CSE583: Programming Languages

David Notkin
15 February 2000
notkin@cs.washington.edu
http://www.cs.washington.edu/education/courses/583

Two weeks: logic and constraint logic programming paradigms

- Use logic and theorem proving as the underlying computational model
- From a set of axioms and rules, a program executes by trying to prove a given hypothesis
- In constraint logic programming, more information is provided about the domain, which can increase the efficiency of the programs significantly

Note

- Many of the following slides were taken (and in some cases adapted), with permission, from Greg Badros

(Symbolic) Logic

- Logic goes back to the Greeks, providing a basis for rational reasoning
- Aristotle's logic was based in natural language, which led to ambiguity
- Philosophers over the years have cast logic symbolically

Logic basics

- There are a set of terms intended to represent facts or properties of the real world
  - r ≡ it rained in Seattle yesterday
  - w ≡ Schell is a wuss
  - p ≡ CSE583 in Winter 2000 is taught on Tuesday evenings
  - x ≡ P = NP
- For most logics, these terms are either true or false

Connectives

- These logical terms can be combined in well-defined ways
- These define rules for combining logical formulae
- Truth tables define the connectives

p	T	F
¬p	F	T

- $p \wedge q$ (and)
- $p \vee q$ (inclusive or)
- $\neg p$ (negation)
- $p \rightarrow q$ (implication)
- …

p	T	F	T	F
q	T	F	F	T
p → q	T	T	F	T

Quantifiers

- Some logics have various quantifiers as well
  - $\exists$ x • x > 0
  - $\forall$ x • x > 0
  - Temporal logics include
    - AG (always globally), EF (there exists a path), AX (always in the next step), …

Formulae

- Formulae written in terms of the connectives can be checked using truth tables
- $p \wedge p \equiv p$
- $\neg(p \wedge q) \equiv \neg p \vee \neg q$
- …

p	T	F
$p \wedge p$	T	F

p	T	F	T	F
q	T	F	F	T
$\neg(p \wedge q)$	F	T	T	T
$\neg p \vee \neg q$	F	T	T	T

$\equiv$

- This is not a connective
- It doesn't define a new logical relationship (like $p \wedge q$) but rather it relates two logical statements
- In particular, it states that both statements are logically equivalent
  - That is, they have the same truth table

Predicate logic and English

If A then B	$\equiv$	$A \rightarrow B$
A only if B	$\equiv$	$A \leftarrow B$
A if and only if B	$\equiv$	$A \leftrightarrow B \equiv B \leftrightarrow A$
A iff B	$\equiv$	$A \leftrightarrow B \equiv B \leftrightarrow A$

"She lives in Seattle only if she lives in Washington State."

Theorems

- Anything you can prove from your terms and rules is a theorem in the logic
  - Remember (and I'll say it again) the *interpretation* of the terms is not part of the logic per se
  - The logic and theorem proving is strictly symbolic manipulation

- r $\equiv$ it rained in Seattle yesterday
- w $\equiv$ Schell is a wuss
- p $\equiv$ CSE583 in Winter 2000 is taught on Tuesday evenings
- x $\equiv$ P = NP

- $p \wedge r$?
- $w \rightarrow p$?
- $p \rightarrow w$?

Tautologies and contradictions

- Tautologies are logical statements that are always true
  - $p \vee \neg p \equiv T$
  - $T \vee p \equiv T$
- Contradictions are logical statements that are always false
  - $p \wedge \neg p \equiv F$
  - $F \wedge p \equiv F$

But...

- ...what if it's not a tautology?
- How do we prove it?
- The most common way is to use a deduction rule called *modus ponens*
  - Prove $p \rightarrow q$
  - Then prove p
  - This in turns proves q
- More soon

Prolog

- To make all this a bit more concrete and to connect it to programming, we'll look at Prolog
  - By far the best known and most influential logic programming language

Prolog deals with relations

```
?- isSquareOf(9,3).          yes
?- isSquareOf(9,2).           no
?- isSquareOf(25,X).         X=5 ;
                            X=-5
?- isSquareOf(X,-3).         X=9
```

- The program (called a query) can run in many directions
  - It's trying to prove the formula given underlying facts
- The query can produce many answers

History of Prolog

- Developed in 1970s by Alan Colmeraur, Robert Kowalski, Phillip Roussel (University of Marseilles, France)
- David H. D. Warren provided foundations of modern implementation in the Warren Abstract Machine for DEC PDP-10 (University of Edinburgh)
- Prolog is basis for numerous other languages such as CLP(R), Prolog III, etc.

Not for general purpose programming

- More restricted computation model of proving assertions from collections of facts and rules
- Think of queries working on a database of facts with rules that permit inferring new facts
- Query is just a theorem to be proven

Why restrict applicability of a language?

- Prolog provides better built-in support for the algorithms and tasks especially useful in search problems
  - Theorem proving is "just" a search problem
- Search problems are incredibly important
  - Exponential complexity
  - But efficient techniques and heuristics help solve practical programs in a timely fashion

Example applications

- Medical patient diagnosis
- Theorem proving
- Solving Rubik's cube
- Type checking
  - Type inference in ML and Haskell is done in this way
- Database querying
- …

A Prolog Program

- Facts and rules
  - a database of information, and rules to infer more facts
- Queries
  - the searches to perform

Example facts and simple queries

```
female(karen).
male(joseph).
male(mark).
male(greg).
male(eric).
person(karen).
a_silly_fact.
```

```
?- a_silly_fact.  yes.
?- male(eric).    yes.
?- female(F).     F = karen.
?- male(bob).     no.
?- person(mark).  no.
```

∃F • female(F)

Syntax for facts

```
predicate.
predicate(arg1,arg2,...).
```

- Begin with lowercase letter
- End with a period (.)
- Numbers and underscores (_) are okay inside identifiers (also called atoms)

Variables

- Begin with an uppercase letter
- Either "instantiated" or "uninstantiated"
- X is instantiated means X stands for a particular value (similar to binding)
- Variables instantiations can be undone
  - Used to produce multiple answers during search
- Multiple uses of the same variable in same scope must refer to same value

Variables are scoped within a query

These two uses of X must represent same value

```
?- person(X), female(X).      X=karen
```

Read the comma as "and"

More facts

```
/* parent(P,C) means P is a
   parent of C */
parent(karen,greg).
parent(joseph,greg).
parent(karen,mark).
parent(joseph,mark).
```

- Interpretation of facts is imposed by the programmer
  - Biological parent? Genetic parent? Adoptive parent? Etc.
- Make assumptions clear!

