Prolog: Beyond the text & Summary

Artificial Intelligence Programming in Prolog

Lecturer: Tim Smith

Lecture 18

29/11/04

Contents

- Prolog: Beyond the text
  - Tcl/tk
  - Java and prolog
  - Visual Prolog
  - ~ COGENT

  * Will not be examined on ‘Beyond the text’. It presents advanced Prolog details beyond the specification of this course*.
- Exam details
- Lecture Summaries

Creating Prolog GUIs

- In AIPP we have only been using Prolog at the command line.
- This makes it seem of limited use, more “retro”, compared to other languages, such as Java, which have significant graphical components.
- *But, Prolog does not have to be just textual!*
- Various techniques exists for creating Graphical User Interfaces (GUIs) for Prolog:
  - Tcl/tk
  - Jasper (Java interface)
  - Visual Basic (not discussed)
  - Visual Prolog^{tm}
- Details on all of these available in the SICStus manual. http://www.sics.se/sicstus/docs/latest/html/sicstus.html/

Tcl/Tk

- Tcl/Tk ("*tickle/tee-kay*")
  - *a scripting language* and
  - toolkit for manipulating *window based interfaces*.
- Very simple to code and quickly prototype cross-platform GUIs.
- You might have come across Tcl/Tk on the HCI course.
- SICStus Prolog contains a Tcl/Tk library (tcltk) which allows GUIs to be controlled and created:
  1. The Prolog program loads the Tcl/Tk Prolog library,
  2. creates a Tcl/Tk interpreter, and
  3. sends commands to the interpreter to create a GUI.
  4. The user interacts with the GUI and therefore with the underlying Prolog system.
- See SICStus manual for Tcl/Tk tutorials.

PROLOG

Tcl/Tk

```
% telephone book example
:- use_module(library(tcltk)).

telephone(fred, '123-456').
telephone(wilbert, '222-2222').
telephone(taxi, '200-0000').
telephone(mary, '00-36-1-666-6666').

go :-
  tk_new([name('Example 2')], T),
  tcl_eval(T, 'entry .name -textvariable name',_),
  tcl_eval(T, 'button .search -text search -command {
          prolog telephone($name,X); ← Prolog query
          set result $prolog_variables(X) }',_),
  tcl_eval(T, 'label .result -relief raised -textvariable
          result', _),
  tcl_eval(T, 'pack .name .search .result -side top -fill
          x', _),
  tk_main_loop.
```

Prolog → Java: Jasper

- We can take advantage of the advanced programming and GUI strengths of Java by using Jasper.
- Jasper is a bi-directional interface between Java and SICStus Prolog.
- Either Java or Prolog can be the *parent application*:
- If *Prolog* is the parent application:
  - Control of Java is via use_module(library(jasper)) which provides predicates for:
    - Initializing the JVM (Java Virtual Machine),
    - Creating and deleting Java objects directly from Prolog ,
    - Method calls,
    - Global and local (object) reference management.
- However, you will probably mostly control Prolog from Java (to take advantage of its search and DB strengths).

Java → Prolog

- If Java is the parent application,
  - the SICStus runtime kernel will be loaded into the JVM using the System.loadLibrary() method and
  - the package (se.sics.jasper) provides classes representing the SICStus run-time system (SICStus, SPTerm, etc).
- This set of Java classes can then be used to
  - create and manipulate terms,
  - ask queries and
  - request one or more solution.
- The results of the Prolog query can then be utilised by the parent program written in Java (e.g. to display output in a GUI).
- A similar package exists for interfacing Prolog to C/C++.

