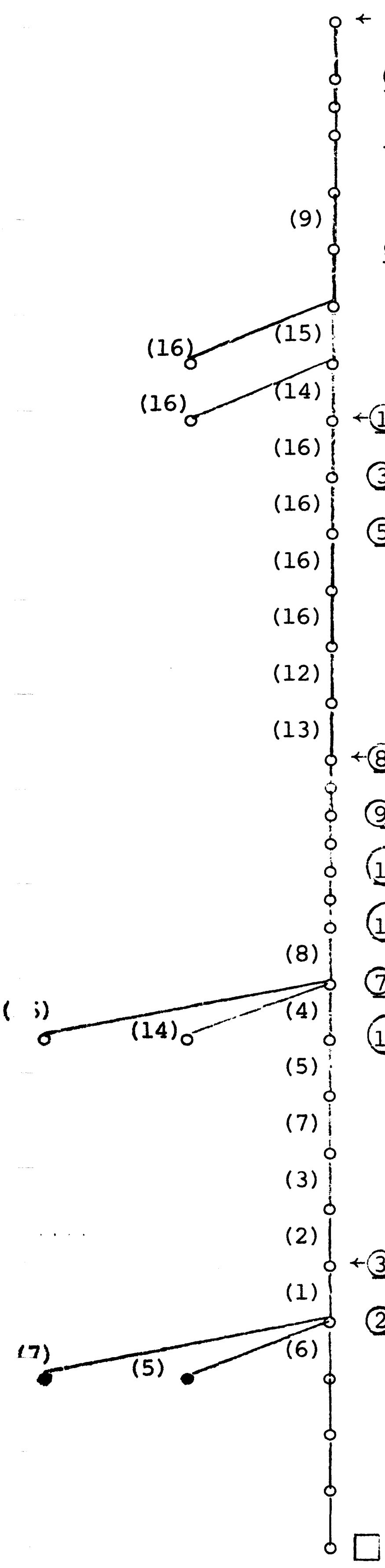
② Diff(clear(A),clear(C)), ⑦ Holds(clear(B),w"), ③ Diff(clear(B),on(B,y')),

④ Diff(clear(B),clear(C)), ⑧ Holds(on(A,y),w"), ⑤ Manip(B),

~~Holds(clear(B),w")~~, Holds(clear(C),w"), Holds(on(B,y'),w"),

⑥ Diff(B,C), Poss(w")

Figure 15. The initial part of a top-down refutation of clauses (1)-(16). The atom **Holds(clear(B),w")** is deleted because it is identical to another atom in the same goal statement.



← Holds(on(C,r),w"),Holds(clear(A),w"),①Diff(clear(A),on(B,y')),
②Diff(clear(A),clear(C)),⑦Holds(clear(B),w"),③Diff(clear(B),on(B,y')),
④Diff(clear(B),clear(C)),⑧Holds(on(A,y),w"),⑤Manip(B),
~~Holds(clear(B),w")~~,Holds(clear(C),w"),Holds(on(B,y'),w"),
⑥Diff(B,C),Poss(w")

←①Holds(on(C,r),do(p(A,B,y),w''')),②Holds(clear(A),do(p(A,B,y),w''')),
③Holds(clear(C),do(p(A,B,y),w''')),④Holds(on(B,y'),do(p(A,B,y),w''')),
⑤Poss(do(p(A,B,y),w'''))

←⑧Holds(on(C,r),w'''),①Diff(on(C,r),on(A,B)),②Diff(on(C,r),clear(y)),
⑨Holds(clear(A),w'''),③Diff(clear(A),on(A,B)),Diff(clear(A),clear(y)),
⑩Holds(clear(C),w'''),④Diff(clear(C),on(A,B)),Diff(clear(C),clear(y)),
⑪Holds(on(B,y'),w''')⑤Diff(on(B,y'),on(A,B)),⑥Diff(on(B,y'),clear(y)),
⑦Manip(A),~~Holds(clear(A),w''')~~,Holds(clear(y),w'''),
⑫Holds(on(A,B),w'''), Diff(A,y),Poss(w''')

←③Diff(clear(A),clear(y)),④Diff(clear(C),clear(y)),
②Holds(clear(y),0),⑤Diff(A,y),①Poss(0)

Figure 16. The remaining part of the top-down refutation whose initial part is in Figure 15.

Non Horn clauses and conditional plans.

The following example is a variation of the Robert-is-always-working example.

In the initial state Robert is either at work or at home depending on whether he is healthy or ill. The goal is to get Robert to the circus. The only action available is go(x,y,z) which allows x to go from y to z. The solution is to construct the conditional plan:

```
If Robert is healthy
     then go(Robert,work,circus)
If Robert is ill
     then go(Robert,home,circus).
```

Figure 17 illustrates a top-down solution together with part of the search space. A theorem-proving system which extends the top-down and bottom-up interpretations to non Horn clauses and simulates the refutation and search space of Figure 17 is described in Chapter 6.

Initial state O.	(1) Poss(O) ←
	(2) Holds(Healthy(Robert),O),Holds(Ill(Robert),O) ←
State-independent laws.	(3) Holds(at(Robert,work),w) ← Holds(Healthy(Robert),w)
	(4) Holds(at(Robert,home),w) ← Holds(Ill(Robert),w)
Goal state w.	(5) ← Holds(at(Robert,circus),w),Poss(w)
State space.	(6) Poss(do(x,w)) ← Poss(w),Pact(x,w)
Precondition.	(7) Pact(go(x,y,z),w) ← Holds(at(x,y),w)
Add list.	(8) Holds(at(x,z),do(go(x,y,z),w))←
Delete list and frame axiom.	(9) Holds(u,do(go(x,y,z),w)) ← Holds(u,w),Diff(u,at(x,y))

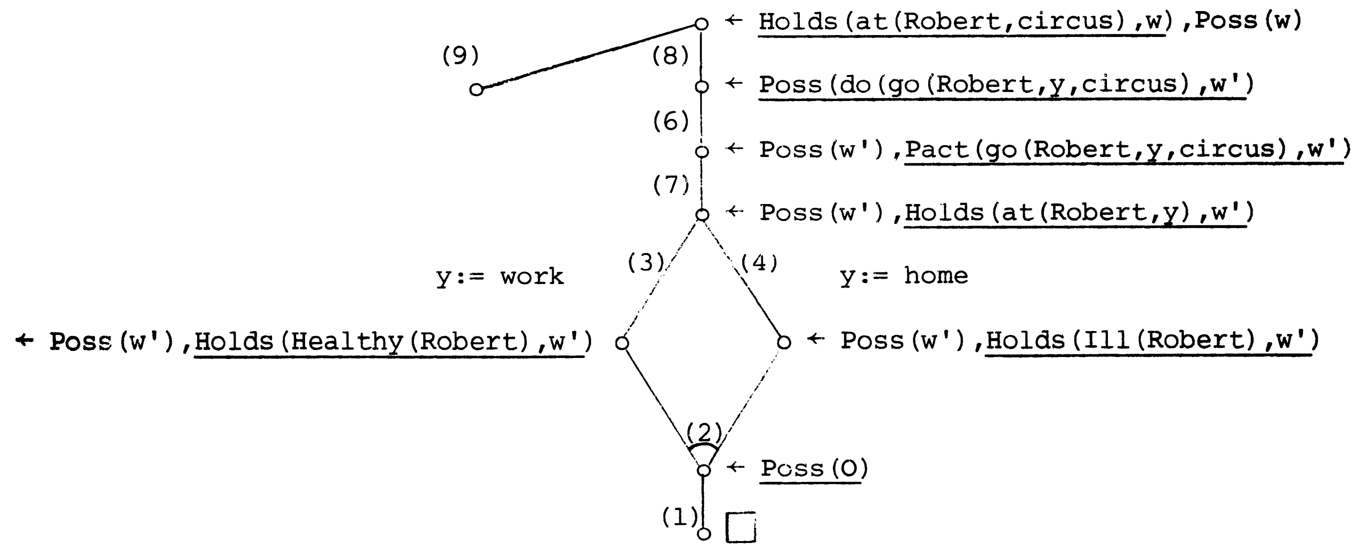

← Holds(at(Robert,circus),w),Poss(w)
(9)
(8)
← Poss(do(go(Robert,y,circus),w')
(6)
← Poss(w'),Pact(go(Robert,y,circus),w')
(7)
← Poss(w'),Holds(at(Robert,y),w')
y:= work
(3)
(4)
y:= home
← Poss(w'),Holds(Healthy(Robert),w')
← Poss(w'),Holds(Ill(Robert),w')
(2)
← Poss(O)
(1)
□

Figure 17. A top-down solution of the problem of getting Robert to the circus.

CHAPTER 4. THE PROBLEM-REDUCTION INTERPRETATION OF PREDICATE LOGIC.

Problem-reduction is a method of problem-solving which has often been applied explicitly or implicitly in Artificial Intelligence {42} and other disciplines. We shall argue that the problem reduction model used in Artificial Intelligence, the reduction of problems to independent subproblems, corresponds to the top-down execution of variable-free Horn clauses. Compared with proof procedures for predicate logic, such a model of problem-solving is inadequate for three reasons:

(1) Effective problem-solving involves the reduction of problems to dependent subproblems. The compatibility required of solutions to dependent subproblems corresponds to the presence of common variables in distinct atoms of a goal statement.

(2) Problems and subproblems generally have both hypotheses and conclusions. In such situations a useful problem-solving strategy is to combine both bottom-up interpretation starting with the hypotheses of the problem and top-down interpretation starting with the conclusion. The association of hypotheses with subproblems involves the use of non Horn clauses.

(3) Problem-reduction often needs to be supplemented by case analysis. The solution of a problem may require the cooperation of several problem-solving methods, each of which solves the problem in a special case. The various methods solve the problem cooperatively when between them they exhaust all possible cases. In predicate logic,case analysis is dealt with by means of non Horn clauses.

The problem-reduction interpretation of predicate logic was observed by Kowalski and Kuehner {26} and was investigated in more detail by Loveland and Stickel {31}. Other authors {12, 54 } have investigated the application of problem-reduction methods to the solution of goal statements containing atoms with distinct variables.

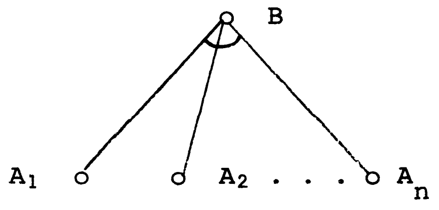
The and-or tree representation of problem-reduction.

In the problem-reduction model of Artificial Intelligence the task is to find a solution to an initially given problem, using a given set of operators which reduce problems to (independent) subproblems and a given set of initially solved subproblems. The task is accomplished by repeatedly applying operators to unsolved subproblems, replacing them by other subproblems, terminating successfully when the initial problem has been replaced by a set of initially solved subproblems.

In the <u>and-or tree representation</u> of a problem-reduction task, every node is labelled by a problem:

(1) The root node is labelled by the initial problem.

(2) If a problem B labels a node and if an operator reduces B to subproblems $A_1,\ldots,A_n$ then the node is connected by a directed arc to each one of n successor nodes labelled by the individual problems $A_1,\ldots,A_n$



The set of n successor nodes is said to be a <u>bundle</u> associated with the given operator. Successors are organised into bundles in order to distinguish whether different successors of the same node are determined by the same operator or by different operators.

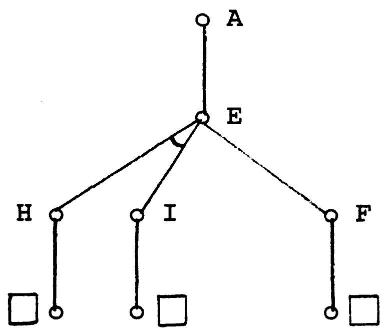
(3) If the problem labelling a node is an initially solved problem then it has a bundle containing one successor labelled by the mark of success $\square$.

A finite subtree of an and-or tree is a <u>solution</u> if

(1) it contains the root node and

(2) whenever it contains a node, not labelled by $\square$, it contains a single bundle of successor nodes.

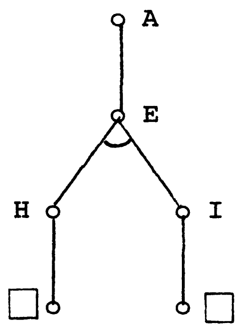
Figure 18 illustrates the and-or tree representation for a simple problem-reduction task. The same figure contains the variables-free Horn clause representation and the two distinct solutions of the initial problem.



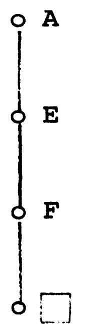
and-or tree representation

Initial problem :	$\leftarrow$ A
Operators :	A $\leftarrow$ E
	E $\leftarrow$ H, I
	E $\leftarrow$ F
Initially solved problems :	H $\leftarrow$
	I $\leftarrow$
	F $\leftarrow$

variable-free Horn clause representation



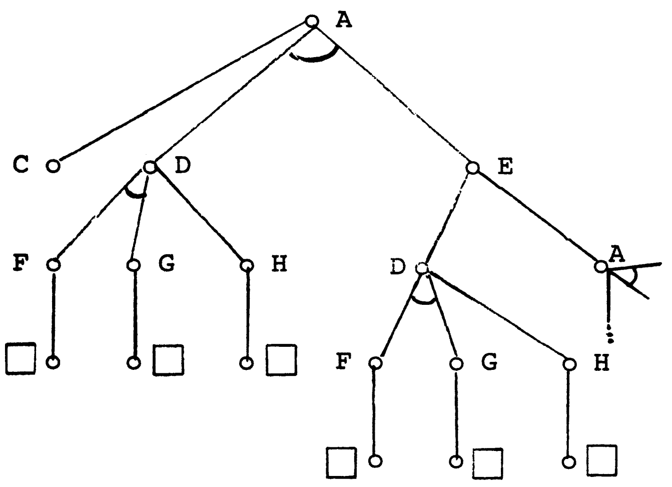
one solution



the other solution

Figure 18. The and-or tree and predicate logic representations of a simple problem-reduction task.

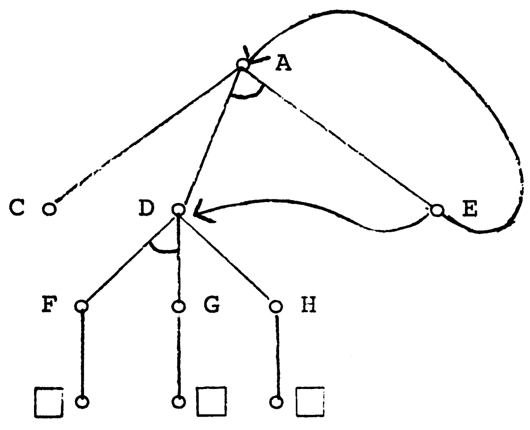
The and-or graph representation is obtained from the and-or tree by identifying nodes labelled by the same problem. Figure 19 illustrates both the and-or tree and the and-or graph representations of the same problem-reduction task.



and-or tree representation

Initial problem :	$\leftarrow$ A
Operators :	A $\leftarrow$ C
	A $\leftarrow$ D, E
	D $\leftarrow$ F, G
	D $\leftarrow$ H
	E $\leftarrow$ D
	E $\leftarrow$ A
Initially solved problems :	F $\leftarrow$
	G $\leftarrow$
	H $\leftarrow$

variable-free Horn clause representation



and-or graph representation

Figure 19. The and-or tree, and-or graph and predicate logic representations of the same problem-reduction task.

The problem-reduction interpretation of Horn clauses.

(1) Interpret a goal statement

$\leftarrow A_1, \ldots, A_n$

containing variables $x_1, \ldots, x_k$

as a command :

Find $x_1 \ldots$ and x_k which solve

the problems $A_1 \ldots$ and A_n.

Any substitution of terms for variables which solves $A_1 \ldots$ and A_n is a solution of the goal statement.

(2) Interpret an <u>assertion</u>

$B \leftarrow$

as a <u>solved problem</u>. It solves with <u>solution</u> θ only problems A which match B with matching substitution θ.

(3) Interpret an <u>operator</u>

$B \leftarrow A_1, \ldots, A_n$

as a <u>solution method</u>. It reduces the solution of problems A which match B (with matching substitution θ) to the solution of the goal statement

$\leftarrow A_1\theta, \ldots, A_n\theta$.

If θ' is a solution of the goal statement then $\theta\theta'$ is a <u>solution</u> of A.

(4) Interpret the <u>null clause</u> as an empty set of problems and therefore as a mark of <u>success</u>.

The problem-reduction interpretation of Horn clauses is basically a top-down interpretation. It differs, however, from the top-down interpretation defined earlier in that it does not specify how the solution of goal statements is to be related to the solution of individual problems. In particular the problem-reduction interpretation leaves open the possibility that a goal statement

$\leftarrow A_1, \ldots, A_n$

might be solved by

(1) independently solving the individual subproblems A_1...and A_n, obtaining individual solutions θ_1...and θ_n, and then

(2) finding a most general substitution θ such that

$\theta_1\theta = \theta_2\theta = \ldots = \theta_n\theta$.

The most general common solution $\theta_i\theta$ of the individual subproblems A_i is a <u>solution</u> of the goal statement. Such a method of solving goal statements is most useful when the individual subproblems are independent (i.e. share no variables). Then the most general common solution always exists and is the union $\theta_1 \cup \ldots \cup \theta_n$ of the individual solutions $\theta_1, \ldots, \theta_n$. In the general case this solution method suggests a notion of generalised and-or tree which deals with the dependencies between subproblems.

Figures 20 and 21 illustrate generalised and-or trees for the fallible Greek and the parsing problems. Each bundle of successor nodes is labelled by the output component of the substitution which matches the problem with the atom in the conclusion of the operator or solved problem which generates the bundle. The output component of a substitution is the part which affects only variables in the problem being solved.

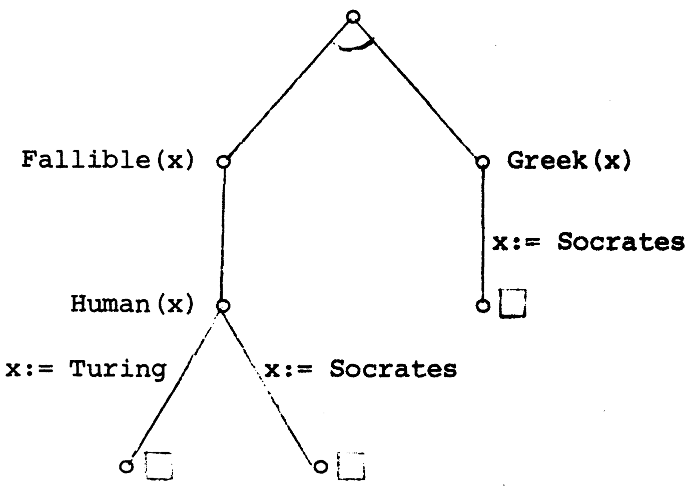


Figure 20. The generalised and-or tree, problem-reduction interpretation of the fallible Greek problem.

