Prolog, pt 1

CS 440: Programming Languages and Translators, Spring 2020

4/7 p.6 (Activity 21)

1. Sources for Study[*]

- Read Chapters 1 and 2 of *Learn Prolog Now*: www.learnprolognow.org/
- *SWI Prolog*: http://www.swi-prolog.org/

2. Prolog and Logic

- Prolog is a programming language based on logic. E.g., Taylor is a fish; All fishes are animals; therefore Tyler is animal.
- At its core, Prolog is a **declarative language**.
  - In Prolog you describe a set of logical facts (properties of data) and relationships between data.
    - E.g., `fish(Taylor)` is a fact and `animal(X) :- fish(X)` is a rule (all fishes are animals).
  - Then you make queries and Prolog tries to prove that your query is true, possibly defining values for variables to do this.
    - E.g., `animal(X)` is true if `X = Taylor`. On the other hand, goldfish(Taylor) would be false. Note this is "false" in the sense of "not provably true" (with just the information so far).
- Compare this with an **imperative language**, which includes Haskell, Java, C, Javascript, and probably every language you've seen.
  - In these languages, you describe data structures, operations on data, and as you write a program, you keep track mentally (or in comments) the properties of your data and how they change as you manipulate it.
  - Although the underlying philosophy behind it is declarative, Prolog does have some imperative features, which can come in handy to avoid problems like infinite recursion as Prolog tries to prove

3. First-Order Logic

- Prolog uses first-order predicate calculus, which includes primitive predicates $P(a_1, a_2, \ldots, a_n)$, conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), implication ($\rightarrow$), and universal ($\forall$) and existential ($\exists$) quantifiers. In addition, you have operations and relations that go along with your domains of values. (E.g., integers come operations like + and – and with primitive relations like <, ≤, and so on.
- **First-order** means that the quantifiers range over values, not predicates, so you might ask if there exists a fish named Wanda, but you can't ask if there's a property that Wanda and Tyler have in common. (For any particular relation, you can ask if Wanda and Tyler have that property.
  - E.g., asking `freshwater(Wanda), freshwater(Tyler)` is okay. (Comma means "and" here.)

[*] I also used *Programming Language Pragmatics*, 4th ed., Michael L. Scott as a reference.

- But you can't ask `P(Wanda), P(Tyler)` where `P` is a variable that stands for a property.

Basics parts of Prolog

4. Basic Datatypes

- **Terms** include
  - **Atoms** are are named constants; e.g., `eggplant`, `purple`, `vegetable`.
    - Atoms begin with a lower-case letter.
  - **Variables** are names that can have values; e.g., `X` might stand for `eggplant` (or it might not).
    - Variables begin with an upper-case letter or an underscore.
  - **Compound terms** combine a *property name* and some arguments: e.g., `likes(alex, eggplant)`.
    - Here, `likes` is the **functor** (this is a different meaning than Haskell's Functor classtype).
    - `likes` has an **arity** of 2 because it takes two arguments. Prolog messages often include the arity, so it might say something about `likes/2`.
  - Built-in values include:
    - Constants: **Numeric constants** are integer and floating-point constants; the **Boolean** constants are `true` and `false`.
    - **Lists**, written with square brackets and commas, such as `[12, alex, student(george)]`.
    - There's also a cons operation. e.g., `[1, 2]` = `[1|[2]]` (In Haskell, `[1,2]` = `[1:[2]]`.)
    - **Strings** `"like this"` are treated as atoms or as lists of characters.
- The logical constructs:
  - **Predicates** are compound terms or arithmetic or equality relation
    - E.g., `needs(eggplant, salt)`, `X < 3`, or `[A, 3, rhubarb]` = `[ice, 3, B]`
  - **Facts** are written as a compound term involving constants.
    - E.g., `color(eggplant, purple)` or `parent(charlie, joey)` .
  - **Rules** include variables and are written using reversed implication: A *conclusion* is *implied by* a *body*.
    - The *conclusion* (a.k.a. *head*) is a single predicate. E.g., `ancestor(charlie, max)`.
    - The *body* is a sequence of predicates separated by commas. (Commas mean “and”.)
      - E.g., `parent(X, Y), ancestor(Y, Z)`. (`X` is a parent of `Y` and `Y` is an ancestor of `Z`.)
    - The "*is implied by*" symbol is `:-` . (In everyday logic, people use ← (the reverse of → implication).)
      - `ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z).`
      - (`X` is an ancestor of `Z` if `X` is a parent of `Y` and `Y` is an ancestor of `Z`.)
    - **Quantification**: You don't include the ∀ or ∃ symbols, but variables that appear in the head (left of `:-`) are implicitly universally quantified; variables that appear in the body but not the head are implicitly existentially quantified.

