F28PL1 Programming Languages

Lecture 18: Prolog 3

Lists

- can build any data structure out of Prolog structures
- structures are ad-hoc polymorphic
  - i.e. can contain arbitrary mixed types
- special operators provided for lists

`[]`

- empty list

`.`

- prefix binary list constructor

`.`(X, Y)

- list with X as head and Y as tail

Lists

- `[...,...]` notation like SML
- e.g. `.(1,.(2,.(3,[]))) ==> [1,2,3]`
- list patterns based on:
  - `.`
  - `[...,...]`
- head/tail match with:

$[H|T]$

  - H matches head
  - T matches tail

First N squares

- the first 0 squares are in the empty list contains
- the first N squares have N2 on the head of the first N-1 squares

```
squares(0,[]).

squares(N,[N1|T]) :-

 N1 is N*N,N2 is N-1,squares(N2,T).

| ?- squares(3,L).

L = [9,4,1]
```

First N squares

- try: `squares(3,T)`
  - try: `squares(3,[N1,T’]) :-`

    `N1 is 3*3,N2 is 3-1,squares(N2,T’)`

    try: `N1 is 3*3` – `N1 is 9`

    try: `N2 is 3-1` – `N2 is 2`

    try: `squares(2,T’)`

    - try: `squares(2,[N1’|T’’]) :-`
    - `N1’ is 2*2,N2’ is 2-1,`
    - `squares(N2’,T’’)`

      try: `N1’ is 2*2` – `N1’ is 4`

      try: `N2’ is 2-1` – `N2’ is 1`

First N squares

- try: squares(1,T’’)
  - try: squares(1,[N1’’|T’’’]) :-
  - N1’’ is 1*1,N2’’ is 1-1,
  - squares(N2’’,T’’’)
    - try: N1’’ is 1*1 – N1’’ is 1
    - try: N2’’ is 1-1 – N2’’ is 0
    - try: squares(0,T’’’)
      - matches: squares(0,[])
      - – T’’’ is []
  - T’’ is [1|[]] == [1]
- T’ is[4|[1]] == [4,1]
- T is [9|[4,1]] == [9,4,1]

List length

- the length of an empty list is 0
- the length of a non-empty list is one more than the length of the tail

```
llength([],0).

llength([_|T],L) :-

 llength(T,L1), L is L1+1.
```

List length

```
| ?- llength([a,b,c],L).
L = 3
```

- try: `llength([a,b,c],L) :-`
  `l([b,c],L1),L is L1+1`
  - try: `llength([b,c],L1)`
  - try: `llength([b,c],L1) :-`
  - `llength([c],L1’),L1 is L1’+1`
    try: `llength([c],L1’)`
    - try: `llength([c],L1’) :-`
    - `llength([],L1’’),`
    - `L1’ is L1’’+1`

List length

- try: `llength([],L1'')`
  - matches: `llength([],0)` - `L1''` instantiated to `0`
- try: `L1' is 0+1` – instantiates `L1'` to `1`
- try: `L1 is 1+1` – instantaites `L1` to `2`
- try: `L is 2+1` – instantiates `L` to `3`

List membership

- is X in a list?
- nothing is in an empty list
- X is in a list whose head is X
- X is in a list if it's in the tail

```
contains(_,[]) :- fail.
contains(X,[X|_]).
contains(X,[_|T) :- contains(X,T).
```

List membership

```
| ?- contains(3,[1,2,3]).
yes
```

- try: `contains(3,[1,2,3]) :-`
  `contains(3,[2,3])`
  - try: `contains(3,[2,3])`
  - try: `contains(3,[2,3]) :-`
  - `contains(3,[3])`
    try: `contains(3,[3])`
    - matches: `contains(3,[3|[]])`

Search pair list

- list of list of pairs `[F,S]`
- given `F` find `S`
- if `F` is the head of the first pair then `S` is the head of the tail of the first pair
- `S` is found by looking for `F` in tail

```
find(F,[[F,S]|_],S).
find(F,[_|T],S) :- find(F,T,S).
```

Search pair list

```
| ?- find(3,[[1,one],[2,two],[3,three]],S).
S = three
```

- try: `find(3,[[1,one],[2,two],[3,three]],S) :-`
  `find(3,[[2,two],[3,three]],S)`
  - try: `find(3,[[2,two],[3,three],S)`
  - try: `find(3,[[2,two],[3,three],S) :-`
    `find(3,[[3,three],S)`
    - try: `find(3,[[3,three],S)`
      - matches: `find(3,[[3,three]|[]],three)`

Ordered list

- an empty list is ordered
- a list with one element is ordered
- a list of more than one element is ordered if the head comes before the head of the tail and the tail is ordered

```
ordered([]).

ordered([A]).

ordered([A|[B|T]]) :- A<B,ordered([B|T]).
```

Ordered list

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

no

- try: `ordered([1,2,4,3]) :-`
  `1<2,ordered([2,3,4])`
  - try: `1<2`
  - try: `ordered([2,4,3])`
  - try: `ordered([2,4,3]) :-`
  - `2<4,ordered([4,3])`
    try: `2<4`
    try: `ordered([4,3])`

