# CSE 3302
# Programming Languages

## Functional Programming Language
## (Introduction and Scheme)

Chengkai Li
Fall 2007



## Disclaimer

- Many of the slides are based on "Introduction to Functional Programming" by Graham Hutton, lecture notes from Oscar Nierstrasz, and lecture notes of Kenneth C. Louden.



## Resources

- Textbook: Chapter 11
- Tutorial:
  - The Scheme Programming Language http://www.scheme.com/tspl3/ (Chapter 1-2)
  - Yet Another Haskell Tutorial http://www.cs.utah.edu/~hal/htut (Chapter 1-4, 7)
- Implementation:
  - DrScheme http://www.drscheme.org/
  - Hugs http://www.haskell.org/hugs/ (download WinHugs)
- (Optional) Further reading:
  - Reference manual: Haskell 98 Report http://haskell.org/haskellwiki/Definition
  - A Gentle Introduction to Haskell 98 http://www.haskell.org/tutorial/



## History

| | |
|---|---|
| ***Lambda Calculus*** (Church, 1932-33) | formal model of computation |
| ***Lisp*** (McCarthy, 1960) ***Scheme***, 70s | symbolic computations with lists |
| ***APL*** (Iverson, 1962) | algebraic programming with arrays |
| ***ISWIM*** (Landin, 1966) | let and where clauses equational reasoning; birth of "pure" functional programming ... |
| ***ML*** (Edinburgh, 1979) ***Caml*** 1985, ***Ocaml*** | originally meta language for theorem proving |
| ***SASL, KRC, Miranda*** (Turner, 1976-85) | lazy evaluation |
| ***Haskell*** (Hudak, Wadler, et al., 1988) | "Grand Unification" of functional languages ... |



## Functional Programming

- Functional programming is a style of programming:

  *Imperative Programming:*
  - Program = Data + Algorithms

  *OO Programming:*
  - Program = Object. message (object)

  *Functional Programming:*
  - Program = Functions Functions

- Computation is done by application of functions



## Functional Programming Languages

- A functional language supports and advocates for the style of FP.
- Important Features:
- ***Everything is function*** (input->function->output)
- ***No variables or assignments*** ( only constant values, arguments, and returned values. Thus no notion of state, memory location)
- ***No loops*** (only recursive functions)
- ***No side-effect*** (Referential Transparency): the value of a function depends only on the values of its parameters. Evaluating a function with the same parameters gets the same results. There is no state. Evaluation order or execution path don't matter. (random() and getchar() are not referentially transparent.)
- **Functions are *first-class values:*** *functions are values, can be parameters and return values, can be composed.*

## We can use functional programming in imperative languages

- Imperative style

```
int sumto(int n)
{ int i, sum = 0;
  for(i = 1; i <= n; i++) sum += i;
  return sum;
}
```

- Functional style:

```
int sumto(int n)
{ if (n <= 0) return 0;
  else return sumto(n-1) + n;
}
```



## Why does it matter, anyway?

The advantages of functional programming languages:

- Simple semantics, concise, flexible
- ``No'' side effect
- Less bugs

It does have drawbacks:

- Execution efficiency
- More abstract and mathematical, thus more difficult to learn and use.

Even if we don't use FP languages:

- Features of recursion and higher-order functions have gotten into most programming languages.



## Functional Programming Languages in Use

Popular in prototyping, mathematical proof systems, AI and logic applications, research and education.

Scheme:
- Document Style Semantics and Specification Language (SGML stylesheets)
- GIMP
- Guile (GNU's official scripting language)
- Emacs

Haskell
- Linspire (commerical Debian-based Linux distribution)
- Xmonad (X Window Manager)

XSLT (Extensible Stylesheet Language Transformations)



# Scheme



## Scheme: Lisp dialect

- Syntax (slightly simplified):

*expression* → *atom* | *list*
*atom* → *number* | *string* | *identifier* | *character* | *boolean*
*list* → '(' *expression-sequence* ')'
*expression-sequence* → *expression* *expression-sequence* | *expression*

- Everything is an expression: programs, data, …

  Thus programs are executed by evaluating expressions.
- Only 2 basic kinds of expressions:
  - atoms: unstructured
  - lists: the only structure (a slight simplification).



## Expressions

| | |
|---|---|
| `42` | —a number |
| `"hello"` | —a string |
| `#T` | —the Boolean value "true" |
| `#\a` | —the character 'a' |
| `(2.1 2.2 3.1)` | —a list of numbers |
| `hello` | —a identifier |
| `(+ 2 3)` | —a list (identifier "+" and two numbers) |
| `(* (+ 2 3) (/ 6 2))` | —a list (identifier "*" and two lists) |

## Evaluation of Expressions

Programs are executed by evaluating expressions. Thus semantics are defined by evaluation rules of expressions.

Evaluation Rules:

- number | string: evaluate to itself
- Identifier: looked up in the environment, i.e., dynamically maintained symbol table
- List: recursively evaluate the elements (more details in following slides)



## Eager Evaluation

- A list is evaluated by recursively evaluating each element:
  - unspecified order
  - first element must evaluate to a function.

    This function is then applied to the evaluated values of the rest of the list. (*prefix form*).

E.g.

| | |
|---|---|
| `3 + 4 * 5` | `(+ 3 (* 4 5))` |
| `(a == b)&&(a != 0)` | `(and (= a b) (not (= a 0)))` |
| `gcd(10,35)` | `(gcd 10 35)` |

- Most expressions use applicative order evaluation (eager evaluation): subexpressions are first evaluated, then the expression is evaluated.

