IKI30820 – Logic Programming
Lists, Operators, Arithmetics
Slide 04

Ari Saptawijaya (slides) Adila A. Krisnadhi (LaTeXadaptation)

Fakultas Ilmu Komputer
Universitas Indonesia

2009/2010 • Semester Gasal

Outline

Representation of Lists I

- List is a simple data structure widely used in programming
- A list is a **sequence** of **any** number of items
- A list in Prolog: `[java,c,haskell,prolog]`
  - External appearance
  - It is a *structured* object, represented as a tree
- List is either empty or non-empty
- Empty list is written as Prolog **atom**: `[]`
- Non-empty list can be viewed as consisting of two things:
  - The first item, called the **head** of the list
  - The remaining part of the list, called the **tail**

Representation of Lists II

- For the list: `[java,c,haskell,prolog]`
  - Head of the list: `java`
  - Tail of the list: `[c,haskell,prolog]`
- Head can be any Prolog object, but the tail has to be a list
- This is also a list: [[a,b],c,d]
  - Head: `[a,b]`
  - Tail: `[c,d]`
- What is the head and the tail of `[[a,b,c]]`?

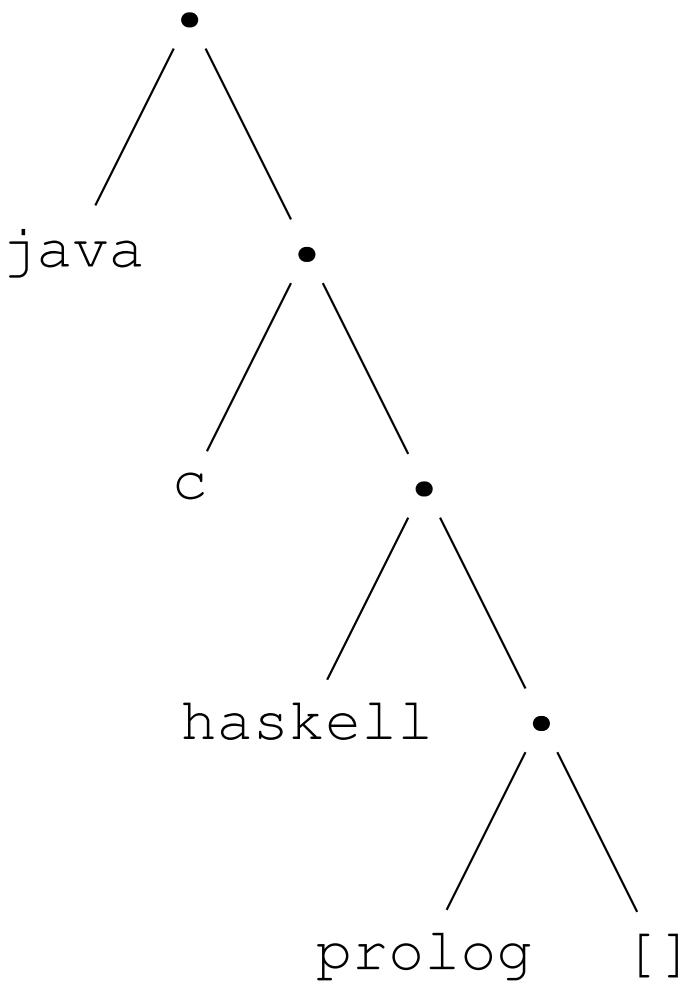
The Functor "." I

- List is a structured object
  - The head and the tail are combined into a structure by a special functor "."
  - A list is represented by: `.(Head,Tail)`
- Tail is in turn a list: either empty or it has its own head and tail
- Example: `[java,c,haskell,prolog]` is actually represented as the term:

  `.(java,.(c,.(haskell,.(prolog,[]))))`

The Functor "." II

- The corresponding tree structure for
  `.(java,.(c,.(haskell,.(prolog,[]))))`



List notation I

- The square bracket notation for list is obviously more convenient.
  - But, internally lists are represented as binary trees
- Prolog always displays a list in square bracket notation

```
?- List = .(a,.(b,.(c,[]))).

?- is_list([a.b]).
```

List notation II

- Prolog provides another notational extension, the vertical bar, which separates the head and the tail
  - We can have one or more elements followed by "`|`" and the list of remaining items
- Alternative ways of writing `[a,b,c]` are
  - `[a,|[b,c]]`
  - `[a,b|[c]]`
  - `[a,b,c|[ ]]`

```
?- [Head|Tail] = [a,b,c].
```