- E.g., `ancestor(X, Z) :- parent(X, Y), ancestor(Y, Z)` is read as "For all X and Z, `ancestor(X, Z)` holds if there exists a `Y` such that `parent(X, Y)` and `ancestor(Y, Z)`".

- **Horn Clause**
  - Written in standard predicate logic form, a Prolog rule looks like $p_1 \wedge p_2 \wedge ... \wedge p_n \rightarrow q$ or equivalently, $\neg p_1 \vee \neg p_2 \vee ... \vee \neg p_n \vee q$. A predicate of this form (a conjunction of logical terms where only one is positive (i.e., not negative = not negated) is a **Horn clause.**
  - In addition to Prolog rules being Horn clauses, facts are also Horn clauses ("`:- true`" is implicit).
    - E.g., `parent(charlie, joey).` is equivalent to `parent(charlie, joey):- true`.

5. Prolog Execution

- A Prolog program consists of a **database** of facts and rules, plus queries.
  - In the database, each fact and rule is followed by a period.
  - A **query** or goal is a logical term followed by a period.
  - (In the discussions, we can leave out the periods it the meaning is clear.)
- In Prolog, we don't run programs as such, we pose queries.
  - Prolog tries to prove a query using backtracking search plus unification of variables.
  - E.g., if `contains(pizza, X)` is a query and `contains(pizza, tomatoes)` is a fact, then Prolog proves the query under substitution $\texttt{X} \equiv \texttt{tomatoes}$. (In the unification lecture, we used the notation $[\texttt{X} \mapsto \texttt{tomatoes}]$ and used *term1* ≡ *term2* to ask if the two terms were unifiable.)
- If Prolog can't find a proof for a query, it returns `false` (in the sense of "can't be proved").
- If it does find a proof, it says so and gives the values of variables it used in the proof.
- You can ask for another proof to see if a different collection of variable values can make the query true.
  - (Once Prolog determines that there can't be any more proofs let, it stops letting you ask.)

6. Some simple examples — Who likes what?

- To run Prolog programs, we'll use the online system SWI-Prolog (at https://www.swi-prolog.org/).
  - Press the *Try SWI-Prolog online* link. Press (Create) Program in the upper left of the new page.
  - We'll enter the database into the left half of the resulting page. Queries go in the bottom right pane and results are shown in the top right pane.

- For the database, enter

```
likes(sam, pizza).
likes(sam, chicken).
likes(charlie, ice_cream).
likes(mr_whiskers, chicken).
likes(mr_whiskers, fish).
```



```
likes(finley, pizza).
likes(finley, ice_cream).
likes(finley, chicken).
likes(X, ice_cream) :- someone(X). % everyone likes ice cream
someone(sam).
someone(charlie).
someone(finley).
someone(mr_whiskers).
```

What does Sam like?

- For a query, enter `likes(sam, X).` and activate the query with ^↵. (control-Enter)
  The first answer `pizza` appears Repeatedly pressing the *Next* button gives you two more answers.

```
X = pizza
X = chicken
X = ice_cream
```

Who likes chicken?

- Similarly, `likes(P, chicken).` should produce

```
P = sam
P = mr_whiskers
P = finley
```

(Everyone) likes ice cream

- We get a number of answers for `likes(F, ice_cream).` (You can press *Next* repeatedly or use a number button to press *Next* that many times.)
  - Only `charlie` and `finley` liking `ice_cream` are facts.
  - But the rule `likes(X, ice_cream) :- someone(X)`. lets Prolog deduce that all five of `sam`, `charlie`, `finley`, and `mr_whiskers` also like `ice_cream` because they are `someone`.
  - Note that `charlie` and `finley` are listed twice; this is because there are two different ways to prove that they like ice cream: From a fact and using a rule.

```
F = charlie
F = finley
F = sam
F = charlie
F = finley
F = mr_whiskers
```

Do Sam and Mr. Whiskers both like something?