A simple rule

Read "if"

```
mother(M,C) :- parent(M,C), female(M).
```

Same M variable

So,

$\forall$ `M,C•((parent(M,C) ∧ female(M)) → mother(M,C))`

Example

```
parent(karen,greg).
parent(joseph,greg).
parent(karen,mark).
parent(joseph,mark).
female(karen).
mother(M,C) :-
  parent(M,C),
  female(M).
```

```
?-mother(karen,greg).
YES
?-mother(karen,X).
X = greg ;
X = mark
?-mother(karen,joseph).
NO
?-mother(joseph,karen).
NO
```


Proving

- To answer these kinds of queries, Prolog must search through the facts and the rules in all possible combinations
- If Prolog can't find a proof, then it says that the theorem is false
  - Closed world assumption
  - How valid is this assumption?
  - `?-parent(hillary,chelsea)`

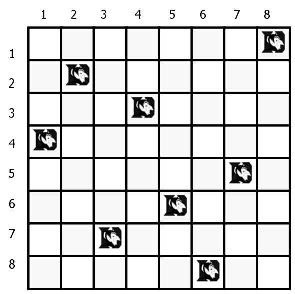
Two interpretations of rule

- Declarative (logic)
  For a given M and C, M is the mother of C if M is the parent of C and M is female
- Procedural (computational)
  Prove M is mother of C by proving subgoals that M is a parent of C and that M is female

Eight Queens: A typical search problem

- Place eight (or n) queens on an $n \times n$ chessboard such that none of them are attacking any of the others
- Recursive solutions are naturally expressed using backtracking
- Solutions in C++, Pascal, Java, etc., are generally around 140–220 lines of uncommented code.

A Solution to Eight Queens



Eight Queens in Prolog

```
/* From Bratko's Prolog Progr. for AI, p. 111 */
solution([]).
solution([X/Y | Others] ) :-
    solution(Others),
    member(Y, [1,2,3,4,5,6,7,8] ),
    noattack( X/Y, Others).

noattack(_, []).
noattack(X/Y, [X1/Y1 | Others]) :-
    Y =\= Y1,
    Y1-Y =\= X1-X,
    Y1-Y =\= X-X1,
    noattack( X/Y, Others).

template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```


Query for solution

```
?- template(S), solution(S).
  S = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1]
                                              ;
  S = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1]
                                              ;
...
```

- **92 solutions in all!**

Our friend the list

```
?- append([1,2,3],[4,5],L).   L=[1,2,3,4,5]
?- append([1,2],M,[1,2,3]).          M=[3]
?- append(A,B,[1,2]).         A=[], B=[1,2]
                               A=[1], B=[2]
                              A=[1,2], B=[]
```

- **append works in multiple directions!**

Declaration of append rule

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

- **Think** declaratively!
- **Think** recursively!

"Return value" is an argument of the rule

```
/* append(X,Y,Z)
   succeeds iff  Z is the list that is
   the list Y appended to the list X */
```

- **Enables it to use any/all of the arguments to compute what's left**
- **Use uninstantiated variables (i.e., those starting with capital letters) to ask for a return value**

Terminology

- simple term — number, variable, or atom
  e.g., `-92  X  greg`
- compound term —
  atom + parenthesized subterms
  e.g., `parent(joe,greg)`
  functor ↗ ↖ sub-term arguments
- Facts, rules, and queries are made from terms… the functor is the predicate
  – Not related to ML functors

Lists

- `[]` is the empty list
- `.` predicate is like Scheme's cons:

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

- `[…]` shorthand syntax:

```
?- A = [1,2,3]                 A=[1, 2, 3]
```

- `[E1...|Tail]` notation

```
?- A = [1,2|3].                A=[1, 2|3]
?- A = [1,2|[3]].              A=[1, 2, 3]
```


Lists need not be homogeneous

```
?- A = "Hi".                     A=[72,105]
?- A = [1,"Hi",greg].     A=[1,[72,105],greg]
?- A = [1,g], B=[A,A].           A=[1,g]
                           B=[[1,g],[1,g]]
?- A = [1,g], B=[A|A].           A=[1,g]
                             B=[[1,g],1,g]
```


Unification of terms

- Similar to ML/Haskell pattern matching

Request to unify

```
?- p(X,foo) = p(bar,Y)       X=bar, Y=foo
?- foo = X.                         X=foo
?- X = foo.                         X=foo
?- foo = bar.                          No
```


Unification of terms S and T

Terms *S* and *T* unify if and only if:

- *S* and *T* are both constants, and they are the same object; or
- *S* is uninstantiated. Substitute *T* for *S* ; or
- *T* is uninstantiated. Substitute *S* for *T* ; or
- *S* and *T* are structures, have same principal functor, and the corresponding components unify.

Recursive definition!

More unification examples

```
?- A=parent(joe,greg),A=parent(X,Y).
                              X=joe, Y=greg
?- A=parent(joe,greg),A=parent(X).       No
?- A=people([joe,greg]),A=people(X).
                                X=[joe,greg]
?- A=[1,2,3,4],A=[X,Y|Z].     X=1,Y=2,Z=[3,4]
```

Unification is implicit in rule application

```
/* simple rule: X is the same as X. */
identity(X,X).

