Map coloring example

Database for map coloring

coloring(A,B,C,D,E,F) :-
    different(A,B),
    different(A,C),
    different(A,D),
    different(A,F),
    different(B,C),
    different(B,E),
    different(C,D),
    different(C,E),
    different(D,E),
    different(E,F).
Map coloring example (continued)

different(yellow,blue).
different(blue,yellow).
different(yellow,red).
different(red,yellow).
different(blue,red).
different(red,blue).

Query:

?- coloring(A,B,C,D,E,F).

Solution found:

A = yellow
B = blue
C = red
D = blue
E = yellow
F = blue
Another approach to map coloring

Create rules that color any map given by query.

Mappings (\( R \) is a region and \( C \) is its color):

\[
\text{lookup}(R, [\text{assign}(R,C) | \_], C).
\]

\[
\text{lookup}(R, [\_ | T], C) :- \text{lookup}(R, T, C).
\]

Colorings (\( M \) is a mapping from regions to colors).

Coloring is valid if different colors for adjacent regions.

\[
\text{valid}(M, []). \]

\[
\text{valid}(M, [\text{adj}(X, [ \_]) | R]) :- \text{valid}(M, R).
\]

\[
\text{valid}(M, [\text{adj}(X, [Y | T]) | R]) :-
\]

\[
\text{lookup}(X, M, Xc), \text{lookup}(Y, M, Yc), \text{different}(Xc, Yc),
\]

\[
\text{valid}(M, [\text{adj}(X, T) | R]).
\]

Assign colors to map regions: one color per region.

\[
\text{assignment}([], [], []). \]

\[
\text{assignment}([\text{assign}(R, \_)|M], [\text{adj}(R, \_)|T]) :-
\]

\[
\text{assignment}(M, T).
\]
Another approach to map coloring (continued)

Search for assignment that colors correctly:

coloring(M,G) :- assignment(M,G), valid(M,G).

Query for the previous graph is:

?- coloring(M, [adj(a, [b, c, d, f]),
               adj(b, [a, c, e]),
               adj(c, [a, b, d, e]),
               adj(d, [a, c, e]),
               adj(e, [b, c, d, f]),
               adj(f, [a, e])]).

Solution found:

M = [assign(a, yellow), assign(b, blue), assign(c, red),
     assign(d, blue), assign(e, yellow), assign(f, blue)]
Prolog in a nutshell

Prolog

- values are terms, and so is abstract syntax:
  ```plaintext
datatype term = VAR of string
  | LITERAL of int
  | APPLY of string * term list
  
  Variables = strings that begin with upper case.
  Functors (symbols) = strings that begin with lower case.
```

- is untyped (everything is a term)
- has no data abstraction
- has no functional abstraction! (relations/predicates instead)
- has no mutable state
- has no explicit control flow

Programs are declarative:
```plaintext
goal = string * term list
clause = goal * goal list
database = clause list
query = goal list
```

- has an unusual evaluation model based on backtracking and unification
Concrete syntax of µProlog in EBNF

\[
toplevel \quad \rightarrow \quad clause \mid query \mid mode-change \mid use
\]

\[
clause \quad \rightarrow \quad goal \ [ \ :- \ goals \ ] .
\]

\[
query \quad \rightarrow \quad goals .
\]

\[
goals \quad \rightarrow \quad goal \ \{ \ , \ goal \ \}
\]

\[
goal \quad \rightarrow \quad term \ is \ term
\]
\[
\quad \mid \quad term \ binary-predicate \ term
\]
\[
\quad \mid \quad predicate \ [ \ ( \ term \ \{ \ , \ term \ \} \ ) \ ]
\]

Clauses may only be typed in entry mode, when prompt is \(-\rightarrow\)
Queries may only be typed in query mode, when prompt is \(-\)
Keyword “is” denotes assignment
Predefined binary predicates include \(+, -, *, /, <, >, =<, >=\)
Concrete syntax of $\mu$Prolog (continued)

\[
\begin{align*}
\text{term} & \quad \rightarrow \quad _{-} \\
& \quad \mid \quad \text{term binary-functor term} \\
& \quad \mid \quad \text{functor} \left[ ( \text{term} \{ , \text{term} \} ) \right] \\
& \quad \mid \quad [ ] \\
& \quad \mid \quad [ \text{term} \{ , \text{term} \} [ | \text{term} ] ] \\
& \quad \mid \quad \text{integer} \\
& \quad \mid \quad \text{variable} \\
\text{mode-change} & \quad \rightarrow \quad [\text{query}]. \mid [\text{rule}]. \mid [\text{fact}]. \mid [\text{clause}]. \mid [\text{user}]. \\
\text{use} & \quad \rightarrow \quad [\text{filename}].
\end{align*}
\]

Underscore (_ _) matches anything (as in ML)
Within terms, “[” and “[” are list brackets (as in ML)
“[query].” switches to query mode, and other four all switch to entry mode mode
Concrete syntax of µProlog (continued)

\[\text{predicate} \rightarrow ! \mid \text{name beginning with lower-case letter}\]

\[\text{binary-predicate} \rightarrow \text{name formed from symbols} \mid % \& \ast - + : = \sim < > / ? \` \$ \backslash\]

\[\text{functor} \rightarrow \text{name beginning with lower-case letter}\]

\[\text{binary-functor} \rightarrow \text{name formed from symbols} \mid % \& \ast - + : = \sim < > / ? \` \$ \backslash\]

\[\text{variable} \rightarrow \text{name beginning with upper-case letter}\]

Exclamation point (!) denotes the “cut” (will be described later)
Running Prolog

If you compile `upr.sml`, the original μProlog interpreter written in ML, it does not run correctly.

