CS3100 - Paradigms of Programming Languages

Introduction

V. Krishna Nandivada

IIT Madras

Academic Formalities

- Programming assignments = 4 x 08 marks,
- Project = 08 marks.
- Quiz 1 = Quiz 2 = 15 marks, Final = 30 marks. Absolute grading.
- Extra marks
  - During the lecture time - individuals can get additional 5 marks.
  - How? - Ask a good question, answer a chosen question, make a good point! Take 0.5 marks each. Max one mark per day per person.
- Attendance requirement – as per institute norms. Non compliance will lead to 'W' grade.
  - Proxy attendance - is not a help; actually a disservice.
- Plagiarism - A good word to know. A bad act to own.
  - Will be automatically referred to the institute welfare and disciplinary committee.

Contact (Anytime) :

Instructor: Krishna, Email: nvk@iitm.ac.in, Office: SSB 406.

TAs : Ramya K, Omkar D, Shashin H, Kranti I, Aoukhsana V C, Nilesh T, Poorbi M D, Rutuj K K, G Mahesh.

What, When and Why of Paradigms of Prog. (Langs)

- **What**:
  - Paradigm: A typical example or pattern.
  - Paradigms of Programming: Different patterns in Programming (languages)
- **When?**
  - When the first programming language was born. ??
    - Plankalküli, by Konrad Zuse, 1942 (not implemented at that time).
    - Short Code, John Mauchly 1950 (first implemented language).
    - Fortran, John Backus and team, 1954 (first widely available GP language+compiler)
- **Why? Study?**
  - A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer.
  - Get understanding of the intrinsic properties behind comp. langs.
  - Understand the relationships between the plethora of languages that you know/will learn.
  - Handy, if you care about the languages that you use/will learn.

Course outline

A rough outline (we may not strictly stick to this).

- Introduction to different paradigms of programming.
- Lambda calculus and functional programming.
- Logic programming
- Concurrent programming.
- Advanced topics (depending on time).

Your friends: Languages and Tools

Start exploring

- Java/C++ familiarity a must.
- Scheme - functional language
- Prolog - logic programming language
- Find the course webpage: http://www.cse.iitm.ac.in/~krishna/cs3100/

Get set. Ready steady go!

Expectations

What qualities are important in a programming language?

1. Should be easy to express the program logic.
2. Should reduce programming errors.
3. Code should run fast.
4. Should support modular compilation.
5. Should help in thinking...
6.

Each of these shapes your expectations about this course

A single instruction programming language

```
subleq a, b, c              ; Mem[b] = Mem[b] - Mem[a]
                            ; if Mem[b] <= 0 goto c
```

- If the optional third argument is missing:
  - the target branch is the address of the following instruction.

```
subleq a, b
```

$\equiv$

```
subleq a, b, L1
L1: ...
```

What does this program do?

Assume that `Mem[z]` contains zero.

```
subleq a, z
subleq z, b
subleq z, z
```

```
Mem[b] = Mem[a] + Mem[b]      ; add a, b
```


What does this program do?

```
subleq b, b
add a, b
```

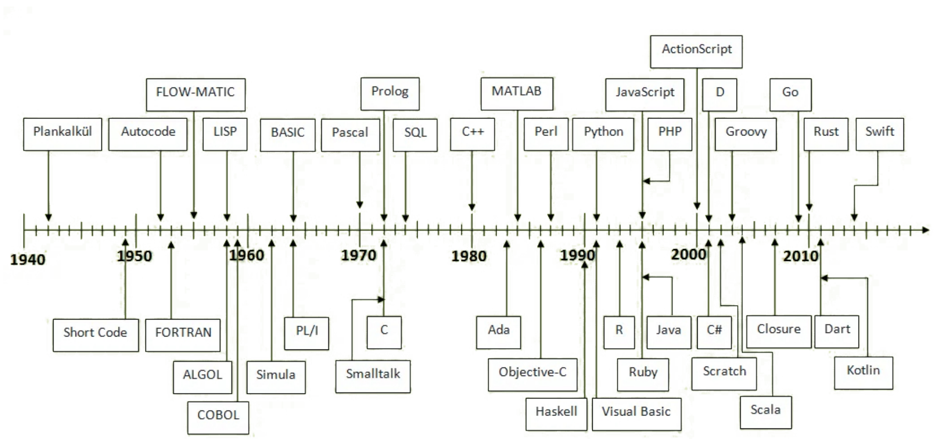
```
Mem[b] = Mem[a]      ; copy a, b
```

- One instruction set computer - read more about its power.

So many languages to choose from. . .

- What all languages do you know?
- Why are there so many languages?
- Has the last language already been developed?

A timeline of programming languages



New languages keep getting developed

There was no

- Java 30 years back.
- C# 20 years back
- Rust 15 years back
- Webassembly 5 years back.

What is the goal of CS3100?

Not necessarily to learn new languages. But to

- Understand the concepts that valid across multiple languages.
- Programming paradigms.
- Learn to a bit on how to design and implement languages.
- Understand the difference between the syntax and fundamental features.

Fun Question

- What is the difference between compilation and interpretation?

Programming Language Description

A computer language is defined in two dimensions:

- Syntax
  - Lexical
  - Syntactic
- Semantics

Example:

```
x = 3 + y;
```

Difference between Syntax and Semantics

Consider an expression language:

```
e := 0 | true | false
     | succ e | if e then e else e | e == e
```

Values in the language:

- `0`, `succ 0`, `succ succ 0` etc,
  - Short form $S^0, S^1, S^2, S^3, \ldots$
  - Numerals: 0, 1, 2, 3, ...
- `true`, and `false`

Valid semantics: Type checking a program

- A program is a closed expression.
- Valid types: *Int* and *Bool*

$$\frac{}{0 : Int}$$

$$\frac{}{false : Bool}$$

$$\frac{}{true : Bool}$$

Valid semantics: Type checking a program (cont.)

$$\frac{e1 : Int \qquad e2 : Int}{e1 == e2 : Bool}$$

$$\frac{e1 : Bool \qquad e2 : Bool}{e1 == e2 : Bool}$$

$$\frac{e : Int}{\texttt{succ}\ e : Int}$$

$$\frac{e : Bool \qquad e1 : t \qquad e2 : t}{\texttt{if}\ e\ \texttt{then}\ e1\ \texttt{else}\ e2 : t}$$

Semantics

- Interpretation of a program is the step by step procedure to reduce a given program to a value.

Interpretation of a program in our expression language

succ $\dfrac{e \to e'}{\texttt{succ}\ e \to \texttt{succ}\ e'}$

equality 1 $\dfrac{e1 \to e1'}{e1 == e2 \to e1' == e2}$

equality 2 $\dfrac{e1 \to e1'}{v == e1 \to v == e1'}$

equality 3 $\dfrac{}{v == v \to \texttt{true}}$

equality 4 $\dfrac{v = S^m \qquad v' = S^{m'} \qquad m \neq m'}{v == v' \to \texttt{false}}$

Semantics (cont.)

equality 5 $\dfrac{}{\texttt{true} == \texttt{true} \rightarrow \texttt{true}}$

equality 6 $\dfrac{}{\texttt{false} == \texttt{false} \rightarrow \texttt{true}}$

equality 7 $\dfrac{}{\texttt{true} == \texttt{false} \rightarrow \texttt{false}}$

equality 8 $\dfrac{}{\texttt{false} == \texttt{true} \rightarrow \texttt{false}}$

Semantics (cont.)

if 1 $\dfrac{e \rightarrow e'}{\texttt{if } e \texttt{ then } e1 \texttt{ else } e2 \rightarrow \texttt{if } e' \texttt{ then } e1 \texttt{ else } e2}$

if 2 $\dfrac{}{\texttt{if true then } e1 \texttt{ else } e2 \rightarrow e1}$

if 3 $\dfrac{}{\texttt{if false then } e1 \texttt{ else } e2 \rightarrow e2}$

Exercise

Consider a simple calculator language:

```
P := D S

D := Type Id; D | e

Type := int | float

S := Assignment | Expr

Assignment := Id = Expr

Expr := Id + Expr | Id - Expr |
        Id * Expr | Id / Expr | Constant
```

Write type rules. Write the operational semantics.

Imperative Programming

- Each program has state.
  - state = Memory/Storage
- Program consists of a series of actions.
- Each action may change the state of the program.
- imperative program: describes <u>how</u> a program operates step by step.

Well understood constructs

- Assignment
  - x = y
  - x = &y
  - x = e
  - x = A[e] or x = *y
  - A[e] = e or *x = y
- Conditional:
  - if-then
  - if-then-else
  - switch-case
  - Ambiguous operator
- Loops:
  - for
  - while
  - do-while
  - break/continue
- Functions: calls and return.

Ambiguous operator

```
let x = Amb(1, 2, 3)
let y = Amb(7, 6, 4, 5)
Amb(x * y = 8)
print x, y
```

One can use Amb operator to non-deterministically make a choice. Remember simulating an NFA.

Fun fact with `continue` operator.

```
i = 2; flag = 0;
do{
     i --;
     flag = flag | (i % 2);
     if (flag) continue;
     printf("i = %d, flag = %d, ", i, flag);

} while (i > 0);
```

Q: Output of the program?

1. Infinite loop
2. i = 1, flag = 1,
3. i = 1, flag = 1, i = 0, flag = 1,
4. No output

Fun code with loops

What is the use of code of the following form?

```
do {

...

} while (false);
```

We can use conditional break statements to jump out of the loop; Avoids goto statements.

Fun code with loops

Consider a Java loop nest of the following form:

```
for (i1...)
  for (i2...)
    for (i3...)
    ...
        for (ik...){
         if (c1)
           exit out of i1 loop
         if (c2)
           proceed with the next iteration of i2 loop

}
```

How to write such code? Goto statements
Using flag variables
break/continue with label.

Variant records

- Background: Each C struct has many fields.
- Background: size of struct = sum of size of all the fields.
- C Union:

```
union ifc_type {
     int ii;
     float ff;
     char cc;
};

union ifc_type x;
```

- How to know what does x contain?

Pascal Variant records

```
type kind = (i, f, c);    (* An enumeration type *)
  node = record (* Has intuitively two fields:
                  "k" and one of the remaining 3 *)
    case k: kind of
      i: (ii: integer);
      f: (ff: float);
      c: (cc: char);
    end;
```


Method calls (brief recollection)

- Parameter passing convention: Call-by value, call-by reference, textual substitution.
- C/Java supports call-by value semantics.
- C++ supports call-by reference.
- C macros support textual substitution.

Tail calls and their impact

- When the last statement executed in the body of a function is a call, such a function-call is called a “tail-call”.
- A function (or a program) is said to be in tail-call form, if every call is in the tail position.
- A tail-call can be replaced by a jump!
- A lot of impact can be seen in tail-recursive functions.

Illustration of tail call elimination

```
int search(int low, int high, int T, int X[]){
  int k;
  if (low > high) return -1; // NOT found
  k = (low + high) / 2;
  if (T == X[k]) return k;
  else if (T < X[k]) return search (low, k-1, T, X);
  else if (T > X[k]) return search (k+1, high, T, X);
}
```

```
int search(int low, int high, int T, int X[]){
  int k;
  Loop:
  if (low > high) return -1; // NOT found
  k = (low + high) / 2;
  if (T == X[k]) return k;
  else if (T < X[k]) {high = k-1;}
  else if (T > X[k]) {low = k+1;}
  goto Loop;
}
```


Outline

1 Introduction
- Formalities
- Overview

2 Imperative Languages
3 Object Oriented Languages
4 Functional Programming Languages
- The Scheme Language

5 Data types and Interpreters
- Interpreters: Stack machine
- Environments for an interpreter
- Cells for Variables
- Closures
- Recursive environments

6 Program Semantics
- Introduction
- Big Step Semantics
- Small Step Semantics

7 Typed Lambda Calculus
- Simply Typed Lambda Calculus

Scheme Language

An interpreted language.
A sample session: (the shell evaluates expressions)

```
$ mzscheme
Welcome to Racket v5.2.
> 3
3
> (+ 1 3)
4
> '(a b c)
(a b c)
> (define x 3)
> x 3
> (+ x 1)
4
```

```
> (define l '(a b c))
> l
(a b c)
> (define u '(+ x 1))
> u
(+ x 1)
> (define u (+ x  1))
> x
3
> u
4
>
```

Procedures

Creating procedures with lambda: `(lambda (x) body)`

```
> (lambda (x) (+ x 1))
#<procedure>
> ((lambda (x) (+ x 1)) 4)
5
> (define mysucc (lambda (x) (+ x 1)))
> (mysucc 4)
5
> (define myplus (lambda (x y) (+ x y)))
> (myplus 3 4)
7
> ((lambda (x y) (+ x y)) 3 4)
7
```

Procedures can take other procedures as arguments:

```
> ((lambda (f x) (f x 3)) myplus 5)
8
```

Q: How are C pointers different than a lambda?

Procedures (contd)

Procedures can return other procedures; this is called Currying:

```
> (define twice
  (lambda (f)
      (lambda (x)
            (f (f x)))))

> (define add2 (twice (lambda (x) (+ x 1))))

> (add2 5)

> 7
```


Kinds of data

- **Basic values** = Symbols ∪ Numbers ∪ Strings ∪ Lists
- **Symbols**: sequence of letters and digits starting with a letter. The sequence can also include other symbols, such as -,$,=,*,/,?, . Numbers: integers, etc.
- **Strings**: "this is a string"
- **Lists**:
  1. the empty list is a list ()
  2. a sequence $(s_1, \cdots s_n)$ where each s_i is a value (either a symbol, number, string, or list)
  3. nothing is a list unless it can be shown to be a list by rules (1) and (2).

