CSCI3180: Principles of Programming Languages

Declarative Programming (Part 1)

Wallace Mak

2022-23 Term 2

Topics

- Why Declarative Programming?
- Logic vs Functional Programming
- Prolog

Why Declarative Programming?

- Properties of imperative (conventional) languages
  - State-oriented: each statement execution changes the abstract machine state
  - Destructive assignment as a fundamental operation
    - E.g., `x = x + 1`
  - Side effects can happen
    - E.g., global variables
  - Difficult to read, write, and verify programs

Why Declarative Programming?

- Properties of declarative languages
  - Simple program semantics: “What You See Is What I Mean” (WYSIWIM)
  - Higher program understandability and verifiability
  - Referential transparency
    - Closer to mathematics
    - Computation by values, not by effects
    - Everything is deterministic

Why Declarative Programming?

- From a software engineering point of view
  - Correctness is extremely important
  - The dynamic and interactive environment makes it easy to experiment and change a program while it is being developed
  - Rapid prototyping and exploratory programming for problems that are so complex that no clear solution is available at the start of investigation

Logic vs Functional Programming

- We will cover two major declarative programming paradigms
  - Logic programming, which is relational
  - Functional programming, which is functional
- They share most of the advantages in terms of flexibility and conciseness
- Both paradigms are popular in the AI research community
- Logic programming is also seen used in expert systems
- Functional programming, as a programming paradigm, is getting popular for general uses in recent years as it is being included in many general-purpose languages

Logic Programming in Prolog

- Prolog
  - Stands for *programmation en logique* (French for *programming in logic*)
  - Is a very different language from anything that you have seen before
- While its pattern matching and derivation strategy are novel, the programming methodology that it suggests is of primary importance
- Prolog programming is divided into two stages
  - Asserting what is true (building a program)
  - Asking for consequences of what has been asserted (running a program)
- A Prolog program is a collection of assertions

Prolog Program

- There are two kinds of clauses (assertions)
  - Facts and rules
  - They are used to express relationships amongst some objects
- Facts can be of various arities
  - E.g.:

```
father(edwyn,caroline).
give(tom,apple,teacher).
```

Arity = 2
"Edwyn is the father of Caroline."

Arity = 3
"Tom gives an apple to the teacher."

- Unary facts denote properties
  - E.g.:

```
red(apple).
number(three).
```

"Apple is red."
"Three is a number."

Prolog Query

- A Prolog program is executed by posing a question (or a query), which is a request to establish that a relation (or a conjunction of relations) is “supported” by some collection of assertions
  - E.g.:

```
father(randy,kari).     mother(kari,mary).
father(george,randy).   mother(kari,peter).

?- father(george,randy).   → yes

?- mother(kari,june).      → no

?- father(X,randy).        → X = george
```

“Is there an X such that X is the father of Randy?”

Prolog Query

- There are many ways to execute a program
  - E.g.:

```
father(randy,kari).      mother(kari,mary).
father(george,randy).    mother(kari,peter).
```

"Is there an X such that Kari is the mother of X?"
(Or: "Is there an X such that X is the child of Kari?")

```
?- mother(kari,X).   →   X = mary;
                         X = peter
```

"Are there X and Y such that X is the father of Y?"

```
?- father(X,Y).      →   X = randy
                         Y = kari;

                         X = george
                         Y = randy
```

"Is there an X such that X is the mother of him/herself?"

```
?- mother(X,X).      →   no
```

Prolog Query

- The general form of a query:

  ?- G_1, …, G_m. $(m \geq 0)$

- E.g.: `?- father(X,Z), father(Z,kari).`

  → `X = george`
     `Z = randy`

"Are there X & Z such that X is the father of Z and Z is the father of Kari?" (Or: "Who is the grandfather of Kari?")

- When $m = 0$, we call it an empty query
- Observation: Prolog accepts assertions and attempts to find substitution(s) (for variables) that make a query follow from what has been asserted to be true

Prolog Terms

- Prolog programs are constructed from terms which can be constants, variables or structures
- Constants
  - They represent a specific object
  - They must start with a lower-case letter
  - They can also be numbers, but we won't be using numbers much as we want to focus on the logical part in pure Prolog
- Variables
  - The normal variables must start with an upper-case letter
- Structures
  - They consist of a functor and a number of arguments
  - E.g.: `bonks(big_doge, small_doge)`

Can you spot all the constants, variables, and structures on p.10?