Certain key features (substitution, unification) were intentionally omitted, to be left as exercises.

The new file `upr-improved.sml` implements working versions of substitution and unification.

You may also use a real Prolog system such as SWI Prolog, which should be installed in the graduate lab. Or visit [http://www.swi-prolog.org/](http://www.swi-prolog.org/) to download.

- Type `[user]`. to go to database entry mode
- Prompt in database entry mode is `|`:
- Type control-D to return back to query mode
- Use `consult(filename)`. to enter rules directly from `filename.pl`
Implementing Prolog: unification

Try to satisfy query by *unifying* it with some conclusion
- predicate and/or functor names must be identical
- number of arguments must be identical
- find substitution unifying arguments
  - Capitalized names are variables which can match anything
- unification is like ML pattern matching, except the “value” can contain variables and Prolog patterns are more general than ML!
Implementing Prolog: backtracking

How to find the “right” conclusion? Backtracking search

To satisfy a goal:
- Try to unify with conclusion of first rule in database
- If successful, apply substitution to first premise, try to satisfy resulting subgoals
- Then apply both substitutions to next premise, and so on…
- If not successful, advance to the next rule in database
- If all rules fail, try again (backtrack) to a previous subgoal

Substitutions accumulate, much as in type inference

See “Byrd box” for details of control flow (on later slide)
Traditional notions

Scoping of variable names is within rule only
- each rule has its own name space for variables

Scoping of predicate/functor names is across entire database

Prolog is essentially untyped
(but together, name and arity can provide a weak notion of “type”)
Unification examples

\[ a(b,C,d,E) \text{ with } x(\ldots) \]
no, doesn’t unify: \( a \) and \( x \) differ

\[ a(b,C,d,E) \text{ with } a(\_,\_,\_) \]
no: different # of args

\[ a(b,C,d,E) \text{ with } a(j,f,G,H) \]
no: \( b \neq j \)

\[ a(b,C,d,E) \text{ with } a(b,f,G,H) \]
yes: by either \( \{ C \mapsto f, G \mapsto d, H \mapsto E \} \)
or \( \{ C \mapsto f, G \mapsto d, E \mapsto H \} \)

\[ a(\text{pred}(X,j)) \text{ with } a(\text{pred}(k,j)) \]
yes: \( \{ X \mapsto k \} \)

\[ a(\text{pred}(X,j)) \text{ with } a(Y) \]
yes: \( \{ Y \mapsto \text{pred}(X,j) \} \)

\[ a(\text{pred}(X,j)) \text{ with } a(X) \]
yes: if \( X \)'s have different scope
(change second \( X \) to \( Y \), and do as above)

\[ a(\text{pred}(X,j)) \text{ with } a(X) \]
no: if \( X \)'s have same scope,
because infinite trees not allowed
(but “occurs check” is often unimplemented)
Simulating functions

No functions, only relations

- “function result” can be represented by extra argument to predicate

Example: \( \text{append}(X, Y, Z) \equiv \text{“Z is the result of appending lists } X \text{ and } Y \text{”} \)

\[
\begin{align*}
\text{append}([], Y, Y). \\
\text{append}([H|X], Y, [H|Z]) & : - \text{ append}(X,Y,Z).
\end{align*}
\]

Difference lists:

\[
\text{contents}(A, \text{difflist}(L,E)) : - \text{ append}(A, E, L).
\]

Representation invariant for difference lists: List \( E \) must be a suffix of list \( L \)

Abstraction function: given by the \text{contents} rule
More about difference lists

\( \text{difflist}([a,b\mid E], E) \) is a “difference list” consisting of \( a \) and \( b \).

\( E \) could be anything: many difference lists consist of only \( a \) and \( b \).

\[
\begin{align*}
\text{difflist}([a,b], []) & \\
\text{difflist}([a,b,c], [c]) & \\
\text{difflist}([a,b,c\mid F], [c\mid F]) & 
\end{align*}
\]

Consing on to the front of a difference list:

\[
\text{cons}(X, \text{difflist}(A,B), \text{difflist}([X\mid A],B)).
\]

Extracting the contents of a difference list:

\[
\text{contents}(A, \text{difflist}(L,E)) :- \text{append}(A,E,L).
\]

Appending lists without recursion!

\[
\text{diffappend}((\text{difflist}(X,Y), \text{difflist}(Y,Z), \text{difflist}(X,Z))
\]

“\( (X - Y) + (Y - Z) = (X - Z) \)”
Backtracking example: computing with difference lists

**Query:**
\[
\text{contents}(W, \text{difflist}([a,b|T], T))
\]

\[
\text{contents}(W, \text{difflist}([a,b|T], T)) \; A=W, \; L=[a,b|T], \; E=T
\]

\[
\text{append}(W,T,[a,b|T]) \; X=W, \; Y=T, \; Z=[a,b|T]
\]

\[
\text{append}([\;], \ldots) \; \text{NO! (occurs check)}
\]

\[
\text{append}([a|X'],T,[a,b|T]) \; W=[a|X'], \ldots
\]

\[
\text{append}(X',T,[b|T])
\]

\[
\text{append}([\;], \ldots) \; \text{NO! (occurs check)}
\]

\[
\text{append}([b|X''],T,[b|T]) \; X'=[b|X''], \ldots
\]

\[
\text{append}(X'',T,T)
\]

\[
\text{append}([\;],T,T) \; X''=[\;], \; Y''=T
\]

**Solution:**
\[
W = [a|X'] = [a,b|X''] = [a,b|\;] = [a,b]
\]

\[
T = Y
\]