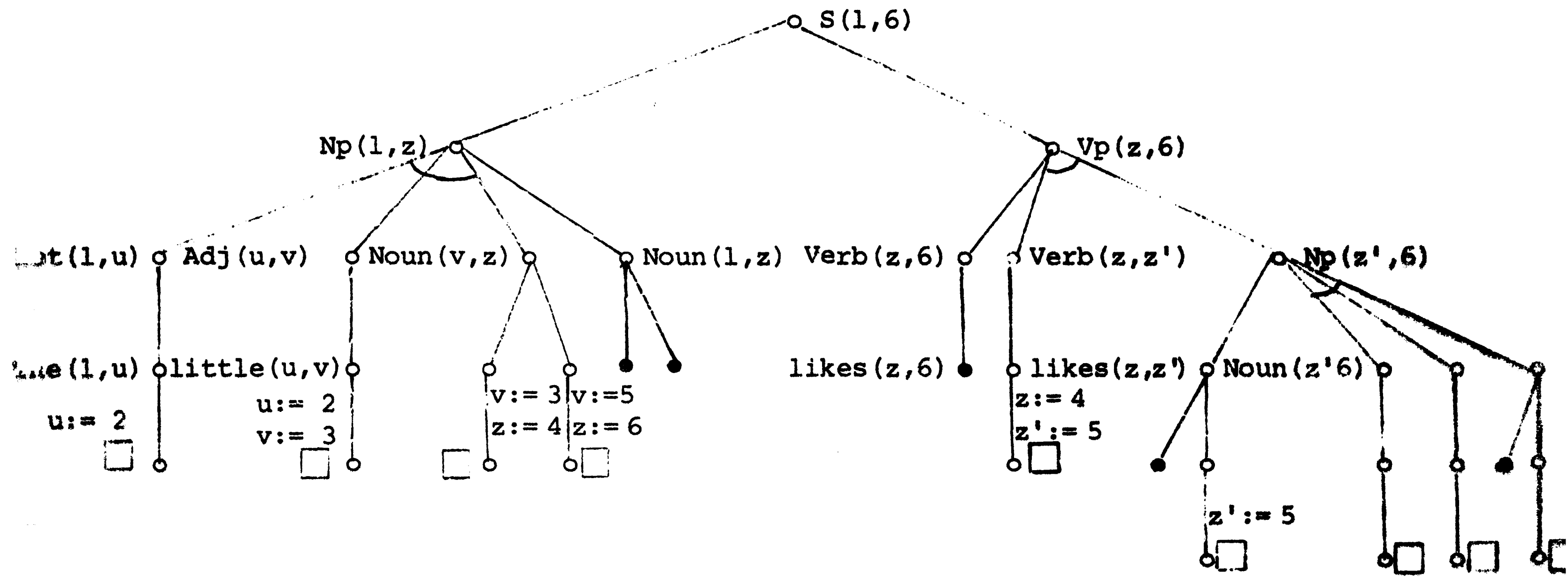
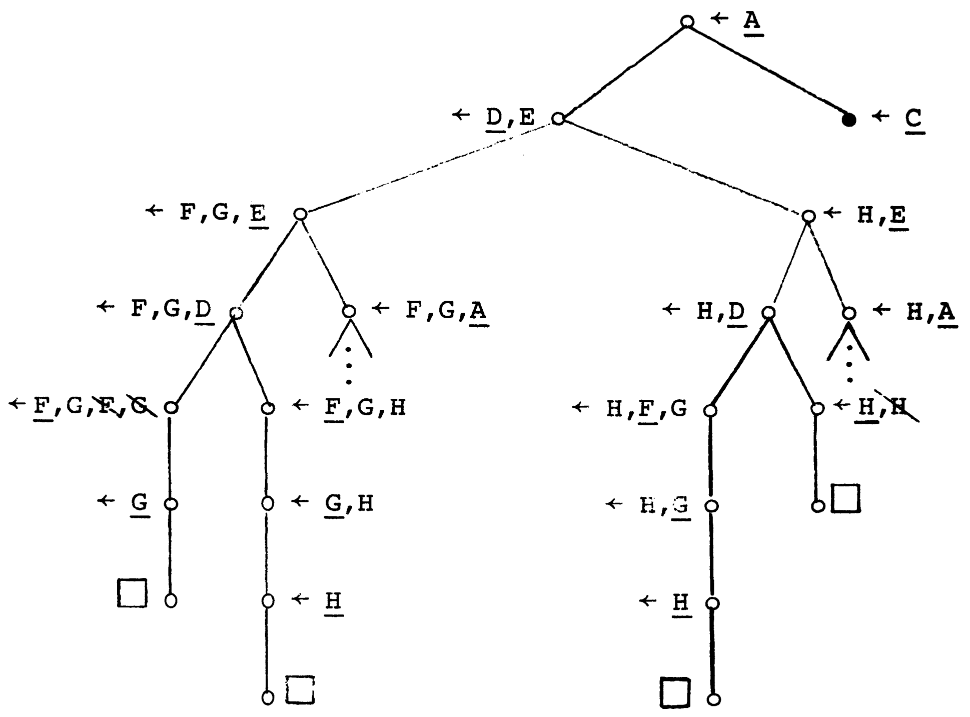


Figure 21. The generalised and-or tree, problem-reduction interpretation of the parsing oblem. Darkened nodes represent unsolvable problems. Certain labels of nodes and arcs are omitted for lack of space.

Most solution methods for problem-reduction tasks (reducing problems to independent subproblems) generate nodes directly in the and-or tree or and-or graph representation {41,22,33}. Other methods {7 , 9} are variations of ones which generate nodes in a top-down search space of goal statements. When subproblems are independent, the methods which generate nodes directly in and-or trees or graphs avoid redundancies hidden in the top-down goal statement method. This is illustrated in Figure 22.



← A

A ← C

A ← D,E

D ← F,G

D ← H

E ← D

E ← A

F ←

G ←

H ←

Figure 22. A top-down search space of goal statements for the problem-reduction task of Figure 19. The hidden redundancy here, which is not involved in the and-or tree and and-or graph representations, is the investigation of all ways of solving E duplicated for all ways of solving D. More generally, given a goal statement ← A,B , n ways of solving A, m ways of solving B and the selection of A before B, the goal statement search space contains n.m branches where the and-or tree would contain n+m branches.

However, when subproblems are dependent, the goal statement method facilitates the communication of information about solutions from one subproblem in a goal statement to another. Such communication makes it possible to avoid investigating a solution of a subproblem when it is incompatible with solutions to other subproblems in the same goal statement. It may be that the goal statement method, supplemented by a procedure which generates new assertions as lemmas whenever subproblems are solved, can avoid its redundancies while retaining the advantage of investigating subproblems in the context of the goal statements in which they occur. Methods for generating and using lemmas were first proposed by Loveland {28} and later investigated by other authors {26 }. In the sequel we shall not investigate further the method of searching for solutions directly in generalised and-or trees and shall reserve our attention instead for the top-down method of generating goal statements.

The selection of subproblems in goal statements.

Successful problem-solving depends importantly on the sequence in which problems are investigated for solution. Selection of different subproblems in the same goal statement gives rise to different search spaces. One search space can be easier to search than another.

Figure 23 shows the different search spaces determined by different choices of subproblems in the initial goal statement of the fallible Greek example. Choosing different subproblems is the difference between finding an x which is fallible and then testing that it is Greek,and finding an x which is Greek and then testing that it is fallible. Here, as in so many examples, the smaller search space is obtained by selecting the subproblem which can be solved in the least number of different ways.

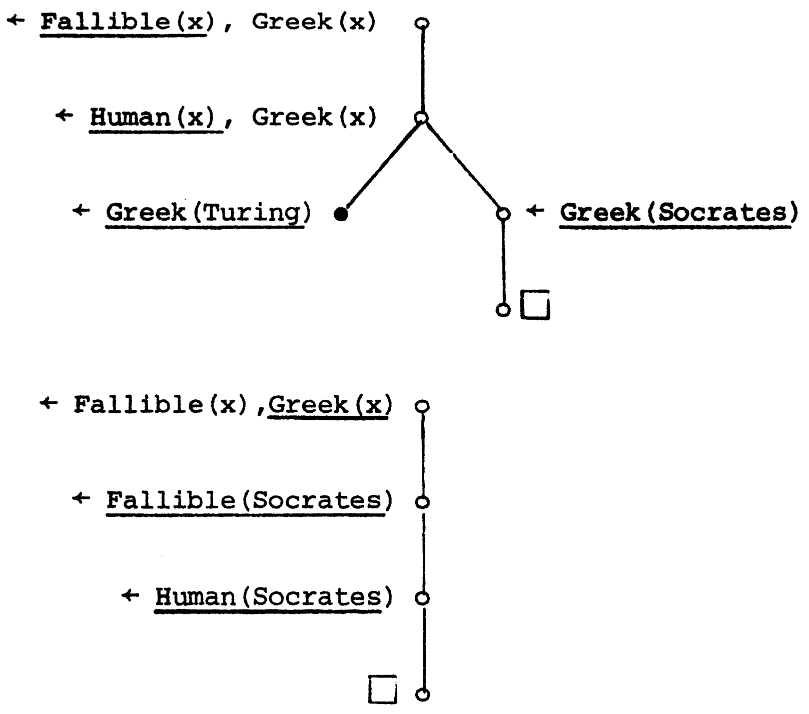


Figure 23. Different search spaces obtained by selecting different subproblems in the initial goal statement of the fallible Greek example.

In the parsing problem, the selection of subproblems determines the sequence in which top-down analysis investigates different parts of the initial string of words. One selection procedure determines a left-to-right analysis, whereas another procedure determines an analysis from right-to-left. Indeed, for every sequence of words in the initial string, there exists a selection procedure which investigates those words in the order in which they occur in the given sequence.

In general, a useful (and potentially most efficient) sequence of investigation is the one determined by the Principle of procrastination (Donald Kuehner's name for the selection rule used in SL-resolution {26}):

Select the subproblem which can be solved in the least number of different ways.

The number of different ways a problem can be solved can be measured by the number of different assertions and operators which match the problem. This number can be computed efficiently using the connection graph theorem-proving system investigated in Chapter 6. A more useful measure still can be obtained by using connection graphs to facilitate n-level look-ahead to estimate the number of different solution methods n-steps long {23}.

A more dramatic illustration of the importance of selection procedures (and of the utility of the principle of procrastination) is provided by the factorial example. In the second goal statement of Figure 8

← Fact(s(O),v),Times(s(s(O)),v,x)

it is necessary to select one of two subproblems. The selection and solution of Fact(s(O),v) before Times(s(s(O)),v,x) results in the deterministic algorithm which

(1) finds the unique v which is the factorial of one, and then
(2) finds the unique x which is two times v.

The selection and solution of Times(s(s(O)),v,x) before Fact(s(O),v) results in the highly non-deterministic algorithm which

(1) generates pairs (v,x) such that two times v is x, and then
(2) tests that v is the factorial of one.

In this example, the difference between the choice of different subproblems is the difference between a usable deterministic algorithm for computing factorial and a useless non-deterministic one. As in the previous example, the smaller search space is the one determined by the principle of procrastination.

Bottom-up and bi-directional methods for problem-reduction tasks.

Problem-reduction tasks can be solved by bottom-up and by combined top-down, bottom-up methods. Indeed both pure bottom-up solution methods {20,33} and combined methods {22} have been investigated for the pre-predicate logic formulation of problem-reduction.

A good heuristic for combining top-down and bottom-up strategies is a generalisation {22} of one formulated by Pohl {45} for path-finding problems:

> Choose at every stage the direction which gives rise to the least number of alternatives.

In the top-down direction, the number of alternatives is the total number of different ways of matching atoms in operators with selected atoms in goal statements. In the bottom-up direction, it is the total number of different ways of matching atoms in operators with assertions. In the connection graph theorem-proving system the Pohl heuristic and the principle of procrastination are unified in a single heuristic of preference for the line of least resistance.

In realistic problem-solving situations, involving a single initial problem and a large set of problem-solving methods and initially solved problems, the Pohl heuristic avoids the combinatorially explosive behaviour of pure bottom-up solution methods. In such a situation, where top-down derivation of new subproblems from old ones is problem-specific behaviour, bottom-up derivation of new solved problems from old ones is problem-independent, general-purpose behaviour, not directed toward the particular problem at hand. The rate of growth of the search space is correspondingly much faster in the bottom-up direction than it is in the top-down direction. In these situations, the Pohl heuristic dictates the selection of the top-down direction of search.

In experimental situations { 46 } bottom-up strategies compare well with top-down strategies. Typically in such situations, the set of solved problems is unrealistically small and consists only of those assertions which are necessary to solve the initial problem. Bottom-up execution does not lead to the general-purpose, problem-independent behaviour characteristic of more realistic situations. Such experimental results give a misleading picture of the relationship between top-down and bottom-up solution methods.

It would be wrong, however, to argue that bottom-up methods are of value only in unrealistic situations. Typically the initial set of assertions contains both general-purpose, problem-independent assertions and special-purpose assertions which constitute part of the formulation of the initial problem. The parsing example, for instance, contains only assertions of the problem-specific kind. The distinction between problem-independent and problem-specific assertions leads to bi-directional strategies of the Bledsoe variety {4}:

> Problems should be solved by a combined strategy which works bottom-up beginning with the initial problem-specific assertions and top-down beginning with the initial goal statement.

Within those constraints the Pohl heuristic is useful for deciding how to divide attention and effort between the two directions of search.

We have just considered the situation in which the initial problem consists of both problem-specific assertions and goal statement. More generally it is useful to consider situations in which the subproblems generated by top-down analysis also consist of both assertions and subgoals. The association of problem-specific assertions with subproblems requires the use of non Horn clauses.

Reduction of problems to subproblems consisting of assertions and goal statement.

A non clausal sentence

$A \leftarrow (B \leftarrow C)$

read

A if (C implies B)

becomes two clauses

$A \leftarrow B$

$A, C \leftarrow$

In the problem-reduction interpretation they can be regarded as stating that

in order to solve A, solve B and assume C.

Matching a goal statement $\leftarrow A'$ (where A' matches A with matching substitution θ) with the two clauses gives rise to a new goal statement and an assertion:

$\leftarrow B\ \theta$

$C\ \theta \leftarrow$

The definition of subset conforms to this pattern:

(1) $\text{Sub}(x,y) \leftarrow \text{Member}(\text{arb}(x,y),y)$

(2) $\text{Sub}(x,y)\ ,\ \text{Member}(\text{arb}(x,y),x) \leftarrow$

Together (1) and (2) can be interpreted as stating that

in order to show that x is a subset of y, show that
arb(x,y) belongs to y, where
arb(x,y) is some member of x.

In fact, (1) and (2) can be used not only to show that a given set x is a subset of y, but also to generate subsets x of a given set y, supersets y of a given set x or pairs x,y standing in the subset relation. This feature of predicate logic "programs" is elaborated upon in Chapter 5.

Figure 24 illustrates the use of (1) and (2) to solve the problem

(3) $\leftarrow \text{Sub}(A,A)$

of showing that every set is a subset of itself. The form of the search space illustrated has not been defined but it is consistent with the informal problem-reduction interpretation and it can be simulated by the connection graph theorem-proving system of Chapter 6.

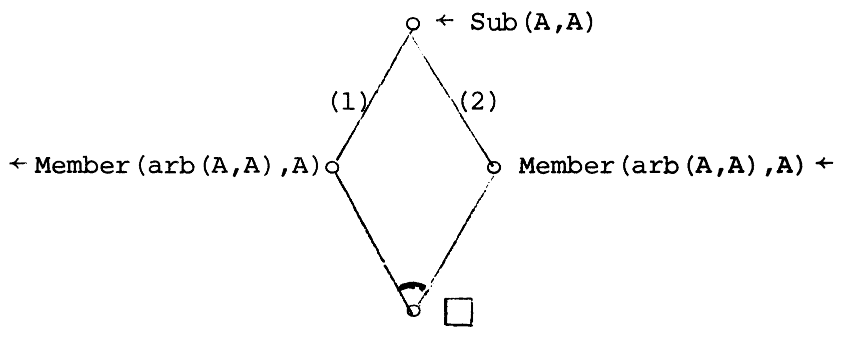


Figure 24. A search space with a mixed top-down, bottom-up representation for the problem of showing that every set is a subset of itself.

The use of non Horn clauses to reduce problems to subproblems consisting of assertions and goal statement is related to the use of non Horn clauses to achieve cooperation of different methods for solving the same problem by case analysis.

Cooperation of problem-solving methods by case analysis.

Robot plan-formation tasks requiring the construction of conditional plans can be interpreted as problem-reduction tasks needing case analysis for their solution.

Such an application of case analysis is needed, for example, to solve the problem of getting Robert to the circus. The initial problem eventually reduces to the problem of finding Robert's location:

← Holds(at(Robert,y),w').

Two different operators match the problem

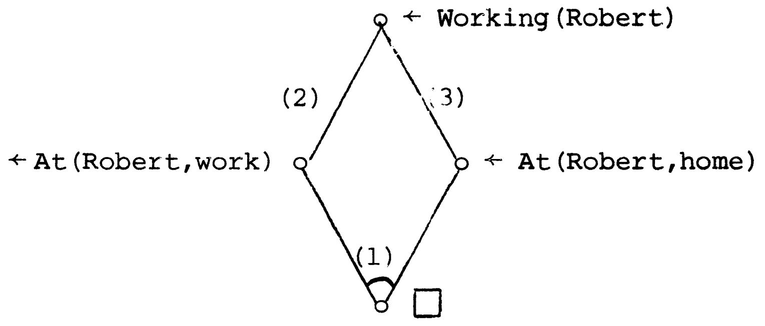
Holds(at(Robert,home),w) ← Holds(ill(Robert),w),
Holds(at(Robert,work),w) ← Holds(healthy(Robert),w).

Neither operator is able to succeed independently of the other. Each operator working alone succeeds in only one of the two cases asserted by the initial non Horn clause

Holds(healthy(Robert),O),Holds(ill(Robert),O) ←.

Cooperatively the two operators solve the problem by exhausting all the cases.

Figure 25 illustrates a similar use of case analysis to enable the cooperation of different operators for solving the problem of showing that Robert is always working.



(1) At(Robert,work), At(Robert,home) ←
(2) Working(x) ← At(x,work)
(3) Working(Robert) ← At(Robert,home)

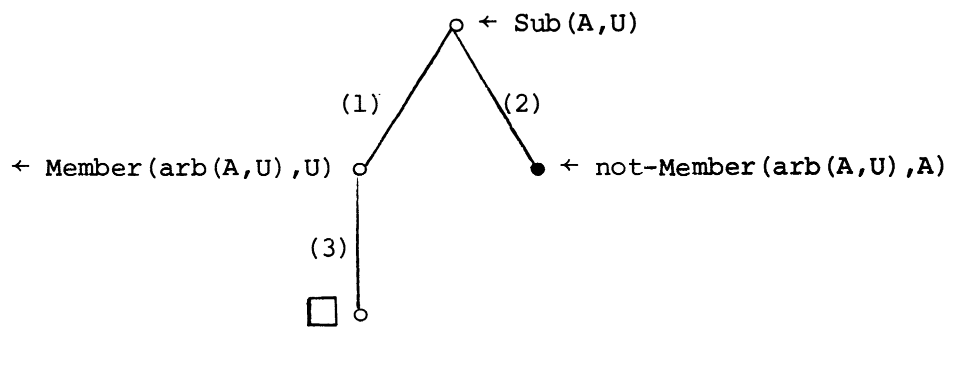
Figure 25. A search space illustrating the use of a non Horn clause for enabling the cooperation of operators by case analysis.

