Final Review

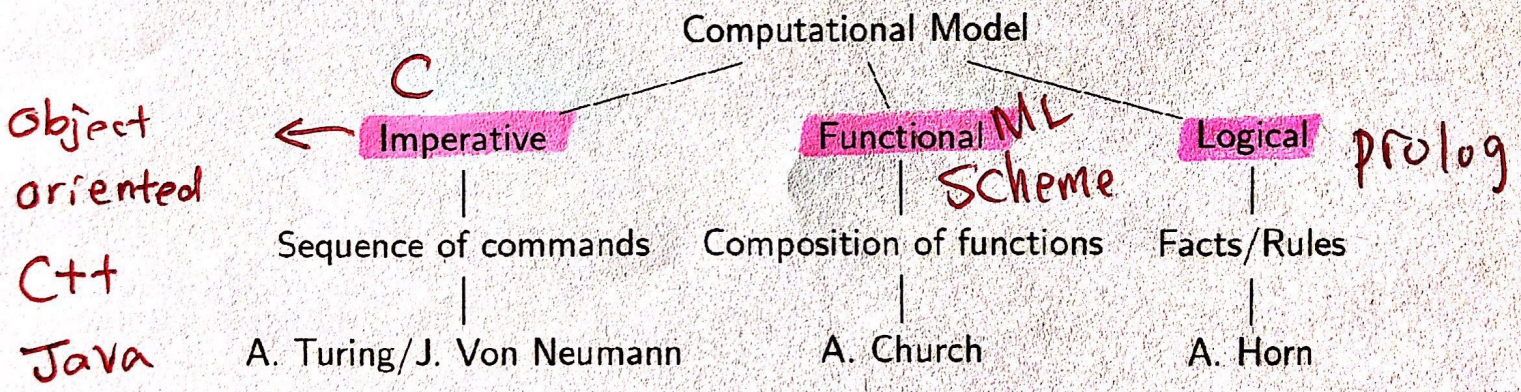
December 5, 2018

Course Objectives

Concepts

- terminologies of programming languages
- write programs in different programming paradigms
- programming language design decisions and implementation: from both syntax and semantics points of view
- theoretical foundations of programming languages such as grammar, formal semantics and lambda calculus

Programming Paradigms



Languages are designed as per the computation models
Langauges evolve to cut across different computation models

Example: Reverse a list

Imperative Programming

```
void reverse(struct node** head_ref)
{
    struct node* prev    = NULL;
    struct node* current = *head_ref;
    struct node* next;
    while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}
```

Functional Programming

```
(define (rev lst)
  (if (null? lst)
    lst
    (append (rev (cdr lst))
            (list (car lst))
    )
  )
)
```

Logic Programming

```
rev([], []).
rev([H|T], L) :-
    rev(T, T1),
    append(T1, [H], L).
```

Programming Language Features ~~and~~ design decisions and implementation

① Concepts in program language features

- Arithmetics: prefix/infix/postfix,
- Variable: a symbol to represent a value or a collection of values
- Def and use of a variable
- Scope: visibility of the variables, which definition to use?
- Bound/free variables: whether the variable is defined in the scope
- Heap: an abstraction representing dynamically allocated memory
- Reference: memory location of variables
- Types, typing/type rules, type systems: categories of values based on their ranges and operations on top of them
- Three parts of the Programming Languages:
  - Atomic computation: arithmetics, assignment
  - Composition: control flow
  - Abstraction: variables and function

first concepts
then examples

Var lang

(2) Concepts in

Language Implementation

- parse, parse tree, abstract syntax tree, grammar, CFG, (left/right most) derivations, ambiguity, operator precedence, associativity, type inference, type checking, environment (value, type), syntax (structure rules), semantics (how to evaluate program constructs to values, what are the rules to compose the program constructs)

- Compiler: Generate executables and run with input
- Interpreter: Parse (Read), Evaluate (Eval), Print loop

③ Concepts in

Functional Programming Languages

((twog)z)

