Language Processing with Prolog

CS4212 – Lecture 2 (live)

Facts About Prolog

- PROLOG = PROgramming in LOGic
  - Implementation of a theorem proving procedure for Horn Clauses, called *resolution*
  - Resolution procedure discovered by Alan Robinson in 1965
  - Prolog invented by Alain Colmerauer in early 1970s
  - Not fully standardized, many dialects available

- Declarative language
  - Variables are similar to mathematical variables: cannot be assigned.
  - Recursion is the main way of performing repetitive computations.
  - Powerful symbolic processing features

- Full-fledged language
  - Powerful meta-processing and reflection capabilities (self modifying code straightforward to implement)
  - Allows modeling of assignment if necessary
  - Many mature implementations available with comprehensive libraries and APIs
  - Interface packages with all mainstream languages and programming platforms

Strengths of Prolog

- Integrated programmable compiler front-end
  - Straightforward modeling of programming language syntax.
  - Program in user-defined language is equated with its AST
- Rule-based language
  - Straightforward encoding of PL semantics
  - Pattern-matching $\Rightarrow$ syntax directed processing
- Naive compilers in under 30 lines of code.
- Preferred Prolog system: SWI-Prolog
  - `www.swi-prolog.org`

Running Prolog

- The system opens with a Read-Eval-Print loop
  - The strings passed on to the system are called *queries*
- An editing window for rules can be opened with the pseudo-query (directive)

  ```
  ?- edit(file('name.pro')).
  ```
- Other directives can be used to load files or libraries, set or get system options, etc.
- The rules in the editing window can be compiled with `C-c C-b`.
  - The rules are analogous to procedures, similar to other systems that have REP loops (Python, Scheme, Ocaml, Haskell).
  - Queries act as main programs that call the defined procedures

Syntactic Components of Prolog

[B]

- Atomic elements
  - Numbers (usually infinite precision)
  - Atoms: identifiers starting with lower case, strings in simple quotes, sequences of special symbols.
  - Variables: identifiers starting with upper case letter or underscore
  - The underscore is called the *anonymous variable*
- Terms (equivalent to structures in C or Ocaml)
  - Atomic element
  - Atom applied to tuple of terms
  - Unbounded nesting
  - Example: `f(g(3),X,b)`
  - Used to model structured data.
- Rules: discussed shortly.
  - Define execution entities similar to procedures in other languages.
  - Model a *logical inference*
- Queries
  - Special kind of rule
  - Serve as a main program
  - Can be either launched at the REP loop, or added to a file to produce a standalone executable

Prolog Variables

- Closer to the concept of mathematical variable
  - Cannot be assigned

- Two states
  - *Unbound* or *free:* initial state, when the variable does not have a value
  - *Bound:* After it has received its value, through pattern matching or equality constraint
  - A bound variable will keep its current value till the end of program's execution

- Variable renamings
  - Every time a rule is selected for execution, its variables are renamed into fresh variables with unseen-before names.
  - This simulates the concept that the variables of a rule are "local" to the rule.
  - If `X1` is to be renamed into `X2`, then the term `f(g(X1,Y),h(Z),X1)` becomes `f(g(X2,Y),h(Z),X2)` (i.e. all occurrences are renamed).

- Anonymous variable
  - Used when we don't care about the value of an argument
  - Each occurrence denotes a different, separate variable, distinct from all the other occurrences

Equality Constraints

- Of the form `Term`$_1$ `=` `Term`$_2$
- Constraint stating that two terms should be identical
- The system tries to find values for variables in `Term`$_1$ and `Term`$_2$ so that the two terms become identical
  - Not always possible
- If constraint is satisfiable, the values found are *the variable bindings* (which cannot be subsequently changed)
- The procedure of checking satisfiability of equality constraint and computing variable bindings: *unification*
  - Part of the resolution procedure
- Equality sign does not mean assignment!

