Principles of Programming Languages
Lecture 7

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig
Concepts of PL by Sebesta
Prog. in Prolog by Clocksin and Mellish
PL Concepts and Constructs by Sethi
ML for the Working Prog. By Paulson
PL Pragmatics by Scott

Prolog: structures – example 2

- Family database – cont'd:

```
family(
person(tom,fox,date(7,may,1950),works(cbc,15200)),
person(ann,fox,date(9,may,1951), works(ctv,25700)),
[person(pat,fox,date(5,may,1973),unemployed),
 person(jim,fox,date(5,may,1973),unemployed)]).

% Let us add more useful rules
husband(X) :- family(X,_,_).
wife(X)      :- family(_,X,_).
child(X)     :- family(_,_,Children),
                member(X,Children).

exists(Person) :-husband(Person);
                 wife(Person);child(Person).

salary(person(_,_,_,works(_,S)),S).
salary(person(_,_,_,unemployed),0).

dateofbirth(person(_,_,Date,_),Date).
```

```
% Find the names of all the people in database
| ?- exists(person(Name,Surname,_,_)).
Name = tom
Surname = fox;
Name = ann
Surname = fox;
Name = pat
Surname = fox;
Name = jim
Surname = fox;
no

% Find all children born in 1973
| ?- child(X),dateofbirth(X,date(_,_,1973)).
X = person(pat,fox,date(5,may,1973),
     unemployed);
X = person(jim,fox,date(5,may,1973),
     unemployed);
no
```

Prolog: complex types - structures

- **Recall: what's a function term?**

  functor(some-parameters) e.g. woman(marry)

- **We can construct complex data structures using nested *function terms* and *lists.***
  - Represents a statement about the world

- **Example 2:**
  - A family consist of 2 persons, and 0 or more children. Each person is either employed for some salary or unemployed.

```
family(person(First-name,Last-name,date(Day,Month,Year),works(Company,Salary)),
       person(First-name,Last-name,date(Day,Month,Year),works(Company,Salary)),
       [person(First-name,Last-name,date(Day,Month,Year),unemployed),
        person(First-name,Last-name,date(Day,Month,Year),unemployed)]).
```

Prolog: structures – example 2

- Family database – cont'd:

```
% To find the names of unemployed people who were born before 1975
| ?- exists(person(Name,Surname,date(_,_,Year),unemployed)),Year < 1975.
Name = pat
Surname = fox
Year = 1973;
Name = jim
Surname = fox
Year = 1973;
no

% To find people born before 1951 whose salary is less than 80000
| ?- exists(Person),dateofbirth(Person,date(_,_,Year)),Year<1951,
salary(Person,Salary), Salary<80000.
Person = person(tom,fox,date(7,may,1950),works(cbc,15200))
Year = 1950
Salary = 15200;
no
```

Prolog: structures – example 2

- **Family database – cont'd:**

```
% A rule to add the salaries
total([],0).
total([Person|Tail],Sum):- salary(Person,S),total(Tail,Rest),Sum is S + Rest.

% To find the total income of family
| ?- family(Husband,Wife,Children),total([Husband,Wife|Children],Income).
Husband = person(tom,fox,date(7,may,1950),works(cbc,15200))
Wife = person(ann,fox,date(9,may,1951),works(ctv,25700))
Children = [person(pat,fox,date(5,may,1973),unemployed),
            person(jim,fox,date(5,may,1973),unemployed)]
Income = 40900;


% Now, let us define how to get the nth child. Note: I left children clause for you to define
| ?- nthchild(N,Family,Child) :- children(Family,ChildList),nth_member(N,ChildList,Child).
```

Prolog: fail & true predicates

- **That something is not true can be said in Prolog by using a special goal: fail , which *always and immediately* fails. That something is true can be said in Prolog by using the goal true, which *always and immediately* succeeds.**

- **The fail/true are inserted between goals as a pseudo-goal:**

```
head1(X) :- goal1a, goal1b, fail, goal1c.        %rule 1
head2(X) :- true.                                %rule 2
```

