

Organization of Programming Languages

CS320/520N

Lecture 11

Razvan C. Bunescu

School of Electrical Engineering and Computer Science

bunescu@ohio.edu

Functional vs. Imperative

- The design of the imperative languages is based directly on the **von Neumann architecture**:
 - Efficiency is the primary concern, rather than the suitability of the language for software development.
 - Heavy reliance on the underlying hardware \Rightarrow (unnecessary) restrictions on software development.
- The design of the functional languages is based on **mathematical functions**:
 - Offer a solid theoretical basis that is also closer to the user.
 - Relatively unconcerned with the architecture of the machines on which programs will run.

Mathematical Functions

- A mathematical function is a **mapping** of members of one set, called the **domain**, to another set, called the **range**:
 - The function $square: \mathbb{Z} \rightarrow \mathbb{N}$, $square(x) = x * x$
 - $square$ is the name of the function
 - x is an element in the domain \mathbb{Z}
 - $square(x)$ is the corresponding element in the range \mathbb{N}
 - $square(x) = x * x$ defines the mapping.
 - The function $fact : \mathbb{N} \rightarrow \mathbb{N}$

$$fact(x) = \begin{cases} 1 & \text{if } x = 0 \\ x * fact(x-1) & \text{if } x > 0 \end{cases}$$

Lambda Expressions

- A **lambda expression** specifies the parameters and the mapping of a nameless function in the following form:

$\lambda x. x * x$ is the lambda expression for the mathematical function
 $\text{square}(x) = x * x.$

$\lambda x. \lambda y. x + y$ corresponds to $\text{sum}(x, y) = x + y$

- Lambda expressions are applied to parameters by placing the parameters after the expression:

$(\lambda x. x * x * x) (2)$ evaluates to 8.

Functional Forms

- A higher-order function, or **functional form**, is one that:
 - either takes functions as parameters,
 - or yields a function as its result,
 - or both.
- Examples of functional forms:
 - functional composition.
 - apply-to-all.

Functional Composition

- **Mathematical Notation:**

- Form: $h \equiv f \circ g$

- Meaning: $h(x) \equiv f(g(x))$

- Example:

- $f(x) \equiv x + 2$ and $g(x) \equiv 3 * x$.

- $h \equiv f \circ g$ is equivalent with $h(x) \equiv (3 * x) + 2$

- **Lambda expression:**

- $\lambda x. x + 2$

- $\lambda x. 3 * x$

- $\lambda f. \lambda g. \lambda x. f (g x)$

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters.
- Mathematical notation:
 - Form: α
 - Function: $h(x) \equiv x * x$
 - Example: $\alpha(h, (2, 3, 4))$ yields $(4, 9, 16)$
- Lambda expression:

Functional Programming and Lambda Calculus

- Functional languages have a formal semantics derived from **Lambda Calculus**:
 - Defined by Alonzo Church in the mid 1930s as a computational theory of recursive functions.
 - The lambda calculus emphasizes expressions and functions, which naturally leads to a **functional** style of programming based on evaluation of expressions by **function application** to argument values.

Imperative Programming and Turing Machines

- **Imperative** programming: computation is performed through statements that change a program state.
- Modeled formally using **Turing Machines**:
 - Defined by Alan Turing in the mid 1930s.
 - Abstract machines that emphasize computation as a series of state transitions driven by symbols on an input tape, which leads naturally to an **imperative** style of programming based on assignment.

Functional Languages and Lambda Calculus

- Theorem (Church, Kleen, Turing):
 - Lambda Calculus and Turing Machines have the same computational power.
- Functional Languages have a denotational semantics based on lambda calculus:
 - the meaning of all syntactic programming constructs in the languages is defined in terms of mathematical functions.

Scheme

- Designed and implemented by Steele and Sussman at MIT in 1975.
- Influenced syntactically and semantically by LISP and conceptually by Algol:
 - Lisp contributed the simple syntax, uniform representation of programs as lists and garbage collected heap allocated data.
 - Algol contributed lexical (static) scoping and block structure.
 - Lisp and Algol both defined recursive functions.

Scheme: Key Features

- Scheme is **statically scoped**:
 - uses the let, let* and letrec operators to define variable bindings within local scopes.
- Scheme has **dynamic** or **latent typing**:
 - types are associated with values at run-time.
 - a variable assumes the type of the value that is bound to at run-time.
- Scheme objects are **garbage-collected**:
 - run-time objects have potentially unlimited lifetime.
- Scheme functions are **first-class objects**:
 - functions can be created dynamically, stored in data structures, returned as results of expressions or other functions.
 - functions are defined as lists \Rightarrow can be treated as data.

