CSI 2165 Winter 2006

Diana Inkpen
SITE
University of Ottawa

Part III

Prolog – Part III

- Input and output
- Prolog databases
- Abstract data structures
  - Tree, stacks, queues, graphs

Character Input

- get(Char)
  - Reads the next character from the standard input device (the keyboard) **skipping non-printing characters** (blanks,tabs, etc)
  - Unifies the ASCII value of the character with Char

```
? - get(Char).
A
Char = 65
? - get(Char).
    A       % spaces before A
Char = 65
```


Character Input

- get0(Char)
  - Reads the next character from the standard input device (the keyboard) **without** skipping non-printing characters
  - Unifies the ASCII value of the character with Char

```
? - get0(Char).
A
Char = 65
? - get0(Char).
      % here I typed a space
Char = 32
```


Character Input - Example

- **Reading a sentence:**

```
get_sentence(Sentence) :-
   get0(FirstChar),
   get_characters(FirstChar, Sentence).

get_characters(46,[46]) :- !.          %46 is the ASCII code for '.'
get_characters(Char, [Char|RestOfSentence]) :-
   get0(NextChar),
   get_characters(NextChar, RestOfSentence).
```


name/2

- name(Atom, List)
  - converts an atom to a list of ASCII values, or vice versa
  - converts a list of ASCII values to an atom

```
?- name(yong, YongList).
YongList = [121, 111, 110, 103] ;

? - name(Atom,[65, 66]).
Atom = 'AB'

? - get_sentence(S), name(A, S).
```

Character Output

- put(Char) - writes Char to the standard output device.
  - Where C is the ASCII code of the character to be output.

```
?- put(65), put(66), put(67).
ABC
```


Term Input

- read(Term) - reads a term from the standard input device
  - The Term will match the input term.

```
? - read(X).
a(b,c,[atom]).
X = a(b,c,[atom])

? - read(X), functor(X, F, A).
stru(arg,[1,2,3],argthree).
X = stru(arg,[1,2,3],argthree)
F = stru
A = 3

? - read(X).
4.2E+01.
X = 42

? - read(X), Y is X.
3 * 2 + 1.
X = 3 * 2 + 1
Y = 7
```


Term Output

- write(Term) - writes the term Term to the standard output device

```
? - write('hello world!').
hello world!

? - write(4.2E+01).
42.0

? - write([1,2,3,4]).
[1,2,3,4]
```

- nl - writes a newline to the standard output device

```
? - nl, write('hello world!'), nl.

hello world!

yes
```


Input/Output - Example

```
likes(john, apples).
likes(john,beer).
likes(john,hockey).

nice_writing :-
    likes(X,Y), write_nicely(likes, X, Y), fail.
nice_writing.

write_nicely(Functor, X, Y) :- convert(X, NewX), write(NewX),
  write(' '), write(Functor), write(' '), write(Y), write('.'), nl, !.

convert(X, NewX) :-  name(X,[First | Rest]),
          upper(First, Capital), name(NewX,[Capital | Rest]).
upper(X,Y) :- Y is X - 32.
```


Input/Output - Redirecting

- Redirecting standard output:
  - tell(Filename) - opens the file named Filename for writing and makes it the current output file (instead of the standard out)
  - told - closes the file that was opened for writing by tell/1.
- Redirecting standard input:
  - see(Filename) - opens the file named Filename for reading and makes it the current input file (instead of the standard in).
  - seen - ends input from the file opened by see/1 and closes the file

```
?- see('facts.pl'), tell('newfacts.pl'), write_facts, seen, told.

write_facts :- read(Fact),   Fact =.. [Name, X, Y],
               write_nicely(Name, X, Y),  write_facts.
write_facts.
```


Explicit File Input/Output

- open(Handle, Filename, Access) → opens the file named Filename. Access can be:
  - r → read
  - w → write
  - a → append
  - rw → read/write
  - ra → read/append
- create(Handle, Filename) → opens a new file (whose name will be Filename) for writing. If a file named Filename already exists, the new file will overwrite the existing file
- close(Filename) → closes the file whose name is Filename

Reading Programs

- consult(Filename) - reads a program from a file
  - **effect:** all clauses in the file Filename are read and will be used by Prolog when answering further questions from the user.
  - if another file is 'consulted' at some later time during the same session, clauses from this new file are simply added at the end of the current set of clauses.
- reconsult(Filename) - similar to consult
  - **effect:** clauses in the file Filename replace existing clauses (if any).