Built-in predicates for lists: membership I

- Membership

  `member(X,L)`

  is true when the object X is a member of the list L

```
?- member(b,[a,b,c]).

?- member(b,[a,[b,c]]).

?- member([b,c],[a,[b,c]]).
```

Built-in predicates for lists: membership II

- The program for membership relation can be based on the following observation:
  X is a member of L if either:
  1. X is the head of L, or
  2. X is the member of the tail of L
- This can be written in two clauses; the first is a simple fact and the second is a rule

```
member(X,[X|Tail]).
member(X,[Head|Tail]) :- member(X,Tail).
```

Built-in predicates for lists: list concatenation I

- `append(L1,L2,L3)` is true if the list L3 is the concatenation of the list L1 and the list L2

```
?- append([a,b],[b,c,d],[a,b,b,c,d]).

?- append([a,b],[b,c,d],[a,b,c,d]).

?- append([a,b],[1,2,3],L).
```

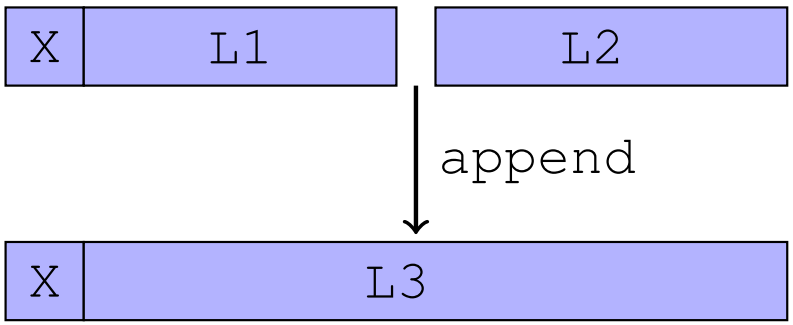
Built-in predicates for lists: list concatenation II

- The definition of append, depending on the first argument, is as follows:
  1. If the first argument is the empty list, then the second and the third must be the same list

     ```
     append([],L,L).
     ```

  2. If the first argument is a non-empty list, then it has a head and a tail of the form [X | L1]. The concatenation of [X | L1] and L2 is the list [X | L3], where L3 is the concatenation of L1 and L2. (See picture)

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



Give the answers of this query

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

?- append(L1,L2,[a,b,c]).
```

List membership using `append`

- The membership relation can be programmed using `append`:

List membership using `append`

- The membership relation can be programmed using `append`:
  - X is a member of list L, if L can be decomposed into two lists so that the second one has X as its head

    ```
    member1(X,L) :- append(L1,[X|L2],L).
    ```

  - Can you write the above rule better to avoid a warning from the Prolog system?

List membership using `append`

- The membership relation can be programmed using `append`:
  - X is a member of list L, if L can be decomposed into two lists so that the second one has X as its head

    ```
    member1(X,L) :- append(L1,[X|L2],L).
    ```

  - Can you write the above rule better to avoid a warning from the Prolog system?
- What about this rule?

```
member1(X,L) :- append([X|L1],L2,L).
```

List membership using `append`

- The membership relation can be programmed using `append`:
  - X is a member of list L, if L can be decomposed into two lists so that the second one has X as its head

```
member1(X,L) :- append(L1,[X|L2],L).
```

  - Can you write the above rule better to avoid a warning from the Prolog system?
- What about this rule?

```
member1(X,L) :- append([X|L1],L2,L).
```

- Try with both versions of `member1`:

```
?- member1(a,[a,b,c]).

?- member1(b,[a,b,c]).
```

Example: Permutation

`permute(L1,L2)` is true if the list L2 is a permutation of the list L1.

Example: Permutation

`permute(L1,L2)` is true if the list L2 is a permutation of the list L1.

Based on two considerations, depending on the first list

1. If the first list is empty, then the second list is also empty.

Example: Permutation

`permute(L1,L2)` is true if the list L2 is a permutation of the list L1.

Based on two considerations, depending on the first list

1. If the first list is empty, then the second list is also empty.
2. If the first list is not empty, then it is of the form [X | L] and a permutation L2 of such list is obtained:
   - if T1 is a permutation of L; and
   - L2 is a list resulting from inserting X to any position in T1.

Example: Permutation

`permute(L1,L2)` is true if the list L2 is a permutation of the list L1.

Based on two considerations, depending on the first list

1. If the first list is empty, then the second list is also empty.
2. If the first list is not empty, then it is of the form [X | L] and a permutation L2 of such list is obtained:
   - if T1 is a permutation of L; and
   - L2 is a list resulting from inserting X to any position in T1.

How then you define `insert(X,T,L)` which is true if the list L is obtained by inserting X to a position in the list T?

