School of Electrical Engineering and Computing

SENG2200/6220 Programming Languages and Paradigms

Topic 9
Functional Programming
– Part 1

Dr Nan Li
Office: ES222
Phone: 4921 6503
Nan.Li@newcastle.edu.au

Topic 9 Overview

Functional Languages
- Features
- History

Scheme
- Syntax
- Lists
- Predefined Functions

Functional Languages

Procedural Languages (e.g., Pascal, C)
- Statement oriented
- Built from code segments (blocks)
- Uses variables and assignment
- Control flow (selection and iteration) is by *control structures*
- Forces sequential execution

The main concern is efficient of execution on Von Neumann architectures

Functional Languages

Procedural Example (Factorial)

```
if n == 0 then return 1
if n > 0 then
  prod = 1
  while n > 0 do
    prod = prod*n
    n = n-1
  return prod
```


Functional Languages

Functional Languages (e.g., Lisp, Scheme)
- Function oriented
- Control flow is by means of *function calls*
- Iteration is by means of *recursion*
  - there are **no looping constructs**
- Functions may be evaluated concurrently

The main concern is representation of the problem
- Tend not to be efficient on traditional architectures

Functional Languages

Functional Example (Factorial)

```
                ⎧
                | n == 0 | 1
Factl(n) ←      ⎨
                |
                | n > 0 | n * Factl(n-1)
                ⎩
```

Functional Languages

7

Imperative Languages

- Operators are applied to values and the results store in variables
- These variables can be used as a source of values at any time in their life-cycle

Functional Languages

- Functions are applied to arguments
- The result of a function may (will!) be an argument of another function
- **There is no concept of variable!**

Functional Languages

8

Recall: a pure function is one with no side-effects

- A purely functional language consists only of pure functions
- No user input/output other than as arguments to/results from the main function
- Not very useful (other than for scientific calculation)
- Most practical functional languages are not pure

If a pure function is called twice with the same arguments, the result is the same both times

- This is called *referential transparency*

Functional Languages – History

9

LISP = **LIS**t **P**rocessing language (1959)

AI research needed a language that:

- Process data in lists (rather than arrays)
- Symbolic computation (rather than numeric)

Syntax is based on *lambda calculus*

- First interpreter just to demonstrate the usefulness of lambda notation

Functional Languages – History

10

Pure LISP

- Purely functional
- *Only* supported two data types: atoms and lists

Scheme (1975)

- A “teaching” version of LISP
- Statically scoped
- First-class functions

Dynamic Scoping

11

Most programming languages you have seen use *static scoping*

- The scope (visibility) of a variable can be determined statically (at “compile” time)
- If two variables are given the same name, static scoping makes it obvious which variable is referred to from any point in the program

Dynamic Scoping

12

Early versions of LISP use *dynamic scoping*

- The scope (visibility) of a variable can only be determined at run time
- If two variables are given the same name, which variable is referred to from a particular point in the program is determined by the order of function calls leading to that point
- Consider the implications for type checking!

Dynamic Scoping


```
func A {
  int a = readint();
  if (a >= 0) B1(); else B2();
}
func B1 { float b; C(); }
func B2 { string b; C(); }
func C { write(a, b); }
```

`a` and `b` are accessible to `C` … but which `b`?

Scheme

Scheme programmes are written using *fully parenthesised prefix notation*

- A function in Scheme is a list…
- whose first element is the function name…
- and whose remaining elements are the arguments

Scheme is (usually) an interpreted language

Coding

- You could use the Racket implementation from your own computer. https://download.racket-lang.org/
- Or use online compilers, e.g., https://repl.it/repls/CoarseSaddlebrownComputergame

Scheme

Examples

```
> ( + 4 3 2 1 )
10
> ( * 4 3 2 1 )
24
> ( gcd 14 21 35 )
7
> ( quit )
```


Scheme

Example (Factorial)

```
> ( DEFINE ( factorial n )
    ( COND
      ( ( = n 0 ) 1 )
      ( ( > n 0 )
        ( * n ( factorial ( - n 1 ) ) )
      )
      ( ( < n 0 ) 'error )
  ) )
#<unspecified>
```


