School of Electrical Engineering and Computing

SENG2200/6220 PROGRAMMING LANGUAGES & PARADIGMS

(S1, 2020)

Logic Programming - Prolog

Dr Nan Li
Office: ES222
Nan.Li@newcastle.edu.au

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Outline

- Logical/Declarative Languages
- Prolog
  - *Terminology*
  - *Facts and Rules*
  - *Queries*
  - *Arithmetic*
  - *Lists*
- Predicate Calculus
- Theorem Proving
- Prolog Limitations

Motivation

- In procedural languages, the program specifies a sequence of instructions for solving a problem
  - *A linear decomposition of the problem*
- In functional languages, the program specifies a set of functions which together solve the problem
  - *A hierarchical decomposition of the problem*

Motivation

- If done well, a functional program can *look like* a statement of the problem
  - *Especially true with pattern matching*

```
fib 0 = 1
fib 1 = 1
fib (n+2) = fib (n+1) + fib n
```

  - *But you are still defining the solution to the problem as a set of solutions to subproblems*

Logic Programming

- *Logical* programming language
  - *aka Declarative programming language*
- Express programs in a form of *symbolic logic*
- Use a logical *inferencing* process to produce results

Logic Programming

- Programming is non-procedural
  - *Programs do not state how a result is to be computed, but rather the form of the result*
- Declarative semantics
  - *As only the result needs to be specified, semantics tend to be simple*
  - *Much simpler than the semantics of imperative languages*

Logic Programming

- Example: to sort a list of numbers

```
sort(Old_list, New_list) :-
        permute(Old_list, New_list),
        sorted(New_list)
```

  - *Read as "sorting Old_list to give New_list is successful if Old_list is a permutation of New_list and New_list is sorted"*
  - *Require definitions for* **`permute`** *and* **`sorted`**

Prolog

- Prolog = PROgramming in LOGic
- Developed as a collaboration between University of Aix-Marseille (for natural language processing) and University of Edinburgh (for automated theorem proving)
- We will use Edinburgh syntax
- Online environment, e.g.,

  https://swish.swi-prolog.org/

Prolog

- Terminology…
- *Term* = a constant, variable or structure
- *Constant* = an atom or an integer
- *Atom* = a symbolic value, either:
  - *a word (letters, digits, and underscores) beginning with a **lowercase** letter*
  - *a string of printable ASCII characters delimited by apostrophes*
- *Variable* = a word (letters, digits, and underscores) beginning with an **uppercase** letter

Prolog

- Prolog works by finding an assignment of values to variables which makes the propositions of the program true
  - *This process is called **instantiation***
  - *The final result (list of matched variables) is called a **unification** - the set of variable values that makes the system true*
- An atomic proposition is represented by a *structure of*
  - *A fact statement*
  - *A headless Horn clause*
  - ***functor (parameter_list) .***

Prolog

- Examples of fact statements

```
lecturer(michael).
tutor(luke).
student(melissa).
payedMore(michael, luke).
payedMore(michael, melissa).
payedMore(luke, melissa).
```

Prolog

- An inference rule proposition is represented by a *headed Horn clause*
  - *A rule statement*

    `consequent :- antecedent.`

  - *consequent = LHS = "then part" = a single term*
  - *antecedent = RHS = "if part" = a single term or a conjunction of several term*
  - *conjunction = logical AND = comma-separated list of terms*

Prolog

- Rule statements are only really useful if one or more of the parameters are variables

```
payedMore(X, Y) :-
        lecturer(X), tutor(Y).
payedMore(X, Y) :-
        tutor(X), student(Y).
payedMore(X, Y) :-
        lecturer(X), student(Y).
```

  - *Note there are multiple rules with the same consequent*

Prolog Queries

- A query in Prolog is as a *goal statement*
  - *This can be a single term, or a conjunction of terms, with or without variables*
  - *Without variables, Prolog answers whether the query is true*
  - *With variables, Prolog answers with one set of values which cause the query to be true – if the user responses with ; it continues to find another set of variables*

Prolog Queries

- Consider **`payedMore(michael, X).`**

  Finds rule *`payedMore(X', Y') :-`*

  `lecturer(X'), tutor(Y').`

  Instantiates *`X'`* $\leftarrow$ *`michael, Y'`* $\leftarrow$ *`X`*

  Looks for *`lecturer(michael)`* $\Rightarrow$ *Yes*

  Looks for *`tutor(Y')`*

  Finds *`tutor(luke).`*

  Instantiates *`Y'`* $\leftarrow$ *`luke`*

  Success! Output *`X = luke`*

Prolog Queries

- The user responds `;` = look for more answers

  Backtrack!

