AI 08226

Prolog
(PROgramming in LOGic)

Dr Darryl N. Davis

Department of Computer Science
University of Hull

Bibliography

- **Prolog Programming for AI, Ivan Bratko,**
  - **3/e, Addison-Wesley, 2001.**
- The Art of Prolog, Sterling, L. and Shapiro, E.,
  - MIT Press, 1986.
- Mastering Prolog, Rob Lucas
  - UCL Press; ISBN: 1857284003, 1996.
- Programming in PROLOG, W.F. Clocksin, C.S. Mellish
  - Springer-Verlag, ISBN: 3540583505, 1994.
- PROLOG for Students, David H. Callear,
  - Letts Educational Ltd; ISBN: 1858050936, 1994.

Introduction

- Prolog is a declarative language
  - It is not procedural or object-based
  - But can build objects in it
- Prolog works in terms of pattern instantiation
- Prolog does not have global variables
  - Other than predicate (or functor) names
- Programs typically built from facts and rules
  - a fact is a bodiless rule
    - ☞ its_sunny.
    - ☞ weather(sunny).

Prolog

- Programs built from predicates
  - which have the value *true* or *fail (aka false)*
- Prolog makes use of *Closed World Assumption*
  - If predicate name exists and cannot match to it
  - then predicate assumed to be false.
- Prolog concerned with relationships between objects and the truth of these
- Prolog programming concerned
  - with defining relations
  - and querying relations

Predicates and Functors

- Prolog predicates called functors
  - the name of the predicate, **<u>cost</u>**(robot, 2000)
  - functor names are alpha-numeric
    - ☞ begin with lowercase alpha
    - ☞ *test*, *relation*, *test123*, *relation1A* are acceptable
    - ☞ *Test*, *123*, *TEST*, *!£$* are not acceptable
  - Typically predicates relate set of objects
    - ☞ color(robot, green).
    - ☞ cost(robot, affordable).
  - Predicates have 0, 1, 2, 3, etc. arguments
    - ☞ cost.
    - ☞ cost(dosh). cost(spaceship, expensive).

Predicate Functors and Arity

- Predicates have functor and arity
  - No space between predicate name & first parenthesis
  - %color(1)
  - color(red).
  - color(green).
  - %color/2
  - color(robot, red).
  - color(grass, green).
  - %color/3
  - color(robot, red, shiny).
  - color(sea, green, dull).

Terms and variables

- NO such thing as GLOBAL variable
  - nearest is set of predicates as functor/arity combinations
- Terms beginning with Uppercase Alpha characters are variables
  - ☞ cost(Thing, X, Y).
- Tokens starting with lower case characters are not.
  - ☞ cost(robot, X, Y).
  - ☞ cost(amigabot, X, Y).

Use of Prolog

- Calling prolog
  - click on the plwin icon in WindowsNT
  - on unix/solaris use pl (or prolog)
- Loading a file
  - consult(Filename).
  - If all prolog have .pl extensions can do
  - ?- consult(myfile). OR ?- [myfile].
  - If not
  - ?- consult('mybizrrefile.txt').
- Exiting prolog
  - ^d or halt.

Hints

- Which prolog
  - available for free from module webpages
- Use the DOT <CR> to finish prolog sequence
  - does not work otherwise!
- Use the SEMI-COLON for multiple answers
  - when using the interpreter
- use halt. to prolog
  - also ^d
- interrupting programs
  - use ^c

Online Help (html manual available)

- help(<TERM>)
  - ?- help(atom).
    - atom(+Term)
    - Succeeds if Term is bound to an atom.
- apropos(<TERM>)
  - ?- apropos(atom).
    - atom/1 Type check for an atom
    - atomic/1 Type check for primitive
- explain(<TERM>)
  - ?- explain(help).
    - "help" is an atom
    - Referenced from 1-th clause of online_help:help/0

Use Comments in Code

- Use Comments to document your code
- Two forms
  - **% Percent Sign - all to right is comment**
  - **/* In-between these delimiters is a comment */**
- For example:
  - /* File : comments.pl */
  - /* Author: D.N.Davis */
  - /* 11:19 AM 11-Sept-2002 */
  - /* AI 08226 Example for Prolog Lectures */
  - % go/0 - predicate to start the program, it stops Prolog
  - go:- halt.

Prolog-AI-L2 (From Here)

- L1 – Prolog basics and interpreter
- L2 – Writing Prolog Code (Getting it do things)

Prolog Syntax

- Constant
  - Integer: 0, 42, -17
  - Real: 1.07, -0.029, 917.1
  - Atoms: wine, x, x1
  - ‘Fred’, ‘file.pl’
    - ☞ not “fred” - that is a list [102, 114, 101, 100]
- Variable:
  - initial upper case X, Value, V12a, UniqueToClause
  - initial underscore _result, _2
  - anonymous _

Assignment and testing on numbers

- **X is 22+1.**

X = 23

- **?- X is X+1.**

ERROR: Arguments are not sufficiently instantiated

- **?- X = X+1.**

X = ... +... +1+1+1+1+1+1+1+1+1

- **?- X = X = 1.**

ERROR: Syntax error: Operator priority clash

ERROR: X =

ERROR: ** here **

ERROR: X = 1 .

Prolog Syntax

- Operators
  - many built in binary and unary operators
  - Arithmetic Infix { /, *, -, +, // , mod}
    - ?- A is 3+4*7.
    - A = 31
  - Arithmetic Prefix (unary) {truncate, floor, round}
    - ?- A is round(34/31).
    - A = 1
  - Cannot use variable name for more than one value
    - ?- A is 3*5, A is A*5, A is sqrt(A).
  - Need to use differently named variables
    - ?- A1 is 3*5, A2 is A1*5, A3 is sqrt(A2).

Prolog Syntax - Facts

- Relation - a predicate with functor and arity
  - university/0
    - university.
  - cycle/1
    - cycle(‘Not Started’).
    - cycle(1).
  - thing/2
    - thing(agent1, agent).
    - ting(object1, obstacle).
  - distance/3
    - distance(agent1, object1, 10).
    - distance(agent1, agent2, 15).

Example Facts – greek.pl

- **male(cronus). male(zeus). male(hades).**
- **male(ares). male(hermes). male(apollo).**
- **female(hera). female(maia). female(leto).**
- **female(artemis). female(iris).**
- **parent(cronus, zeus). parent(cronus, hades).**
- **parent(zeus, ares). parent(zeus, hermes).**
- **parent(zeus, apollo). parent(zeus, artemis).**
- **parent(zeus, iris). parent(hera, ares).**
- **parent(hera, iris). parent(leto, apollo).**
- **parent(leto, artemis). parent(maia, hermes).**

Loading Files into SWI-Prolog

- ?- [greek].
- % d:/Teaching/AI/Code/GREEK.PL compiled 0.00 sec, 2,336 bytes
- % Accessing the data
- ?- listing(male).
- ?- listing(female).
- ?- listing(parent).
- ?- male(A).
- 2 ?- male(A).
- A = cronus
- 3?- female(A).
- A = hera ;
- A = maia ;
- A = leto ;
- A = artemis ;
- A = iris ;
- No

More on the use of **not**

- female(athene). male(hermes).
- not(male(X)).
- not(female(X)).
- not(male(X)), not(female(X)).
- not((male(X) , female(X))).
- male(X) , female(X).
- not((male(X) ; female(X))).
- male(X) ; female(X).
- % now add
- female(hermaphrodite). male(hermaphrodite).

Negation of Facts

- Logic Unary (Infix)
- not/1, not(predicate).
- If predicate known then negation can be done
  - not(male(hermes)). No
  - not(female(hermes)). Yes
- If predicate not known then negation is an error
  - not(greek(hermes)). Fail – greek/1 undefined
- Can use not to surround more than one clause BUT…
  - ?- not(male(hermes), female(hermes)).
  - ERROR: Undefined procedure: not/2
  - ERROR: However, there are definitions for:
  - ERROR: not/1
  - No
  - 3 ?- not((male(hermes), female(hermes))).
  - Yes

Prolog and logic (in later lectures)

- Prolog based on first order predicate calculus
  - constants, variables, compound terms,
  - not ($\neg$), and ($\wedge$), or ($\vee$),
  - implies ($\Leftarrow$) (($\Rightarrow$)), equivalence ($\Leftrightarrow$)
  - for all ($\forall$), there exists ($\exists$)
- $\exists \bullet \text{woman}(y) \wedge \text{parent}(y, x)$
  $\Leftarrow \forall x \bullet \text{man}(x) \wedge \neg(x = \text{ash})$
- $\exists \bullet \text{woman}(y) \wedge \text{parent}(y, x)$
  $\Leftarrow \forall x \bullet \text{woman}(x) \wedge \neg(x = \text{elm})$

Clauses and Programs

- Unit Clause
  - constant or relation terminated with a fullstop
    - ☞ toolkit_initialised.
    - ☞ perceives(agent1, [agent2, object1]).
    - ☞ knows(agent1, nothing).

- Non-unit Clause
  - constructed from constants and relations using, binary operators, ":-" ","
    - ☞ knows(A, B):- perceives(A, B).
    - ☞ agent(A) :- thing(A, agent).
    - ☞ perceived(A) :- perceives(_wildcard, A).