  This is an inductive definition, which will play an important part in our reasoning. We will often solve problems (e.g., write procedures on lists) by following this inductive definition.

List Processing

Basic operations on lists:

- `car`: if `l` is $(s_1 \cdots s_n)$, then `(car l)` is s_1.
  - The car of the empty list is undefined.

  ```
  > (define l '(a b c))
  > (car l)
  > a
  ```
- `cdr`: if `l` is $(s_1\ s_2 \cdots\ s_n)$, then `(cdr l)` is $(s_2\ \cdots\ s_n)$.
  - The cdr of the empty list is undefined.

  ```
  > (cdr l)
  > (b c)
  ```

  Combining car and cdr:

  ```
  > (car (cdr l))
  > b
  > (cdr (cdr l))
  > (c)
  ```

Building lists

- `cons`: if `v` is the value `s`, and `l` is the list $(s_1 \cdots s_n)$, then `(cons s l)` is the list (`v` $s_1 \ \cdots \ s_n$).
- `cons` builds a list whose `car` is `s` and whose `cdr` is `l`.

```
(car (cons s l)) = v
(cdr (cons s l)) = l
```

```
cons : value * list -> list
car  : list -> value
cdr  : list -> list
```


Genesis of the names

- Lisp was originally implemented on the IBM 704 computer, in the late 1950s.
- The 704 hardware had special support for splitting a 36-bit machine word into four parts:
  1. an "address part" of fifteen bits,
  2. a "decrement part" of fifteen bits,
  3. a "prefix part" of three bits,
  4. a "tag part" of three bits.
- Precursors to Lisp included functions:
  1. car = "Contents of the Address part of Register number",
  2. cdr = "Contents of the Decrement part of Register number",
  3. cpr = "Contents of the Prefix part of Register number",
  4. ctr = "Contents of the Tag part of Register number".
- The alternate `first` and `last` are sometimes more preferred. But `car` and `cdr` have some advantages: short and compositions.
  - cadr = car cdr, caadr = car car cdr, cddr = cdr cdr
- `cons` = constructs memory objects.

Boolean related

Literals:

```
#t, #f
```

Predicates:

```
(number? s)                          (number?  3)
(symbol? s)                          (symbol? 'a)
(string? s)                          (string? "Hello")
(null? s)                            (null? '())
(pair? s)                            (pair? '(a . b))
(eq? s1 s2)     -- works on symbols  (eq? '(a b) '(a b))
                                     (eq? 'a 'a)
                                     (eq? "a" "a")
(equal? s1 s2) -- recursive          (equal? "a" "a")
(= n1 n2)       -- works on numbers  (= 2 2)
(zero? n)                            (zero? x)
(> n1 n2)                            (> 3 2)
```

Conditional:

```
(if bool e1 e2)
```


Building boolean predicates

- We can build complex boolean expressions:

```
(and v1 v2)

(or v1 v2)

(not v1)
```

Some special functions

Scheme supports `and`, `or`, `not` etc for making complex boolean expressions.
How to define them using what we already know?

```
(define (not x)
  (if x #f #t))

(define (and x y)
  (if x y #f))


(define (or x y)
  (if x #t y))
```

```
?
```

// Not exactly

Recursive Procedures

- Say we want to write the power function: $e(n,x) = x^n$.
- $e(n,x) = x \times e(n-1,x)$
- At each stage, we used the fact that we have the problem solved for smaller n — Induction.

Recursive procedures

```
(define e
  (lambda (n x)
    (if (zero? n)
        1
        (* x
           (e (- n 1) x)))))
```

Why does this work? Let's prove it works for any n, by induction on n:

1. It surely works for n=0.
2. Now assume (for the moment) that it works when `n = k`. Then it works when `n=k+1`. Why? Because `(e n x) = (* x (e k x))`, and we know `e` works when its first argument is `k`. So it gives the right answer when its first argument is `k + 1`.

Structural Induction

- **The Moral**: If we can reduce the problem to a smaller subproblem, then we can call the procedure itself ("recursively") to solve the smaller subproblem.
- Then, as we call the procedure, we ask it to work on smaller and smaller subproblems, so eventually we will ask it about something that it can solve directly (eg `n=0`, the basis step), and then it will terminate successfully.
- Principle of structural induction: If you always recur on smaller problems, then your procedure is sure to work.

Recursive procedures

```
(define fact
  (lambda (n)
      (if (zero? n) 1 (* n (fact (- n 1))))))

(fact 4) = (* 4 (fact 3))
         = (* 4 (* 3 (fact 2)))
         = (* 4 (* 3 (* 2 (fact 1))))
         = (* 4 (* 3 (* 2 (* 1 (fact 0)))))
         = (* 4 (* 3 (* 2 (* 1 1))))
         = (* 4 (* 3 (* 2 1)))
         = (* 4 (* 3 2))
         = (* 4 6)
         = 24
```

- Each call of `fact` is made with a promise that the value returned will be multiplied by the value of `n` at the time of the call; and
- thus `fact` is invoked in larger and larger control contexts as the calculation proceeds.

Loops

Java

```
int fact(int n) {
  int a=1;
  while (n!=0) {
    a=n*a;
    n=n-1;
  }
  return a;
}
```

Scheme

```
(define fact-iter
   (lambda (n)
      (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
   (if (zero? n)
     a
     (fact-iter-acc (- n 1)
                    (* n a)))))
```

Q: Is it not a recursive function?

Trace - loop

```
(fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

- `fact-iter-acc` is always invoked in the same context (in this case, no context at all).
- When `fact-iter-acc calls` itself, it does so at the "tail end" of a call to fact- iter-acc. That is, no promise is made to do anything with the returned value other than return it as the result of the call to `fact-iter-acc`.
- Thus each step in the derivation above has the form `(fact-iter-acc n a)`.

Many way switch - cond

```
(cond
  (test1 exp1)
  (test2 exp2)
  ...
  (else exp_n))
```

```
(cond
  ((equal? x 1) 'one)
  ((equal? x 2) 'two)
  (else 'large))
```

Let

When we need local names, we use the special form let:

```
(let ((var1 val1)
      (var2 val2)
      ...)
      exp)
```

```
(let ((x 3)
      (y 4))
(* x y))
```

```
(let ((x 5))
     (let ((f (+ x 3))
           (x 4))
          (+ x f)))
```


Limitations of let

Scheme

```
(define fact-iter
   (lambda (n)
      (fact-iter-acc n 1)))

(define fact-iter-acc
  (lambda (n a)
   (if (zero? n) a
     (fact-iter-acc (- n 1) (* n a)))))
```

Can we write a local recursive procedure?

```
(define fact-iter
  (lambda (n)
    (let ((fact-iter-acc
            (lambda (n a)
              (if (zero? n)
                   a
                   (fact-iter-acc (- n 1) (* n a))))))
      (fact-iter-acc n 1))))
```


Local recursive procedures

The scope of fact-iter-acc doesn't include it's definition. Instead, we can use `letrec`:

```
(letrec
 ((name1 proc1)
  (name2 proc2)
  ...)
  body)
```

`letrec` creates a set of mutually recursive procedures and makes their names available in the body. So we can write:

```
(define fact-iter
  (lambda (n)
    (letrec ((fact-iter-acc
               (lambda (n a)
                 (if (zero? n) a
                   (fact-iter-acc (- n 1)
                                  (* n a))
             ))))
             (fact-iter-acc n 1))))
```


Revise the scope of let and letrec

```
(let (var1 exp1) (var2 exp2) S)

var1 is visible inside S.
var2 is visible inside S.

(letrec (var1 exp1) (var2 exp2) S)

var1 is visible in exp1, exp2, and S.
var2 is visible in exp1, exp2, and S.
```

One requirement: no reference be made to `var1` and `var2` during the evaluation of `exp1`, and `exp2`.
This requirement is easily met if `exp1` and/or `exp2` are `lambda` expressions - reference to the variables `var1` and `var2` are evaluated only only when the resulting procedure is invoked.

Examples with let and letrec

```
(letrec ((x (+ x 1))) x)  -- undefined.

(let ((x 3))
   (letrec ((x (+ x 1))) x))  -- undefined.

(letrec ((x y) (y 1)) x) -- undefined

(letrec ((x (lambda () (+ y 1))) (y 3)) (x))   -- 4

(let ((x 2))
     (let ((x 3))
          (let ((y (+ x 4)))
               (* x y))))   = 21
```

$\not\equiv$

```
(let ((x 2))
     (let ((x 3)
           (y (+ x 4)))
          (* x y))) = 18
```


Argument sequencing

Arguments are evaluated before procedure bodies.

- In `((lambda (x y z) body) exp1 exp2 exp3)`

  exp1, exp2, and exp3 are guaranteed to be evaluated before body, but we don't know in what order exp1, exp2, and exp3 are going to be evaluated, but they will all be evaluated before body.
- This is precisely the same as
  `(let ((x exp1) (y exp2) (z exp3)) body)`

  In both cases, we evaluate exp1, exp2, and exp3, and then we evaluate body in an environment in which x, y, and z are bound to the values of exp1, exp2, and exp3.

Sample Problem

Q: Find the maximum number in a list.
A: Define a function that takes one argument (A list of numbers) and returns the largest element.
A: Write the recursive definition first.

```
(define largest
  (letrec ((largest-ele
             (lambda (ll e)
               (if (empty? ll)
                   e
                   (if (>= e (car ll))
                       (largest-ele (cdr ll) e)
                       (largest-ele (cdr ll) (car ll)))))))
    (lambda (ll)
      (if (empty? ll)
          'Empty-list
          (largest-ele (cdr ll) (car ll))))))
```


Sample problem (contd).

```
(define (largest2 nums)
  (cond ((empty? (cdr nums)) (car nums))
        (else
         (cond ((>= (car nums) (largest2 (cdr nums)))
                (car nums))
               (else
                (largest2 (cdr nums))))
        )))
```

Practice problems (with and without using letrec)

- Find the 'n' the element in a given list. (Input: a list and n. Output: error or the n'th element)
- symbol-only? – checks if a given list contains only symbols. List $\rightarrow$ boolean
- member?: (List, element) $\rightarrow$ boolean
- remove-first: List $\rightarrow$ List
- replace-first: (List, elem) $\rightarrow$ List
- remove-first-occurrence: (List, elem) $\rightarrow$ List
- remove-all-occurrences: (List, elem) $\rightarrow$ List

Merge sort

- Recall:
  - Create two halves.
  - Sort both halves
  - Merge

```
(define (merge list1 list2)
  (cond
    ((null? list1) list2)
    ((null? list2) list1)
    (else
     (let ((f1 (car list1))
           (f2 (car list2)))
       (if (<= f1 f2)
           (cons f1 (merge (cdr list1) list2))
           (cons f2 (merge list1 (cdr list2)))
           )))))

(merge (list  1 2 3 10) '(7 8 9))
```


Sorting - Insertion sort.

- Idea of insertion sort?
- Sort (A):
  - If A has zero or one element- it is already sorted.
  - Else
    - sort the tail,
    - insert the head at the appropriate place.
  - insert-appropriate (sortedList, elem): ?

```
(define sort (lambda (l)
  (letrec ((insert-appropriate (lambda (ll e)
    (cond ((empty? ll) (cons e ll))
     ((< e (car ll)) (cons e ll))
     (else (cons (car ll) (insert-appropriate (cdr ll) e)))
     ))))
   (cond ((empty? l) '())
      (else (insert-appropriate (sort (cdr l)) (car l)))
     ))))

(sort (list 2 1 4 56 23 21))
```


Sequencing in Scheme

- A common idiom in C/Java type of languages is specifying a sequence of statements.
- Sequencing using ";" is just a syntactic sugar!
- Can we simulate ";" operator using the scheme syntax we have learned so far?

```
S1;          ≡ (let ((x (S1)))
S2               (S2))
```

Sequencing in Scheme

- Scheme also supports an explicit sequencing operator

```
(begin (e1)
       (e2)
)

(let ()
  (begin
       (define x 20)
       (define y 22))
       (+ x y))
```


Additional control constructs

```
(when test-exp e1 e2 e3)

(until test-exp e1 e2 e3)
```

- `when`: If the `test-exp` is true, execute `e1`, `e2`, `e3` in sequence.
- `until`: If the `false` is false, execute `e1`, `e2`, `e3` in sequence.

switch-case

```
(let ([x 4] [y 5])
  (case (+ x y)
    [(1 3 5 7 9) 'odd]
    [(0 2 4 6 8) 'even]
    [else 'out-of-range]))
```

Q: What is the difference between `cond` and `case`?

Applying a procedure

- Goal: Given a list of arguments; apply them to an operator.
- Example: Apply “+” on `'(2 3)`

```
(apply + '(2 3))

(apply min '(1 8 0 2 5))

(define first
  (lambda (ll)
    (apply (lambda (x . y) x) ll)))

(define rest
  (lambda (ll)
    (apply (lambda (x . y) y) ll)))

(first '(a b c d))

(rest '(a b c d))
```

Mapping a function to a list

```
(map abs '(1 -2 3 -4 5 -6))
```

Q: How to define `map`?

```
(define map
  (lambda (f ll)
      (if (empty? ll) '(
                 (cons (f (car ll)) (map f (cdr ll))))))
(map abs (list 2 1 4 -56 23 21)))
```


Delayed execution

```
(delay expr) ; create a promise, won't be evaluated,
             ; till it is "forced".