  Looks for another `tutor(Y')` $\Rightarrow$ *No*

  Backtrack!

  Finds another rule `payedMore(X', Y') :-`

  `tutor(X'), student(Y').`

  Instantiates `X'` $\leftarrow$ `michael`, `Y'` $\leftarrow$ `X`

  Looks for `tutor(michael)` $\Rightarrow$ *No*

  Backtrack!

Prolog Queries

Finds another rules `payedMore(X', Y') :-`

`lecturer(X'), student(Y').`

Instantiates $X' \leftarrow michael$, $Y' \leftarrow X$

Looks for `lecturer(michael)` $\Rightarrow$ *Yes*

Looks for `student(Y')`

Finds `student(melissa).`

Instantiates $Y' \leftarrow melissa$

Success! Output $X = melissa$

Prolog Queries

- The user responds `;` = look for more answers

  Backtrack!

  Looks for another `student(Y')` $\Rightarrow$ *No*

  Backtrack!

  No more rules matching `payedMore(X, Y)`

  Failed! Output ***No***

- The system should always respond with No unless the query contains no variables or the user aborts before all answers are found

Prolog Queries

- The **=** operator forces two variables to *unify*
  - *In the previous example, Prolog implicitly instantiates*
    `Y' = X`

    ```
    same(X, Y) :- X = Y.
    ```

- The **`not`** operator succeeds if it's argument fails

```
different(X, Y) :- not(X = Y).
siblings(X, Y) :-
    parent(P, X), parent(P, Y),
    not(X = Y).
```

Prolog Arithmetic

- Prolog supports integer variables and integer arithmetic
  - `+ - * / div mod`
  - ***`is`*** *operator: takes an arithmetic expression as right operand and variable as left operand*
  - `A is B / 10 + C`
  - ***`is`*** *forces evaluation – assumes* ***`B`*** *and* ***`C`*** *have been instantiated but* ***`A`*** *has not*
  - *This is not the same as an assignment statement!*

Prolog Arithmetic

- Example

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

Prolog Arithmetic

- Prolog supports comparison of integers
  - ***=:= =\= > >= < =<***
  - *These also force evaluation like* ***is***
- Examples

  1 + 2 =:= 2 + 1 ⇒ Yes

  1 + 2 = 2 + 1 ⇒ No

  1 + A = B + 2 ⇒ A=2, B=1

Prolog Lists

- Prolog has support for lists
  - *The elements of a list can be any terms, including nested lists*
  - *Examples*

    `[apple, banana, orange, grape]`

    `[]` = an empty list

    `[X | Y]` = the list with head **`X`** and tail **`Y`**

Prolog Lists

- Example

```
length([], 0).
length([_|T], N) :-
    length(T, N1),
    N is 1 + N1.
```

  - *Note: _ is the anonymous variable – it holds a place but is never unified or output*
  - *E.g.,*
    - *Query:* `length([1,m,p],X).`
    - *Output:* `X = 3`

Prolog Lists

- Example

```
sorted([]).
sorted([_]).
sorted([X1, X2]) :- X2 >= X1.
sorted([X1, X2 | T]) :-
   X2 >= X1, sorted([X2 | T]).
```

Prolog Lists

- Examples

```
append([], L, L).
append([H | L1], L2, [H | L3]) :-
    append(L1, L2, L3).

reverse([], []).
reverse([H | T], L) :-
    reverse(T, Rev),
    append(Rev, [H], L).
```

Predicate Calculus

- Prolog is built upon the foundation of *first-order predicate calculus*
- A particular form of *symbolic logic* used for logic programming
- Provides a means of…
  - *expressing propositions*
  - *expressing relationships between propositions*
  - *describing how new propositions can be inferred from other propositions*