We have seen how the two clauses in the definition of the subset relation can be interpreted as reducing a problem to a subproblem consisting of an assertion and a subgoal. It is also possible to interpret the same clauses as two different operators for solving the problem of showing x is a subset of y:

(1) Sub(x,y) ← Member(arb(x,y),y)

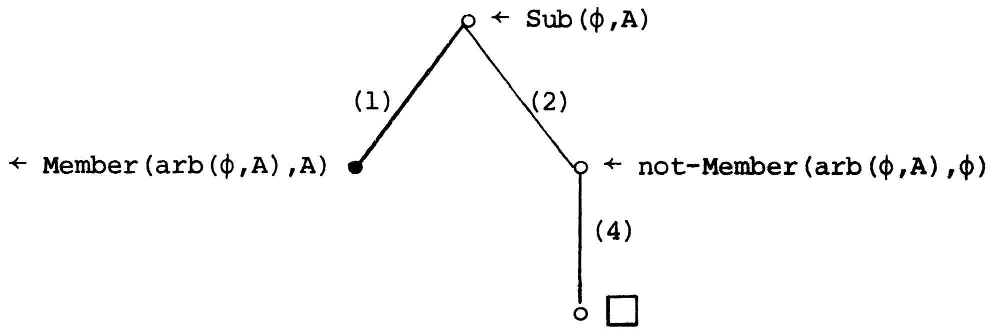
(2) Sub(x,y) ← not-Member(arb(x,y),x)

It is necessary to reinterpret the second clause as an operator which replaces the problem of showing that x is a subset of y by the problem of showing that arb(x,y) is not a member of x. With this interpretation (1) and (2) behave as different ways of solving the same problem. Sometimes one method works independently of the other. At other times both methods need to cooperate. Figure 26 shows that (1) alone is sufficient for solving the problem of showing that every set A is a subset of the universal set U. Figure 27 shows that (2) alone is sufficient for solving the problem of showing that the empty set ϕ is a subset of every set A. Figure 28, on the other hand, shows that (1) and (2) need to cooperate in order to solve the problem of showing that A is a subset of B, where A contains no more than a and b and B contains a, b and c.



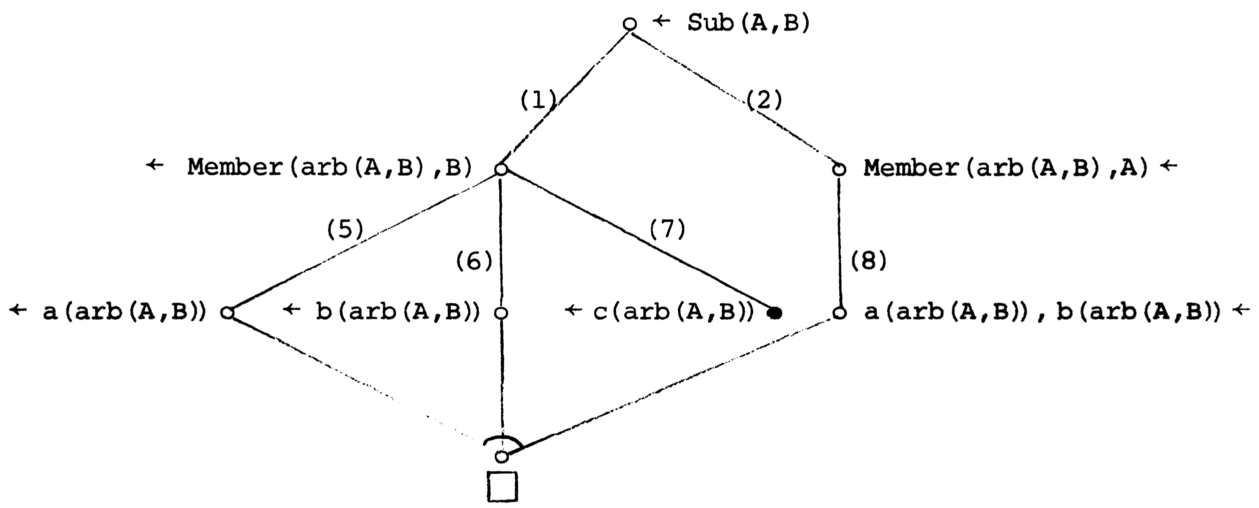
(3) Member(x,U) ←

Figure 26. A top-down search space determined by the problem of showing that every set is a subset of the universal set (defined by clause(3)).



(4) not-Member(x,ϕ) ←

Figure 27. A top-down search space determined by the problem of showing that the empty set (defined by (4)) is a subset of every set.



(5) Member(x,B) ← a(x) (6) Member(x,B) ← b(x)

(7) Member(x,B) ← c(x) (8) a(x),b(x) ← Member(x,A)

Figure 28. A mixed top-down bottom-up search space for the problem of showing A is a subset of B.

CHAPTER 5. THE PROCEDURAL INTERPRETATION OF PREDICATE LOGIC.

A Horn clause

$$B \leftarrow A_1,\ldots,A_n$$

is interpreted as the declaration of a procedure whose name is B and whose body $\{A_1,\ldots,A_n\}$ is a set of procedure calls A_i. Top-down derivations are computations. Generation of a new goal statement from an old goal statement

$$\leftarrow A_1,\ldots,A_{i-1},\underline{A_i},A_{i+1},\ldots,A_n$$

by matching the selected procedure call A_i with the name A of a procedure

$$A \leftarrow B_1,\ldots,B_m$$

is procedure invocation.

The distinction between the input and output variables of a procedure declaration depends upon the context in which it is invoked. For a given procedure invocation, the input variables are the variables in the procedure declaration which are affected by the matching substitution. The variables affected in the procedure call are the output variables. Computation proceeds by successive approximation. The output component of the matching substitution transmits partial output which improves the current approximation to the desired fully specified output.

The absence of an explicit input-output distinction has important consequences. In particular a procedure can be defined for the purpose of testing that a given n-tuple of terms holds in a given relationship. The same procedure can also be used to generate as output any subset of terms in the n-tuple, given the other terms as input.

A program, consisting of a set of clauses, is activated by an initial goal statement. The output of the program is the output component of any solution of the activating initial goal statement.

Predicate logic programs incorporate many features of standard programming languages. In particular, they include recursion, the ability of a procedure to contain a procedure call to another copy of itself. Recursion is used when a procedure declaration contains a procedure call which matches the name of another copy of the same procedure declaration.

Regarded as a programming language predicate logic is <u>non-deterministic</u> in three different ways.

(1) Several procedures can have a name which matches a selected procedure call. In such a situation, a given program and activating goal statement determine several computations. The order in which different computations are generated is <u>not determined</u>.

Distinct computations can give rise to distinct solutions,

$\underline{x} := \underline{t}_1 \;\ldots$ and $\underline{x} := \underline{t}_n$

of the initial goal statement. ($\underline{x}$ is the m-tuple of variables occurring in the initial goal statement and $\underline{t}_i$ is an m-tuple of terms.) It is <u>not determined</u> which solution will be returned as output.

(2) The selection of different procedure calls in the same goal statement gives rise to different search spaces of computations. The program does <u>not determine</u> how procedure calls are selected.

(3) Several procedure calls may need to cooperate in order to execute successfully a given selected procedure call. The resulting output may be ambiguous:

$\underline{x} := \underline{t}_1 \;\ldots$ or $\underline{x} := \underline{t}_n$.

Such an output does <u>not determine</u> the value of the output variables $\underline{x}$ unambiguously.

These three different kinds of non-determinism are often confused in the theory of computation. In particular the first and third kinds of non-determinism are easily confused because of the ambiguity of <u>and</u> and <u>or</u>.

In the procedural interpretation, terms function as the data structures which are manipulated by the program and serve as its input and output.

Such a use of terms as data structures gives predicate logic many of the characteristics of a list-processing language such as LISP { }. Top-down execution of clauses corresponds to the standard way of interpreting programs. Bottom-up execution generally leads to problem-independent, combinatorially explosive behaviour.

In other programs, sets of problem-specific assertions function as data structures. Top-down execution of clauses interrogates the data base of assertions. Bottom-up execution manipulates it, adding new assertions, possibly deleting old ones. The parsing problem is a good example of such a program. The wider use of sets of assertions as data structures promises to make a useful contribution to the general methodology of computer programming.

The procedural interpretation of Horn clauses.

A goal statement

$$\leftarrow A_1,\ldots,A_n$$

is a set of procedure calls. A Horn clause

$$B \leftarrow A_1,\ldots,A_n$$

is a declaration of a procedure whose name is B and whose body is the set of procedure calls A_i. If n = 0 then the procedure has an empty body. The null clause

$$\square$$

is interpreted as the halt statement.

The name of a procedure identifies the procedure calls to which it can respond. It asserts the names of the problems which it can solve.

Thus the definition of factorial consists of two procedure declarations:

(1) $\mathrm{Fact}(0,s(0)) \leftarrow$

(2) $\mathrm{Fact}(s(x),u) \leftarrow \mathrm{Fact}(x,v),\mathrm{Times}(s(x),v,u)$.

The first procedure responds to procedure calls which match its name Fact(0,s(0)). It replaces such a procedure call by an empty set of procedure calls and returns as output the output component of the matching substitution. The second procedure likewise responds to procedure calls which match its name Fact(s(x),u). It replaces such a call by the non-empty set of procedure calls

$$(\text{Fact}(x,v), \text{Times}(s(x),v,u))\ \theta_I$$

where θ_I is the input component of the matching substitution θ. It returns as an approximation to its output the output component of θ.

Notice that the procedural interpretation is identical to the top-down interpretation of Horn clauses. Computations are top-down derivations and successfully terminating computations are refutations. The single refutation in Figure 9 is a successfully terminating computation. The solution

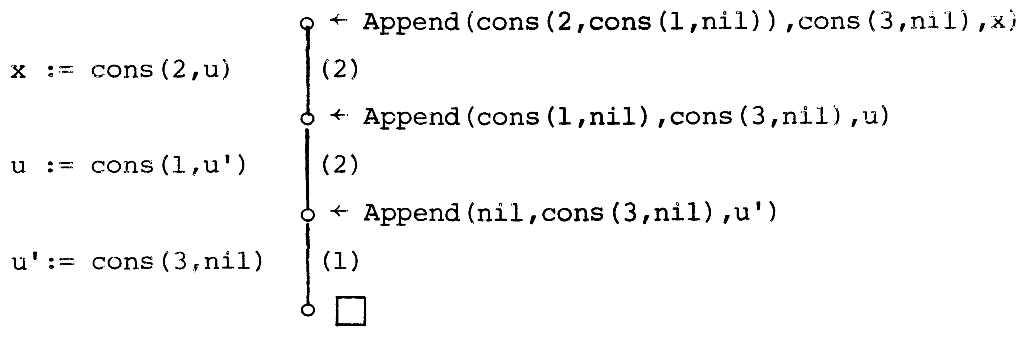
$$x := s(s(0))$$

of the initial goal statement is the desired output of the program.

Computation by successive approximation to output.

In conventional programming languages, functions, subroutines and procedures return output only when they have successfully terminated computation. In predicate logic, at every stage of a computation, a procedure transmits partial output to the calling environment. The successive partial outputs accumulate and generate successive approximations to the output. These successive approximations are generated whether or not the computation eventually succeeds.

Figure 29 illustrates the computation, by successive approximation, of the list which results from appending cons(3,nil) to cons(2,cons(1,nil)).



← Append(cons(2,cons(1,nil)),cons(3,nil),x)
x := cons(2,u)
(2)
← Append(cons(1,nil),cons(3,nil),u)
u := cons(1,u')
(2)
← Append(nil,cons(3,nil),u')
u' := cons(3,nil)
(1)
□

(1) Append(nil,x,x) ←

(2) Append(cons(x,y),z,cons(x,u)) ← Append(y,z,u)

Figure 29. Computation by successive approximation to output.

Every step in the computation adds new information about the output variable x. After the first step it is determined that

x is cons(2,u)

whether or not the computation eventually succeeds. After the second step

x is cons(2,cons(1,u')).

Finally, after the third step, the output variable is fully specified,

x is cons(2,cons(1,cons(3,nil))),

and the computation terminates successfully.

Because of the lack of explicit input-output distinction, computation can proceed by successive approximation to input as well as to output. The utility of both kinds of computation by approximation will be discussed later in connection with the admissible pairs example.

No input-output distinction.

The lack of explicit input-output distinction has as one of its consequences that a procedure defined with the intention of generating output t, given input s, can also be used to generate s as output, given t as input. Thus the procedure Append(s,t,u) might be defined with the purpose of generating as output u the list which results from appending together the two lists s and t given as input. In a conventional list-processing language the procedure could be used only to compute the input-output relation for which it was defined. But in predicate logic the same procedure can be used to test that Append(s,t,u) holds given s, t and u as input, to generate s from t and u, to generate t from s and u, to generate s and t from u, or in general to generate any subset of {s,t,u} given the rest of the set as input. There are 2^3 such input-output relations which are computed by the single procedure Append.

Figure 30 illustrates the use of Append to compute the list x such that cons(2,cons(1,cons(3,nil))) results from appending cons(3,nil) to x. Although the search space contains an unsuccessfully terminating computation it contains only one successful one.

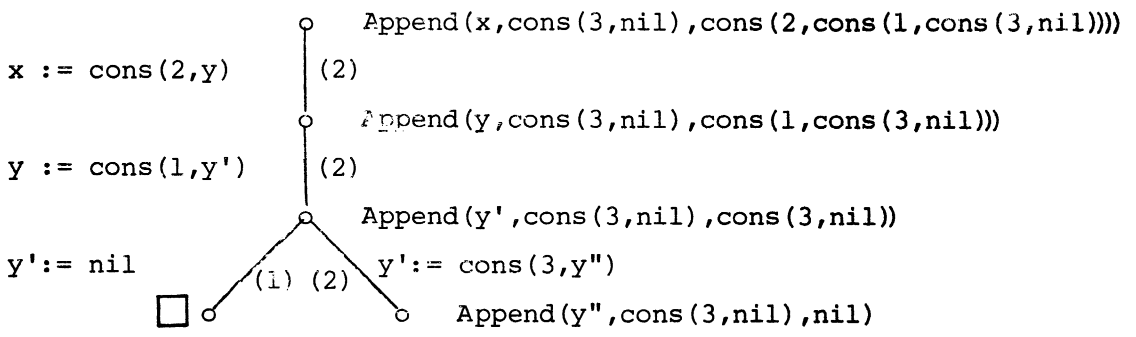

Append(x,cons(3,nil),cons(2,cons(1,cons(3,nil))))
x := cons(2,y)
(2)
Append(y,cons(3,nil),cons(1,cons(3,nil)))
y := cons(1,y')
(2)
Append(y',cons(3,nil),cons(3,nil))
y':= nil
(1)
(2)
y':= cons(3,y")
Append(y",cons(3,nil),nil)

Figure 30. Use of the same procedure to compute a different input-output relation.

Non-determinism$_1$: The scheduling of procedures when several match a procedure call.

Typically a procedure is defined with the intention of using it to compute a function (in the case of Append, to compute the unique list which results from appending together two other lists given as input). Used in the way originally intended, the procedure behaves deterministically in the sense that it computes a unique output for a given set of inputs. (In the case of Append, the procedure behaves deterministically in the stronger, more important, sense that the search space of computations contains no branch points.)

The inverse of a function is generally many-valued and therefore not a function. Using a procedure to compute the inverse of the function originally intended changes the procedure from one which behaves deterministically to one which behaves non-deterministically. In particular, Append is many-valued when it is used to output a pair of lists s,t which partitions a list u, given as input (so that the relation Append(s,t,u) holds between input and output). Such an application of Append is non-deterministic$_1$ because the space of computations contains branch points. Figure 31 illustrates such a branching search space of computations, determined by the problem of partitioning the list cons(2,cons(1,cons(3,nil))).

Notice the economy obtained by structuring the space of computations as a tree. In particular, the two different partitions

x:= cons(2,cons(1,nil)) , y:= cons(3,nil) and
x:= cons(2,cons(1,cons(3,nil))) , y:= nil

are obtained from the same initial computation of the common approximate solution

x:= cons(2,cons(1,x")) .

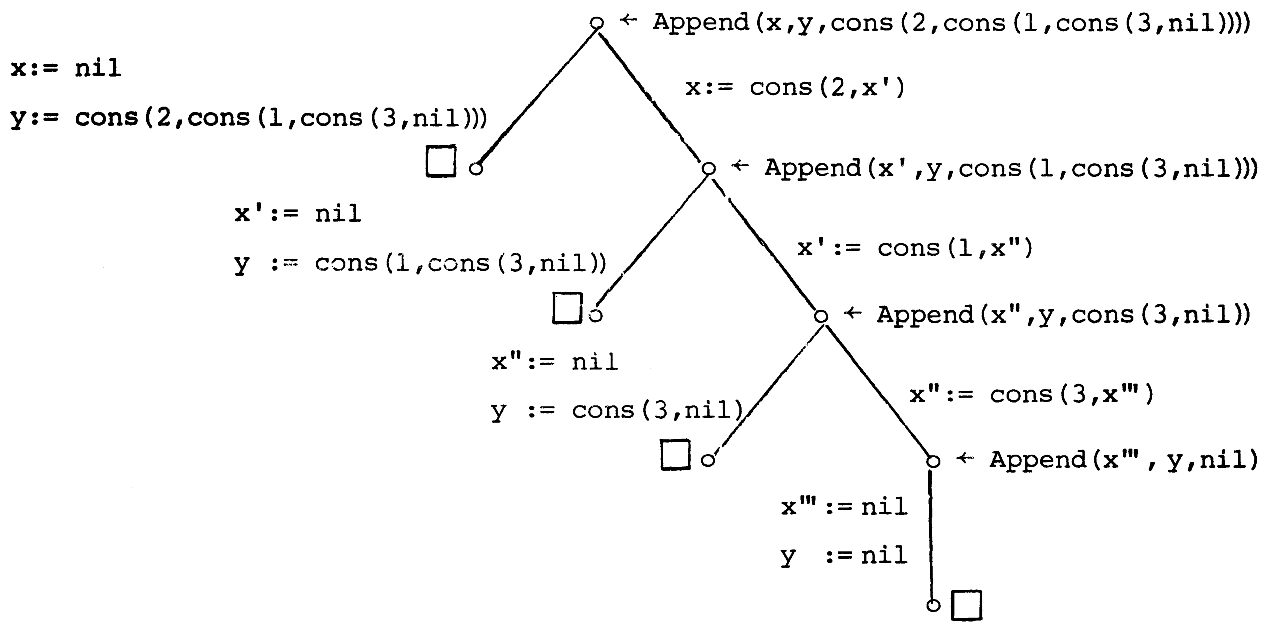


Figure 31. Non-determinism$_1$: a branching space of computations.

The non-determinism associated with a branching search space of computations concerns the scheduling of the processor(s) which generates computations in the search for one which terminates successfully. The scheduling of the processor is not determined by the program. When several procedures match a given procedure call, it is not determined which procedure will be tried first. Nor is it determined whether one will be tried before the others or whether all will be tried simultaneously (in parallel). When the initial goal statement has several distinct solutions (associated with different successful computations) it is not determined which solution the processor will return as the output of the program.

Non-determinism$_2$: The scheduling of procedure calls in the body of a procedure.