?- identity( p(X,foo), p(bar,Y) ).
                              X=bar, Y=foo

/* could have written the rule as: */
identity(X,Y) :- X = Y.
```


Unification in append rule

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

`append([1,2,3],A,[1,2,3,4]).` **results in**

```
  [X|Xs] = [1,2,3], A=Ys,
  [X|Zs] = [1,2,3,4]
```

and thus:

```
  X=1, Xs=[2,3], Zs=[2,3,4]
```

so must prove: `append([2,3],A,[2,3,4])`

Trace of app([1,2,3],A,[1,2,3,4])

```
T Call:(7)   app([1,2,3], _G235, [1,2,3,4])
T Call:(8)   app([2,3], _G235, [2,3,4])
T Call:(9)   app([3], _G235, [3,4])
T Call:(10)  app([], _G235, [4])
T Exit:(10)  app([], [4], [4])
T Exit:(9)   app([3], [4], [3,4])
T Exit:(8)   app([2, ], [4], [2,3,4])
T Exit:(7)   app([1,2,3], [4], [1,2,3,4])
```


Trace of app([1,2,3],[4],Z)

```
T Call:( 7)app([1,2,3], [4], _G191)
T Call:( 8)app([2,3], [4], _G299)
T Call:( 9)app([3], [4], _G302)
T Call:(10)app([], [4], _G305)
T Exit:(10)app([], [4], [4])
T Exit:( 9)app([3], [4], [3,4])
T Exit:( 8)app([2,3], [4], [2,3,4])
T Exit:( 7)app([1,2,3], [4], [1,2,3,4])
```


Horn clauses

- **Prolog cannot handle arbitrary logic over relations**
- **It supports Horn clauses only**
  - $p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow q$
  - p
- **Implication is written "backwards"**
  - `mother(M,C) :- parent(M,C), female(M).`
  - `female(karen).`
- **Cannot say**
  - $p_1 \wedge p_2 \rightarrow q_1 \wedge q_2$
  - And other stuff

Why this limitation?

- **It allows for a simple underlying theorem prover**
  - Without it, theorem provers produce tons of interim and often unnecessary results
- **This search process for proving queries is called *resolution* (Robinson)**
  - Horn clauses allow a goal-oriented, backtracking search

Example

```
parent(karen,greg).
parent(joseph,greg).
parent(karen,mark).
parent(joseph,mark).
female(karen).
mother(M,C) :-
  parent(M,C),
  female(M).
```

```
?-mother(karen,X).
X = greg ;
X = mark
```

- Find the first functor that unifies to the query
- Use the conjuncts on the right of that rule as subgoals
  - Prove them in order
  - If all are true, it is proven
  - Backtrack for additional answers or until failure

Resolution

- Ordering is especially important for recursive rule sets
- The two sets of rules on the right are logically equivalent
  - Conjunction is commutative
- But they are computationally very different

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

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


Arithmetic in Prolog

`?- X=2+3, X=5.` No

- `2+3` **does not unify with** `5`
  - `2+3` **is an** unevaluated **expression that is not the same as the literal** `5`
    - **like x/y was in the 8 queens**

`?- X is 2+3, X=5.` Yes

(`is`: Force arithmetic evaluation)

Change for a dollar: J.R. Fisher

```
change([H,Q,D,N,P]) :-
  member(H,[0,1,2]), /* Half-dollars */
  member(Q,[0,1,2,3,4]), /* etc. */
  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 50*H + 25*Q +10*D + 5*N,
  S =< 100,
  P is 100-S.
```


Arithmetic only works forward in Prolog

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

?- sum(4,5,Z).          Z=9
?- sum(4,Y,9).          Error!
```

- **We'll come back to this in CLP(R), which can do it both ways!**

member rule

```
/*member(X,Y) succeeds iff
  X is a member of the list Y. */
```

Example uses:

```
?- member(1,[1,2,3]).        Yes
?- member(7,[1,2,3]).         No
?- member(3,foo).             No
?- member(foo,[]).            No
```

Definition of member

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

- X is a member of a list starting with X; and X is a member of a list starting with anything as long as it is a member of the rest of the list

The declarative interpretation falls short…

Two queries with identical logical semantics:

```
X = [1,2,3], member(7,X).
member(7,X), X = [1,2,3].

?- X = [1,2,3], member(7,X). No
?- member(7,X), X=[1,2,3].   …
```

Infinite computation

More uses of member

```
?- member(X,[]).                No
?- member(X,[1,2,3]).            1 ;
                                 2 ;
                                 3 ;
                                No
?- member(7,X).       X = [7|_G219] ;
               X = [_G218, 7|_G222] ;
        X = [_G218, _G221, 7|_G225]
  ;…
```


Reminder: How Prolog tries to prove

- Rule order
  - Select the first applicable rule from top to bottom
- Goal order
  - Prove subgoals from left to right

Evidence of top to bottom rule order

```
?- male(X).               X=joseph ;
                            X=mark ;
                            X=greg ;
                            X=eric ;
                                No
```

- Order of results mirrors the order that the facts appeared in the database

Order of rules matters!

```
mem1(X,[X|_]).
mem1(X,[_|T]) :- mem1(X,T).
```

versus

```
mem2(X,[_|T]) :- mem2(X,T).
mem2(X,[X|_]).
```

➪ Which is more efficient?

Trace of mem1(2,[1,2,3])

```
T Call:  (  8) mem1(2, [1, 2, 3])
T Call:  (  9) mem1(2, [2, 3])
T Exit:  (  9) mem1(2, [2, 3])
T Exit:  (  8) mem1(2, [1, 2, 3])
```


Trace of mem2(2,[1,2,3])