Predicate Calculus

- Proposition
  - *A logical statement that may or may not be true*
  - *Consists of objects and relationships of objects to each other*
- Objects in propositions are represented by *simple terms*: either *constants* or *variables*
- Constant: a symbol that represents an object
- Variable: a symbol that can represent different objects at different times
  - *Not the same concept as variables in imperative languages*

Predicate Calculus

- *Atomic propositions* consist of <u>*compound terms*</u>
- Compound term:
  - *One element of a mathematical relation, written like a mathematical function*
  - *Two parts*
  - *Functor: function symbol that names the relationship*
  - *Ordered list of parameters*

Predicate Calculus

- Fact:
  - *A compound term where all parameters are constants*

```
lecturer(michael)
tutor(luke)
student(melissa)
likes(michael, steak)
likes(michael, seafood)
likes(luke, seafood)
```

Predicate Calculus

- Query:
  - *A compound term where at least one parameter is a variable*

```
student(X)
likes(michael, Y)
likes(Z, seafood)
```

Predicate Calculus

- Compound proposition:
  - *Two or more atomic propositions…*
  - *Connected by an operator*
  - *Logical operators:* $\neg \ \cap \ \cup \ \equiv \ \supset \ \subset$
  - *Grouping by parentheses* ( )
  - *Quantifiers:* $\forall \ \exists$

Predicate Calculus

Name	*Symbol*	*Example*	*Meaning*
negation	$\neg$	$\neg$ a	not a
conjunction	$\cap$	a $\cap$ b	a and b
disjunction	$\cup$	a $\cup$ b	a or b
equivalence	$\equiv$	a $\equiv$ b	a is equivalent to b
implication	$\supset$ $\subset$	a $\supset$ b a $\subset$ b	a implies b b implies a

Predicate Calculus

- Examples

```
likes(michael, steak) ⊃ getsFat(michael)

getsIodinePoisoning(luke) ⊂  likes(luke,
   seafood)

retiresComfortably(michael) ⊂
     ( worksHard(michael)
       ∩ ¬ madeRedundant(michael) )
     ∪ winsLotto(michael)
```

Predicate Calculus

- Variables are introduced by logical *quantifiers*

Name	*Example*	*Meaning*
universal	$\forall X.P$	For all X, P is true
existential	$\exists X.P$	There exists a value of X such that P is true

Predicate Calculus

- Examples

$\exists X.(\ getsIodinePoisoning(X) \subset likes(X, seafood)\ )$

$\neg\ \forall X.(\ getsIodinePoisoning(X) \subset likes(X, seafood)\ )$

$\exists X.(\ (retiresComfortably(X) \subset (worksHard(X) \cap \neg\ madeRedundant(X)) \cup winsLotto(X)\ ) \supset lecturer(X)\ )$

Predicate Calculus

- There may be an exponential number of ways to write the same compound proposition
- Clausal form
  - $B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$
  - *if all the* **A***s are true, then at least one* **B** *is true*
  - *Antecedent: right side of* $\subset$
  - *Consequent: left side of* $\subset$
- Any predicate calculus proposition can be converted algorithmically into clausal form

Theorem Proving

- Restrict each proposition to a *Horn clause*
  - *Clausal form with zero or one atomic propositions in the consequent*
  - *An empty consequent (a headless Horn clause) is used to express facts*
  - *Most propositions can be stated as Horn clauses*

Theorem Proving

- Resolution
  - *An algorithmic process whereby a new proposition is inferred from two existing propositions*
  - *If the consequent of one rule is part of the antecedent of another rule then merge the two rules…*
  - $\mathbf{Resolve}(A \subset B \cap P,\ P \subset C)$

    $\Rightarrow A \cap P \subset B \cap P \cap C$
  - *Then eliminate common parts* P

    $\Rightarrow A \subset B \cap C$

Theorem Proving

- Propose a theorem then test whether it is true given the known axioms and theorems
- Proof by construction
  - *For existential theorems*
  - *Search the known axioms for a combination which makes the proposed theorem true*
- Proof by contradiction
  - *For universal theorems*
  - *Search the known axioms to ensure there is no combination for which the proposed theorem is not true*