Equality Constraint Queries

- Equality constraints can be passed as queries directly to the REP
- Variable bindings printed out as answers to the query
- Multiple constraints can be passed into a single query, separated by commas
  - Comma has the role of conjunction
  - All constraints must be satisfied simultaneously
  - Variables with the same name are the same in different constraints
  - Constraints will be attempted one by one, from left to right
  - Each constraint will produce a set of bindings; each bound variable will denote its current value when attempting subsequent constraints.
  - REP prints the overall answer, which satisfies all constraints simultaneously

Equality Constraint Query Examples


```
1 ?- f=f.
true.

2 ?- f=g.
false.

3 ?- f(a) = f(a).
true.

4 ?- f(a,b) = f(a,c).
false.

5 ?- f(a,X) = f(Y,b).
X = b,
Y = a.

6 ?- X = f(a).
X = f(a).

7 ?- f(a,g(b,c)) = X.
X = f(a, g(b, c)).

8 ?- f(a,g(b,Y)) = X.
X = f(a, g(b, Y)).

9 ?- f(a,g(b,Y)) = X, Y=c.
Y = c,
X = f(a, g(b, c)).
```

```
10 ?- X = f(X).
X = f(X).

11 ?- X = f(X), X = a.
false.

12 ?- set_prolog_flag(occurs_check,true).
true.

13 ?- current_prolog_flag(occurs_check,X).
X = true.

14 ?- X = f(X).
false.

15 ?- set_prolog_flag(occurs_check,false).
true.

16 ?- X = f(X).
X = f(X).

17 ?- f(X1,X2,X3,X4) = f(g(X2,X2),g(X3,X3),g(X4,X4),a).
X1 = g(g(g(a, a), g(a, a)), g(g(a, a), g(a, a))),
X2 = g(g(a, a), g(a, a)),
X3 = g(a, a),
X4 = a.

18 ?- f(_,_) = f(a,b).
true.
```

Arithmetic Constraints

- Bindings can also be computed as results to aritmetic equations
  - Convenient way of performing arithmetic.
    `?- [library(clpfd)].` must be loaded first.

- Example: `X + 3 #= 5` binds `X` to `2`.

- Note the use of the `#=` operator

- Other valid arithmetic constraint operators: `#<`, `#>`, `#=<`, `#>=`, `#\=`.

- The use of the *impure* operators `<`, `>`, `=<`, `>=`, `\=` is not necessary in this module.

- More complicated systems of equations may need the use of explicit search procedures, details in the example

Arithmetic Constraints Example

[C]

```
19 ?- [library(clpfd)].
true.

20 ?- X+3 #= 5.
X = 2.

21 ?- X+Y #= 5, 4-Y #= 2.
X = 3,
Y = 2.

22 ?- X #>= 3, X #=< 3.
X = 3.

23 ?- 0 #=< X + Y, X #< Y, Y #=<1.
X in -1..0,
X#=<Y+ -1,
X+Y#=_G1133,
Y in 0..1,
_G1133 in 0..1.
```

```
24 ?- 0 #=< X + Y, X #< Y, Y #=<1, label([X]).
X = -1,
Y = 1 ;
X = 0,
Y = 1.

25 ?- X + Y #= 3, X - Y #= 1.
1+Y#=X,
X+Y#=3.

26 ?- X + Y #= 3, X - Y #= 1, X in -10..10, label([X]).
X = 2,
Y = 1.

27 ?- 0 =< X + Y, X < Y, Y =<1.
ERROR: =</2: Arguments are not sufficiently instantiated
```

Rules

- Of the form:

```
head_atom(...)
        :- body_atom1(...),...,body_atomk(...).
```

- The :- has the meaning of *reverse implication*
  - $p \leftarrow q$ means p is *implied* by q, or p if q

- The body may be empty ⇒ *fact*.

```
head_atom(...).
```

  - The full-stop at the end is significant and should not be omitted
- Describes a logical inference

- Provides a computational way to solve a query
  - If an atom from a query *matches* the head of a rule, then that atom can be replaced by the body of the rule.