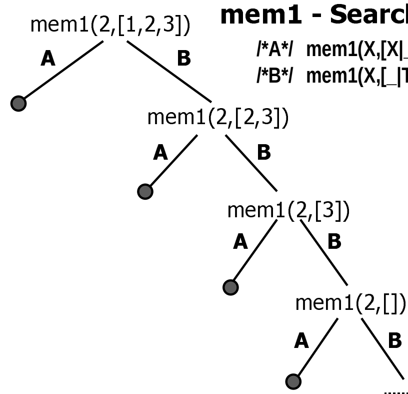
```
T Call:  (  8) mem2(2, [1, 2, 3])
T Call:  (  9) mem2(2, [2, 3])
T Call:  ( 10) mem2(2, [3])
T Call:  ( 11) mem2(2, [])
T Fail:  ( 11) mem2(2, [])
T Redo:  ( 10) mem2(2, [3])
T Fail:  ( 10) mem2(2, [3])
T Redo:  (  9) mem2(2, [2, 3])
T Exit:  (  9) mem2(2, [2, 3])
T Exit:  (  8) mem2(2, [1, 2, 3])
```


mem1 was faster

- mem1 had the base case listed first
  - base case had no sub-goals

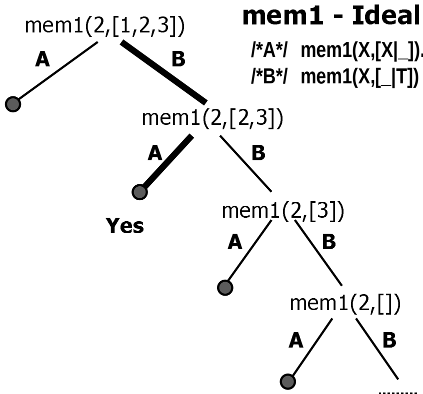
mem1 - Search Structure

/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).



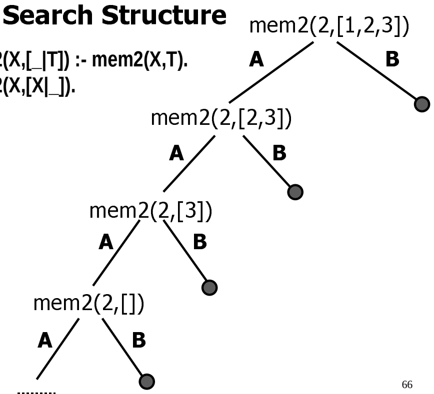
mem1 - Ideal Solution

/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).



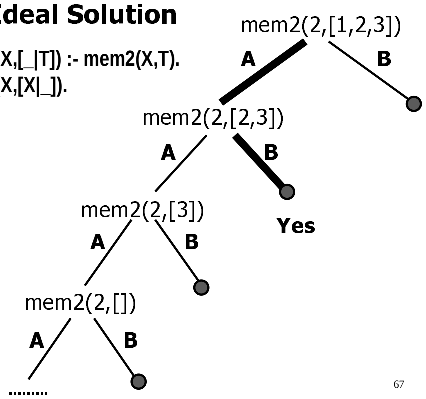
mem2 - Search Structure

/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).



mem2 - Ideal Solution

/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).



67

Prolog is not an oracle!

- Resolution gives strict rules
  - First applicable rule
  - Prove subgoals in order

How Prolog Proceeds - 1

/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).

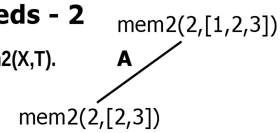
mem2(2,[1,2,3])

Matches **A**,
Unify [_|T]
with [1,2,3]
So T = [2,3],
and must prove
mem2(X,T).

69

How Prolog Proceeds - 2

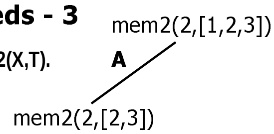
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).



70

How Prolog Proceeds - 3

/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).

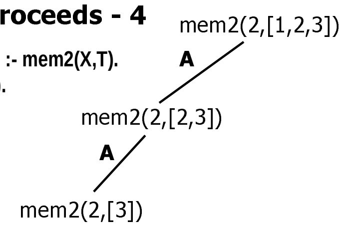


Matches **A**,
Unify [_|T]
with [2,3]
So T = [3],
and must prove
mem2(X,T).

71

How Prolog Proceeds - 4

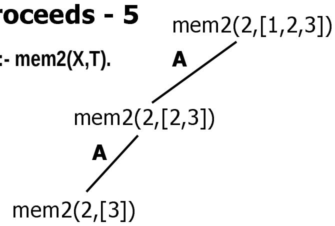
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).



72

How Prolog Proceeds - 5

/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).

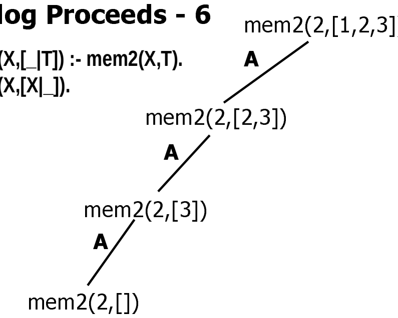


Matches **A**,
Unify [_|T]
with [3]
So T = [],
and must prove
mem2(X,T).

73

How Prolog Proceeds - 6

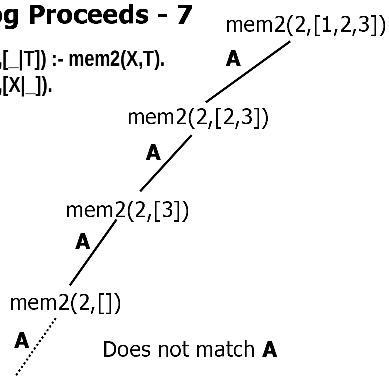
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).



74

How Prolog Proceeds - 7

/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).

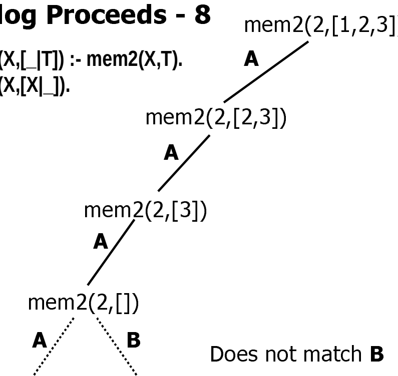


Does not match **A**

75

How Prolog Proceeds - 8

/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).