  (correspondingly in imperative language: arguments are evaluated at a call site before they are passed to the called function.)



## Lazy Evaluation: Special Forms

- `if` function `(if a b c)`:
  - `a` is always evaluated
  - Either `b` or `c` (but not both) is evaluated and returned as result.
  - `c` is optional. (if `a` is false and `c` is missing, the value of the expression is undefined.)

  e.g., `(if (= a 0) 0 (/ 1 a))`
- `cond`:

  `(cond (e1 v1) (e2 v2) ... (else vn))`
  - The `(ei vi)` are considered in order
  - `ei` is evaluated. If it is true, `vi` is then evaluated, and the value is the result of the `cond` expression.
  - If no `ei` is evaluated to true, `vn` is then evaluated, and the value is the result of the `cond` expression.
  - If no `ei` is evaluated to true, and `vn` is missing, the value of the expression is undefined.

  `(cond ((= a 0) 0) ((= a 1) 1) (else (/ 1 a)))`



## Lazy Evaluation: Special Forms

- `define` function:

  declare identifiers for constants and function, and thus put them into symbol table.

  `(define a b)`: define a name

  `(define (a p1 p2 …) b1 b2 …)`: define a function a with parameters `p1 p2` ….

  the first expression after define is never evaluated.

  e.g.,
  - `define x (+ 2 3)`
  - `(define (gcd u v)`
    `(if (= v 0) u (gcd v (remainder u v))))`



## Lazy Evaluation: Special Forms

- `Quote`, or `'` for short, has as its whole purpose to *not* evaluate its argument:

  `(quote (2 3 4))` or `'(2 3 4)` returns just `(2 3 4)`.

  (we need a list of numbers as a data structure)

- `eval` function: get evaluation back

  `(eval '(+ 2 3))` returns 5



## Other Special Forms

- `let` function:

  create a binding list (a list of name-value assocations), then evaluate an expression (based on the values of the names)

  `(let ((n1 e1) (n2 e2) …) v1 v2 …)`

  e.g., `(let ((a 2) (b 3)) (+ a b))`

- Is this assignment?

## Lists

List

- Only data structure
- Used to construct other data structures.
- Thus we must have functions to manipulate lists.

- `cons:` construct a list
  `(1 2 3)` = `(cons 1 (cons 2 (cons 3 '())))`
  `(1 2 3)` = `(cons 1 '(2 3))`
- `car`: the first element (head), which is an expression
  `(car '(1 2 3))` = `1`
- `cdr`: the tail, which is a list
  `(cdr '(1 2 3))` = `(2 3)`



## Data structures

```
(define L '((1 2) 3 (4 (5 6))))
(car (car L))
(cdr (car L))
(car (car (cdr (cdr L))))
```

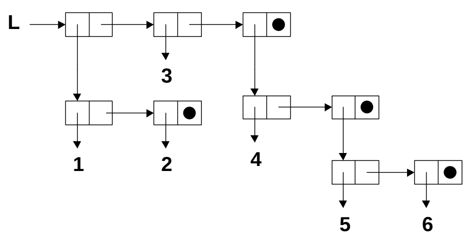
Note: `car(car = caar`
`cdr(car = cdar`
`car(car(cdr(cdr = caaddr`



## Box diagrams

a List = (head expression, tail list)

- L = ((1 2) 3 (4 (5 6))) looks as follows in memory






## Other list manipulation operations: based on car, cdr, cons

- ```
  (define (append L M)
          (if (null? L)
              M
              (cons (car L) (append (cdr L) M))
          )
  )
  ```
- ```
  (define (reverse L)
     (if (null? L)
         M
         (append (reverse (cdr L)) (list (car L)))
     )
  )
  ```



## Lambda expressions /function values

- A function can be created dynamically using a `lambda` expression, which returns a value that is a function:
  `(lambda (x) (* x x))`
- The syntax of a `lambda` expression:
  (lambda *list-of-parameters exp1 exp2 …*)
- Indeed, the "function" form of `define` is just syntactic sugar for a `lambda`:
  `(define (f x) (* x x))`
  is equivalent to:
  `(define f (lambda (x) (* x x)))`



## Function values as data

- The result of a `lambda` can be manipulated as ordinary data:

```
> ((lambda (x) (* x x)) 5)
25

> (define (add-x x) (lambda(y)(+ x y)))
> (define add-2 (add-x 2))
> (add-2 15)
17
```

## Higher-order functions

- higher-order function:
  a function that returns a function as its value
  or takes a function as a parameter
  or both
- E.g.:
  - `add-x`
  - `compose` (next slide)



## Higher-order functions

```
(define (compose f g)
  (lambda (x) (f (g x))))

(define (map f L)
  (if (null? L) L
      (cons (f (car L))(map f (cdr L)))))

(define (filter p L)
  (cond
    ((null? L) L)
    ((p (car L)) (cons (car L)
                       (filter p (cdr L))))
    (else (filter p (cdr L)))))
```



## `let` expressions as lambdas:

- A `let` expression is really just a lambda applied immediately:
  `(let ((x 2) (y 3)) (+ x y))`
  is the same as
  `((lambda (x y) (+ x y)) 2 3)`
- This is why the following `let` expression is an error if we want x = 2 throughout:
  `(let ((x 2) (y (+ x 1))) (+ x y))`
- Nested `let` (lexical scoping)
  `(let ((x 2)) (let ((y (+ x 1))) (+ x y)))`