Prolog Terms: Structures

- Suppose we want to represent a location on a map
  - We can use a structure with two components: a latitude (`p`) and a longitude (`q`)
  - The location can then be represented by a term of the form `loc(p,q)`, where `loc` is the functor and `p` and `q` are the arguments
- The choice of functors has no inherent meaning
  - It is entirely a matter of convenience
  - The reader's convenience takes precedence over that of the writer

Prolog Terms: Functor

- Do not confuse “functor” with “function”
  - Functor simply glue some objects into a composite object
  - It does not compute anything
- A functor may have no argument at all
  - We omit the parentheses
  - Such a term looks like a constant and will be treated as one in all respects
- A functor may have only one argument
  - This is useful if we want to label that argument with the term’s functor as some sort of property
    - E.g.: `south(32)`
    - E.g.: `loc(north(45),east(72))` is an example term representing a geographical location

Prolog Rules

- Some concepts are based on the others
- For example, to find a parent, we have to ask *two* questions to obtain a single piece of information
  - E.g.: `?- father(X,kari).` `?- mother(X,kari).`
- If either of the above queries succeeds, a parent of Kari is found
- The user has to translate the "parent" concept into the "father" and "mother" concepts since the program has no knowledge of parenthood
- Rules encapsulate facts (knowledge)
  - We can encode the lacking knowledge by asserting rules, defining "parent" in terms of "father" and "mother"

Prolog Rules

- A rule is of the form

$$H \texttt{ :- } B_1, \ \ldots, \ B_n. \qquad (n \geq 0)$$

  where H is the head and $B_1, \ \ldots, \ B_n$ is the body of the rule
- The `:-` symbol is read as *if* and the commas are read as *and*
- Variables in a rule are universally quantified
  - E.g.:

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

"*For all* X & Y, X is a parent of Y *if* X is a father (mother) of Y."

```
?- parent(X,kari).  →   X = randy
```

```
father(randy,kari).
father(george,randy).
mother(kari,mary).
mother(kari,peter).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
```

Can you see why Randy is a parent of Kari?

Prolog Rules Examples

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

"*For all* X, Y, & Z, X is a grandfather of Z *if* X is a father of Y *and* Y is a parent of Z."

```
?- grandfather(X,Y).
→ X = george    X = randy    X = randy
  Y = kari      Y = mary     Y = peter
```

```
father(randy,kari).
father(george,randy).
mother(kari,mary).
mother(kari,peter).
parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).
grandfather(X,Z) :-
father(X,Y), parent(Y,Z).
```

Are you able to verify them?

Prolog Rules Examples

- Rules for the “ancestor” relation can be defined as follows
  - A parent is an ancestor
  - A parent of an ancestor of an individual X is also an ancestor of X
- The above knowledge can be translated into Prolog as follows:

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

- Rules with no body ($n = 0$) are called unconditional rules
  - E.g.: `loves(X,doge).`
  - E.g.: `loves(doge,X).`

“Everybody loves doge.”
“Doge loves everybody.”

Answer and Response

- In case of success, the answer substitution may assign a constant to some of the variables in the query
- The query with the answer substitution applied to it is the answer, to be distinguished from the “yes” or “no” response
  - E.g.:

```
?- father(X,randy).
X = george
father(george,randy)
yes
```

Answer substitution

Answer

Response

Answer and Response

- If the query contains no variables, then the answer substitution is vacuous - the query itself is the answer, in case of success
  - E.g.:
    ```
    ?- father(george,randy).
    yes
    ```
- When a program consists of facts only, then for an answer to be correct the answer must literally appear as a fact in the program
- If rules are present, we can obtain answers not occurring as facts in the program
  - E.g.:
    ```
    ?- grandfather(george,X).
    X = kari
    ```

Deriving Answers

- The answer "`grandfather(george,kari)`" in the previous slide does not occur as a fact in the program
- We will explore the following with the next example
  - How can we be sure that the answer is correct with respect to the program?
  - How does Prolog derive the answer?

Reduction

```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).   R1
parent(X,Y) :- mother(X,Y).                     R2
mother(caroline,nina).                          R3
father(edwyn,caroline).                         R4
```

- Suppose the initial query is:

  Q_1: `?- grandfather(U,nina).`

- The rule R_1 can be used to reduce this query to another (hopefully easier to answer) by matching the query with the head of R_1, with as result the substitution `X = U` and `Z = nina`

Q_1: `?- grandfather(U,nina).` → `X = U` and `Z = nina` → Q_2: `?- father(U,Y), parent(Y,nina).`

Reduction

```
Q2: ?- father(U,Y), parent(Y,nina).
```

- We now have a compound query in hand
- It can be reduced by selecting a "sub-query" from it, say the 1st, again finding a rule with a matching head, and replacing the selected sub-query by the rule's body and then applying the matching substitution to the entire resulting query

```
Q2: ?- father(U,Y), parent(Y,nina).
```

```
father(edwyn,caroline). R4
```

The body of R_4 is empty.

