Programming Languages

2nd edition

Tucker and Noonan

Chapter 15
Logic Programming

Q: How many legs does a dog have if you call its tail a leg?
A: Four. Calling a tail a leg doesn't make it one.

Abraham Lincoln

Contents

15.1 Logic and Horn Clauses

A Horn clause has a head *h*, which is a predicate, and a body, which is a list of predicates $p_1, p_2, \ldots, p_n$.

It is written as:

$$h \leftarrow p_1, p_2, \ldots, p_n$$

This means, "*h* is true only if $p_1, p_2, \ldots$, and p_n are simultaneously true."

E.g., the Horn clause:

$$snowing(C) \leftarrow precipitation(C), freezing(C)$$

says, "it is snowing in city *C* only if there is precipitation in city *C* and it is freezing in city *C*."

Horn Clauses and Predicates

Any Horn clause

$$h \leftarrow p_1, p_2, \ldots, p_n$$

can be written as a predicate:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow h$$

or equivalently:

$$\neg(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \vee h$$

But not every predicate can be written as a Horn clause.

E.g., $literate(x) \rightarrow reads(x) \vee writes(x)$

Resolution and Unification

If *h* is the head of a Horn clause

$$h \leftarrow terms$$

and it matches one of the terms of another Horn clause:

$$t \leftarrow t_1, h, t_2$$

then that term can be replaced by *h*'s terms to form:

$$t \leftarrow t_1, terms, t_2$$

During resolution, assignment of variables to values is called *instantiation*.

Unification is a pattern-matching process that determines what particular instantiations can be made to variables during a series of resolutions.

Example

The two clauses:

$speaks(Mary, English)$

$talkswith(X, Y) \leftarrow speaks(X, L), speaks(Y, L), X \neq Y$

can resolve to:

$talkswith(Mary, \mathrm{Y}) \leftarrow speaks(Mary, English),$

$speaks(Y, English), \mathrm{Mary} \neq Y$

The assignment of values *Mary* and *English* to the variables *X* and *L* is an instantiation for which this resolution can be made.

15.2 Logic Programming in Prolog

In logic programming the program declares the goals of the computation, not the method for achieving them.

Logic programming has applications in AI and databases.

- *Natural language processing (NLP)*
- *Automated reasoning and theorem proving*
- *Expert systems (e.g., MYCIN)*
- *Database searching, as in SQL (Structured Query Language)*

Prolog emerged in the 1970s. Distinguishing features:

- *Nondeterminism*
- *Backtracking*

15.2.1 Prolog Program Elements

Prolog programs are made from *terms*, which can be:

- *Variables*
- *Constants*
- *Structures*

Variables begin with a capital letter, like Bob.

Constants are either integers, like 24, or atoms, like the, zebra, 'Bob', and '.'.

Structures are predicates with arguments, like:

n(zebra), speaks(Y, English), and np(X, Y)

- The *arity* of a structure is its number of arguments (1, 2, and 2 for these examples).

Facts, Rules, and Programs

A Prolog *fact* is a Horn clause without a right-hand side. Its form is (note the required period .):

term.

A Prolog *rule* is a Horn clause with a right-hand side. Its form is (note :- represents $\leftarrow$ and the required period .):

term :- $term_1$, $term_2$, … $term_n$.

A Prolog *program* is a collection of facts and rules.

Example Program

```
speaks(allen, russian).
speaks(bob, english).
speaks(mary, russian).
speaks(mary, english).
talkswith(X, Y) :- speaks(X, L), speaks(Y, L), X \= Y.
```

This program has four facts and one rule.

The rule *succeeds* for any instantiation of its variables in which all the terms on the right of := are simultaneously true. E.g., this rule succeeds for the instantiation X=allen, Y=mary, and L=russian.

For other instantiations, like X=allen and Y=bob, the rule *fails*.

Searching for Success: Queries

A *query* is a fact or rule that initiates a search for success in a Prolog program. It specifies a search goal by naming variables that are of interest. E.g.,

```
?- speaks(Who, russian).
```

asks for an instantiation of the variable Who for which the query speaks(Who, russian) succeeds.

A program is loaded by the query consult, whose argument names the program. E.g.,

```
?- consult(speaks).
```

loads the program named speaks, given on the previous slide.

Answering the Query: Unification

To answer the query:

```
?- speaks(Who, russian).
```

Prolog considers every fact and rule whose head is speaks. (If more than one, consider them in order.)