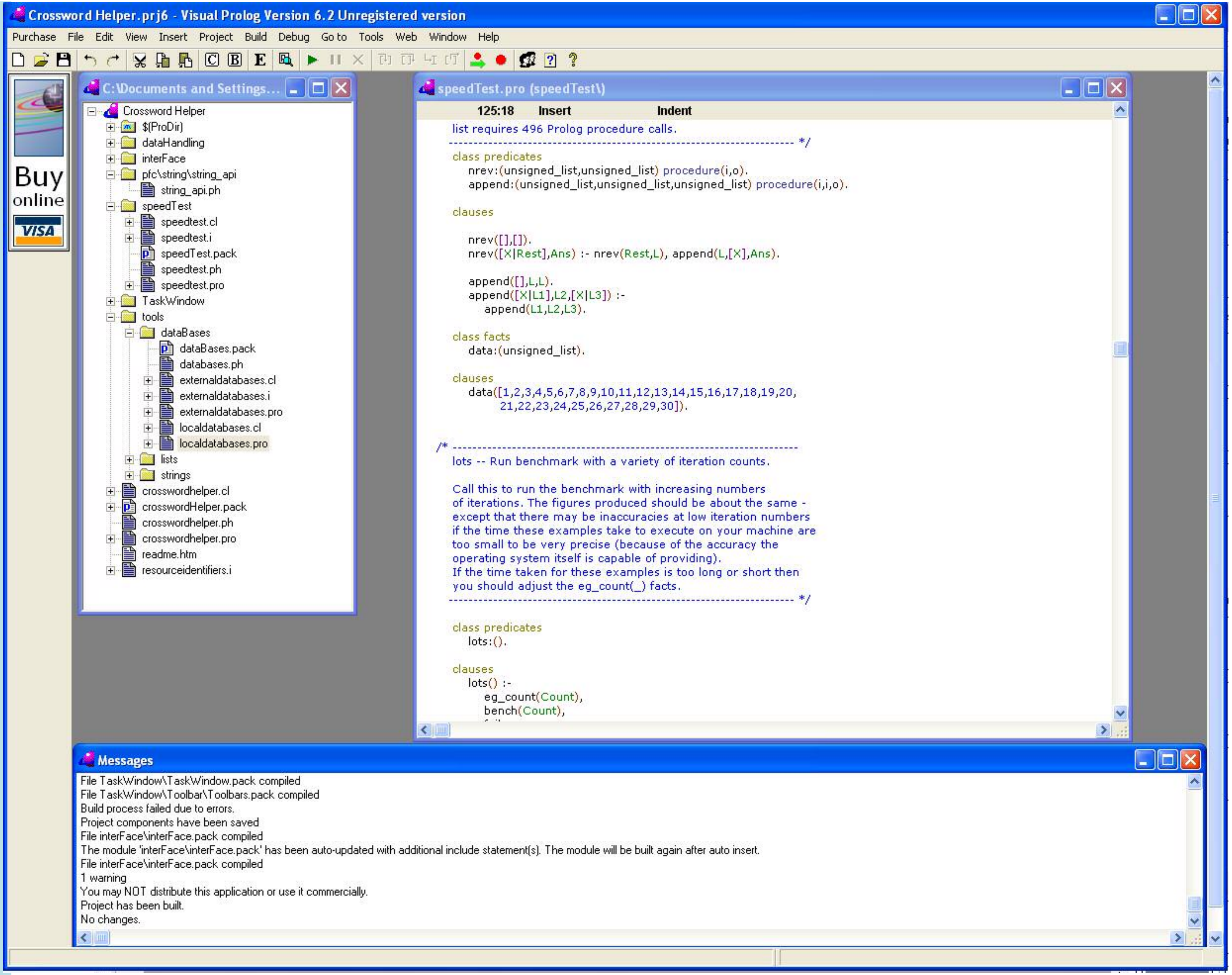
Visual Prolog

- So far, we have only discussed creating GUIs.
- Most other languages also provide a *visual development environment* (VDE) to simplify the task of programming.
- Visual Prolog (http://www.visual-prolog.com/) is a language and VDE used to create stand-alone Prolog programs with Windows-standard GUIs.
- Contains:
  - an editor
  - debugger
  - compiler
  - GUI editors
- Based on Turbo Prolog and PDC Prolog **not** ISO Prolog so there are a few idiosyncrasies but mostly familiar.
- Allows direct coding or automatic code writing through the use of *Wizards*.
- A free non-commercial version is available.

Programming in Visual Prolog

- Programs are written in modified Prolog code.
- Predicate definitions are written as normal but are identified as serving a particular function.
- Incorporates ideas from *object-orienting programming*:
  - programs are split up into *classes* which control the scope of clauses, variables, and constants.
  - classes are stored in separate files.
- Extra code controls how the logical computation interfaces with the GUI.
- The GUI editor allows Dialog boxes and Menus to be created and coded using a Wizard.
- Supports memory management, linkage to other languages (e.g. HTML, Java, C/++) and Windows functions.