Prolog: fail & true predicates

- **Examples:**
  - How to represent "*Mary likes all animals but snakes*"?

```
% If X is a snake then Mary likes X is not ture
% otherwise if X is an animal then Mary likes X
snake(cobra).
cat(persian).
animal(X):-snake(X);cat(X).
likes(mary,X):-snake(X),fail.   % rule 1
likes(mary,X):-animal(X).       % rule 2
! ?- likes(mary,cobra).         % query
yes
% Why did we get a wrong answer?
%  rule 1 failed because of fail but
% Prolog interpreter went on to unify
% with rule 2 which succeeded so we
% got yes
```

```
snake(cobra).
cat(persian).
animal(X):-snake(X);cat(X).
likes(mary,X):-snake(X),!,fail. % rule 1
likes(mary,X):-animal(X).       % rule 2
! ?- likes(mary,cobra).         % query
no
%Why did we get the right answer?
% rule 1 failed because of fail and the !
% told the interpreter not to try rule 2
% because since we have reached
% there(aka to the !), it must be a snake
```

Prolog: fail & true predicates

- **Examples – cont'd:**

  – How to represent "*not(Goal) relation*"?

```
% if Goal succeeds then not(Goal) fails,
% otherwise not(Goal) succeeds
not(P) :- P,!,fail.
not(P) :- true.
```

Prolog: dynamic programming

- **What is it?**

- **Adding/removing rules:**
  - assert(C). % adds a clause to the database
  - asserta(C). % adds a clause to the beginning of the database
  - assertz(C). % adds a clause to the end of the database
  - retract(C). % removes a clause from the database

- **Examples:**

```
| ?- crisis.
no
| ?- assert(crisis).
yes
| ?- crisis.
yes
| ?- retract(crisis).
yes
| ?- crisis.
no
```

```
fast(ann).  slow(tom).   slow(pat).
| ?- assert( (faster(X,Y) :- fast(X), slow(Y) ) ).
yes
| ?- faster(A,B).
….
| ?- retract( slow(X) ).
 X = tom;
 X = pat;
no
faster( ann,_).
no
```

Prolog: dynamic programming – cont'd

- **Useful applications of *assert* (*improving efficiency*):**
  - Caching a single solution:

    ```
    solve(Problem,Solution) :- …..
    | ?- solve(problem1, Solution), asserta(solve (problem1,Solution) ).
    ```

  - Caching a table of solutions, *for example*:

    ```
    maketable :- L = [0,1,2,3,4,5,6,7,8,9],member(X,L),member(Y,L),Z is X * Y,
                 asserta(product(X,Y,Z)).
    | ?- maketable.                    % create product table and add it to the top
    yes
    | ?- product(A,B,8).                % query to find all A's and B's whose product is 8
    A = 1   B = 8;          A = 2   B = 4;
    A = 4   B = 2;          A = 8   B = 1;
    no
    ```

- **Excessive/careless use of assert/retract yields programs that are hard to read:**
  - Relations that hold true at one point will not hold true at some other time
  - At different times, the same question receive different answers

Prolog: input/output

- **Reading from input:**
  - Read a term from the user `read(X).`

- **Writing to output:**
  - output the term X `write(X).`
  - New line `nl.`
  - Output N spaces `tab(N).`
  - Outputting a list:

```
writelist([]).
writelist([X|L]) :- write(X), nl, writelist(L).
```

```
| ?- writelist([12,14]).
12
14
```

  - Outputting a sequence of characters as *

```
bars([]).
bars([N|L]) :- stars(N), nl, bars(L).
stars(N) :- N > 0, write(*), N1 is N-1, stars(N1).
stars(N):- N =< 0.
```

```
| ?- bars([3,4,6,5]).
***
****
******
*****
```

Prolog: input/output

- **Interactive program example:**

```
cube :- write('Next item, please: '),read(X),process(X).
process(stop):- !.
process(N) :- C is N * N * N, write('Cube of '), write(N),
                write('is '),write(C),nl,cube.
| ?- cube.
Next item, please: 5.
Cube of 5 is 125
Next item, please: 10.
Cube of 10 is 1000
```

Prolog: pros & cons

- **Cons:**
  - Horn clauses have limited expressive power
  - Closed world assumptions *(anything not mentioned is false)*
  - Ordering of clauses change the result
  - Because horn clause is the basic construct, you must program carefully to avoid *infinite loops*
    - There is no 1-solution-fits-all for these problems…

- **Pros:**
  - Pattern matching
  - Backtracking
  - Unification
  - Rules and goals are also data (*dynamic programming*).
  - The logical model is powerful

Using Prolog with C/C++

- Prolog to C++ bridge:
  http://www.swi-prolog.org/pldoc/package/pl2cpp.html

- Prolog to Java bridge:
  http://www.gnu.org/software/gnuprologjava/

Java & Javascript

Imperative Languages (e.g. C/C++)