Scheme: Key Features

- Scheme data objects (e.g. lists) are **first-class objects**:
 - they are all heap-allocated; can be returned as results from functions, and combined to form larger data structures.
- Scheme supports many different types:
 - numbers, characters, strings, symbols, and lists.
 - integers, real, complex, and arbitrary precision rational numbers.
- Scheme includes a large set of built-in functions for manipulation of lists and other data objects.
- Arguments to functions are always **passed by value**:
 - actual arguments are always evaluated before a function is called, whether or not the function needs the values (**eager**, or **strict evaluation**).

Syntax and Naming Conventions

- Scheme programs are made of:
 - keywords, variables, structured forms (e.g. lists), numbers, characters, strings, quoted vectors, quoted lists, whitespace, and comments.
- Identifiers (keywords, variables and symbols) are formed from the characters a-z, A-Z, 0-9, and ?!.+-* /<=>:\$%^&_~
 - identifiers cannot start with 0-9,-,+.
- Predicate names end in the question mark symbol:
 - eq?, zero?, string=?
- Type predicates are the name of the type followed by a ?:
 - pair?, string?

Syntax and Naming Conventions

- Builtin character, string, and vector functions start with the name of the type:
 - string-append, ...
- Functions that convert one type of object to another use the \rightarrow symbol:
 - string \rightarrow number
- Strings are formed using double quotes:
 - “Hello, world!”
- Numbers are just numbers:
 - 100, 3.14
- Some function names are overloaded (e.g., +, *, /).

Simple Expressions

- An expression in Scheme has the form $(E_1 E_2 \dots E_n)$:
 - E_1 evaluates to an operator.
 - E_2 through E_n are evaluated as operands.
- Some examples using the Dr. Scheme interpreter:
 - $(+ 1 2 3 4) \Rightarrow 10$
 - $(+ 1 (* 2 3) 4) \Rightarrow 11$
- Scheme does **dynamic type checking** and **automatic type coercion**:
 - $(+ 2.5 10) \Rightarrow 12.5$

Simple Expressions

- Scheme uses inner-most evaluation:

- arguments are evaluated first, then substituted as parameters to functions:

```
(define (square x) (* x x))
```

```
(square (+ 2 3)) ⇒ (square 5) ⇒ (* 5 5) ⇒ 25
```

- once the subexpression `(+ 2 3)` is evaluated, the memory for this list can be garbage collected.

- Functions can also be defined using lambda expressions:

```
(define square (lambda (x) (* x x)))
```

```
(square 0.1) ⇒ 0.01
```

Top Level Bindings: DEFINE

- A Function for constructing functions DEFINE :
 1. To bind a symbol to an expression
e.g., `(DEFINE pi 3.141593)`
Example use: `(DEFINE two_pi (* 2 pi))`
 2. To bind names to lambda expressions
e.g., `(DEFINE (square x) (* x x))`
Example use: `(square 5)`
- The evaluation process for DEFINE is different! The first parameter is never evaluated. The second parameter is evaluated and bound to the first parameter.

Delayed Evaluation: QUOTE

- QUOTE takes one parameter; returns the parameter w/o evaluation.
 - `(quote (+ 1 2 3)) ⇒ (+ 1 2 3)`
- The Scheme interpreter, named EVAL, always evaluates parameters to function applications before applying the function.
- Use QUOTE to avoid parameter evaluation when it is not appropriate.
- Can be abbreviated with the apostrophe prefix operator:
 - ``(+ 1 2 3) ⇒ (+ 1 2 3)`
 - `(eval `(+ 1 2 3)) ⇒ 6`
 - `(define sum123 `(+ 1 2 3))`
 - `sum123 ⇒ (+ 1 2 3)`
 - `(eval sum123) ⇒ 6`
 - ``x ⇒ x`

Predicate Functions

- Boolean values:
 - #T is true and #F is false
 - sometimes () is used for false.
- Relational predicates:
 - =, <>, >, <, >=, <=
- Numerical predicates:
 - EVEN?, ODD?, ZERO?, NEGATIVE?

