Prolog

Syntax

Prolog programs are constructed from *terms*: constants, variables, or structures.

Constants can be either atoms or numbers:

- **Atoms** are strings of characters starting with a lowercase letter or enclosed in apostrophes.
- **Numbers** are strings of digits with or without a decimal point and a minus sign.

Variables are strings of characters beginning with an uppercase letter or an underscore.

Structures consist of a **functor** or **function symbol**, which looks like an atom, followed by a list of terms inside parentheses, separated by commas. Structures can be interpreted as predicates (relations):

```
likes(john,mary).
male(john).
sitsBetween(X,mary,helen).
```

Figure A.1 depicts the following structure as trees:

```
person(name('Kilgore','Trout'),date(november,11,1922))
tree(5, tree(3,nil,nil), tree(9,tree(7,nil,nil),nil))
```

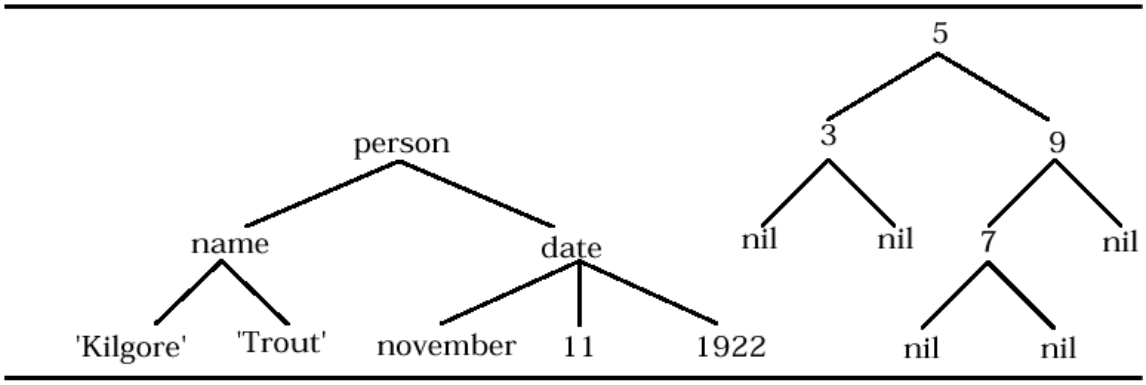

Figure A.1: Structured objects

A Prolog program is a sequence of statements – *clauses* – of the form

P_0 :- P_1, P_2, …, P_n.

where each of P_0, P_1, …, P_n is an atom or a structure.

A period terminates every clause.

A clause can be read declaratively as

P_0 is true if P_1 and P_2 … P_n are true

or procedurally as

To satisfy goal P_0, satisfy goal P_1 and then P_2 and then … and then P_n.

P_0 is the *head* goal; the conjunction of goals P_1, P_2, …, P_n is the *body* of the clause.

A clause without a body

P.

is a *unit clause* or *fact* and means

P is true.

or

goal P is satisfied.

A clause without a head,

?- P_1, P_2, …, P_n.

is a goal clause or *query* and means

Are P_1 and P_2 and … P_n true?

or

Satisfy goal P_1 and then P_2 and then … and then P_n.

A Prolog program consists of

- a database of facts about the given information and
- conditional clauses or *rules* about how additional info. can be deduced from the facts.

A query sets the Prolog interpreter into action.

BNF Syntax for Prolog

<program> ::= <clause list> <query> | <query>
<clause list> ::= <clause> | <clause list> <clause>
<clause> ::= <predicate> **.** | <predicate> **:-** <predicate list> **.**
<predicate list> ::= <predicate> | <predicate list> **,** <predicate>
<predicate> ::= <atom> | <atom> **(** <term list> **)**
<term list> ::= <term> | <term list> **,** <term>
<term> ::= <numeral> | <atom> | <variable> | <structure>
<structure> ::= <atom> **(** <term list> **)**
<query> ::= **?-** <predicate list> **.**
<atom> ::= <small atom> | ' <string> '
<small atom> ::= <lowercase letter> | <small atom> <character>
<variable> ::= <uppercase letter> | <variable> <character>
<lowercase letter> ::= **a | b | c | d |** ... **| x | y | z**
<uppercase letter> ::= **A | B | C | D |** ... **| X | Y | Z | _**
<numeral> ::= <digit> | <numeral> <digit>
<digit> ::= **0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
<character> ::= <lowercase letter> | <uppercase letter>
| <digit> | <special>
<special> ::= **+ | - | * | / | \ | ^ | ~ | : | . | ? | @ | # | $ | &**
<string> ::= <character> | <string> <character>

Figure A.2: BNF for Prolog

Prolog contains a large set of predefined predicates and notational variations (e.g., infix symbols) not defined in this grammar.