Source

```
int main() {
  int nIndex,nSum;
  for( nIndex=0; nIndex<10;nIndex++)
      nSum =+ 2 * nIndex;
}
```

Assembly

```
 .file   "foo.c"
                .text
                .p2align 4,,15
.globl main
                .type   main, @function
main:           push    BP
                mov     $9, AX
                mov     SP, BP
                sub     $8, SP
                and     $-16, SP
                .p2align 4,,15
.L6:            dec     AX
                jns     .L6
                mov     BP, SP
                pop     BP
                ret
                .size   main, .-main
                .ident  "GCC: (GNU) 3.3.1"
```

Binary

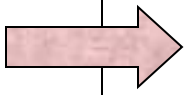
```
01010101010001
10101010101111
10101001010101
10010101001000
00000001101111
00000000000000
11111111100001
```

gcc -O2 -S -c foo.c

Java

Source

```
int main() {
  int nIndex,nSum;
  for( nIndex=0; nIndex<10;nIndex++)
      nSum =+ 2 * nIndex;
}
```



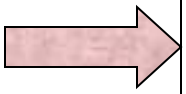
Machine Independent Assembly (bytecode)

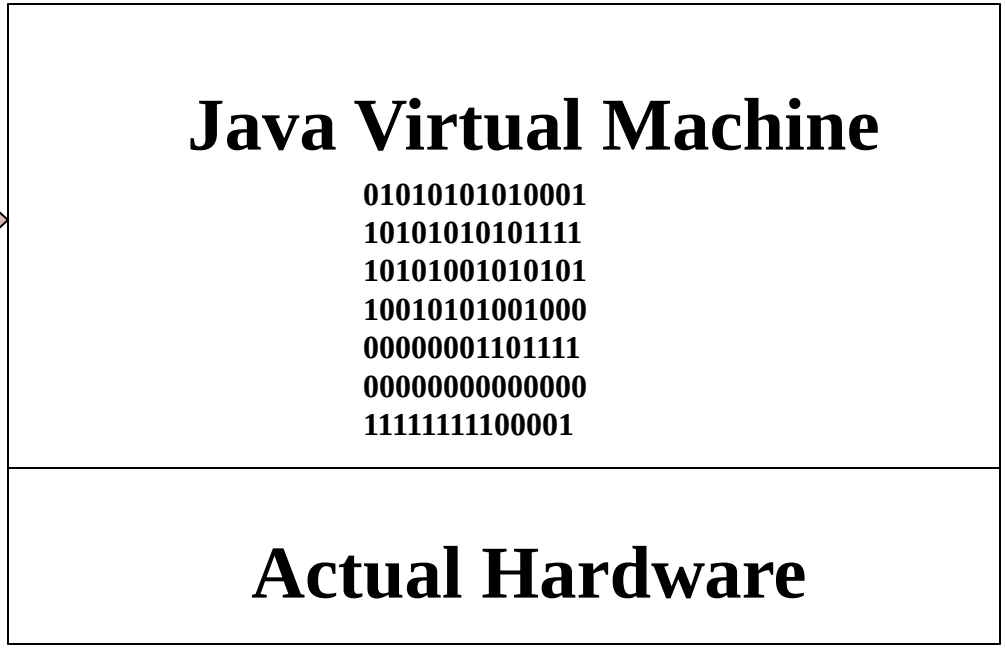
```
.file   "foo.c"
                .text
                .p2align 4,,15
.globl main
                .type   main, @function
main:           push   BP
                mov    $9, AX
                mov    SP, BP
                sub    $8, SP
                and    $-16, SP
                .p2align 4,,15
.L6:            dec    AX
                jns     .L6
                mov     BP, SP
                pop    BP
                ret
                .size   main, .-main
                .ident  "GCC: (GNU) 3.3.1"
```

javac foo.java

Java

Machine Independent Assembly (bytecode)





java foo.class

Java

Java Virtual Machine

01010101010001
10101010101111
10101001010101
10010101001000
00000001101111
00000000000000
11111111100001

Actual Hardware

Java Virtual Machine

01010101010001
10101010101111
10101001010101
10010101001000
00000001101111
00000000000000
11111111100001

Actual Hardware

Java Virtual Machine

01010101010001
10101010101111
10101001010101
10010101001000
00000001101111
00000000000000
11111111100001