(force promise-expr) ; evaluate the expression, first time.
                     ; Memoize after that.
```


Example, delayed computation

```
(define stream-car
  (lambda (s)
    (car (force s))))

(define stream-cdr
  (lambda (s)
    (cdr (force s))))

(define counters
  (letrec ((next (lambda (n)
                  (delay (cons n (next (+ n 1)))))))
    (next 1)))

(stream-car (stream-cdr counters)) ;

2
```


Example, delayed computation (contd.)

```
(define stream-add
  (lambda (s1 s2)
    (delay (cons
             (+ (stream-car s1) (stream-car s2))
             (stream-add (stream-cdr s1) (stream-cdr s2))))))

(define even-counters
  (stream-add counters counters))

(stream-car even-counters)
(stream-car (stream-cdr even-counters))

2 4
```

Folding

- Folding (a.k.a. reduce or accumulate) reduces a sequence of terms to a single term.
- Requires: a binary operator, an initial (or identity) value, and a sequence.
- We can fold left or right.

```
(define (fold-right f init-val ll)
  (if (null? ll)
      init-val
      (f (car ll)
         (fold-right f init-val (cdr ll)))))

(define (fold-left f init-val ll)
  (if (null? ll)
      init-val
      (fold-left f
                 (f init-val (car ll))
                 (cdr ll))))
```


Fold exampls

```
(fold-right + 0 '(1 2 3 4))
(fold-left - 0 '(1 2 3 4))

(+ 1 (+ 2 (+ 3 (+ 4 0))))
(- (- (- (- 0 1) 2) 3) 4)

(fold-left
  (lambda (a . args) (append args a))
  '()
  '(question the is be to not or be to))

to be or not to be is the question
```


Map vs Apply

- Map `(map ff ll)` : Invoke `ff` on each element of `ll` and return a list containing the results of each such invocation.
  - Example: `(map + '(1 2) '(3 4))` ;— `'(4 6)`
- Apply `(apply ff ll)`: Invoke `ff` by passing arguments given as a list in `ll`.
  - Another form of Apply: `(apply ff obj1 obj2 ...list)` ≡ `(ff obj1 obj2 elem-of-ll)`
  - Example `(apply + 1 -2 3 '(10 20))` ≡ `(+ 1 -2 3 10 20) ;--- 32`

```
(define whoami
  (lambda (ll)
    (apply map list ll))
)

(whoami '((1 2 3) (4 5 6)))
```

whoami = transpose

Values

```
(values)

(values 1) ;--- 1

(values 1 2 3) ;-- 1
                   2
                   3

(define head&tail
 (lambda (ls)
     (values (car ls) (cdr ls))))

(head&tail '(a b c))  ;--- a
                           (b c)
```

Call with values

```
(call-with-values producer consumer)

(call-with-values
  (lambda () (values 'bond 'james))
  (lambda (x y) (cons y x)))

;--- (james .  bond)
```


Outline

1. Introduction
   - Formalities
   - Overview
2. Imperative Languages
3. Object Oriented Languages
4. Functional Programming Languages
   - The Scheme Language
5. Data types and Interpreters
   - Interpreters: Stack machine
   - Environments for an interpreter
   - Cells for Variables
   - Closures
   - Recursive environments
6. Program Semantics
   - Introduction
   - Big Step Semantics
   - Small Step Semantics
7. Typed Lambda Calculus
   - Simply Typed Lambda Calculus
   - Parametric Polymorphism - System F

Data Types and their Representations

- Want to define new data types.
  - a specification - tells us what data (and what operations on that data) we are trying to represent.
  - implementation - tells us how we do it.
- We want to arrange things so that you can change the implementation without changing the code that uses the data type (user = client; implementation = supplier/server).
- Both the specification and implementation have to deal with two things: the data and the operations on the data.
- Vital part of the implementation is the specification of how the data is represented. We will use the notation $\lceil v \rceil$ for "the representation of data 'v'.

Numbers

- **Data specification**: Non negative numbers.
- **operations**:

$$\begin{array}{lcl} \texttt{zero} & = & \lceil 0 \rceil \\ (\texttt{is-zero?}\ \lceil n \rceil) & = & \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases} \\ (\texttt{succ}\ \lceil n \rceil\ ) & = & \lceil n+1 \rceil \\ (\texttt{pred}\ \lceil n+1 \rceil\ ) & = & \lceil n \rceil \end{array}$$

- **Extensions to do other operations**: Should work irrespective of the underlying representation.

```
(define plus
  (lambda (x y)
     (if (is-zero? x) y
          (succ (plus (pred x) y)))))
```

  - Irrespective of the representation (plus $\lceil x \rceil \lceil y \rceil$) = $\lceil x+y \rceil$

Scheme Representation of Numbers

$\lceil n \rceil$ = the Scheme integer n

```
(define zero 0)
(define is-zero? zero?)
(define succ (lambda (n) (+ n 1)))
(define prec (lambda (n) (- n 1)))
```


Unary Representation of Numbers

$\lceil 0 \rceil$ ()
$\lceil n+1 \rceil$ = (cons #t $\lceil n \rceil$)

- So the integer n is represented by a list of n #t's.
- Satisfy the specification:

```
(define zero = '())

(define is-zero? null?)

(define succ
  (lambda (n) (cons #t n)))

(define pred cdr)
```


Data Representation (contd). Example 2: Finite functions

- **Data specification**: a function whose domain is a finite set of Scheme symbols, and whose range is unspecified.
- **Specification of operation**: Aka - the interface

empty-ff = $\lceil \phi \rceil$
(apply-ff $\lceil f \rceil$ s) = $f(s)$
(extend-ff s v $\lceil f \rceil$) = $\lceil g \rceil$

where $g(s') = \begin{cases} v & s' = s \\ f(s') & \text{Otherwise} \end{cases}$

- Interface gives the type of each procedure and a description of the intended behavior of each procedure.

Procedural Representation

$f = \lceil \{(s_1, v_1), \ldots, (s_n, v_n)\} \rceil$ iff $(f\ s_i) = v_i$.
Implement the operations by:

```
(define apply-ff
  (lambda (ff z) (ff z)))

(define empty-ff
  (lambda (z)
    (error 'env-lookup
           (format "couldn't find ~s" z))))

(define extend-ff
  (lambda (key val ff)
    (lambda (z)
      (if (eq? z key)
        val
        (apply-ff ff z)))))
```

Procedural Representation

Examples

```
> (define ff-1 (extend-ff 'a 1 empty-ff))
> (define ff-2 (extend-ff 'b 2 ff-1))
> (define ff-3 (extend-ff 'c 3 ff-2))
> (define ff-4 (extend-ff 'd 4 ff-3))
> (define ff-5 (extend-ff 'e 5 ff-4))
> ff-5
<Procedure>
> (apply-ff ff-5 'd)
4
> (apply-ff empty-ff 'c)
error in env-lookup: couldn't find c.
> (apply-ff ff-3 'd)
error in env-lookup: couldn't find d.
>(define ff-new (extend-ff 'd 6 ff-4))
> (apply-ff ff-new 'd)
> 6
```


Association-list Representation

$\lceil\{(s_1,v_1),\ldots,(s_n,v_n)\}\rceil = ((s_1.v_1)\ldots(s_n.v_n))$

```
(define empty-ff '())

(define extend-ff
  (lambda (key val ff)
    (cons (cons key val) ff)))

(define apply-ff
  (lambda (alist z)
    (if (null? alist)
      (error 'env-lookup
                (format "couldn't find ~s" z))
      (let ((key (caar alist))
            (val (cdar alist))
            (ff (cdr alist)))
        (if (eq? z key) val (apply-ff ff z))))))
```


Association-list Representation

Examples

```
> (define ff-1 (extend-ff 'a 1 empty-ff))
> (define ff-2 (extend-ff 'b 2 ff-1))
> (define ff-3 (extend-ff 'c 3 ff-2))
> (define ff-4 (extend-ff 'd 4 ff-3))
> ff-4
((d . 4) (c . 3) (b . 2) (a . 1))
> (apply-ff ff-4 'd)
4
```

Useless Assignment: Specification and Implementation of Stack as a type.

Outline

1 Introduction
- Formalities
- Overview

2 Imperative Languages

3 Object Oriented Languages

4 Functional Programming Languages
- The Scheme Language

5 Data types and Interpreters
- Interpreters: Stack machine
- Environments for an interpreter
- Cells for Variables
- Closures
- Recursive environments

6 Program Semantics
- Introduction
- Big Step Semantics
- Small Step Semantics

7 Typed Lambda Calculus
- Simply Typed Lambda Calculus

Interpreters

The complexity of Interpreters depend on the language under consideration.

- Simple/Complex
- Environments
- Cells
- Closures
- Recursive Environments

Stack machine grammar

- **Goal**: interpreter for a stack machine.
- The machine will have two components: an action and a stack.
- The stack contains the data in the machine.
- We will represent the stack as a list of Scheme values, with the top of the stack at the front of the list.
- The action represents the instruction stream being executed by the machine.
-
```
Action ::= halt
         | incr; Action
         | add; Action;
         | push Integer ; Action
         | pop; Action
```
- Our interpreter - `eval-action`: takes an action and a stack and returns the value produced by the machine at the completion of the action.
- Convention: the machine produces a value by leaving it on the top of the stack when it halts.

Specification of Operations

Specification for `eval-action`. Our VM

- What `(eval-action a s)` does for each possible value of `a`.

```
(eval-action halt s) = (car s)

(eval-action incr; a (v w ...)) =
          (eval-action a (v+1 w ...))

(eval-action add; a (v w x ...)) =
          (eval-action a ((v+w) x ...))

(eval-action push v; a (w ...)) =
          (eval-action a (v w ...))

(eval-action pop; a (v w ...)) =
          (eval-action a (w ...))
```

- Is the specification complete? How to prove the same?

Representation of Operations

To write Scheme code to implement the specification of `eval-action`, we need to specify a representation of the type of actions. (Our bytecode).

- A simple choice - use lists.
$$\begin{array}{lcl} \lceil halt \rceil & = & \text{(halt)} \\ \lceil incr; a \rceil & = & \text{(incr . } \lceil a \rceil \text{)} \\ \lceil add; a \rceil & = & \text{(add . } \lceil a \rceil \text{)} \\ \lceil push\ v; a \rceil & = & \text{(push v . } \lceil a \rceil \text{)} \\ \lceil pop; a \rceil & = & \text{(pop . } \lceil a \rceil \text{)} \end{array}$$
- An action is represented as a list of instructions.
- Typical action is `(push 3 push 4 add halt)`

A Stack Machine Interpreter

```
(define eval-action
  (lambda (action stack)
    (let ((op-code (car action)))
      (case op-code
        ((halt)
         (car stack))
        ((incr)
         (eval-action (cdr action)
           (cons (+ (car stack) 1) (cdr stack))))
        ((add)
         (eval-action (cdr action)
           (cons (+ (car stack) (cadr stack)) (cddr stack))))
        ((push)
         (let ((v (cadr action)))
           (eval-action (cddr action) (cons v stack))))
        ((pop)
         (eval-action (cdr action) (cdr stack)))
        (else
          (error 'eval-action "unknown op-code:" op-code))))))
```


Interpreter in action

Running the Interpreter

```
> (define start
   (lambda (action)
     (eval-action action '())))


> (start '(push 3 push 4 add halt))
7
```


Outline

1 Introduction
- Formalities
- Overview

2 Imperative Languages

3 Object Oriented Languages

4 Functional Programming Languages
- The Scheme Language

5 Data types and Interpreters
- Interpreters: Stack machine
- Environments for an interpreter
- Cells for Variables
- Closures
- Recursive environments

6 Program Semantics
- Introduction
- Big Step Semantics
- Small Step Semantics

7 Typed Lambda Calculus
- Simply Typed Lambda Calculus

Interpreters (contd.): Environment

- An environment is a finite function - that maps identifiers to values.
- Why do we need an environment?
- **Specification**:

$$\begin{array}{lcl} \texttt{empty-Env} & = & \lceil \phi \rceil \\ (\texttt{apply-Env}\ \lceil f \rceil\ s) & = & f(s) \\ (\texttt{extend-Env}\ s\ v\ \lceil f \rceil) & = & \lceil g \rceil \end{array}$$

where $g(s') = \begin{cases} v & s' = s \\ f(s') & \text{Otherwise} \end{cases}$

Environment implementation

```
(define empty-env
  (lambda () '()))

(define extend-env
  (lambda (id val env)
    (cons (cons id val) env)))


(define apply-env
  (lambda (env id)
    (if (or (null? env) (null? id))
      null
      (let ((key (caar env))
            (val (cdar env))
            (env-prime (cdr env)))
        (if (eq? id key) val (apply-env env-prime id))))))

(define extend-env-list
  (lambda (ids vals env) ... )
```


extend-env-list

```
(define extend-env-list
  (lambda (ids vals env)
    (if (null? ids)
        env
        (extend-env-list
          (cdr ids)
          (cdr vals)
          (extend-env (car ids) (car vals) env)))))
```

Home reading: Read Scheme alist representation and see how the above routines can be compacted.

Interpreter with environment

```
(define eval-Expression
  (lambda (Expression)
    (record-case Expression
      ...
      (PlusExpression  (Tkn1 Tkn2 Expression1 Expression2 Tkn3)
        (+ (eval-Expression Expression1)
           (eval-Expression Expression2))))

      (Identifier (Token) (apply-env env Token))
     ... ))

(define run
  (lambda ()
       (record-case root
          (Goal (Expression Token)
          (eval-Expression Expression (empty-env)))
    (else (error 'run ``Goal not found''))))))
```


Extending an environment - let expression

```
(LetExpression (Token1 Token2 Token3
                        List Token4 Expression Token5)
  (let* ((ids  (get-ids List))
         (exps (get-exprs List))
         (vals (map (lambda (Expression)
                      (eval-Expression Expression env))
                     exps))
         (new-env (extend-env-list ids vals env)))
    (eval-Expression Expression new-env)))
```

```
> (map cdr '((1 2 3) (3 4 5)))
((2 3) (4 5))
```

Useless assignment: How to interpret Let*?

Outline

1. Introduction
   - Formalities
   - Overview
2. Imperative Languages
3. Object Oriented Languages
4. Functional Programming Languages
   - The Scheme Language
5. Data types and Interpreters
   - Interpreters: Stack machine
   - Environments for an interpreter
   - Cells for Variables
   - Closures
   - Recursive environments
6. Program Semantics
   - Introduction
   - Big Step Semantics
   - Small Step Semantics
7. Typed Lambda Calculus
   - Simply Typed Lambda Calculus
   - Parametric Polymorphism - System F

Update to variables

- One undesirable feature of Scheme: assignment to variables.
- A variable has a name and address where it stores the value, which can be updated.