And it allows a special syntax for lists – see below.

A Prolog Example

We develop an example incrementally.

User queries are shown in boldface followed by the response by the Prolog interpreter.

Comments start with the symbol % and continue to the end of the line.

Some facts:

```
parent(chester,irvin).
parent(chester,clarence).
parent(chester,mildred).
parent(irvin,ron).
parent(irvin,ken).
parent(clarence,shirley).
parent(clarence,sharon).
parent(clarence,charlie).
parent(mildred,mary).
```

Some queries:

```
?- parent(chester,mildred).
yes
?- parent(X,ron).
X = irvin
yes
?- parent(irvin,X).
X = ron;
X = ken; % The user-typed semicolon asks the
no       % system for more solutions.
?- parent(X,Y).
X =chester
Y = irvin % System will list all of the parent
yes       % pairs, one at a time,if semicolons
          % are entered.
```

Additional facts:

```
male(chester).
female(mildred).
male(irvin).
female(shirley).
male(clarence).
female(sharon).
male(ron).
female(mary).
male(ken).
male(charlie).
```

Additional queries:

```
?- parent(clarence,X), male(X).
X = charlie
yes
?- male(X), parent(X,ken).
X = irvin
yes
?- parent(X,ken), female(X).
no
```

Prolog obeys the “closed world assumption” that presumes that any predicatethat cannot be proved must be false.

```
?- parent(X,Y), parent(Y,sharon).
X = chester
Y = clarence
yes
```

These queries suggest definitions of several family relationships.

Some rules:

```
father(X,Y) :- parent(X,Y), male(X).
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
paternalgrandfather(X,Y) :- father(X,Z),
                            father(Z,Y).
sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

The scope of a variable in Prolog is solely the clause in which it occurs.

Additional queries:

```
?- paternalgrandfather(X,ken).
X = chester
yes
?- paternalgrandfather(chester,X).
X = ron;
X = ken;
X = shirley;  % Note the reversal of the roles
X = sharon;   % of input and output.
X = charlie;
no
?- sibling(ken,X).
X = ron;
X = ken;
no
```

The inference engine concludes that `ken` is a sibling of `ken` since we have both

`parent(irvin,ken)` and
`parent(irvin,ken)`

To avoid this consequence, the description of `sibling` needs to be more carefully constructed.

Predefined Predicates

1. The equality predicate = permits infix as well as prefix notation – e.g.,

```
?- ken = ken.
yes
?- =(ken,ron).
no
?- ken = X. % Can a value be found for X to make
X = ken     %  it the same as ken?
yes % The equal operator represents the notion
    %  of unification.
```

2. \+ ("not") is a unary predicate:

\+ P is true if P cannot be proved false and false if it can – e.g.,

```
?- \+ (ken=ron).
yes
?- \+ (mary=mary).
no
```

The closed world assumption – that any property not recorded in the database isn't true – governs how \+ works.

Since the behavior of \+ diverges from the logical not of predicate calculus, we use \+ as little as possible (not at all in the laboratory exercises).

The following is a new sibling rule (the previous rule must be removed):

```
sibling(X,Y) :- parent(Z,X), parent(Z,Y),
                \+ (X=Y).
```

Queries:

```
?- sibling(ken,X).
X = ron;
no
?- sibling(X,Y).
X = irvin
Y = clarence; % sibling is a symmetric relation.
X = irvin % 3 sets of siblings produce 6 answers.
Y = mildred;
X = clarence % The database allows 14 answers.
Y = irvin;
X = clarence
Y = mildred;
X = mildred
Y = irvin;
Y = mildred
X = clarence % No semicolon here.
yes
```

A relation may be defined with several clauses:

```
closeRelative(X,Y) :- parent(X,Y).
closeRelative(X,Y) :- parent(Y,X).
closeRelative(X,Y) :- sibling(X,Y).
```

There's an implicit **`or`** between the three definitions of the relation `closeRelative`.

This disjunction may be abbreviated using semicolons as