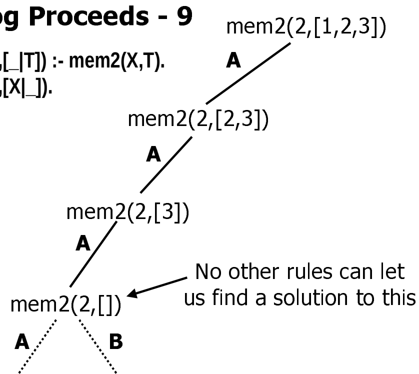


Does not match **B**

76

How Prolog Proceeds - 9

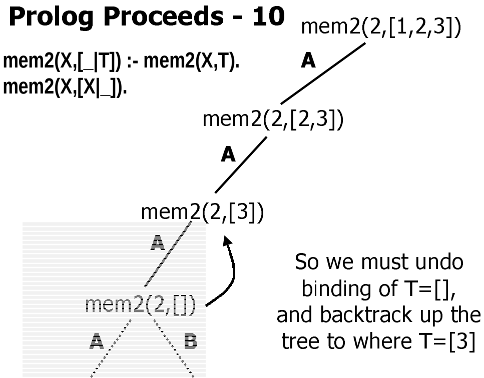
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).



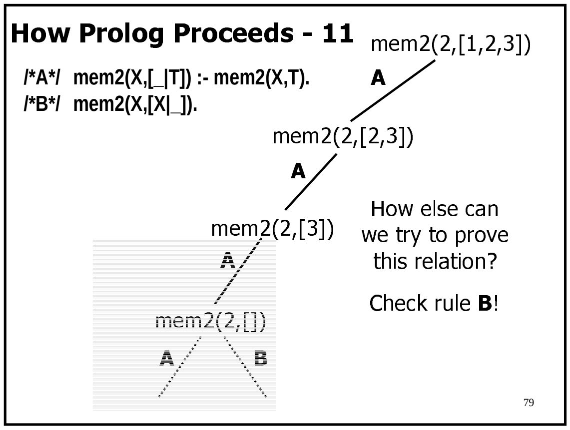
77

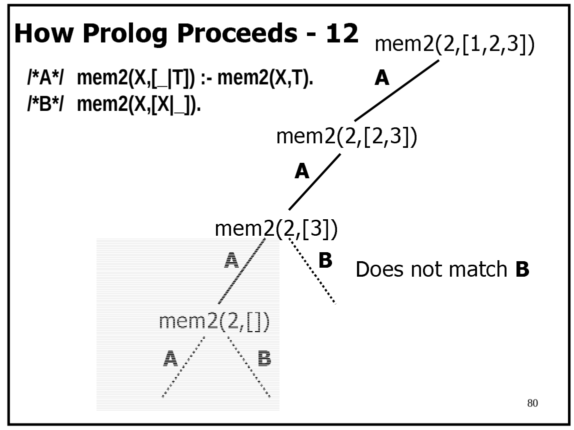
How Prolog Proceeds - 10

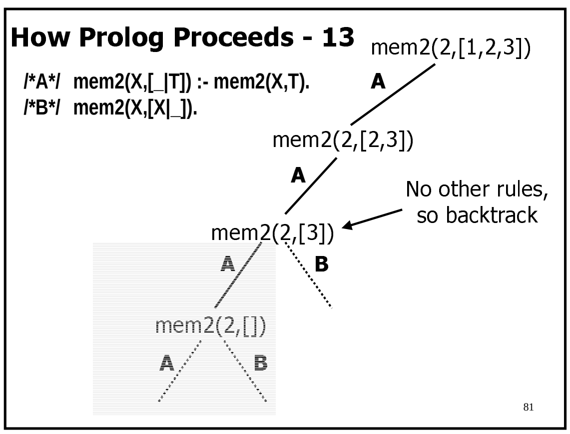
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).

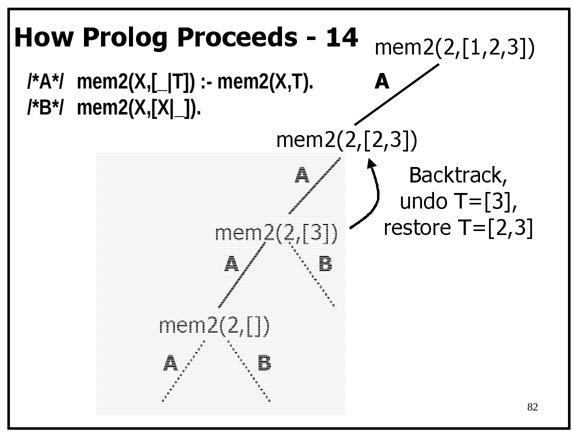


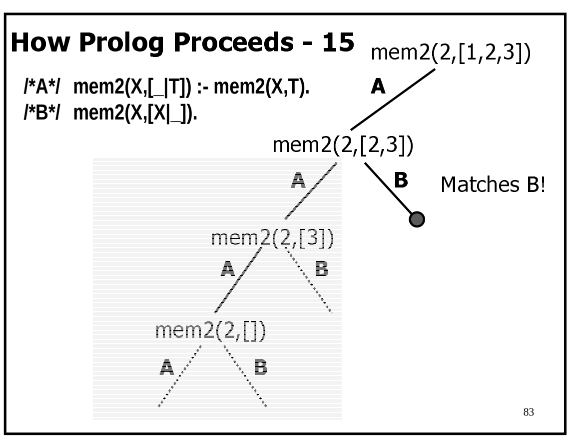
78

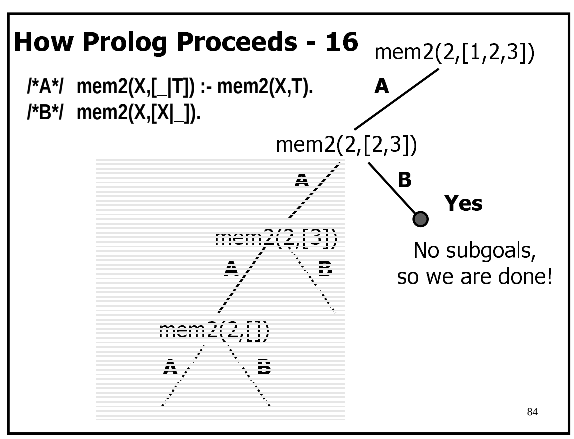

How Prolog Proceeds - 11
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
mem2(2,[3])
A
mem2(2,[])
A
B
How else can we try to prove this relation?
Check rule B!
79