```
(define make-cell
  (lambda (value)
    (cons '*cell value)))

(define deref-cell cdr)

(define set-cell! set-cdr!)
```

- When we extend an environment, we will create a cell, store the initial value in the cell, and bind the identifier to the cell.

```
(define extend-env
  (lambda (id value env)
    (cons (id (make-cell value)) env)))
```


Example: Interpreting a let expression

```
(let ((x 7))
  (+ (let ((y x)
           (x (+ 2 x)))
         (* x y)) x)
```


Outline

1. Introduction
   - Formalities
   - Overview
2. Imperative Languages
3. Object Oriented Languages
4. Functional Programming Languages
   - The Scheme Language
5. Data types and Interpreters
   - Interpreters: Stack machine
   - Environments for an interpreter
   - Cells for Variables
   - Closures
   - Recursive environments
6. Program Semantics
   - Introduction
   - Big Step Semantics
   - Small Step Semantics
7. Typed Lambda Calculus
   - Simply Typed Lambda Calculus
   - Parametric Polymorphism - System F

Include headers in Scheme

```
(load "recscm.scm")
(load "records")
(load "tree")
```


Closures

To represent user-defined procedures, we will use closures.

```
(define-record closure (formals body env))
```


Closures

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
     ...
     (ProcedureExp (Token1 Token2 Token3
                    List Token4 Expression Token5)
       (make-closure List Expression env))
     (Application (Token1 Expression List Token2)
       (let*
         ((clos (eval-Expression Expression env))
          (ids  (get-formals clos))
          (vals (map (lambda (Exp)
                       (eval-Expression Exp env))
                     List))
          (static-env (get-closure-env clos))
          (new-env
            (extend-env-list ids vals static-env)))
          (body (get-body clos))
        (eval-Expression body new-env)))
     ...)))
```


If Stmt

```
(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
     ...

     (IfExpression (Token1 Token2 Expression1
                    Expression2 Expression3 Token3)
       (if (eval-Expression Expression1 env)
           (eval-Expression Expression2 env)
           (eval-Expression Expression3 env)
     ...)))
```

Outline

1 Introduction
- Formalities
- Overview

2 Imperative Languages

3 Object Oriented Languages

4 Functional Programming Languages
- The Scheme Language

5 Data types and Interpreters
- Interpreters: Stack machine
- Environments for an interpreter
- Cells for Variables
- Closures
- Recursive environments

6 Program Semantics
- Introduction
- Big Step Semantics
- Small Step Semantics

7 Typed Lambda Calculus
- Simply Typed Lambda Calculus
- Parametric Polymorphism - System F

Recursive Environments for recursive definitions

- We need two kinds of environment records.
  - Normal environments contain cells.
  - A recursive environment contains a RecDeclarationList. If one looks up a recursively-defined procedure, then it gets closed in the environment frame that contains it:

```
(define-record normal-env (ids vals env))

(define-record rec-env (recdecl-list env))

(define eval-Expression
  (lambda (Expression env)
    (record-case Expression
      ...
      (RecExpression (Token1 Token2 Token3
                      List Token4 Expression Token5)
        (eval-Expression
          Expression
          (make-rec-env List env)))
      (else (error ...)))))
```


Recursive environments

```
(define apply-env
 (lambda (env id)
   (record-case env
     ...
     (rec-env (recdecl-list old-env)
       (let ((id-list (get-ids recdecl-list)))
         (if (member? id id-list)
           (let* ((RecProc (get-decl id recdecl-list))
                  (ProcExpr (get-proc-expr RecProc)))

             (make-cell (make-closure  ;; a cell
                  (get-formals ProcExpr)
                  (get-body ProcExp) env)))
           (apply-env old-env id)))))))
```


Last class

Interpreters

- A Environment
- B Cells
- C Closures
- D Recursive environments
- E Interpreting OO (MicroJava) programs.

Outline

1. Introduction
   - Formalities
   - Overview
2. Imperative Languages
3. Object Oriented Languages
4. Functional Programming Languages
   - The Scheme Language
5. Data types and Interpreters
   - Interpreters: Stack machine
   - Environments for an interpreter
   - Cells for Variables
   - Closures
   - Recursive environments
6. Program Semantics
   - Introduction
   - Big Step Semantics
   - Small Step Semantics
7. Typed Lambda Calculus
   - Simply Typed Lambda Calculus
   - Parametric Polymorphism - System F

Introduction

- An interpreter executes a program as per the semantics.
- An interpreter can be viewed as an executable description of the semantics of a programming language.
- Program semantics is the field concerned with the rigorous mathematical study of the meaning of programming languages and models of computation.
- Formal ways of describing the programming semantics.
  - Operational semantics - execution of programs in the language is described directly (in the context of an abstract machine).
    - Big-step semantics (with environments) -is close in spirit to the interpreters we have seen earlier.
    - Small-step semantics (with syntactic substitution) - formalizes the inlining of a procedure call as an approach to computation.
  - Denotational Semantics - each phrase in the language is translated to a denotation - a phrase in some other language.
  - Axiomatic semantics - gives meaning to phrases by describing the logical axioms that apply to them.

Lambda Calculus

- The traditional syntax for procedures in the lambda-calculus uses the Greek letter λ (lambda), and the grammar for the lambda-calculus can be written as:

$$e ::= x \mid \lambda x.e \mid e_1 e_2$$
$$x \in \text{Identifier (infinite set of variables)}$$

- Brackets are only used for grouping of expressions. Convention for saving brackets:
  - that the body of a λ-abstraction extends "as far as possible."
  - For example, $\lambda x.xy$ is short for $\lambda x.(xy)$ and not $(\lambda x.x)y$.
  - Moreover, $e_1e_2e_3$ is short for $(e_1e_2)e_3$ and not $e_1(e_2e_3)$.

Extension of the Lambda-calculus

We will give the semantics for the following extension of the lambda-calculus:

$$e ::= x \mid \lambda x.e \mid e_1 e_2 \mid c \mid succ\ e$$
$$x \in \text{Identifier (infinite set of variables)}$$
$$c \in \text{Integer}$$

Outline

1. Introduction
   - Formalities
   - Overview
2. Imperative Languages
3. Object Oriented Languages
4. Functional Programming Languages
   - The Scheme Language
5. Data types and Interpreters
   - Interpreters: Stack machine
   - Environments for an interpreter
   - Cells for Variables
   - Closures
   - Recursive environments
6. Program Semantics
   - Introduction
   - Big Step Semantics
   - Small Step Semantics
7. Typed Lambda Calculus
   - Simply Typed Lambda Calculus
   - Parametric Polymorphism - System F

Big step semantics

Here is a big-step semantics with environments for the lambda-calculus.

$$\begin{array}{lcl} w, v & \in & Value \\ v & ::= & c \mid (\lambda x.e, \rho) \\ \rho & \in & Environment \\ \rho & ::= & x_1 \mapsto v_1, \cdots x_n \mapsto v_n \end{array}$$

The semantics is given by the following five rules:

(1) $$\rho \vdash x \rhd v \;\; (\rho(x) = v)$$

(2) $$\rho \vdash \lambda x.e \rhd (\lambda x.e, \rho)$$

(3) $$\frac{\rho \vdash e_1 \rhd (\lambda x.e, \rho') \quad \rho \vdash e_2 \rhd v \quad \rho', x \mapsto v \vdash e \rhd w}{\rho \vdash e_1 e_2 \rhd w}$$

(4) $$\rho \vdash c \rhd c$$

(5) $$\frac{\rho \vdash e \rhd c_1}{\rho \vdash \texttt{succ}\ e \rhd c_2} \quad \lceil c_2 \rceil = \lceil c_1 \rceil + 1$$

Outline

1. Introduction
   - Formalities
   - Overview
2. Imperative Languages
3. Object Oriented Languages
4. Functional Programming Languages
   - The Scheme Language
5. Data types and Interpreters
   - Interpreters: Stack machine
   - Environments for an interpreter
   - Cells for Variables
   - Closures
   - Recursive environments
6. Program Semantics
   - Introduction
   - Big Step Semantics
   - Small Step Semantics
7. Typed Lambda Calculus
   - Simply Typed Lambda Calculus
   - Parametric Polymorphism - System F

Small step semantics

- In small step semantics, one step of computation = either one primitive operation, or inline one procedure call.
- We can do steps of computation in different orders:

```
> (define foo
     (lambda (x y) (+ (* x 3) y)))
> (foo (+ 4 1) 7)
22
```

Let us calculate:

```
(foo (+ 4 1) 7)
=>   ((lambda (x y) (+ (* x 3) y))
      (+ 4 1) 7)
=>   (+ (* (+ 4 1) 3) 7)
=> 22
```

Small step semantics (contd.)

We can also calculate like this:

```
(foo
(+ 4 1) 7)

=> (foo 5 7)

=>   ((lambda (x y) (+ (* x 3) y))
       5 7)

=> (+ (* 5 3) 7)

=> 22
```


Free variables

A variable x occurs free in an expression E iff x is not bound in E.Examples:

- no variables occur free in the expression

```
(lambda (y) ((lambda (x) x) y))
```

- the variable y occurs free in the expression

```
((lambda (x) x) y)
```

An expression is closed if it does not contain free variables.
A program is a closed expression.

Methods of procedure application

Call by value

```
((lambda (x) x)
((lambda (y) (+ y 9)) 5))

=> ((lambda (x) x) (+ 5 9))

=> ((lambda (x) x) 14)

=> 14
```

Always evaluate the arguments first

- Example: Scheme, ML, C, C++, Java

Methods of procedure application

Call by name (or lazy-evaluation)

```
((lambda (x) x)
          ((lambda (y) (+ y 9) 5))

=> ((lambda (y) (+ y 9)) 5)

=> (+ 5 9)

=> 14
```

Avoid the work if you can

- Example: Miranda and Haskell

Lazy or eager: Is one more efficient? Are both the same?

Difference

- Q: If we run the same program using these two semantics, can we get different results?
- A:
  - If the run with call-by-value reduction terminates, then the run with call- by-name reduction terminates. (But the converse is in general false).
  - If both runs terminate, then they give the same result.

Church Rosser theorem

$$\begin{tikzcd} & E \arrow[dl] \arrow[dr] & \\ E1 \arrow[dr] & & E2 \arrow[dl] \\ & V & \end{tikzcd}$$

Call by value - too eager?

Sometimes call-by-value reduction fails to terminate, even though call-by- name reduction terminates.

```
   (define delta (lambda (x) (x x)))
    (delta delta)
=>  (delta delta)
=>  (delta delta)
=>  ...
```

Consider the program:

```
   (define const (lambda (y) 7))
   (const (delta delta))
```

- call by value reduction fails to terminate; cannot finish evaluating the operand.
- call by name reduction terminates.

Summary - calling convention

- call by value is more efficient but may not terminate
- call by name may evaluate the same expression multiple times.
- Lazy languages uses - call-by-need.
- Languages like Scala allow both call by value and name!

Beta reduction

- A procedure call which is ready to be "inlined" is called a beta-redex. Example `( (lambda (var) body ) rand )`
- In lambda-calculus call-by-value and call-by-name reduction allow the choosing of arbitrary beta-redex.
- The process of inlining a beta-redex for some reducible expression is called beta-reduction.

  `( (lambda (var) body ) rand ) -> body[var:=rand]`
- η conversion: A simple optimization:

$$(\lambda x\,(E\;x)) \;=\; E$$

- A conversion when applied in the left-to-right direction is called a reduction.

Notes on reduction

- Applicative order reduction - A β reduction can be applied only if both the operator and the operand are already values. Else?
- Applicative order reduction (call by value), example: Scheme, C, Java.

Notes on reduction

- Is there a reduction strategy which is guaranteed to find the answer if it exists? – leftmost reduction (lazy evaluation).
- leftmost-reduction – reduce the β-redex whose left parenthesis comes first
- A lambda expression is in normal form if it contains no β-redexes.
- An expression in normal form – cannot be further reduced. e.g. constant or (lambda (x) x)
- Church-Rosser theorem $\rightarrow$ expression can have at most one normal form.
- leftmost reduction will find the normal form of an expression if one exists.

