

Announcements

- **C++ project due today**
(note: no late hand-ins!)
- **Short homework on parameter passing due on Wednesday**
- **Final exam date was wrong on web page**
(it's actually May 10th, not 12th); let us know ASAP if you have a conflict

1

Predicates on Terms

- **Accessing fields of a record:**
`datehas (year, thedate (Y,_,_), Y) .`
`datehas (month, thedate (_,M,_) , M) .`
`datehas (day, thedate (_,_,D) , D) .`
`| ?- A=thedata (2004,4,14) , datehas (year, A, Y) .`
`D = thedate (2004, 4, 14)`
`Y = 2004 ;`
`| ?- A=thedata (2004,4,D) , datehas (year,A,Y) ,`
`datehas (month,A,M) , datehas (day,A,7) .`
`A = thedate (2004,4,7) , D = 7, M = 4, Y = 2004 ?`
- **Standard order on terms:**
`| ?- thedate (2004,3,25) @< thedate (2004,4,7) .`
`yes`
`| ?- thedate (2004,3,25) @> thedate (2004,4,7) .`
`no`

2

Terms vs Predicates

- **You can't assert this and have it be a term:**
`thedata(2004,4,15).` ← the predicate `thedata`
- **The term must be encapsulated in a predicate:**
`taxday(thedata(2004,4,15)).`
 - after this, you can then query:
| `?- taxday(X), datehas(year, X, Y).`
`X = thedate(2004, 4, 15)`
`Y = 2004`

3

Predicates on Terms

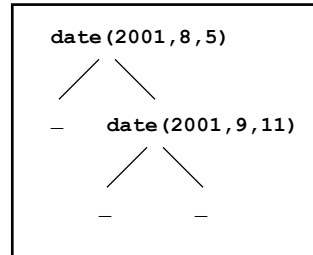
- **Binary search trees (Sethi, pp. 440-442)**
 - `memtree(A,T) → data A is in tree T`
- ```
memtree(Data,tree(Data,_,_)).
memtree(Data,tree(D,Left,_)) :-
 Data @< D, memtree(Data,Left).
memtree(Data,tree(D,_,Right)) :-
 Data @> D, memtree(Data,Right).
```
- **Data** is the label of some tree node whose root is labeled **D** if **Data** < **D** and **Data** is the label of some node in the left [right] child of **D**

4

# Predicates on Terms

- Binary search trees (Sethi, pp. 440-442):

```
mmtree(Data, tree(Data, _, _)).
mmtree(Data, tree(D, Left, _)) :-
 Data @< D,
 mmtree(Data, Left).
mmtree(Data, tree(D, _, Right)) :-
 Data @> D,
 mmtree(Data, Right).
| ?- mmtree(thedate(2001, 8, 5), T),
 mmtree(thedate(2001, 9, 11), T).
T = tree(thedate(2001, 8, 5), _A,
 tree(thedate(2001, 9, 11), _B, _C))
```



5

# Extra-Logical Predicates

- Arithmetic:

```
| ?- X is 2+3.
```

evaluated

```
X = 5 ? yes
```

not evaluated (it's just a term)

```
| ?- X is 2+3, X = 2+3.
```

```
no
```

```
| ?- X is 2+3, Y is 4*X.
```

```
X = 5, Y = 20 ? yes
```

```
| ?- X is 2+3, Y is 4*X, X > Y.
```

```
no
```

6

## Extra-Logical Predicates

- **Metalogic:**

**var (Term) :** true if **Term** is currently uninstantiated.

**nonvar (Term) :** true if **Term** is currently instantiated  
(i.e. has a partial definition)

**ground (Term) :** true if **Term** is completely instantiated  
(i.e. no variables in it anywhere)

- **Example:**

```
plus(X,Y,Z) :- ground(X), ground(Y), Z is X+Y.
```

```
plus(X,Y,Z) :- ground(X), ground(Z), Y is Z-X.
```

```
plus(X,Y,Z) :- ground(Y), ground(Z), X is Z-Y.
```

```
| ?- plus(4,Y,7).
```

```
Y = 3 ?
```

7

## Extra-Logical Predicates

- **Finding all solutions:**

**findall (Template, Goal, Bag)**

succeeds with **Bag** instantiated to a list of the instances of **Template** that correspond to the solutions of **Goal**.