The use of the Semi-colon - or

- Alternative matches
- **state(wet):-**
- **(fell_in_the_sea ; raining_heavily).**
  - On many occasions this becomes unclear
  - with more complex clauses
  - Alternatively produce multiple clauses (preferable)
- **state(wet):-**
- **fell_in_the_sea.**
- **state(wet):-**
- **raining_heavily.**

Predicates: Multiple Instances Allowed

- Unit Clauses (Facts) % male/1

```
male(apollo).
male(zeus).
```

- Non-Unit Clauses (Rules) % sibling/2

```
sibling(X, Y):-
    father(Z, X),
    father(Z, Y).
sibling(X, Y):-
    mother(Z, X),
    mother(Z, Y).
```

Prolog programs

- Consist of clauses of facts, rules and questions
  - Facts (unit clauses)
    - thing(agent1, agent).
    - agent(agent2, static, up, 55, 67).
  - Rules (non-unit clauses)
    - agent(Name) :- thing(Name, agent).
    - agent(Name):- agent(Name, _state, _direction, _x, _y).
  - Questions (via the interpreter or other code)
    - ?- agent(A).
    - A is agent1.
    - Yes.
- A relation is specified by facts and rules
- A procedure is set of clauses about the same relation

Example Prolog File: program1.pl

- % An example about agent and objects
- thing(agent1, agent). thing(agent2, agent).
- thing(object1, object).
- location(agent1, 10, 10). location(agent2, 15, 35).
- location(object1, 10, 30).
- % Distance/3 evaluates distance between things
- distance(Thing1, Thing2, Distance):-

```
        location(Thing1, X1, Y1),
        location(Thing2, X2, Y2),
        Xdiff is X1-X2, Xsq is Xdiff*Xdiff,
        Ydiff is Y1-Y2, Ysq is Ydiff*Ydiff,
        Distance is sqrt(Xdiff+Ydiff).
```

Querying the program in swi-prolog

```
% PROGRAM1.PL compiled 0.00 sec, 4,700 bytes
?- distance(agent1, agent2, X).
   X = 25.4951
?- distance(A,B,0).
   A = agent1
   B = agent1
?- distance(A,B,C).
   A = agent1
   B = agent1
   C = 0 ;
   A = agent1
   B = agent2
   C = 25.4951 ;
etc.
```

Hints about Debug

- ^c results in the help prompt
  - Action (h for help) ?
  - Typing an 'h' gets us:
    - a: abort b: break
    - c: continue e:
    - g: goals t: trace
  - 'e' s the program immediately
  - 'a' aborts whatever you've type so far
  - 'c' continues where you left off
  - 'h' gets us the 'Action' help menu, again

Program Execution

- Prolog tries to satisfy goals (questions) by
  - Matching the head of a rule
  - Satisfying the body of the rule
    - ☞ (unification is from left to right)
      - success →
    - ☞ Head :- Term$_1$, Term$_2$, …, Term$_n$.
      - ← failure
    - ☞ If a specific instance of a term fails,
    - ☞ Prolog looks for another instance to match
    - ☞ If a term fails, Prolog backtracks and
    - ☞ attempts to re-evaluate previous terms.
    - ☞ If a rule fails
    - ☞ Prolog tries to find another matching head (in sequence)

Example Prolog Execution 0

- % example prolog database using fact/2
- **fact(grass, green).**
- **fact(sky, blue).**
- **fact(sun, yellow).**
- **fact(sea, green).**
- **fact(desert, yellow).**
- % example rules using rule/1
- **rule(X):- fact(X, Color), fact(Y, Color), X \= Y.**
- **rule(X):- fact(T1, X), fact(T2, X), T1 \= T2.**

Example Prolog Execution I

- ?- **rule(A).**
  - Matches against first instance of rule/1
    - ☞ **A** unified to **X**
  - Look to first term of body i.e. **fact(X, Color),**
    - ☞ matches to first instance of fact/2
    - ☞ **X** unified to **grass**, **Color** unified to **green**
  - Look to second term of body i.e. **fact(Y, Color),**
    - ☞ **Color** is instantiated to **green** from first term
    - ☞ matches to first instance of fact/2
    - ☞ **Y** unified to **grass**
  - Look to third term of body, i.e. **X \\= Y.**
    - ☞ both **X** and **Y** unified to **grass**, it fails
    - ☞ backtracks to retry second term

Example Prolog Execution II

- Rematch second term of body i.e. **fact(Y, Color),**
  - **Color** is instantiated to **green** from first term
  - tries to match serially to fact/2
  - matches to fourth instance of fact/2
  - **Y** unified to **sea**
- Look to third term of body, i.e. **X \= Y.**
  - **X** unified to **grass**, **Y** unified to **sea**
  - these are different so comparison succeeds
- Body now fully instantiated
  - **X = grass**
- At this point can press return or type in "**;**"
  - if latter now looks for further matches

Example Prolog Execution III

- Look to first term of body i.e. **fact(X, Color),**
  - ☞ looks to second instance of fact/2
  - ☞ **X** unified to **sky**, **Color** unified to **blue**
- Look to second term of body i.e. **fact(Y, Color),**
  - ☞ **Color** is instantiated to **blue** from first term
  - ☞ matches to second instance of fact/2
  - ☞ **Y** unified to **sky**
- Look to third term of body, i.e. **X \= Y.**
  - ☞ both **X** and **Y** unified to **sky**, it fails
  - ☞ backtracks to retry second term
- Cannot find another match to **fact(VAR, blue)**
  - ☞ backtracks to retry first term

Example Prolog Execution IV

- Look to first term of body i.e. **fact(X, Color),**
  - ☞ looks to third instance of fact/2
  - ☞ **X** unified to sun, **Color** unified to **yellow**
- Look to second term of body i.e. **fact(Y, Color),**
  - ☞ **Color** is instantiated to **yellow** from first term
  - ☞ matches to fifth instance of fact/2
  - ☞ **Y** unified to **desert**
- Look to third term of body, i.e. **X \= Y.**
  - ☞ **X** unified to **sun**, and **Y** unified to **desert**, it succeeds
- Body now fully instantiated
  - ☞ **X = sun**
- At this point can press return or type in "**;**"
  - ☞ if latter now looks for further matches

Example Prolog Execution V

- Look to first term of body i.e. **fact(X, Color),**
  - looks to fourth instance of fact/2
  - **X** unified to **sea**, **Color** unified to **green**
- Look to second term of body i.e. **fact(Y, Color),**
  - **Color** is instantiated to **green** from first term
  - matches to first instance of fact/2
  - **Y** unified to **grass**
- Look to third term of body, i.e. **X \= Y.**
  - **X** unified to **sea** and **Y** unified to **grass**, it succeeds
- Body now fully instantiated
  - **X = sea**
- At this point can press return or type in ";"
  - if latter now looks for further matches

Full Set of Results:

- **d:SEA.PL compiled, 0.11 sec, 1,328 bytes.**
- **?- rule(X).**
- **X = grass ;** % rule/1 (1)
- **X = sun ;** % rule/1 (1)
- **X = sea ;** % rule/1 (1)
- **X = desert ;** % rule/1 (1)
- **X = green ;** % rule/1 (2)
- **X = yellow ;** % rule/1 (2)
- **X = green ;** % rule/1 (2)
- **X = yellow ;** % rule/1 (2)
- **No** % exhausted possibilities
- **?-**

On Prolog Execution

- Prolog explores choices in order
  - backtracking ensures that all possibilities are tried
  - if necessary
- Backtracking is an activity under control of Prolog not the programmer
  - Programmer can force backtracking or stop it
  - using repeat, fail, true and ! (cut) operators
    - ☞ covered in later lectures
- Prolog has no global variables
  - Scope of a variable is a clause

Pictures of program execution

- Trees can be used to represent space
  - of possible solutions to a goal
- Consider:

```
fact(grass, green).
fact(sky, blue).
fact(sea, green).
```

- % example rules using rule/1

```
rule(X):- fact(X, Color), fact(Y, Color), not(X == Y).
```

- with the goal

```
?- rule(A).
```

Tree1

rule(A)

fact(X, Color), fact(Y, Color),not(X==Y).

fact(grass, green),
fact(grass,green),
not(grass==grass)

This path fails - try another

Tree2

fact(X, Color), fact(Y, Color),not(X==Y).

fact(grass, green), fact(Y, green),not(grass==Y).

fact(grass, green),
fact(grass,green),
not(grass==grass)

fact(grass, green),
fact(sky,blue),
not(X==Y)

This path fails too!

Tree3 – first of the succeeds

rule(A)

fact(X, Color), fact(Y, Color),not(X==Y).

fact(grass, green), fact(Y, green),not(grass==Y).

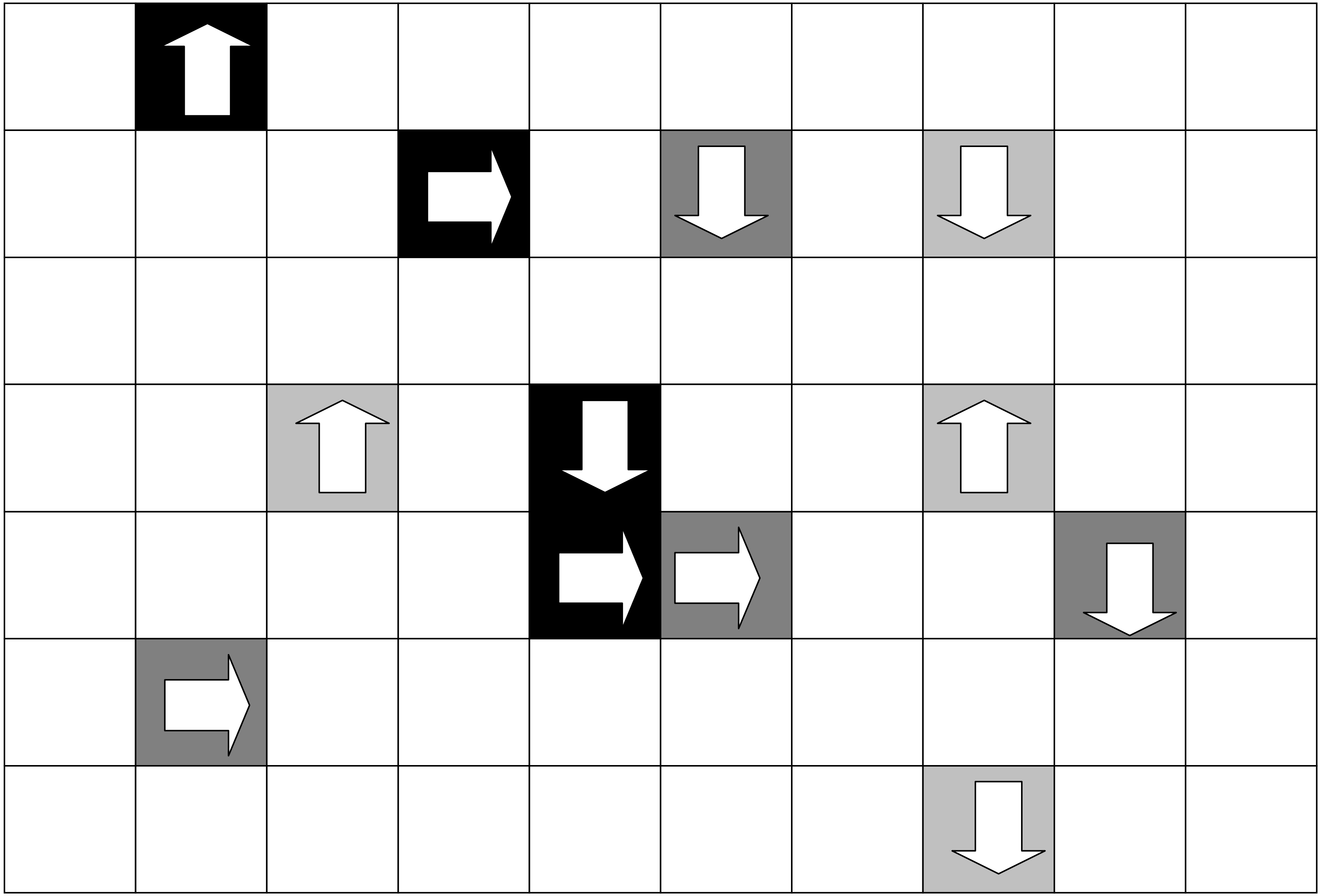
fact(grass, green),
fact(grass,green),
not(grass==grass)

fact(grass, green),
fact(sky,blue),
not(X==Y)

fact(grass, green),
fact(sea, green),
not(grass==sea)

Many other paths, some succeed, most fail!