Predicate Functions: Equality

1. Use `eq?` to compare two atoms:

- `(eq? 'a 'a) ⇒ #t`
- `(eq? 1.0 1.0) ⇒ #f`

2. Use `eqv?` to compare two numbers or characters:

- `(eqv? 1.0 1.0) ⇒ #t`
- `(eqv? "hello" "hello") ⇒ #f`

3. Use `equal?` to compare two objects for structural equality:

- `(equal? "hello" "hello") ⇒ #t`

Builtin Logical Operators

- Logical operators:
 - `(and <e1> ... <en>)`
 - `(or <e1> ... <en>)`
 - `(not <e1>)`
- Parameter evaluation:
 - expressions are evaluated left to right:
 - short-circuit evaluation for `and` and `or`.
- Examples:
 - `(and (< x 10) (> x 5))`
 - `(define (<= x y) (or (< x y) (= x y)))`
 - `(define (<= x y) (not (> x y)))`

Control Flow: IF

- The special form IF:
 - (IF <predicate> <then_exp> <else_exp>)
 - (IF <predicate> <then_exp>)
- Examples:
 - (define (abs x)
 (if (< x 0)
 (- 0 x)
 x))
 - ((if #f + *) 2 3)

Control Flow: COND

- Multiple selection using the special form COND with the general form:

(COND

 (*predicate_1* *expr* {*expr*})

 (*predicate_2* *expr* {*expr*})

 ...

 (*predicate_k* *expr* {*expr*})

(ELSE *expr* {*expr*}))

- Returns the value of the last expression in the first pair whose predicate evaluates to true

Control Flow: COND

- ```
(define (abs x)
 (cond ((< x 0) (- 0 x))
 (else x)))
```
- ```
(define (compare x y)
  (cond
    (> x y) "x is greater than y")
    (< x y) "y is greater than x")
    (else "x and y are equal")))
```

Reading Assignment

- Chapter 15 from the textbook (15.1 to 15.5)
- Chapters 1 & 2 from the Scheme programming book at <http://www.scheme.com/tspl3/>
- DrScheme is installed on the prime machines (p1 & p2).
 - you can also install it on your Win/Linux/Mac machine by downloading it from www.drscheme.org.
- Familiarize yourself with the Scheme interpreter by typing in examples from the textbook or lecture notes.
 - set the language to “Standard (R5RS)”.

Factorial in Scheme

- ```
(define (factorial x)
 (if (= x 0)
 1
 (* x (factorial (- x 1)))))
```
- ```
(define factorial (lambda (x)
  (if (= x 0)
      1
      (* x (factorial (- x 1)))))
```

Lambda Expressions in Scheme

- `(lambda (<formal parameters>) <body>)`
 - When the lambda expression is evaluated, **the environment in which it is evaluated is remembered.**
 - When the procedure is called, the environment is augmented with bindings of formal params to actual params.
 - The expressions in the body are evaluated sequentially in order.
- **Example:**
 - `((lambda (x y) (* x y)) 2 3) ;; multiply 2 with 3`

Let Expressions

- Allow the definition of local variable bindings.
- General form:

```
(LET ((<name1> <expression1>)
      (<name2> <expression2>)
      ...
      (<namek> <expressionk>))
  body
)
```

- Evaluate all expressions;
- Bind the values to the names;
- Evaluate the body.

Let Expressions

- `(define pi 3.14)`
- `(define (sum-of-pi-squared) (+ (square pi)
 (square pi)))`
- `(define (sum-of-pi-squared)
 (let ((pi-squared (square pi)))
 (+ pi-squared pi-squared)))`
- Which is more efficient?

Let Expressions are Lambda Expressions

- “Syntactic sugar” for lambda expressions:

```
((lambda (<name1> ... <namek>)
  (<body>))
  <expr1>
  ...
  <exprk>)
```

- the result of the lambda expression is an anonymous procedure.
- all the argument expressions are evaluated before the procedure is called (because of call-by-value semantics).
- when the procedure is called, the variables for the formal parameters are bound to the values of the argument expressions and used in evaluating the body of the procedure.

Let* Expressions

- General form:

```
(LET* ((<name1> <expression1>)
      (<name2> <expression2>)
      ...
      (<namek> <expressionk>))
  body
)
```

- The bindings are performed sequentially, from left to right.
- \Rightarrow earlier variable bindings apply to later variable bindings.