- **Examples:**

```
| ?- [family].
```

```
...
```

```
| ?- findall(D,ancestor(zeus,D),L).
```

```
L = [ares,dionysus,harmonia,semele,dionysus] ?
```

```
| ?- findall((X,Y),sibling(X,Y),L).
```

```
L = [(edward,alice),(alice,edward)] ?
```

```
| ?- findall((X,Y), (sibling(X,Y), X @< Y), L).
```

```
L = [(alice, edward)] ;
```

8

# Factorial

## First try:

- loops after it finds a solution (if there is one)

```
fact(0,1).
```

```
fact(X,Y) :- fact(Xp,Yp), X is Xp+1, Y is Yp*X.
```

```
?- fact(X,Y).
```

```
X = 0 Y = 1 ;
```

```
X = 1 Y = 1 ;
```

```
X = 2 Y = 2 ;
```

```
X = 3 Y = 6 ;
```

```
X = 4 Y = 2
```

```
?- fact(5,X).
```

```
X = 120 ;
```

```
ERROR: Arithmetic: evaluation error: `float_overflow`
```

```
?- fact(X,120).
```

```
X = 5 ;
```

```
ERROR: Arithmetic: evaluation error: `float_overflow`
```

9

# Factorial

## Second try

- no loops, but gets instantiation errors since the arithmetic can't be solved backwards

```
fact(0,1).
```

```
fact(X,Y) :- X > 0, Xp is X-1, fact(Xp,Yp), Y is Yp*X.
```

```
?- fact(5,X).
```

```
X = 120 ;
```

```
No
```

```
?- fact(X,120).
```

```
ERROR: Arguments are not sufficiently instantiated
```

```
?- fact(X,Y).
```

```
X = 0 Y = 1 ;
```

```
ERROR: Arguments are not sufficiently instantiated
```

10

# Factorial

## Third try

- no loops, no errors, but can't be solved backwards

```
fact(0,1).
```

```
fact(X,Y) :- ground(X),
 X > 0, Xp is X-1, fact(Xp,Yp), Y is Yp*X.
```

```
?- fact(5,X).
```

```
X = 120 ;
```

```
No
```

```
?- fact(X,120).
```

```
No
```

```
?- fact(X,Y).
```

```
X = 0 Y = 1 ;
```

```
No
```

11

# Negation by Failure

- **Negation isn't equivalent to logical *not* in Prolog**
  - Prolog can only assert that something is true
  - Prolog cannot assert that something is false, but only that it cannot be proven with the given rules
- **not()** turns a “yes” into a “no”, or vice versa
  - if the “yes” comes with unifications, they are thrown away
  - not(X) can only return “yes” or “no”
  - not(not(X)) is different from X

12

## Negation by Failure: implementation

```
not(X) :- X, !, fail.
```

```
not(_) .
```

- if X succeeds in first rule, then the goal fails because of the last term.
- if we type “;” the cut (!) will prevent us from backtracking over it or trying the second rule so there is no way to undue the fail.
- if X fails in the first rule, then the goal fails because subgoal X fails. the system tries the second rule which succeeds, since “\_” unifies with anything.

13

## Negation example

```
mem(X, [X|_]) .
```

```
mem(X, [_|Tail]) :- mem(X, Tail) .
```

```
notmem(X, Y) :- \+ mem(X, Y) .
```

```
?- notmem(2, [3,4,5]) .
```

Yes

```
?- mem(X, [3,4,5]) .
```

```
X = 3 ;
```

```
X = 4 ;
```

```
X = 5 ;
```

```
?- notmem(X, [3,4,5]) .
```

No

```
?- not(notmem(X, [3,4,5])) .
```

```
X = _G300 ;
```

No

14

## Generate and Test

- The basic “control flow” model provided by backtracking Prolog (others are possible...)
- Well suited to search problems solvable with greedy algorithms; can also be used for harder problems, but it can take exponential time
- One example: factorial (the first try)
  - it generated all  $(n, n!)$  pairs starting at  $(0,1)$  until the desired one “went by”

15

## Generate and Test

- Making change for a dollar  
**change ( [Q, D, N, P] ) .**
  - generates or checks that Q quarters, D dimes, N nickels and P pennies add up to \$1

16

# Change

```
change ([Q,D,N,P]) :-
 member (Q, [0,1,2,3,4]),
 member (D, [0,1,2,3,4,5,6,7,8,9,10]),
 member (N, [0,1,2,3,4,5,6,7,8,9,10,
 11,12,13,14,15,16,17,18,19,20]),
 S is 25*Q + 10*D + 5*N,
 S <= 100,
 P is 100-S.
```

```
?- change ([3,N,D,0]).
```

```
N = 0 D = 5 ;
```

```
N = 1 D = 3 ;
```

```
N = 2 D = 1 ;
```

```
No
```

(example after J.R.Fisher)

17