```
closeRelative(X,Y) :-
     parent(X,Y) ; parent(Y,X) ; sibling(X,Y).
```

The three clauses (or single abbreviated clause) are said to define a "procedure" named `closeRelative`.

The *arity* of this procedure is two, i.e., `closeRelative` takes two arguments.

The identifier `closeRelative` may be used as a different predicate with other arities.

Recursion

Suppose we want to define a predicate "X is an ancestor of Y," which is true if

`parent(X,Y)` or

`parent(X,Z)` and `parent(Z,Y)` or

`parent(X,Z)`, `parent(Z,Z1)`, and `parent(Z1,Y)` or

…

A recursive definition is required to allow an arbitrary depth for the definition.

The first case above serves as the basis for the recursive definition.

The remaining cases are handled by an inductive step.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

To continue our example, we add some more facts:

```
parent(ken,nora).          female(nora).
parent(ken,elizabeth).     female(elizabeth).
```

Fig. A.3 shows the parent relation between the twelve people defined in our database.

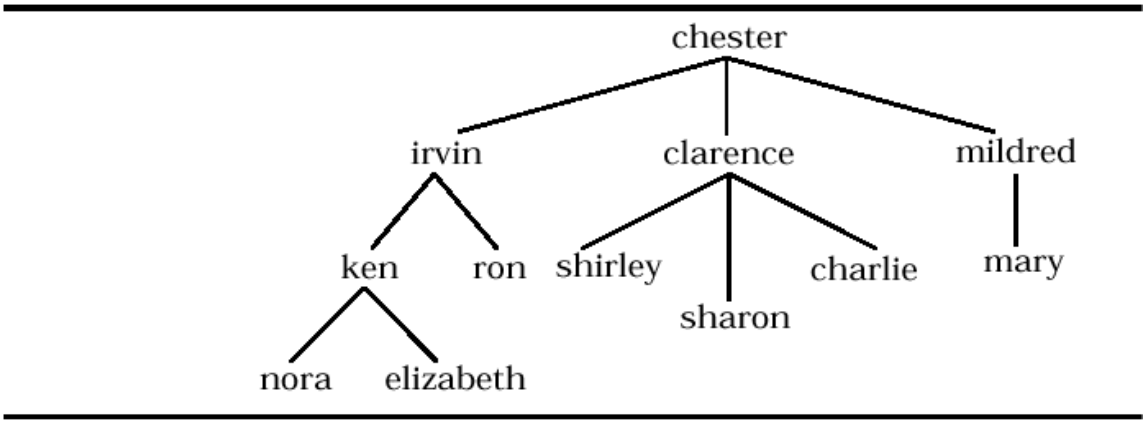

Figure A.3: A Family Tree

Some queries:

```
?- ancestor(mildred,mary).
yes % because parent(mildred,mary).

?- ancestor(irvin,nora).
yes % because
    %  parent(irvin,ken) and
    %  ancestor(ken,nora) because parent(ken,nora).

?- ancestor(chester,elizabeth).
yes % because
    %  parent(chester,irvin)
    %  and ancestor(irvin,elizabeth)
    %   because parent(irvin,ken) and
    %   ancestor(ken,elizabeth)
    %    because parent(ken,elizabeth).

?- ancestor(irvin,clarence).
no % because parent(irvin,clarence) is not provable
   %   and,whoever is substituted for Z, it is
   %   impossible to prove parent(irvin,Z) and
   %   ancestor(Z,clarence).
```

All possibilities for `Z` are tried that make `parent(irvin,Z)` true, namely

`Z=ron` and

`Z=ken`,

and both

`ancestor(ron,clarence)` and
`ancestor(ken,clarence)`

fail.

Control Aspects

Since efficiency is a concern, Prolog interpreters follow a certain deterministic strategy for discovering proofs.

1. The order in which the clauses defining a given predicate are tested (the *rule order* or *clause order*) is top to bottom (as they appear in the text of the program).

Rule order determines the order in which answers are found – e.g.,

```
ancestor2(X,Y) :- parent(X,Z), ancestor2(Z,Y).
ancestor2(X,Y) :- parent(X,Y).