How Prolog Proceeds - 12
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
mem2(2,[3])
A
B
Does not match B
mem2(2,[])
A
B
80


How Prolog Proceeds - 13
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
No other rules, so backtrack
mem2(2,[3])
A
B
mem2(2,[])
A
B
81


How Prolog Proceeds - 14
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
Backtrack, undo T=[3], restore T=[2,3]
mem2(2,[3])
A
B
mem2(2,[])
A
B
82


How Prolog Proceeds - 15
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
B
Matches B!
mem2(2,[3])
A
B
mem2(2,[])
A
B
83


How Prolog Proceeds - 16
/*A*/ mem2(X,[_|T]) :- mem2(X,T).
/*B*/ mem2(X,[X|_]).
mem2(2,[1,2,3])
A
mem2(2,[2,3])
A
B
Yes
No subgoals, so we are done!
mem2(2,[3])
A
B
mem2(2,[])
A
B
84

Search algorithm

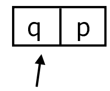
- Depth first visitation of the nodes in the search tree
- What about rules with multiple goals?

Multiple sub-goals

p :- a, b, c.
p :- m, f.
q :- m, n.
r :- q, p.
r :- a, n.
n.
a.

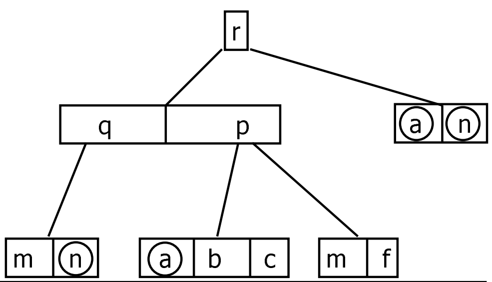
?- r.

Nodes will now contain multiple sub-goals each the root of a new tree

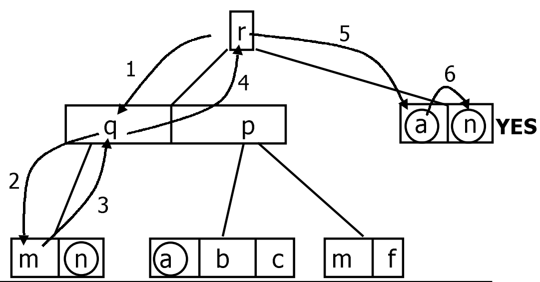


Must prove **q**, and then prove **p**

Another search tree…

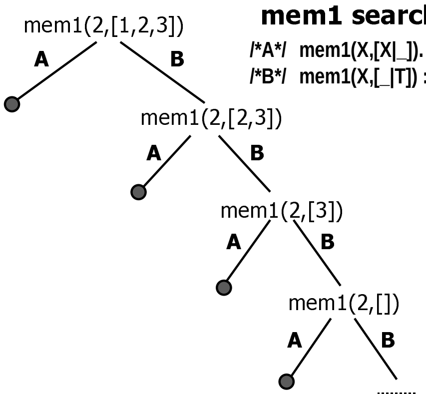


Tracing through the tree



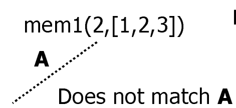
mem1 search tree

/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).



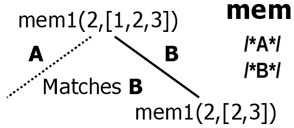
mem1 search trace - 1

/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).



mem1 search trace - 2

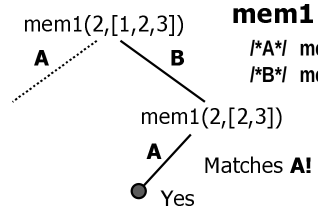
/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).



91

mem1 search trace - 3

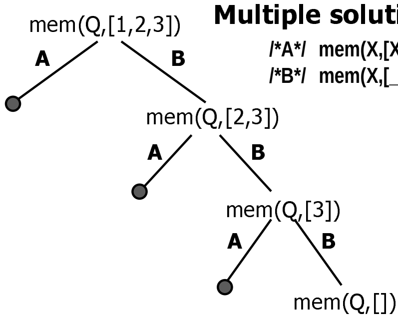
/*A*/ mem1(X,[X|_]).
/*B*/ mem1(X,[_|T]) :- mem1(X,T).



92

Multiple solutions trace 1

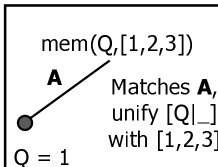
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).



93

Multiple solutions trace 2

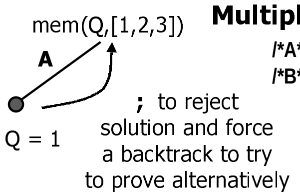
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).



94

Multiple solutions trace 3

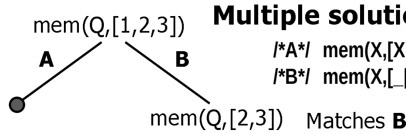
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).