The body of both a goal statement and a procedure declaration is a set of procedure calls. In a conventional programming language the body of a procedure is likely to be a sequence of procedure calls. In predicate logic, the procedure definition does not specify the sequence in which procedure calls are to be executed. For this reason predicate logic programs are non-deterministic. It is not determined$_2$ which procedure call should be activated first. Nor is it determined$_2$ whether one procedure call should be executed without interpretation and terminated before the activation of the others. It is not even determined$_2$ whether one call should be executed before another or whether all should be executed in parallel.

An example of the non-determinism$_2$ of the scheduling of procedure calls in a goal statement

$\leftarrow$ Fact(s(O),v),Times(s(s(O)),v,u)

was discussed in the preceding chapter. Activation of the Fact procedure call before the Times procedure call results in the usual deterministic$_1$, recursive algorithm for computing factorial. Activation of Times before Fact results in an intolerably inefficient, but none-the-less correct, non-deterministic$_1$ algorithm. In general the scheduling of procedure calls determines$_2$ the size of the search space of all computations. Different scheduling rules determine$_2$ different search spaces,some of which may be significantly easier to search than others.

Effective scheduling of procedure calls depends importantly on the input-output distribution of variables in the procedure call which activates the given procedure. For example, when the procedure

Grandparent(x,y) $\leftarrow$ Parent(x,z),Parent(z,y)

is used to test whether x is a grandparent of y, it does not matter greatly which procedure call, Parent(x,z) or Parent(z,y), is activated before the other. According to the principle of procrastination, if x has fewer than two children z, then it is better to find a child of x and then test that it is a parent of y. But if x has more than two children then it is better to find a parent of y and then test that it is a child of x.

More important is the scheduling of procedure calls in this example when only one of x and y is given as input and the other is desired as output. If it is required to find a grandchild y of x then it is important first to find a child z of x and then to find a child y of z. But if it is required to find a grandparent x of y then it is much better to find a parent z of y and then to find a parent x of z than it is first to find a pair x, z such that x is a parent of z and then to test that z is a parent of y.

It is important therefore for the scheduling of procedure calls in the body of a procedure to be sensitive to the context in which it is called. To fix scheduling by a single initial ordering of procedure calls is to define a scheduling appropriate for the initially intended input-output relation, but possibly unusable for other input-output relations. As important as this point is, some critics have wrongly judged that dynamic scheduling is less efficient than static ordering {43}.

Another restriction on the scheduling of procedure calls, which it is useful to remove, is the restriction that procedure calls be executed last-in-first-out. Given, for example, the goal statement

$\leftarrow \underline{P(x,y)}, Q(y)$

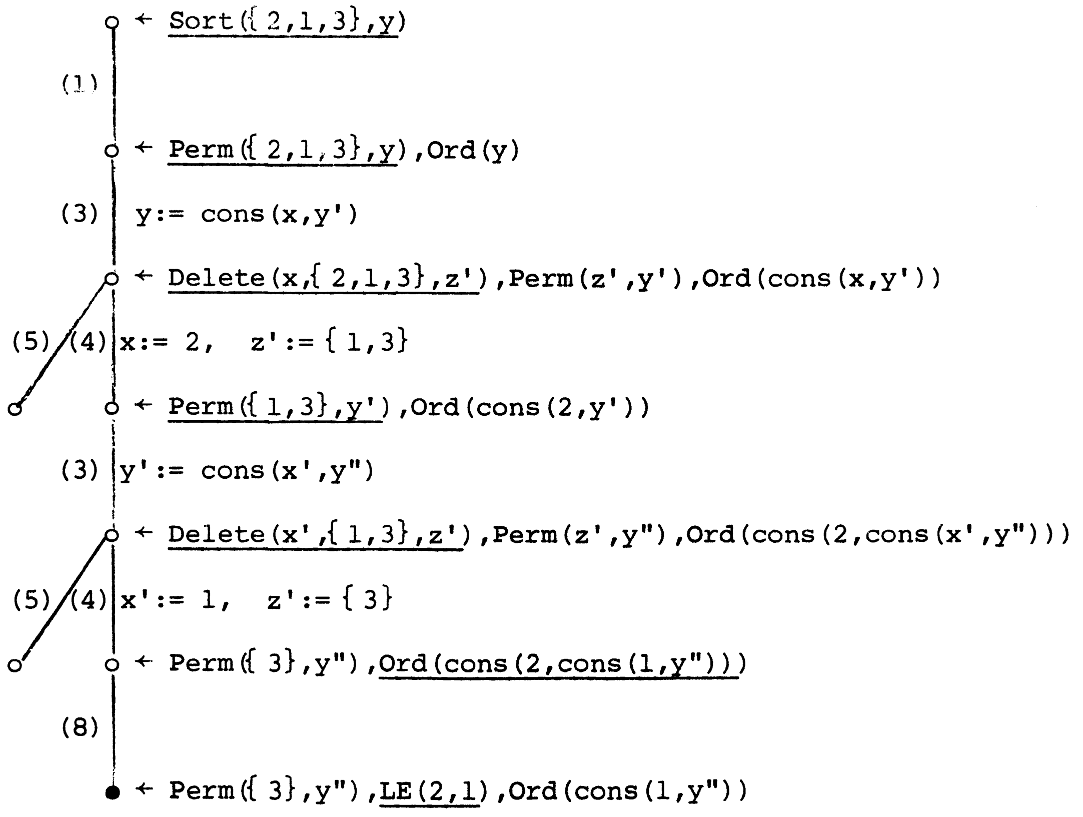
where P is selected before Q and where x is input variable and y is output variable, the successive approximations to the output of P are transmitted as successive approximations to the input of Q. If execution proceeds in a last-in-first-out manner, then Q is selected only after P successfully terminates and the variable y is fully specified. In many situations, however, it is possible and desirable to interrupt the execution of P and to run Q with partially specified input, returning to the execution of P when Q desires further specification of its input. Such a method of combining the execution of P and Q is especially useful when P non-deterministically generates a number of different outputs y. If Q terminates unsuccessfully for some partial specification of y, then it is possible to abandon the computation of P which gave rise to that value of y. This is illustrated in Figure 32. The initial goal, which is to sort the list {2,1,3} is replaced by

$\leftarrow \underline{\mathrm{Perm}(\{2,1,3\},y)}, \mathrm{Ord}(y)$

the goal of generating a permutation of {2,1,3} and testing that it is ordered. Instead of executing the goal statement last-in-first-out and waiting until Perm generates the fully specified permutation {2,1,3} before recognising that it is not ordered, Ord tests the partially specified output cons(2,cons(1,x)) of Perm, recognises that it is not ordered and abandons the corresponding branch of the search space.

Removing the last-in-first-out restriction on the scheduling of procedure calls is useful in some cases. In other cases, such as in the admissible pairs example, it is essential. There it is necessary to execute the two procedure calls Double(x,y) and Triple(x,y) in parallel, communicating partial output from one procedure call to serve as partial input for the other. Unlike the preceding example, where Perm could run without the help of Ord, in this example, neither Double nor Triple could run with tolerable efficiency without help from the other. The execution of Double and Triple as communicating, parallel processes is shown in Figure 33.

```
 (1) Sort(x,y) ← Perm(x,y),Ord(y)
 (2) Perm(nil,nil) ←
 (3) Perm(z,cons(x,y)) ← Delete(x,z,z'),Perm(z',y)
 (4) Delete(x,cons(x,y),y) ←
 (5) Delete(x,cons(y,z),cons(y,u)) ← Delete(x,z,u)
 (6) Ord(nil) ←
 (7) Ord(cons(x,nil)) ←
 (8) Ord(cons(x,cons(y,z))) ← LE(x,y), Ord(cons(y,z))
 (9) LE(x,x) ←                 (10) LE(1,2) ←
(11) LE(2,3) ←                 (12) LE(1,3) ←
```



← Sort({2,1,3},y)
(1)
← Perm({2,1,3},y),Ord(y)
(3)
y:= cons(x,y')
← Delete(x,{2,1,3},z'),Perm(z',y'),Ord(cons(x,y'))
(5)
(4)
x:= 2, z':={1,3}
← Perm({1,3},y'),Ord(cons(2,y'))
(3)
y':= cons(x',y")
← Delete(x',{1,3},z'),Perm(z',y"),Ord(cons(2,cons(x',y")))
(5)
(4)
x':= 1, z':={3}
← Perm({3},y"),Ord(cons(2,cons(1,y")))
(8)
← Perm({3},y"),LE(2,1),Ord(cons(1,y"))

Figure 32. An unsuccessful attempt to sort {2,1,3} by generating a partially specified permutation cons(2,cons(1,y")) which is not ordered.

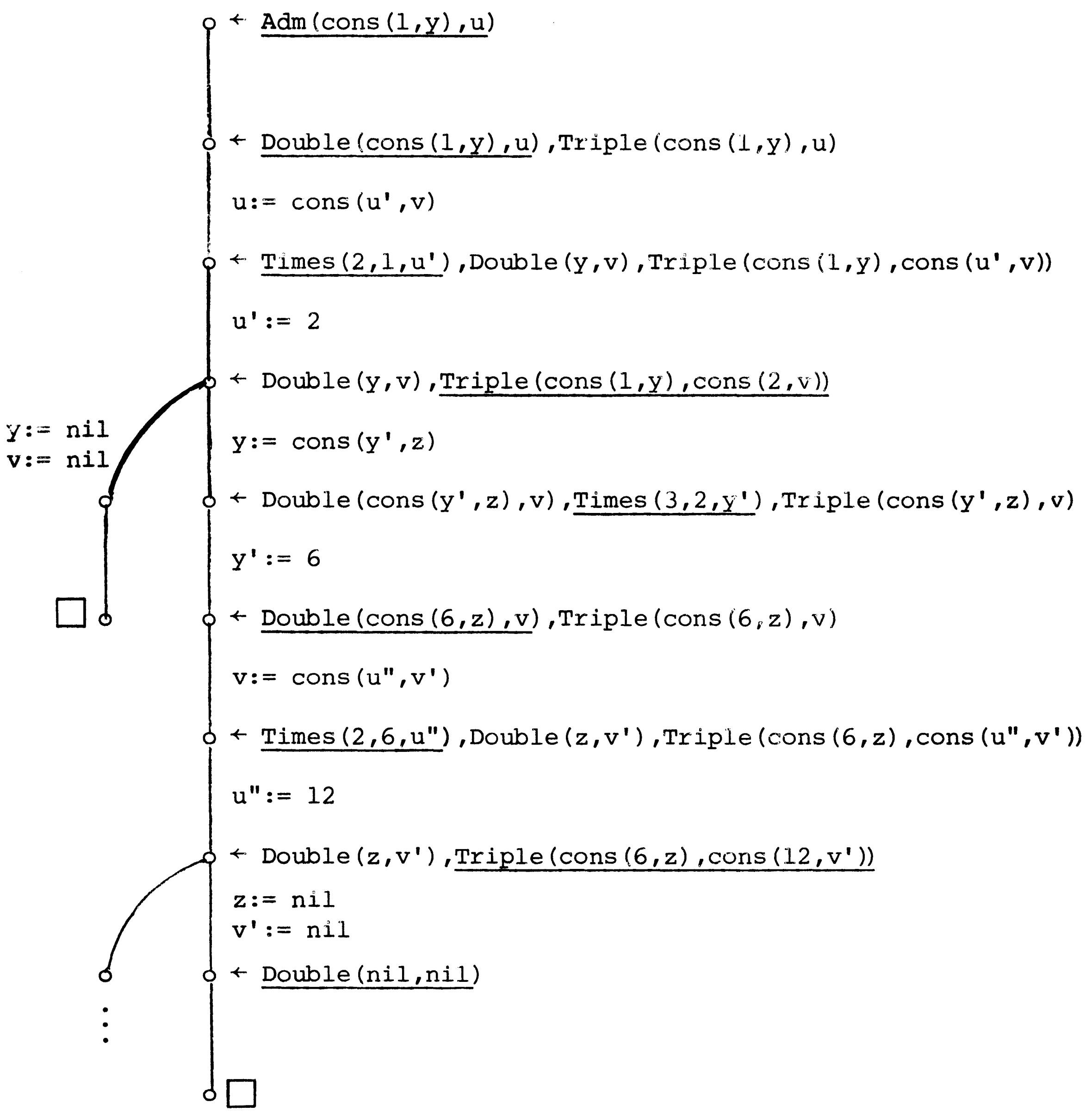

← Adm(cons(1,y),u)
← Double(cons(1,y),u),Triple(cons(1,y),u)
u:= cons(u',v)
← Times(2,1,u'),Double(y,v),Triple(cons(1,y),cons(u',v))
u':= 2
← Double(y,v),Triple(cons(1,y),cons(2,v))
y:= nil
v:= nil
y:= cons(y',z)
← Double(cons(y',z),v),Times(3,2,y'),Triple(cons(y',z),v)
y':= 6
← Double(cons(6,z),v),Triple(cons(6,z),v)
v:= cons(u",v')
← Times(2,6,u"),Double(z,v'),Triple(cons(6,z),cons(u",v'))
u":= 12
← Double(z,v'),Triple(cons(6,z),cons(12,v'))
z:= nil
v':= nil
← Double(nil,nil)

Figure 33. The execution of Double and Triple in the admissible pairs example as a pair of communicating, parallel processes.

Non-determinism$_3$: The underspecification of output.

The problem of getting Robert to the circus

$\leftarrow$ Holds(at(Robert,circus),w)

can be thought of as an initial procedure call with output variable w. With this interpretation, the successful computation illustrated in Figure 17 is arrived at by combining by means of case analysis two different computations, each with a different associated approximation to the desired output. The final output computed in this way is "underspecified"

w:= do(go(Robert,home,circus),O)
 or do(go(Robert,work,circus),O).

If instead of the case statement

Holds(healthy(Robert),O),Holds(ill(Robert),O) $\leftarrow$

we had the two (consistent) assertions

Holds(healthy(Robert),O) $\leftarrow$
Holds(ill(Robert),O) $\leftarrow$

then the output would have been "overdefined"

w:= do(go(Robert,home,circus),O)
 and do(go(Robert,work,circus),O).

In fact the processor returns upon termination only one of the two possible outputs. It is not determined$_1$ which of the two results it returns.

In this example the ambiguity between and and or does not arise. It does arise however in the parsing example. It is not clear how to express correctly such a fact as

the word "fish" can be used as either a noun or a verb.

Should it be interpreted as a non Horn clause

Noun(x,y),Verb(x,y) $\leftarrow$ Fish(x,y)

or as two Horn clauses

Noun(x,y) $\leftarrow$ Fish(x,y)
Verb(x,y) $\leftarrow$ Fish(x,y).

The Horn clause interpretation is consistent with the previous formulation of the parsing problem in Chapter 2 and has been applied to the parsing of ambiguous sentences elsewhere {23}. It is not known, however, whether the parsing problem can be consistently formulated with the non Horn clause interpretation.

Sets of assertions as data structures.

The factorial, append, and admissible pairs examples illustrate the use of terms as data structures. The parsing example (and to a more limited extent, the robot plan-formation example also) illustrates the use of sets of assertions as data structures.

When terms are used as data structures, top-down execution of procedures behaves similarly to recursive execution in conventional programming languages. Bottom-up execution is problem-independent and combinatorially explosive. Its only application seems to be in the theory of computation where it can be used to justify certain rules, such as Scott induction, for proving properties of programs.

When sets of problem-specific assertions are used as data structures, top-down execution of procedures interrogates the data base. Bottom-up execution is problem-dependent manipulation of the data base, deriving new assertions from old ones. In some cases the procedure which derives the new assertion from the old one may contain the only atom which matches the old assertion. In such a case, the old assertion can be deleted when the new one is generated. Then bottom-up execution behaves as a destructive assignment operation which overwrites part of the contents of the data base. The deletion of clauses after all operators have been applied to some selected atom (of which destructive assignment in the data base of assertions is a special case) is a characteristic feature of the connection graph theorem-proving system.

It is instructive to compare the previous formulation of the parsing problem with a different formulation which uses terms as data structures.

```
 (1)  ← S(cons(The,cons(little,cons(mouse,cons(likes,cons(cheese,nil))))))
 (2)  S(z) ← Np(x),Vp(y),Append(x,y,z)
 (3)  Np(x) ← Noun(x)
 (4)  Np(u) ← Det(x),Adj(y),Noun(z),Append(x,y,v),Append(v,z,u)
 (5)  Vp(x) ← Verb(x)
 (6)  Vp(z) ← Verb(x),Np(y),Append(x,y,z)
 (7)  Det(cons(The,nil)) ←
 (8)  Adj(cons(little,nil)) ←
 (9)  Noun(cons(mouse,nil)) ←
(10)  Verb(cons(likes,nil)) ←
(11)  Noun(cons(cheese,nil)) ←
```

Here the assertions (7)-(11) are problem-independent. In a realistic parsing problem they would constitute a small subset of a larger set containing such assertions as

```
Noun(cons(girl,nil)) ←
Verb(cons(guides,nil)) ←
```

which are irrelevant to the particular problem at hand. Therefore bottom-up execution of (1)-(11) is problem-independent. The problem-dependent information is encoded in the term in the initial goal statement. The components of this list-like data structure cannot be overwritten as they can be when the problem-specific data is encoded as a set of assertions and is manipulated bottom-up by a connection graph theorem-prover.

Notice also in this example the procedure calls Append,which have no analogue in the earlier formulation of the parsing problem. When sets of assertions are used as data structures, the program has direct access to the individual assertions in those sets. Direct access to assertions is like direct access to the components of an array. When terms are used as data structures, then special procedures like Append need to be invoked in order to access the contents of data structures.

A still less satisfactory formulation of the parsing problem is the one suggested by the formalism of formal language theory. The function symbol f is an associative concatenation function. The associativity of f needs to be dealt with by supplementing clauses (1)-(11) below with axioms of associativity or by modifying the matching algorithm {44}.

```
 (1) ← S(f(The,f(little,f(mouse,f(likes,cheese)))))
 (2) S(f(x,y)) ← Np(x),Vp(y)
 (3) Np(x) ← Noun(x)
 (4) Np(f(x,f(y,z))) ← Det(x),Adj(y),Noun(z)
 (5) Vp(x) ← Verb(x)
 (6) Vp(f(x,y)) ← Verb(x),Np(y)
 (7) Det(The) ←
 (8) Adj(little) ←
 (9) Noun(mouse) ←
(10) Verb(likes) ←
(11) Noun(cheese) ←
```

This formulation of the parsing problem, although easy to read, suffers from all the problems of the preceding formulation which uses lists as data structures. Moreover, it suffers additional problems with the treatment of the concatenation function.

The satisfactory behaviour of predicate logic programs depends upon good programming style. Different programs for solving the same problem can be logically equivalent but can have very different pragmatic characteristics.

But even the best program will not evoke reasonable behaviour from an unreasonable program executor. As mentioned in Chapter 2, until recently most proof procedures for predicate logic have behaved as very unreasonable executors of predicate logic programs. The recent elaboration of top-down and connection graph theorem-provers has significantly improved the quality of inference systems. There remains the problem of improving the intelligence of search strategies.

Selection of direction and scheduling of procedures and procedure calls.

Predicate logic programs do not specify the direction (top-down or bottom-up) in which procedures should be executed. When several procedures match a given procedure call, they do not specify how the different procedures should be scheduled. When several procedure calls occur in the body of a given procedure declaration or goal statement, they do not specify how the different procedure calls should be scheduled. The specification of these choices can be made either by the processor without help from the program writer or by the programmer in a separate auxiliary language. In the latter case it is desirable to separate the machine-independent statement of the predicate logic program from the machine-oriented specifications in the auxiliary language.