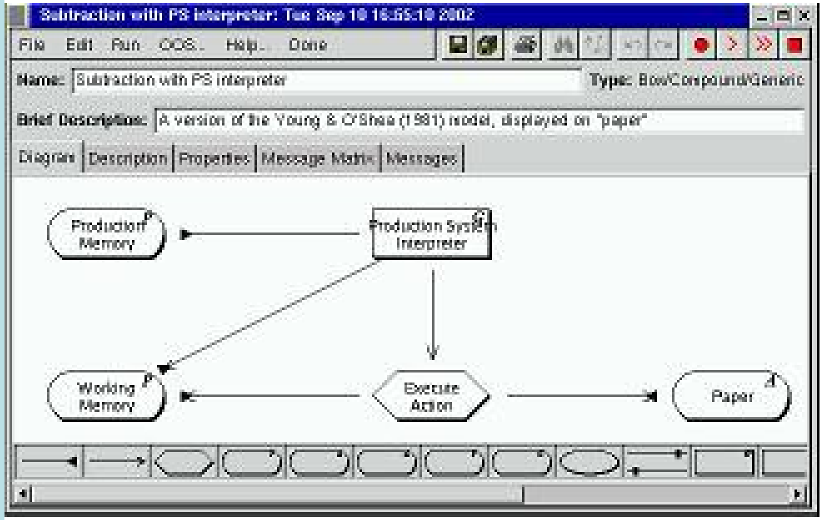
PROLOG

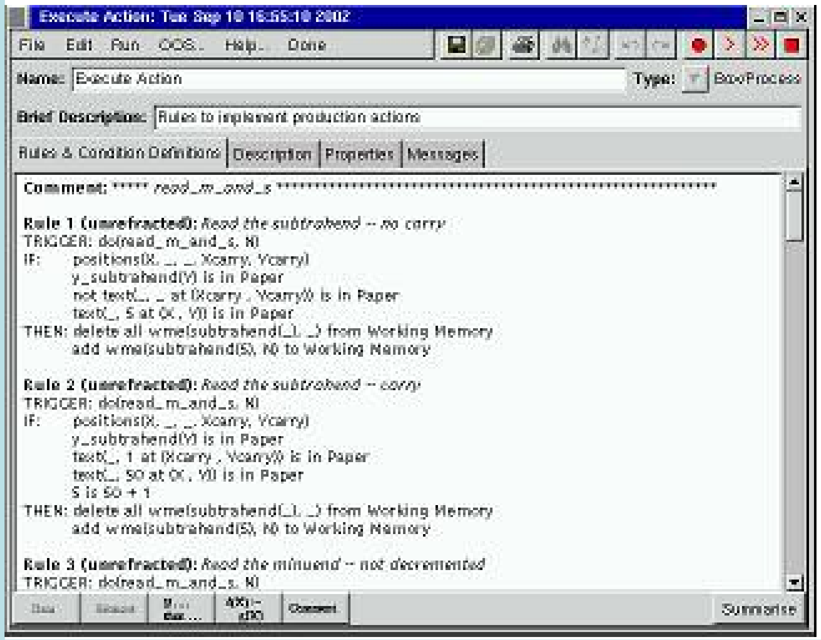

Crossword Helper.prj6 - Visual Prolog Version 6.2 Unregistered version
Purchase File Edit View Insert Project Build Debug Go to Tools Web Window Help
Buy online
VISA
C:\Documents and Settings...
Crossword Helper
$(ProDir)
dataHandling
interFace
pfc\string\string_api
string_api.ph
speedTest
speedtest.cl
speedtest.i
speedTest.pack
speedtest.ph
speedtest.pro
TaskWindow
tools
dataBases
dataBases.pack
databases.ph
externaldatabases.cl
externaldatabases.i
externaldatabases.pro
localdatabases.cl
localdatabases.pro
lists
strings
crosswordhelper.cl
crosswordHelper.pack
crosswordhelper.ph
crosswordhelper.pro
readme.htm
resourceidentifiers.i
speedTest.pro (speedTest\)
125:18 Insert Indent
list requires 496 Prolog procedure calls.
-- */
class predicates
nrev:(unsigned_list,unsigned_list) procedure(i,o).
append:(unsigned_list,unsigned_list,unsigned_list) procedure(i,i,o).
clauses
nrev([],[]).
nrev([X|Rest],Ans) :- nrev(Rest,L), append(L,[X],Ans).
append([],L,L).
append([X|L1],L2,[X|L3]) :-
append(L1,L2,L3).
class facts
data:(unsigned_list).
clauses
data([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
21,22,23,24,25,26,27,28,29,30]).
/* --
lots -- Run benchmark with a variety of iteration counts.
Call this to run the benchmark with increasing numbers
of iterations. The figures produced should be about the same -
except that there may be inaccuracies at low iteration numbers
if the time these examples take to execute on your machine are
too small to be very precise (because of the accuracy the
operating system itself is capable of providing).
If the time taken for these examples is too long or short then
you should adjust the eg_count(_) facts.
-- */
class predicates
lots:().
clauses
lots() :-
eg_count(Count),
bench(Count),
Messages
File TaskWindow\TaskWindow.pack compiled
File TaskWindow\Toolbar\Toolbars.pack compiled
Build process failed due to errors.
Project components have been saved
File interFace\interFace.pack compiled
The module 'interFace\interFace.pack' has been auto-updated with additional include statement(s). The module will be built again after auto insert.
File interFace\interFace.pack compiled
1 warning
You may NOT distribute this application or use it commercially.
Project has been built.
No changes.

COGENT

- Prolog can also be found at the base of other systems.
