The current topic: Prolog

✓ Introduction
✓ Object-oriented programming: Python
✓ Functional programming: Scheme
✓ Python GUI programming (Tkinter)
✓ Types and values
• Logic programming: Prolog
  – Next up: Introduction
• Syntax and semantics
• Exceptions

Announcements

- Lab 2 has been marked.
  - A marking report has been sent to your **ECF** email address.
  - The deadline for re-mark requests is **next Friday (November 14th)**.
  - If you lost marks for **exercise 3**:
    For exercise 3 (`replace`), the handout stated that `replace` takes two symbols and a nested list, and implied that the nested list contained symbols but not numbers. However, the automarking test cases passed numbers as the first two parameters, and included numbers in the nested list. If you lost marks because of this (that is, your code works according to the specifications given in the handout, but failed the test cases because they included numbers), send me an email (a re-mark request form is not required for this).
- Reminder: The project is due on **November 17th**.

Logic programming

- A program consists of facts and rules -- a *knowledge base*.
- Running a program means asking queries.
- The language tries to find one or more ways to prove that the query is true.
  - You don't have to figure out how to do it.
- So you say *what* you want, rather than *how* to find it.
  - For example, consider writing code for sorting a list. In logic programming, you describe the *properties* of a sorted list, rather than giving the *steps* that need to be followed to sort a list.
    - "`newL` is a sorted version of `L` if `newL` has the same elements as `L` **and** for every pair `x`, `y` of adjacent elements in `newL`, we have `x ≤ y`."
  - Another example: SQL queries. (But note that SQL is not considered a logic programming language.)

Logic programming

In a Prolog program you tell the computer:

- Here are some argument values.
  - For example, here is a list `myList`.
- Here's a statement involving the values given and some other unknown values.
  - For example, `sorted(myList, L)`, asserting that `L` is a sorted version of `myList`.
- Tell me what the unknowns have to be to make the statement true.
  - For example, tell me a value (or values) of `L` that makes `sorted(myList, L)` true.

Logic programming and Prolog

- Prolog is the only major logic programming language.
  - Developed in the early 1970s by Alain Colmerauer, Phillippe Roussel, and Robert Kowalski.
  - Prolog = "***Pro***grammation en ***log***ique" ("Programming in logic")
- Prolog is often said to be unsuited to expressing algorithms, but this misrepresents the language.
  - It's about logic *and* algorithms.
  - Algorithms are expressed in a recursive style that is different from Scheme.
  - But we'll concentrate on the logic side of Prolog, with only a glance at the algorithmic side.
- Reference: Sebesta, chapter 16.

The Prolog we'll use

- Prolog flavours vary less than Scheme flavours.
- We'll use SWI Prolog.
  - http://www.swi-prolog.org/
  - On ECF, use the command: **`pl`**
  - Version 5.2.10 is installed on ECF.
  - This version does **not** appear to be available on the SWI Prolog website. If you download the current version (5.6.x), make sure you test your code using the version on ECF before submitting. Also, note the current version has a slightly different user interface that the version on ECF.
- There are various Prolog IDEs available.
  - But for this course it will be enough to use a text editor along with `pl`.

The SWI pl interface

- All commands end with a period.
- To quit: `halt.`
  - With a period!
  - Alternative: Use ctrl-d.
- The prompt: `?-`
  - If you forget the period:

```
?- parent(albert,X)
|    .
```

- To retry a query: `;`
- You can't have a blank between the functor (the relation name, e.g. `parent`) and the left parenthesis.

Prolog statements

- There are three kinds of statements.
  - Facts.
  - Rules.
  - Queries.
- Facts simply state information we can assume. For example:

```
university(uoft).
campus(uoft, stgeorge).
campus(uoft, utm).
course(csc326).
offered(stgeorge, csc326).
```

  - We (users, **not** Prolog itself) might interpret these statements to mean that uoft is a university, stgeorge and utm are campuses of uoft, csc326 is a course, and the course csc326 is offered on the stgeorge campus. There isn't any single "correct" interpretation, and the interpretation is irrelevant to Prolog.
- Rules allow Prolog to derive new facts from existing facts.

Prolog statements

- Queries (or goals) state questions that Prolog answers using *only* the facts and rules it has been given. For example, using the facts from the previous slide:

```
?- university(uoft).
Yes
?- university(york).
No
?- university(X).
X = uoft ;
No
?- campus(uoft, C).
C = stgeorge ;
C = utm ;
No
?- course(C), offered(stgeorge, C).
C = csc326 ;
No
```


Another example

- Suppose we assert these facts:

```
male(albert).
female(alice).
male(edward).
female(victoria).
parent(albert,edward).
parent(victoria,edward).
parent(albert,alice).
parent(victoria,alice).
```

- `albert`, `alice`, `edward` and `victoria` are *atoms*.
- `male`, `female` and `parent` are *predicates*.

- Then let's ask some questions:

```
?- male(albert).
Yes
?- male(victoria).
No
```

- We say we are "performing a query" or "consulting the program or knowledge base".

Variables

```
?- female(Person).
Person = alice ;
Person = victoria ;
No
?- parent(Person, edward).
Person = albert ;
Person = victoria ;
No
?- parent(Person, edward), female(Person).
Person = victoria ;
No
```

- Observe that variable names are capitalized.
- Variables in a query are implicitly "quantified existentially": *Is there any* `Person` such that the `Person` is `edward`'s `parent` and is `female`? Prolog finds an instantiation of `Person` that makes the query true, and tells us about it.

