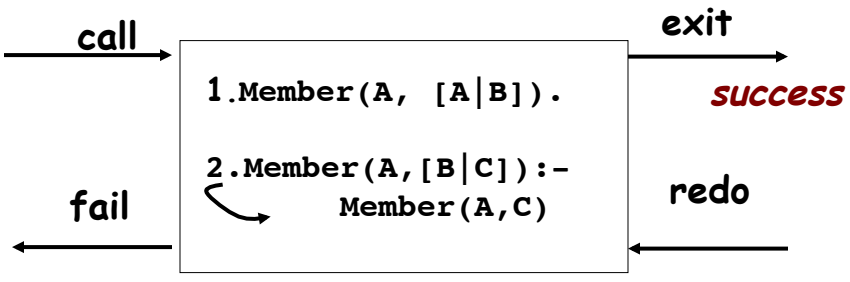
Prolog-2nd Lecture

- Tracing in Prolog
  - Procedural interpretation of execution
  - Box model of Prolog predicate rule
  - How to follow a Prolog trace?
- Trees in Prolog – use nested terms
- Unification
  - Informally
  - Formal description
  - Problems in compilation
- Cut (!) subgoal and how it changes control flow
- An Anomaly: Occurs check

Prolog Predicate - Box Model



This box represents 1 invocation w recursive rules; use one box per recursive subgoal called.

Example SWI-Prolog Trace

```
\> more memberAppend.pl
member(A,[A|B]).
member(A,[B|C]) :- member(A,C).
append([],A,A).
append([A|B],C,[A|D]) :- append(B,C,D).
\> swipl
?- consult("memberAppend.pl").
?- trace.
true.
?- [trace]  ?- member(a,[a,b,c]).
   Call: (7) member(a, [a, b, c]) ? creep
   Exit: (7) member(a, [a, b, c]) ? creep //R1
true.
```

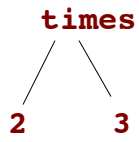


```
[trace]  ?- member(a,[b,c,X]).
   Call: (7) member(a, [b, c, _G1789]) ? creep //try R1
   Call: (8) member(a, [c, _G1789]) ? creep //try R2
   Call: (9) member(a, [_G1789]) ? creep //try R2
   Exit: (9) member(a, [a]) ? creep //R1
   Exit: (8) member(a, [c, a]) ? creep
   Exit: (7) member(a, [b, c, a]) ? creep
X = a ;  //find another answer
   Redo: (9) member(a, [_G1789]) ? creep // try R2
   Call: (10) member(a, []) ? creep //R2
   Fail: (10) member(a, []) ? creep
   Fail: (9) member(a, [_G1789]) ? creep
   Fail: (8) member(a, [c, _G1789]) ? creep
   Fail: (7) member(a, [b, c, _G1789]) ? creep
false.

[trace]  ?-
```

Trees in Prolog

- Can use Prolog terms to represent trees
  2 * 3 can be `times(2,3)`

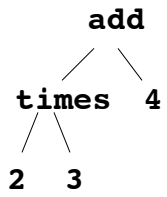


  - Then can design recursive Prolog clauses to "walk" the tree, gathering terms.
  - Example, generating code from an abstract syntax tree for an arithmetic expression

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(add(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).
```

Generating Code from AST



A Prolog data structure:
`add(times(2,3),4))`
representation for
2*3+4

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

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

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

Note code generated here is a correct in-order traversal but will not generate correct expressions from the input because it ignores operator precedence. You can also think of this as showing how to traverse the parse tree in DF order.

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

```
treewalk(W,[W]) :- integer(W).
```

- Tree exploration generates an in-order traversal of the nodes
- and `times` clauses work the same

How treewalk.pl works?

- First, explore left subtree and get its code bound to W1 (left operand)

```
treewalk(times(X,Y),Walk) :-
  treewalk(X,W1), ...
```

- Second, explore right subtree and get its code bound to W2 (right operand)

```
... treewalk(Y,W2),...
```

- Third, insert proper operator for this node

```
… append(W1,[*], A1), ...
```

- Fourth, append rest of expression

```
… append(A1,W2,Walk).
```


Unification Examples

```
unify(X,Y):- X = Y.
| ?- unify(a,X).
X = a.
| ?- unify(a,X),unify(X,Y).
X = Y = a.
| ?- unify(a,X),unify(b,Y),unify(X,Y).
false
| ?- unify(X,Y).
X = Y.
| ?- unify(X,Y), unify(X,a).
X = Y, Y = a.
| ?- unify(X,dummy(a)).
X = dummy(a).
| ?- unify(X,dummy(Y)).
X = dummy(Y).
```

Unification, Informally

- Intuitively, unification between 2 Prolog terms tries to associate values with the variables so that the resulting trees, are isomorphic, including matching labels

Unification, Informally

- Given a subgoal <functor>(<term>{, <term>}) how to unify it with a clause head?
  - Rule head 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* (σ) is a finite map from variables to terms in the language

```
append([A|B],Y,[A|Z]):-   Rule head
```

query `?- append([a,b],[c],W)`

σ: A → a, B → [b], Y → [c], W → [a | Z]

- A term U is an *instance* of another term T, if there is a substitution σ such that U = T σ.

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 σ_1 and σ_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

  `times(Z,times(Y,7)) and times(4,W)`

  σ_1: `Z=4, Y=plus(3,5),W=times(plus(3,5),7)`

  σ_2: `Z=4, W=times(Y,7)`

  Which substitution is simpler or less restrictive on the values of the variables? σ_2

Most General Unifier

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

  ?- `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

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.

```
eat_lunch(joe,X):-available(X),cheap(X),!,
                  sick(joe, X).
```

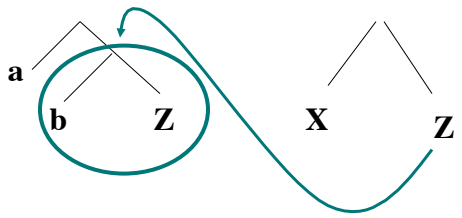
use `eat_lunch` predicate in another computation: ...

```
eat_lunch(joe,Y),...
```

If backtrack into `eat_lunch`, can't retry `available(X)` or `cheap(X)`, and can't try another rule for `eat_lunch(joe,Y)`.

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 |Z] = [X | Z] X → a, Z → [b, Z]



Occurs Check

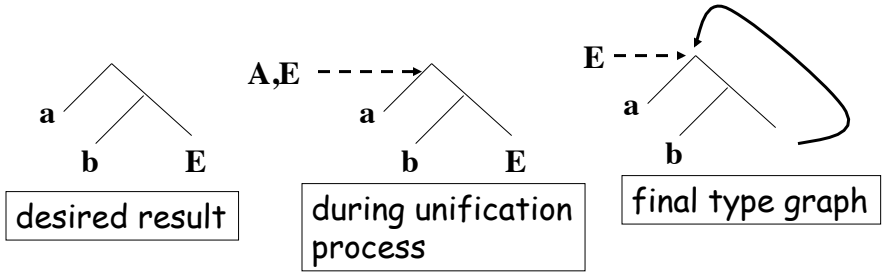
- If try to evaluate value of Z, compiler will return `Z=[b,b,b,...` a value that results in an infinite loop in the Prolog interpreter
- Unification should check that it doesn't unify a variable with a term containing that same variable
- Occurs check was left out of Prolog by Colmerauer because of 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

`?-append([ ],E,[a,b|E] )`

need to unify with `append([ ],A,A)` resulting in A → E and A → [a, b | E]



Can't be built without occurs check