Theorem Proving

- Bottom-up resolution, **forward chaining**
  - *Begin with facts and rules and attempt to infer the goal*
  - *Works well with a large set of possibly correct answers*
- Top-down resolution, **backward chaining**
  - *Begin with a goal and attempt to find a sequence that leads to a set of facts*
  - *Works well with a small set of possibly correct answers*
- Prolog implementations use backward chaining

Theorem Proving

- Matching
  - *Finding a fact or the consequent of a rule which could satisfy the goal proposition*
- Unification
  - *Finding values for all variables in a rule such that it satisfies the goal proposition*
- Instantiation
  - *Assigning temporary values to the variables of one atomic proposition within the (sub)goal*

Theorem Proving

- Depth-first search
  - *Recursively unify one subgoal before attempting to unify any others at this level*
- Breadth-first search
  - *Attempt to unify all subgoals in parallel*
  - *Can be faster but requires much greater computing resources*
- Prolog uses depth-first search

Theorem Proving in Prolog

- Inside Prolog:
  - *Search the database (top-to-bottom in the source file) for a match for the current goal*
  - *If the match contains variables, then instantiate any variables possible with their values*
  - *If the match is a fact, then return success*
  - *If the match is a rule, solve for each subgoal (left-to-right in the source file)...*

Theorem Proving in Prolog

- If all subgoals succeed, then return success
- If a subgoal fails, then
  - *Uninstantiate any variables of that subgoal*
  - *Backtrack to the previous subgoal and look for the next match*

Theorem Proving in Prolog

- Helpful to assume every rule has an extra subgoal
  - *This subgoal displays the current values of all variables then awaits the user's response*
  - *If the user says to stop, then this subgoal succeeds and the theorem is proved*
  - *If the user asks for another solution, then this subgoal fails and backtracking occurs*

Prolog Examples

```
lecturer(michael).
tutor(luke).
student(melissa).

payedMore(X, Y) :- lecturer(X), tutor(Y).
payedMore(X, Y) :- tutor(X), student(Y).
payedMore(X, Y) :- lecturer(X), student(Y).

?- trace.
?- payedMore(michael, X).
```

Prolog Examples

```
professor(peter).
payedMore(X, _) :- professor(X).
payedMore(_, X) :- student(X).

?- payedMore(X, michael).
?- payedMore(peter, X).
?- payedMore(X, peter).
```

Prolog Limitations

- Prolog can only report "Yes" or "No" based on the knowledge in its database of facts and rules
  - *Called the closed-world assumption*
  - *"Yes" means it can satisfy the proposition given the facts and rules in the database*
  - *"No" does not mean "false"*
  - *"No" means it cannot satisfy the proposition **given the facts and rules in the database***
  - *Prolog is a true/fail system*

Prolog Limitations

- Consider this example

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

Now

`sibling(X, Y).` *responds*

```
   X = michael, Y = michael
```

Prolog Limitations

- Solution 1:

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

  - *For every pair of people in the database, add* `notSame(person1, person2).`

- Solution 2:

```
sibling(X, Y) :- parent(P, X), parent(P,
    Y), not(X = Y).
```

  - *Works for this simple example*

Prolog Limitations

- **`not(X)`** succeeds if **`X`** fails
- **`not(X)`** fails if **`X`** succeeds
  - *So you couldn't put **`not(X = Y)`** before the **`parent()`** propositions, as **`X=Y`** will always succeed, hence **`not(X=Y)`** will always fail*

Prolog Limitations

- Does **`not(not(X)) = X`**?
  - *Recall that, if **`func(X)`** fails, then **`X`** is uninstantiated before backtracking*
  - ***`not(X)`** fails, so **`X`** is uninstantiated*
  - ***`not(not(X))`** succeeds…*
  - *but **`X`** is left uninstantiated!!*

Prolog Limitations

- In a purely logical programming language, the order of evaluating subgoals is indeterminate
  - *Effectively evaluated in parallel*
- Prolog is not purely logical
  - *Always matches propositions top-to-bottom*
  - *Always evaluates subgoals left-to-right*

References

- R. W. Sebesta, “Concepts of Programming Languages”, 9th Edition, Addison-Wesley, 2010 (Chapter 16)
- I. Bratko, “Prolog Programming for Artificial Intelligence”, Addison-Wesley, 1986