- COGENT is a rule-base language and visual development environment for cognitive modelling.
  - Cognitive Objects within a Graphical EnviroNmenT
- Models of cognitive systems (e.g. memory, reasoning, problem solving) can be developed by
  - drawing flow charts,
  - filling in forms, and
  - modifying cognitive modules (e.g. memory buffers, I/O).
- The user develops computational models without the need for direct coding.
- However, the resulting programs are similar to Prolog and the VDE can be bypassed to code rules directly.

COGENT





- COGENT highlights the suitability of Prolog for AI.
- Artificial Intelligence <u>should</u> endeavour to create computational systems that replicate the functions of natural cognitive systems.
- Prolog was developed as a logic-based programming language precisely because logic is considered as a suitable representation for human reasoning.
- Therefore, Prolog is <u>THE</u> AI programming language.

Summary: Beyond the text

- There are few 'real' reasons for not considering Prolog for use in commercial settings.
- Most of the aesthetic and practical issues can be resolved by using Visual Prolog or creating GUIs.
  – However, building GUIs complicates what would otherwise be a very simple, economical Prolog program.
  – So, stick to text unless you have a real reason why your program needs a GUI.
- Prolog can be used to solve most symbolic computation problems using *concise* and *efficient* programs.
- Sometimes it may not be the first language you think of but *don't dismiss outright*.
- Due to its flexibility you can make it do virtually anything you want. You just have to know how.

Part 2:
Summary and Recap

AIPP Examination

- To be held between late April and mid May.
- 1.5 hr exam. 70% of course mark.
- *One compulsory section*:
  - testing your general Prolog knowledge. Consisting of
    - short answer questions,
    - deciphering prewritten predicates,
    - writing small predicates.
- *Choose one section from two alternatives*.
  - Longer answer questions consisting of:
    - Must develop or adapt a **short** program;
    - Might utilise specific techniques (e.g. DCG, sentence manipulation, planning, operators, etc).
    - Have to write descriptions of theory as well as code.
- <u>No text books permitted.</u>
- Look at course website for link to previous papers (vary in relevance).