Agents in a Discrete World: See CWK



Modelling Automata

- A Finite State Machine has a logical structure
  - Can be used for Very Simple Agents
- Defined on its input and internal state
  - `Input: SpaceFree | SpaceBlock`
  - `State: AKSF | AKSB | Static`
  - `Output: Nothing|TurnLeft|MoveAhead`
  - `StateChangeFunction : Input+Internal→Internal`
  - `OutputFunction : Input+Internal →Output`

Specification of FSM as Table

Input	State	Change	Output
SpaceFree	AKSF	AKSF	Ahead
SpaceFree	AKSB	AKSF	Ahead
SpaceFree	Static	AKSF	Nothing
SpaceBlock	AKSF	AKSB	Nothing
SpaceBlock	AKSB	AKSB	TurnLeft
SpaceBlock	Static	AKSB	Nothing

FSM as prolog - state change function


```
%statechange/3
% statechange(input, state, newstate).
statechange(spacefree,aksf,aksf).
statechange(spacefree,aksb,aksf).
statechange(spacefree,static,aksf).
statechange(spaceblock,aksf,aksb).
statechange(spaceblock,aksb, aksb).
statechange(spaceblock,static,aksb).
```

- % many alternative codes

FSM as prolog - output function

```
% output/3
% output(input, state, output).
output(spacefree,aksf,ahead).
output(spacefree,aksb,ahead).
output(spacefree,static,nothing).
output(spaceblock,aksf,nothing).
output(spaceblock,aksb, turnleft).
output(spaceblock,static,nothing).
```

- % many alternative codes

THE Full FSM in prolog : fsm1.pl

```
% fsm/4
% fsm(input, state, newstate, output).
% defined over statechange and output
fsm(Input, State, NewState, Output):-
   statechange(Input, State, NewState),
   output(Input, NewState, Output).
?- [fsm1].
% FSM1.PL compiled 0.00 sec, 1,812 bytes
?- fsm(A,B,C,D).
A = spacefree B = aksf  C = aksf  D = ahead
```

Clause Order

- Matching determines which clauses can be used to solve a goal
- Clauses tried in order
  - sequentially in consulted order
  - left to right in bodies
- If progress halts, backtracking is tried

- Matching (clause instantiation) is depth-first search through prolog clauses

Matching

- General rules for matching two terms S and T;
  - If S and T are constants
  - Then they match only if they are the same entity
    - ☞ **?- 2 == 2.**
    - ☞ **Yes**
    - ☞ **?- a = a.**
    - ☞ **Yes**
    - ☞ **?- 'Term1' == 'Term2'.**
    - ☞ **No**
    - ☞ **?- "String1" = "String1".**
    - ☞ **Yes**

Matching

- General rules for matching two terms S and T;
  - If S is a variable and T is anything
  - Then they match.
    - ☞ **?- S = _var.**
    - ☞ **S = _G147**
    - ☞ **Yes**
    - ☞ **?- S= _.**
    - ☞ **S = _G123**
    - ☞ **Yes**
    - ☞ **?- S = a.**
    - ☞ **S = a**
    - ☞ **Yes**

Matching of Variables

- Order of clauses matters
  - **?- A = B, A = 1, B = 1.**
  - **Yes**
  - **?- A = 1, B = 1, A = B.**
  - **Yes**
  - **?- A = 1, B = 1, A == B.**
  - **Yes**
  - **?- A == B, A = 1, B = 1.**
  - **No**
  - **? A = B, A == B.**
  - **Yes**
  - **So? ?- A == B, A = B. ?- A = B, A == B.**

Matching

- General rules for matching two terms S and T;
  - If S and T are clauses then they match only if
  - S and T have the same principal functor and
  - all their corresponding components match.
    - **?- date(D, M, 2001) = date(D1, january, Y1).**
    - **D = _G339**
    - **M = january**
    - **D1 = _G339**
    - **Y1 = 2001**
    - **Yes**
    - **?- date(D, M, 2001) = date(D1, january, 2020).**
    - **No**

Matching

- Matching in prolog always results in the most general instantiation
  - committing variables to the least possible extent
  - ?- date(D, M, 2001) = date(D1, january, Y1),
  - date(D, M, 2001) = date(1, M, Y).
  - Instantiations

First Goal	Second Goal	Third Goal
D = D1	D = 1	D = 1, D1 = 1
M = january	M = M	M = january
Y1 = 2001	Y = 2001	Y1 = 2000, Y = 2001

  - Consecutive goals leading to more specific values

Unification

- Position in unification is the same as in matching
  - two atoms unify if they are equal
  - one variable and one atom
    - the variable is instantiated to the value of the atom
  - structures unify if:
    - they have the same functor
    - they have the same number of components
    - corresponding components unify
  - E.g. p(X, b) and p(a, Y) unify with {a/X, b/Y}
  - p(X, X) and p(a, b) do not unify
  - p(X, f(Y) and p(Y, f(a)) unify with {a/X, a/Y}

Unification & The *occurs* check

- Unification is indeterminate for two components
  - IF one being a variable and another being a term
  - if the variable appears in the term
- occurs(X):- p(X) = p(f(X)).
  - X = f(f(f(f(f(f(f(f(f(f(...))))))))))
- **Matching**:
- a process that determines whether a clause can be used to solve goal.
  - Many prologs do not include occurs checks
  - If two terms unify they match
  - If two terms match, they may not unify.

Exploiting matching

- Achieve computation by clause application, selection and construction

```
summary(item(book,
              author(pratchett),
              title(the_colour_of_magic),
              code(124753)),
         book(pratchett, the_color_of_magic) ).
opus(O):-
   summary(item(book,author(_A), title(_T),code(_)), O).
?- opus(O).
O = book(pratchett, the_color_of_magic)
```

Testing data

- valid(day, Day):-
- Day > 0, Day < 32.
- valid(month, Month):-
- Month > 0, Month < 13.
- ?- valid(day, 0). no
- ?- valid(month, 0). no
- ?- valid(day, 29). yes
- ?- valid(month, 11). yes
- ?- valid(day, 32). no
- ?- valid(month, 13). no

Procedural programming - I/O

- write(Text) and nl
  - **outputs text (wrapped in single apostrophes)**
  - **?- write('Hi - some meaningless text').**
  - **?- write('Yet some more text'), nl.**
  - careful with the apostrophes, I.e. do not write
  - **write('Freda's piece of text').**
- read(Term) - reads a term from the input.
  - ?- read(Term).
  - |: a.
  - Term = a
  - Yes

Example :Month Conversion program

```
go:-          write('Integers to Names of Months'), nl,
              write('Enter the month as an integer:  '),
              read(Month),
              write('The month is '),
              month(Month), nl.
month(1):-    write('January').
month(2):-    write('Febuary').

month(12):-   write('December').
month(_):-    write('Unknown - Incorrect Entry').
```

- ?- go.
- Integers to Names of Months
- Enter the month as an integer: 9.
- The month is September

- ?- go.
- Integers to Names of Months
- Enter the month as an integer: -1.
- The month is Unknown - Incorrect Entry

- ?- go.
- Integers to Names of Months
- Enter the month as an integer: a.
- The month is Unknown - Incorrect Entry

Arithmetic Program

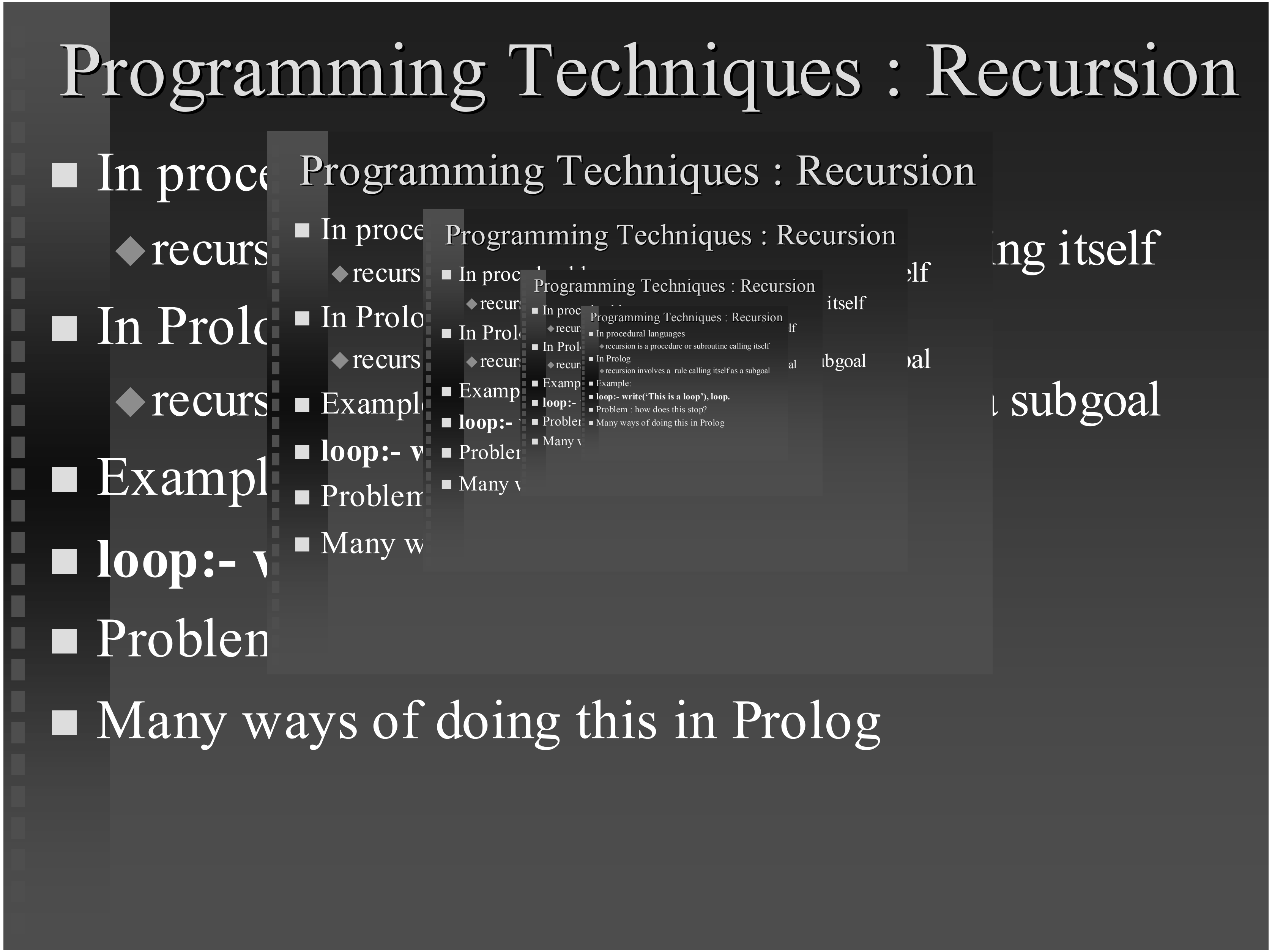
- run:- get_a_number(Num1),
- get_a_number(Num2), nl,
- write('a : Add.'), nl,
- write('b: Subtract.'), nl,
- write('c: Multiply.'), nl,
- write('d: Divide.'), nl,
- write('e: .'), nl,
- write('Your choice: '), read(Choice), nl,
- choice(Num1, Num2, Choice), nl.
- get_a_number(N):- write('Enter a number: '),
- read(N).