- Examples (specification of arcs and paths in a graph)

```
arc(a,b).  arc(a,c).  arc(b,c). % facts specifying the graph
path(X,Y) :- arc(X,Y).
                   % we have a path from X to Y if we have an arc from X to Y
path(X,Y) :- arc(X,Z), path(Z,Y).
                   % we hava a path from X to Y if there exists some Z
                   % such that there is an arc from X to Z and
                   % there is a path from Z to Y
```

More Prolog Terminology

[C]

- *Predicate:* atomic formula of the form `p(arg`$_1$`,...,arg`$_k$`)`, expected to be either *true* or *false*.

- *Query:* conjunction of predicates
  - The system is queried whether the predicates are simultaneously true.
  - If the query has variables, the system tries to find bindings for variables that make the query *true*.
    - If such bindings are found, the system returns them as *answer*
    - If no such bindings exist, the system returns *false*
  - If the query has no variables, the system answers either *true* (success) or *false* (failure).

- *Predicate definition:* set of rules with the same predicate on the LHS
  - Similar to a procedure in procedural languages – this may be misleading to the beginner
  - Define your predicates and rules so that they make *logical sense*
  - Forget about sequential execution

- *Prolog program:* set of predicate definitions

Factorial in Prolog

[C]

```
fact(0,1).
fact(N,R) :-
    N #>0, N1 #= N-1, R #= R1*N, fact(N1,R1).

28 ?- fact(5,120).
true ;
false.

29 ?- fact(5,121).
false.

30 ?- fact(5,X).
X = 120 ;
false.

31 ?- fact(X,120).
X = 5 ;
false.

32 ?- fact(X,Y).
X = 0,
Y = 1 ;
X = Y, Y = 1 ;
X = Y, Y = 2 ;
X = 3,
Y = 6 ;
X = 4,
Y = 24 ;
X = 5,
Y = 120 .
```

```
fact(0,1) :- !.
fact(N,R) :-
    N #>0, N1 #= N-1, R #= R1*N, fact(N1,R1).

33 ?- fact(5,120).
true.

34 ?- fact(5,121).
false.

35 ?- fact(5,X).
X = 120.

36 ?- fact(X,120).
X = 5.

37 ?- fact(X,Y).
X = 0,
Y = 1.
```

The *cut* **!** : pseudo-predicate that avoids redundant searches

Execution


```
fact(0,1).
fact(N,R) :-
    N #>0, N1 #= N-1, R #= R1*N, fact(N1,R1).
```

The Solution Search Process

- The system processes the rules symbolically, performing a process similar to algebraic simplification

- An exhaustive search for a solution is performed
  - Inherently inefficient
  - Leads to concise, expressive programs

- Programs exhibit modes of use that may not have been designed by the programmer
  - Function inversion in factorial
  - Generation of infinite stream of solution sets

Prolog Operators

- Operators are just *sugared atoms*

- The query `?- +(2,3) = 2 + 3` succeeds (i.e. returns *true*)
  - `+(2,3)` and `2+3` are two ways of expressing the same thing.
  - The arithmetic evaluation `X #= +(2,3)` will still bind `X` to `5`.
  - Operators are not typed at the language level: `?- * * * = *(*,*).` succeeds.
  - Notice the spaces in `* * *`. Without spaces, `***` is a single atom.

- Specific predicates may enforce types
  - The query `?- X #= * * * .` yields a semantic error.

- Operators are defined with *associativity* and *precedence*.
  - `?- X + Y = 1 + 2 + 3.` binds `X=1+2` and `Y = 3`
  - `?- X + Y = 1*2+3.` binds `X=1*2` and `Y = 3`
  - `?- X*Y = 1+2*3.` fails.

Operator Declarations

[A]

- Predefined operators cannot be redefined.
  - The set is dependent on the system; some commonalities exist.