Resolution and unification locate all the successes:

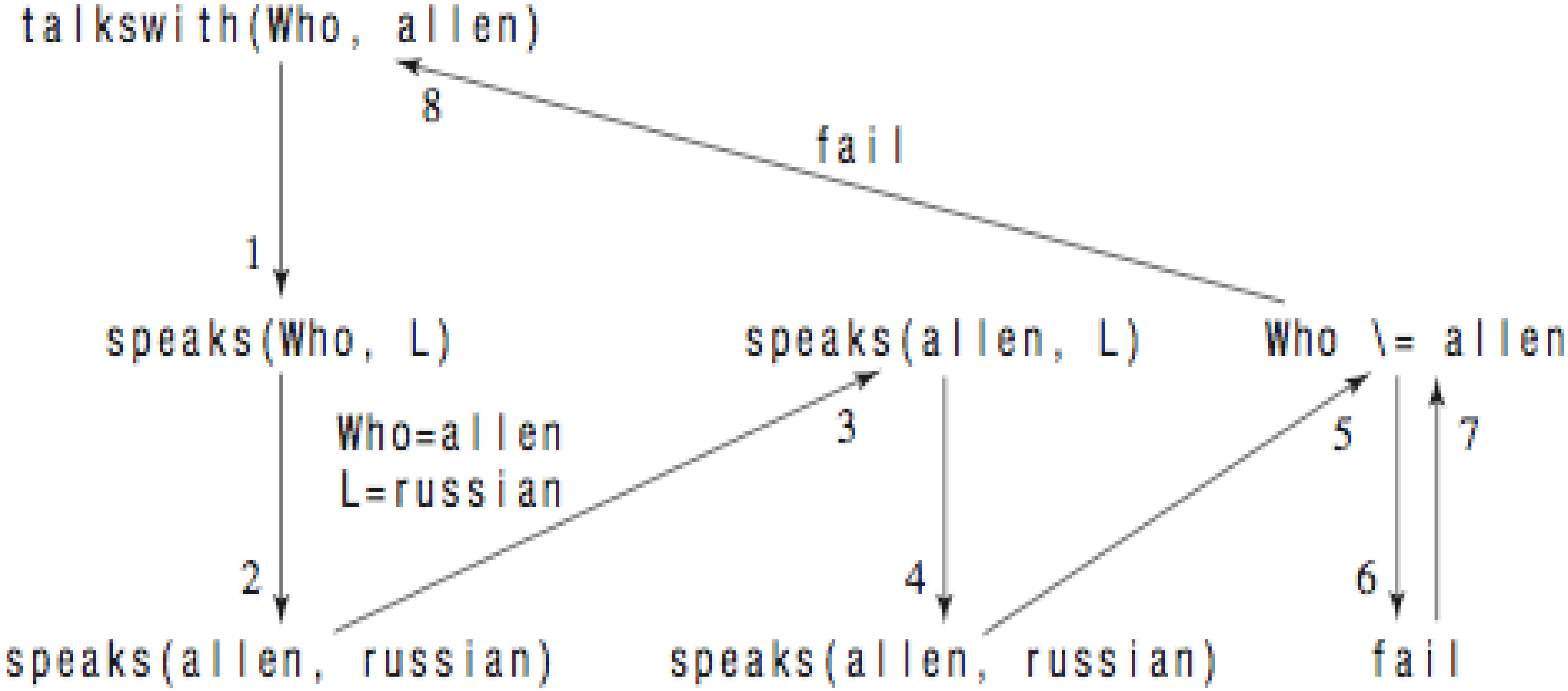
```
Who = allen ;
Who = mary ;
No
```

- *Each semicolon (;) asks, "Show me the next success."*

Search Trees

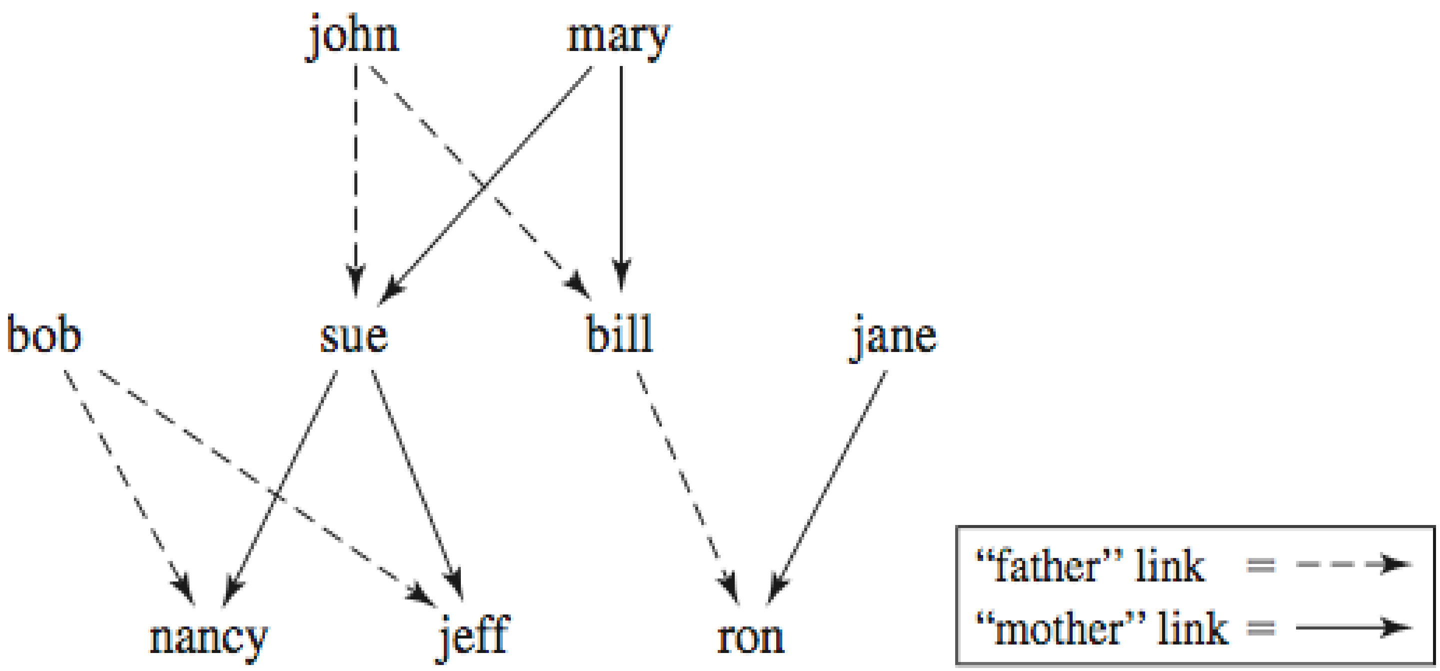
First attempt to satisfy the query ?- talkswith(Who, allen).

Fig 15.2



Database Search - The Family Tree

Fig 15.4



Prolog Program

Fig 15.3

```
mother(mary, sue).        father(john, sue).
mother(mary, bill).       father(john, bill).
mother(sue, nancy).       father(bob, nancy).
mother(sue, jeff).        father(bob, jeff).
mother(jane, ron).        father(bill, ron).

parent(A,B) :- father(A,B).
parent(A,B) :- mother(A,B).
grandparent(C,D) :- parent(C,E), parent(E,D).
```

Some Database Queries

Who are the parents of jeff?

```
?- parent(Who, jeff).
Who = bob;
Who = sue
```

Find all the grandparents of Ron.

```
?- grandparent(Who, ron).
```

What about siblings? Those are the pairs who have the same parents.

```
?- sibling(X, Y) :- parent(W, X), parent(W, Y), X\=Y.
```

Lists

A *list* is a series of terms separated by commas and enclosed in brackets.

- *The empty list is written* [].
- *The sentence "The giraffe dreams" can be written as a list:* [the, giraffe, dreams]
- *A "don't care" entry is signified by _, as in*

  [_, X, Y]
- *A list can also be written in the form:*

