# An Introduction (I) to SML, Java & Prolog



# Prolog



## Logic-Oriented Style

- Relations
- Solving problems with relations!
- There are a number of dialects
  - SWI-Prolog



## Terms

- Everything in Prolog is built from *terms*.
- Three kinds of terms:
  - Constants: integers, real numbers, atoms
  - Variables
  - Compound terms



## Term: Constants

- Integer constants: `123`
- Real constants: `1.23`
- Atoms:
  - A **lowercase letter** followed by any number of additional letters, digits or underscores: `name`
  - But an atom is not a variable; more like a string constant.



## Term: Variables

- Any name beginning with an **uppercase letter** or an underscore, followed by any number of additional letters, digits or underscores: X, Parent, Child
- Most of the variables you write will start with an uppercase letter.
  - Those starting with an underscore, including _, get special treatment.

## Term: Compound Terms

- An atom followed by a parenthesized, comma-separated list of one or more terms:
  - `x(y,z)`, `+(1,2)`, `.(1,[])`, `parent(adam,seth)`, `x(Y,x(Y,Z))`
- A compound term is not a function call: as structured data



## Syntax of Terms

*<term>* `::=` *<constant>*
`|` *<variable>*
`|` *<compound-term>*

*<constant>* `::=` *<integer>*
`|` *<real number>*
`|` *<atom>*

*<compound-term>* `::=` *<atom>* `(` *<termlist>* `)`



## Facts

*<fact>* `::=` *<term>* `.`

- The simplest kind of thing in the database is a *fact*: a term followed by a period.
- `Parent`(X,Y) = X is the parent of Y.

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).
```



## Rules

*<rule>* `::=` *<term>* `:-` *<termlist>* `.`

*<termlist>* `::=` *<term>* `|` *<term>* `,` *<termlist>*

*head*

```
greatgrandparent(GGP,GGC)
:-
    parent(GGP,GP),
    parent(GP,P),
    parent(P,GGC).
```

*conditions*



## Rules

*head*

```
greatgrandparent(GGP,GGC) :-
    parent(GGP,GP),
    parent(GP,P),
    parent(P,GGC).
```

*conditions*

- A rule says how to prove something: to prove the head, prove the conditions/



## A Prolog Program

- A program consists of a list of *clauses*

*<clause>* `::=` *<fact>*
`|` *<rule>*

## A Prolog Program

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).

greatgrandparent(GGP,GGC) :-
  parent(GGP,GP), parent(GP,P),
  parent(P,GGC).
```



## Example: A Prolog Program

```
% COMMENTS parent relation

parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).
```

- **parent**(X,Y) = X is the parent of Y.



## SWI-Prolog Environment

- Prompting for a query with **?-**



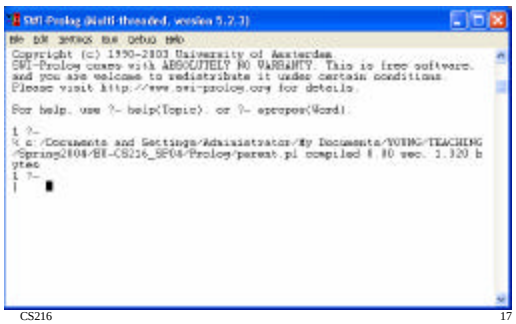
## The consult Predicate

- Predefined predicate to read a program from a file into the database.
- File relations (or **relations.pl**) contains the parent facts.

```
?- consult(relations).
% relations compiled 0.00 sec, 0 bytes

Yes
?-
```



## The consult Predicate





## Prolog Queries

```
?- parent(margaret,kent).

Yes
?- parent(fred,pebbles).

No
?- parent(margaret,kent)
|    .

Yes
?-
```

## Prolog Queries

```
?- parent(P,jean).

P = herbert

Yes
?- parent(P,esther).

No
```



## Queries With Variables



## Conjunction Queries

```
?- parent(margaret,X), parent(X,holly).

X = kim

Yes
```



## Multiple Solutions

```
?- parent(margaret,Child).

Child = kim ;

Child = kent ;

No
```

- By typing **;** rather than Enter, you ask the Prolog system to find more!