Reading facts or rules from a file

```
?- [facts].
Warning: (/u/prof/ajuma/prolog/facts.pl:3):
        Clauses of male/1 are not together in the source-file
Warning: (/u/prof/ajuma/prolog/facts.pl:4):
        Clauses of female/1 are not together in the source-file
% facts compiled 0.00 sec, 1,536 bytes
Yes
```

- The actual file name is "facts.pl".
- Comments in your file must start with % (for single-line comments) or be enclosed by /* and */ (for multi-line comments).

Typing facts interactively

```
?- likes(bob, candy).
ERROR: Undefined procedure: likes/2
?- [user].
|: likes(bob, candy).
|: % user://1 compiled 0.02 sec, 216 bytes
Yes
?- likes(bob,X).
X = candy
Yes
```

- Using `user` in place of a file name causes Prolog to enter a mode where facts and rules can be entered by the user. To exit this mode, use ctrl-D.

Logic review: symbols

- We'll use the following symbols:

∨ or
∧ and
¬ not
⊃ implies
⇔ if and only if, or "is equivalent to"
∀ for all
∃ there exists

Logic review: implication

- "P ⊃ Q" often causes trouble.
  - You can read it as "P implies Q" or "if P then Q".
  - But it does not mean "P causes Q":
    - e.g., The implication
      "We are indoors" ⊃ "Wheels are round"
      is true but there's no causation involved.
  - And it does not mean "Q is true".
    - e.g., The implication
      "Today is Sunday" ⊃ "Wheels are square"
      is true even though wheels aren't square.

- Implication can be expressed using "or" and "not":
  P ⊃ Q ⇔ ¬P ∨ Q

Logic review: quantifiers

- ∀x [P(x)] : For every single x, P(x) is true.
  - ∀ is the *universal* quantifier.
- ∃x [P(x)] : For at least one x, P(x) is true.
  - ∃ is the *existential* quantifier.

- It's a good idea to use brackets for clarity, especially if there are nested quantifiers.

- In logic programming, the central question is existential: "Does there exist a set of values such that …?"

A logic example

- The universe: a, b, c, d, e
- Facts:
  - P is true of a, b, d
    - That is: P(a), P(b), P(d)
  - R is true of (a, c), (d, d), (b, e)
    - That is: R(a,c), R(d,d), R(b,e)
- Then we know that:
  - ∀x [[∃y R(x, y)] ⊃ P(x)]
  - ∀x [∀y [R(x, y) ⊃ P(x)]]

Another logic example

- The universe: students in this class.
- Predicates:
  - passexam(X): X passes the CSC326 exam.
  - prolog(X): X understands Prolog.
  - python(X): X understands Python.
  - functional(X): X understands functional programming.
- Everyone who understands functional programming, Python, and Prolog passes the exam:
  ∀x [[functional(X) ∧ python(X) ∧ prolog(X)] ⊃ passexam(x)]
- If someone who doesn't understand Prolog passes the exam, then everyone passes the exam:
  [∃x (¬prolog(X) ∧ passexam(X))] ⊃ ∀x passexam(x)

Another logic example

- If someone doesn't understand Python, no one passes the exam.
  [∃x ¬python(X)] ⊃ ∀x ¬passexam(x)

  or, equivalently:

  [∃x ¬python(X)] ⊃ ¬∃x passexam(x)
- No one who doesn't understand functional programming understands Prolog.
  ∀x [¬functional(X) ⊃ ¬prolog(X)]

  or, equivalently:

  ∀x [prolog(X) ⊃ functional(X)]

  or, equivalently:

  ¬∃x [¬functional(X) ∧ prolog(X)]

Prolog rules

- We can state rules of our own:

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

- And then we can make queries:

```
?- sibling(albert,victoria).
No
?- sibling(edward,alice).
Yes
?- sibling(alice,edward).
Yes
```

Prolog rules

- Variables at the head of the rule are (implicitly) quantified universally, and those in the body are quantified existentially (as in queries):
  - The rule we defined says that "For *all* X, Y, X and Y are siblings if there is *some* P such that P is parent to both X and Y."
- A query involving a variable:

```
?- sibling(edward, Sib).
Sib = edward ;
Sib = alice ;
Sib = edward ;
Sib = alice ;
No
```

- Why do we get the same answer twice?
  - There are two parents that yield each answer. That is, there are two instantiations of P that make `sibling(edward, alice)` hold, and there are two instantiations of P that make `sibling(edward, edward)` hold.

Meaning of rules

- A Prolog rule has this form:
  - `c :- a1, a2, a3, …, ak.`
- and this meaning (ignoring the quantification issue):
  - a1 ∧ a2 ∧ a3 ∧ … ∧ ak ⊃ c.
  - A logical statement of this form is known as a "Horn clause".
- Rules for Horn clauses:
  - There can be zero or more *antecedents* (the a's).
  - There cannot be more than one *consequent* (the c).

How to say it

- a1 ∨ a2 ∨ a3 ⊃ c:

```
c :- a1.
c :- a2.
c :- a3.
```

  - The ; operator gives a more concise way to express the above.

```
c :- a1; a2; a3.
```

- a1 ∧ a2 ∧ a3 ⊃ c1 ∧ c2:

```
c1 :- a1, a2, a3.
c2 :- a1, a2, a3.
```

- a1 ∧ a2 ∧ a3 ⊃ c1 ∨ c2 :
  - Can't be done!

Exercises

- Run `pl` on ECF.
  - Enter some facts interactively, and then make some queries.
  - Now add some rules interactively, and make queries.
  - Load facts and rules from a file, and make queries.