PROLOG

1: Introduction to Prolog

- Prolog = Programming in Logic
- ISO standard is based on Edinburgh Syntax.
- Derived from Horn Clauses:
    - (parent(X,Z) $\wedge$ ancestor(Z,Y)) $\supset$ ancestor(X,Y)
- Prolog is a declarative programming language:
    - We ask our programs questions and they are proved using a logic incorporated in the interpreter.
- A Prolog program is a database consisting of:
    - **facts:** name('Bob Parr').
    - **rules**: incredible(X):- name(X), X = 'Bob Parr'.
- Prolog is good at Symbolic AI.
- Prolog is bad at complex math, I/0, interfaces….

2: Prolog Fundamentals

- A Prolog program consists of predicate definitions.
- A predicate denotes a property or relationship between objects.
- Definitions consist of clauses.
- A clause has a head and a body (Rule) or just a head (Fact).
- A head consists of a predicate name and arguments.
- A clause body consists of a conjunction of terms.
- Terms can be constants, variables, or compound terms.
- We can set our program goals by typing a command that unifies with a clause head.
- A goal unifies with clause heads in order (top down).
- Unification leads to the instantiation of variables to values.
- If any variables in the initial goal become instantiated this is reported back to the user.

3: The central ideas of Prolog

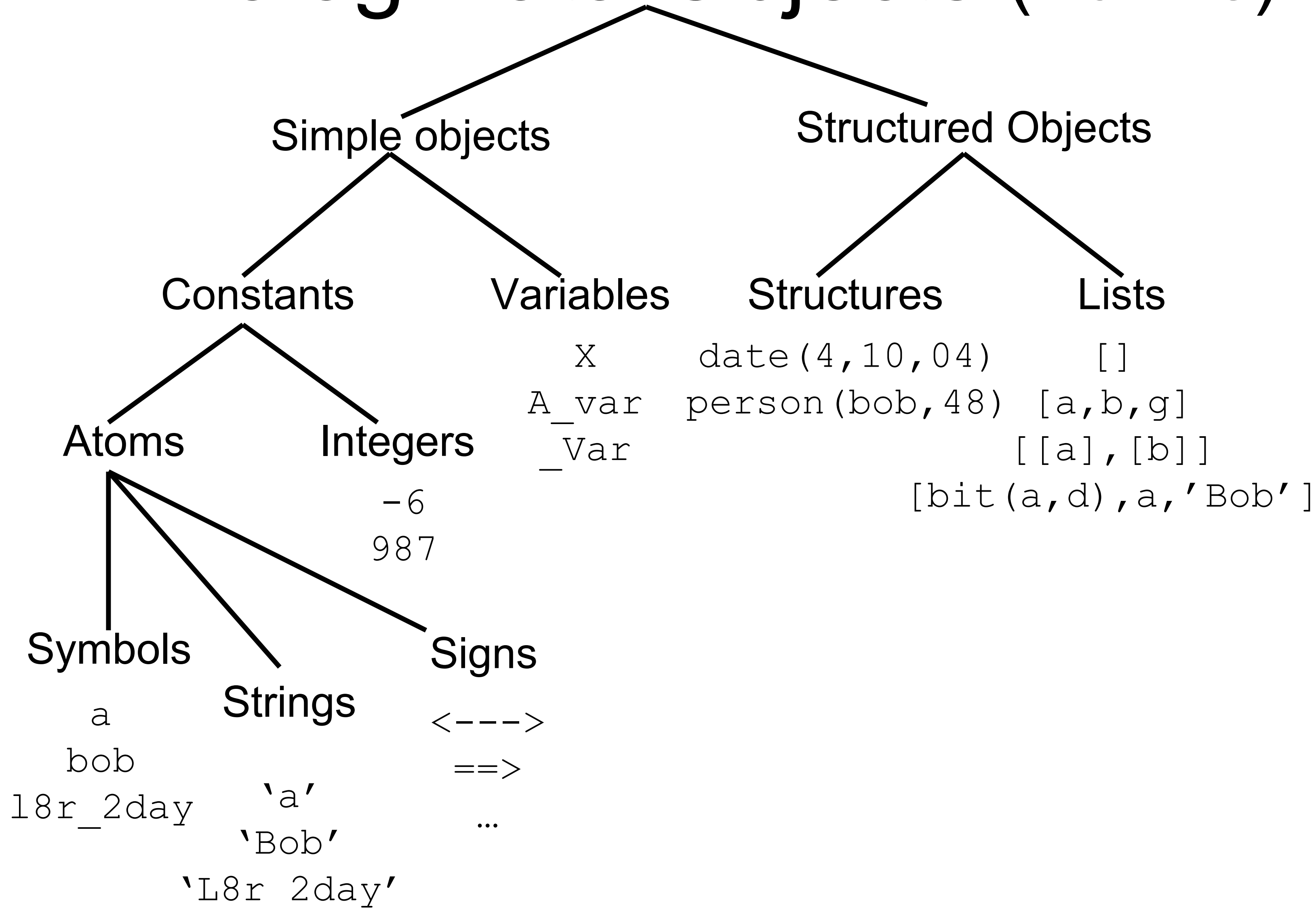
- SUCCESS/FAILURE
  - any computation can “**succeed**" or “**fail**", and this is used as a ‘**test**‘ mechanism.
- MATCHING
  - any two data items can be compared for similarity (X==Y), and values can be bound to variables in order to allow a match to succeed (X =Y).
- SEARCHING
  - the whole activity of the Prolog system is to search through various options to find a combination that succeeds.
    - Main search tools are backtracking and recursion
- BACKTRACKING
  - when the system fails during its search, it returns to previous choices to see if making a different choice would allow success.