- Side effect
- Currying, curried form
- High order functions: return a function or take a function as a parameters
- Lambda calculus: $\beta-$reduction (substitute formal parameters using actual parameters), $\alpha-$conversion (renaming)
- Evaluation order
- Church encoding: natural numbers, computation (+, -, pred, succ) on natural numbers, boolean values, and branches
- List and pair: car, cdr, cons, list

(a, [2-3])

Logic Programming

- Fact, rules and questions
- First order logic: predicate and quantifiers
- Unification and most general unifier
- Backtracking

Formal Definition of CFG

A CFG is a tuple $G = (\Sigma, V, S, P)$, where

- Σ is a set of terminals
- V is a set of non-terminals such that $\Sigma \cap V = \emptyset$
- $S \in V$ is a start non-terminal
- P is a set of product rules, each of the form: $X \longrightarrow \omega$, such that $X \in V$ and $\omega \in (\Sigma \cup V)^+$

Formal Definition of CFG: Real number example

A CFG for real number is a tuple $G = (\Sigma, V, S, P)$, where

- $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, .\}$
- $V = \{\texttt{real-number}, \texttt{part}, \texttt{digit}, \texttt{sign}\}$
- $S = \texttt{real-number}$
- P (Backus-Naur form/BNF)

$$
\begin{aligned}
\texttt{real-number} &\longrightarrow \texttt{sign part . part} \\
&\quad \mid \texttt{sign part} \\
\texttt{sign} &\longrightarrow \texttt{+} \mid \texttt{-} \\
\texttt{part} &\longrightarrow \texttt{digit} \mid \texttt{digit part} \\
\texttt{digit} &\longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Leftmost vs. Rightmost Derivation

Derivation of $1 + 2 + 3$

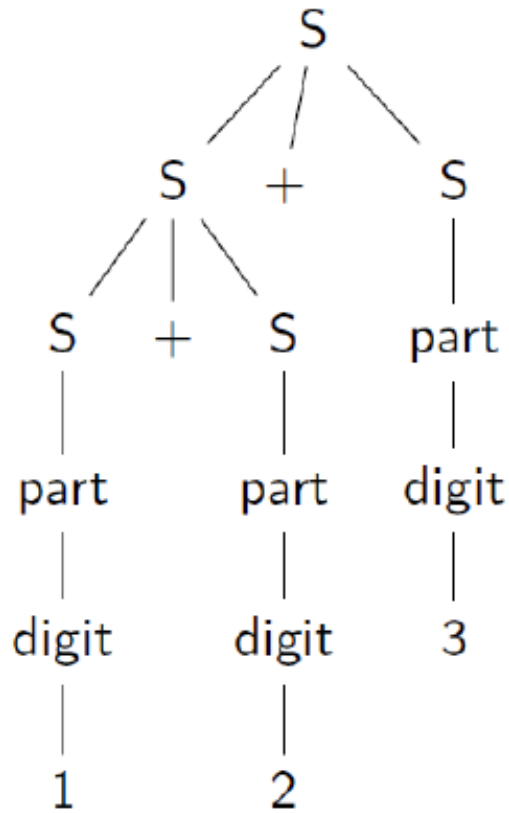
Leftmost	**Rightmost**
$S \rightarrow S + S$	$S \rightarrow S + S$
$\rightarrow$ part $+ S$	$\rightarrow S +$ part
$\rightarrow$ digit $+ S$	$\rightarrow S +$ digit
$\rightarrow 1 + S$	$\rightarrow S + 3$
$\rightarrow 1 + S + S$	$\rightarrow S + S + 3$
$\rightarrow 1 +$ part $+ S$	$\rightarrow S +$ part $+ 3$
$\rightarrow 1 +$ digit $+ S$	$\rightarrow S +$ digit $+ 3$
$\rightarrow 1 + 2 + S$	$\rightarrow S + 2 + 3$
$\rightarrow 1 + 2 +$ part	$\rightarrow$ part $+ 2 + 3$
$\rightarrow 1 + 2 +$ digit	$\rightarrow$ digit $+ 2 + 3$
$\rightarrow 1 + 2 + 3$	$\rightarrow 1 + 2 + 3$

Parse Tree

A parse tree results from the derivation sequence.