?- ancestor(irvin,Y).
Y = ron, ken, nora, elizabeth
    % Four answers returned separately.
?- ancestor2(irvin,Y).
Y = nora, elizabeth, ron, ken
    % Four answers returned separately.
```

2. The left-to-right order in which terms (subgoals) are listed on the RHS of a rule (the *goal order*) is the order in which the interpreter tries to solve them.

Goal order determines the shape of the search tree that the interpreter explores.

A poor choice of goal order may give a search tree with an infinite branch down which the interpreter gets trapped – e.g.,

```
ancestor3(X,Y) :- ancestor3(Z,Y), parent(X,Z).
ancestor3(X,Y) :- parent(X,Y).
```

```
?- ancestor(irvin,elizabeth).
yes
?- ancestor3(irvin,elizabeth).
```

This query invokes a new query

```
ancestor3(Z,elizabeth), parent(irvin,Z).
```

which invokes

```
ancestor3(Z1,elizabeth), parent(Z,Z1),
parent(irvin,Z).
```

which invokes

```
ancestor3(Z2,elizabeth), parent(Z1,Z2),
parent(Z,Z1), parent(irvin,Z).
```

which invokes …

The eventual result is a message such as

"Out of local stack during execution; execution aborted."

The problem with this last definition of the ancestor relation is the left recursion with uninstantiated variables in the first clause.

If possible, the leftmost goal in the body of a clause should be nonrecursive so that a pattern match occurs and some variables are instantiated before a recursive call is made.

Lists

A list of terms can be represented between brackets – e.g., `[a,b,c]`.

Here the *head* of the list is `a`, the *tail* is `[b,c]`.

The tail of, e.g., `[a]` is `[]` (the empty list).

Lists may contain lists as elements – e.g., `[a,[b,1],3,[c]]` is a list of four elements.

As a special form of direct pattern matching, `[H|T]` matches any list with at least one element:

- `H` matches the head of the list,
- `T` matches the tail.

A list of elements is permitted to the left of the vertical bar – e.g., `[X,a,Y|T]` matches any list with at least three elements whose second element is the atom `a`:

- `X` matches the first element,
- `Y` matches the third element, and
- `T` matches the rest of the list (possibly empty) after the third element.

Using these pattern matching facilities, values can be specified as the intersection of constraints instead of by direct assignment.

Lists are ordinary structures with syntactic sugar added.

The notation abbreviates terms constructed with the predefined "`.`" function symbol and the special atom `[]`.

E.g.,

`[a,b,c]` abbreviates `.(a,.(b,.(c,[]))).`

`[H|T]` abbreviates `.(H,T).`

`[a,b|X]` abbreviates `.(a,.(b,X)).`

List Processing

1. Define `last(L,X)` to mean "X is the last element of the list L".

The last element of a singleton list is its only element.

```
last([X], X).
```

The last element of a list with two or more elements is the last item in its tail.

```
last([H|T], X) :- last(T, X).
```

```
?- last([a,b,c], X).
X = c
yes
?- last([ ], X).
no
```

The "illegal" operation of requesting the last element of an empty list simply fails, allowing the caller to try alternative subgoals.

Predicate `last` acts as a generator when run "backwards":

```
?- last(L, a).
L = [a];
L = [ _5, a];  % The underline indicates system-
L = [ _5, _9, a];  % generated variables.
L = [ _5, _9, _13, a] …
```

Variable `H` in the definition of `last` plays no part in the body of the rule – it doesn't need a name.

Prolog has anonymous variables, denoted by an underscore:

```
last([_|T],X) :- last(T,X).
```

Another example:

```
father(F) :- parent(F,_), male(F).
```

The scope of an anonymous variable is its single occurrence.

(The authors prefer using named variables for documentation.)

2. Define `member(X,L)` to mean "`X` is a member of the list L".

For this predicate we need two clauses,

- one as a basis case and
- the second to define the recursion that corresponds to an inductive specification.

The predicate succeeds if `X` is the first element of L.

```
member(X, [X|T]).
```

If the first clause fails, check if `X` is a member of the tail of `L`.

```
member(X, [H|T]) :- member(X,T).
```

If the item is not in the list, the recursion eventually tries a query of the form `member(X,[])`.

This fails since the head of no clause for `member` has `[]` as second argument.

3. Define `delete(X,List,NewList)` to mean

"The variable `NewList` is to be bound to a copy of `List` with all instances of `X` removed".

When `X` is removed from an empty list, we get the same empty list.

```
delete(X,[ ],[ ]).
```

When an item is removed from a list with that item as its head, we get the list that results from removing the item from the tail of the list (ignoring the head).

```
delete(H,[H|T],R) :- delete(H,T,R).
```

If the previous clause fails, `X` is not the head of the list, so we retain the head of `L` and take the tail that results from removing `X` from the tail of the original list.

```
delete(X,[H|T],[H|R]) :- delete(X,T,R).
```

4. Define `union(L1,L2,U)` to mean

> "The variable `U` is to be bound to the list that contains the unionof the elements of `L1` and `L2`".

If the first list is empty, the result is the second list.