- Any atom not already defined as operator can be added to the set. Pseudo-predicate that adds an operator

  `?- op(`*Precedence*`,` *Associativity*`,` *Atom*`)`

Use `?- help(op).` to see the predefined operators

- Example:

```
44 ?- op(450,xfy,a). % 'a' is a new operator with precedence 450 (between + and *), right-associative
true.

45 ?- X a Y = 1 a 2 a b.
X = 1,
Y = 2 a b.

46 ?- X a Y = (1 a 2) a 3.
X = 1 a 2,
Y = 3.

47 ?- X + Y = 1 a 2 + 3 a 4.
X = 1 a 2,
Y = 3 a 4.

48 ?- X a Y = 1 a (2 + 3 a 4).
X = 1,
Y = 2+3 a 4.

49 ?- X a Y = 1 * 2 a b * c.
X = 1*2,
Y = b*c.

50 ?- X a Y = a a a.
X = Y, Y = (a).
```

Associativity specifier:

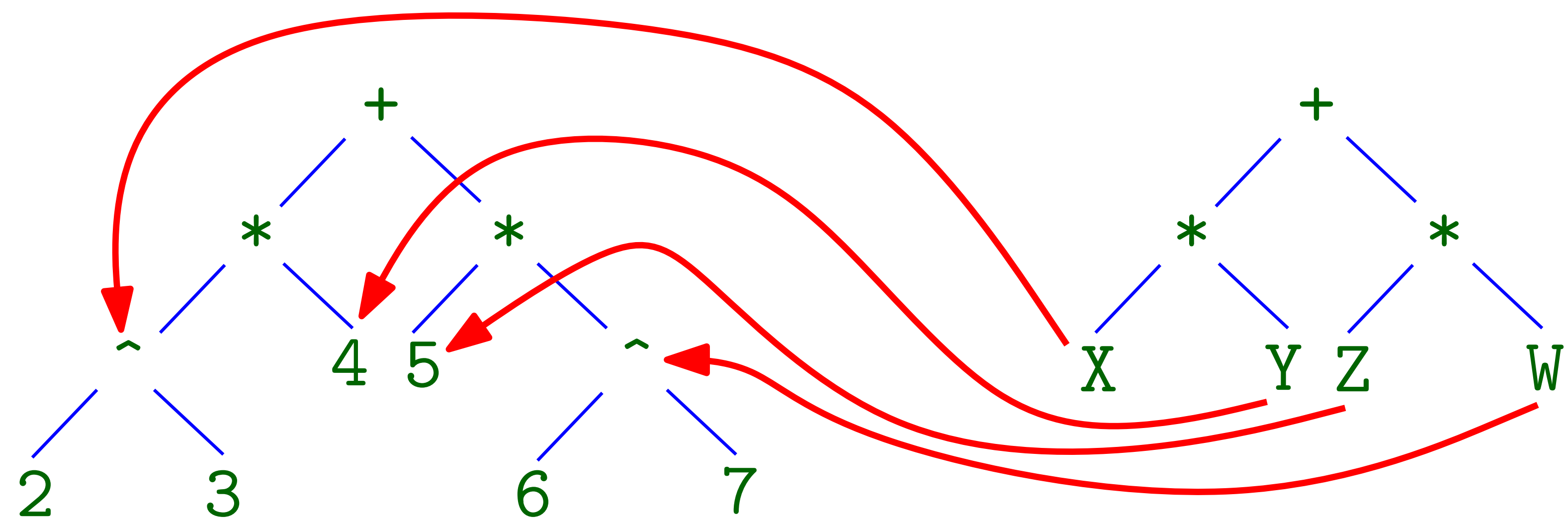
- `xfx` : binary non-associative operator
- `xfy` : binary right-associative operator
- `yfx` : binary left-associative operator
- `yfy` : not allowed
- `xf` : unary postfix non-associative operator
- `yf` : unary postfix associative operator
- `fx` : unary prefix non-associative operator
- `fy` : unary prefix associative operator

Syntactic Trees

[C]

