Prolog II

- Unification
  - Informally
  - Formal description
  - Problems in compilation
- Factorial
  - Example of generate and test
  - Cut (!)

Trees

- Can use Prolog terms to represent trees
  \[ \text{times}(2, 3) \]
  - Then can design recursive Prolog clauses to “walk” the tree, gathering terms.
  - Example, generating code from an abstract syntax tree for an arithmetic expression
Example

treewalk(W,[W]) :- integer(W).
treewalk(times(X,Y),Walk) :- treewalk(X,W1),
   treewalk(Y,W2), append(W1,[*],A1),
   append(A1,W2,Walk).
treewalk(plus(X,Y), Walk):- treewalk(X,W1),
   treewalk(Y,W2), append(W1, [+],A1),
   append(A1,W2,Walk).
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).

```
times
  X
  Y
plus
  X
  Y
```

Prolog-II, BGR, Fall05

Generating Code from AST

This Prolog query produces code from the tree represented as a Prolog data structure (a term):

```
?-treewalk(plus(times(2,3),4),X).
```

A Prolog data structure:

```
plus(times(2,3),4)
```

This code represents the expression $2 \times 3 + 4$.

X = [2, *, 3, +, 4]

Note code generated here is a correct inorder traversal but will not generate correct expressions from the input because it ignores operator precedence.
How treewalk.pl works?

- Second argument is always the code which corresponds to the AST which is the first argument.
- Base case finds leaf nodes which are integer constants with Prolog built-in
  \[
  \text{treewalk}(W, [W]) :- \text{integer}(W).
  \]
- Tree exploration generates an inorder traversal of the nodes
- \text{Plus} and \text{times} clauses work the same.

---

How treewalk.pl works?

- First, explore left subtree and get its code bound to \(W1\) (left operand)
  \[
  \text{treewalk}(\text{times}(X,Y), \text{Walk}) :-
  \text{treewalk}(X, W1), \ldots
  \]
- Second, explore right subtree and get its code bound to \(W2\) (right operand)
  \[
  \ldots \text{treewalk}(Y, W2),\ldots
  \]
- Third, insert proper operator for this node \(\ldots\)
  \[
  \text{append}(W1, [\ast], A1), \ldots
  \]
- Fourth, append rest of expression
  \[
  \ldots \text{append}(A1, W2, \text{Walk}).
  \]
Unification Examples

unify(X,Y):- X = Y.
| ?- unify(a,X).
 X=a  ;
no
| ?- unify(a,X),unify(X,Y).
 X=Y=a  ;
no
| ?- unify(a,X),unify(b,Y),unify(X,Y).
 no
| ?- unify(X,Y).
 X=Y=_24  ;
no

Unification Examples

unify(X,Y):- X = Y.
| ?- unify(X,Y), unify(X,a).
 X=Y=a
| ?- unify(X,dummy(a)).
 X=dummy(a)
| ?- unify(X,dummy(a)),unify(X,Y).
 X=Y=dummy(a)
| ?- unify(X,dummy(Y)).
 X=dummy(Y),
 Y=_45  ;
no
Unification, Informally

- Intuitively, unification between 2 Prolog terms tries to associate values with the variables so that the resulting trees, representing the terms, are isomorphic (including matching labels)
- To use a Prolog rule, we must unify the head of the rule with the subgoal to be proved, “matching” term by term

Unification, Informally

- Given a subgoal `<functor>(<term>{, <term>})` how to unify it with a clause head?
  - Rule and subgoal have same name
  - Any uninstantiated variable matches any term
    - If term is also an uninstantiated variable, this means if either takes on a value, they both do
  - Integer and symbolic constants match themselves, only
  - A structured term matches another term iff
    - Has same relation name
    - Has same number of components (that is, terms within parentheses) and corresponding components match
  - Lists unify by matching element by element
Unification

- Unification looks for the most general (or least restrictive) value to assign
- A substitution ($\sigma$) is a finite map from variables to terms in the language

$$\text{append}([A|B], Y, [A|Z]) : -$$

Rule head

query $\sigma$: $A \rightarrow a$, $B \rightarrow [b]$, $Y \rightarrow [c]$, $W \rightarrow [a|Z]$

- A term $U$ is an instance of another term $T$, if there is a substitution $\sigma$ such that $U = T \sigma$.