  [Head | Tail]
- *The functions*

  append *joins two lists, and*

  member *tests for list membership.*

append Function

```
append(X, [], X).
append([Head | Tail], Y, [Head | Z]) :-  append(Tail, Y, Z).
```

This definition says:

1. Appending the empty list to any list (X) returns an unchanged list (X again).
2. If Y is appended to Tail to get Z, then a list one element larger [Head | Tail] can be appended with Y to get [Head | Z].

Note: The last parameter designates the result of the function. So a variable must be passed as an argument.

member Function

```
member(X, [X | _]).
member(X, [_ | Y]) :- member(X, Y).
```

The test for membership succeeds if either:

1. X is the head of the list [X | _]
2. X is not the head of the list [_ | Y] , but X is a member of the list Y.

Notes: *pattern matching* governs tests for equality.

Don't care entries (_) mark parts of a list that aren't important to the rule.

More List Functions

X is a *prefix* of Z if there is a list Y that can be appended to X to make Z; i.e.,

```
prefix(X, Z) :- append(X, Y, Z).
```

Similarly, Y is a *suffix* of Z if there is a list X to which Y can be appended to make Z; i.e.,:

```
suffix(Y, Z) :- append(X, Y, Z).
```

So finding all the prefixes (suffixes) of a list is easy. e.g.:

```
?- prefix(X, [my, dog, has, fleas]).
X = [];
X = [my];
X = [my, dog];
…
```