Ordered list

- try: `ordered([4,3]) :-`
- `4<3,ordered([4])`
  - try: `4<3`
    - fail
  - fail
- fail
- fail
- fail

List insert

- inserting V into an empty list gives a list with V
- inserting V into a list with a head and a tail:
  - gives a list with V on the front of the old list, if V comes before the old head
  - gives a list with the old head on the front of the list from inserting V into the old tail, otherwise

```
insert(V,[],[V]).

insert(V,[H|T],[V|[H|T]]) :- V<H.

insert(V,[H|T],[H|T1]) :- insert(V,T,T1).
```

List insert

```
| ?- insert(3,[1,2,4],L).
L = [1,2,3,4]
```

- try: `insert(3,[1,2,4],L)`
  - try: `insert(3,[1,2,4],[3|[1|[2,4]]) :- 3<1`
  - try: `3<1`
    - fail & backtrack
  - try: `insert(3,[1,2,4],[1|T1] :-`

    `insert(3,[2,4],T1)`

    try: `insert(3,[2,4],T1)`
    - try: `insert(3,[2,4],[3|[2|[4]]) :- 3<2`
      - try: `3<2`
        - fail & backtrack

List insert

- try: `insert(3,[2,4],[2|T1’] :-`
- `insert(3,[4],T1’)`

try: `insert(3,[4],T1’)`

- try: `insert(3,[4],[3|[4|[]]) :-`
- `3<4`

try: `3<4`

`T1’ is [3|[4|[]] == [3,4]`

– `T1 is [2|T1’] == [3,4]] == [2,3,4]`

- `L is [1|T1] == [1|[2,3,4]] == [1,2,3,4]`

List sort

- an empty list is sorted
- a list is sorted when the head is inserted into the sorted tail

```
ssort([],[]).

ssort([H|T],L) :-
ssort(T,T1),insert(H,T1,L).
```

List sort

```
| ?- sort([3,2,1],L).
L = [1,2,3]
```

- try: `ssort([3,2,1],L)`
- try: `ssort([3,2,1],L) :-`
  `ssort([2,1],T1), insert(3,T1,L)`
  - try: `ssort([2,1],T1)`
  - try: `ssort([2,1],T1) :-`
  - `ssort([1],T1'), insert(2,T1',T1)`
    try: `ssort([1],T1')`
    - try: `ssort([1],T1') :-`
    - `ssort([],T1''),`
    - `insert(1,T1'',T1')`

List sort

- try: ssort([],T1’’)
  - matches: ssort([],[])
  - - T1’’ is[]
- try: insert(1,[],T1’)
  - succeeds - T1’ is[1]
- try: insert(2,[1],T1)
  - succeeds - T1 is [1,2]
- try: insert(1,[1,2],L)
- succeeds - L is[1,2,3]

List to database

- given

```
[[1,one],[2,two],[3,three]]
```

- put:

```
word(1,one).
word(2,two).
word(3,three)
```

- in DB

List to database

- for empty list, stop
- for non-empty list with `[N,W]` in head, assert `word(N,W)` and add tail of list to DB

```
wordsToDB([]).

wordsToDB([[N,W]|T]) :-
assert(word(N,W)),wordsToDB(T).
```

List to database

```
| ?- wordsToDB([[1,one],[2,two],[3,three]])
yes
| ? – word(2,X).
X = two
```

- try: `wordsToDB([[1,one],[2,two],[3,three]])`
  - try: `assert(word(1,one))` – `word(1,one)` now in DB
  - try: `wordsToDB([[2,two],[3,three]])`
    - try: `assert(word(2,two))` – `word(2,two)` now in DB
      - try: `wordsToDB([[3,three]])`
        - try: `assert(word(3,three))` – `word(3,three)` now in DB
          - try: `wordsToDB([])`
            - matches: `wordsToDB([])`

Database to list

- suppose the database holds facts about people and their ages:

```
age(al,18).

age(bea,19).

age(cam,20).

age(deb,21).
```

- suppose we want to make a list of pairs of people and their ages
- use the technique for counting database entries

Database to list

- start with an empty list
- initiate search and set `P` to final list

```
people(P) :- assert(ages([])),getAges(P).
```

- for next age fact, add details to list
- at end, get final list

```
getAges(P) :- age(N,A),getAge(N,A).
getAges(P) :- retract(ages(P)).
```

Database to list

- to add age detail:
  - retract list
  - assert list with new detail
  - fail without backtracking

```
getAge(N,A) :-
 retract(ages(P)),
 assert(ages([[N,A]|P])),
 !,fail.
```

Database to list

```
| ?- people(L).
L = [[deb,21],[cam,20],[bea,19],[al,18]]
```