- % choice/3 - now perform what was asked with checks
- choice(_,N1, N2):-
-     (not(number(N1)); not(number(N2))),
-     write('Numbers not entered when asked'), nl.
- choice(a, N1, N2):-   X is N1+N2,
-                             write('Answer = '), write(X), nl.
- choice(b,N1, N2):-   X is N1-N2,
-                             write('Answer = '), write(X), nl.
- choice(c,N1, N2):-   X is N1*N2,
-                             write('Answer = '), write(X), nl.
- choice(d,N1, N2):-   X is N1/N2,
-                             write('Answer = '), write(X), nl.
- choice(e,_,_):-   halt.
- choice(_,_,_):-   write('Strange Values Entered'), nl.

Unacceptable Prolog!

- The following will not be accepted by Prolog
- **run:-   Total = 0,**
- **write(‘Enter a number: ‘), read(Num),**
- **Total is Total + Num,**
- **write(‘Enter another number: ‘), read(Num),**
- **Total is Total + Num ,**
- **write(‘Total is ‘), write(Total), nl.**
- A major difference between declarative and procedural programming languages
- While subgoals may appear like instructions
- They remain goals with unification & matching rules
  - ie variable names can be used once in a call

Programming Techniques : Recursion



Programming Techniques : Recursion

- In procedural languages
  - recursion is a procedure or subroutine calling itself
- In Prolog
  - recursion involves a rule calling itself as a subgoal
- Example:
- **loop:- write(‘This is a loop’), loop.**
- Problem : how does this stop?
- Many ways of doing this in Prolog

Conditional Recursion

- loop :-   write('TYPE end TO END'), read(Word),
-                    (Word = end; loop).
- This is equivalent to a REPEAT … UNTIL loop.
- Usual way in Prolog is to place terminating condition before recursive loop
  - Called Head Recursion
- loop(end).
- loop(_):-   write('Type end to END: '),
-                          read(Word), loop(Word).
- Equivalent to a WHILE … DO
  - terminating condition tested at start of the loop

Prolog So Far

- *L1 – Prolog basics and interpreter*
- *L2 – Atoms, Numbers, Arithmetic, Clause Types*
- *L3 – Programs, Matching and Execution*
- *L4 – Modelling Finite State Automata, Matching*
- *L5 – Unification, Numbers, Testing, Procedural*
- **L6 – Recursion and Lists**

Counted Recursion*

- **% Bad Style making use of ";"**
- **loop(N):-   write('The value of N is: '),**
- **write(N), nl, M is N-1, (M = 0; loop(M)).**
- **% Preferred Style**
- **loop(0).**
- **loop(N):-write('The value of N is: '),**
- **write(N), nl, M is N-1, loop(M).**
- **% With extra checks on value of N**
- **loop(0).**
- **loop(N):-   N > -1,**
- **write('The value of N is: '),**
- **write(N), nl,**
- **M is N-1, loop(M).**
- **loop(X):-   write('Loop Error – Undefined Arg: '),**
- **write*X), nl.**

Example of Recursion*

- **factorial(1, 1).**
- **factorial(N, Factorial):- M is N-1,**
- **factorial(M, Factorial1),**
- **Factorial is Factorial1*N.**
- **?- trace, factorial(3,F).**
- **factorial(3, _G352) , _L146 is 3-1 ?, 2 is 3-1 ?**
- **factorial(2, _L147) ?, _L159 is 2-1 ?, 1 is 2-1 ?**
- **factorial(1, _L160) ? factorial(1, 1) ?**
- **_L147 is 1*2 ?, 2 is 1*2 ?, factorial(2, 2) ?,**
- **_G352 is 2*3 ?, 6 is 2*3 ?, factorial(3, 6) ?,**
- **F = 6**

- **power2(0, 1).**
- **power2(N, Result):-   M is N-1,**
- **power2(M, PartialRes),**
- **Result is PartialRes*2.**
- ?- trace, power2(3,R).
- power2(3, _G208) ?   _L133 is 3-1 ? 2 is 3-1 ?
- power2(2, _L134) ?   _L146 is 2-1 ? 1 is 2-1 ?
- power2(1, _L147) ?   _L159 is 1-1 ? 0 is 1-1 ?
- power2(0, _L160) ?   power2(0, 1) ?
- _L147 is 1*2 ?   2 is 1*2 ?   power2(1, 2) ?
- _L134 is 2*2 ?   4 is 2*2 ?   power2(2, 4) ?
- _G208 is 4*2 ?   8 is 4*2 ?   power2(3, 8) ?
- R = 8

Lists in Prolog*

- List: an ordered sequence of elements (of any length)
- Elements of a list can be:
  - constants: integer, real, atom
  - variables
  - facts
  - other lists
  - rules
- Elements of a list are enclosed in square brackets
  - [1, 2, 3] [a, b, c, d]
  - [peach, pear, plum] [peach, 2, b, [3, 5, 7, 9]]
  - [ ] % the empty list

Lists, characters and “strings”

- note:
  - **?- A = “A String”.**
  - **A = [65, 32, 83, 116, 114, 105, 110, 103]**
  - **Yes**
- name/2
  - a standard Prolog predicate that converts atoms into lists of ASCII characters and vice-versa
  - Remember when using this that Atom is not an atom
  - But ‘Atom’ is
  - As is atom, aTom, aTOM etc.

Lists, characters and “strings”

- **?- name(mrRusty, A).**
- **A = [109, 114, 82, 117, 115, 116, 121]**
- **?- name(Atom, [82, 117, 115, 116, 121]).**
- **Atom = 'Rusty'**
- **?- name('Atom', A).**
- **A = [65, 116, 111, 109]**
- **?- name(atom, A).**
- **A = [97, 116, 111, 109]**
- **?- name(Atom,[]).**
- **Atom = ''**

Handling Lists*

- Lists are manipulated by separating the Head from the Tail
- [Head | Tail]
- [1 | [2, 3, 4, 5]]
- [a | [b, c, d]]
- [pear | [peach, plum]]
- The [H | T] notation can be used to build new lists
- ?- Z = [a, b, c, d, e], Z = [H|T].
- Z = [a, b, c, d, e], H is a, T is [b, c, d, e]
- ?- Z = [a, b, c, d], X is [1 | Z]
- X = [1, a, b, c, d]

Handling Lists*

- The [Head | Tail] notation can be used to separate the first few elements of a list from the Tail
- [First, Second | Tail]
  - [1, 2 | [3, 4, 5]]
  - [a, b | [c, d]]
  - [pear, peach | [plum]]
  - [a, b, c | []]
  - ?- [F, S | Tail] = [a, b, c, d, e].
  - F = a
  - S = b
  - Tail = [c, d, e]

Accessing Elements of A List*

- Lists are sequential structures
  - Access is via recursion
  - Remove head, then head from tail, ... , until empty list

- **writelist([]).**
- **writelist([H|Tail]):- write(H), nl, writelist(Tail).**
- **?- trace, writelist([5, 4, 3]).**
- **writelist([5, 4, 3]) ? write(5) ? nl ? writelist([4, 3]) ?**
- **writelist([4, 3])? write(4) ? nl ? writelist([3]) ?**
- **writelist([3])? write(3) ? nl ? writelist([]) ?**
- **writelist([]) ?**

Building a List from Input*

- **buildlist([H|Tail]):-   read(H), not (H = end),**
- **buildlist(Tail).**
- **buildlist([]).**

- **?- buildlist(Alist).**
- : zebadee.
- : dylan.
- : florence.
- : brian.
- : end.
- Alist = [zebadee, dylan, florence, brian]
- Yes

Tracing Buildlist

- **?- trace, buildlist(Alist).**
- **buildlist(_G237) ? read(a) ? not a=end ?**
- **buildlist(_G328) ? read(b) ? not b=end ?**
- **buildlist(_G334) ? read(end) ? not end=end ?**
- **end=end ?**
- **Fail: (11) not end=end ?**
- **Redo: buildlist(_G334) ? buildlist([]) ?**
- **Alist = [a, b]**
- **Yes**
- buildlist works because
  - Alist = [a, b, | []] unifies with Alist = [a, b]

Testing for a list - is_list/1 (built-in)

- **is_list([]).**
- **is_list([_ | Tail]):- is_list(Tail).**
  - ?- is_list(a). No
  - ?- is_list("a"). Yes
  - ?- is_list([a, b, c, d]). Yes
  - ?- is_list([a | b, c, d]). NO!
  - ?- is_list([a | [b, c, d]]). Yes
  - ?- is_list([fred | freda]). Yes
  - ?- is_list([atom|List]). NO!