Autonomous search strategies for scheduling the generation of clauses have been investigated for both top-down and bottom-up theorem-proving {20,38,59}. These strategies use merit orderings or evaluation functions to guide the generation of clauses in the search space. Arguments against the adequacy of such search strategies have been advanced by Hayes {1, 17}. He argues convincingly that the kind of information employed in these strategies is not the kind of information needed for effective problem-solving. He argues also that the information necessary to control the behaviour of the processor can be most effectively supplied by the programmer in a separate auxiliary control language. We endorse his conclusion that the design of autonomous search strategies is not a useful short-term research objective, whereas the design of effective control languages is.

The control of search strategies concerns the scheduling of different procedures which match the same procedure call. Useful control primitives for scheduling procedures can be found in programming languages like PLANNER {18}, MICROPLANNER {55}, CONNIVER {56}, POPLER {10}, SAIL {13}, QA4 {52} and QLISP {47} and can be applied in predicate logic.

The recommendation list, which allows a user to specify the sequence in which different procedures should be tried, is a control primitive of this sort.

The control methods used to sequence procedure calls in conventional programming languages are generally restricted to sequential execution, parallel execution and execution by co-routines. These methods are insensitive to variations in the distribution of input and output variables. For this reason they are not entirely adequate for controlling execution in predicate logic programs. Fortunately, the principle of procrastination, which delays the execution of a procedure call when it matches many procedure names, is a useful autonomous control strategy which works in surprisingly many cases. To the extent that it fails in a number of important cases, it needs to be improved or supplemented by user-specified advice conveyed to the processor in an auxiliary control language.

Finally, especially when sets of assertions are used as data structures, the processor has to choose between top-down and bottom-up execution. In programming languages of the PLANNER family, the direction in which procedures are to be executed is specified in advance by the types associated with procedure declarations (consequent theorem type if the direction is top-down, antecedent theorem type if it is bottom-up). Moreover each procedure call is assigned the type of the procedures which it is allowed to invoke. The Bledsoe and Pohl heuristics, on the other hand, are context-dependent, direction-choosing strategies of the autonomous kind. Other autonomous strategies apply, moreover, when the connections between matching procedure calls and procedure names are explicitly represented in a connection graph. These heuristics will be discussed in the next chapter.

Both autonomous strategies and control languages have useful contributions to make towards the more effective selection of direction and scheduling of procedures and procedure calls. Some day it may be possible to design entirely autonomous strategies which execute programs satisfactorily without help from the programmer. In the meantime it will be necessary for the programmer to help the program executor by communicating to it in an auxiliary control language the control information needed for satisfactory execution of programs.

<u>The pragmatic content of predicate logic programs</u>.

It is a common view that predicate logic is a specification language, whose "programs" have semantic but not pragmatic content. This view is taken by Hayes {17} who would incorporate all pragmatic information into the advice written in the auxiliary control language.

Our contrary view is that to ignore the pragmatic aspects of predicate logic programs is to encourage the writing of "uncontrollable" programs. Different programs which have the same semantic content can have very different pragmatic characteristics. One program might be regarded as a useful specification of the problem, but be unusable for efficient computation. Another equivalent program might run efficiently but be difficult to recognise as computing the same intended input-output relation. Proving the correctness of such a program amounts to proving its equivalence to the specification program.

A good example of the pragmatic content of predicate logic programs is provided by the sorting problem studied by van Emden {11}. The previous program for sorting lists,

```
Sort(x,y) ← Perm(x,y),Ord(y),
```

is best regarded as a specification of sortedness. Even the scheduling of procedure calls which uses Ord to monitor the partial output of Perm does not produce an efficient sorting algorithm. But straightforward sequential execution produces Quicksort {19} from the following program:

```
Sort*(nil,nil) ←
Sort*(cons(x,y),z) ← Part(x,y,u,v),Sort*(u,u'),
                     Sort*(v,v'),
                     Append(u',cons(x,v'),z).
```

Here Part(x,y,u,v) holds when u is the list of all members of y which are less than or equal to x and v is the list of all members of y which are greater than x.

Sort and Sort* are equivalent in the sense that Sort(s,t) and Sort*(s,t) hold for the same pairs of terms s,t. Sort is useful for specifying the notion of sortedness but useless for efficiently sorting lists. Sort* is efficient for sorting lists but less convincing as a specification of sortedness.

Problems of predicate logic programming.

First-order predicate logic has the limitation that procedures cannot serve as data structures to be interrogated and manipulated by other procedures. This limitation can be overcome by employing higher-order instead of first-order logic. Unfortunately, at the present time there do not seem to exist for higher-order logic proof procedures which behave as reasonable program executors. The situation may improve with time. But until then it is possible to solve the problem without leaving first-order logic.

The Holds predicate used in the robot plan-formation problem provides a way of using first-order logic to gain some of the expressive power and problem-solving capabilities of higher-order logic. First-order set theory provides similar capabilities, as do first-order theories of lambda-conversion. In particular, Moore has written a first-order predicate logic program to interpret higher-order programs written in a LISP-like language, BAROQUE {40}. It may be that eventually proof procedures for higher-order logic will be improved and will provide a more satisfactory programming language than first-order logic.

Another problem of predicate logic is the problem of programming style. In particular, the choice of data structures, terms or sets of assertions, is especially important. It would be beneficial to obtain for other problem domains the advantages gained in the parsing example by using sets of assertions instead of terms. It would be useful to reformulate the previously investigated list-processing problems using sets of assertions to represent lists. In the robot plan-formation problem, the use of sets of assertions to represent states might have interesting new implications for the frame problem.

A related problem of programming style concerns the effective use of non Horn clauses. The use of non Horn clauses to declare a data base of assertions local to a given procedure call has been illustrated in the subset examples. It seems likely that such applications of non Horn clauses have wider applicability.

In the recent past, significant advances have been made by abandoning the use of the Equality predicate. It seems reasonable to expect that further advances will be made by continued investigations into the pragmatics of programming style.

Finally, perhaps the most important problem concerns the details of implementation: how best to represent clauses inside the computer? The Boyer-Moore structure-sharing representation { 6 } provides one solution, which needs to be improved. Such an improvement might be suggested by studying the application to predicate logic of the Bobrow-Wegbreit { 5 } implementation methods.

Another, less difficult, problem of implementation concerns the design of a useful external form for predicate logic programs. In particular the external form might admit sentences in non clausal form, such as

$$A \leftarrow (B \leftarrow C)$$

instead of

$$A \leftarrow B \quad \text{and} \quad A, C \leftarrow ,$$

and

$$B_1 \text{ and } B_2 \text{ and } \ldots \text{ and } B_m \leftarrow A_1, \ldots, A_n$$

instead of

$$\begin{array}{l} B_1 \leftarrow A_1, \ldots, A_n \\ B_2 \leftarrow A_1, \ldots, A_n \\ \quad \vdots \\ B_m \leftarrow A_1, \ldots, A_n \ . \end{array}$$

A programming language based on predicate logic, PROLOG, has been implemented at the University of Aix-Marseille, Luminy. It uses a last-in-first-out, top-down theorem-prover as interpreter and uses backtracking to sequence procedures. Despite the limitations of the program executor, PROLOG is surprisingly fast and easy to use. PROLOG programs have been written for natural language question-answering (Colmerauer and Pasero), symbolic integration (Bergman and Kanoui), and robot-plan formation (Warren).

CHAPTER 6. CONNECTION GRAPHS.

In Chapter 2 we defined two theorem-proving systems, one top-down, the other bottom-up, both restricted to Horn clauses. In other chapters we found useful applications for mixed top-down and bottom-up execution and for non Horn clauses to associate local data with subproblems and to achieve the cooperation of different procedures by case analysis. In this chapter we shall define a system which provides the ability to mix directions of execution and to deal adequately with non Horn clauses.

The new theorem-proving system has other desirable properties. All clauses are stored in a graph. An atom in the hypothesis of a clause is connected by a link in the graph to another atom in the conclusion of a different clause if the two atoms match. Storing clauses by means of the connections between them has the desirable consequence that all atoms in assertions and goal statements are directly linked to atoms in the procedures which can operate on them. When an atom in an assertion or goal statement is activated, the matching procedures are accessed without searching through the entire set of procedures but by following links in the connection graph. The intersection of bi-directional top-down and bottom-up strategies is facilitated since intersection can occur only by means of links in the graph.

The generation of new clauses is accomplished by activating links in the connection graph. Top-down execution is performed by activating a link between a goal statement and a procedure; bottom-up execution by activating a link between assertions and a procedure. When a link is activated the associated clause is generated, added to the graph and links between atoms in the new clause and atoms in the rest of the graph are constructed from the links on the parent clauses. The activated link is deleted from the graph.

At any time any link in the graph can be activated. If the link selected for activation connects two procedures, then the new clause is itself a procedure. Macro-processing of procedure calls in conventional programming languages is a special case of such derivation of new procedures.

If an atom in a clause is linked to no other atoms in the graph, then all links connected to that atom have already been activated and deleted from the graph. The entire clause containing the unlinked atom may also be deleted from the graph. This operation of deleting clauses containing unlinked atoms is of great importance. It means that a goal statement can be deleted when all procedures which apply to its selected atom have already been used to generate new goal statements. It means that an assertion can be deleted when all procedures containing matching atoms have been applied to the assertion in order to derive new assertions. When macro-processing is used to generate new procedures from old ones, it justifies deletion of the old procedures.

A precise definition of the connection graph theorem-proving system is given at the end of this chapter, after the informal description and examples.

The initial connection graph, deletion of clauses containing unlinked atoms and deletion of tautologies.

Figure 34 illustrates the initial connection graph for the set of clauses in the example of the fallible Greek.

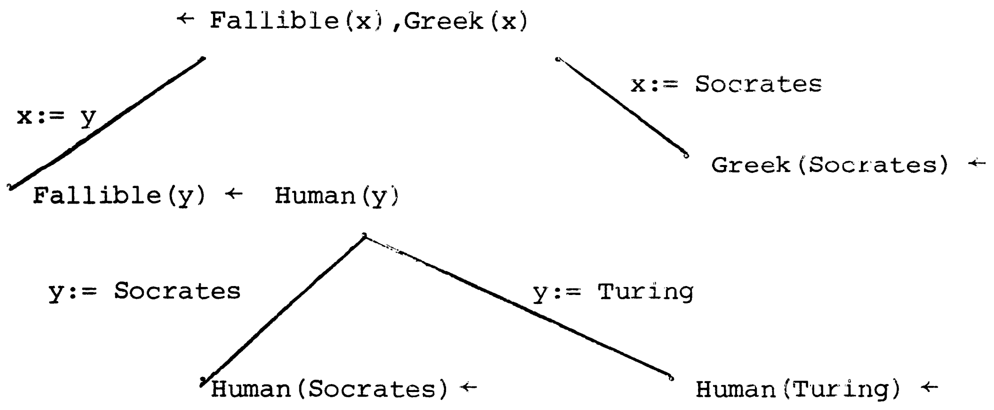
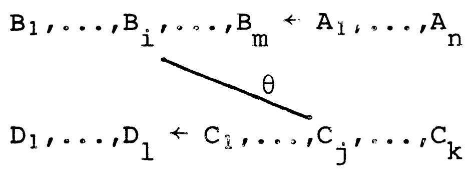


Figure 34. The initial connection graph for the fallible Greek example.

Given an initial set of clauses, the <u>initial connection graph</u> is obtained by inserting, for every pair of matching atoms B_i and C_j on opposite sides of the arrow in distinct clauses, a link between the two atoms:


$B_1,\ldots,B_i,\ldots,B_m \leftarrow A_1,\ldots,A_n$
θ
$D_1,\ldots,D_l \leftarrow C_1,\ldots,C_j,\ldots,C_k$

The link is labelled by the matching substitution θ. For the purpose of matching, variables in clauses are renamed in such a way that distinct clauses have distinct variables.

Robinson's purity principle {49} implies that any clause containing an unlinked atom can be deleted from a set of clauses without affecting its consistency (or inconsistency). Figure 35 illustrates application of the purity principle to transform an initial connection graph. The set of clauses in the initial connection graph is inconsistent if and only if the set of clauses in the transformed connection graph is inconsistent.

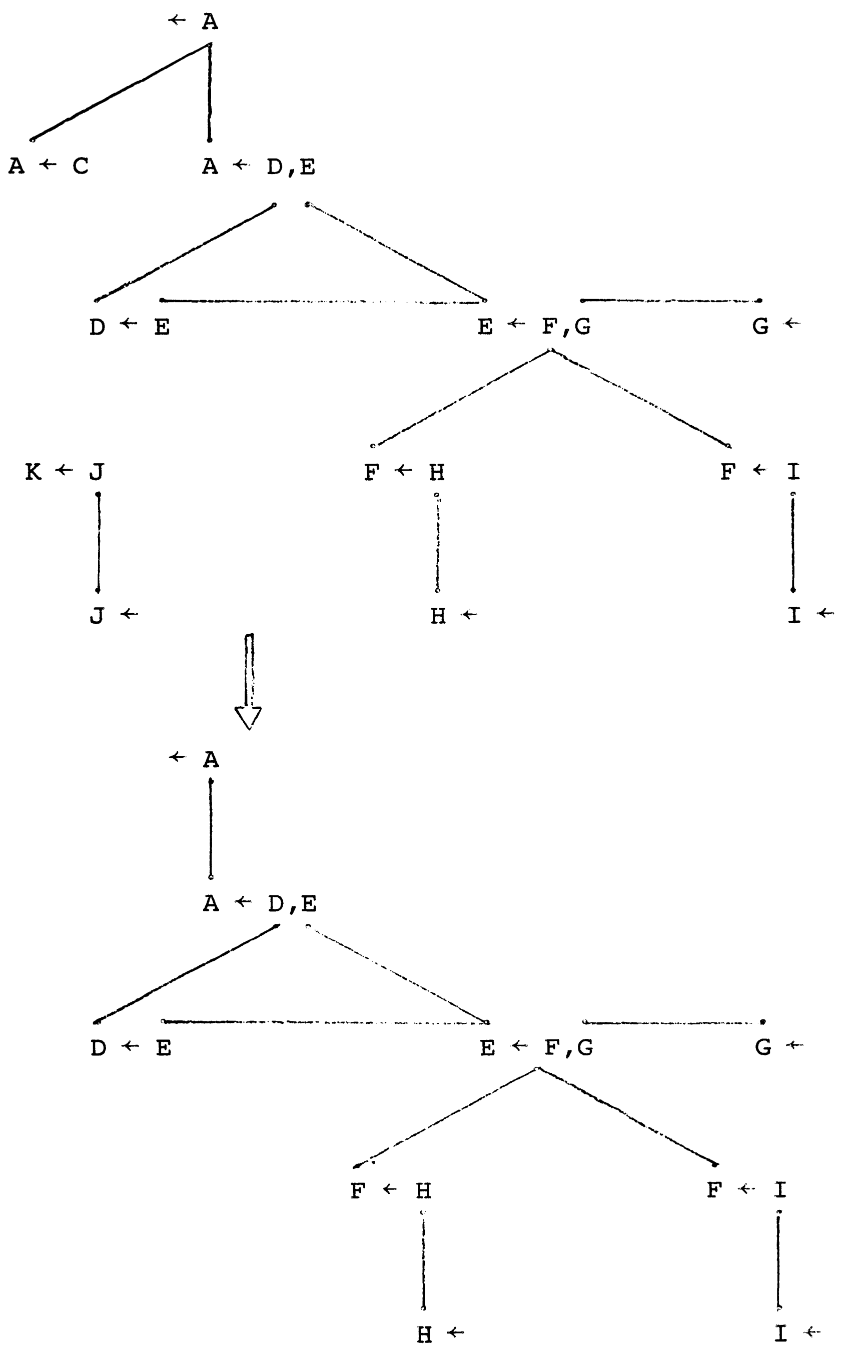


Figure 35. Transformation of the initial connection graph for the set of clauses in Figures 3-7. Clauses A ← C and K ← J are deleted because they contain unlinked atoms. After K ← J is deleted, the clause J ← contains an unlinked atom and is also deleted from the graph.

Without affecting the consistency of a set of clauses, a clause can be deleted if it is a tautology, i.e. if it contains identical atoms on different sides of the arrow. Figure 36 illustrates the transformation of a connection graph by successive deletion of tautologies and by application of the purity principle.

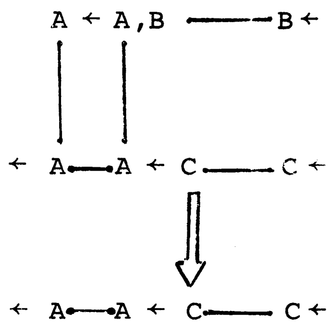


Figure 36. Deletion of a tautology. The clause A ← A,B is deleted because it is a tautology. But then B ← is deleted because it now contains an unlinked atom.

The activation of links in connection graphs.

The basic operation in connection graphs is that of deriving a new clause by activating a link connecting atoms in the parent clauses. The new clause is added to the graph and the activated link is deleted. The atoms in the new clause are linked to atoms in other clauses.

The new links are found not by searching the entire graph for atoms which match the new atoms in the new clause but by testing whether the atoms A which are linked to atoms B in the parent clauses match the atoms B' in the new clause which descend from B. Thus every new link descends from one or more old links connecting atoms in the parent clauses to other clauses in the graph.

Figures 37 and 38 illustrate the successive activation of links connected to goal statements, simulating a top-down refutation of the fallible Greek example.

Figures 39 - 43 illustrate a mixed top-down, bottom-up analysis for the parsing problem. Application of the purity principle in order to delete clauses takes place as soon as possible and is not displayed explicitly.