Name clashes

- Care must be taken to avoid name clashes. Example:

```
((lambda (x)
   (lambda (y) (y x)))
  (y 5))
```

  should not be transformed into

```
(lambda (y) (y (y 5)))
```

- The reference to y in (y 5) should remain **free**!
- The solution is to change the name of the inner variable name `y` to some name, say `z`, that does not occur free in the argument `y 5`.

```
  ((lambda (x)
     (lambda (z) (z x)))
   (y 5))

=>  (lambda (z) (z (y 5))) ;; y is free.
```


Substitution

- The notation $e[x := M]$ denotes e with M substituted for every free occurrence of x in such that a way that name clashes are avoided.
- We will define $e[x := M]$ inductively on e.

$x[x := M]$	$\equiv$	M
$y[x := M]$	$\equiv$	$y\ (x \neq y)$
$(\lambda x.e_1)[x := M]$	$\equiv$	$(\lambda x.e_1)$
$(\lambda y.e_1)[x := M]$	$\equiv$	$\lambda z.((e_1[y := z])[x := M])$ (where $x \neq y$ and z does not occur free in e_1 or M).
$(e_1 e_2)[x := M]$	$\equiv$	$(e_1[x := M])(e_2[x := M])$
$c[x := M]$	$\equiv$	c
$(succ\ e_1)[x := M]$	$\equiv$	$succ\ (e_1[x := M])$

- The renaming of a bound variable by a fresh variable is called alpha-conversion.
- Q: Can we avoid creating a new variable in the fourth rule ?

Small step semantics

Here is a small-step semantics with syntactic substitution for the lambda-calculus.

$$v \in Value$$
$$v ::= c \mid \lambda x.e$$

The semantics is given by the reflexive, transitive closure of the relation $\rightarrow_V$

$$\rightarrow_V \subseteq Expression \times Expression$$

(6) $$\lambda x.e\ v \rightarrow_V e[x := v]$$

(7) $$\frac{e_1 \rightarrow_V e_1'}{e_1 e_2 \rightarrow_V e_1' e_2}$$

(8) $$\frac{e_2 \rightarrow_V e_2'}{v e_2 \rightarrow_V v e_2'}$$

(9) $$\mathtt{succ} c_1 \rightarrow_V c_2 (\lceil c_2 \rceil = \lceil c_1 \rceil + 1)$$

(10) $$\frac{e_1 \rightarrow_V e_2}{\mathtt{succ}\ e_1 \rightarrow_V succ e_2}$$

Outline

1 Introduction
- Formalities
- Overview

2 Imperative Languages

3 Object Oriented Languages

4 Functional Programming Languages
- The Scheme Language

5 Data types and Interpreters
- Interpreters: Stack machine
- Environments for an interpreter
- Cells for Variables
- Closures
- Recursive environments

6 Program Semantics
- Introduction
- Big Step Semantics
- Small Step Semantics

7 Typed Lambda Calculus
- Simply Typed Lambda Calculus
- Parametric Polymorphism - System F

Types are Ubiquitous

- Q: Write a function to print an Array of integers?

```
void printArr(int A[]){
   for (int i=0;i<A.length;++i){
       System.out.println(A[i]);
   }
}
```


What is a Type?

- A type is an invariant.
- For example, in Java

  ```
  int v;
  ```

  specifies that `v` may only contain integer values in a certain range.
- Invariant on what?
- About what?

Why Types?

Advantages with programs with types – three (tall?) claims:

- Readable : Types provide documentation;
  "Well-typed programs are more readable".

  Example: bool equal(String s1, String s2);
- Efficient: Types enable optimizations;
  "Well-typed programs are faster".

  Example: c = a + b
- Reliable: Types provide a safety guarantee;
  "Well-typed programs cannot go wrong".

Programs with no-type information can be **unreadable, inefficient, and unreliable.**

Example language

- λ-calculus.
- Admits only two kinds of data: integers and functions.
- Grammar of the language:

$$\begin{array}{lcl} e & ::= & x \mid \lambda x.e \mid e_1 e_2 \mid c \mid succ\ e \\ x & \in & \text{Identifier (infinite set of variables)} \\ c & \in & \text{Integer} \end{array}$$

Simply typed lambda calculus

- Types: integer types and function types.
- Grammar for the types:

$$\tau ::= \text{Int} | \tau_1 \to \tau_2$$

- Extend the signature of a lambda: $\lambda x : \tau.e$ – every function specifies the type of its argument.
- Examples: $\left\{ \begin{array}{lcl} 0 & : & \text{Int} \\ \lambda x : \text{Int} .(succ\ x) & : & \text{Int} \to \text{Int} \\ \lambda x : \text{Int} .\lambda y : \text{Int} .succ\ x + y & : & \text{Int} \to \text{Int} \to \text{Int} \end{array} \right.$
- These are simple types - each type can be viewed as a finite tree.
  ~~polymorphic types, dependent types~~
- Infinitely many types.

Type environment

- Type environment $A : Var \to types$.
- A type environment is a partial function which maps variables to types.
- ϕ denotes the type environment with empty domain.
- Extending an environment A with (x,t) - given by $A[x : t]$
- Application - $A(y)$ gives the type of the variable y.
- Type Evaluation: $A \vdash e : t$ — e has type t in environment A.
- Q: How to do type evaluation?

Type rules

- The judgement $A \vdash e : t$ holds, when it is derivable by a finite derivation tree using the following type rules.

$$A \vdash x : t (A(x) = t) \quad (1)$$

$$\frac{A[x:s] \vdash e : t}{A \vdash \lambda x : s.e : s \to t} \quad (2)$$

$$\frac{A \vdash e_1 : s \to t,\ A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \quad (3)$$

$$A \vdash 0 : \mathsf{Int} \quad (4)$$

$$\frac{A \vdash e : \mathsf{Int}}{A \vdash \mathsf{succ}\ e : \mathsf{Int}} \quad (5)$$

- Exactly one rule for each construct in the language. Also note the axioms
- An expression e is well typed if there exist A, t such that $A \vdash e : t$ is derivable.

Type rules

$$A \vdash x : t (A(x) = t) \quad (1)$$

$$\frac{A[x:s] \vdash e : t}{A \vdash \lambda x : s.e : s \to t} \quad (2)$$

$$\frac{A \vdash e_1 : s \to t,\ A \vdash e_2 : s}{A \vdash e_1 e_2 : t} \quad (3)$$

$$A \vdash 0 : \mathsf{Int} \quad (4)$$

$$\frac{A \vdash e : \mathsf{Int}}{A \vdash \mathsf{succ}\ e : \mathsf{Int}} \quad (5)$$

Return1 Return2 Return3

Example type derivations

- $\phi \vdash 0 : \mathsf{Int}$
- succ

$$\cfrac{\cfrac{\phi[x : \mathsf{Int}\ ] \vdash x : \mathsf{Int}}{\phi[x : \mathsf{Int}\ ] \vdash \mathsf{succ}\ x : \mathsf{Int}}}{\phi \vdash \lambda x : \mathsf{Int}\ .\mathsf{succ}\ x : \mathsf{Int}\ \to \mathsf{Int}}$$

Examples of type rules

- Identity function:

$$\frac{\phi[x : \mathsf{Int}\ ] \vdash x : \mathsf{Int}}{\phi \vdash \lambda x : \mathsf{Int}\ .x : \mathsf{Int}\ \to \mathsf{Int}}$$

- Apply

$$\cfrac{\cfrac{\cfrac{\phi[f : s \to t][x : s] \vdash f : s \to t \qquad \phi[f : s \to t][x : s] \vdash x : s}{\phi[f : s \to t][x : s] \vdash f\ x : t}}{\phi[f : s \to t] \vdash \lambda x : s.f\ x : s \to t}}{\phi \vdash \lambda f : s \to t.\lambda x : s.f\ x : (s \to t) \to (s \to t)}$$

Type derivation for SKI

- I-combinator - identity function.
- K-combinator - **K** , when applied to any argument x returns a constant function **K** x, which when applied to any argument y returns x.
  K $x\, y = x$

$$\frac{\dfrac{\phi[x:s][y:t] \vdash x:s}{\phi[x:s] \vdash \lambda y:t.x : t \rightarrow s}}{\phi \vdash \lambda x:s.\lambda y:t.x : s \rightarrow (t \rightarrow s)}$$

- S-combinator, for substitution: **S** $xyz = xz(yz)$
  Useless assignment - derive the type derivation for **S** combinator.
- SKI is turing complete. Actually, SK itself is turing complete. – Self study.

SKI combinators

- SKSK $\Rightarrow$ KK (SK) $\Rightarrow$ K
- $\mathrm{SKI(KIS)} \Rightarrow \mathrm{SKII} \Rightarrow \mathrm{KI(II)} \Rightarrow \mathrm{KII} \Rightarrow \mathrm{I}$
- $KS(I(SKSI)) \Rightarrow KS(I(KI(SI))) \Rightarrow KS(I(KII)) \Rightarrow KS(II) \Rightarrow KSI \Rightarrow S$
- $\mathrm{SKIK} \Rightarrow \mathrm{KK(IK)} \Rightarrow \mathrm{KKK} \Rightarrow \mathrm{K}$

Example underivable term

- succ $(\lambda x : t.e)$
  (Recall Rule 5):

$$\frac{A \vdash e : \text{Int}}{A \vdash \text{succ } e : \text{Int}}$$

$$\text{underivable}\ \frac{A \vdash \lambda x : t.e : \text{Int}}{A \vdash \text{succ } (\lambda x : t.e) : \text{Int}}$$

- No rule to derive the hypothesis $\phi \vdash \lambda x : t.e$.
- succ $(\lambda x : t.e)$ has no simple type.

Outline

1 Introduction
2 Imperative Languages
3 Object Oriented Languages
4 Functional Programming Languages
5 Data types and Interpreters
6 Program Semantics
7 Typed Lambda Calculus
- Simply Typed Lambda Calculus
- Parametric Polymorphism - System F

Polymorphism - motivation

- AppTwiceInt = $\lambda f : \text{Int} \to \text{Int} . \lambda x : \text{Int} . f\ (f\ x)$
  AppTwiceRcd = $\lambda f : (l : \text{Int}) \to (l : \text{Int}) . \lambda x : (l : \text{Int}) . f\ (f\ x)$
  AppTwiceOther =
  $\lambda f : (\text{Int} \to \text{Int}) \to (\text{Int} \to \text{Int}) . \lambda x : (\text{Int} \to \text{Int}) . f\ (f\ x)$
- Breaks the idea of abstraction: Each significant piece of functionality in a program should be implemented in just one place in the source code.

Polymorphism - variations

- Type systems allow single piece of code to be used with multiple types are collectively known as polymorphic systems.
- Variations:
  - Parametric polymorphism: Single piece of code to be typed generically (also known as, let polymorphism, first-class polymorphism, or ML-style polymorphic).
    - Restricts polymorphism to top-level `let` bindings.
    - Disallows functions from taking polymorphic values as arguments.
    - Uses variables in places of actual types and may instantiate with actual types if needed.
    - Example: ML, Java Generics

```
(let ((apply  lambda f. lambda a (f a)))
  (let ((a (apply succ 3)))
    (let ((b (apply zero? 3))) ...
```

  - Ad-hoc polymorphism - allows a polymorphic value to exhibit different behaviors when viewed using different types.
    - Example: function Overloading, Java `instanceof` operator.
  - subtype polymorphism: A single term may get many types using subsumption.
- Java 1.5 onwards admits Parametric, Ad-hoc and subtype polymorphism.

Parametric Polymorphism - System F

- System F discovered by Jean-Yves Girard (1972)
- Polymorphic lambda-calculus by John Reynolds (1974)
- Also called second-order lambda-calculus - allows quantification over types, along with terms.

System F

- Definition of System F - an extension of simply typed lambda calculus.

Lambda calculus recall

- Lambda abstraction is used to abstract terms out of terms.
- Application is used to supply values for the abstract types.

System F

- A mechanism for abstracting types of out terms and fill them later.
- A new form of abstraction:
  - $\lambda X.e$ – parameter is a type.
  - Application – $e[t]$
  - called type abstractions and type applications (or instantiation).

Type abstraction and application

- $$(\lambda X.e)[t_1] \rightarrow [X \rightarrow t_1]e$$

Examples

- - $$id = \lambda X.\lambda x : X.x$$

    Type of id : $\forall X.X \rightarrow X$
  - $$applyTwice = \lambda X.\lambda f : X \rightarrow X.\lambda a : X\, f\ (f\ a)$$

    Type of $applyTwice$: $\forall X.(X \rightarrow X) \rightarrow X \rightarrow X$

Extension

- Expressions:
$$e ::= \cdots |\lambda X.e|e[t]$$
- Values
$$v ::= \cdots |\lambda X.e$$
- Types
$$t ::= \cdots |\forall X.t$$
- typing context:
$$A ::= \phi|A, x : t|A, X$$

Evaluation

- $$\text{type application 1} — \frac{e_1 \rightarrow e_1'}{e_1[t_1] \rightarrow e_1'[t_1]}$$
- type appliation 2 — $(\lambda X.e_1)[t_1] \rightarrow [X \rightarrow t_1]e_1$

Typing rules

- $$\text{type abstraction} \frac{A, X \vdash e_1 : t_1}{A \vdash \lambda X.e_1 : \forall X.t_1}$$
- $$\text{type application} \frac{A \vdash e_1 : \forall X.t_1}{A \vdash e_1[t_2] : [X \rightarrow t_2]t_1}$$

Examples

- $id = \lambda X.\lambda x : X\ x$

  $id : \forall X.X \to X$

  type application: id [Int] : Int $\to$ Int

  value application: id[Int] $0 = 0$: Int
- $applyTwice = \lambda X.\lambda f : X \to X.\lambda a : Xf\ (f\ a)$

  $ApplyTwiceInts = applyTwice$ [Int]

  $applyTwice$[Int]$(\lambda x :$ Int $.succ(succx))$ 3 = 7

Polymorphic lists

List of uniform members

- `nil` : $\forall X.List\ \ X$
- `cons`: $\forall X.X \to List\ X \to List\ X$
- `isnil`: $\forall X.List\ X \to bool$
- `head`: $\forall X.List\ X \to X$
- `tail`: $\forall X.List\ X \to List\ X$

Example

- Recall: Simply typed lambda calculus - we cannot type $\lambda x.x\ x$.
- How about in System F?
- selfApp : $(\forall X.X \to X) \to (\forall X.X \to X)$

Church literals

Booleans

- `tru` = $\lambda t.\lambda f.t$
- `fls` = $\lambda t.\lambda f.f$
- Idea: A predicate will return `tru` or `fls`.
- We can write `if pred s1 else s2` as (`pred s1 s2`)

Building on booleans

- and = $\lambda b.\lambda c.b\ c$ fls
- or = ? $\lambda b.\lambda c.b$ tru c
- not = ?