```
del(X,[X|Tail],Tail).
del(X,[Y|Tail],[Y|T]) :- del(X,Tail,T).

insert(X,Y,Z) :- del(X,Z,Y).

permutation([],[]).
permutation([X|L],L2) :-
    permutation(L,T1),
    insert(X,T1,L).

?- permutation([a,b,c],[c,a,b]).

?- permutation([a,b,c],X).
```

Outline

Operators

- Operators are defined to improve the readability of source-code
- For example, without operators, to write `a*b+c*d`, one would have to write: `+(*(a,b),*(c,d))`
- A number of operators have been predefined. All operators, except for the comma (,) can be redefined by the user.

1200	xfx	-->, :-
1200	fx	:-, ?-
1150	fx	dynamic, discontiguous, initialization, module_transparent, multifile, thread_local, volatile
1100	xfy	;, |
1050	xfy	->, **op***->
1000	xfy	,
954	xfy	\
900	fy	\+
900	fx	~
700	xfx	<, =, =.., =@=, =:=, =<, ==, =\=, >, >=, @<, @=<, @>, @>=, \=, \==, **is**
600	xfy	:
500	yfx	+, -, /\, \/, xor
500	fx	+, -, ?, \
400	yfx	*, /, //, rdiv, <<, >>, **mod**, rem
200	xfx	**
200	xfy	^

Defining new operators I

op(+Precedence, +Type, **:Name**)

Declares **Name** to be an operator of type `Type` with precedence `Precedence`

- **Name** can also be a list of names, in which case all elements of the list are declared to be identical operators
- `Precedence` is an integer between 0 and 1200. Precedence 0 removes the declaration. The higher precedence the lower the binding power of the operator.
- `Type` is one of: $x\,f$, $y\,f$, $x\,f\;x$, $x\,f\;y$, $y\,f\;x$, $f\;y$, or $f\;x$
- The 'f' indicates the position of the functor, while x and y indicate the position of the arguments (postfix: $x\,f$, $y\,f$, infix: $x\,f\;x$, $y\,f\;x$, $x\,f\;y$, prefix: $f\;x$, $f\;y$)

Defining new operators II

- 'y' should be interpreted as "on this position a term with precedence lower or equal to the precedence of the functor should occur"
- For 'x' the precedence of the argument must be strictly lower than the precedence of the functor
- The precedence of a term is 0, unless its principal functor is an operator, in which case the precedence is the precedence of this operator
- A term enclosed in brackets (...) has precedence 0

Example

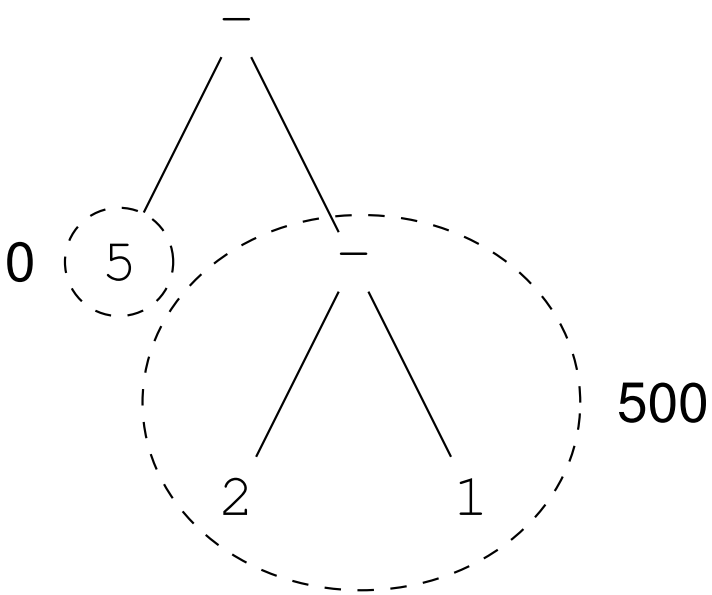
Using predefined operator '-', how would 5-2-1 be interpreted?

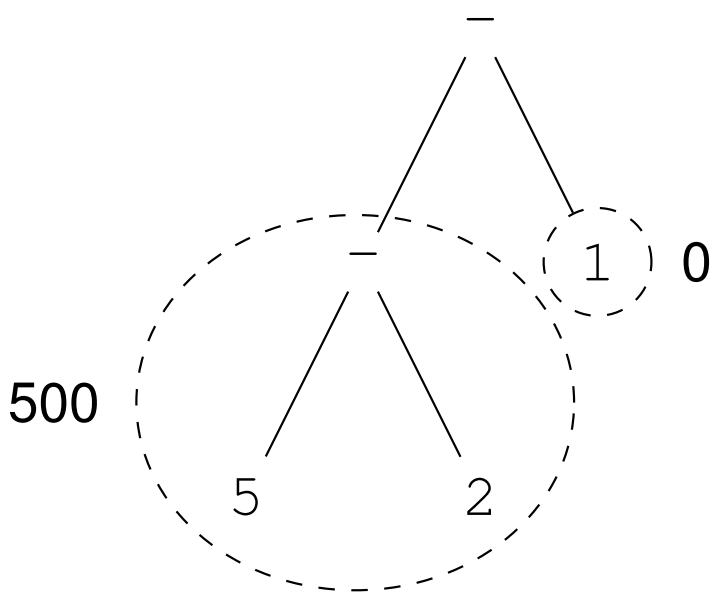
- Operator '-' is defined with precedence as 500 and type yfx.
- Would that be intepreted as 5-(2-1) or (5-2)-1?

Example

Using predefined operator '-', how would 5-2-1 be interpreted?

- Operator '-' is defined with precedence as 500 and type yfx.
- Would that be intepreted as 5-(2-1) or (5-2)-1?





- Programmer can define new operators byinserting into the program special kinds of clauses, called **directives**, starting with "`:-`"
- Note that operator definition do not specify any operation or action