4: Recursion, Structures, and Lists

- Prolog's proof strategy can be represented using AND/OR trees.
- Tree representations allow us trace Prolog's search for multiple matches to a query.
- They also highlight the strengths and weaknesses of recursion (e.g. economical code vs. infinite looping).
- Recursive data structures can be represented as *structures* (`functor(component)`) or *lists* (`[a,b,X,a(1)]`).
- Structures can be unified with variables then used as commands: `X=member(x,[a,d,x]), call(X).`
- Lists can store ordered data and allow its sequential processing through *recursion*.

PROLOG

4: Prolog Data Objects (*Terms*)



5: List Processing

- Lists can be decomposed by unifying with [Head|Tail]
- Base case: `is_a_list([]).`
- Recursive cases: `is_a_list([_|T]):- is_a_list(T).`
- Using focused recursion to stop infinite loops.
  - only recurse on smaller parts of the problem.
- Lists are *deconstructed during recursion* then *reconstructed on backtracking*.
- Showed three techniques for collecting results:
  - Recursively find a result, then revise it at each level.
    - listlength/3
  - Use an accumulator to build up result during recursion.
    - reverse/3
  - Build result in the head of the clause during backtracking.
    - append/3

6: Built-in Predicates.

`var(X)`	is true if X is currently an uninstantiated variable.
`nonvar(X)`	is true if X is not a variable, or already instantiated
`atom(X)`	is true if X currently stands for an atom
`number(X)`	is true if X currently stands for a number
`integer(X)`	is true if X currently stands for an integer
`float(X)`	is true if X currently stands for a real number.
`atomic(X)`	is true if X currently stands for a number or an atom.
`compound(X)`	is true if X currently stands for a structure ([a] or b(a)).
`ground(X)`	is true if X does not contain any uninstantiated variables.
`arg(N,Term,A)`	is true if A is the Nth argument in Term.
`functor(T,F,N)`	is true if F is the principal functor of T and N is the arity of F: `functor(father(bob),father,1).`
`Term =.. L`	is true if L is a list that contains the principal functor of Term, followed by its arguments: `father(bob) =.. [father,bob].`

6: All Solutions

- Built-in predicates that repeatedly call a goal P, instantiating the variable X within P and adding it to the list L.
- They succeed when there are no more solutions.
- Exactly simulate the repeated use of ';' at the SICStus prompt to find all of the solutions.

`findall(X,P,L)` = `find all of the Xs, such that X satisfies goal P and put the results in list L'.

e.g. `findall(X,(member(X,[2,5,6,4,7]),X>4),L).` → `L=[5,6,7].`

`setof(X,P,L)`= It produces the ***set*** of all X that solve P, with any duplicates removed, and the results ***sorted***.

`bagof(X,P,L)`= Same as setof/3 but contains duplicates and results aren't sorted.