Building pairs

- pair = $\lambda f.\lambda s.\lambda b.b\ f\ s$
- To build a pair: `pair v w`
- fst = $\lambda p.p$ tru
- snd = $\lambda p.p$ fls

Church numerals

- $c_0 = \lambda s.\lambda z.\ z$
- $c_1 = \lambda s.\lambda z.\ s\ z$
- $c_2 = \lambda s.\lambda z.\ s\ s\ z$
- $c_3 = \lambda s.\lambda z.\ s\ s\ s\ z$

Intuition

- Each number n is represented by a combinator c_n.
- c_n takes an argument s (for successor) and z (for zero) and apply s, n times, to z.
- c_0 and fls are exactly the same!
- This representation is similarto the unary representation we studies before.
- scc = $\lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$

Examples - derive the types

- $a = \lambda x.\lambda y.x$
- $b = \lambda f.\ (f\ 3)$
- $c = \lambda x.\ (+(head\ x)\ 3)$
- $d = \lambda f.\ ((f\ 3), (f\ \lambda y.\ y))$
- appTwice = $\lambda f.\ \lambda x.f\ f\ x$

Type inference

- Goal: Given a program with some types.
- Infer "consistent" types of all the expressions in the program.

First order unification

- Goal: To do type inference
- Given: A set of variables and literals and their possible types.
  - Remember: type = constraint.
- Target: Does the given set of constraints have a solution? And if so, what is the most general solution?
- Unification can be done in linear time: M. S. Paterson and M. N. Wegman, "Linear Unification", Journal of Computer and System Sciences, 16:158–167, 1978.
- We will instead present a simpler to understand, complex to run algorithm.

Definitions

- We will stick to simple type experssions generated from the grammar:

$$t ::= t \rightarrow t | \text{Int} | \alpha$$

  where α ranges over type variables.
- Example:

$$((\text{Int} \rightarrow \alpha) \rightarrow \beta)[\alpha := \text{Int}, \beta := (\text{Int} \rightarrow \text{Int})] = (\text{Int} \rightarrow \text{Int}) \rightarrow (\text{Int} \rightarrow \text{Int})$$

$$((\text{Int} \rightarrow \alpha) \rightarrow \gamma)[\alpha := \text{Int}, \beta := (\text{Int} \rightarrow \alpha)] = (\text{Int} \rightarrow \text{Int}) \rightarrow \gamma$$

- We say given a set of type equations, we say a substituion σ is an unifier or solution if for each of the equation of the form $s = t$, $s\sigma = t\sigma$.
- Substituions can be composed:

$$t(\sigma \; o \; \theta) = (t\sigma)\theta$$

- A substituion σ is called a most general solution of an equation set provided that for any other solution θ, there exists a substituon τ such that $\theta = \sigma \; o \; \tau$

Unification algorithm for Type inference (Hindley-Milner)

Input: G: set of type equations (derived from a given program).
Output: Unification σ

1. failure = `false`; σ = $\{\}$.
2. while $G \neq \phi$ and $\neg$ failure do
   1. Choose and remove an equation e from G. Say $e\sigma$ is $(s = t)$.
   2. If s and t are variables, or s and t are both Int then continue.
   3. If $s = s_1 \rightarrow s_2$ and $t = t_1 \rightarrow t_2$, then $G = G \cup \{s_1 = t_1, s_2 = t_2\}$.
   4. If $(s = \text{Int}$ and t is an arrow type) or vice versa then failure = `true`.
   5. If s is a variable that does not occur in t, then $\sigma = \sigma \; o \; [s := t]$.
   6. If t is a variable that does not occur in s, then $\sigma = \sigma \; o \; [t := s]$.
   7. If $s \neq t$ and either s is a variable that occurs in t or vice versa then failure = `true`.
3. end-while.
4. if (failure = true) then output "Does not type check". Else o/p σ.

Q: Composability helps?
Q: Cost?

Examples

$$\alpha = \beta \to \text{Int}$$
$$\beta = \text{Int} \to \text{Int}$$

$$\alpha = \text{Int} \to \beta$$
$$\beta = \alpha \to \text{Int}$$

Type inference algorithm (Hindley-Milner)

Input: G: set of type equations (derived from a given program).
Output: Unification σ

1. failure = `false`; σ = $\{\}$.
2. while $G \neq \phi$ and $\neg$ failure do
   1. Choose and remove an equation e from G. Say $e\sigma$ is $(s = t)$.
   2. If s and t are variables, or s and t are both Int then continue.
   3. If $s = s_1 \to s_2$ and $t = t_1 \to t_2$, then $G = G \cup \{s_1 = t_1, s_2 = t_2\}$.
   4. If $(s =$ Int and t is an arrow type) or vice versa then failure = `true`.
   5. If s is a variable that does not occur in t, then $\sigma = \sigma\ o\ [s := t]$.
   6. If t is a variable that does not occur in s, then $\sigma = \sigma\ o\ [t := s]$.
   7. If $s \neq t$ and either s is a variable that occurs in t or vice versa then failure = `true`.
3. end-while.
4. if (failure = true) then output "Does not type check". Else o/p σ.

"Occurs" check

- Ensures that we get finite types.
- If we allow recursive types - the occurs check can be omitted.
  - Say in $(s = t)$, $s = A$ and $t = A \to B$. Resulting type?
- What if we are interested in System F - what happens to the type inference? (undecidable in general)

Self study.

Outline

1. Introduction
2. Imperative Languages
3. Object Oriented Languages
4. Functional Programming Languages
5. Data types and Interpreters
6. Program Semantics
7. Typed Lambda Calculus

Ack: Slides borrowed heavily from KC@IITM.

Imperative Vs Functional Vs Logic

```
int sum(int []arr){
  int S = 0;
  for (int i=0;i<arr.length;i++){
    S += arr[i];
  }
}
```

In Scheme

```
(define (sum arr)
   (if (empty? arr) 0 (+ (car arr) (sum (cdr arr)))))
```

Logic Programming

```
sum([],0).
sum([H | T], N) :- sum(T,M), N is H+M.
```


Declarative Vs Operational

- This Prolog program says what the sum of a list is.
  - Scheme and Java programs were about how to compute the sum.
- In particular, prolog program does not define control flow through the program.
  - program is a collection of facts and rules.

Prolog Program and Answers to Questions

```
               +------------------+
Queries ==> |  Facts + Rules   | ==> Answers
               +------------------+
                  Prolog Program
```

Facts and Rules together build up a database of relations.

Relational view of the sum program

The program:

```
sum([],0).
sum([H | T], N) :- sum(T,M), N is H+M.
```

inductively defines a table (of infinite number of rows) of relations:

```
+-------------+
| List  | Sum |
|-------|-----|
| []    | 0   |
| [1]   | 1   |
| [1,2] | 3   |
| [2]   | 2   |
| ...   | ... |
```


Programs = Relations, Queries? = Lookup

Program defines a table of relations. And queries are look ups in the table!

```
?- sum([1,2,3],X).

X = 6
```


Why this declarative view?

- Many problems in computer science are naturally expressed as declarative programs.
  - Rule-based AI, Program Analysis (asking questions on code), Type Inference, queries on graphical programs, UIs.

  But the programmer has to convert this to Von Neumann Architecture (Input, CPU, Memory, Output).

Logic Program Can Save the Day

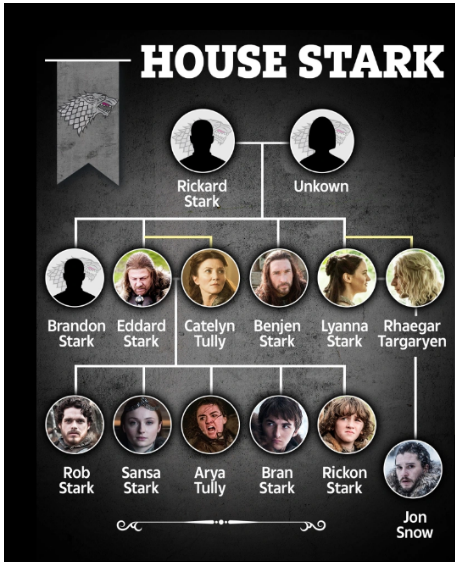
- Logic programming the programmer to declaratively express the program
- The compiler will figure out how to compute the answers to the queries.

```
Prolog = Logic (programmer) + Control (compiler)
```

Prolog

- Is one of the first logic programming languages
  - to be precise, it is a family of languages that differ by the choice of control.
- Invented in 1972, and has many different implementations
  - We will use SWI-Prolog for our study.

House of Stark

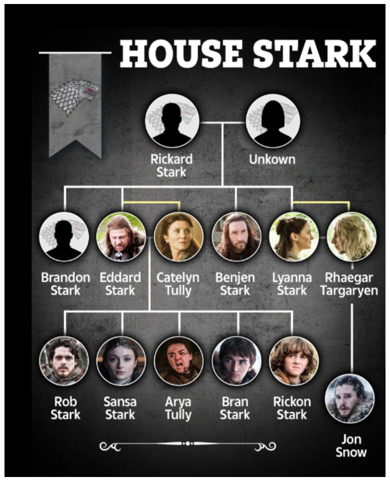


Prolog Terms

Prolog programs are made up of terms.

- Constants: 1,2,3.14,robb,'House Stark', etc.
  - also known as atoms.
- Variables: Always begin with a capital letter.
  - X, Y, Sticks, _
- compound terms: male(robb), father(ned,robb).
  - Top-function symbol/functor: male, father
  - arity: Number of arguments; male = 1, father = 2.
    - top function symbols also written down explicitly with arity such as male1, father2.

House of Stark




```
father(rickard,ned).
father(rickard,brandon).
father(rickard,lyanna).
father(ned,robb).
father(ned,sansa).
father(ned,arya).
```

Query:

```
?- father(ned,sansa).
```

Ans: True

```
?- father(rickard,sansa).
```

Ans: False

Closed World Assumption

We know that Ned is the father of Bran.
Let us query our program.

```
?- father(ned,bran).
```

```
false.
```

Closed World Assumption:

- Prolog only knows the fact that it has been told.
- Assumes false for everything else.
- Interesting interactions with negation (we will see this later).

Existential Queries

Q: “Who is the father of Arya?”

```
?- father(X,arya).
```

```
X = ned.
```

Q: “Who are Robb’s children?”

```
?- father(robb,X).
```

```
false.
```


Rules

So far what we have done could have been done with a relational database.

- Rules define further facts inductively from other facts and rules.
  - Rules have a head and body. H :- B1, B2, B3, . . . , BN
  - H is true if B1 $\wedge$ B2 $\wedge \ldots \wedge$ BN is true.

Rules (example)

```
parent(X,Y) :- father(X,Y).

ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

```
Added 3 rule(s).
```

Observe that Z only appears on the RHS of the last rule.

Rules (contd).

```
?- ancestor(rickard,X).