```
:- op(650,xfx,suka).
unyil suka bakso.
iin suka tempe.
unyil suka mie.

budi suka X :- unyil suka X.
budi suka X :- X suka tempe.
```

Exercise

- Write an operator definition for "`dan`" so that we can write:

```
unyil suka bakso dan mie dan coklat
```

- Add a rule to the following program "suka":

```
unyil suka bakso.
iin suka tempe.
unyil suka mie.
unyil suka coklat.
```

so that we can ask queries like:

```
?- Siapa suka bakso dan mie.
Siapa = unyil
Yes

?- Siapa suka bakso dan coklat dan mie.
Siapa = unyil
Yes
```

`current_op(?Precedence, ?Type, ?:`**`Name`**`)`

succeeds when **`Name`** is currently defined as an operator of type `Type` with precedence `Precedence`.

```
?- current_op(P,T,suka).
P = 650
T = xfx
Yes

?- current_op(P,T,-).
P = 500
T = yfx
Yes
```

More example

- De Morgan's equivalence theorem:

$$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$$

- Can be expressed in Prolog term as:

```
equivalence(not(and(A,B)),or(not(A),not(B)))
```

- With a suitable set of operators

```
:- op(800, xfx, <===> ).
:- op(700, xfy, v).
:- op(600, xfy, &).
:- op(500, fy, ~).
```

- The formula can now be written as:

```
~(A & B) <===> ~A v ~B.
```

Outline

Arithmetic

- The general arithmetic predicates all handle expressions.
- An expression is either a simple number or a function. The arguments of a function are expressions.
- Note that '=' does not cause any arithmetic operation. Use '**is**' to activate arithmetic operation.

```
?- Y = 1+3.