- try: `people(L)`
  - try: `people(L) :-`

    `assert(ages([])),getAges(L)`

    try: `assert(ages([]))`
    - `ages([])` now in DB

    try: `getAges(L)`
    - try: `getAges(L) :-`
    - `age(N,A),getAge(N,A)`

Database to list

- try: `age(N,A)`
  - matches: `age(al,18)` – `N` is `al` and `A` is `18`
- try: `getAge(al,18)`
  - try: `getAge(al,18) :-`
  - `retract(ages(P)),`
  - `assert(ages([[al,18]|P])),`
  - `!,fail`

    try: `retract(ages(P))`
    - matches: `ages([])` – `P` is `[]`

    try: `assert(ages([[al,18]|[]))`
    - `ages([[al,18]])` now in DB

    try: `!, fail` - backtrack

Database to list

- try: `age(N,A)`
  - matches: `age(bea,19)` – `N` is `bea` and `A` is `19`
- try: `getAge(bea,19)`
  - try: `getAge(bea,19) :-`
  - `retract(ages(P)),`
  - `assert(ages([[bea,19]|P])),`
  - `!,fail`

    try: `retract(ages(P))`
    - matches: `ages([[al,18]])` – `P` is `[[al,18]]`

    try: `assert(ages([[bea,19]|[[al,18]]))`
    - `ages([[bea,19],[al,18]])` now in DB

    try: `!, fail` - backtrack

Database to list

- try: `age(N,A)`
  - matches: `age(cam,20)` – `N` is `cam` and `A` is `20`
- try: `getAge(cam,20)`
  - `...`
  - `ages([[cam,20],[bea,19],[al,18]])`
  - now in DB
  - `...`
- try: `age(N,A)`
  - matches: `age(deb,21)` – `N` is `deb` and `A` is `21`
- try: `getAge(deb,21)`
  - `...`
  - `ages([[deb,21],[cam,20],`
  - `[bea,19],[al,18]])` now in DB
  - `...`

Database to list

- try: `age(N,A)`
  - fails
  - `...`
- try: `getAges(L) :- retract(ages(L))`
  - `L` is `[[deb,21],[cam,20],[bea,19],[al,18]]`

Input/output

- I/O based on *streams*
- current input stream
  - initially terminal
- current output stream
  - initially display

Term I/O

`read(X)`

- instantiate *X* to next term from current input stream
- prompt is: `|:`
- end term with: `.`

```
| ?- read(X).
|:  hello.
X = hello
```

- `^D` returns `end_of_file`

Term I/O

`write(`*X*`)`

- display *X*'s value on current output stream

```
| ?- write(hello).
hello
yes
| ?-
```

- value can be any Prolog term
- will be displayed using Prolog syntax

`nl`

- writes a newline

Term I/O

- continuously send terms from current input to current output
- check if next term is `end_of_file` before output

```
copyTerms1(end_of_file).
copyTerms1(X) :-
 write(X),
 read(Y),
 copyTerms1(Y).
copyTerms :- read(X),copyTerms1(X).
```

Term I/O

```
| ?- copyTerms.
|: hello.
hello
|: [1,2,3].
[1,2,3]
|: yellow(banana).
yellow(banana)
|: ^D
yes
```

Term I/O

- make list of terms from current input stream
- at `end_of_file`, list is empty
- otherwise, put next term on front of list from getting rest of terms

```
getTerms1(end_of_file,[]).

getTerms1(X,[X|L]) :- read(Y),getTerms1(Y,L).

getTerms(L) :- read(X),getTerms1(X,L).
```

Term I/O

```
| ?- getTerms(X).
|: time.
|: for.
|: lunch.
|: soon.
|: ^D
X = [time,for,lunch,soon]
```

Character

- atom with one letter
- e.g. `a b c ... z 0 1 ... 9 + - * / ...`
- quoted letter or escape character
- e.g. `‘A’ ...’Z’ ‘\n’ ‘\t’`

NB:

```
| ?- a = ‘a’.
yes
```

- but:

```
| ?- A = ‘A’.
A = ‘A’
```

Character I/O

`get_char(`*X*`)`

- instantiate *X* to next character from current input
- do not end chracter input with .

`put_char(`*X*`)`

- display value of *X* as character to current output

Character I/O

- continuously send characters from current input to current output

```
copyChars1(end_of_file).
copyChars1(X) :-
 put_char(X),
 get_char(Y),
 copyChars1(Y).
copyChars :- get_char(X),copyChars1(X).
```

Character I/O

```
| ?- copyChars.
|: once upon a time
|: there were three little computers
there were three little computers
|: ^D
yes
```

Character I/O

- make list of characters from current input stream

```
getChars(L) :-
get_char(X),getChars1(X,L).

getChars1(end_of_file,[]).

getChars1(X,[X|L]) :-
get_char(Y),getChars1(Y,L).
```

File I/O

`open(`*`file`*`,`*`mode`*`,`*`X`*`)`

- open stream for file in specified mode
- *`file`* `==>` file path – usually in `'...'`
- *`mode`* `==>` `read` or `write`
- *`X`* `==>` instantiated to name of stream for *file*

File I/O

`set_input(`*X*`)`

- change current input stream to *X*

`set_output(`*X*`)`

- change current output stream to *X*

`close(`*X*`).`

- close stream *X*

File I/O

- copy file to file

```
copyFile(X,Y) :-
 open(X,read,F1),set_input(F1),
 open(Y,write,F2),set_output(F2),
 copyChars,
 close(F1),close(F2).
| ?- copyFile('l18.pl','l18.pl.copy').
yes
```