```
union([ ],L2,L2). % clause 1
```

If the head of `L1` is a member of `L2`, it may be ignored since a union does not retain duplicate elements.

```
union([H|T],L2,U) :-
        member(H,L2), union(T,L2,U). % clause 2
```

If the head of `L1` is a not member of `L2` (clause 2 fails), it must be included in the result.

```
union([H|T],L2,[H|U]) :-
        union(T,L2,U). % clause 3
```

In the last two clauses, recursion is used to find the union of the tail of `L1` and the list `L2`.

5. Define `concat(X,Y,Z)` to mean

"The concatenation of lists X and Y is Z".

In the Prolog literature, this predicate is frequently called `append`.

```
concat([ ], L, L).       % clause a
concat([H|T], L, [H|M]) :-
        concat(T, L, M). % clause b
```

```
?- concat([a,b,c], [d,e], R).
R = [a,b,c,d,e]
yes
```

The inference that produced this answer is illustrated by the search tree in Figure A.4.

When the last query succeeds, the answer is constructed by unwinding the bindings:

```
R = [a | M]
  = [a | [b | M1]]  =  [a,b | M1]
  = [a,b | [c | M2]]  =  [a,b,c | M2]
  = [a,b,c | [d,e]]  =  [a,b,c,d,e].
```

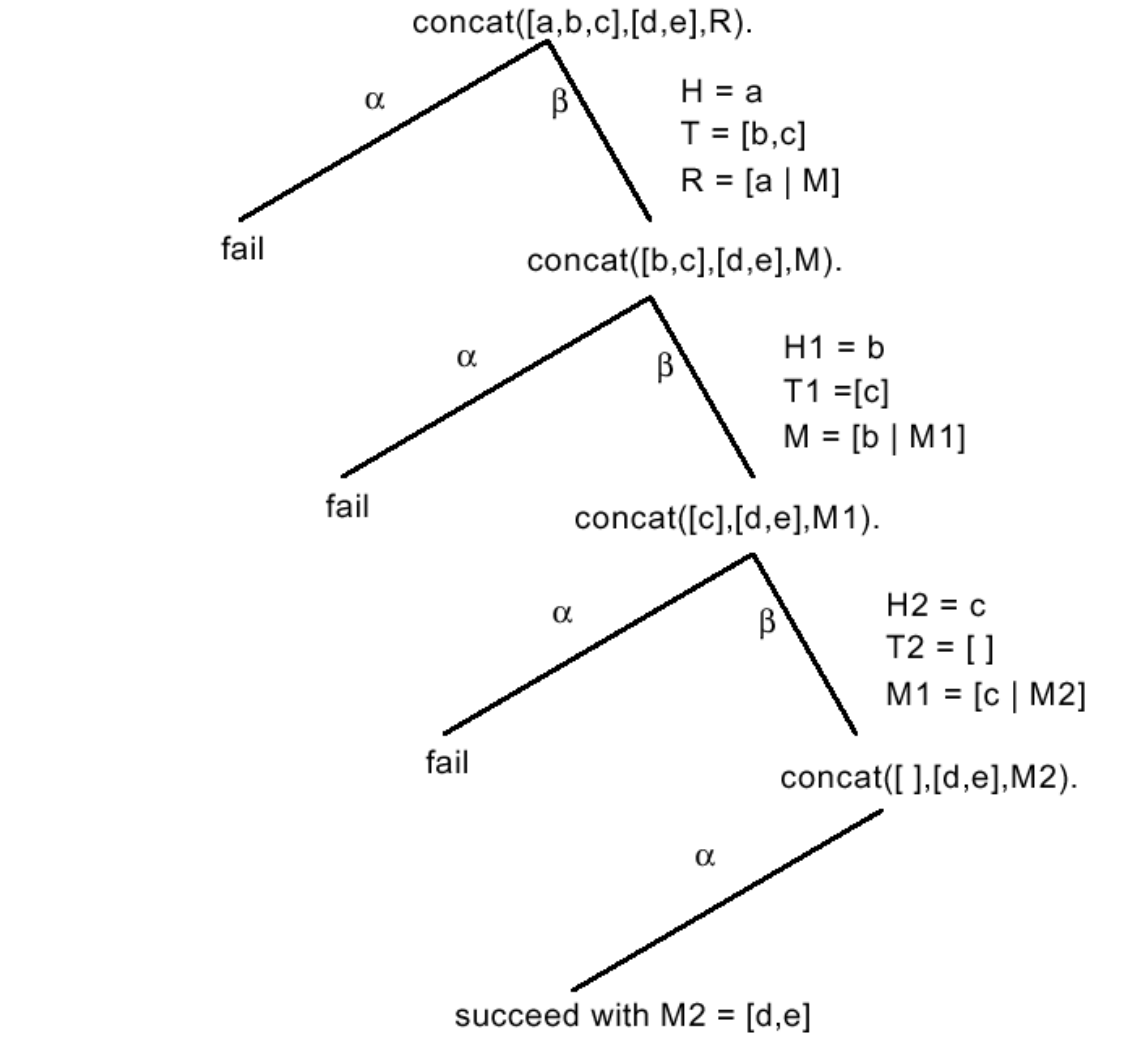

Figure A.4: A Search Tree for concat

Figure A.5 shows the search tree for another application of `concat` using semicolons to generate all the solutions.

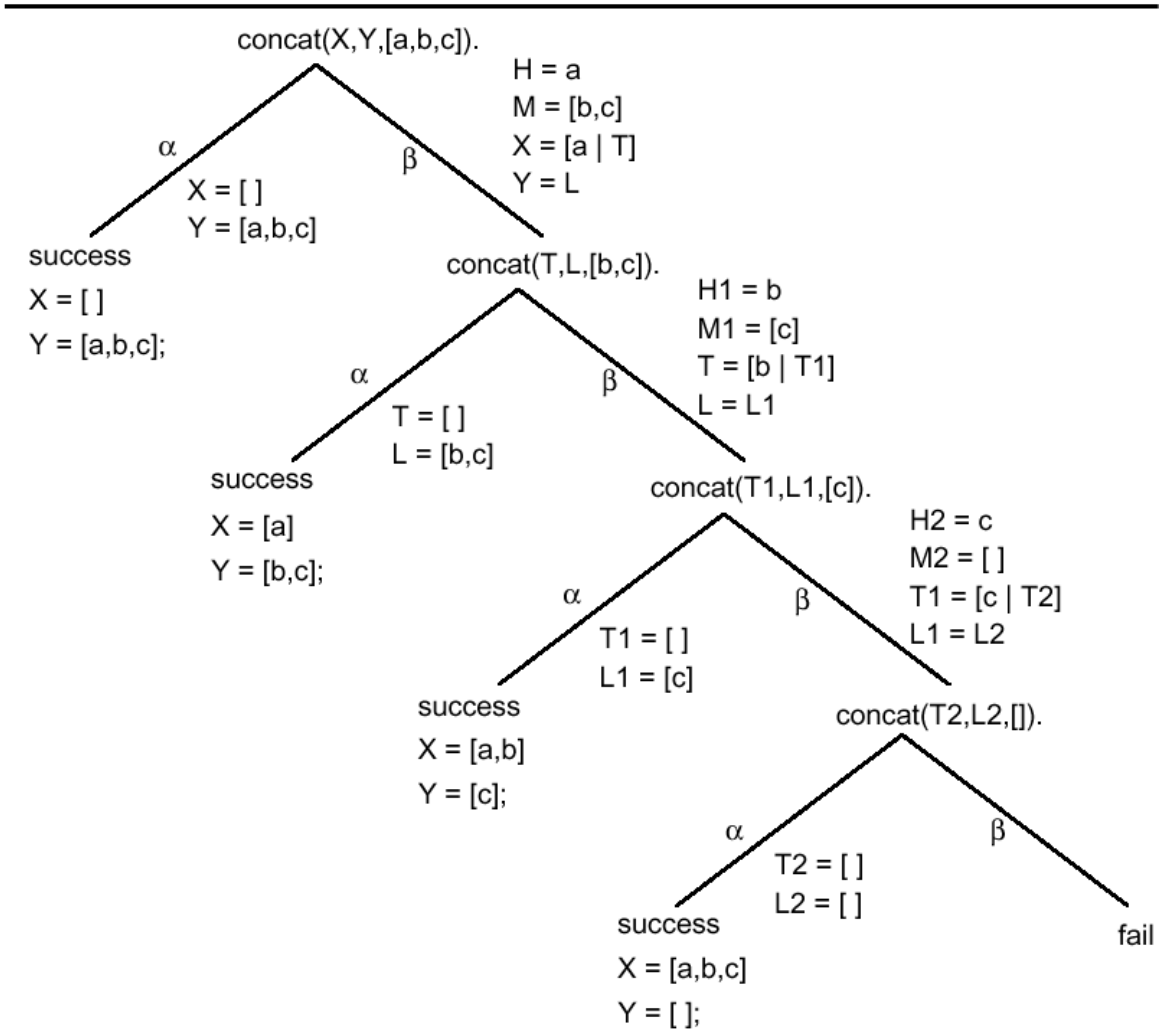

Figure A.5: Another Search Tree for concat

6. Define `reverse(L,R)` to mean "the reverse of list `L` is `R`".