Useful List Predicates*

- Reversing a list - reverse/2 (built-in)
  - List elements transferred from front of first list to head
  - of second list
  - When first list is empty, the second list is the final list

  - **reverse(L1, L2):- rev(L1, [], L2).**
  - **rev([], L, L).**
  - **rev([H | T], L2, L3):- rev(T, [H | L2], L3).**

  - **?- reverse([1, 2, 3, 4], L).**
  - **L = [4, 3, 2, 1]**

Useful List Predicates*

- Testing for list membership - member/2 (built-in)
  - Predicate succeeds when element matches to head of list
  - Otherwise recursively calls itself with the tail of the list until the list is empty

  - **member(X, [X | _var]).**
  - **member(X, [_var | T]):- member(X, T).**

  - **?- member(heron, [bat, heron, owl]).**
  - **Yes**

Useful List Predicates*

- Appending to lists - append/3 (built-in)
  - Predicate appends list L2 onto end of list L1 to give L3

  - **append([ ] , L, L).**
  - **append([H | L1], L2, [H | L3]):-**
  - **append(L1, L2, L3).**

  - **?- append([1, 2], [3, 4], L).**
  - **L = [1, 2, 3, 4]**

Useful List Predicates

- Appending to lists - reverse/2 (built-in)
  - Predicate reverses list order from L1 to L2 or inverse

```
% reverse/2 makes use of reverse/3
reverse( [ ] , [ ]).
reverse( [H | L],  Reverse):-
            reverse(L, [H], Reverse).
reverse( [ ], Reverse, Reverse).
reverse([H | L], L2, Reverse):-
            reverse(L, [H | L2], Reverse).
```

Useful List Predicates*

- Delete item from lists - delete/3 (built-in)
  - ☞ Takes list and deletes all instances of item to build second list
    - **delete([ ] , _var , [ ]).**
    - **delete([X | L1] , X, L2):-   delete(L1, X, L2).**
    - **delete([H | L1] , X, [H | L2]):- delete(L1, X, L2).**
    - ?- delete([a, b, c], a, L).
    - L = [b, c]
    - ?- delete([a, b, c], [a], L).
    - L = [a, b, c]
    - ?- delete([a, b, c, a], a, L).
    - L = [b, c]

Deleting the elements of a list

- A different delete predicate
  - **delete1(Item, [Item|Tail], Tail).**
  - **delete1(Item, [H | Tail], [H | List]):-**
  - **delete1(Item, Tail, List).**
  - ?- delete1(a, [a, b, c], L).
  - L = [b, c]
  - ?- delete1(I, [a, b, c], L).
  - I = a   L = [b, c] ;
  - I = b   L = [a, c] ;
  - I = c   L = [a, b] ;
  - No

Permuting the elements of a list

```
permutation([ ], [ ]).
permutation(List, [ H | Tail ]):-
        delete1(H, List, Rest),
        permutation( Rest, Tail ).
```

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

- ?- trace, permutation([a, b, c], L).
- delete1(a, [a, b, c], [b, c]) ? permutation([b, c], [b, c]) ?
- L = [a, b, c] ;
- delete1(c, [b, c], [b]) ? creep permutation([b], [b]) ?
- permutation([b, c], [c, b]) ?
- permutation([a, b, c], [a, c, b]) ?
- L = [a, c, b] ;
- delete1(b, [a, b, c], [a, c]) ? permutation([a, c], [a, c]) ?
- L = [b, a, c] ;
- delete1(c, [a, c], [a]) ? permutation([a], [a]) ?
- permutation([a, c], [c, a]) ?
- permutation([a, b, c], [b, c, a]) ?
- L = [b, c, a]

Built-In List Predicates*

- =../2 ``Univ." Term to list conversion
- atom_chars/2 Convert between atom and list of ASCII values
- name/2 Convert between atom and list of ASCII characters
- concat_atom/2 Append a list of atoms
- concat_atom/3 Append a list of atoms with separator
- string_to_list/2 Conversion between string and list of ASCII
- is_list/1 Type check for a list
- proper_list/1 Type check for list
- append/3 Concatenate lists
- member/2 Element is member of a list
- delete/3 Delete all matching members from a list
- select/3 Select element of a list

More Built-in List Predicates*

■ nth0/3	N-th element of a list (0-based)
■ nth1/3	N-th element of a list (1-based)
■ last/2	Last element of a list
■ reverse/2	Inverse the order of the elements in a list
■ flatten/2	Transform nested list into flat list
■ length/2	Length of a list
■ merge/3	Merge two sorted lists
■ list_to_set/2	Remove duplicates
■ sort/2	Sort elements in a list
■ checklist/2	Invoke goal on all members of a list
■ maplist/3	Transform all elements of a list
■ sublist/3	Determine elements that meet condition

Asserting Predicates

- assert/1 - assert(?Term)
  - adds term to predicate database
  - **?- listing(fish).**
  - **fish(tuna, big).**
  - **fish(shark, big).**
  - **?- assert(fish(guppy, small)).**
  - **?- listing(fish).**
  - **fish(tuna, big).**
  - **fish(shark, big).**
  - **fish(guppy, small).**
  - Adds asserted term to end of database of predicates

Asserting Predicates

- assert/1 - will duplicate term in predicate database
  - **?- listing(fish).**
  - **fish(tuna, big).**
  - **fish(shark, big).**
  - **?- assert(fish(tuna, big)).**
  - **?- listing(fish).**
  - **fish(tuna, big).**
  - **fish(shark, big).**
  - **fish(tuna, big).**
  - Adds asserted term to end of database of predicates

Asserting Predicates

- assert/1 - can assert rules too
  - **?- listing(big).**
  - **[WARNING: No predicates for `big']**
  - **No**
  - **?- assert(big(X):- fish(X, big)).**
  - **X = _G279**
  - **?- listing(big).**
  - **big(A) :- fish(A, big).**
  - **?- big(F).**
  - **F = tuna ;**
  - **F = shark**

Static and Dynamic Clauses

- Cannot assert and retract functor/arity
- IF static clause

```
fact( apple, fruit, edible).
fact( jackfruit, fruit, disgusting ).
```

- fact/3 cannot be asserted or retracted

- Have to make dynamic using HEAD-Less Rule

```
:- dynamic( fact/3).
```

- Then the above code
- Can now retract or assert fact/3

Asserting Predicates - Variations

- ?- apropos(assert).
- assert/1 Add a clause to the database
- asserta/1 Add a clause to the database (first)
- assertz/1 Add a clause to the database (last)
- ?- listing(fish).
- fish(guppy, small).
- fish(tuna, big).
- ?- asserta(fish(shark, big)).
- ?- listing(fish).
- fish(shark, big).
- fish(guppy, small).
- fish(tuna, big).

Retracting Predicates

- retract/1 - retract(?Term)
  - deletes term to predicate database
  - **?- listing(fish).**
  - **fish(tuna, big).**
  - **fish(shark, big).**
  - **?- retract(fish(shark, big)).**
  - **?- listing(fish).**
  - **fish(tuna, big).**
  - **?- retract(fish(tuna, _)).**
  - **?- listing(fish).**
  - **Yes**

Retracting Predicates

- Can give different levels of specificity
  - **?- listing(fish).**
  - **fish(tuna, big).**
  - **fish(shark, big).**
  - **fish(marlin, big).**
  - **?- retract(fish(shark, _)).**
  - retracts first instance matching to fish(shark, _)
  - **?- retract(fish(_, _)).**
  - retracts first instance matching to fish(_, _)
  - **?- listing(fish).**
  - **fish(marlin, big).**

Retracting Predicates

- Can use retract to delete rules too
  - retract((big(X):- fish(X, big))).
  - X = _G291
  - ?- listing(big).
  - Yes

- Retractall/1 - remove all instances matching term
  - ?- retractall(fish(_,_)).
  - **?- listing(fish).**
  - **Yes**

Does a clause exist?

- clause/2
- clause(?Head, ?Body)
  - Succeeds when Head can be unified with a clause head and Body with the corresponding clause body.
  - Gives alternative clauses on backtracking.
  - For facts Body is unified with the atom true.
  - Normally clause/2 is used to find clause definitions for a predicate, but it can also be used to find clause heads for some body template.

Does a clause exist?

- ?- listing.
- **big(X):- fish(X, big) .**
- **fish(tuna, big).**
- **fish(marlin, big).**
- **?- clause(fish(F, S), Body).**
- F = tuna
- S = big
- Body = true ;

- F = marlin
- S = big
- Body = true ;

Does a clause exist?

- ?- listing.
- **big(X):- fish(X, big) .**
- **fish(tuna, big).**
- **fish(marlin, big).**

- What Rules Exist with given Body

- ?- clause(Head, fish(F,S)).
- Head = big(_G237)
- F = _G237
- S = big ;
- No

Combining clause and retract

- ?- listing.
- **big(X):- fish(X, big) .**
- **fish(tuna, big).**
- **fish(marlin, big).**

- If rule with fish/2 as body exists retract it

- ?- clause(Head, fish(F,S)), retract(Head:- fish(F, S)).
- Head = big(_G237)
- F = _G237
- S = big ;
- No

Finding all occurrences of a predicate

- findall/3 - built in predicate
  - findall(+Var, +Goal, -Bag)
  - Creates a list of the instantiations Var gets successively on backtracking over Goal and unifies the result with Bag. Succeeds with an empty list if Goal has no solutions.
    - ☞ Also see: help(setof), help(bagof)
  - **fish(shark, big). fish(guppy, small). fish(tuna, big).**
  - **?- findall(Fish, fish(Fish, _), Fishes).**
  - **Fish = _G315**
  - **Fishes = [shark, guppy, tuna]**

More on Findall

- **?- assert(fish(shark, big)).**
- **?- assert(fish(tuna, big)).**
- **?- assert(fish(guppy, small)).**
- **?- findall(Fish, fish(Fish, big), Fishes).**
- **Fish = _G327**
- **Fishes = [shark, tuna]**
- Now add further fact
- **?- assert(fish(shark, big)).**
- **?- findall(Fish, fish(Fish, big), Fishes).**
- **Fish = _G327**
- **Fishes = [shark, tuna, shark]**