7: Controlling Backtracking

- Clearing up equality: `=`, `is`, `=:=`, `=\=`, `==`, `\==`, `\+`
- Controlling backtracking: the cut `!.` Succeeds when first called and commits proof to the clause it is in. Fails on backtracking (REDO).
  - **Efficiency:** avoids needless REDO-ing which cannot succeed.
  - **Simpler programs:** conditions for choosing clauses can be simpler.
  - **Robust predicates:** definitions behave properly when forced to REDO.
- **Green cut** = cut doesn't change the predicate logic as clauses are mutually exclusive anyway = **good**
- Red cut = without the cut the logic is different = **bad**
- Cut – fail: when it is easier to prove something is false than true.

PROLOG

PROLOG

8: State-Space Search

- State-Space Search can be used to find optimal paths through problem spaces.
- A state-space is represented as *a downwards-growing tree* with nodes representing states and branches as legal moves between states.
- Prolog's unification strategy allows a simple implementation of *depth-first search*.
- The efficiency of this can be improved by performing *iterative deepening* search (using backtracking).
- *Breadth-first* search always finds the shortest path to the goal state.
- Both depth and breadth-first search can be implemented using an *agenda*:
  - *depth-first* adds new nodes to the *front* of the agenda;
  - *breadth-first* adds new nodes to the *end*.

9: Informed Search Strategies

- *Blind search*: Depth-First, Breadth-First, IDS
  - Do not use knowledge of problem space to find solution.
- *vs. Informed search*
- *Best-first search*: Order agenda based on some measure of how 'good' each state is.
- *Uniform-cost:* Cost of getting to current state from initial state = **`g(n)`**
- *Greedy search:* Estimated cost of reaching goal from current state

  = *Heuristic evaluation function,* **`h(n)`**
- *A* search:* **`f(n) = g(n) + h(n)`**
- Admissibility: **`h(n)`** never *overestimates* the actual cost of getting to the goal state.
- Informedness: A search strategy which searches less of the state-space in order to find a goal state is more *informed.*

10: Definite Clause Grammars

- We can use the --> DCG operator in Prolog to define grammars for any language.

```
e.g. sentence -->  noun_phrase, verb_phrase
```

- The grammar rules consist of *non-terminal symbols* (e.g. NP, VP) which define the structure of the language and *terminal symbols* (e.g. Noun, Verb) which are the words in our language.
- The Prolog interpreter converts the DCG notation into conventional Prolog code using *difference lists*.

```
|?- sentence([‘I’,like,cheese],[]).
```

- We can add *arguments* to non-terminal symbols in our grammar for any reason (e.g. number agreement).
- We can also add pure Prolog code to the right-hand side of a DCG rule by enclosing it in { }.

11: Parsing and Semantics in DCGs

- A basic DCG only recognises sentences.
- A DCG can also interpret a sentence and extract a rudimentary representation of its meaning:
- A Parse Tree: identifies the grammatical role of each word and creates a structural representation.

```
sentence(s(NP,VP)) --> noun_phrase(NP), verb_phrase(VP).
```

- Logical Representation: we can construct Prolog terms from the content of the sentence.
  - `intrans_verb(Somebody,paints(Somebody)) --> [paints].`
  - These can then be used as queries passed to the Prolog interpreter
  - e.g. “Does jim paint?” would be converted to paints(jim) by the DCG and if a matching fact existed in the database the answer would be “yes”.

PROLOG

12: Input/Output

write/[1,2]	write a term to the current output stream.
nl/[0,1]	write a new line to the current output stream.
tab/[1,2]	write a specified number of white spaces to the current output stream.
put/[1,2]	write a specified ASCII character.
read/[1,2]	read a term from the current input stream.
get/[1,2]	read a **printable** ASCII character from the input stream (i.e. skip over blank spaces).
get0/[1,2]	read an ASCII character from the input stream
see/1	make a specified file the current **input** stream.
seeing/1	determine the current **input** stream.
seen/0	close the current **input** stream and reset it to user.
tell/1	make a specified file the current **output** stream.
telling/1	determine the current **output** stream.
told/0	close the current **output** stream and reset it to user.
name/2	arg1 (an atom) is made of the ASCII characters listed in arg2

13: Sentence Manipulation

- Tokenizing a sentence:
  - use name/2 to convert a sentence into a list of ASCII
  - group characters into words by identifying spaces (32)
- A Tokenized sentence can then be input to a DCG and Prolog queries generated based on its meaning.
- Morphological processing: words can be transformed (e.g. pluralised) by *pattern-matching* ASCII lists and appending suffixes.
- Pattern-matching can also be used to implement `stupid' Chat-Bots, e.g. ELIZA

```
rule([i,hate,X,'.'], [do,you,really,hate,X,?]).
```

- But pattern-matching is not as flexible as DCG parsing and does not extract any meaning.