The Prolog Database

- The facts and rules (clauses) in a Prolog program are often referred to as the **Prolog Database**. Querying the Prolog program is analogous to querying a database.
- All the Prolog programs we have seen so far have been **static** databases: the facts and rules are entered by the programmer *before* execution, and remain unchanged during execution.
- There are built-in predicates that allow us to modify the database **dynamically**: assert/1 and retract/1

Note: assert/1 and retract/1 affect only the working **memory** of Prolog, not your program file

Assert

- This built in predicate allows us to add clauses to the Prolog Database during execution.
- There are three forms for this predicate:
  - assert/1 → adds a clause at the **end** of the database
  - asserta/1 → adds a clause at the **beginning** of the database
  - assertz/1 → adds a clause at the **end** of the database

Assert

- Example(**Demo**): start SWI-Prolog and try the following.

```
?- listing.
?- assert(happy(mia)).
?- listing.
?- assert(happy(vincent)).
?- assert(happy(mike)).
?- assert(happy(butch)).
?- assert(happy(vincent)).
?- listing.
?- assert(naïve(X) :- happy(X)).
?- listing.
```


Retract

- This predicate allows us to delete clauses. A handy feature of retract/1 is that it will instantiate any uninstantiated variables in its argument.
- It has only one form: retract/1 → it will retract the **first** clause that matches its argument.
- It is backtrack-able when you use variables!

Retract

- Example(**Demo-cont.**):

```
?- retract(happy(mike)).
?- listing.
?- retract(happy(vincent)).
?- listing.
?- retract(naïve(X) :- happy(X)).
?- listing.
```

Assert - Example

```
addition_table(A) :-
    member(B, A),
    member(C, A),
    D is B + C,
    assert(sum(B, C, D)),
    fail.

?- addition_table([0,1,2,3,4,5,6,7,8,9]).
?- listing.
```

What if we don't have "fail"?

?- retract(sum(X, Y, Z). **Or**
?- retract(sum(X, Y, Z), fail.

Find and Keep

- How can we find and keep all the solutions to some query?
  - Find
    - use ';' – it is manual
    - use 'fail' - at each step the variables become uninstantiated, all values are tried, then fails
    - use 'findall'
  - Keep
    - use assert to keep partial results in the memory

Assert – Another Example (1/2)

- The **Fibonacci** sequence –
  - a sequence of numbers in which each number is the sum of the two previous numbers in the sequence: 1, 1, 2, 3, 5, 8, 13, …
- The standard recursive solution for finding the N^{th} number in the Fibonacci sequence:

```
fibonacci(1, 1).
fibonacci(2, 1).
fibonacci(N, FibN) :- N1 is N-1,  N2 is N-2
   fibonacci(N1, FibN1),  fibonacci(N2, FibN2),
   FibN is FibN1 + FibN2.
```

This program is extremely inefficient. How inefficient is it?

Assert – Another Example (2/2)

- The problem with fibonacci/2 is the fact that it cannot remember the answers to previous calls, which means that it must recompute every number in the sequence each time it's called. We can give it memory, though by using assert.

```
:- dynamic fibonacci/2.
fibonacci(1,1).
fibonacci(2,1).
fibonacci(N,FibN) :- N1 is N-1, N2 is N-2,
   fibonacci(N1,FibN1),   fibonacci(N2,FibN2),
   FibN is FibN1 + FibN2,
   asserta(fibonacci(N,FibN)).
```

- Now, whenever the program computes the N^{th} number in the sequence, it asserts it as a fibonacci/2 fact, **before** the other clauses for fibonacci/2.

Data Types

- Prolog is not a *typed* language like Pascal or C. That is, when we write a Prolog program, we can use objects without telling the interpreter what are the *types* of the objects.
- But Prolog has some type system: integer, float, atom, structure, list, …

Abstract Data Types

- Built-in data types in a programming language are
  - *primitive* (integer, float, atom) or
  - *complex* (it is made up of other built-in types, either primitive or complex)
- *Abstract* data-types are normally *not* built-in. They usually represent kinds of complex objects in the world and the operations that are performed on these objects.
- There is no special notation for abstract data types in Prolog. The format of an abstract data type is arbitrary; its interpretation is based on convention.