Compound Terms - Findall

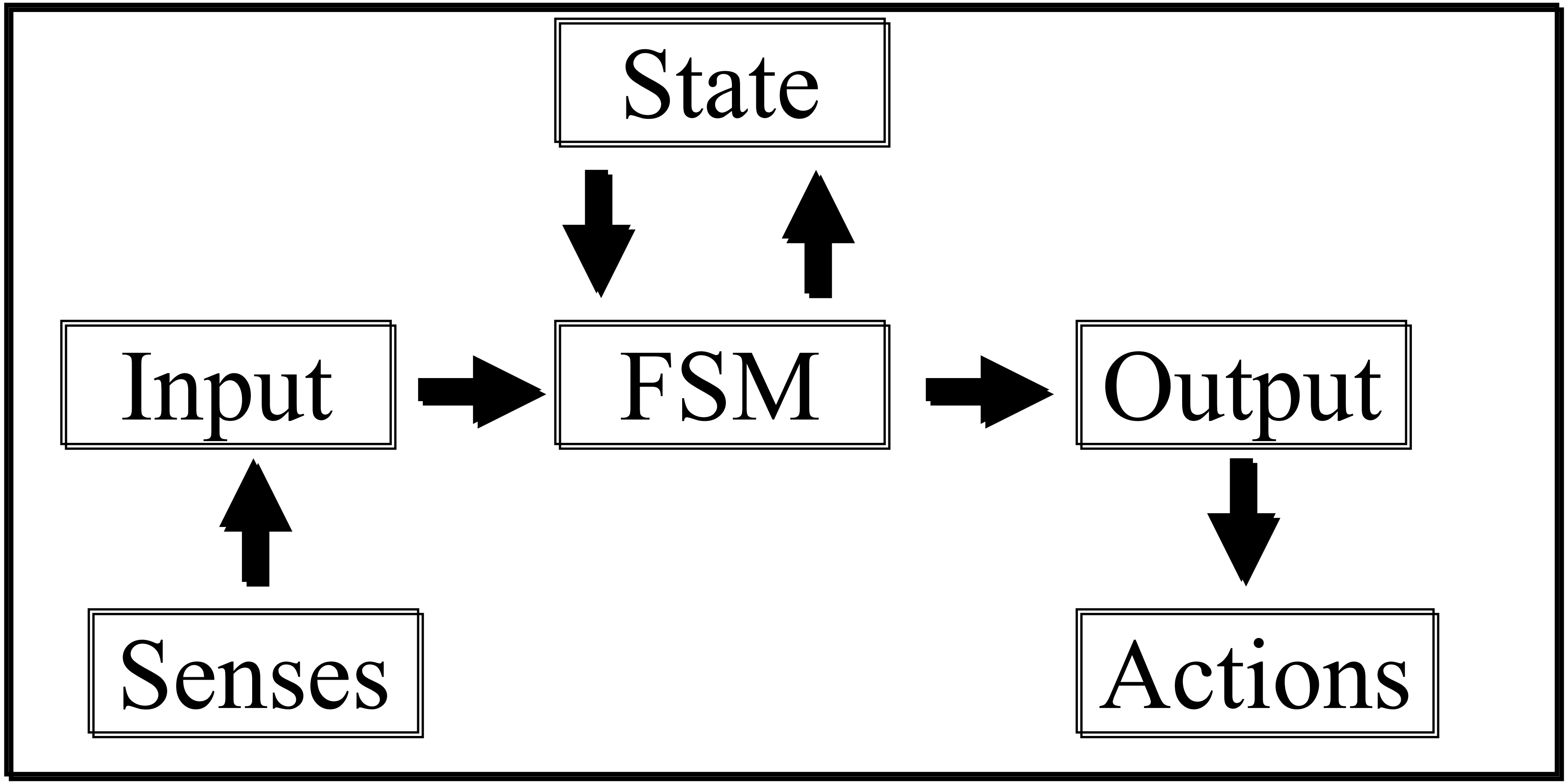
- Compund terms can be used to associate items
  - ?- (A:B) = (atom1:atom2).
  - A = atom1
  - B = atom2
- Can be used to pull items together in lists
- Multiple joining characters permitted
  - atom1:atom2, atom1-atom2, at1+at2, at1/at2 etc
- Compare: question of programming preference
  - [[bert, french, singer], [jake, dutch, dj]]
  - [bert-french-singer, jake-dutch-dj]

Compound Terms via Findall

- **fish(shark, big)**
- **fish(tuna, big)**
- **fish(guppy, small)**
- **?- findall(Fish-Size, fish(Fish, Size), AList).**
- **Fish = _G327**
- **Size = _G329**
- **AList = [shark-big, tuna-big, guppy-small]**
- ?- findall([A, B], fish(A, B), As).
- A = _G157
- B = _G160
- As = [[shark, big], [guppy, small], [tuna, big]]

Using a FSM to Control a simple agent

- logical structure



Prolog for Senses for FSM-Agent


```
% FSM Controller Agent
% agent/5 - agent(Name, X, Y, State, Direction).
% object/3 - object(Name, X, Y).
% define thing over agent/5 and object/3
thing(X):- agent(X, _v2, _v3, _v4, _v5).
thing(X):- object(X, _v2, _v3).
% define sense/2 over all things other than agent
senses(Agent, Things):-
   agent(Agent, _v2, _v3, _v4, _v5),
   findall(X, (thing(X), not(X==Agent)), Things).
```

```
input(_Agent, [], spacefree):- !.
input(Agent, [Thing | _rest], spaceblock):-
    spaceblock(Agent, Thing), !.
input(Agent, [_head|Rest], Input):-
    input(Agent, Rest, Input).
spaceblock(Agent, Thing):-
    agent(Agent,Xagent, Yagent,_state, Direction),
    object(Thing, Xobject, Yobject),
    block(Xagent, Yagent, Direction, Xobject, Yobject).
spaceblock(Agent, Thing):-
    agent(Agent,Xagent, Yagent,_state, Direction),
    agent(Thing, Xobject, Yobject, _state2, _direction),
    block(Xagent, Yagent, Direction, Xobject, Yobject).
```

When is a thing a block?

```
%  block/5
% block(Xagent, Yagent, Direction, Xobject, Yobject).
% Only need to define spaceblock cases
% Top left is origin in this world
block(X, Y1, up, X, Y2):- Y2 is Y1-1, !.
block(X, Y1, down, X, Y2):- Y2 is Y1+1, !.
block(X1, Y, left, X2, Y):- X2 is X1-1, !.
block(X1, Y, right, X2, Y):- X2 is X1-1, !.
```

Mapping Output to Actions

```
%  actions/5
% actions(Direction, Output, X,Y, NX, NY,NewDirection),
actions(up, ahead, X, Y, X, NY, up):- NY is Y-1.
actions(right, ahead, X, Y, NX, Y, right):- NX is X+1.
actions(down, ahead, X, Y, X, NY, down):- NY is Y+1.
actions(left, ahead, X, Y, NX, Y, left):- NX is X-1.
actions(up, turnleft, X, Y, X, Y, left):- !.
actions(right, turnleft, X, Y, X, Y, up):- !.
actions(down, turnleft, X, Y, X, Y, right):- !.
actions(left, turnleft, X, Y, X, Y, down):- !.
actions(Direction, nothing, X, Y, X, Y, Direction):- !.
```

How is an agent run?

```
% run_agent/1
run_agent(Agent):-
    agent(Agent, X, Y, State, Direction),
    senses(Agent, Things),
    input(Agent, Things, Input),
    fsm(Input, State, NewState, Output),
    actions(Direction, Output, X,Y, NX, NY,NewDirection),
    retract( agent(Agent, X, Y, State, Direction) ),
    assert( agent(Agent, NewX, NewY,
                    NewState, NewDirection)).
```

Full FSM-Agent Execution

FSM1.PL compiled 0.00 sec, 8,128 bytes

?- assert(agent(a1, 10,19, aksf, up)).

?- assert(agent(a2,10,20,aksf,up)).

?- run_agent(a2), listing(agent).

agent(a2, 10, 20, aksb, up).

?- run_agent(a2), listing(agent).

agent(a2, 10, 20, aksb, up).

?- run_agent(a2), listing(agent).

agent(a2, 9, 20, aksf, left).

Compound Terms

- Can use simple terms as arguments to predicates
  - ?- assert(predicate(arg1, arg2, 8)).
  - ?- predicate(A,B,C), atom(A), atom(B), integer(C).
  - Yes
  - ?- predicate(A,B,C), atom(A), atom(B), atom(C).
  - No
- Can use simple terms as list elements
  - ?- assert(predicate([a, b, c, d])).
  - ?- predicate([H|R]), atom(H).
  - Yes

Compound Terms As Associations

- However in some situations want to associate
- For example, sense - associate distance and object
  - assert(sense(agent1, [agent2, object1], [25, 50])).
  - is one way but need to manipulate both lists
  - output(_ag, [ ], [ ]).
  - output(Agent, [H1|R1], [H2|R2]):-
  - write(‘Distance from ‘), write(Agent), write(‘ to ‘),
  - write(H1), write(‘ is ‘), write(H2), nl,
  - output(Agent, R1, R2).
    - ?- sense(A,B,C), output(A,B,C).
    - Distance from agent1 to agent2 is 25
    - Distance from agent1 to object1 is 50

Alternative is to use compound terms

- Advantage is sort only one list
  - assert(sense(agent1, [(agent2, 15), (object1,5)])).
  - But need to manipulate compound term
  - output(_ag, [ ]).
  - output(Agent, [(H1,H2)|R]):-
  - write(‘Distance from ‘), write(Agent), write(‘ to ‘),
  - write(H1), write(‘ is ‘), write(H2), nl,
  - output(Agent, R).
    - ?- sense(A,B), output(A,B).
    - Distance from agent1 to agent2 is 15
    - Distance from agent1 to object1 is 5
  - See file: compound.pl

Alternative: Lists as compound terms

- Syntax of Compound terms up to user
  - assert(sense(agent1, [[agent2, 15], [object1,5]])).
  - Need to manipulate nested list compound term
  - output(_ag, [ ]).
  - output(Agent, [[H1,H2]|R]):-
  - write(‘Distance from ‘), write(Agent), write(‘ to ‘),
  - write(H1), write(‘ is ‘), write(H2), nl,
  - output(Agent, R).
    - ?- sense(A,B), output(A,B).
    - Distance from agent1 to agent2 is 15
    - Distance from agent1 to object1 is 5

Alternative: Structure it yourself

- Syntax of Compound terms up to user
  - assert(sense(agent1, [agent2-5, object1-15])).
  - Need to manipulate this form of compound term
  - output(_ag, [ ]).
  - output(Agent, [H1-H2|R]):-
  - write('Distance from '), write(Agent), write(' to '),
  - write(H1), write(' is '), write(H2), nl,
  - output(Agent, R).
    - ?- sense(A,B), output(A,B).
    - Distance from agent1 to agent2 is 5
    - Distance from agent1 to object1 is 15
- Note some constructors NOT allowed

Programming Practice

- Write a program to determine number of occurrences of given item in a list

```
frequency(_, [ ], 0).
frequency(Item, [ Item | Rest ], N):-
    frequency(Item, Rest, N1),
    N is N1 +1.
frequency(Item, [ H | Rest ], N):-
    not( Item = H ),                % An optional line?
    frequency(Item, Rest, N).
```

- ?- frequency(A, [a, b, c, a], N).
- A = a N = 2

Tracing Frequency

- trace, frequency(b, [a, b, c, b], N).
- not b=a ?   frequency(b, [b, c, b], _G329) ?
- frequency(b, [c, b], _L149) ?
- not b=c ? frequency(b, [b], _L149) ?
- frequency(b, [], _L177) ?   frequency(b, [], 0) ?
- 1 is 0+1 ?   frequency(b, [b], 1) ?
- frequency(b, [c, b], 1) ?
- 2 is 1+1 ?   frequency(b, [b, c, b], 2) ?
- frequency(b, [a, b, c, b], 2) ?
- N = 2;
- No

Variation1 on Frequency


```
frequency(_term, [ ], 0).
frequency(Item, [ Item | Rest ], N):-
    frequency(Item, Rest, N1),
    N is N1 +1.
frequency(Item, [ _term | Rest ], N):-
    frequency(Item, Rest, N).
```

- ?- frequency(a,[a, b, a, c], N).
- N = 2 ;
- N = 1 ;
- N = 1 ;
- N = 0 ;
- No

Variation 2 on Frequency with Cut (!)

```
frequency(_term, [ ], 0).
frequency(Item, [ Item | Rest ], N):-
    !, frequency(Item, Rest, N1),
    N is N1 +1.
frequency(Item, [ _term | Rest ], N):-
    frequency(Item, Rest, N).
```

- ?- frequency(a,[a, b, a, c], N).
- N = 2 ;
- No
- ?- frequency(A,[a, b, a, c], N).
- A = a   N = 2 ;
- No

Controlling Backtracking - the Cut !

