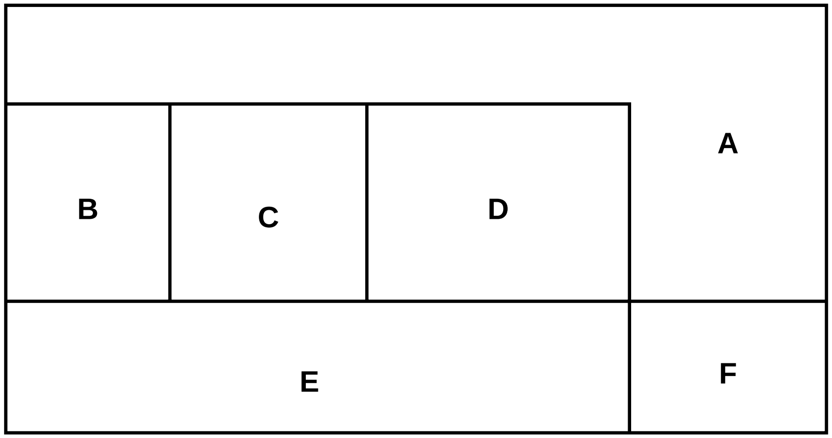
Map coloring example



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):

```
lookup(R,[assign(R,C)|_],C).
lookup(R,[_|T],C) :- lookup(R,T,C).
```

Colorings (`M` is a mapping from regions to colors).
Coloring is valid if different colors for adjacent regions.

```
valid(M,[ ]).
valid(M,[adj(X,[ ])|R]) :- valid(M,R).
valid(M,[adj(X,[Y|T])|R]) :-
  lookup(X,M,Xc),lookup(Y,M,Yc),different(Xc,Yc),
  valid(M,[adj(X,T)|R]).
```

Assign colors to map regions : one color per region.

```
assignment([ ],[ ]).
assignment([assign(R,_)|M],[adj(R,_)|T]) :-
  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:

```
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:

```
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

$$
\begin{array}{lcl}
\textit{toplevel} & \longrightarrow & \textit{clause} \mid \textit{query} \mid \textit{mode-change} \mid \textit{use} \\
\textit{clause} & \longrightarrow & \textit{goal}\ [\ :-\ \textit{goals}\ ]\ \textbf{.} \\
\textit{query} & \longrightarrow & \textit{goals}\ \textbf{.} \\
\textit{goals} & \longrightarrow & \textit{goal}\ \{\ \textbf{,}\ \textit{goal}\ \} \\
\textit{goal} & \longrightarrow & \textit{term}\ \textbf{is}\ \textit{term} \\
 & \mid & \textit{term}\ \textit{binary-predicate}\ \textit{term} \\
 & \mid & \textit{predicate}\ [\ \textbf{(}\ \textit{term}\ \{\ \textbf{,}\ \textit{term}\ \}\ \textbf{)}\ ]
\end{array}
$$

Clauses may only be typed in entry mode, when prompt is $->$

Queries may only be typed in query mode, when prompt is $?-$

Keyword “is” denotes assignment

Predefined binary predicates include $+$, $-$, $*$, $/$, $<$, $>$, $=<$, $>=$

Concrete syntax of μProlog (continued)

term	$\longrightarrow$	_
	|	*term binary-functor term*
	|	*functor* $[$ **(** *term* $\{$ **,** *term* $\}$ **)** $]$
	|	**[ ]**
	|	**[** *term* $\{$ **,** *term* $\}$ $[$ $\vert$ *term* $]$ **]**
	|	*integer*
	|	*variable*
mode-change	$\longrightarrow$	**[query].** $\vert$ **[rule].** $\vert$ **[fact].** $\vert$ **[clause].** $\vert$ **[user].**
use	$\longrightarrow$	**[** *filename* **].**

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

Concrete syntax of μProlog (continued)

predicate	$\longrightarrow$	**!** $\vert$ **name beginning with lower-case letter**
binary-predicate	$\longrightarrow$	**name formed from symbols** `|%^&*-+:=~<>/?`$\`
functor	$\longrightarrow$	**name beginning with lower-case letter**
binary-functor	$\longrightarrow$	**name formed from symbols** `|%^&*-+:=~<>/?`$\`
variable	$\longrightarrow$	**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/` 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)`	with	`x( ... )`	no, doesn't unify: `a` and `x` differ
`a(b,C,d,E)`	with	`a(_,_,_)`	no: different # of args
`a(b,C,d,E)`	with	`a(j,f,G,H)`	no: $\texttt{b} \neq \texttt{j}$
`a(b,C,d,E)`	with	`a(b,f,G,H)`	yes: by either $\{\texttt{C} \mapsto \texttt{f}, \texttt{G} \mapsto \texttt{d}, \texttt{H} \mapsto \texttt{E}\}$ or $\{\texttt{C} \mapsto \texttt{f}, \texttt{G} \mapsto \texttt{d}, \texttt{E} \mapsto \texttt{H}\}$
`a(pred(X,j))`	with	`a(pred(k,j))`	yes: $\{\texttt{X} \mapsto \texttt{k}\}$
`a(pred(X,j))`	with	`a(Y)`	yes: $\{\texttt{Y} \mapsto \texttt{pred(X,j)}\}$
`a(pred(X,j))`	with	`a(X)`	yes: if `X`'s have different scope (change second `X` to `Y`, and do as above)
`a(pred(X,j))`	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: `append(X, Y, Z)` ≡ "`Z` is the result of appending lists `X` and `Y`"

```
append([ ], Y, Y).
append([H|X], Y, [H|Z]) :- append(X,Y,Z).
```

Difference lists:

```
contents(A, difflist(L,E)) :- append(A, E, L).
```

Representation invariant for difference lists: List `E` must be a suffix of list `L`

Abstraction function: given by the `contents` rule

More about difference lists

`difflist([a,b|E], E)` is a "difference list" consisting of `a` and `b`.

`E` could be anything: many difference lists consist of only `a` and `b`.

```
difflist([a,b], [ ])
difflist([a,b,c], [c])
difflist([a,b,c|F], [c|F])
```

Consing on to the front of a difference list:

```
cons(X, difflist(A,B), difflist([X|A],B)).
```

Extracting the contents of a difference list:

```
contents(A, difflist(L,E)) :- append(A,E,L).
```

Appending lists without recursion!

```
diffappend(difflist(X,Y), difflist(Y,Z), difflist(X,Z))
```

"$(X-Y)+(Y-Z)=(X-Z)$"

Backtracking example: computing with difference lists

Query: `contents(W, difflist([a,b|T], T))`

```
contents(W, difflist([a,b|T], T)) ; A=W, L=[a,b|T], E=T
 append(W,T,[a,b|T]) ; X=W, Y=T, Z=[a,b|T]
 append([ ], ...) NO! (occurs check)
 append([a|X'],T,[a,b|T]) ; W=[a|X'], ...
   append(X',T,[b|T])
   append([ ], ...) NO! (occurs check)
   append([b|X''],T,[b|T]) ; X'=[b|X''], ...
     append(X'',T,T)
     append([ ],T,T) ; X''=[ ], Y''=T
```

Solution:

`W = [a|X'] = [a,b|X''] = [a,b|[ ]] = [a,b]`

`T = Y`