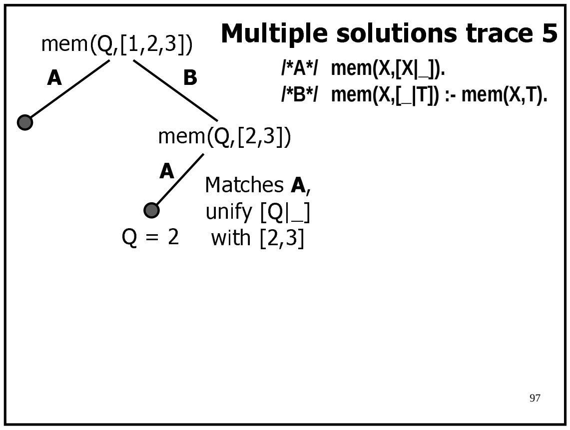
95

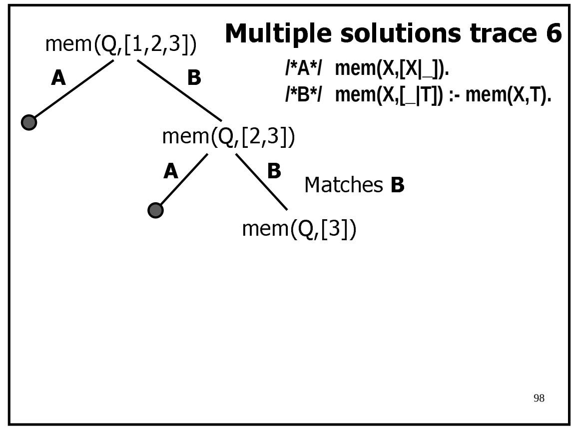
Multiple solutions trace 4

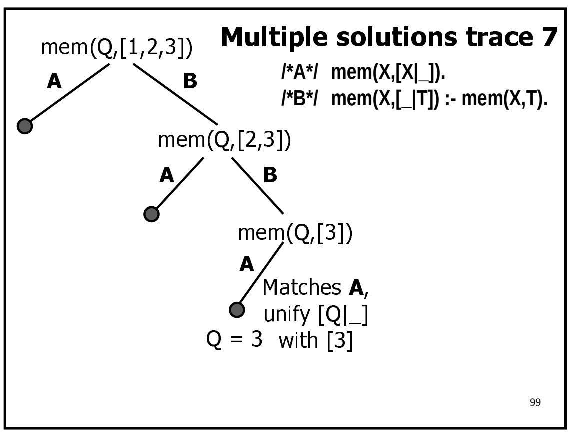
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).

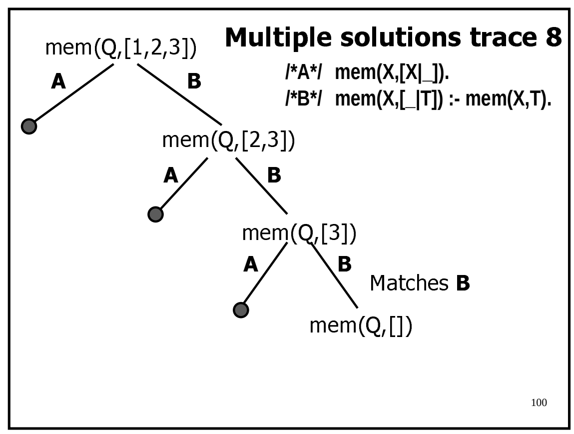


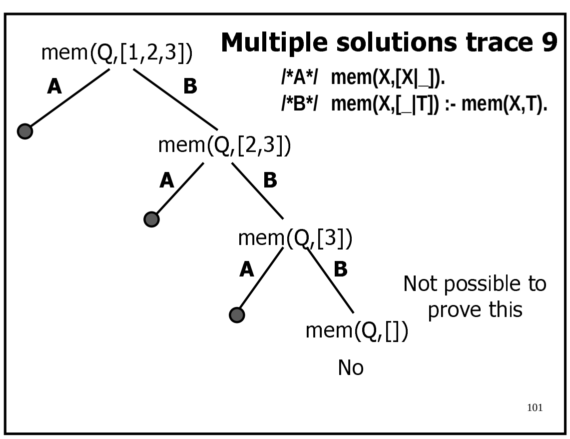
96

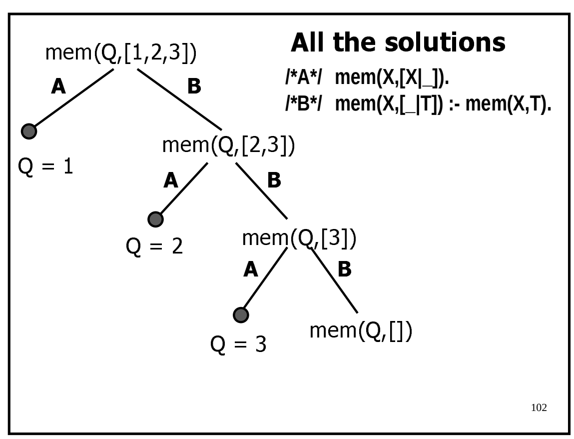

Multiple solutions trace 5
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
mem(Q,[1,2,3])
A
B
mem(Q,[2,3])
A
Matches **A**,
unify [Q|_]
with [2,3]
Q = 2
97


Multiple solutions trace 6
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
mem(Q,[1,2,3])
A
B
mem(Q,[2,3])
A
B
Matches **B**
mem(Q,[3])
98


Multiple solutions trace 7
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
mem(Q,[1,2,3])
A
B
mem(Q,[2,3])
A
B
mem(Q,[3])
A
Matches **A**,
unify [Q|_]
with [3]
Q = 3
99


Multiple solutions trace 8
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
mem(Q,[1,2,3])
A
B
mem(Q,[2,3])
A
B
mem(Q,[3])
A
B
Matches **B**
mem(Q,[])
100


Multiple solutions trace 9
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
mem(Q,[1,2,3])
A
B
mem(Q,[2,3])
A
B
mem(Q,[3])
A
B
Not possible to
prove this
mem(Q,[])
No
101


All the solutions
/*A*/ mem(X,[X|_]).
/*B*/ mem(X,[_|T]) :- mem(X,T).
mem(Q,[1,2,3])
A
B
Q = 1
mem(Q,[2,3])
A
B
Q = 2
mem(Q,[3])
A
B
Q = 3
mem(Q,[])
102

Backtracking is not always what we want

- Patterns may match where we do not intend
- Backtracking is expensive—we may know more about our problem and can help the algorithm be “smarter”
- We may want to specify a situation that we know definitively results in failure

delete_all example

```
/* delete_all(List,E,Result) means that
   Result is a a list just like List except
   all elements E are missing. */
delete_all([], E, []).

delete_all([E|Tail], E, Res) :-
     delete_all(Tail, E, Res).

delete_all([Head|Tail], E, [Head|Res]) :-
     delete_all(Tail, E, Res).
```