## Multiple Solutions

```
?- parent(Parent,kim),
parent(Grandparent,Parent).

Parent = margaret
Grandparent = esther ;

Parent = margaret
Grandparent = herbert ;

No
```



## Unification

- Pattern-matching using Prolog terms
- Two terms unify if there is some way of binding their variables that makes them identical

## Example

```
parent(kim,holly).
parent(margaret,kim).
parent(margaret,kent).
parent(esther,margaret).
parent(herbert,margaret).
parent(herbert,jean).

greatgrandparent(GGP,GGC) :-
  parent(GGP,GP), parent(GP,P), parent(P,GGC).
```

- **greatgrandparent**(X,Y) = X is the greatgrandparent of Y.



## Query

```
?-
greatgrandparent(esther,GreatGrandchild).

GreatGrandchild = holly

Yes
```



## Rules Using Other Rules

```
grandparent(GP,GC) :-
  parent(GP,P), parent(P,GC).

greatgrandparent(GGP,GGC) :-
  grandparent(GGP,P),
parent(P,GGC).
```

- **greatparent**(X,Y) = X is the greatparent of Y.



## Recursive Rules

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

- **X** is an ancestor of **Y** if:
  - Base case: **X** is a parent of **Y**
  - Recursive case: there is some **Z** such that **Z** is a parent of **Y**, and **X** is an ancestor of **Z**



## The = Predicate

- The goal =(X,Y) succeeds if and only if X and Y can be unified:

```
?- =(parent(adam,seth),parent(adam,X)).

X = seth

Yes
?- parent(adam,seth)=parent(adam,X).

X = seth

Yes
```

- An operator is just a predicate for which a special abbreviated syntax is supported



## Arithmetic Operators

- Predicates **+**, **-**, ***** and **/** are operators

```
?- X = +(1,*(2,3)).

X = 1+2*3

Yes
?- X = 1+2*3.

X = 1+2*3

Yes
```

## Arithmetic Operators

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

X = 1
Y = 2*3

Yes
?- 1+2 = 2+1.

No
?- 7 = 1+2*3.

No
```



## The is Predicate

```
?- X is 1+2.

X = 3

Yes
?- 7 = 1+2*3.

No
```



## Example: The Factorial Predicate

```
factorial(0,1).
factorial(N, F) :- N >0, N1 is N-1,
factorial(N1,F1), F is N*F1.
?- factorial(0,X).
X = 1

Yes
?- factorial(4,X).
X = 24

Yes
```



## The not Predicate

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

Yes
?- not(member(4,[1,2,3])).

Yes
```

- not(X) succeeds if X fails



## Lists

```
?- X = [].
X = []

Yes
?- X = .(1,.(2,.(3,[]))).

X = [1, 2, 3]

Yes
?- X = [1,2,3].

X = [1, 2, 3]

Yes
```



## Lists

## List Notation With Tail

```
?- .(X,Y) = [1,2,3].

X = 1
Y = [2, 3]

Yes
?- [1,2|X] = [1,2,3,4,5].

X = [3, 4, 5]

Yes
```



## The append Predicate

- append(X,Y,Z) succeeds if and only if Z is the result of appending the list Y onto the end of the list X

```
append([], B, B).
append([Head|TailA], B, [Head|TailC])
:- append(TailA, B, TailC).

?- append([1,2],[3,4],Z).

Z = [1, 2, 3, 4]

Yes
```



## Predicate – More than A Function

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

X = [1, 2]

Yes
```

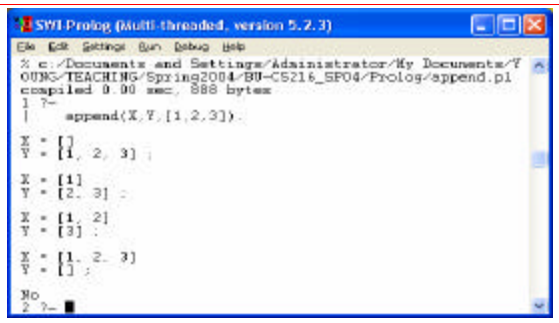
```
?- append(X,Y,[1,2,3]).