Abstract Data Types

- **trees**:
  - create a new tree
  - search through a tree for an element
  - insert an element in a tree
  - traverse a tree
  - delete an element from a tree
- **stacks:**
  - create a new stack
  - push an element onto a stack
  - pop an element off a stack
- **queues:**
  - create a new queue
  - add an element to the end of a queue
  - remove an element from the front of a queue
  - check the element at the front of a queue
  - check if a queue is empty

Binary Search Trees

- Trees can be represented as structures like this:
  `bt(Key, LeftSubTree, RightSubTree)`
- Leaves can be denoted like this: `bt(key, nil, nil)`
- nil is just a constant to represent an empty tree
- A binary search tree is a binary tree such that the value of the key at a node is larger than the keys in the left subtree and smaller than the keys in the right subtree.

- To test whether a key is in a binary search tree we a implement btMember(E,T):

```
btMember(E, bt(E, _L, _R)).
btMember(E, bt(E1, L, _R)):- E<E1, btMember(E, L).
btMember(E, bt(E1, _L, R)):- E>E1, btMember(E, R).
```


Binary Search Trees (cont.)

- To insertion a new element into the BST, we implement btInsert(E,T,T1), where T1 is the new tree after the key E is added to T.

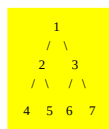
```
btInsert(E, nil, bt(E, nil, nil)).
btInsert(E, bt(E, L, R), bt(E, L, R)).
btInsert(E, bt(E1, L, R), bt(E1, L1, R)):-
   btInsert(E, L, L1), E < E1.
btInsert(E, bt(E1, L, R), bt(E1, L, R1)):-
   btInsert(E, R, R1), E > E1.

test(T4):- btInsert(7, nil, T), btInsert(5, T, T1),
           btInsert(9, T1, T3), btInsert(3, T3, T4).

?- test(T), btMember(5,T).
```


Traversing a Binary Tree

- pre-order - visit the node,
  then visit the left subtree,
  then visit the right subtree
  [1, 2, 4, 5, 3, 6, 7]
- in-order - visit the left subtree,
  then visit the node,
  visit the right subtree
  [4, 2, 5, 1, 6, 3, 7]
- post-order - visit the left subtree,
  then visit the right subtree,
  then visit the node
  [4, 5, 2, 6, 7, 3, 1]



Tree Example - Height-balanced BT

- The height of its left subtree and the height of its right subtree are at most equal, which means their difference is not greater than one.
- Write a predicate `hbal_tree/2` to construct height-balanced binary trees for a given height. The predicate should generate all solutions via backtracking. Put the letter 'x' as information into all nodes of the tree.

- Example:

```
?- hbal_tree(3,T).

T = t(x, t(x, t(x, nil, nil), t(x, nil, nil)), t(x, t(x, nil, nil), t(x, nil, nil))) ;
T = t(x, t(x, t(x, nil, nil), t(x, nil, nil)), t(x, t(x, nil, nil), nil)) ;
etc.....
No
```


Tree Example - Height-balanced BT

```
hbal_tree(0,nil) :- !.
hbal_tree(1,t(x,nil,nil)) :- !.
hbal_tree(D,t(x,L,R)) :-
   D > 1, D1 is D - 1, D2 is D - 2,
   distr(D1,D2,DL,DR), hbal_tree(DL,L),
   hbal_tree(DR,R).

distr(D1,_,D1,D1).
distr(D1,D2,D1,D2).
distr(D1,D2,D2,D1).
```

Stacks

- A stack is a list with a restriction on accessing the elements: the last element added to the list must be the first element to come out of the list.
  - when we add an element, we **push** it onto the stack
  - when we remove an element, we **pop** it off the stack
- Exercise: write the following predicates:
  - new_stack(-NewStack) - creates an empty stack
  - pop(?Element, +OldStack, -NewStack) - pops one element off the stack
  - push(+Element, +OldStack, -NewStack) - pushes one element onto the stack

Stacks

```
?- new_stack(S1), push(5, S1, S2), push(7, S2, S3), push(2, S3, S4),
    push(9, S4, S5), pop(S5, E1, S6), pop(S6, E2, S7), pop(S7, E3, S8),
    pop(S8, E4, S9).
S1 = stack([])
S2 = stack([5])
S3 = stack([7,5])
S4 = stack([2,7,5])
S5 = stack([9,2,7,5])
E1 = 9
S6 = stack([2,7,5])
E2 = 2
S7 = stack([7,5])
E3 = 7
S8 = stack([5])
E4 = 5
S9 = stack([])
yes
```