A query for delete_all

```
?- delete_all([1,2,3],2,R).        R=[1,3] ;
                                 R=[1,2,3] ;
                                        No
```

Why this solution?

delete_all can succeed in any of three ways...

```
delete_all([], E, []).

delete_all([E|Tail], E, Res) :-
     delete_all(Tail, E, Res).

delete_all([Head|Tail], E, [Head|Res]) :-
     delete_all(Tail, E, Res).
```

- Order in file only tells which rules are attempted first—later matching rules can be used after backtracking!

delete_all has multiple matching rules

```
delete_all([E|Tail], E, Res) :-
     delete_all(Tail, E, Res).

delete_all([Head|Tail], E, [Head|Res]) :-
     delete_all(Tail, E, Res).

delete_all([2,3],2,R).
```

- Can be proven using either of the above!
  R=[3], or R=[2,3]

Third rule contained implicit assumption

```
delete_all([Head|Tail], E, [Head|Res]) :-
     delete_all(Tail, E, Res).
```

- Want above rule to apply only when Head is not E
- That is exactly the complement of rule 2
- So we can make the algorithm only try rule 3 if rule 2 did not succeed

Use a "cut" — !

- We can make rule 2 prevent backtracking with the "cut" operator, written !

```
delete_all([E|Tail], E, Res) :-
     delete_all(Tail, E, Res), !.
```

- Now the search algorithm will not try any other rules for delete_all after the above rule succeeds
- ! succeeds and stops further backtracking for more results

The query again

```
?- delete_all([1,2,3],2,R).        R=[1,3] ;
                                        No
```

- Now we get only the single correct solution!

Cut divides problem into backtracking regions

```
foo := a, b, c, !, d, e, f.
```

- Search may try various ways to prove a, b, and c, backtracking freely while solving those sub-goals
- Once a, b, and c are proved, that sub-answer is frozen, and d, e, f must be proved without changing a, b, or c

Controversy over cut

- Prolog is meant to be declarative
- cut operator alters the behavior of the built-in searching algorithm
- No declarative interpretation for cut—
  you must think about resolution to understand its effects

cut and not

- We can write the not predicate using a cut operator:

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

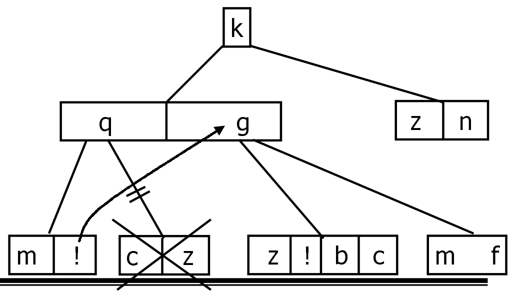
- Uses built-in fail predicate that always fails
- Cut operator prevents the search algorithm from backtracking to use the second rule to satisfy P when the first rule already failed
- 2nd rule applies only if P cannot be proven

!, fail combination

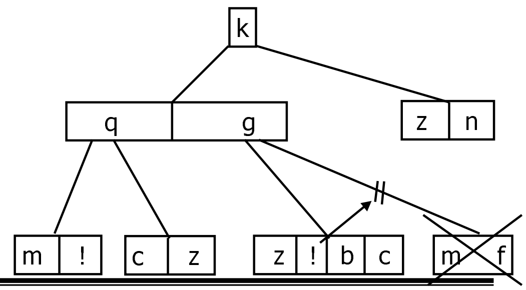
- Another common use of the cut is with fail
- Use to force failure in special cases that are easy to rule out immediately

```
average_taxpayer(X) :-
  lives_in_bermuda(X), !, fail.
average_taxpayer(X) :-
  /* complicated rules here… */
```

Tree view of a cut



Tree view of a cut



Icon

- **Icon is a programming language developed by Ralph Griswold**
  - **Based in part on SNOBOL**
- **A very cool language that (roughly) takes the best of C and of Prolog**
  - **It's very much an imperative language, but with a kick from backtracking**
- **Much of these notes are from an Icon overview**
  - `http://www.cs.arizona.edu/icon/docs/ipd266.htm`

Expression evaluation

- **Expressions in Icon don't (only) return a value**
- **Rather, they succeed (and return a value) or fail**
  - **Ex: find(s1,s2) looks for string s1 in string s2**
  - **If it finds it, it succeeds and returns the index where s2 was found**

Generators

- **What's cool is that many expressions can produce multiple results**
  - `find("i", "mississippi")`
  - **{2,5,8,11}**
- **You can iterate over these results easily**
  - `every j := find("i", "mississippi") do write(j)`
  - `every write(find("i", "mississippi"))`
  - **This "generator" expression stores state and can resume to produce new (additional) results**

Examples

- `every k := i to j do f(k)`
- `every f(i to j)`
- `every write(find("i", s1) | find("i", s2))`
- `every write(find("i",s1|s2))`
- `(i | j | k) = (0 | 1)`

Writing generators

```
procedure findodd(s1, s2)
  every i := find(s1, s2) do
    if i % 2 = 1 then
      suspend i
end
every write(findodd(s1,s2))
```


In some sense, that's it

- In other dimensions, Icon is very much like any imperative language
- But adding this rich notion of expression evaluation is very expressive
  – It's very much like Prolog --- in essence, using unification and resolution --- but in a limited scope rather than for the entire computational model

Other cool things

- String scanning
  – `line ? while tab(upto(&letters)) do write(tab(many(&letters)))`
- Sets
  –
```
words := set()
while line := read() do
   line ? while tab(upto(&letters)) do
      insert(words,
             tab(many(&letters)))
every write(!words)
```


More

- Tables
  –
```
words := table(0)
while line := read() do
   line ? while tab(upto(&letters)) do
      words[tab(many(&letters))] +:= 1
```


Next week: CLP(R)

- Arithmetic only works forward in Prolog
  – But we know arithmetic works both directions
- By adding knowledge of a domain to Prolog-like languages, we can solve lots of richer problems, and solve them faster