U = edwyn and
Y = caroline

```
Q3: ?- parent(caroline,nina).
```

Derivation

```
grandfather(X,Z) :- father(X,Y), parent(Y,Z).   R1
parent(X,Y) :- mother(X,Y).                      R2
mother(caroline,nina).                           R3
father(edwyn,caroline).                          R4
```

Q_1: `?- grandfather(U,nina).`

$\downarrow R_1$ {X = U, Z = nina}

Q_2: `?- father(U,Y), parent(Y,nina).`

$\downarrow R_4$ {U = edwyn, Y = caroline}

Q_3: `?- parent(caroline,nina).`

$\downarrow R_2$ {X = caroline, Y = nina}

Q_4: `?- mother(caroline,nina).`

$\downarrow R_3$ {}

Q_5: `?-`

Or □, the empty query

Answer: `grandfather(edwyn,nina).`

- A derivation is a sequence of queries and substitutions
- Each query (except the first) is the result of a reduction of the predecessor in the sequence
- The corresponding substitution in the derivation is the result of the reduction

Successful Derivation

- A derivation is successful if it ends in the empty query
- Every successful derivation gives an answer, which is the initial query with all the substitutions applied to it in the order as they occur in the derivation
- Answers of successful derivations are logical consequences of the program
- In other words, answers of successful derivations are correct with respect to the program

Successful Derivation

- To avoid variable name clashes in the process of derivation, it is important to realize that variables in rules serve only as place holders
  - E.g.: `parent(X,Y) :- father(X,Y).`

    is the same as

    `parent(A,B) :- father(A,B).`
- The matching mechanism in Prolog is two-way, and is called unification
  - E.g.: `father(X, caroline)` matches `father(edwyn, Y)`
    by the substitution `X = edwyn` and `Y = caroline`
- Given a program and a query, there can be more than one successful derivations for the query

Example: Axiomatization of Natural Numbers

- `0`: the number zero
- `s(X)`: the successor of `X` (or `X+1`)
- E.g.: `s(0)`→1, `s(s(0))`→2, `s(s(s(0)))`→3, …

```
sum(0,X,X).
sum(s(X),Y,s(Z)) :- sum(X,Y,Z).
```

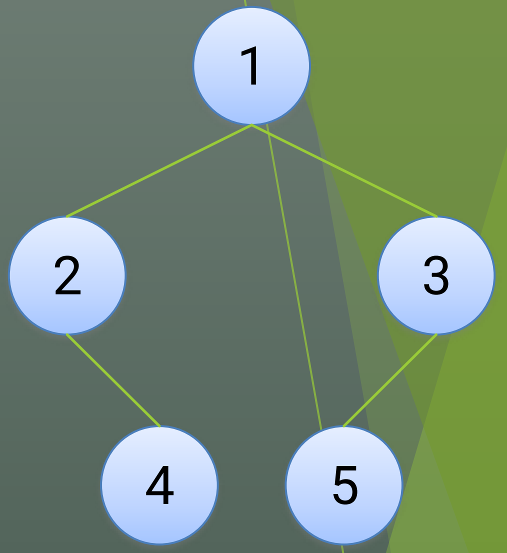
```
?- sum(s(0),s(0),X).
→ X = s(s(0))
```

```
?- sum(X,s(0),s(s(0))).
→ X = s(0)
```

```
?- sum(X,Y,s(s(0))).
→ X = 0          X = s(0)   X = s(s(0))
→ Y = s(s(0))    Y = s(0)   Y = 0
```

Can you draw the successful derivations associated with these answers?

Example: Representing Binary Trees



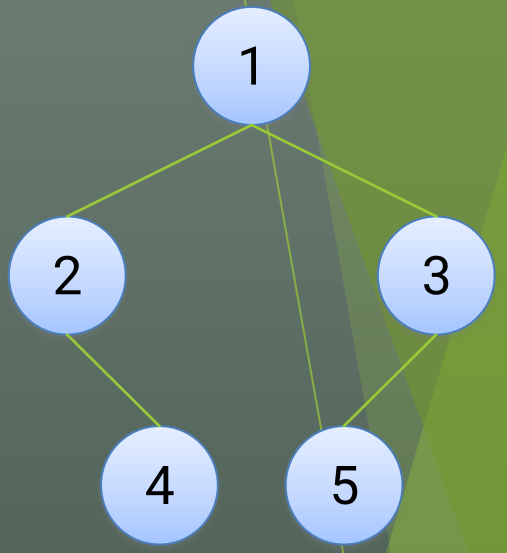

```
btMember(E,bt(L,E,R)).
btMember(E,bt(L,Rt,R)) :- btMember(E,L).
btMember(E,bt(L,Rt,R)) :- btMember(E,R).
```

```
?- btMember(4,bt(bt(nil,2,bt(nil,4,nil)),
                 1,bt(bt(nil,5,nil),3,nil))).
yes
```

```
?- btMember(E,bt(bt(nil,2,bt(nil,4,nil)),
                 1,bt(bt(nil,5,nil),3,nil))).
→ E = 1  E = 2  E = 4  E = 3  E = 5
```

```
bt(bt(nil,
      2,
      bt(nil,4,nil)),
   1,
   bt(bt(nil,5,nil),
      3,
      nil))
```

Example: Representing Binary Trees




```
btMember(E,bt(L,E,R)).
btMember(E,bt(L,Rt,R)) :- btMember(E,L).
btMember(E,bt(L,Rt,R)) :- btMember(E,R).
```

```
?- btMember(1,T), btMember(2,T), btMember(3,T).
```

```
bt(bt(nil,
      2,
      bt(nil,4,nil)),
   1,
   bt(bt(nil,5,nil),
      3,
      nil))
```

What are the answers? Can you draw the successful derivations associated with these answers?