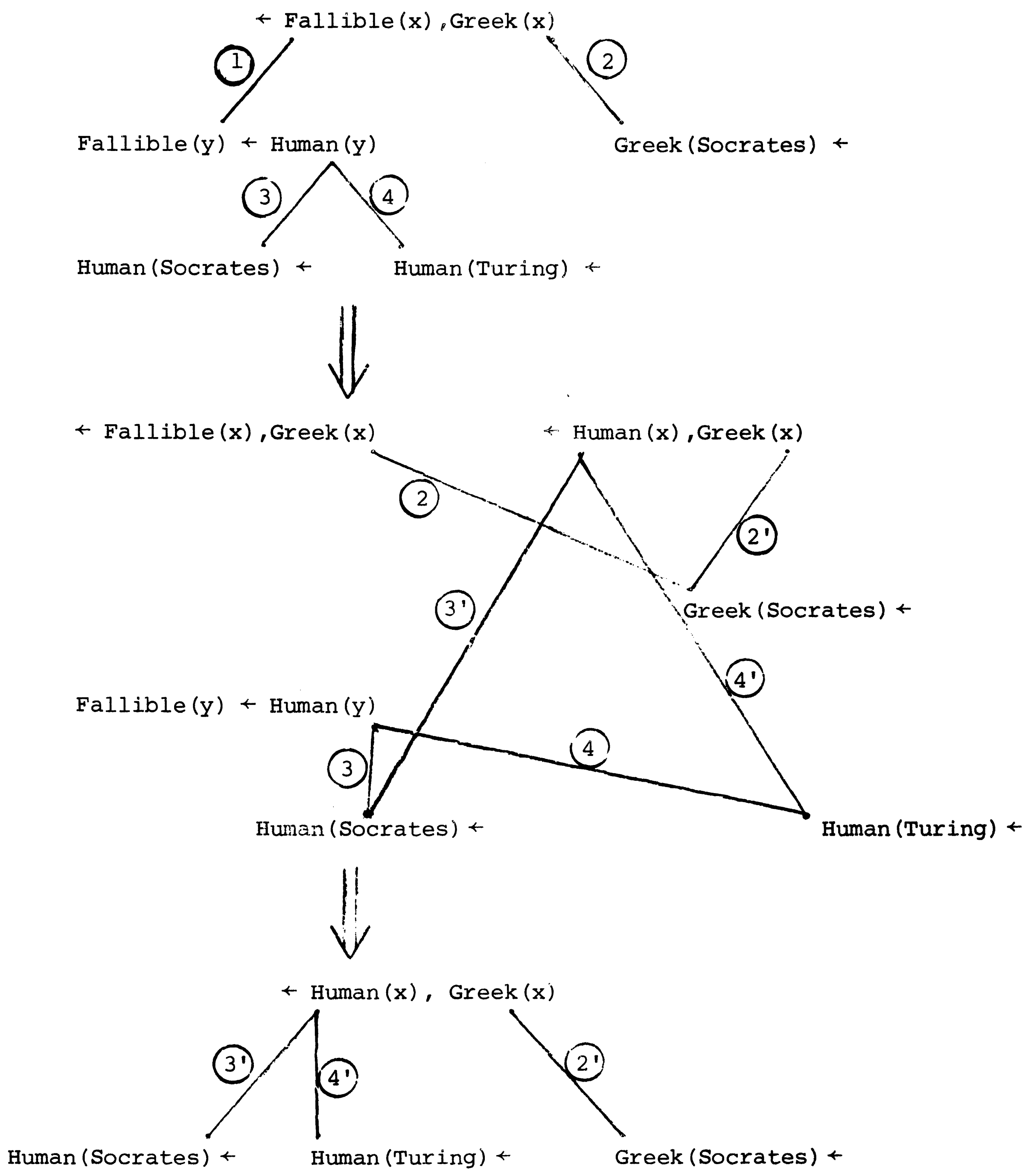


Figure 37. Top-down activation of links for the fallible **Greek example**. The second graph is obtained from the first by activating link ①. The new links ②', ③' and ④' descend from the old links ②, ③and ④ respectively. The third graph is obtained from the second by deleting clauses containing unlinked atoms. The links ②, ③ and ④ connected to the deleted clauses are also deleted. Substitutions labelling links are omitted in order to simplify the figure.

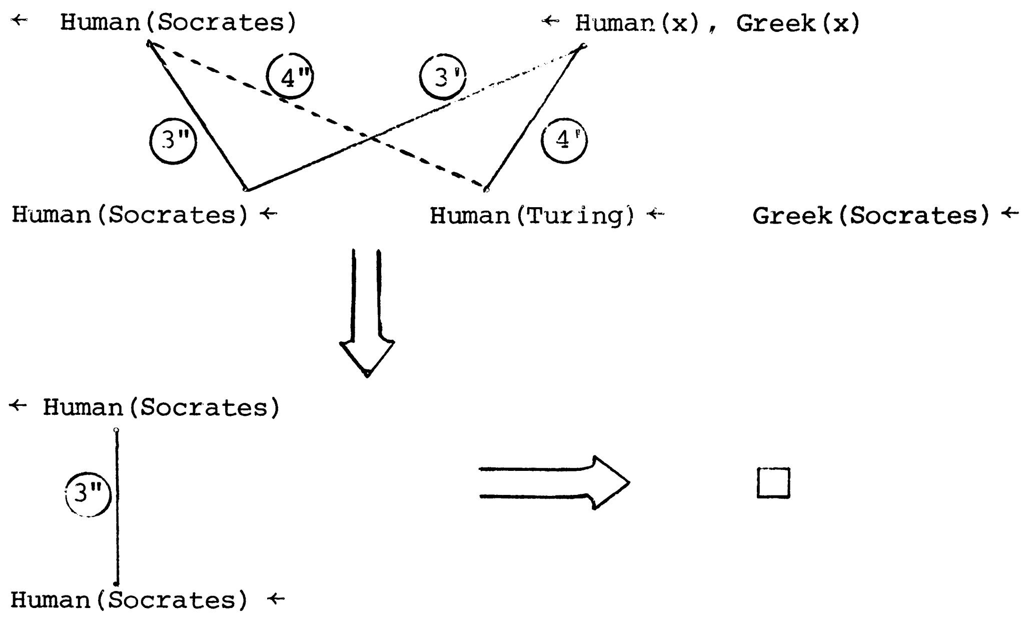


Figure 38. Continuation of the top-down activation of links initiated in Figure 37. The first graph in this figure is obtained by activating link (2') in the last graph of Figure 37. Link (3") descends from (3'). The attempt to construct a link (4") descending from (4') fails. The second graph is obtained from the first by successively deleting all clauses which contain unlinked atoms. Finally the null clause is obtained by activating the only link (3") which remains in the graph.

Typically such deletion of clauses is made possible by the activation and deletion of links which gives rise to unlinked atoms. In particular, if the activated link is the only link connected to an atom in one of the parent clauses, then that parent clause is deleted when the link is activated. In favourable circumstances, the activated link is the only link connected to both atoms at the ends of the link in the parent clauses. In such cases, both parent clauses are deleted when the link is activated.

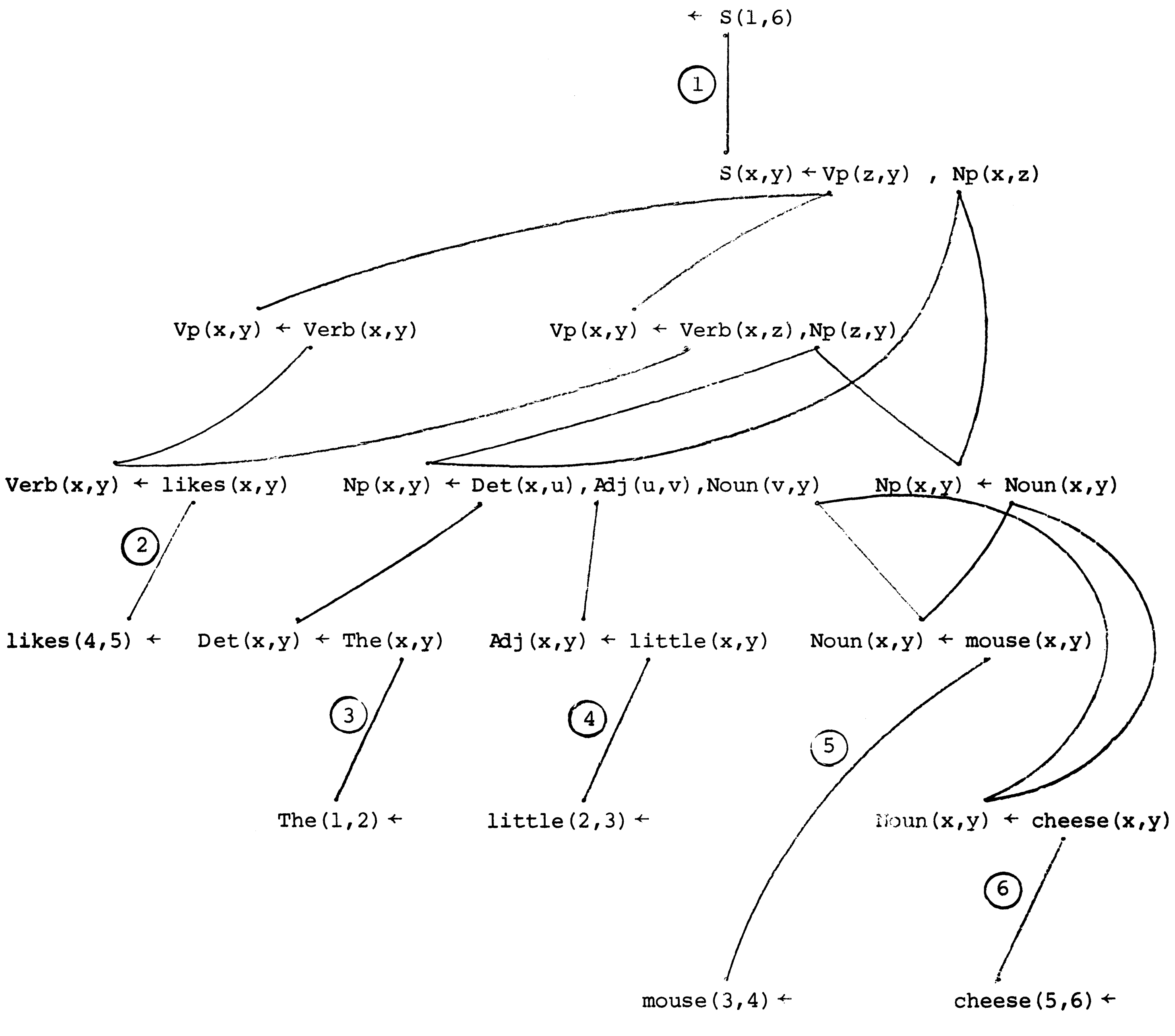
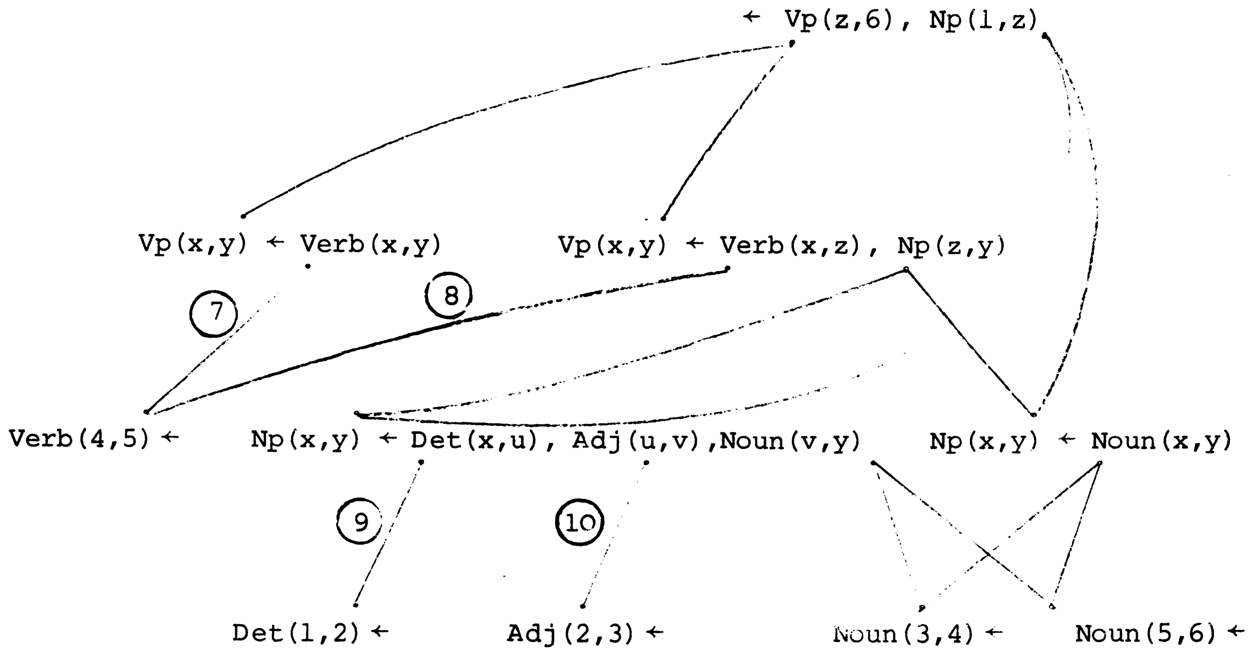
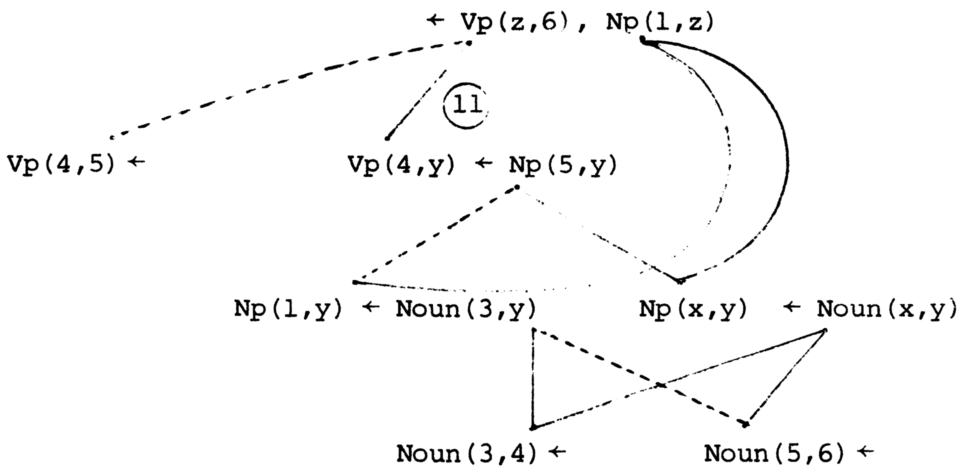


Figure 39. The initial connection graph for the parsing problem. The links (1) - (6) are selected for activation because deletion of those links allows all the parent clauses to be deleted from the graph. Activation of (1) initiates a top-down analysis of the sentence and activation of (2) - (6) initiates a bottom-up analysis.



<u>Figure 40</u>. The connection graph which results from activating links (1) - (6) in Figure 39. Activation of links (7) and (8) generates two new clauses and results in the deletion of the three parent clauses. Activation of (9) results in the deletion of both parent clauses. Subsequent activation of the descendant of link (10) in the new clause results in deletion of both parents, one of them the clause just generated. Activation of (7) - (10) corresponds to bottom-up analysis.



<u>Figure 41</u>. The connection graph which results from activating (7) - (10) in Figure 40. The dotted lines represent unsuccessful attempts to add new links, descending from old links, to the graph. Activation of (11) corresponds to top-down analysis and results in the deletion of both parent clauses.

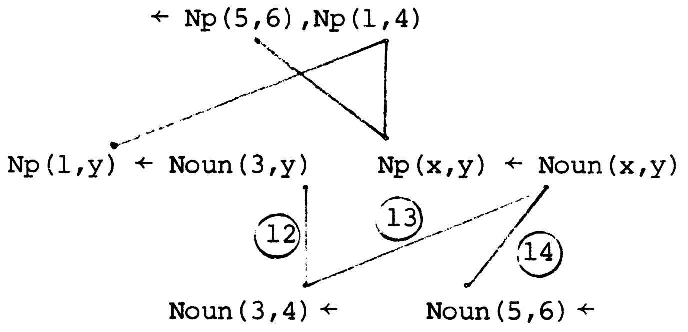


Figure 42. The connection graph which results from activating (11) in Figure 41. Activation of (12) - (14) corresponds to bottom-up analysis, generates three new assertions but results in the deletion of the four parent clauses.

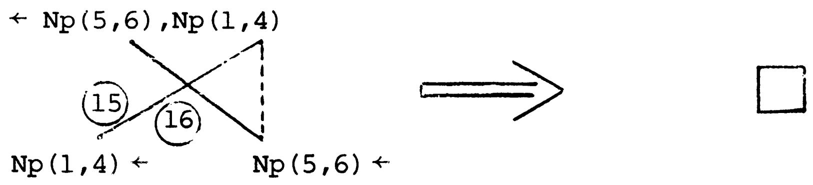


Figure 43. The connection graph which results from activating (12) - (14) in Figure 42. Activation of link (15) and then the descendant of (16) connected to the new clause results in the generation of the null clause and in the deletion of all other clauses.

Resolution.

The selection of links for activation in a connection graph is like the selection of atoms in a goal statement in the top-down interpretation: at every stage, any link may be selected. The new clause obtained by activating a link is called a resolvent of the two parent clauses connected by the link. The resolvent associated with a link between two clauses

$$\begin{tikzcd} \mathcal{B} \cup \{B\} \arrow[drr, dash] & \leftarrow \mathcal{A} & \\ \mathcal{D} \leftarrow \mathcal{C} \cup & & \{C\} \end{tikzcd}$$

is the clause

$$(\mathcal{B} \cup \mathcal{D} \leftarrow \mathcal{A} \cup \mathcal{C})\,\theta$$

Resolution as originally defined {49} incorporates the additional operation of factoring clauses. A clause $C\theta$ is a factor of a clause C,

$$\mathcal{B}_1 \cup \dots \cup \mathcal{B}_m \leftarrow \mathcal{A}_1 \cup \dots \cup \mathcal{A}_n ,$$

if θ is a most general simultaneous unifier of the family of sets of atoms

$$\{ \mathcal{A}_1, \dots, \mathcal{A}_n, \mathcal{B}_1, \dots, \mathcal{B}_m \}$$

For example the clause Grandparent(x,y) ← Parent(x,z),Parent(z,y) has two factors:

Grandparent(x,y) ← Parent(x,x) and

Grandparent(x,y) ← Parent(x,z),Parent(z,y).

Later we shall see an example which requires the use of the factoring operation.

Top-down generation of goal statements and bottom-up generation of assertions are special cases of resolution. In addition, resolution deals with the activation of links between two procedures and with the activation of links connected to non Horn clauses.

Figures 44 and 45 and Figures 46-48 illustrate alternative connection graph refutations of the non Horn clause, Robert-is-always-working example. The first refutation (Figures 44 and 45) proceeds basically in the top-down direction, whereas the second refutation (Figures 46-48) proceeds basically bottom-up.

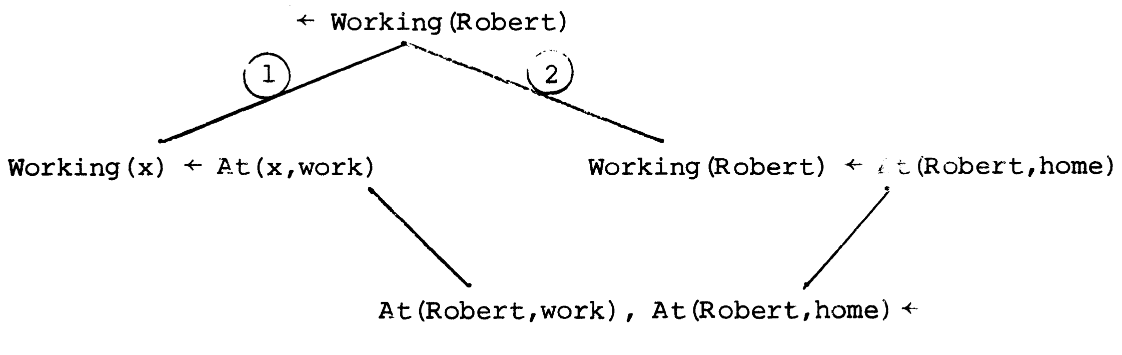

← Working(Robert)
1
2
Working(x) ← At(x,work)
Working(Robert) ← At(Robert,home)
At(Robert,work), At(Robert,home) ←

Figure 44. The initial connection graph for the Robert-is-always-working example. Activation of links (1) and (2) initiates a top-down analysis and results in the deletion of the three parent clauses.