X = ned ;
X = brandon ;
X = lyanna ;
X = robb ;
X = sansa ;
X = arya .
```


Rules: Exercise.

Define mother, cousin, uncle, aunt, sibling.

Test

```
material(gold).
material(aluminium).
process(bauxite,alumina).
process(alumina,aluminium).
process(copper, bronze).
valuable(X) :- material(X).
valuable(X) :- process(X,Y),
               valuable(Y).
```

```
?- valuable(gold).
?- valuable(bauxite).
?- valuable(bronze).
?- valuable(copper).

   true.
   true.
   false.
   false.
```


Unification

At the core of how Prolog computes is Unification.

There are 3 rules for unification:

1. Atoms unify if they are identical
   - `a` and `a` unify, but not `a` and `b`.
2. Variables unify with anything.
3. Compound terms unfiy only if their top-function symbols and arities match and their arguments unify recursively.

Unification: Quiz

Which of these unify?

1. a & a yes
2. a & b no
3. a & A yes
4. a & B yes
5. tree(l,r) & A yes

Unification Quiz 2

Which of these unify?

1. tree(l,r) & tree(B,C) yes
2. tree(A,r) & tree(l,C) yes
3. tree(A,r) & tree(A,B) yes
4. A & a(A) yes (mostly), occurs check disabled by default
5. a & a(A) no

First Order Logic: Intro

	term	:=	constant | variable | function
Terms and Functions:	function	:=	`f(t1, t2, ..., tn)`

// `f` is function symbol, and `t1, t2, ...tn` are all terms.

Example: Natural numbers

Consider the terms for encoding natural numbers N.

- Constant: Let z be 0
- Functions: Given the natural numbers x and y, let the function
  - $s(x)$ represent the successor of x.
  - $mul(x,y)$ represent the product of x and y.
  - $square(x)$ represent the square of x.

First order Logic

$t \in$ term	:=	constant | variable | function
$f, g \in$ formulas	:=	`p(t1, t2, ..., tn)` // `p` is a predicate
		$\neg f \mid f \vee g \mid f \wedge g \mid f \rightarrow g \mid f \leftrightarrow g$
		$\forall X.f \mid \exists X.f$, where X is a variable.

Predicates on natural numbers

- even(x) - the natural number is even.
- odd(x) - the natural number is odd.
- prime(x) - the natural number is prime.
- divides (x,y) - the natural number x divides y.
- le (x, y) - the natural number x is less than or equal to y
- gt (x, y) - the natural number x is greater than y.

Precedence of the first-order logic operators

1. $\neg$
2. $\vee$
3. $\wedge$
4. $\rightarrow$, $\leftrightarrow$
5. $\forall$, $\exists$

$$(((\neg b) \wedge c) \rightarrow a)$$

can be simplified to

$$\neg b \wedge c \rightarrow a$$

Inference Rules

$$\frac{f \quad f \rightarrow g}{g} \; (\rightarrow E) \qquad \frac{\forall x.\, f(x)}{f(t)} \; (\forall E)$$

$$\frac{f(t)}{\exists x.\, f(x)} \; (\exists I) \qquad \frac{f \quad g}{f \wedge g} \; (\wedge I)$$

Interpretation

- What we have seen so far is a syntactic study of first-order logic.
  - Semantics = meaning of first-order logic formulas.
- Given an alphabet A, from which terms are drawn from and a domain D, an interpretation maps:
  - each constant $c \in A$ to an element in D.
  - each n-ary function $f \in A$ to a function $D^n \rightarrow D$.
  - each n-ary predicate $p \in A$ to a relation $D_1 \times \cdots \times D_n$.

Interpretation - example.

Let us choose the domain of natural numbers N with

- The constant z maps to 0.
- The function $s(x)$ maps the function s(x) = x + 1
- The predicate le maps to $\leq$.

Models

- A <u>model</u> for a set of first-order logic formulas is equivalent to the assignment to truth variables in predicate logic.
- A interpretation M for a set of first-order logic formulas P is a model for P iff every formula of P is true in M.
- If M is a model for f, we write $M \models f$, which is read as “models” or “satisfies”.

Models example

Take $f = \forall y.le(z,y)$. The following are models for f:

- Domain N, z maps to 0, $s(x)$ maps to $s(x) = x+1$, and le maps to $\leq$
- Domain N , z maps to 0, $s(x)$ maps to $s(x) = x+2$, and le maps to $\leq$.
- Domain N , z maps to 0, $s(x)$ maps to $s(x) = x$, and le maps to $\leq$.

whereas the following aren't:

- The integer domain Z,
- Domain N, z maps to 0, $s(x)$ maps to $s(x) = x+1$, and le maps to $\geq$

Quiz

Take $f = \forall y.le(z,y)$. Is the following a model for f?

- Domain N, z maps to 0, $s(x)$ maps to $s(x) = x+1$, and le maps to $<$

Interpretation Vs Model

An interpretation often (but not always) provides a way to determine the truth values of sentences in a language.

If a given interpretation assigns the value True to a sentence, the interpretation is called a model of that sentence.

A interpretation M for a set of first-order logic formulas P is a model for P iff every formula of P is true in M.

Models

- A set of forumulas P is said to be satisfiable if there is a model for M for P.
- Some formulas do not have models. Easiest one is $f \wedge \neg f$.
  - Such (set of) formulas are said to be unsatisfiable.

Logical consequence and validity

Given a set of formulas P, a formula f is said to be a logical consequence of P iff for every model M of P, $M \models f$.

How can you prove this?

- Show that $\neg f$ is false in every model M of P.
  - Equivalent to, $P \cup \neg f$ is **unsatisfiable**.

A formula f is said to be **valid**, if it is true in every model (written as $\models f$).

Theorem: It is undecidable whether a given first-order logic formula f is **valid**.

Restricting the Language

- Clearly, the full first-order logic is not a practical model for computation as it is undecidable.
  - How can we do better?
- Restrict the language such that the language is semi-decidable.
- A language is said to be decidable if there exists a turing machine that
  - accepts every string in L and
  - rejects every string not in L
- A language is said to be semi-decidable if there exists a turing machine that
  - accepts every string in L and
  - for every string not in L, rejects it or loops forever.

Definite Logic Programs

- Definite clauses are such a restriction on first-order logic that is semi-decidable.
- Prolog is basically programming with definite clauses.
- In order to define definite clauses formally, we need some auxiliary definitions.

Definite Clauses

- An **atomic forumla** is a formula without connectives.
  - $\text{even}(x)$ and $\text{prime}(x)$
  - but not $\neg\text{even}(x)$, $\text{even}(x) \vee \text{prime}(y)$
- A **clause** is a first-order logic formula of the form $\forall(L_1 \vee \ldots \vee L_n)$, where every L_i is an atomic formula (a postive literal) or the negation of an atomic formula (a negative literal).
- A **definite clause** is a clause with exactly one positive literal.
  - $\forall(A_0 \vee \neg A_1 \ldots \vee \neg A_n)$
  - Usually written down as, $A_0 \leftarrow A_1 \wedge \ldots \wedge A_n$, for $n \geq 0$.
  - or more simply, $A_0 \leftarrow A_1, \ldots, A_n$, for $n \geq 0$.
- A **definite program** is a finite set of definite clauses.

Definite Clauses and Prolog

- Prolog facts are definite clauses with no negative literals.
- The prolog fact even(z) is equivalent to
  - the definite clause $\forall z, even(z) \leftarrow T$,
  - where T stands for true.
- Prolog rules are definite clauses.
  - The prolog rule ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y) is equivalent to
    - the definite clause $\forall x,y,z.$ ancestor$(x,y) \leftarrow$ parent $(x,z)\wedge$ ancestor (z,y).
    - equivalent ot $\forall x,y,$ ancestor$(x,y) \leftarrow \exists z.$ parent $(x,z)\wedge$ ancestor (z,y).

Consistency of Definite Clause Programs

- Every definite clause program has a model!
- Proof
  - there is no way to encode negative information in definite clause programs.
  - Hence, there is no way to construct an inconsistent system (such as $f \wedge \neg f$).
- Therefore, every definite clause program has a model.

Prolog Query Resolution

- Let us assume that the prolog program P is the family tree of House Stark encoded before.
- We would like to answer "is Rickard the ancestor of Robb?"
  - $q =$ancestor $(rickard, robb)$.
- We construct a logical statement
  - $\neg$ ancestor $(rickard, robb)$.

  which is the negation of the original question
- The system attempts to show that $\neg$ ancestor $(rickard, robb)$ is false in every model of P.
  - equivalent to showing $P \cup \neg$ ancestor $(rickard, robb)$ is unsatisfiable.
- Then, we can conclude that for every model M of P, $M \models q$.
  - that is, "Rickard is the ancestor of Robb".

SLD Resolution

- The whole point of restricting the first-order logic language to definite clauses is to have a better decision procedue.
- There is a semi-decidable decision procedure for definite clauses called SLD resolution.
  - SLD = Selective Linear Resolution with Definite Clauses.
    - given an unsatisfiable set of formulae it is guaranteed to derive false
    - however given a satisfiable set, it may never terminate.

Example Logic program

On the board. Instructor, TAs, Students, TASibling . . .

Prolog example: Finding the last element

```
last([H],H).
last([_ | T], V) :- last(T, V).
```

Trace it in SWI Prolog.

Quick Q

```
What happens if I ask for last([],X) ?

1. pattern match exception
2. Prolog says false.
3. Prolog says true, X = []
4. Prolog says true, X = ???.
```

: Ans: 2

Prolog: is vs =

```
sum([],0).
sum([H | T], N) :- sum(T,M), N is M+H
```

```
sum([],0).
sum([H | T], N) :- sum(T,M), N = M+H
```

= used for unification. `is` used for arithmetic equality.

`is` vs =

```
A = 1+2.

A is 1+2.

A is *(3,+(1,2)).


There is support for +,*, /, <, =<, >, >= ,etc.
```


Revisiting len function

```
len([],0).
len([_ | T], N) :- len(T,M), N is M+1.
```

Trace it

```
len2([],Acc,Acc).
len2([H|T],Acc,N) :- M is Acc+1, len2(T,M,N).
```

Trace it

Last Call Optimization

- This technique is applied by the prolog interpreter
- The last clause of the rule is executed as a branch and not a call.
- We can only do this if the rule is determinate up to that point.
  - No further choices for the rule

Prolog example.

What does the following code do?

```
foo([],Q,Q).
foo([H | P], Q, [H | R]) :- foo(P, Q, R).

foo([1,2], X, [1, 2, 3, 4]).

foo([1,2], [3, 4], X).

foo(X, [3, 4], [1,2,3,4]).
```


Choice Points

- Choice points are locations in the search where we could take another option.
- If there are no choice points left then Prolog doesn't offer the user any more answers.

Quiz

What is the result of the query `len(A,2)`?

1. Error due uninstantiated arithmetic expression.
2. `[ _,_ ]`
3. Query does not terminate.
4. Error due to invalid arguments.

Ans: 2. Trace it.

Limiting the number of results

```
limit(1,len(A,2)). % Number of results limited to 1.
```


Algorithm = Logic + Control

- Logic: facts, rules and queries
- Control: how Prolog chooses the rules and goals, among several available options.
  - There are two main control decisions: Rule Order, Goal order.
- Rule Order. Given a program with a collection of facts and rules, in which order do you choose to pick rule to unify.
  - SWI-Prolog chooses the first applicable rule in the order in which they appear in the program.
- Goal order. Given a set of goals to resolve, which goal do you choose
  - SWI-Prolog: chooses the left-most sub-goal.
- Rule order and goal order influences program behaviour.

Goal order: can change the solution. Rule order: can affect the search for the solution.

Substring in Prolog

```
<-------X-------->
+--------------------------------+
|      |    S    |               |
+--------------------------------+
<--------------Z----------------->
```

We can specify this is seemingly equivalent ways.

- prefix X of Z and suffix S of X.
- suffix S of X and prefix X of Z.

Prolog Queries.

```
my_append([],Q,Q).
my_append([H | P], Q, [H | R]) :- my_append(P, Q, R).
prefix(X,Z) :- my_append(X,Y,Z).
suffix(Y,Z) :- my_append(X,Y,Z).
```


Prolog queries

```
prefix([a,b,c],[a,b,c,d]).

suffix(S,[a,b,c]).

prefix([a,b,c],[a,b,c,d]), suffix(S,[a,b,c]).

prefix(suffix(S,[a,b,c], [a,b,c],[a,b,c,d])).
```

The order of goals has no impact (here).

Goal order may alter the answer

```
prefix(X,[b]), suffix([a],X).

suffix([a],X), prefix(X,[b]).
```

Ans: first one: `false`.
Ans: second one: ?

Goal order may change the solution.

```
limit(1, (prefix(X,[b]), suffix([a],X))).

limit(2, (prefix(X,[b]), suffix([a],X))).
```

Trace.

Rule order can effect the search (for solutions)

```
appen2([H | P], Q, [H | R]) :- appen2(P, Q, R).
appen2([],Q,Q).
```

Difference in search?

```
my_append(X,[c],Z).
```

Trace.

```
appen2(X,[c],Z).
```

Trace.

Occurs Check Problem

Consider the query:

```
my_append([],E,[a,b|E]).
```

Goes for an infinite loop. Why?

- In order to unify this with, `append([],Y,Y)` , we will unify `E = [a,b | E]` ,
  - whose solution is `E = [a,b,a,b,a,b,...]`.
- In the name of efficiency, most prolog implementations do not check whether `E` appears on the RHS term.
  - infinite loop on unification.

Enable Occurs check

```
set_prolog_flag(occurs_check,true).
```

```
my_append([],E,[a,b|E]).
```

Returns false. Trace it.