`2^3*4+5*6^7` $\equiv$ `+(*(^(2,3),4),*(5,^(6,7)))`

`X * Y + Z * W` $\equiv$ `+(*(X,Y),*(Z,W))`




```
51 ?- X * Y + Z * W = 2 ^ 3 * 4 + 5 * 6 ^ 7 .
X = 2^3,
Y = 4,
Z = 5,
W = 6^7.

52 ?- X * Y + Z * W = a ^ b * c + d * e ^ f .
X = (a)^b,
Y = c,
Z = d,
W = e^f.
```

Modeling a Programming Language

[C]

```
?- op(1099,yf,;).
?- op(960,fx,if).
?- op(959,xfx,then).
?- op(958,xfx,else).
?- op(960,fx,while).
?- op(959,xfx,do).

2 ?- ( Instruction1 ; Instruction2 ; while ( B ) do { S } ) =
|    ( a = 240 ; b = 144 ;
|      while ( a \= b ) do {
|          if ( a < b )
|              then { b = b - a }
|              else { a = a - b }
|      } ).
Instruction1 = (a=240),
Instruction2 = (b=144),
B = (a\=b),
S = (if a<b then b=b-a else a=a-b).
```

A Simple Interpreter (1)

[C]

```
eval(N,_,_,N) :- integer(N), !.

eval(a,A,_,A).
eval(b,_,B,B).

eval((E1-E2),A,B,Result) :-
    eval(E1,A,B,R1), eval(E2,A,B,R2), Result #= R1-R2.

eval((E1\=E2),A,B,1) :-
    eval(E1,A,B,R1), eval(E2,A,B,R2), R1 #\= R2,!.
eval((_\=_),_,_,0).

eval((E1<E2),A,B,1) :-
    eval(E1,A,B,R1), eval(E2,A,B,R2), R1 #< R2, !.
eval((_<_),_,_,0).
```

A Simple Interpreter (2)


```
interp((a = Expr),Ain,B,Aout,B) :- eval(Expr,Ain,B,Aout).
interp((b = Expr),A,Bin,A,Bout) :- eval(Expr,A,Bin,Bout).

interp((if B then { S1 } else { S2 } ),Ain,Bin,Aout,Bout) :-
    eval(B,Ain,Bin,Bval),
    (   Bval #= 1
    ->  interp(S1,Ain,Bin,Aout,Bout)
    ;   interp(S2,Ain,Bin,Aout,Bout) ).
```

A Simple Interpreter (3)


```
interp((while B do { S }),Ain,Bin,Aout,Bout) :-
    interp(
        (if B then {
                S ;
                while B do { S }
              }
              else { a = a }
        ),Ain,Bin,Aout,Bout).

interp((S1;S2),Ain,Bin,Aout,Bout) :-
    interp(S1,Ain,Bin,Aaux,Baux),
    interp(S2,Aaux,Baux,Aout,Bout).
```

A Simple Interpreter (4)


```
?- Program =
     ( a = 240 ; b = 144 ;
       while ( a \= b ) do {
           if ( a < b )
               then { b = b - a }
               else { a = a - b }
       } ),
     interp(Program,0,0,A,B).
A = 48
B = 48
```

```
?- Program =
     ( a = 0 ; b = 0 ;
       while ( a \= 10 ) do {
           a = a - -1 ;
           b = b - (0 - a)
       } ),
     interp(Program,0,0,A,B).
A = 10
B = 55
```

Advanced Programming Techniques in Prolog

- We need several advanced features, and system library predicates, to write efficient, elegant programs.
  - Lists, dictionaries, dynamic generation of names
  - Introduced in the next online video recording

- Debugging Prolog programs requires special training
  - Using the IDE's debugger requires rather deep understanding of Prolog's execution model — not a feasible option for this module.
  - Instead, we shall learn how to add printing statements to reveal the internal state of the program

Conclusion

- Prolog is a rule-based language where we express relationships between data items
  - Thinking sequentially is counterproductive
- Prolog operators have the power of dynamically defining syntax analyzers
  - PL keywords can be defined as operators
  - Source programs can be expressed as Prolog terms
  - We get access to the hierarchic structure of the program
- PL manipulation is best done in a rule-based fashion
- Toy interpreter in about 30 lines of Prolog program