CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by (with modifications): Ira Woodring

Logic Languages

(Follows the Sebesta Text Chapter 16)

Logic languages allow us to create programs via symbolic logic. Logical inferences are used to find the answers we seek.

Logic langugages, like functional languages, are **declarative** because they specify *what* a program should do rather than specifying transformations to machine state (imperative programming)

Programs written in logic languages will not look like "programs" at all - at least not what we're used to seeing. They are more like databases of facts (sometimes called a Knowledge Base).

Logic programs are made up of declarations instead of procedures. We then consult the declarations to find solutions to our problems.

Logic languages are based upon something called **Predicate Calculus**.

A predicate calculus (a.k.a. First Order Logic) uses **propositions**. These are logical statements that may or may not be true. Formal logic was made to allow us to check the validity of propositions.

Example (not in Prolog)

```
I like dogs. (proposition)
Dogs are animals. (proposition)
I like animals. (proposition -- true if others are true)
```

Some example propositions in Prolog:

```
man(jake)
like(bob, steak)
```

These should be read as "Jake is a man." and "Bob likes steak."

It should be noted that in the above example, `man`, `jake`, `like`, `bob`, `steak` are all constants.

Also note that there is no real meaning here implied by the computer; we assign meaning to `man(jake)` and `like(bob, steak)`. `like(bob, steak)` could instead mean, for example, that `bob` is similar to `steak`.

General process of logical program is to add facts to knowledge base and then query knowledge base for interesting information

We will be using the language Prolog to study logical languages. Prolog programs are collections of statements that are made from **terms**.

Prolog statements built up from **terms**

Terms used to construct **facts** and **rules**

Once facts and rules established, **queries** (a.k.a. **goals**) constructed to get useful information

Terms

A term in Prolog is a:

- constant,
- variable, or
- structure

Terms

Constant is either atom or an integer.

Atom is symbolic value composed of letters, digits, underscores

Cannot begin with uppercase letter or underscore

For example:

```
cat
my_Atom
atom57
```

Terms

Variable in Prolog is string of letters, digits, underscores, that starts with uppercase letter or underscore

For instance:

```
parent(X, Y).
```

X and Y are variables, but `parent` is not

Terms

Structure represents "atomic proposition" from predicate calculus, which is relationship between objects

Consists of two parts:

- **Functor** names relationship
- parameters specify particular values

```
functor(parameter list)
```

Terms

Example sturctures:

```
parent(bob, sue).
dog(fido).
speed(light, "really fast").
speed(sound, "mostly fast").
```

Once again, we give meaning to these based on how we read them

Prolog does not care about meaning -- it tracks relationships

To construct knowledge base, Prolog allows us to create facts and rules

Facts

Fact is some proposition assumed to be true

Single structure on its own (with `.` at the end) interpreted as fact

For instance:

```
female(shelley).
female(mary).
male(bill).
male(jake).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

Rules

Rules -- implications between propositions

```
ancestor(mary, shelley) :- mother(mary, shelley).
```

This should be read as "If mary is the mother of shelley, then mary is an ancestor of shelley."

Rules

Notice general form:

```
consequence :- antecedent_expression.
```

Follows familiar "if-then" pattern, but written in reverse

If `antecedent_expression` is true, `consequence` is true

Rules

Variables make rules apply in general case, which can be more useful

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

Specifies general relationships between ideas of `parent`, `mother`, `father`, and `grandparent`

Queries

Queries/goals get information *out* of knowledge base rather than in

In simplest case, could query whether some fact exists:

```
man(fred).
```

Returns `true` if fact exists in database

Queries

Power of Prolog comes from deducing facts not explicitly given

With knowledge base

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
parent(anakin, leia).
parent(leia, kylo).
```

system deduces

```
?- grandparent(anakin, kylo).
true.
```

Queries

If fact does not exist in database, *or goal cannot be proven true*, `false` is returned

Important: `false` does not prove proposition false, merely that proposition cannot be proven from database

Queries

Given rules about parent/grandparent relationship and

```
parent(leia, kylo).
grandparent(anakin, kylo).
```

system deduces

```
?- parent(anakin, leia).
false.
```

Happens to be true, but not deducible from these facts (Anakin could be paternal grandparent)