```
set_prolog_flag(occurs_check,error).
```

Would throw an error.

Some Definitions

- A substitution is a finite set of pairs of terms $\{X_1/t_1, X_2/t_2, \ldots X_n/t_n\}$, where
  - each t_i is a term and
  - each X_i is a variable

  such that $X_i \neq t_i$ and $X_i \neq X_j$ if $i \neq j$.
- The empty substitution is denoted by ε.
- For example, $\sigma = \{X/[1,2], Y/Z, Z/f(a,b)\}$ is a valid substitution.

Q: What about: $\{X/Y, Y/X, Z/Z, A/a1, A/a2, m/n\}$
Y, Y, N, N, N, N.

Application of substitution

- The application of substitution σ to a variable X, written as $X\sigma$ is defined as

$$X\sigma = \begin{cases} t, & \text{if } X/t \in \sigma \\ X, & \text{otherwise.} \end{cases}$$

Let $\sigma = \{X_1/t_1, X_2/t_2, \ldots X_n/t_n\}$, E be a term or a formula. The application $E\sigma$ of σ to E is obtained by simultaneously replacing every occurrence of X_i in E with t_i.
Given $\sigma = \{X/[1,2], Y/Z, Z/f(a,b)\}$, and $E = f(X,Y,Z)$,
$E\sigma = f([1,2], Z, f(a,b))$.
Now, $E\sigma$ is known as an instance of E.

Composition of Substitution

Consider two substitutions θ and σ. Then, the composition is defined as $\theta\sigma$. Intuitively, we will apply the substitution θ before σ in $\theta\sigma$.

Consider $\theta = \{X/Y, Z/a\}$ and $\sigma = \{Y/X, Z/b\}$. Then, $\theta\sigma = \{Y/X, Z/a\}$.

Let $\theta = \{X_1/s_1, \ldots, X_n/s_n\}$ and $\sigma = \{Y_1/t_1, \ldots, Y_n/t_n\}$ be two substitutions. The composition $\theta\sigma$ is the set obtained from the set:

$$\{X_1/s_1\sigma, \ldots, X_n/s_n\sigma, Y_1/t_1, \ldots, Y_n/t_n\}$$

- by removing all $X_i/s_i\sigma$ for which X_i is syntactically equal to $s_i\sigma$ and
- by removing those Y_i/t_i for which Y_i is the same as some X_j.

Composition of Substitution

Let θ, σ and γ be substitutions, ϵ be empty substitution, and let E by a term or a formula. Then:

- $E(\theta\sigma) = (E\theta)\sigma$
- $(\theta\sigma)\gamma = \theta(\sigma\gamma)$
- $\epsilon\theta = \theta\epsilon = \theta$.
- $\theta = \theta\theta$ iff $Dom(\theta) \cap Range(\theta) = \emptyset$.

In general, composition of substitutions is not commutative. For example,

$$\{X/f(Y)\}\{Y/a\} = \{X/f(a), Y/a\} \neq \{Y/a\}\{X/f(Y)\} = \{Y/a, X/f(Y)\}$$

Composition of Substitution

Let s and t be two terms. A substitution σ is a unfier for s and t if $s\sigma$ and $t\sigma$ are syntactically equal.

Let $s = f(X,Y)$ and $t = f(g(Z),Z)$. Let $\sigma = \{X/g(Z), Y/Z\}$ and $\theta = \{X/g(a), Y/a, Z/a\}$. Both σ and θ are unfiers for s and t.

A substitution is σ is more general than another substitution θ if there exists a substitution ω such that $\theta = \sigma\omega$.

In the previous example, $\theta = \sigma\{Z/a\}$. Hence, σ is more general than θ.

Composition of Substitution

A unfier is said to be the most general unfier (mgu) of two terms if it is more general than any other unfier of the terms.

A pair of terms may have more than one most general unifier. For example, for the terms $f(X,X)$ and $f(Y,Z)$, the unifiers $\theta = \{X/Y, Z/Y\}$ and $\sigma = \{X/Z, Y/Z\}$ are both most general unifier.

$\theta = \sigma\{Z/Y\}$ and $\sigma = \theta\{Y/Z\}$.

If the unfiers θ and σ are both mgus, then there is a substitution $\gamma = \{X_1/Y_1, \ldots, X_n/Y_n\}$ where X_i and Y_i are variables such that $\theta = \sigma\gamma$.

Intuitively, θ can be obtained from σ by **renaming** the variables.

Composition of Substitution

What is the mgu of $f(X,Y,Z)$ and $f(Y,Z,a)$?

1. $\{X/a, Y/a, Z/a\}$
2. $\{X/b, Y/b, Z/b\}$
3. $\{X/Y, Z/Y\}$
4. ε

Ans: 1.

Algorithm to Compute MGU

Given two terms T_1 and T_2, output θ the mgu, if one exists, else FAIL.

Algorithm: $mgu(T_1, T_2)$.

```
Initialise
  Substitution θ = ∅,
  Stack Σ to T1 = T2,
  Failed = false.
while (Σ not empty && not Failed) {
  pop X = Y from Σ
  case
    X is a variable that does not occur in Y:
      substitute Y for X in Σ and in θ
      add X/Y to θ
    Y is a variable that does not occur in X:
      substitute X for Y in Σ and in θ
      add Y/X to θ
    X and Y are indentical constants or variables:
      continue
    X is f(X1,...,Xn) and Y is f(Y1,...,Yn):
      push Xi = Yi, i=1 to n to Σ
    otherwise:
      Failed = true
}
If Failed = true, then return FAIL else return θ
```


Trace the mgu algo

Q: mgu($f(X,Y,Z)$,$f(X,Y,Z)$)

	θ	Σ	Failed
Init.	ϕ	$[f(X,Y,Z) = f(X,Y,Z))]$	false
1.	ϕ	$[X = Y, X = Z, Y = a]$	false
2.	$\{X/Y\}$	$[Y = Z, Y = a]$	false
3.	$\{X/Z, Y/Z\}$	$[Z = a]$	false
4.	$\{X/a, Y/a, Z/a\}$	[ ]	false

Building an Abstract Interpreter to Solve Constraints

1. **Input.** A goal G, and a program P.
2. **Output.** An instance of G that is a logical consequence of P, or false otherwise.
3. Set R = G. // resolvent
4. while (R is not empty)
   1. choose a goal A from resolvent. // goal order.
   2. choose a (renamed) clause $A' \leftarrow B_1, B_2, \cdots, B_n$ from P // rule order.
      - such that $\theta = A$ and A' is the mgu.
   3. if no such goal and clause exist break;
   4. replace A by B_1, B_2, $\ldots B_n$ in R.
   5. apply $theta$ to R and G.
5. If R is empty, output G.
6. Else output false.

A few points about the algorithm

- The algorithm is non-deterministic.
- The abstract interpreter does not explicitly encode backtracking (recover from bad choices) and choice points (present more than one result).
- The program is said to be deterministic, if there is exactly one clause from the program to reduce each goal.
  - No backtracking and choice points are necessary.

Back to Prolog

Open Lists

- So far all of our uses of variables have been in queries or rules, but not in terms representing objects.
- Here is a open list which has a prefix of [1,2].

```
?- L = [1,2 | X]
L = [1, 2|X].
```

- We can (pretend to) extend the list L by unifying X with something else.

```
?- L = [1,2 | X], X = [3 | Y]
L = [1, 2, 3|Y],
X = [3|Y].
```

- Such lists are said to be <u>open</u> lists.
- The ending variable is referred to as the end marker variable of the list.
  - An empty open list consist of just an end- marker variable.
  - A list is closed if it is not open

Queues using open lists

- A queue is represented by `q(L,E)`, where
  - `L` is be an open list
  - `E` is some suffix (end-marker) of `L`.
- The contents of the queue are the elements in `L` that are not in `E`.
- We will use predicates enter and leave to capture elements entering and leaving the queue.
  - enter(a,Q,R): when an element a enters the queue Q, we get the queue R.
  - leave(a,Q,R): when an element a leaves the queue Q, we get the queue R.

Implementing Queues

```
setup(q(X,X)).
leave(A, q(X,Z), q(Y,Z)) :- X = [A | Y].
enter(A, q(X,Y), q(X,Z)) :- Y = [A | Z].
wrapup(q([],[])). % empty queue
```

```
leave(A, q(X,Z), q(Y,Z)) :- X = [A | Y].
```

while leave removes an element from the prefix.

```
enter(A, q(X,Y), q(X,Z)) :- Y = [A | Z].
```

enter removes (exposes) element from the suffix!

Queue

```
?- setup(Q), enter(a,Q,R), enter(b,R,S),
   leave(X,S,T), leave(Y,T,U), wrapup(U).

Q = q([a, b], [a, b]),
R = q([a, b], [b]),
S = q([a, b], []),
X = a,
T = q([b], []),
Y = b,
U = q([], []).
```

Q: What are the lengths of Q, R, S, T, U? 0, 1, 2, 1, 0.

Deficient Queues

Interestingly, the implementation also works where arbitrary elements are first popped and then unfied with elements pushed later.

```
?- setup(Q), leave(X,Q,R), leave(Y,R,S),
   enter(a,S,T), enter(b,T,U), wrapup(U).
Q = q([a, b], [a, b]),
X = a,
R = q([b], [a, b]),
Y = b,
S = q([], [a, b]),
T = q([], [b]),
U = q([], []).
```

What is the length of Q, R, S, T, and U? 0, -1, -2, -1, 0

Compacting the Queues

```
setup(q(X,X)).
leave(A, q(X,Z), q(Y,Z)) :- X = [A | Y].
enter(A, q(X,Y), q(X,Z)) :- Y = [A | Z].
wrapup(q([],[])). % empty queue
```

Can be compacted to:

```
setup(q(X,X)).
leave(A, q([A|Y],Z), q(Y,Z)).
enter(A, q(X,[A|Z]), q(X,Z)).
wrapup(q([],[])). % empty queue
```


Motivating Difference Lists

Recall

```
append([],Q,Q).
append([H | P], Q, [H | R]) :- append(P,Q,R).
```

If `L1=[1,2,3] and L2=[4,5,6]`,`append(L1, L2, X)?`
Q: If

```
L1 = [1,2,3 | A]
L2 = [4,5,6 | B]
```

`append(L1, L2, X)` will derive `X = [1,2,3,4,5,6|B]`.

Take from a List

`take(HasX,X,NoX)` removes exactly one element X from the list HasX with the result list being NoX.

```
take([H|T],H,T).
take([H|T],R,[H|S]) :- take(T,R,S).
```

Read the second clause as, “Given a list `[H|T]` you can take `R` from the list and leave `[H|S]` if you can take `R` from `T` and leave `S`”.

```
?- take([1,2,3],1,Y).

?- take([2,3],1,X).

?- take([1,2,3,1],X,Y).
```

Trace

```
perm([],[]).
perm(L,[H|T]) :- take(L,H,R), perm(R,T).
```


Sorted List

Check if a given list is sorted.

```
sorted([]).
sorted([H]).
sorted([A,B|T]) :- A =< B, sorted([B|T]).

?- sorted([1,2,3,4]).

true.

?- sorted([1,3,2,4]).

false.
```


Sort a list

```
permsort(L,SL) :- perm(L,SL), sorted(SL).

?- permsort([1,3,5,2,4,6], SL).
```

Too expensive to generate all permutations and search.

Quick Sort

```
part([],Y,[],[]).
part([X|Xs],Y,[X|Ls],Rs) :- X =< Y, part(Xs,Y,Ls,Rs).
part([X|Xs],Y,Ls,[X|Rs]) :- X > Y, part(Xs,Y,Ls,Rs).

?- part([6,5,3,2,1,0],4,X,Y).

Y = [ 6, 5 ], X = [ 3, 2, 1, 0 ] .

quicksort([H|T],SL) :-
  part(T,H,Ls,Rs),
  quicksort(Ls,SLs),
  quicksort(Rs,SRs),
  append(SLs,[H|SRs],SL).
quicksort([],[]).
```

Cut Operator

// From geeksforgeeks

- In Prolog, the cut (!) is a special operator with two features:
  - Always succeeds.
  - Cannot be backtracked.

```
max_element(X, Y, X) :- X > Y.
max_element(X, Y, Y) :- X =< Y.
```

Execute `max_element(5, 2, Ans).`
Does not stop after checking the first rule.

Using Cut operator

```
% If X > Y, max element is X and do not
% check the next rule for the current goal
max_element(X, Y, X) :- X > Y, !.

% If the first rule fails, then Y will
% definitely be the max element.

% Therefore, no need to put conditionals anymore
max_element(X, Y, Y).
```


With and Without cut

```
is_member1(X, [X | _]).
% If the head of the list is X

is_member1(X, [_ | Rest]) :- is_member1(X, Rest).
% else recur for the rest of the list
```

Vs

```
is_member2(X, [X | _]) :- !.
% If the head of the list is X

is_member2(X, [_ | Rest]) :-    is_member2(X, Rest).
% else recur for the rest of the list
```


One more example with Cut

Delete the first occurrence of an element from a list.

```
delete_element(_, [], []).

delete_element(X, [X | L], L) :- !.

delete_element(X, [Y | L], [Y | L1]) :-
    delete_element(X, L, L1).
```

Cut operation continued

```
eval(plus(A,B),C) :- eval(A,VA), eval(B,VB), C is VA + VB.
eval(mult(A,B),C) :- eval(A,VA), eval(B,VB), C is VA * VB.
eval(A,A).

?- eval(plus(1,mult(4,5)),X).
```

```
eval2(plus(A,B),C) :- !, eval2(A,VA), eval2(B,VB),
                         C is VA + VB.
eval2(mult(A,B),C) :- !, eval2(A,VA), eval2(B,VB),
                         C is VA * VB.
eval2(A,A).

?- eval2(plus(1,mult(4,5)),X).
```


Quiz1

```
p(a).
p(b).
r(c).
q(X) :- p(X), !.
q(X) :- r(X).
```

?- q(X).

Quiz 2

```
p := a, b.
p :- c.
```

1. 1. $p \leftrightarrow (a \wedge b) \vee c$.
2. 1. $p \leftrightarrow (a \wedge b) \wedge c$.
3. 1. $p \leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$.
4. 1. $p \leftrightarrow a \wedge (b \vee c)$.

Ans: 1

Quiz 2

```
p := a, !, b.
p :- c.
```

1. 1. $p \leftrightarrow (a \wedge b) \vee c$.
2. 1. $p \leftrightarrow (a \wedge b) \wedge c$.
3. 1. $p \leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$.
4. 1. $p \leftrightarrow a \wedge (b \vee c)$.

Ans: 3 Since the cut above changes the logical meaning of the program, it is known as Red cut.

Green Cut

```
split([],[],[]).
split([H|T],[H|L],R) :- H < 5, !, split(T,L,R).
split([H|T],L,[H|R]) :- H >= 5, split(T,L,R).
```

The cut in split does not change the logical meaning of the program.
Hence, it is called Green cut.