Let* Expressions are Lambda Expressions

- Let* examples:
 - `(define x 0)`
 - `x` \Rightarrow 0
 - `(let ((x 2) (y x)) y)` \Rightarrow 0
 - `(let* ((x 2) (y x)) y)` \Rightarrow 2
- Binding order is important \Rightarrow lexically nest the lambda expressions and the application to arguments:
 - `((lambda (x) ((lambda (y) y) x)) 2)` \Rightarrow 2

Lists in Scheme

- Almost everything in Scheme is a list:
 - the interpreter evaluates most lists as an operator followed by operands, and returns a result.
 - `(+ 1 2 3 4) ⇒ 10`
 - list is evaluated as an expression, result is 10.
 - ``(+ 1 2 3 4) ⇒ (+ 1 2 3 4)`
 - result is a list of symbols
 - the empty list is denoted by `()`.
- Examples:
 - ``(colorless green ideas sleep furiously)`
 - ``((green) ideas (((sleep) furiously)) ())`

List Operations: CAR and CDR

- CAR takes a list parameter; returns the first element of that list e.g.

(CAR ' (A B C)) yields A

(CAR ' ((A B) C D)) yields (A B)

- CDR takes a list parameter; returns the list after removing its first element e.g.

(CDR ' (A B C)) yields (B C)

(CDR ' ((A B) C D)) yields (C D)

List Creation: CONS and LIST

- CONS:
 - takes two parameters:
 - the first can be either an atom or a list;
 - the second is a list;
 - returns a new list that includes the first parameter as its first element and the second parameter as the remainder.
 - `(CONS 'A ' (B C)) ⇒ (A B C)`
- LIST:
 - takes any number of parameters;
 - returns a list with the parameters as elements.
 - `(list 'a 'b 'c) ⇒ (a b c)`

Pairs

- CONS can also be used to create **pairs** or **improper lists**:

> (cons 'a 'b) \Rightarrow (a . b)

> (car '(a . b)) \Rightarrow a

> (cdr '(a . b)) \Rightarrow b

- When the second argument is a list, the result is a list:

> (cons 'a '(b)) \Rightarrow (a b)

> (car '(a b)) \Rightarrow a

> (cdr '(a b)) \Rightarrow (b)

Predicates on Lists

- LIST? takes one parameter; it returns #T if the parameter is a list; otherwise #F
 - (list? '()) ⇒ #t
 - (list? (cons 'a '())) ⇒ #t
- NULL? takes one parameter; it returns #T if the parameter is the empty list; otherwise #F
 - (null? '()) ⇒ #T
- EQUAL?
 - (equal? '(a b) (list 'a 'b)) ⇒ #T

Scheme Functions: Example

- `member` takes as parameters an atom and a simple list:
 - returns `#T` if the atom is in the list;
 - returns `#F` otherwise.

```
(DEFINE (member atom list)
  (COND
    ((NULL? list) #F)
    ((EQ? atom (CAR list)) #T)
    (ELSE (member atom (CDR list))))
  )
)
```

Scheme Functions: Example

- `equalsimp` takes two simple lists as parameters:
 - returns `#T` if the two simple lists are equal;
 - returns `#F` otherwise.

```
(DEFINE (equalsimp lis1 lis2)
  (COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((EQ? (CAR lis1) (CAR lis2))
     (equalsimp (CDR lis1) (CDR lis2)))
    (ELSE #F)
  ))
```

Scheme Functions: Example

- `equal` takes two general lists as parameters:
 - returns `#T` if the two lists are equal;
 - returns `#F` otherwise.

```
(DEFINE (equal list1 list2)
  (COND
    ((NOT (LIST? list1)) (EQ? list1 list2))
    ((NOT (LIST? list2)) #F)
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((equal (CAR list1) (CAR list2))
     (equal (CDR list1) (CDR list2)))
    (ELSE #F)))
```

Scheme Functions: Example

- `append` takes two lists as parameters:
 - returns the first parameter list with the elements of the second parameter list appended at the end.

```
(DEFINE (append list1 list2)
  (COND
    ((NULL? list1) list2)
    (ELSE (CONS (CAR list1)
                 (append (CDR list1) list2))))
)
```

Functional Forms in Scheme

- Functional Composition:

- $(\text{CDR } (\text{CDR } '(A B C))) \Rightarrow (C)$
- HW: define a function that is the composition of `cdr` with `cdr`.