Actual Hardware

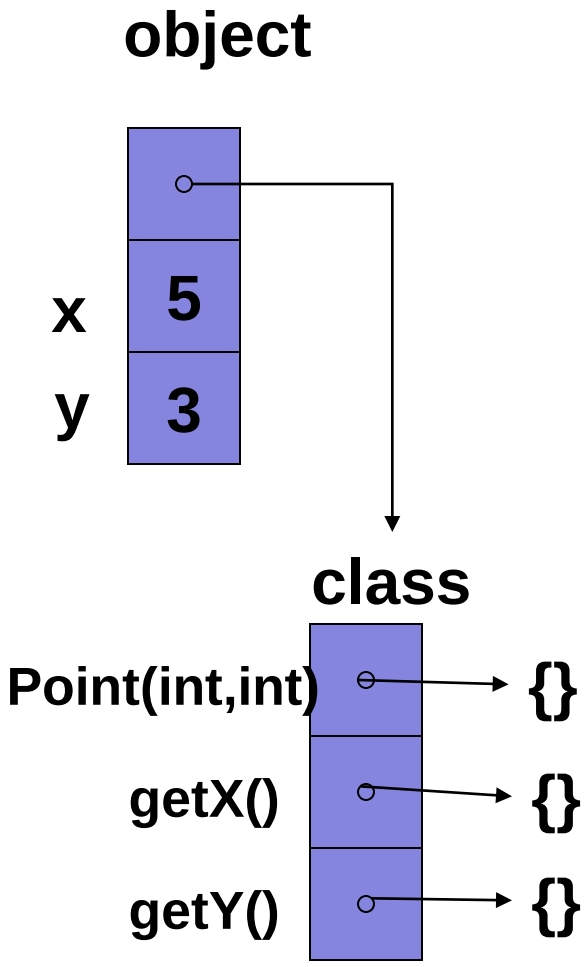
java foo.class

Background

- Turing's great insight: programs are just another kind of data
  - Source code is text
  - Manipulate it line by line, or by parsing expressions
- Compiled programs are data, too
  - Integers and strings are bytes in memory that you interpret a certain way
  - Instructions in methods are just bytes too
- No reason why a program can't inspect itself

How Reflection Works

```
class Point {
  public Point(int x, int y) {
    x = 5; Y = 10;
  }
  public int getX() {
        return x;
  }
  public int getY() {
        return y;
    }
  protected int x, y;
}
```



The class Class

- Instances of the class Class store information about classes
  - Class name
  - Inheritance
  - Interfaces implemented
  - Methods, members, etc.

- Can look up instances:
  - By name
  - From an object

Showing a type

```
public static void showType(PrintStream out,
                                                String className)
            throws ClassNotFoundException
{
  Class thisClass = Class.forName(className);
  String flavour    =  thisClass.isInterface() ? "interface" : "class";
  out.println(flavour + " " + className);
  Class parentClass = thisClass.getSuperclass();
  if (parentClass != null) {
    out.println(" extends " + parentClass.getName());
  }
  Class[] interfaces = thisClass.getInterfaces();
  for (Class interf : interfaces) {
    out.println("  implements " + interf.getName());
  }
}
```

Output for type example

class java.lang.Object

class java.util.HashMap
 extends java.util.AbstractMap
  implements java.util.Map
  implements java.lang.Cloneable
  implements java.io.Serializable

class Point
 extends java.lang.Object

Examining class contents

```
public static void showContents(PrintStream out,
                                boolean hideObject,
                                String name)
    throws ClassNotFoundException {

  Class cls = Class.forName(name);

  out.println(name);

  showMembers(out, hideObject, name + " fields",    cls.getFields());

  showMembers(out, hideObject, name + " constructors",cls.getConstructors());

  showMembers(out, hideObject, name + " methods",     cls.getMethods());
}
```

Examining class contents

```
public static void showMembers( PrintStream out,
                                boolean hideObject,
                                String title,
                                Member[] members) {
  out.println("  " + title);
  for (Member mem : members) {
    if (mem.getDeclaringClass() == Object.class) {
      if (hideObject) {
        continue;
      }
    }
    out.println("\t" + mem);
  }
}
```

Point *(somewhat edited)*

Point fields

Point constructors

public Point(java.lang.String,int,int)

public Point(int,int)

Point methods

public java.lang.String Point.toString()

public java.lang.String Point.getName()

public void Point.setName(java.lang.String)

public int Point.getX()

public void Point.setX(int)

public int Point.getY()

public void Point.setY(int)