Scheme

Example (Factorial) (continued)

```
> ( factorial 0 )
1
> ( factorial 4 )
24
> ( factorial -1 )
error
```


Scheme Atoms

Atomic Types…

Identifiers/Symbols

- Can start with any character except `#"();0123456789`
- Can contain any characters except `()`

Boolean constants

- `#t` and `#f`

Character constants

- `#\` followed by any single character, `#\space` or `#\newline`

19

Scheme Atoms

String constants

- Any sequence of characters except `\` or `"` in `"strings"`
- Escaped characters `\\` and `\"`

Numeric constants

- A sequence of digits, possibly containing a decimal point `.` or exponent mark `e` or `E`
- Possibly preceded by `#b` (binary), `#o` (octal), `#d` (decimal), `#x` (hexadecimal), `#e` (exact) or `#i` (inexact)

20

Scheme Atoms

With the exception of strings and character constants…

Scheme is case InSeNsitive
(in Scheme standard R5RS)

But you should not rely on this!

Scheme is ALSO case sensitive
(in Scheme standard R6RS etc)

- e.g., Racket implementation

21

Scheme Lists

Atoms can be combined into lists

```
'( 1 2 3 4 5 )
'( red green blue )
'( "green" "eggs" "ham" )
'( 2112 "budd" #t )
```


22

Scheme Lists

Lists can be nested

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

The empty list `()` is a constant

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


23

Scheme Lists

By default, Scheme will attempt to evaluate any list as a function call

`(` ***func arg1 arg2*** … `)`

- Each ***arg*** may itself be a function call
- ***func*** may be a variable or function call (see the `LAMBDA` function later)
- There is no guarantee on the order of evaluation
- Arguments may even be evaluated in parallel!

24

Scheme Lists

By default, Scheme will attempt to evaluate any list as a function call

`(` ***func arg1 arg2*** … `)`

The `QUOTE` function suppresses evaluation of its *single* argument

- `( + 2 3 )` calls function `+` with argument `2` and `3`
- `( QUOTE ( + 2 3 ) )` evaluates to the list with three items –> the token `+` and the numbers `2` and `3`
- `'( + 2 3 )` is shorthand for `(QUOTE(+ 2 3))`

Scheme Lists

The **QUASIQUOTE** function acts like **QUOTE**

- Shorthand `` `( + 2 3 ) ``

But can be unquoted

`` `( 1 ( UNQUOTE ( + 2 3 ) ) ) `` evaluates to `( 1 5 )`

- Shorthand `` `( 1 ,( + 2 3 ) ) ``

`` `( 1 ( UNQUOTE-SPLICING '(+ 2 3) ) ) `` evaluates to `( 1 + 2 3 )`

- Shorthand `` `( 1 ,@'( + 2 3 ) ) ``

Scheme Lists

A list is actually a sequence of *pairs*

- The first element of a pair is the *car* – a "pointer" to the value of the element
- The second element of a pair is the *cdr* – a pointer to the next element of the list
- The cdr of the last element is the *empty list* `()`

Scheme Lists

A pair is written as two expressions separated by a dot

`( 1 . ( 2 . ( 3 . () ) ) )` ⇒

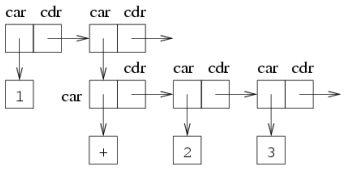
`( 1 2 3 )`

If the last element of the list is not an empty list then it is an *improper list*

`( 1 . ( 2 . ( 3 . 4 ) ) )` ⇒

`( 1 2 3 . 4 )`

Scheme Lists



Scheme Lists

`( PAIR?` ***expr*** `)`

- Is true if the expression results in a pair

`( CAR` ***pair*** `)`

- Returns the car of the pair

`( CDR` ***pair*** `)`

- Returns the cdr of the pair

It is an error to use **CAR** or **CDR** on the empty list

Scheme Lists

`( LIST?` ***expr*** `)`

- Is true if the expression results in a list

`( NULL?` ***expr*** `)`

- Is true if the expression results in the empty list

`( LENGTH` ***list*** `)`

- Returns the length of the list

Scheme Lists

(LIST-TAIL *list k*)