```
reverse([ ], [ ]).
reverse([H|T], L) :- reverse(T, M),
                     concat(M, [H], L).
```

In executing `concat`, the depth of recursion corresponds to the number of times that items from the first list are attached (cons) to the front of the second list.

So the work done by `concat` is proportional to the length of the first list.

When `reverse` is applied to a list of length *n*, the `concat` calls have first arguments of lengths, *n*-1, *n*-2, …, 2, 1.

So the complexity of `reverse` is proportional to n^2.

7. An improved reverse using an accumulator:

```
rev(L, R) :- help(L, [ ], R).
help([ ], R, R).
help([H|T], A, R) :- help(T, [H|A], R).
```

Predicate `help` is called *n* times if the original list is of length *n*.

So the complexity of `rev` is proportional to *n*.

Note that `help` is tail recursive.

Sorting in Prolog

We need relations for comparing numbers (equal, "`=:=`", and not equal, "`=\=`", are discussed later):

```
M < N,  M =< N,  M > N,  M >= N
```

These require that both operands be numeric atoms or arithmetic expressions whose variables are bound to numbers.

Insertion Sort

We sort the tail `T` of a list (recursively) then insert the head `X` into its proper place in the tail.

```
insertSort([ ], [ ]).
insertSort([X|T], M) :- insertSort(T, L),
                        insert(X, L, M).

insert(X, [H|L], [H|M]) :- H<X, insert(X, L, M).
insert(X, L, [X|L]).
```

The clauses for `insert` are order dependent.

We remove this dependence by distinguishing the case where `L` is empty and explicitly stating the conditions for both remaining cases.

```
insert(X, [ ], [X]).
insert(X, [H|L], [X,H|L]) :- X=<H.
insert(X, [H|L], [H|M]) :- X>H, insert(X,L,M).
```

Quick Sort

We split the list into those items less than or equal to the *pivot* and those greater than the pivot.

We arbitrarily chose the first number in the list as the pivot.

After the two lists are sorted (recursively), they are concatenated with the pivot in the middle to form an overall sorted list.

Splitting is done by the predicate

```
partition(P, List, Left, Right)
```

where

`P` and `List` are inputs,

`P` is a pivot for list `List`,

`Left` and `Right` are outputs,

`Left` gets bound to the list of all elements in `List` less than or equal to `P`, and

`Right` gets bound to the list of all elements in `List` greater than `P`.

```
partition(P, [ ], [ ], [ ]).
partition(P, [A|X], [A|Y], Z) :- A=<P,
         partition(P, X, Y, Z).
partition(P, [A|X], Y, [A|Z]) :- A>P,
         partition(P, X, Y, Z).

quickSort([ ], [ ]).
quickSort([H|T], S) :-
         partition(H, T, Left, Right),
         quickSort(Left, NewLeft),
         quickSort(Right, NewRight),
         concat(NewLeft, [H|NewRight], S).
```

The Logical Variable

A variable in an imperative language is not the same concept as a variable in mathematics:

1. A program variable refers to a memory location whose content may change.

2. A variable in mathematics stands for a value that, once determined, won't change.

E.g., the equations $x + 3y = 11$ and $2x - 3y = 4$ specify value for x and y (viz., $x = 5$ and $y = 2$).

A variable in Prolog is called a *logical variable*; it acts like a mathematical variable.

3. Once a logical variable is bound to a value (an *instantiation* of it), the binding can be altered only if the pattern matching that caused the binding is undone by backtracking.

4. The destructive assignment of imperative languages can't be done in logic programming.

5. Terms in a query change only by having variables filled in for the first time.

6. An iterative accumulation of a value is got by having each instance of a recursive rule take the values passed to it and compute values for new variables that are then passed to another call.

7. Since a logical variable is "write-once", it is more like a constant identifier with a dynamic defining expression as in Ada (or Pelican) than a variable in an imperative language.

The power of logic programming comes from using the logical variable in structures to direct the pattern matching.

Results are constructed by binding values to variables according to the constraints imposed by the structures of the arguments of the goal term and the head of the clause being matched.

The order that variables are constrained is generally not critical.

The construction of complex values can be postponed as long as logical variables hold their places in the structure being constructed.

Equality and Comparison in Prolog

Unification

`T1 = T2` succeeds if term `T1` can be unified with term `T2`.