- Each node in the tree is a terminal or non-terminal in the production rule.
- Each edge in the tree from a non-terminal results from the application of production rule on the non-terminal.
- Application of production rule always result in new nodes in the tree.
- A terminal is a leaf node

Parse Tree for 1 + 2 + 3



Ambiguity

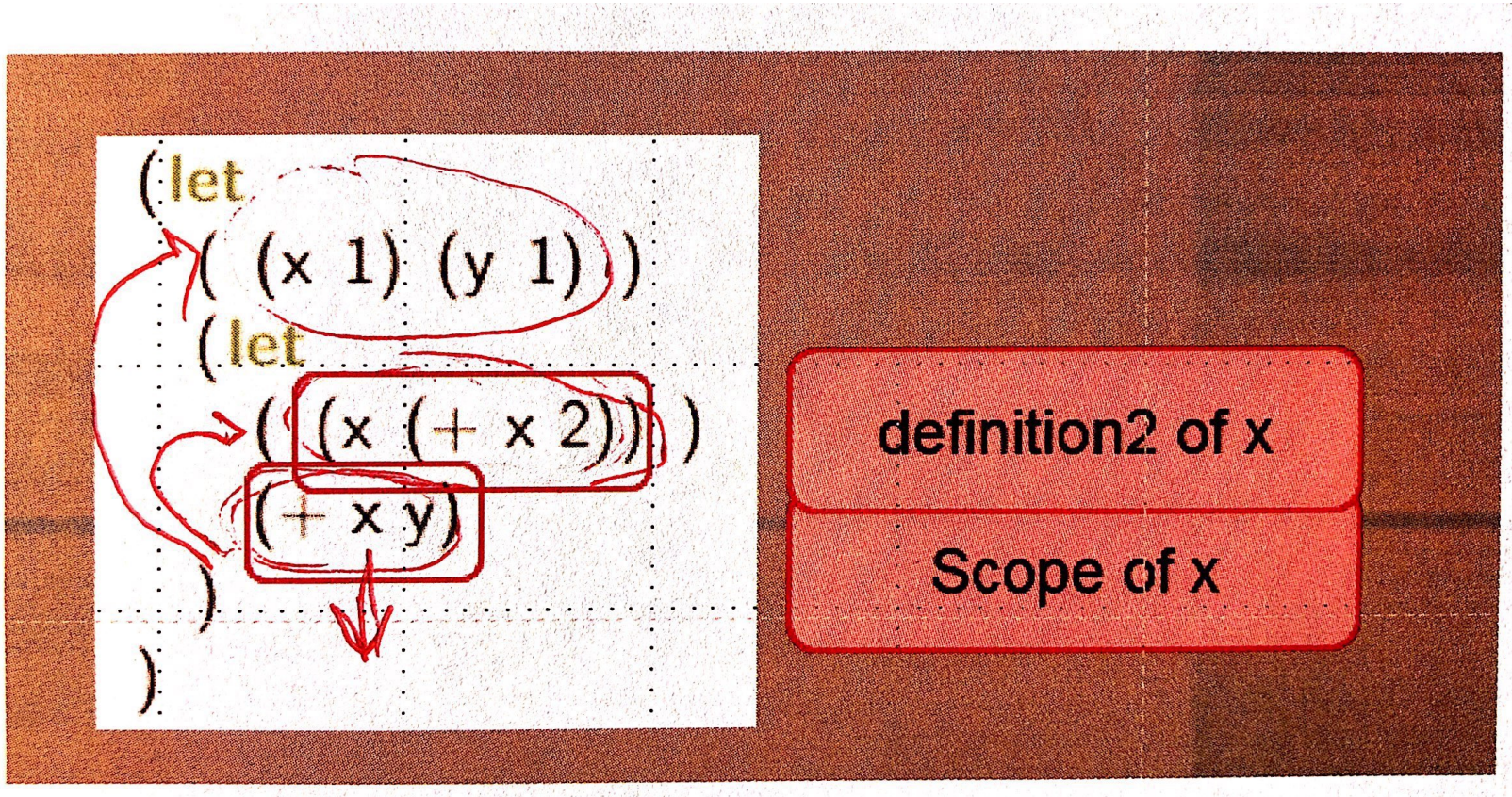
A grammar is ambiguous if there exists at least two distinct parse trees for the derivation of the same string.