- The cut ! is an operator that halts backtracking
  - The matching to the left cannot be undone for a goal

```
?- trace, frequency(a,[a, b, a, c], N).
N = 2 ;
Redo:  ( 12) frequency(a, [], _L162) ? creep
Fail:  ( 12) frequency(a, [], _L162) ? creep
Fail:  ( 11) frequency(a, [c], _L162) ? creep
Fail:  ( 10) frequency(a, [a, c], _L135) ? creep
Fail:  (  9) frequency(a, [b, a, c], _L135) ? creep
Fail:  (  8) frequency(a, [a, b, a, c], _G323) ? creep
No
```

Goals and Cut

- Suppose rule **p** with subgoals **q, r, s, t**
  - Subgoal **s** has subgoals **a, b, c, d**

    p:- q, r, s, t.

    s:- a, b, !, c, d.
  - Suppose **s** succeeds as far as **d**, which fails
  - backtracking among **c, d** until they all fail
  - unable to pass cut and because rule **s** has not failed as a whole yet
  - Prolog moves to **s** in the parent rule, then **r**
  - if **r** succeeds a second time, execution moves forward
  - and **s** is tried afresh (from the top of the database)

Repetition using the cut

- Using **repeat/0** - a standard Prolog predicate
  - Always succeed, provide an infinite number of choice points.

```
input1:- read(X),
    write('You typed: '), write(X), nl, X = end.
loop1:- repeat, input1.

input2:- read(X),
    write('You typed: '), write(X), nl, !, X = end.
loop2:-   repeat, input2.
```

  - loop2 (the cut) has the advantage of continuing with no stack problems - data is thrown away on lhs of cut

Running the two versions

```
?- input1.
   : hi.
   You typed: hi
  No
?- loop1.
  : hi.
  You typed: hi
  : hi2.
  You typed: hi2
  : end.
  You typed: end
Yes
```

```
?- input1.
   : end.
   You typed: end
  Yes
?- loop2.
  :hi.
  You typed: hi
  :hi2.
  You typed: hi2.
  :end.
  You typed: end.
 Yes
```

Mutually Exclusive Rules

- If cuts are placed at the end of a clause
  - Whether unit or non-unit, it succeeds at most just once
- Useful for one and only one match required

```
fish(marlin, big).
fish(sailfish, big).
% same_rule/2 no cut
same_rule(1, F):- fish(F, big).
same_rule(2, F):- fish(F, big).
% cut_rule - same as above but with terminating cut
cut_rule(1, F):- fish(F, big), !.
cut_rule(2, F):- fish(F, big), !.
```

So what happens?

- ?- same_rule(X,Y).
- X = 1 Y = marlin ;
- X = 1 Y = sailfish ;
- X = 2 Y = marlin ;
- X = 2 Y = sailfish ;
- No

- ?- cut_rule(X,Y).
- X = 1 Y = marlin ;
- No

Forcing new solutions & using cut

```
member1(X, [X|_]).
member1(X, [ _term |T]):- member1(X, T).
```

- member1 will succeed for every occurrence of X in the list

```
member2(X, [X| _term ]):- !.
member2(X, [ _term |T]):- member2(X, T).
```

- member2 will succeed just the once if X in list at least once or more

Run Time

- member1 will succeed four times here

```
?- member1(a, [a, b, c, a, a, a, b]), write('hi'), nl, fail.
hi
hi
hi
hi
No
```

- member2 will succeed just the once

```
?- member2(a, [a, b, c, a, a, a, b]), write('hi'), nl, fail.
hi
No
```

The use of cut in negation

- Represent “Mary likes all animals but snakes”

```
animal(snake).
animal(tiger).
likes(mary, snake):- !, fail.
likes(mary, X):- animal(X).
```

```
?- likes(mary,tiger).
Yes
?- likes(mary,snake).
No
?- likes(A,B).
No
```

The use of cut in negation - version2

- Represent “Mary likes all animals but snakes”

animal(snake).

animal(tiger).

likes(mary, X):- animal(X), not(X=snake).

```
?- likes(A,B).
A = mary
B = tiger ;
No
?- likes(mary,snake).
No
```

Negation as failure

- Defining the relation different(X, Y)
- Different as in:
  - X and Y are not literally the same
  - X and Y do not match
  - the values of arithmetic expressions X and Y differ
- Take Second Definition

```
different(X, X):- !, fail.
different(X, Y).
```

```
?- different(a, b).          ?- different(a, a).
   Yes                          No
```

Negation as failure

- **?- animal(X), animal(Y), different(X, Y).**

  X = snake   Y = tiger ;

  X = tiger   Y = snake ;

  No

- Alternative definition for different/2

  different(X, Y):-   not(X = Y).

Principles of Good Programming

- Correctness
  - Above all a program should be correct
- Efficiency
  - Good programs do not needlessly waste computer time and memory space
- Transparency (readability)
  - Avoid tricks that obscure the meaning of a program

Principles of Good Programming

- Modifiability
  - Good programs should be easy to modify and extend
- Robustness
  - Programs should not crash immediately the user enters incorrect or unexpected data
- Documentation
  - A minimal documentation program comments and header

Guidelines for achieving above

- *THINK* about the problem to be solved.
  - When the problem is understood and solution well thought through, the actual coding is faster and easier
- Use *STEPWISE REFINEMENT*.
  - From top-level to bottom-level solution.
  - Each refinement step should intellectually manageable
    - i.e. small and clear enough
- Programming is CREATIVE, especially for beginners new to specific programming concepts
  - With experience becomes less of an art and more of a craft. Ideas from similar problems we already know of.

Thinking about prolog programs

- Ontological Delineation
  - How do we find ideas for reducing problems to more easier subproblems?
  - How do we find proper subproblems?
  - Problem P solved by solving $\{P_1, P_2, \ldots P_i, \ldots P_n\}$
- Aspects to consider
  - Recursion
  - Generalisation
  - Structuralisation
  - Pictures

Use of Recursion

- Principle is to split the given problem into cases
- Two groups
  - 1. Trivial or *boundary* cases
  - 2. General cases whose solution is constructed from solutions of simpler versions of the problem
- Recursion applies so naturally to prolog
  - Basic representation (lists) have recursive structure
  - A list is either empty
    - *boundary* case
  - Or has a head and a tail that is itself a list
    - general case

Consider intersection of sets problem

- **intersection(Set1, Set2, Result).**
- This can be split into two cases:
  - Boundary case: **Set1 = [ ] or Set2 = []**
    - ☞ If **Set1 = [ ]** then **Result = [ ]**, regardless of **Set2**.
  - General Case: **Set1 = [Head | Tail]**
  - To transform a list of this form do:
    - ☞ Transform head of the list
      - If **Head member** of **Set2** then **Result** $\rightarrow$ **[Head | Result]**
    - ☞ Then recursively call with **Tail**
      - **Tail** $\rightarrow$ **[NewHead | NewTail]**
    - ☞ Eventually problem reduces to <u>*Boundary Case*</u> **[ ]**

Testing your code: Set UNION

- Set union - sets can be represented as lists
  - $\{ a, b, c, d \} \cup \{ c, d, e \} \Rightarrow \{ a, b, c, d, e \}$

```
% Base case for recursion
setunion( [ ], X, X).
% Recurse without Head if member of L1
setunion( [H | T ], L1, L2):-
            member(H, L1),
            setunion(T, L1, L2).
% Otherwise Recurse with Head
setunion( [ H | T ], L1, [ H | L2 ] ):-
            setunion( T, L1, L2).
```

Test cases: Set UNION without not

setunion([],[a],L).

L = [a] ;

No

?- setunion([a],[],L).

L = [a]

Yes

?- setunion([a],[a],L).

L = [a] ;

L = [a, a] ;

No

Here is a problem

So Use CUT - Set UNION with cut

```
setunion( [ ], X, X).
% Add cut so cannot backtrack
setunion( [H | T ], L1, L2):-
        member(H, L1),
        !,
        setunion(T, L1, L2).
setunion( [ H | T ], L1, [ H | L2 ] ):-
        setunion( T, L1, L2).
```

- Goal only succeeds the once

Testing setunion with The CUT

```
?- setunion([a],[],L).
L = [a] ;
No
?- setunion([a],[a],L).
L = [a] ;
No
?- setunion([b, a],[a, b],L).
L = [a, b] ;
No
?- setunion([b, a],[a, b, c, d],L).
L = [a, b, c, d]
Yes
```

Set UNION using not

```
setunion( [ ], X, X).
setunion( [ H | T ], L1, [ H | L2 ] ):-
            not(member(H, L1)),
            setunion( T, L1, L2).
setunion( [_H | T ], L1, L2):-
            setunion(T, L1, L2).
```

- Goal succeeds just the once
- ?- setunion([a, b, c], [a, d, e], L).

  L = [b, c, a, d, e] ;

  No

Set Intersection using not

- Set intersection - sets can be represented as lists
  - $\{ a, b, c, d \} \cap \{ c, d, e \} \Rightarrow \{ c, d \}$

```
intersect( [ ], X, [ ]).
intersect( [H | T ], L1, [ H | L2 ] ):-
            member(H, L1),  delete(L1, H, L3),
            intersect(T, L3, L2).
intersect( [ H | T ], L1,  L2 ):-
            not(member(H, L1)), intersect( T, L1, L2).
```

  - ?- intersect([a, b, c], [a, c, d, e], L).

    L = [a, c] ;

    No

Set Difference using not

- Set intersection - sets can be represented as lists
  - { a, b, c, d } ⊗ { c, d, e } ⇒ { a, b, e }

```
difference( [ ], X, X).
difference( [H | T ], L1, L2  ):-
            member(H, L1),  delete(L1, H, L3),
            difference(T, L3, L2).
difference( [ H | T ], L1,  [H | L2] ):-
            not member(H, L1), difference( T, L1, L2).
```

  - ?- difference([a, b, c], [a, c, d, e], L).