```
| ?- f(X,b) = f(g(a),Y).
X = g(a)
Y = b
yes
```

Numerical Comparisons

`=:=, =\=, <, >, =<, >=`

Evaluate both expressions and compare the results.

```
| ?- 5<8.
yes
| ?- 5 =< 2.
no
| ?- N =:= 5.
! Error in arithmetic expression: not a number
!(N not instantiated to a number)
no
| ?- N = 5, N+1 =< 12.
N = 5 % The unification N = 5 causes a binding
      % of N to 5.
yes
```

Forcing Arithmetic Evaluation (is)

`N is Exp`

Evaluate the arithmetic expression `Exp` and try to unify the resulting number with `N`, a variable or a number.

```
| ?- M is 5+8.
M = 13
yes
| ?- 13 is 5+8.
yes
| ?- M is 9, N is M+1.
M = 9
N = 10
yes
| ?- N is 9, N is N+1.
no % N is N+1 can never succeed.
| ?- 6 is 2*K.
! Error in arithmetic expression: not a number
! (K not instantiated to a number)
no
```

Consider the definition of `factorial`:

The factorial of 0 is 1.

```
fac(0,1).
```

The factorial of `N` > 0 is `N` times the factorial of `N`-1.

```
fac(N,F) :- N>0,
    N1 is N-1,
    fac(N1,R),
    F is N*R.
```

```
| ?- fac(5,F).
F = 120
yes
```

Identity

`X == Y`

Succeed if the terms currently instantiated to `X` and `Y` are literally identical, including variable names.

```
| ?- X=g(X,U), X==g(X,U).
yes
| ?- X=g(a,U), X==g(V,b).
no
| ?- X\==X. % “X \== X” is the negation of “X == X”
no
```

Term Comparison (Lexicographic)

`T1 @< T2, T1 @> T2, T1 @=< T2, T1 @>= T2`

```
| ?- ant @< bat.
yes
| ?- @<(f(ant),f(bat)). % infix predicates may
yes                     % also be entered as prefix
```

Term Construction

`T =.. L`

`L` is a list whose head is the atom corresponding to the principal functor of term `T` and whose tail is the argument list of that functor in `T`.

"`=..`" is pronounced "univ."

```
| ?- T =.. [@<,ant,bat], T.  % Some versions of
T = ant@<bat                 % require call(T)
yes
| ?- T =.. [@<,bat,bat], T.
no
| ?- T =.. [is,N,5], T.
N = 5,
T = (5 is 5)
yes
| ?- member(X,[1,2,3,4]) =.. L.
L = [member,X,[1,2,3,4]]
yes
```

Input and Output Predicates

`get0(N)`

`N` is bound to the ascii code of the next character from the current input stream (normally the terminal keyboard).

When the current input stream reaches its end of file, a special value is bound to `N` and the stream is closed.

The special value depends on the Prolog system, but two possibilities are:

26, the code for control-Z or

-1, a special end of file value.

`put(N)`

The character whose ascii code is the value of `N` is printed on the current output stream (normally the terminal screen).

`see(F)`

The file whose name is the value of `F` becomes the current input stream.

`seeing(F)`

`F` is bound to the name of the current input file.

`seen`

Close the current input stream.

`tell(F)`

The file whose name is the value of `F` becomes the current output stream.

`telling(F)`

`F` is bound to the name of the current input file.

`told`

Close the current output stream.

`read(T)`

The next Prolog term in the current input stream is bound to T.

The term in the input stream must be followed by a period.

`write(T)`

The Prolog term bound to `T` is displayed on the current output stream.

`tab(N)`

`N` spaces are printed on the output stream.

`nl`

Newline prints a linefeed character on the current output stream.

`abort`

Immediately terminate the attempt to satisfy the original query and return control to the top level.

`name(A,L)`

`A` is a literal atom or a number, and `L` is a list of the ascii codes of the characters comprising the name of `A`.

```
| ?- name(A,[116,104,101]).
A = the
| ?- name(1994,L).
L = [49, 57, 57, 52]
```

`call(T)`

Assuming `T` is instantiated to a term that can be interpreted as a goal, `call(T)` succeeds if and only if `T` succeeds as a query.

Some Prolog systems use simply `T` instead of `call(T)`.