digit $\longrightarrow$ 0 | 1 | ... | 9

part $\longrightarrow$ digit | digitpart

$S \longrightarrow$ part | S + S | S - S | S * S | S / S

- 1 + 2
- 1 + 2 + 3
- 1 + 2 - 3
- 1 + 2 * 3


(let
((x 1) (y 1))
(let
((x (+ x 2)))
(+ x y)
)
)
definition2 of x
Scope of x

- ***free variable***, a variable occurs free in an expression
  - if it is not defined by an enclosing let expression
- in program "*x*",
  - the variable x occurs free
- in program "*(let ((x 1)) x)*",
  - x is bound in enclosing let expression, hence x is not free

VALUE OF LETEXP

$$\frac{\texttt{value exp}_i\ \texttt{env}_0 = \texttt{v}_i, \text{for i} = 0...\text{k} \qquad \texttt{env}_{i+1} = (\texttt{ExtendEnv var}_i\ \texttt{v}_i\ \texttt{env}_i), \text{for i} = 0...\text{k} \qquad \texttt{value exp}_b\ \texttt{env}_{k+1} = \texttt{v}}{\texttt{value (LetExp (var}_i\ \texttt{exp}_i\texttt{),for i = 0...k exp}_b\texttt{) env}_0 = \texttt{v}}$$

Current Expression	Current Environment
(let ((x 1)) (let ((y 2)) (let ((x 3)) x)))	Empty
(let ((y 2)) (let ((x 3)) x))	$x \mapsto 1 :: \text{Empty}$
(let ((x 3)) x)	$y \mapsto 2 :: x \mapsto 1 :: \text{Empty}$
x	$x \mapsto 3 :: y \mapsto 2 :: x \mapsto 1 :: \text{Empty}$
3	$x \mapsto 3 :: y \mapsto 2 :: x \mapsto 1 :: \text{Empty}$
(let ((x 3)) 3)	$y \mapsto 2 :: x \mapsto 1 :: \text{Empty}$
(let ((y 2)) 3)	$x \mapsto 1 :: \text{Empty}$
(let ((x 1)) 3)	Empty
3	Empty

Examples: Calling the Lambda function

```
(                          //Begin function call syntax
  (lambda (x) x)           //Operator: function being called
  1                        //Operands: list of actual parameters
)                          //End function call syntax
```

```
(
  (lambda (x y) (+ x y))
  1 1
)
```

Examples: Combine with Let and Define

```
(let
   (( identity  (lambda (x) x)))          //Naming the function
   ( identity  1)                         //Function  call
)
```

```
$ (define square (lambda (x) (* x x)))
$ (square 1.2)
1.44
```

Pair and List

1. Pair: 2 tuple (fst, snd)
2. List: empty list, or 2 tuple
3. a list is a pair, a pair is not necessarily a list
4. Lists are constructed by using the cons keyword, as is shown here:
   > (cons 1 (list))
   (1)

Recursive Function

- Recursive function mirror the definition of the input data type

$$List := (list) \mid (cons\ \texttt{val}\ List),\ \texttt{where val} \in \texttt{Value}$$

```
(define append
  (lambda (lst1 lst2)
    (if (null? lst1) lst2
      (if (null? lst2) lst1
        (cons (car lst1) (append (cdr lst1) lst2))
      )
    )
  )
)
```

High Order Function with Data Structures

Rajan's book page 94:

```
(define pair
   (lambda (fst snd)
     (lambda (op)
       (if op fst snd)
     )
   )
)
(define apair (pair 3 4))
(define first (lambda (p) (p #t)))
$ (first apair)
```

Currying

Model multiple argument lambda abstractions as a combination of single argument lambda abstraction

```
(define plus
   (lambda (x y) (+ x y)))

(define plusCurry
   (lambda (x)
    (lambda (y)
      (+ x y)
    )
   )
)
```

Evaluate Call Expressions

```
(define identity
   (lambda (x) x)
)
$(identity i)
```

1. *Evaluate operator.* Evaluate the expression whose value will be the function value. For example, for the call expression `(identity i)` the variable expression `identity`'s value will be the function value.