    L = [b, d, e] ;

    No

Generalisation

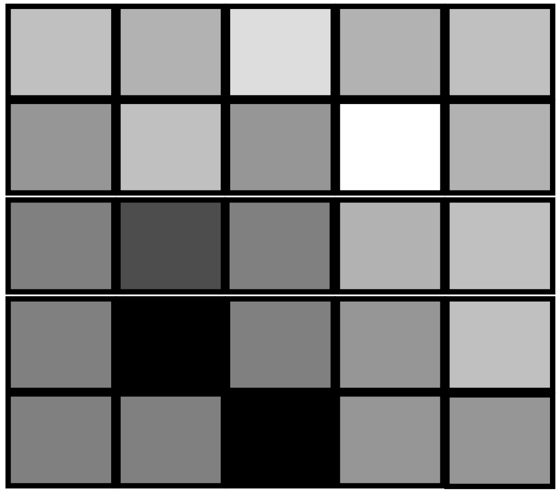
- The idea is to generalise a problem such that its solution can be formulated recursively
- Generalisation of a relation involves the introduction of one or more extra arguments
- Paradoxically, for some problems, the generalised version of the problem is easier to solve
  - See Bratko and the Eight Queens [pp 103]
- Add constraints on general problem so it becomes
- The original problem

Structuralisation

- What are the objects and relations of the problem
- Try drawing figures of the problem or aspects of it
  - These can map onto semantic networks
  - Semantic networks can map onto logic and then prolog
- As prolog is logic based
  - So try and represent them using predicate logic
  - Where this breaks down analyse why
    - Is a more trivial or general case of the problem easier?
    - Solve that and modify solution to fit
  - Do we need lists - if so what form
  - What alternatives are?

Example of structuralisation

- Consider GO board
- How to represent it?
  - predicates or lists
- Predicates of form
  - grid(X,Y, Color)
- or List of form
  - [1-1-grey, 1-2-white | Tail]
- What are the benefits
  - Try running a small board of 5 by 5 for both
  - What are the benefits of either representation?



Using Pictures

- Graphical representations may help in the perception of some essential relations and qualities of the problem
- Problems can be illustrated by
  - nodes denoting objects
  - with connections between nodes representing relations
- Structured data objects in Prolog can be pictured as trees
- Declarative nature of Prolog facilitates the translation of pictorial representations into prolog Code - see lectures on semantic networks

Some rules of good Prolog style

- Program clauses should be short
- Procedures should be short
- Mnemonic names should be used
- Layout is important
  - Use space, blank lines and indentation
- The cut operator should be used with care
  - better avoided
- The not procedure can sometimes lead to surprising behavior - but preferable to cut
- The use of a semi-colon may obscure the meaning

merge - bad style

```
merge1(List1, List2, List3):-
List1=[ ], !,
List3=List2;      % 1st list empty
List2=[ ], !,
List3=List1;      % 2nd list empty
List1=[X|List4],
List2=[Y|List5],
(X<Y, !, Z=X, merge1(List4,List2, List6);
Z=Y, merge1(List1, List5, List6)),
List3 = [Z|List6].
```

merge - better style

```
merge2([ ], L, L):- !. %prevent redundant solutions
merge2(L, [ ], L).              % boundary cases

% following clause does ordering test
merge2([H1|Rest1],[H2|Rest2],[H1|Rest3]):-
            H1 < H2, !,         %prevent backtracking
            merge2(Rest1, [H2|Rest2], Rest3).

% No ordering test needed on this clause
merge2(List1,[H|Rest2],[H|Rest3]):-
            merge2(List1, Rest2, Rest3).
```

Debugging

- Prolog provides a special debugging aid for *tracing a goal*
- Tracing is activated using the predicate **trace**
  - displays useful information during execution
- Entry info: predicate name and values of args
- Exit info:
  - if success the value of arguments that satisfy goal
  - otherwise an indication of failure
- Re-entry info:
  - invocation of the same goal caused by backtracking

- ?- trace, merge2([1,2],[1],L).
- Call: (8) merge2([1, 2], [1], _G266) ? creep
- Call: (9) 1<1 ? creep
- Fail: (9) 1<1 ? creep
- Redo: (8) merge2([1, 2], [1], _G266) ? creep
- Call: (9) merge2([1, 2], [], _G383) ? creep
- Exit: (9) merge2([1, 2], [], [1, 2]) ? creep
- Exit: (8) merge2([1, 2], [1], [1, 1, 2]) ? creep

- L = [1, 1, 2]
- Yes

Sorting lists - Bubble Sort

- **bubblesort(List, Sorted)**.
  - Assume ordering relation **gt(X,Y)**
    - ☞ meaning **X** is greater than **Y**
    - ☞ If **X** and **Y** are numbers then could use
    - ☞ **gt(X, Y) :- X > Y.**
  - find two adjacent elements **X**, **Y** in **List**
  - such that **gt(X, Y)** - swap **X** and **Y** in **List** to get **List1**
  - sort **List1**
  - if there no pair **X**, Y such that **gt(X,Y)**
  - then **List** is **Sorted**

Bubble Sort

```
bubblesort(List, Sorted):-
    swap(List, List1), !,    % A Useful Swap?
    bubblesort(List1, Sorted).
bubblesort(Sorted, Sorted).       % Boundary

swap( [ X, Y | Rest], [Y, X | Rest] ):-
    gt(X, Y).
swap( [ Z | Rest1 ], [ Z | Rest2 ]):-
    swap(Rest1, Rest2).
```

Insert Sort

- To sort a non-empty list, L = [H | T]
  - sort the tail T of L
  - Insert the head H into a position in the sorted tail such that the result is sorted

```
insertsort( [ ], [ ] ).
insertsort( [ X | T ], Sorted ):-
   insertsort( T, SortedT ), % sort tail
   insert(X, SortedT, Sorted ).    % insert and sort
insert(X, [Y | Sorted], [ Y | Sorted1 ]):- gt(X, Y), !,
                insert(X, Sorted, Sorted1).
insert( X, Sorted, [ X | Sorted ]).
```

Running swap.pl - with time/1

- ?- bubblesort([1, 3, 9, 0, 6, 8, 4, 5, 7, 2],S).

  S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- ?- insertsort([1, 3, 9, 0, 6, 8, 4, 5, 7, 2],S).

  S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- ?- time(bubblesort([1, 3, 9, 0, 6, 8, 4, 5, 7, 2],S)).

  556 inferences in 0.00 seconds (Infinite Lips)

  S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- ?- time(insertsort([1, 3, 9, 0, 6, 8, 4, 5, 7, 2],S)).

  107 inferences in 0.00 seconds (Infinite Lips)

  S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Running the merges with time/1

- ?- time(merge1([0, 2, 4, 6, 8], [1, 3, 5, 7, 9], L)).
  131 inferences in 0.00 seconds (Infinite Lips)
  L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  Yes

- ?- time(merge2([0, 2, 4, 6, 8], [1, 3, 5, 7, 9], L)).
  25 inferences in 0.00 seconds (Infinite Lips)
  L = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  Yes

Manipulating Characters

- atom_chars/2
  - Convert between atom and list of ASCII values
- atom_char/2
  - Convert between atom and ASCII value
- name/2
  - Convert between atom and list of ASCII characters
- string_to_list/2
  - Conversion between string and list of ASCII
- put/1 places ASII value on output stream
- get0/1 Read next character
- get/1 Read first non-blank character

Manipulating Characters

- atom_chars/2
  - ?- atom_chars(atom, Chars).

    Chars = [97, 116, 111, 109]
  - ?- atom_chars(Atom, “Chars”).

    Atom = ‘Chars’

- atom_char/2
  - ?- atom_char(a, Char).

    Char = 97
  - ?- atom_char(Atom, 97).

    Atom = a

Manipulating Characters

- name/2
  - ?- name(atom, List).

    Chars = [97, 116, 111, 109]
  - ?- name(Atom, "abc").

    Atom = abc

- string_to_list/2
  - ?- string_to_list('Atom',L).

    L = [65, 116, 111, 109]
  - ?- string_to_list(String, "StringOfAscii").

    String = "StringOfAscii"

Manipulating Characters

- ?- string_to_list(String, [65, 111]), atom(String).

  No
- ?- String = 'atom', string_to_list(S, List), atom(S).

  S = atom

  List = [97, 116, 111, 109]

  Yes
- ?- string_to_list(String, [65, 111]), string_to_atom(String,Atom), atom(Atom).

  String = "Ao"

  Atom = 'Ao'

  Yes

Manipulating Characters

- Input and Output of ASCII values
  - ?- put(65), put(66), put(67).

    ABC

  - ?- get0(C).

    : a

    C = 97

    ?- get0(C), get0(A).

    : as

    C = 97

    A = 115

  - ?- get(C).

    : a

    C = 97

Testing the Type of Terms

- var/1 Type check for unbound variable
- nonvar/1 Type check for bound term
- ground/1 Verify term holds no unbound variables
- atom/1 Type check for an atom
- atomic/1 Type check for primitive
- integer/1 Type check for integer
- number/1 Type check for integer or float
- float/1 Type check for a floating point number

- ?- var(X), X = 2.
  X = 2
  Yes

  ?- nonvar(X), X = 2.
  No

- ?- X = 2, var(X).
  No

  ?- X = 2, nonvar(X)
  X =2
  Yes

- ?- integer(X), X =2.
  No

  ?- X = 2, integer(X).
  X = 2
  Yes

- ?- X = 2, atomic(X).
  X = 2
  Yes

  ?- X = 2, atom(X).
  No

  ?- X = a, atom(X).
  X = a
  Yes

- ?- var(X), X = 2.
  X = 2
  Yes
- ?- X = 2, var(X).
  No
- ?- nonvar(X), X = 2
  No
- ?- X = 2, nonvar(X).
  X = 2.
  Yes
- ?- integer(X), X =2.
  No
- ?- X = 2, integer(X).
  X =2
  Yes
- ?- X = 2, atom(X).
  No
- ?- X = a, atom(X).
  X = a
  Yes
- ?- atom(X), X =a.
  No
- ?- X = 2, atomic(X).
  X = 2
  Yes

What does the following code do?

```
enigma( [ ] ):-        write( '.'), nl.
enigma( [ L | Lt ]):-  write(' '),
                       name(L, [ Z | Zt ]),
                       Y is Z - 32,
                       name(X, [ Y | Zt ]),
                       write(X),
                       enigma( Lt ).
```

Comments

- % This is a comment
- /* This is another Comment */
- Layout

```
% File:        mycomments.pl
% author:      fred platocrates
% date:        04:15am 10 Junus 350BC
% updated:     Thursday, April 13, 2000
% purpose:     Header for basic comments file
%%%%%%%%%%%%%%%%%%%%%%%
% go/0 this starts the program which exits prolog
go:-     halt.
```

Getting a listing

- listing/0 lists all the loaded code
  - some of which is not yours

```
d:/LECTURES/AI1/PROLOG/MYCOMM~1.PL
    compiled, 0.00 sec, 640 bytes.
?- listing.
    '$user_query'(1, []) :-
                listing.
    go :-
        halt.
%   Foreign: window_title/2
Yes
```

Getting a listing

- listing/1 lists all the predicate clauses that match

```
?- listing(merge2).
  merge2([], A, A) :- !.
  merge2(A, [], A).
  merge2([A|B], [C|D], [A|E]) :-
          A<C, !,
        merge2(B, [C|D], E).
  merge2(A, [B|C], [B|D]) :-
        merge2(A, C, D).
  Yes
```