- We can ask if `sam` and `mr_whiskers` have any liked food in common by entering a second term.
  - The query is `likes(sam, X), likes(mr_whiskers, X).` (Again, note the period.)
  - To find a solution, Prolog looks for a proof of `likes(sam,` X) [it finds the fact `likes(sam, pizza)`] and then tries to prove `likes(mr_whiskers, X)` for the same `X`.

- For `X` ≡ `pizza`, this fails, so Prolog looks for a different proof of `likes(sam,` X), comes up with `X` ≡ `chicken` as another fact, and then tries to prove `likes(mr_whiskers, X)` for `X` ≡ `chicken`. This time Prolog succeeds, so it returns `X` ≡ `chicken` as an answer.
- If we ask for another proof, we'll eventually deduce that `sam` likes `ice_cream` (seen earlier with the everyone-loves-ice-cream rule) and eventually deduce that `mr_whiskers` also likes `ice_cream`, so X ≡ ice_cream is the second answer.
- If we ask for another proof, Prolog will try to prove that `sam` likes a third `X` but it fails, which ends the query processing.

Do Sam and Mr. Whiskers like different foods?

- What about two different foods, where `sam` likes one and `mr_whiskers` likes the other?
- The "not equal" test is `\=`, so a first attempt is `likes(sam, X), likes(mr_whiskers, Y), X \= Y`.
  - The database says that `sam` likes `pizza` and `mr_whiskers` likes `fish`, but not vice versa.
  - This solution, however, comes up with `X` = `pizza`, `Y` = `chicken`, but though `mr_whiskers` doesn't like `pizza`, both `sam` and `mr_whiskers` like `chicken`.
  - If we think about the quantification, we realize that we asked if there exist `X` and `Y` that are different where `sam` likes `X` and `mr_whiskers` likes `Y`. We didn't say anything about `sam` not liking `Y` or `mr_whiskers` not liking `X`.
- To ask if there's a food that one likes but not the other, we need a **logical not** operation.
  - SWI uses `not(...)` to negate the term on the inside.
  - For `not(likes(mr_whiskers, X))`, Prolog tries to prove `likes(mr_whiskers, X)` and if it fails, then the logical not of the likes succeeds.
    - (So "not" here means "isn't provable", which is different from "demonstrably false".)
  - As a quick check, we can try `likes(sam, X)`, `not(likes(mr_whiskers, X))`, which comes back with `X` ≡ `pizza` and no other solution.
  - Similarly, `likes(mr_whiskers, X)`, `not(likes(sam, X))`, which comes up with `X` ≡ `fish` as the only solution.
- So to ask, "Are there two foods where `sam` likes one and `mr_whiskers` likes the other but not vice versa?" we can use `likes(sam, X), not(likes(mr_whiskers, X)), likes(mr_whiskers, Y), not(likes(sam, Y))`. This coms up with `X` ≡ `pizza` and `Y` ≡ `fish` as the only solution, which is what we wanted.

If you run out of space

- If SWI Prolog complains about running out of space for calculating results, you can erase a result by selecting the circled `x` at the top right of the result.
- To get rid of all results, find the triple bar icon at the top right of the top right window (the one with the owl). Select *Clear* to clear everything.

Activity Problems for Lecture 21

Turns out the Exercises in Chapters 1 and 2 of Learn Prolog Now make for pretty reasonable activity questions

For Exercises 1.1 - 1.3, `big kahuna burger` is not an atom (embedded blanks), . `'loves(Vincent,mia)'` and (`Butch kills Vincent`) are not terms. For Exercise 1.4, you don't need separate `is_killer(X)` and `kills(X, Y)` terms. You will need relations/properties like `married`, `dead`, `footmassage`, `loves`, `good_dancer`, `nutritious`, and `tasty`. For Exercise 1.5, type it into SWI Prolog to verify your answers.

For Exercise 2.1, the apostrophes around `Bread` and `bread` don't change their meaning; `food(bread)` can only unify with itself or with `food`(Variable); `food` with three arguments can't match `food` with two arguments; you can't have `X ≡ drink(beer)` and `X ≡ food(bread)` simultaneously. Exercise 2.4 (the crossword placement problem) is actually pretty hard, so skip it.