X = []
Y = [1, 2, 3] ;

X = [1]
Y = [2, 3] ;

X = [1, 2]
Y = [3] ;

X = [1, 2, 3]
Y = [] ;

No
```



## Predicate – More than A Function




## Predicate – More than A Function



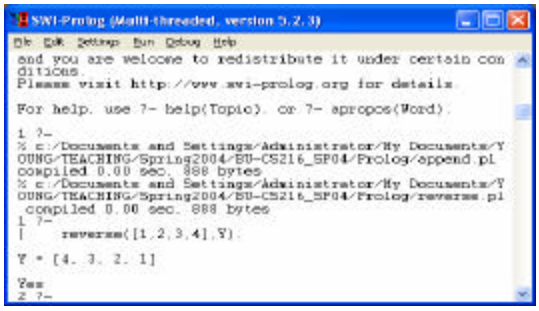



## The reverse Predicate

- Predefined reverse(X,Y) unifies Y with the reverse of the list X

```
reverse([],[]).
reverse([Head|Tail],X) :-
  reverse(Tail,Y),
  append(Y,[Head],X).

?- reverse([1,2,3,4],Y).

Y = [4, 3, 2, 1] ;

No
```

## The reverse Predicate






## The Corresponding Program in SML

**Java**

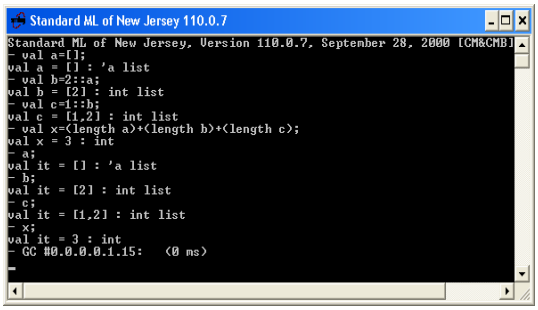
```
class Driver {
    public static void main(String[]
  args) {
      IntList a = new IntList(null);
      IntList b = a.cons(2);
      IntList c = b.cons(1);
      int x = a.length() +
              b.length()+
              c.length();
      a.print();
      b.print();
      c.print();
      System.out.println(x);
    }
  }
```

**SML**

```
val a=[];
val b=2::a;
val c=1::b;
val x=(length a)+
      (length b)+
      (length c);
a;
b;
c;
x;
```



## The Corresponding Program in SML






## The Corresponding Program in Prolog

**Java**

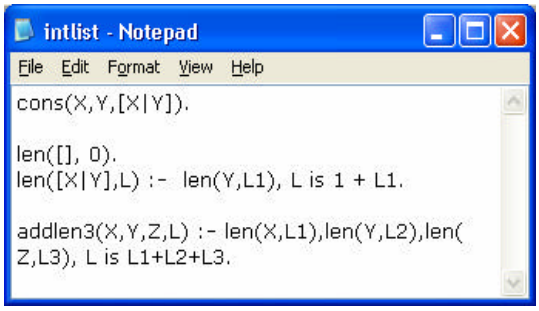
```
class Driver {
    public static void main(String[]
  args) {
      IntList a = new IntList(null);
      IntList b = a.cons(2);
      IntList c = b.cons(1);
      int x = a.length() +
              b.length()+
              c.length();
      a.print();
      b.print();
      c.print();
      System.out.println(x);
    }
  }
```

**Prolog**

?



## The Corresponding Program in Prolog






## The Corresponding Program in Prolog

## The Corresponding Program in Prolog