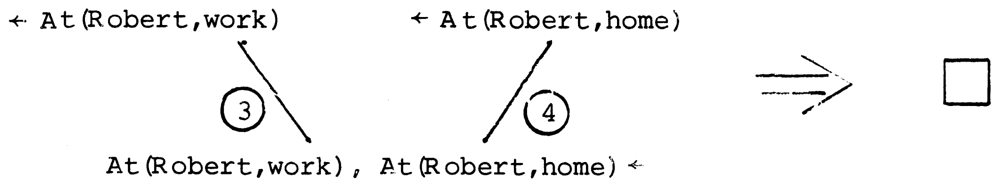
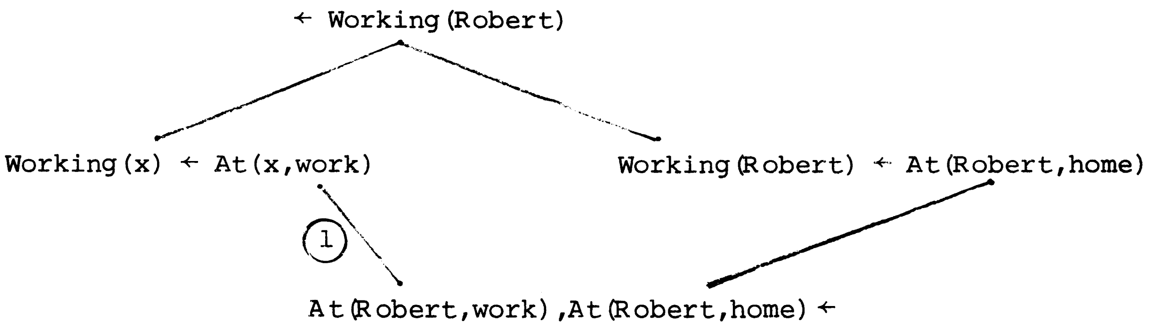
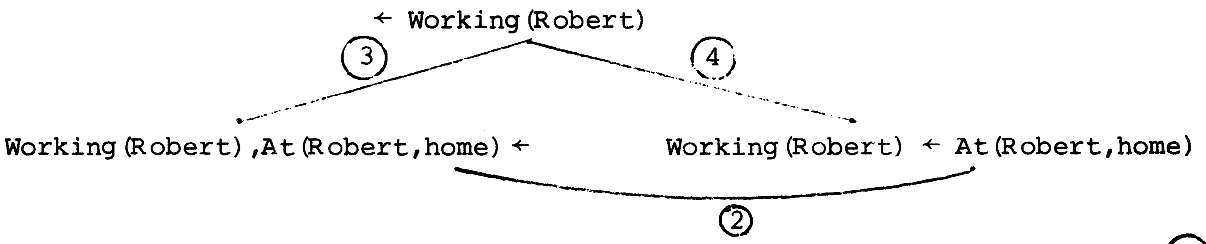


Figure 45. The connection graph which results from the activation of links (1) and (2) in Figure 44. Activation of (3) results in the deletion of the two parent clauses. Activation of the descendant of (4) in the resolvent results in the generation of the null clause and in the deletion of all other clauses.



← Working(Robert)
Working(x) ← At(x,work)
Working(Robert) ← At(Robert,home)
1
At(Robert,work),At(Robert,home) ←

Figure 46. The initial connection graph for the Robert-is-always-working example. Activation of (1) initiates a kind of bottom-up analysis.



← Working(Robert)
3
4
Working(Robert),At(Robert,home) ←
Working(Robert) ← At(Robert,home)
2

Figure 47. The connection graph which results from the activation of (1) in Figure 46. Activation of (2) continues the kind of bottom-up analysis begun in Figure 45.

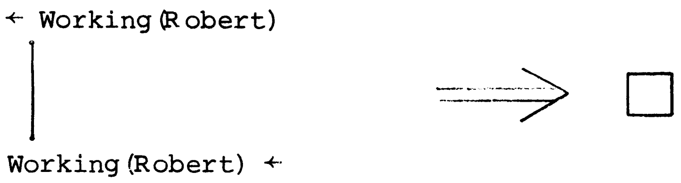
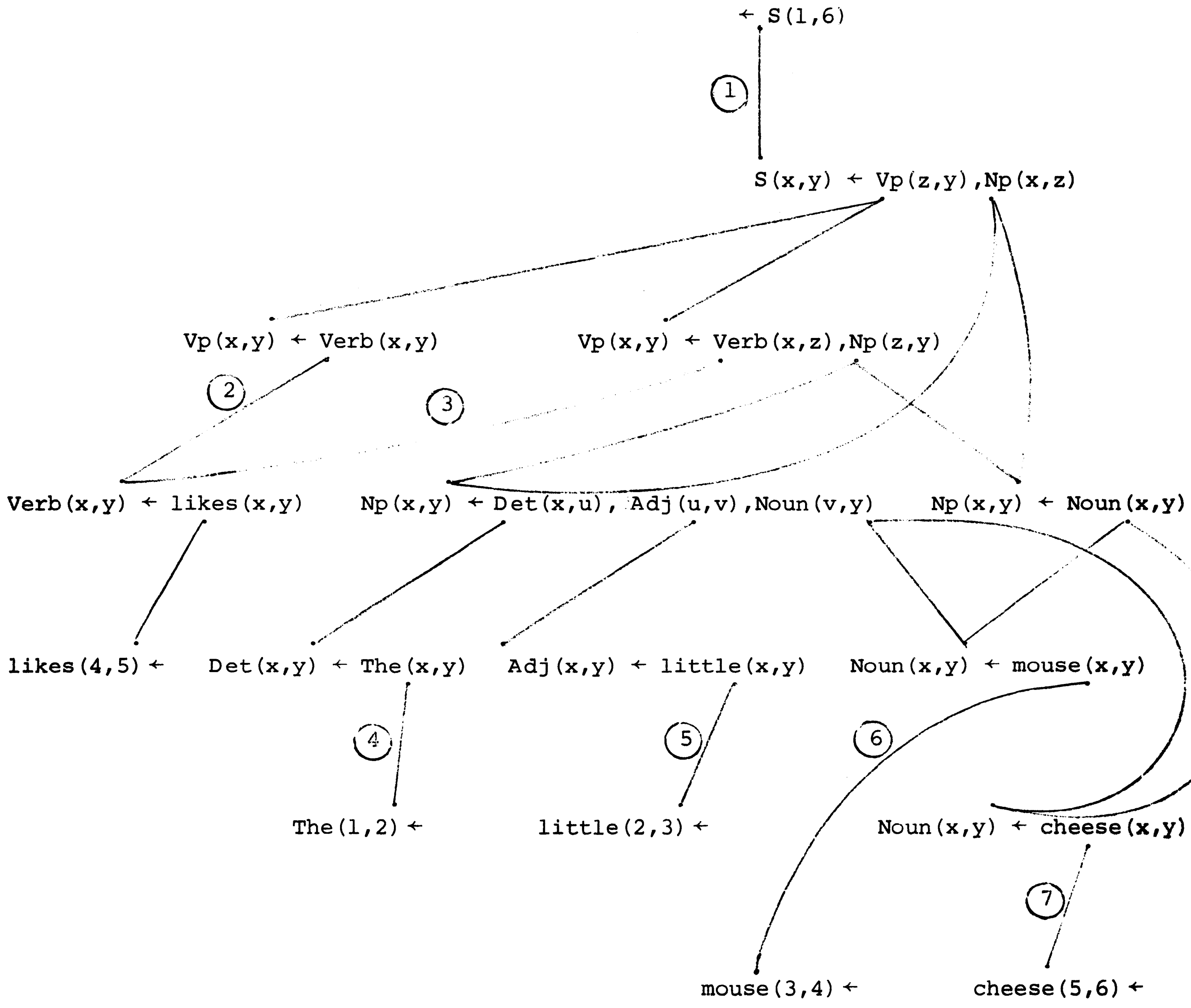


Figure 48. The connection graph which results from the activation of (2) in Figure 47. The only link in this graph descends from both links (3) and (4) in Figure 47. Activation of this link results in the generation of the null clause.

Figures 49-53 illustrate an alternative refutation of the set of clauses in the parsing problem. The activation of links (2),(3) and (13) in this example derives new procedures from old ones.



<u>Figure 49</u>. The initial connection graph for the parsing problem. Activation of links (1) - (7) results in the deletion of all the parent clauses. Activation of (2) and (3) derives new procedures from old ones and is an example of <u>macroprocessing</u>: all procedure calls Verb(s,t) are eliminated from the program. The definition of the Verb procedure is also eliminated.

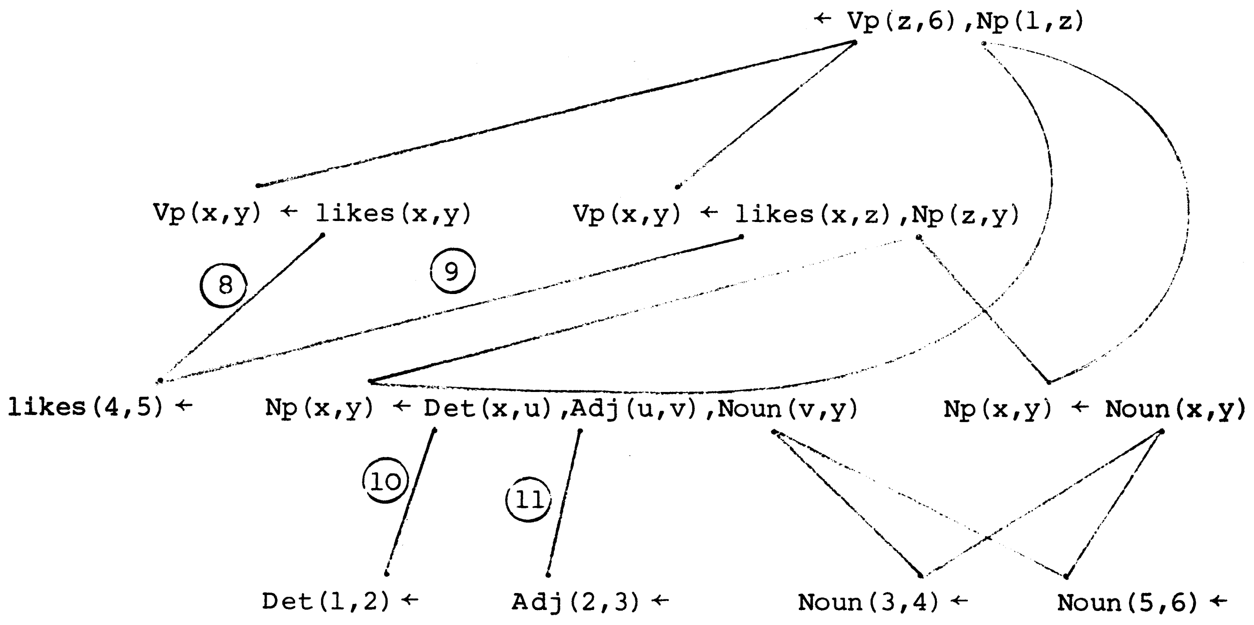


Figure 50. The connection graph which results from activating (1) - (7) in Figure 49. Activation of (8),(9) and (10) and then the descendant of (11) results in the deletion of all parent clauses.

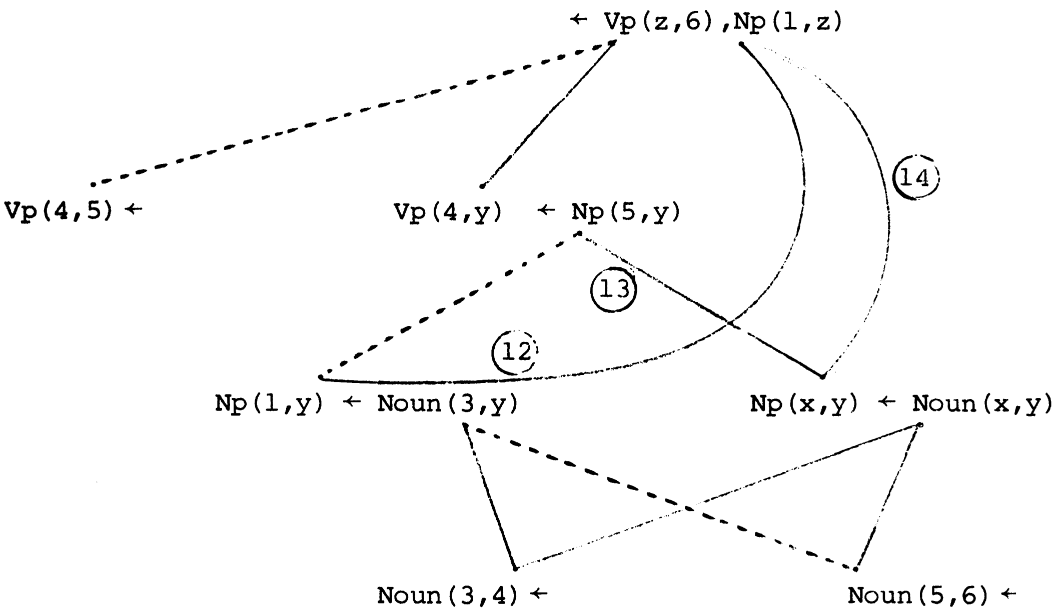


Figure 51. The connection graph which results from activating (8) - (11) in Figure 50. Activation of links (12) and (14) corresponds to top-down analysis, whereas activation of (13) derives a new procedure from old procedures.

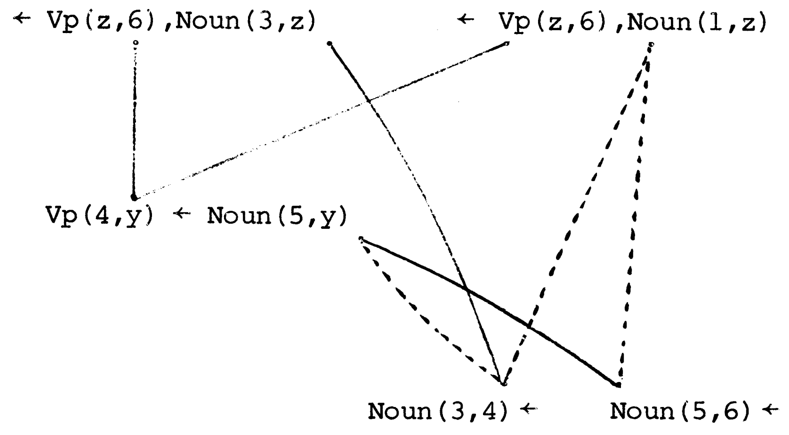


Figure 52. The connection graph which results from activating links ⑫ - ⑭ in Figure 51.

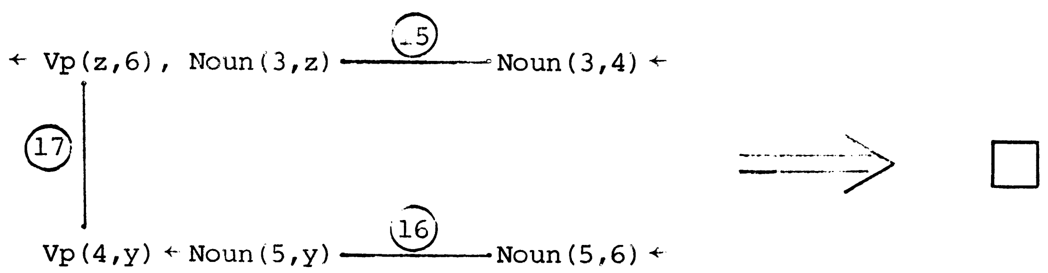


Figure 53. The connection graph which results from deleting clauses containing unlinked atoms in the graph of Figure 52. Activating ⑮ and ⑯ and then the descendant of ⑰ results in generation of the null clause.

Self-resolving clauses.

Before defining the connection graph theorem-proving system more precisely we need to illustrate the treatment of self-resolving clauses and factoring.

A self-resolving clause is one which resolves with a copy of itself. For example

Append(cons(x,y),z,cons(x,u)) ← Append(y,z,u)

resolves with the copy

Append(cons(x',y'),z',cons(x',u')) ← Append(y',z',u')

A matching substitution is

y:= cons(x',y'), z:= z', u:= cons(x',u')

and the corresponding resolvent is

Append(cons(x,cons(x',y')) , z', cons(x,cons(x',u'))) ← Append(y',z',u').

Figures 54-56 illustrate a combined top-down, bottom-up strategy for appending the list {3} to {2,1}.

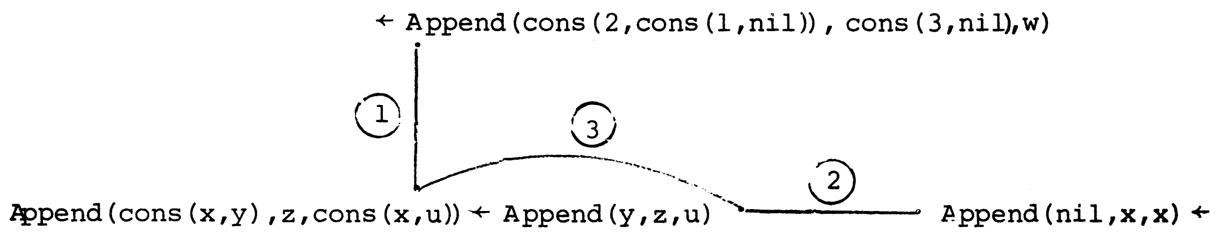

← Append(cons(2,cons(1,nil)), cons(3,nil),w)
1
3
2
Append(cons(x,y),z,cons(x,u)) ← Append(y,z,u)
Append(nil,x,x) ←

Figure 54. The initial connection graph for appending {3} to {2,1}. The link ③ is a "pseudo-link" representing the real link between atoms in different copies of the self-resolving clause. Pseudo-links are not activated directly but are used to help construct the new links connected to atoms in newly generated clauses. Activation of link ① initiates a top-down execution strategy. The new derived goal statement replaces the old goal statement.

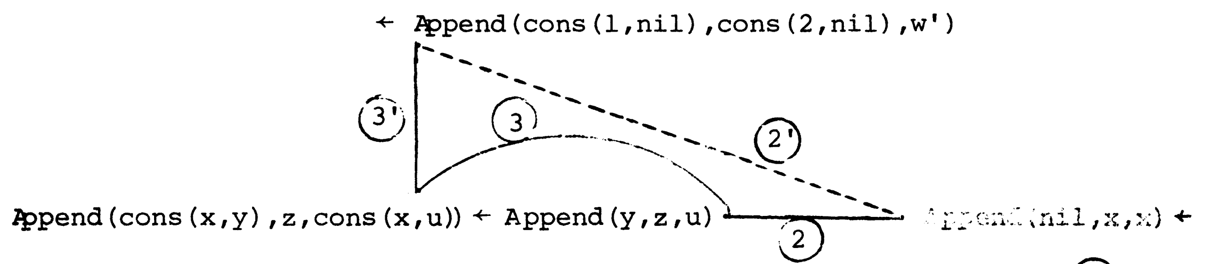
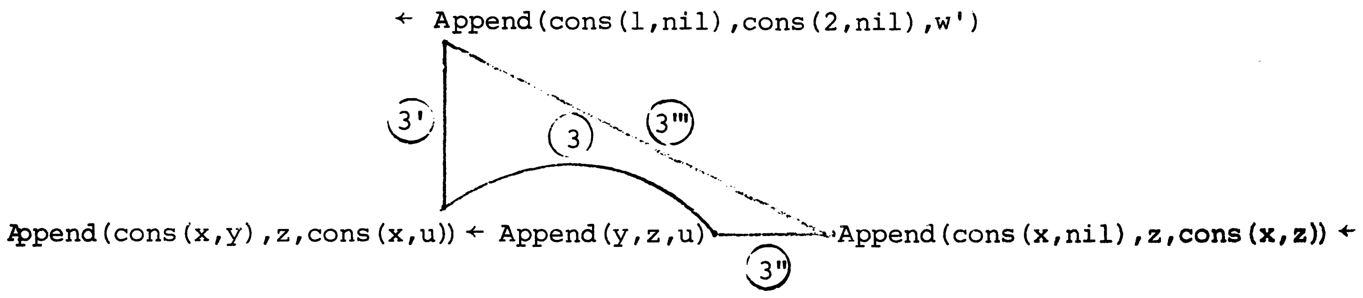

← Append(cons(1,nil),cons(2,nil),w')
3'
3
2'
2
Append(cons(x,y),z,cons(x,u)) ← Append(y,z,u)
Append(nil,x,x) ←

Figure 55. The connection graph which results from activating ① in Figure 54. The new link (3') descends from the pseudo-link ③. The unsuccessful link (2') descends from ②. Activation of link ② results in the deletion of the parent assertion and executes the recursive Append procedure bottom-up.