Unification

- Two terms $S,T$ unify if they have a common instance $U$; that is,

$$S \sigma_1 = T \sigma_2 = U$$

- Note: if variable $X$ is contained in both $S$ and $T$, then $\sigma_1$ and $\sigma_2$ both must have the same substitution for $X$.
- If two terms unify, they can be made identical under some substitution
**Unification**

There may be more than one substitution to unify two terms

\[ \text{times}(Z, \text{times}(Y, 7)) \text{ and } \text{times}(4, W) \]

\( \sigma_1: Z = 4, Y = \text{plus}(3, 5), \]
\[ \quad W = \text{times}(\text{plus}(3, 5), 7) \]

\( \sigma_2: Z = 4, W = \text{times}(Y, 7) \)

Which substitution is simpler or less restrictive on the values of the variables? \( \sigma_2 \)

---

**Most General Unifier**

- We say \( \gamma \) is the *most general unifier (mgu)* of two terms, \( T \) and \( W \), iff for all other unifiers \( \sigma \) of \( T \) and \( W \), \( T\sigma \) is an instance of \( T\gamma \); therefore, \( \sigma \) can be obtained by a substitution \( \delta \) applied to \( \gamma \), \( \sigma = \gamma \cdot \delta \)

?\- member(A, B) returns \( A = \_123, B = [A| \_] \)
when it could return \( A = \_123, B = [A, b] \) or \( A = \_123, B = [A, c, d] \) etc. Note, the 2nd and 3rd B values are obtainable from the mgu by additional substitutions
Occurs Check

• There are problems with the unification done in some Prolog compilers, which result in an unbounded unification being attempted. Called an **occurs check**
  
  $[a,b \mid Z] = [X \mid Z]$  
  $X \rightarrow a, Z \rightarrow [b, Z]$

• If try to evaluate value of Z, compiler will return $Z = [b, b, b, ...]$ a value that results in an infinite loop in the Prolog interpreter

• Unification should check that it doesn’t unify a variable with a term containing that same variable

• **Occurs check** was left out of Prolog by Colmerauer because of efficiency (to avoid the run-time cost)
  
  – Current Prolog compilers have it
  – Example of safety yielding to efficiency ($O(n)$ instead of $O(n^2)$ on list concatenation)
Occurs Check  

Useful recursive type to build, a not-fully-evaluated list

\[ ?- \text{append}([\ ], E, [a, b|E]) \]

need to unify with \text{append}([\ ], A, A)
resulting in \(A \rightarrow E\) and \(A \rightarrow [a, b | E]\)

Can’t be built without occurs check

Generate and Test Paradigm

- Use of cut (!) to change evaluation order of Prolog clauses.
- Already saw cut in definition of \(\rightarrow\)
- A typical programming style in Prolog is \textit{generate and test}
  - Can write clauses to generate values and test if they satisfy the desired condition
  - Factorial example \textit{fact.pl}
  - N Queens example \textit{queens.pl}
Factorial

- Function to calculate X factorial if X is bound to an integer value

```prolog
factorial (0,1).
factorial(X,Y) :- W is X-1, factorial(W,Z), Y is Z*X.
```

If X is not bound to an integer value, then first subgoal (is clause) is undefined.

- A top-down calculation: \( n! = (n-1)! \cdot n \)

Factorial

- Add a guard to 2nd rule:

```prolog
factorial (0,1).
factorial(X,Y) :- integer(X), W is X-1, factorial(W,Z), Y is Z * X.
```

This builds \( f(n) \) from \( f(n-1) \), stepping down to \( f(0) \). If we query this new 2nd clause with \( \text{factorial}(Y,6) \), it will not match, but it will not abort, either.
Factorial

• How about a bottom-up definition?
  \[ f(0, 1) \]
  \[ f(X, Y) :- f(W, Z), \text{X is } W+1, \text{Y is } Z\times(W+1). \]

Here we calculate \( f(3, Y) \) by building it up from \( f(0, 1), f(1, 1), f(2, 2), f(3, 6) \).

• This new definition works for \( f(3, Y) \) and \( f(X, 6) \)
  but what about \( f(X, 5) \)? It will infinitely loop on
  this query. We need a way to control the
  backtracking mechanism, so it stops computation
  once a factorial value greater than 5 is returned.

Cut

• Cut (!)
  – Commits system to all choices made since the
    parent goal was invoked
  – If the parent predicate is re-entered by a
    backtracking computation, it cannot be re-
    satisfied. Instead a previous predicate must be re-
    satisfied.

\[
\text{eat\_lunch(joe, X)} :- \text{available(X)}, \text{cheap(X)}, !, \\
\text{sick(joe, X)}. \\
\text{use eat\_lunch predicate in another computation:} \\
... \text{eat\_lunch(joe, Y)}, ...
\]

If backtrack into \text{eat\_lunch}, can’t retry \text{available(X)} or \text{cheap(X)},
and can’t try another rule for \text{eat\_lunch(joe,Y)}. 

Factorial, finally

```
fact(0,1).
fact(X,Y):-fact(W,Z),X is W+1,Y is Z*(W+1).
f2(X,Y):-integer(Y),fact(W,Z),Z>=Y,!,Z=Y, W=X.  
f2(X,Y):-integer(X),var(Y),fact(X,Y),!.
f2(X,Y):-fact(X,Y).
```

Look at cases:

- `f2(int,var)` - uses 2nd `f2` rule for generation
- `f2(var or int, int)` - uses 1st `f2` rule to check (int,int) or generate (var, int)
- `f2(var,var)` - uses 3rd `f2` rule to generate factorial pairs