?- Y is 1+3.
```

Predicate: `is`

`Number` **`is`** `+Expr`

- Succeeds when `Number` has successfully been unified with the number which `Expr` evaluates to.
- Note that normally, **`is`**/2 will be used with unbounded left operand. If equality is to be tested,=:=/2 should be used.

Arithmetic predicates

- `between(+Low, +High, ?Value)`
  `Low` and `High` are integers. `High` $\geq$ `Low`. If `Value` is an integer, `Low` $\leq$ `Value` $\leq$ `High`. When `Value` is a variable, it is successively bound to all integers between `Low` and `High`.
- `succ(?Int1, ?Int2)`
  succeeds if `Int2 = Int1 + 1`. At least one of the arguments must be instantiated to an integer
- `plus(?Int1, ?Int2, ?Int3)`
  succeeds if `Int3=Int1+Int2`. At least two of the three arguments must be instantiated to integers.

Other arithmetic predicates

- `Expr1 > Expr2`
  succeeds when expression `Expr1` evaluates to a larger number than `Expr2`
- `Expr1 < Expr2`
- `Expr1 =< Expr2`
- `Expr1 >= Expr2`
- `Expr1 =\= Expr2`
  succeeds when expression `Expr1` evaluates to a number non-equal to `Expr2`
- `Expr1 =:= Expr2`
  succeeds when expression `Expr1` evaluates to a number equal to `Expr2`

Arithmetic function

Arithmetic functions are terms which are evaluated by the arithmetic predicates, such as '**`is`**'

- `-Expr` minus
- `Expr1 + Expr2` addition
- `Expr1 - Expr2` subtraction
- `Expr1 * Expr2` multiplication
- `Expr1 / Expr2` division
- `Expr1 // Expr2` Integer division

Other functions I

- Power: `Expr1 ** Expr2` (the same as ^)
- `IntExpr1` **`mod`** `IntExpr2`
  Result: `IntExpr1-(IntExpr1//IntExpr2)*IntExpr2`
- Remainder of division: `IntExpr1 rem IntExpr2`
  Result: `float_fractional_part(IntExpr1/IntExpr2)`
- `exp(Expr)`. Result: e^{Expr}
- `abs(Expr)`, `sign(Expr)`, `max(Expr1,Expr2)`, `min(Expr1,Expr2)`
- `random(Int)`
  evaluates to random integer i for which $0 \leq i < Int$. The seed of this random generator is determined by the system clock when SWI- Prolog started
- `round(Expr)`, `float(Expr)`, `truncate(Expr)`, `floor(Expr)`, `ceiling(Expr)`, `sqrt(Expr)`

Other functions II

- `sin(Expr), cos(Expr), tan(Expr)`
  `Expr` is in radian
- `asin(Expr), acos(Expr), atan(Expr)`
  Result is the angle in radian
- `log(Expr)`
  Result: $\ln Expr$
- `log10(Expr)`
  Result: $\log Expr$
- `pi`
  evaluates to the mathematical constant π (3.14159)
- `e`
  evaluates to the mathematical constant e (2.71828)

Other functions III

- `IntExpr1 >> IntExpr2`
  Bitwise shift `IntExpr1` by `IntExpr2` bits to the right
- `IntExpr1 << IntExpr2`
  Bitwise shift `IntExpr1` by `IntExpr2` bits to the left
- `IntExpr1 \/ IntExpr2`
  Bitwise 'or' `IntExpr1` and `IntExpr2`
- `IntExpr1 /\ IntExpr2`
  Bitwise 'and' `IntExpr1` and `IntExpr2`
- `IntExpr1 xor IntExpr2`
  Bitwise 'exclusive or' `IntExpr1` and `IntExpr2`
- `\ IntExpr`
  Bitwise negation

Example

```
?- 1+3 =:= 3+1.

?- 1+3 == 3+1.

?- 1+A=B+3.

?- X is sin(pi/2).
```

Example: Finding GCD

Given two positive integers, X and Y, their GCD (greatest common divisor) D can be found according to three cases:

Example: Finding GCD

Given two positive integers, X and Y, their GCD (greatest common divisor) D can be found according to three cases:

1. If X and Y are equal, then D is equal to X
2. If X¡Y, then D is equal to the GCD of X and Y-X
3. If Y¡X, then do the same as in case (2) with X and Y interchanged

Example: Finding GCD

Given two positive integers, X and Y, their GCD (greatest common divisor) D can be found according to three cases:

1. If X and Y are equal, then D is equal to X
2. If X¡Y, then D is equal to the GCD of X and Y-X
3. If Y¡X, then do the same as in case (2) with X and Y interchanged

```
gcd(X,X,X).
gcd(X,Y,D) :- X<Y, Y1 is Y-X, gcd(X,Y1,D).
gcd(X,Y,D) :- Y<X, gcd(Y,X,D).
```

Example: length of a list

- **`length`**`(List,N)`
  succeeds when N is the number of elements in the list List

Example: length of a list

- `length(List,N)`
  succeeds when N is the number of elements in the list List
- Two cases:
  1. If the list is empty then its length is 0
  2. If the list is not empty then `List = [Head|Tail]`; and its length is equal to 1 plus the length of the tail `Tail`

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