← Append(cons(1,nil),cons(2,nil),w')
3'
3
3'''
Append(cons(x,y),z,cons(x,u)) ← Append(y,z,u)
Append(cons(x,nil),z,cons(x,z)) ←
3''

<u>Figure 56</u>. The connection graph which results from activating (2) in Figure 57. The new link (3") descends from the pseudo-link (3). The new link (3"') descends from (3'). Activation of (3"') intersects the top-down and bottom-up analyses and results in the generation of the null clause.

<u>Factoring and the soldiers</u>.

In the following example the factoring operation is necessary in order to obtain a refutation:

Suppose that

(1) all soldiers kill all people who do not kill themselves, and

(2) all soldiers kill only people who do not kill themselves.

Show that

(3) there are no soldiers.

In a non clausal form of predicate logic (1) and (2) become (1') and (2') respectively:

(1') Kill(x,y) ← not-Kill(y,y),Sold(x)

(2') not-Kill(y,y) ← Kill(x,y),Sold(x)

In clausal form (1') and (2') are respectively (1") and (2"). The negation of (3) is (3"):

(1") Kill(x,y),Kill(y,y) ← Sold(x)

(2") ← Kill(x,y),Kill(y,y),Sold(x)

(3") Sold(Robert) ←

Here Robert is an arbitrary name for the soldier whose existence is expected to contradict (3). The set of clauses {(1"),(2"),(3")} is inconsistent. Figures 57-59 illustrate an unsuccessful attempt to obtain a connection graph refutation without use of the factoring operation. The successful refutation obtained in Figure 59 is made possible by adding to the graph the factor

Kill(Robert,Robert) ←

of the clause

Kill(Robert,y) , Kill(y,y) ←

already in the graph.

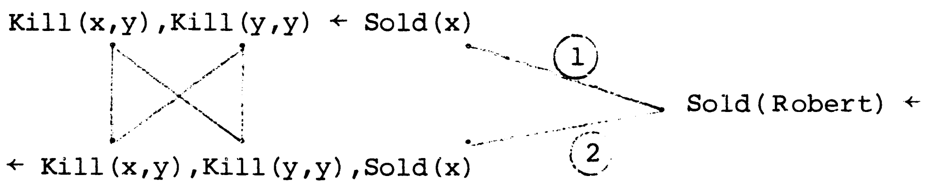


Figure 57. The initial connection graph for the soldiers example. Activation of $\textcircled{1}$ and $\textcircled{2}$ results in the deletion of the three parent clauses.

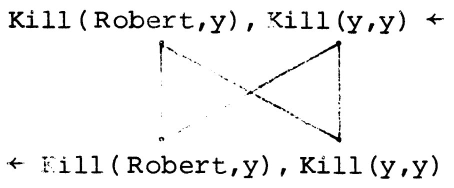


Figure 58. The connection graph obtained by activating $\textcircled{1}$ and $\textcircled{2}$ in Figure 57. There exists no way of activating links which eventually results in the generation of the null clause. In order to obtain a refutation it is necessary to add to the graph a factor of either one or both of the clauses in the graph.

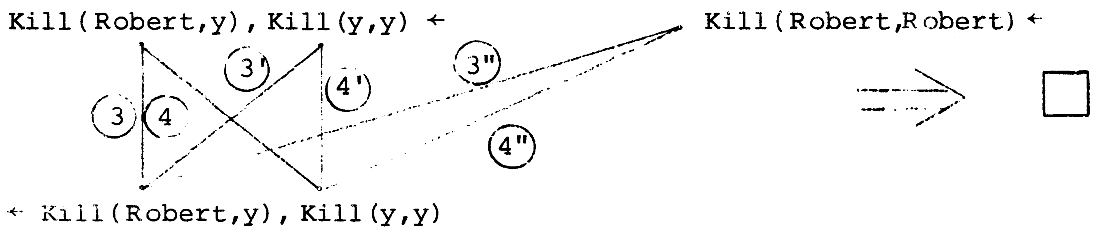


Figure 59. The graph which results from adding a factor of one of the clauses in the graph. The new link $\textcircled{3''}$ descends from both $\textcircled{3}$ and $\textcircled{3'}$; the link $\textcircled{4''}$ descends from $\textcircled{4}$ and $\textcircled{4'}$. Activation of link $\textcircled{3''}$ and then of the descendant of $\textcircled{4''}$ results in generation of the null clause.

Definition of the connection graph theorem-proving system.

The following definition makes more precise the connection graph theorem-proving system illustrated in the preceding examples. In order to simplify the definition, pseudo-links are not made explicit.

(1) Initialise the graph. Let S be the set of all factors of clauses in the initial set of clauses. Form the initial connection graph by inserting a link between all matching atoms on different sides of the arrow in different clauses in S. Label each link by the matching substitution

(2) Repeat the following procedure until the null clause is generated.

  (a) Select a link in the graph. Activate it by generating the associated resolvent and all its factors. Delete the activated link. Add the resolvent and its factors to the graph.

  (b) Connect the atoms in the new clauses by links to other atoms in the graph.

  Suppose that an atom $L\theta$ in a new clause descends from an atom L in one of the parent clauses. Suppose that the old atom L is connected by a link to an occurrence of an atom K. If the substitution σ labelling the link between L and K is compatible with θ, then insert a link between the new atom $L\theta$ and the occurrence of K. Label the link by the matching substitution.

  Suppose that an atom in a new clause matches an atom on the other side of the arrow in another new clause or in one of the parent clauses. Insert a link between the matching atoms and label it by the matching substitution.

  (c) Delete from the graph any tautology and any clause containing an unlinked atom. Delete all links connected to atoms in the deleted clause.

The generation of all factors of input clauses and of resolvents is excessively redundant. Redundancy can be controlled by imposing restrictions on the factoring operation. The following restrictions have proved useful in other theorem-proving systems.

(1) Idem-factoring ({25} , {21}).
Only generate factors $C\theta$ of non Horn clauses C of the form

$$\mathcal{B}_1 \cup \mathcal{B}_2 \leftarrow$$

where θ is a most general unifier of $\mathcal{B}_1$. Do not activate any links connected to atoms in $\mathcal{B}_2\theta$ in the factor $C\theta$.

(2) m-factoring ({21} , {26}).
Generate all factors of input clauses. Generate only those factors of resolvents which do not identify distinct atoms from the same parent.

Neither of these factoring methods makes special use of the connection graph structure. It may be that more satisfactory factoring methods will be obtained by relating the generation of factors to the presence of certain links in the connection graph.

The selection and activation of links is non-deterministic. Different sequences for scheduling the activation of links should lead to the same result and should differ only by leading to that result with more or less efficiency. In order to secure this objective it is necessary to avoid methods of selection which indefinitely postpone the activation of certain links needed for a refutation. This can be achieved by employing selection methods which eventually select every link for activation.

In addition to employing negative criteria which guard against the dangers of indefinite postponement, it is necessary to use positive criteria which prefer the selection of one link to another. These criteria can be formulated by the user and conveyed to the theorem-prover in an auxiliary control language. Or they can be general-purpose strategies which are pre-programmed into the theorem-prover. In either case, it is preferable in general to select links whose activation least complicates the graph.

Simplification takes place when the resolvent replaces both its parents or when it replaces one of its parents but contains fewer atoms and links than the deleted parent. The selection of links, which temporarily complicate the graph but eventually simplify it, is facilitated by the look-ahead computations described in {23}. Preference should also be given to the selection of links connected to clauses which descend from the initial goal statement or problem-specific assertions. The combination of these two preference strategies combines in connection graphs the principle of procrastination with the Pohl and Bledsoe heuristics.

A more detailed investigation of connection graphs, emphasising their historical relationship with top-down resolution systems, is reported in the original publication {23}.

REFERENCES.

{1} Anderson, D.B. and Hayes, P.J. 'An arraignment of theorem-proving or the logicians' folly'. D.C.L. Memo No 54, University of Edinburgh, 1972.

{2} Baxter, L.D. 'An efficient unification algorithm'. Applied Analysis and Computer Science Technical Report CS-73-23, University of Waterloo, 1973.

{3} Bledsoe, W.W. 'Splitting and reduction heuristics in automatic theorem-proving'. _Artificial Intelligence 2_, 1971, pp.55-77.

{4} Bledsoe, W.W. and Bruell, P. 'A man-machine theorem-proving system'. _Third International Joint Conference on Artificial Intelligence_, 1973, pp. 56-65.

{5} Bobrow, D.G. and Wegbreit, B. 'A model for control structures for artificial intelligence programming languages'. _Proceedings of IJCAI_, Stanford, California, August 1973.

{6} Boyer, R.S. and Moore J S. 'The sharing of structure in theorem-proving programs'. _Machine Intelligence 7_, (eds Meltzer, B. and Michie, D.) Edinburgh University Press, 1972, pp. 101-16.

{7} Chang, C.L. and Slagle, J.R. 'An admissable algorithm for searching and/or graphs'. _Artificial Intelligence 2_, 1971, pp. 117-28.

{8} Colmerauer, A. 'Les systèmes-Q ou un formalisme pour analyser et synthetiser des phrases sur ordinateur'. Publication interne No 43, Dépt d'Informatique, Université de Montréal.

{9} Daniel, L. 'And-or graphs and critical path'. To appear in the _Proceedings of IFIP 1974_.

{10} Davies, J. 'POPLER 1.5 Reference Manual'. T.P.U. Report No. 1, University of Edinburgh, May 1973.

{11} Emden, M.H. van 'Predicate logic as a specification language for automatic programming'. In preparation.

{12} Ernst, G.W. 'The utility of independent subgoals in theorem-proving'. _Information and Control_, April 1971.

{13} Feldman, J.A. et al. 'Recent developments in SAIL - an ALGOL-based language for artificial intelligence'. _FJCC_, 1972.

{14} Fikes, R.E. and Nilsson, N.J. 'STRIPS: a new approach to the application of theorem-proving to problem solving'. _Artificial Intelligence 2_, 1971, pp. 189-208.

REFERENCES (continued)

{15} Foster, J.M. List Processing, Macdonald/Elsevier, 1967.

{16} Foster, J.M. Automatic Syntactic Analysis, Macdonald/Elsevier, 1970.

{17} Hayes, P.J. "Computation and deduction". Internal memo, University of Essex, 1973.

{18} Hewitt, C. 'Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot.' AI Memo No 251, MIT , Project MAC, April 1972.

{19} Hoare, C.A.R. 'Algorithm 64'. Comm A.C.M., Vol. 4, 1961, p. 321.

{20} Kowalski, R. 'Search strategies for theorem-proving'. Machine Intelligence 5, (eds. Meltzer, B. and Michie, D.) Edinburgh University Press, 1969, pp. 181-201

{21} Kowalski, R. 'Studies in the completeness and efficiency of theorem-proving by resolution'. Ph.D. Thesis, University of Edinburgh, 1970.

{22} Kowalski, R. 'And-or graphs, theorem-proving graphs and bi-directional search'. Machine Intelligence 7 , (eds Meltzer, B. and Michie, D.) Edinburgh University Press, 1972, pp. 167-94.

{23} Kowalski, R. 'A proof procedure using connection graphs'. D.C.L. Memo No 74, University of Edinburgh, 1973.

{24} Kowalski, R. 'Predicate logic as programming language'. To appear in the Proceedings of IFIP 1974.

{25} Kowalski, R. and Hayes, P.J. 'Semantic trees in automatic theorem-proving'. Machine Intelligence 4, (eds Meltzer, B. and Michie, D.) Edinburgh University Press, 1968, pp.87-101.

{26} Kowalski, R. and Kuehner, D. 'Linear resolution with selection function'. Artificial Intelligence 2, 1971, pp. 227-60.

{27} Kuehner, D. 'Some special purpose resolution systems'. Machine Intelligence 7, (eds. Meltzer, B. and Michie, D.), Edinburgh University Press, 1972, pp. 117-28.

{28} Loveland, D.W. 'A simplified format for the model-elimination theorem-proving procedure'. J.A.C.M., vol. 16, 1969, pp. 349-63.

{29} Loveland, D.W. 'A linear format for resolution'. Proceedings IRIA Symposium on Automatic Demonstration, Versailles, France, Springer-Verlag, 1970, pp. 147-62.

{30} Loveland, D.W. 'A unifying view of some linear Herbrand procedures'. J.A.C.M. Vol. 19, 1972, pp.366-84

REFERENCES (continued)

{31} Loveland, D.W. and Stickel, M.E. 'A hole in goal trees: some guidance from resolution theory'. Third International Joint Conference on Artificial Intelligence, 1973, pp. 153-161.

{32} Luckham, D. 'Refinement theorems in resolution theory'. Proceedings IRIA Symposium on Automatic Demonstration, Versailles, France, Springer-Verlag, 1970, pp. 162-90.

{33} Martelli, A. and Montanari, U. 'Additive and/or graphs'. Third International Joint Conference on Artificial Intelligence, Stanford University, 1973, pp. 1-11.

{34} McCarthy, J. et al. 'LISP 1.5 Programmers' Manual'. MIT Press 1962.

{35} McCarthy, J. 'Programs with common sense'. Reprinted in Semantic Information Processing (ed. Minsky, M.) MIT Press, 1968.

{36} Meltzer, B. 'Theorem-proving for computers: some results on resolution and renaming'. Computer Journal, Vol. 8, 1966, pp. 341-3.

{37} Michie, D., Ross, R. and Shannan, G.J. 'G-deduction'. Machine Intelligence (eds Meltzer, B. and Michie, D.) Edinburgh University Press, 1972, pp.141-65.

{38} Minker, J., Fishman, D.H. and McSkimin, J.R. 'The Q* algorithm - a search strategy for a deductive question-answering system'. Third International Joint Conference on Artificial Intelligence, Stanford University, 1973, pp. 31-7.

{39} Minsky, M.L. 'Descriptive languages and problem solving'. Semantic Information Processing (ed. Minsky, M.) MIT Press, 1968.

{40} Moore, J S. 'Computational logic: structure sharing and proof of program properties, Part I'. D.C.L. Memo No 67, University of Edinburgh.

{41} Nilsson, N.J. 'Searching problem solving and game playing trees for minimal cost solutions'. Proceedings IFIP Congress, 1968, pp. 125-30.

{42} Nilsson, N.J. 'Problem solving methods in artificial intelligence'. McGraw-Hill, 1971.

{43} Overbeek, R.A. 'Review of "Some special purpose resolution system".' Computing Reviews, Vol. 14, No 10, 1973, p. 469.

{44} Plotkin, G.D. 'Building-in equational theories'. Machine Intelligence 7 (eds Meltzer, B. and Michie, D.)Edinburgh University Press, 1972, pp. 73-90.

{45} Pohl, I. 'Bi-directional search'. Machine Intelligence 7, (eds Meltzer, B. and Michie, D.) Edinburgh University Press, 1972, pp. 127-40.

REFERENCES (continued)

{46} Reboh, R. et al. 'Study of automatic theorem-proving'. Artificial Intelligence Center Technical Note 75, Stanford Research Institute, November 1972.

{47} Reboh, R. and Sacerdoti, E. 'A preliminary QLISP Manual'. SRI AI Center Technical Note 81, August 1973.

{48} Reiter, R. 'Two results on ordering for resolution with merging and linear format'. J.A.C.M. Vol. 15, No 4, 1971, pp. 630-46.

{49} Robinson, J.A. 'A machine-oriented logic based on the resolution principle'. Journal of the Association for Computing Machinery, Vol. 12, 1965, pp. 23-41.

{50} Robinson, J.A. 'Automatic deduction with hyper-resolution'. International Journal of Computer Mathematics, Vol. 1, 1965, pp. 227-34.

{51} Robinson, J.A. 'Computational logic: the unification computation'. Machine Intelligence 6, (eds Meltzer, B. and Michie, D.) Edinburgh University Press, 1971, pp. 63-72.

{52} Rulifson, J.F. et al. 'QA4: a procedural calculus for intuitive reasoning'. SRI AI Center Technical Note 73, November 1973.

{53} Slagle, J.R. 'Automatic theorem-proving with renamable and semantic resolution'. J.A.C.M. Vol. 14, 1967, pp. 687-97.

{54} Slagle, J.R. 'Heuristic search programs'. Theoretical Approaches to Non-numerical problem-solving, Lecture notes in Operations Research and Mathematical Systems No 28, Springer-Verlag, 1970.

{55} Sussman, G.J. and Winograd, T. 'MICRO-PLANNER Reference Manual'. AI Memo No. 203, MIT Project MAC, July 1970.

{56} Sussman, G.J. and McDermott, D.V. 'Why conniving is better than planning'. AI Memo No 255A, MIT Project MAC, April 1972.

{57} Sussman, G.J. 'A computational model of skill acquisition'. Ph.D. Thesis, MIT, 1973.

{58} Venurini-Zilli, M. 'Complexity of the unification algorithm'. Internal Report, Istituto per le Applicazioni del Calcolo, Rome, 1973.

{59} Wos, L., Carson, D.F. and Robinson, G.A. 'The unit preference strategy in theorem-proving'. Proceedings of the AFIPS 1964 Fall Joint Computer Conference, Vol. 26, pp. 616-21.

ACKNOWLEDGEMENTS.

These lecture notes were prepared for a course of six lectures on predicate logic as a programming language in artificial intelligence in the Advanced Course on The Foundations of Computer Science, 20-30 May 1974, Amsterdam, The Netherlands. I am indebted to Professor de Bakker for his invitation to contribute to the lecture course and for his encouragement in the preparation of these lecture notes. Earlier versions of these lectures were presented in June 1973 at the Polytechnic of Milan and the Laboratorio de Elettronica Industriale in Padua, and in October 1973 in the School of Artificial Intelligence, University of Edinburgh.

The material in these lecture notes owes much to the beneficial interactions I have had with my colleagues, Alain Colmerauer, Maarten van Emden, Pat Hayes, Phillipe Roussel and David Warren. Maarten van Emden and Luis Pereira have helped by reading through the final draft of these notes. Mrs Jean Parker helped by typing these notes under difficult conditions and with great competence.