- Apply-to-All:

- one form in Scheme is `map`, which applies a given function to all elements of a given list.

```
(DEFINE (map fun lis)
  (COND
    ((NULL? lis) ())
    (ELSE (CONS (fun (CAR lis))
                 (map fun (CDR lis)))))
  ))
```

Procedures That Return Procedures

```
> (define (make-adder (num))  
    (lambda (x)  
      (+ x num)))
```

```
> ((make-adder 10) 9) ⇒ ?
```

```
> ((lambda (x) (+ x 10)) 9) ⇒ ?
```

Functions that build Scheme code

- It is possible in Scheme to define a function that builds Scheme code and requests its interpretation.
- This is possible because the interpreter is a user-available function, `EVAL`.

Functions that build Scheme code

- Building a function that adds a list of numbers:

```
((DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+ lis))))
))
```

- The parameter is a list of numbers to be added;
 - `adder` inserts a `+` operator and evaluates the resulting list.
 - Use `CONS` to insert the atom `+` into the list of numbers.
 - Be sure that `+` is quoted to prevent evaluation.
 - Submit the new list to `EVAL` for evaluation.

Conceptually Infinite Lists in Scheme

- A doomed attempt to define the infinite list of integers:

```
> (define ints  
    (lambda (n)  
      (cons n (ints (+ n 1)))))
```

```
> (define integers (ints 1))
```

Conceptually Infinite Lists in Scheme

- **Delayed Evaluation:** delay the creation of remaining integers until needed.

```
> (define ints  
    (lambda (n)  
      (cons n (lambda () (ints (+ n 1))))))
```

```
> (define integers (ints 1))
```

```
> integers ⇒ (1 . #<procedure>)
```

- How do we access elements in the list?

Conceptually Infinite Lists in Scheme

- **Head** – can get the head with car:

```
> (define head car)
```

```
> (head integers) ⇒ Value: 1
```

- **Tail** – must force the evaluation of the tail:

```
> (define tail
```

```
  (lambda (list)
```

```
    ((cdr list))))
```

```
> (tail integers) ⇒ (2 . #<procedure>)
```

```
> (head (tail (tail integers))) ⇒ ?
```

Conceptually Infinite Lists in Scheme

- **Element** – get the n-th integer:

```
> (define element
  (lambda (n list)
    (if (= n 1)
        (head list)
        (element (- n 1) (tail list)))))
> (element 6 integers) ⇒ 6
> (element 6 (tail integers)) ⇒ ?
```

Conceptually Infinite Lists in Scheme

- **Take** – get the first n integers:

```
> (define take
  (lambda (n list)
    (if (= n 0)
        '()
        (cons (head list)
              (take (- n 1) (tail list))))))

> (take 5 integers) ⇒ (1 2 3 4 5)
> (take 3 (tail integers)) ⇒ ?
```

The Fibonacci Numbers

- The Fibonacci numbers as a conceptually infinite list:

```
> (define fibs
    (lambda (a b)
      (cons a (lambda () (fibs b (+ a b))))))
```

```
> (define fibonacci (fibs 1 1))
```

```
> (take 10 fibonacci)
⇒ (1 1 2 3 5 8 13 21 34 55)
```

```
> (element 10 (tail fibonacci)) ⇒ ?
```

The Sum of Two Infinite Lists

```
> (define sum
  (lambda (list1 list2)
    (cons (+ (head list1) (head list2))
          (lambda ()
             (sum (tail list1)
                  (tail list2)))))))
```

```
> (take 10 (sum integers integers))
⇒ (2 4 6 8 10 12 14 16 18 20)
```

```
> (take 5 (sum integers fibonacci))
⇒ ?
```

The Sum of Two Infinite Lists

- What does the following list correspond to?

```
> (define foo
    (cons 1
          (lambda ()
            (cons 1
                  (lambda ()
                    (sum foo (tail foo))))))))
```

```
> (take 10 foo) ⇒ ?
```

Reading Assignment

- Chapter 15 from the textbook (15.9 & 15.10)
- Chapters 4, 5 & 6 from the Scheme programming book at <http://www.scheme.com/tspl3/>
- DrScheme is installed on the prime machines (p1 & p2).
 - you can also install it on your Win/Linux/Mac machine by downloading it from www.drscheme.org.
- Familiarize yourself with the Scheme interpreter by typing in examples from the textbook or lecture notes.
 - set the language to “Standard (R5RS)”.