2. *Evaluate operands.* For each expression that is in place of a formal parameter, evaluate it to a value. For example, for the call expression `(identity i)` the variable expression `i`'s value will be the only operand value.

3. *Evaluate function body.* This step has three parts.

   a) Find the expression that is the body of the function value,

   b) create a suitable environment for that body to evaluate, and

   c) evaluate the body.

Smallest Universal Programming Language

- A single transformation rule (variable substitution)
- A single function definition scheme
- $e ::= x|\lambda x.e|e0e1$

<expression>	:=	<name> | <function> | <application>
<function>	:=	λ <name>.<expression>
<application>	:=	<expression><expression>

As a programming language, sometimes a concrete implementation of lambda calculus also supports predefined constants such as '0' '1' and predefined functions such as '+' '*'; we add parenthesis for clarity

Formal Semantics of the Language

- $((_x\ \lambda\ (x)\ e_1\ )_x\ e_2)$: Evaluate the expression e_1 by replacing every ("free") occurrences of x in e_1 by e_2. I.e., $e_1[x \mapsto e_2]$ (β-reduction)

$((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)$

$(_y\ \lambda\ (y)\ (+\ x\ y)\ )_y[x \mapsto 1]$

$(_y\ \lambda\ (y)\ (+\ 1\ y)\ )_y$

Ordering in Evaluation

$$((_x \ \lambda\ (x)\ (+\ x\ 1)\ )_x ((_y \ \lambda\ (y)\ (+\ y\ 1)\ )_y \ 2\ )\ )$$

After β-reduction

Either $(+\ (\ (_y \ \lambda\ (y)\ (+\ y\ 1)\ )_y \ 2\ )\ 1)$

Or $(\ (_x \ \lambda\ (x)\ (+\ x\ 1)\ )_x (+\ 2\ 1)\ )$

Encoding Natural Numbers

zero $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ x\ )_x\ )_f$

one $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$

two $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ (f\ x))\ )_x\ )_f$

n $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ \ldots(f\ x)\ldots))\ )_x\ )_f$

Assume f is operation and x is the object on which the operation is done.

Example

What is the semantics of

- $((two\ g)\ z)$: two applications of g on z.

$$(((_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ (f\ x))\ )_x\ )_f\ g)\ z) = (g\ (g\ z))$$

- $((n\ g)\ z)$: n applications of g on z, where n is a natural number.
- $(_z\ \lambda\ (z)\ ((n\ g)\ z)\ )_z$: n applications of g on the formal parameter z, where n is a natural number. This result is a function (z if the formal parameter of the function).

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$(succ\ zero)$
$= ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n\ zero)$
$= \quad (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f$

$(zero\ f) = ((_g\ \lambda\ (g)\ (_y\ \lambda\ (y)\ y\ )_y\ )_g\ f) = (_y\ \lambda\ (y)\ y\ )_y$

Therefore,
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f =$
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((_y\ \lambda\ (y)\ y\ )_y\ x))\ )_x\ )_f =$
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f =$
one

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c$

$$
\begin{aligned}
&(((ite\ true)\ s_1)\ s_2) = \\
&(((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2) = \\
&\qquad (((_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((true\ t)\ e)\ )_e\ )_t\ s_1)\ s_2) = \\
&\qquad\qquad ((_e\ \lambda\ (e)\ ((true\ s_1)\ e)\ )_e\ s_2) = \\
&\qquad\qquad\qquad ((true\ s_1)\ s_2) = \\
&(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x\ s_1)\ s_2) = \\
&\qquad ((_y\ \lambda\ (y)\ s_1\ )_y\ s_2) = s_1
\end{aligned}
$$

RefLang

- Expressions for allocating a memory location, dereferences a location reference, assign a new value to an existing memory location, free previously allocated memory location
- Examples:
  $(ref 1)
  loc: 0
- Value: the location at which memory was allocated (next available memory location)
- Side effect: assign value 1 to the allocated memory location
- Value and type are known from the expression

Design Decisions – Heap

Heap size is finite, programming languages adopt strategies to remove unused portions of memory so that new memory can be allocated.