Queries

Goals become more interesting when we allow variables

Given

```
father(anakin, luke).
```

we can query either of the following

```
?- father(X, luke).
X = anakin.
?- father(anakin, X).
X = luke.
```

Queries

Notice that variables in logic languages are not like imperative variables

We do not modify them or manage state

They simply allow flexibility in facts, rules and queries -- only knowledge base itself instantiates variables, and only when checking query

Queries

If more than one possibilty results from query, Prolog returns all of them (until you tell it to stop)

```
father(anakin, luke).
father(anakin, leia).
```

```
?- father(anakin, X).
X = luke ;
X = leia.
```

Arithmetic

We can perform arithmetic with Prolog as well. Prolog will evaluate an `is` expression. We need to be careful; we can type

```
X = 3 + 2
```

But this will set X to `3 + 2`; it will not be evaluated.

Arithmetic

However, if we use the `is` functor the expression will be evaluated before assignment:

```
X is 3 + 2.
```

Now the system will respond with `X = 5.`.

Arithmetic

We can use these mechanisms in our rules. For instance, if we have a knowledge base that keeps track of the average speeds on a racetrack for each car and the amount of time each car was on the track, we could determine the distance the any car covered.

Arithmetic

```
speed(ford, 100).
speed(chevy, 105).
speed(dodge, 95).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
dist(X, Y) :- speed(X, Speed), time(X, Time), Y is Speed * Time.
```

Arithmetic

We can now query this system:

```
?- dist(ford, Ford_Distance).
Ford_Distance = 2000.
```

Should we wish, we could ask for the trace of resolution:

```
trace.
true.

[trace]  ?- dist(ford, Ford_Distance).
   Call: (8) dist(ford, _3660) ? creep
   Call: (9) speed(ford, _3882) ? creep
   Exit: (9) speed(ford, 100) ? creep
   Call: (9) time(ford, _3882) ? creep
   Exit: (9) time(ford, 20) ? creep
   Call: (9) _3660 is 100*20 ? creep
   Exit: (9) 2000 is 100*20 ? creep
   Exit: (8) dist(ford, 2000) ? creep
Ford_Distance = 2000.
```

Lists

Prolog also allows us to have lists. We can create one as follows:

```
[ apple, grape, orange, watermelon, kiwi ]
[ [ pop, diet_pop ], [orange, grape, apple], water ]
```

Notice a list can be made of atoms and lists.

Lists

Lists in Prolog are much like lists in functional languages. To the langauge, a list can be broken down into two parts - the head and the tail.

Just like the functional counterparts the head is the first element in the list and the tail is everything else:

Lists

```
[H|T] = [ apple, orange, grape, kiwi, lemon ].
H = apple,
T = [orange, grape, kiwi, lemon].
```

Notice the "|"?

Lists

The "|" character allows us to decompose a list. For instance, in the above code we decomposed the list into a head (H) and tail (T). But, we can do other decompositions as well:

```
?- [X,Y | W] = [ apple, banana, grape, kiwi ].
X = apple,
Y = banana,
W = [grape, kiwi].
```

Lists

In fact, we don't even need to use a name for a variable if we aren't going to use it. Just like we saw in Python, we can use the "_" character for anything we don't care about:

```
?- [_, Y | _] = [ apple, banana, grape, kiwi ].
Y = banana.
```

Lists

All this is great of course - and useful. But at some point we need a way to perform list iteration.

Just like functional languages though, there is no such thing as an iterator. Instead, we make use of our trusty friend, recursion!

Lists

Here is the classic example of the "member" function:

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

"member" determines if an element is a member of a list.

Lists

What is going on with `member`?

As usual with recursion, we have general case and base case

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

Base case: if X is first element, X is member of list

Lists

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

Otherwise, X is member of list if X is member of sublist starting from second element

Lists

Fortunately for us though, there are a lot of built-in functions we can make use of that perform simple tasks. Such as:

```
?- include(>(3), [1,2,3,4,5,6,7], X).
X = [1, 2].
?- exclude(<(30), [1, 2, 3, 4, 60, 70, 80], X).
X = [1, 2, 3, 4].
```

Lists

You probably noticed that these seem to do the opposite of what the "look" like they should do. They are actually filtering based on the condition, not accepting based on the conditional.