14: Database Manipulation

- assert(Clause): add clauses to the database (DB)
  - asserta(Clause): add as the first predicate definition.
  - assertz(Clause): add as the last predicate definition.

- retract(Clause): remove a clause from the DB
- retractall(Head): remove all clauses with Head

- :- dynamic a/2, b/3. Predicates must be declared as synamic before they can be manipulated.
- clause(Head,Body): finds first clause with a particular Head and Body (these can be variables).

- \`Caching\` solutions.
  - `solve(problem1, Sol), asserta(solve(problem1, Sol).`
- \`Listing' solutions to an output file.
  - once new facts are asserted, they can be written to a new file, saving them for later use.

PROLOG

15: Planning

- A Plan is a sequence of actions that changes the state of the world from an Initial state to a Goal state.
- Planning can be considered as a *logical inference problem*.
- *STRIPS* is a classic planning language.
  - It represents the *state of the world* as a list of facts.
  - *Operators* (actions) can be applied to the world if their preconditions hold.
    - The effect of applying an operator is to *add* and *delete* states from the world.
- A linear planner can be easily implemented in Prolog by:
  - representing operators as `opn(Name,[PreCons],[Add],[Delete]).`
  - choosing operators and applying them in a depth-first manner,
  - using backtracking-through-failure to try multiple operators.

16(1): More Planning

- *Blocks World* is a very common Toy-World problem in AI.
- *Means-Ends Analysis* (MEA) can be used to plan backwards from the Goal state to the Initial state.
  - MEA often creates more direct plans,
  - but is still inefficient as it pursues goals in any order.
- *Goal Protection*: previously completed goals can be protected by making sure that later actions do not destroy them.
  - Forces generation of direct plans through backtracking.
- *Best-first Planning* can use knowledge about the problem domain, the order of actions, and the cost of being in a state to generate the 'cheapest' plan.
- *Partial-Order Planning* can be used for problems that contain multiple sets of goals that do not interact.

16(2): Prolog Operators

- Operators can be declared to create
  - novel compound structures, (e.g. 15 hr 45 min) or
  - a predicate in a non-conventional position (e.g. 5hr <<< 6hr).
- All operators have:
  - *Precedence*: a value between 200 and 1200 that specifies the grouping of structures made up of more than one operator.
  - *Associativity*: a specification of how structures made up of operators with the same precedence group.

    = The arguments of an operator (**`f`**) must be:
    - of a strictly lower precedence value (notated **`x`**), or
    - of an equal or lower precedence value (notated **`y`**).
- Operators are defined using **`op/3`**: **`:- op(700, xfx, <<<).`**
- Once an operator has been defined it can be defined as a predicate in the conventional way.

17: Meta-Interpretation

- Controlling the flow of computation: call/1
  - Representing logical relationships
    - conjunctions ($P \wedge Q$): (FirstGoal, OtherGoals)
    - disjunctions ($P \vee Q$) : (FirstGoal; OtherGoals)
    - conjunctive not ¬ ($P \wedge Q$) : \+ (FirstGoal, OtherGoals)
  - if.....then....else.....
    - X -> Y; Z

- Meta-Interpreters
  - clause(Head,Body)
  - left-to-right interpreter
  - right-to-left interpreter
  - breadth-first: using an agenda
  - best-first: using ground/1
  - others

```
solve(true).

solve(Goal) :-
      \+ Goal = (_, _),
        solve(Body).

solve((Goal1, Goal2)) :-
        solve(Goal1),
        solve(Goal2).
```

PROLOG

```
|?- write('Goodbye World'), fail.
Goodbye World
no
```