- manual memory management: the language provides a feature (e.g. free in C/C++) to deallocate memory and the programmer is responsible for inserting memory deallocation at appropriate locations in their programs.
- automatic memory management: the language does not provide explicit feature for deallocation. Rather, the language implementation is responsible for reclaiming ununsed memory (Java, C#).

How individual memory locations in the heap are treated:

- untyped heap: the type of value stored at a memory location is not fixed, can be changed during program execution
- typed heap: each memory location has an associated type and it can only contain values of that type, the type of value stored at a memory location doesn't change during the program's execution

Design Decisions – Reference (pointers)

1. Explicit references: references are program objects available to the programmer
2. Implicit references: references only available to implementation of the language
3. Reference arithmetic: references are integers and thus we can apply arithmetic operations
4. Deref and assignment only: get the value stored at that location in the heap, assignment can change the value stored at that location in the heap

Examples of Typelang: which one is correct, which one is incorrect?

- $ (define pi : num 3.14159265359)
- $ (define r : Ref num(ref : num 2))
- $ (define u : unit (free (ref : num 2)))
- $ (define iden : (num –> num) (lambda (x : num) x))
- $ (define id : (num –> num) (lambda (x : (num –> num)) x))
- $ (define fi : num ”Hello”)
- $ (define f : Ref num(ref : bool #f))
- $ (define t : unit (free (ref : bool #t)))

$$\frac{tenv \vdash e_i : \texttt{num}, \forall i \in 1..n}{tenv \vdash (AddExp\ e_0\ e_1\ \ldots\ e_n) : \texttt{num}}\ (\textsc{AddExp})$$

(LetExp)

$$\frac{\begin{array}{c} tenv \vdash e_i : t_i, \forall i \in 0..n \\ tenv_n = (ExtendEnv\ var_n\ t_n\ tenv_{n-1}) \ \ldots \\ tenv_0 = (ExtendEnv\ var_0\ t_0\ tenv) \\ tenv_n \vdash e_{body} : t \end{array}}{tenv \vdash (LetExp\ var_0, \ldots, var_n, t_0, \ldots, t_n, e_0, \ldots, e_n, e_{body}) : t}$$

Prolog

Facts

```
isamother(mary).          %% Horn Clause with no Antecedent
childof(tom, mary).       %% Horn Clause with no Antecedent
childof(jerry, mary).     %% Horn Clause with no Antecedent
```

Rules

```
%% Rule: Horn Clause with antecedent and with variables
%%       X and Y are universally quantified
loves(X, Y) :-
        isamother(X), childof(Y, X).

%%       X is universally quantified
%%       Y, Z are existentially quantified
hassibling(X) :-
        childof(X, Y), childof(Z, Y).
```

Query

```
%% Query: Horn Clause with no consequent
?- loves(mary, X). %% X is exitentially quantified
?- hassibling(jerry).
```

Logical Implications

- Prolog input: facts and rules (relations)
- Queries: does some inference hold?
- Proof by application logical implication

Unification

Given two atomic formula (predicates), they can be unified if and only if they can be made syntactically identical by replacing the variables in them by some terms.

- Unify `childof(jane, X)` and `childof(jane, mary)`?
  yes by replacing `X` by `mary`
- Unify `childof(jane, X)` and `childof(jane, Y)`?
  yes by replacing `X` and `Y` by the same individual
- Unify `childof(jane, X)` and `childof(Y, mary)`?
  yes by replacing `X` by `mary`, and `Y` by `jane`
- Unify `childof(jane, X)` and `childof(tom, Y)`? No.

Most General Unifier

MGU results from a substitution that bounds free variables as little as possible

- Unify `p(X, f(Y))` and `p(g(Z), W)`
  - $[\texttt{X} \mapsto \texttt{g(a)},\ \texttt{Y} \mapsto \texttt{b},\ \texttt{W} \mapsto \texttt{f(b)}]$
  - $[\texttt{X} \mapsto \texttt{g(Z)},\ \texttt{W} \mapsto \texttt{f(Y)}]$ MGU