Stacks

- The implementation of stacks in Prolog is trivial: lists in Prolog can be used to represent stacks directly.

```
new_stack(stack([])).
push(Elem, stack(Stack), stack([Elem | Stack])).
pop(stack([Top | Stack]), Top, stack(Stack)).
```


Queues

- A queue is a list in which the first element added to the list must be the first element that comes out of the list.
  - when we add an element, we **enqueue** it
  - when we remove an element we **dequeue** it
  - we can check the element at the head of the queue
- we can test if the queue is empty
- Exercise: implement the following predicates:
  - new_queue(-NewQueue) - creates an empty queue
  - enqueue(+Element,+OldQueue, -NewQueue) - enqueues an element
  - dequeue(-Element, +OldQueue, -NewQueue) - dequeues an element
  - head_of_queue(-Head, +Queue) - check the head of a queue
  - empty_queue(+Queue) - checks if a queue is empty

Queues

```
?- new_queue(Q1), enq(5, Q1, Q2), enq(7, Q2, Q3), enq(2, Q3,
    Q4), enq(9, Q4, Q5), headq(Q5, H), deq(Q5, E1, Q6),
    deq(Q6, E2, Q7), deq(Q7, E3, Q8), deq(Q8, E4, Q9),
    emptyq(Q9).
Q1 = q([])
Q2 = q([5])
Q3 = q([5,7])
Q4 = q([5,7,2])
Q5 = q([5,7,2,9])
H = 5
E1 = 5
Q6 = q([7,2,9])
E2 = 7
Q7 = q([2,9])
E3 = 2
Q8 = q([9])
E4 = 9
Q9 = q([])
```


Queues

- Implementing queues is insignificantly more complicated than stacks.

```
new_queue(q([])).
enq(Elem, q(Queue), q(Queue2)) :-
        append(Queue, [Elem], Queue2).
deq(q([Head | Queue]), Head, q(Queue)).
headq(q([Head | Q]), Head).
emptyq(q([])).
```

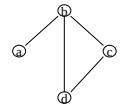
Graphs

- A graph is defined by:
  - set of **nodes** - could be a simple list
  - set of **edges** (*arcs*) - the form of representing the edges depends on the type of graph (**directed**, **not directed**, the arcs have *costs* attached or not, etc).

Representing a Graph

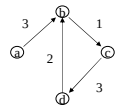
- Represent each edge as separate facts:

```
connected(a,b).
connected(b,c).
connected(d,b).
connected(d,c).
```



- directed graph:

```
arc( a, b, 3).
arc( b, c, 1).
arc( d, b, 2).
arc( c, d, 3).
```



Representing a Graph

- Represent the whole graph as one data object. We can represent it as a pair of two sets - one of nodes, one of edges:

```
G1 = graph( [a,b,c,d], [ e(a,b), e(b,c), e(d,c), e(b,d)]).
G2 = graph( [a,b,c,d], [ e(a,b,3), e(b,c,1), e(c,d,3), e(d,b,2)]).
```

- Represent the graph by associating to each of its nodes a list of neighbors:

```
G1 = [ [a, [b]], [b, [a,c,d]], [c, [b,d]], [d, [b,c]] ]
G2 = [ [a, [b/3]], [b, [c/1]], [c, [d/3]], [d, [b/2]]
```

etc.

Finding a Path Through a Graph

- G - graph; A, Z - two nodes in the graph; P - path between A and Z

```
find_path(+A,+Z,+G, -P)
```

to find an *acyclic* path P, in G, between A and G:

If A = Z then P is [A]

otherwise find an acyclic path P1, from some node Y to Z, and find a path from A to Y, avoiding the nodes in P1.

Finding a Path Through a Graph

```
path(A,Z,Graph,Path) :-
    path1(A,[Z],Graph,Path).

path1(A,[A|Path1],_,[A|Path1]).
path1(A,[Y|Path1],Graph,Path):-
     adjacent(X,Y,Graph),
     not(member(X,Path1)),          % No-Cycle condition
     path1(A,[X,Y|Path1],Graph,Path).

adjacent(X,Y,graph(Nodes,Edges)):-
    member(e(X,Y),Edges); member(e(Y,X),Edges).
```


Finding a Path Through a Graph

- **Result:**

```
?- G1=graph([a,b,c,d],[e(a,b),e(b,c),e(d,c),e(b,d)]),
   path(a,d,G1,P).

G1= graph([a,b,c,d],[e(a,b),e(b,c),e(d,c),e(b,d)])
P=[a,b,d]
```