- Returns the result of omitting the first ***k*** elements of the list
- Same as applying cdr ***k*** times

(LIST-REF *list k*)

- Returns the ***k***th element of the list
- Indices starting from 0
- Same as **(CAR (LIST-TAIL *list k*))**

Scheme Functions

Functions in Scheme are first-class types

- Functions can be assigned to variables
- Functions can be returned by other functions

(LAMBDA (*var1* … *varn*) *body*)

- A function that expects ***n*** parameters, with parameters ***var1*** … ***varn***
- ***body*** is a sequence of one or more expressions
- Calling the function returns the value of the last expression in ***body***

Scheme Functions

Scheme checks that the number of arguments matches the number of formal parameters

(LAMBDA . *var body*)

- A function which expects zero-or-more arguments
- The list of arguments is bound to ***var***

(LAMBDA (*var1* … *varn* . *varn+1*) *body*)

- A function which expects ***n***-or-more arguments
- The first ***n*** arguments are bound to ***var1*** … ***varn***
- The list of remaining arguments is bound to ***varn+1***

Scheme Variables

(DEFINE *var expr*)

- Evaluates ***expr*** and binds the result to ***var***
- This can only be used at the top-level of code
- The scope of ***var*** is the remainder of the program
- If ***var*** already exists, then it is overwritten
- Effectively creates and intialises a global variable

(SET! *var expr*)

- Evaluates ***expr*** and stores it in an already defined ***var***

Scheme Variables

(DEFINE (*func var1* … *varn*) *body*)
⇔ (DEFINE *func* (LAMBDA (*var1* … *varn*) *body*))

(DEFINE (*func var1* … *varn* . *varn+1*) *body*)
⇔ (DEFINE *func* (LAMBDA (*var1* … *varn* . *varn+1*) *body*))

Scheme Variables

(LET ((*var1 expr1*) … (*varn exprn*)) *body*)

- Evaluates each ***expr*** and binds the result to the corresponding ***var***
- The scope of each ***var*** is the expression ***body***
- These variables may mask variables declared at higher levels
- There is no guarantee on the order in which variables are bound

Scheme Variables

(LET* ((*var1 expr1*) … (*varn exprn*)) *body*)

- Equivalent to LET, except…
- Expressions are evaluated and variables are bound in order, ***var1*** through ***varn***
- The scope of each ***var*** is the ***body*** **and** all ***vars*** and ***exprs*** after it in the order

Scheme Variables

(LETREC ((*var1 expr1*) … (*varn exprn*)) *body*)

- Equivalent to LET, except…
- The scope of each ***var*** is the ***body*** and **all** ***vars*** and ***exprs***
- Each ***expr*** is evaluated knowing that the ***vars*** exist, but not knowing their values
- Used to create mutually-recursive functions

Scheme Variables

```
> x
ERROR: unbound variable:  x
> ( DEFINE x 1 )
> ( LET ( ( y 2 ) ( z x ) )
    `( ,x ,y ,z ) )
(1 2 1)
> ( LET ( ( y 2 ) ( z y ) )
    `( ,x ,y ,z ) )
ERROR: unbound variable:  y
```


Scheme Variables

```
> ( LET* ( ( y 2 ) ( z y ) )
    `( ,x ,y ,z ) )
(1 2 2)
> ( LETREC (
    ( A ( LAMBDA (x)
      ( if (> x 0) ( B (- x 1) ) 0 ) ) )
    ( B ( LAMBDA (y)
      ( if (> y 0) ( A (- y 1) ) 1 ) ) )
  ) `( ,( A 4 ) ,( A 5 ) ,( B 4 )
      ,( B 5 ) ) )
(0 1 1 0)
```


Scheme Control Structures

(BEGIN *expr1* … *exprn*)

- Evaluates the expressions *in order* and returns the result of ***exprn***

(IF *test then else*)

- Evaluates ***test***
- If ***test*** is true, then it evaluates and returns ***then***
- Otherwise, it evaluates and returns ***else***
- Only **#f** is false, *everything* else is true

Scheme Control Structures

(NOT *expr*)

- The logical not of ***expr***

(AND *expr1* … *exprn*)

- Evaluates each ***expr*** in order until one fails, then it returns false
- If no ***exprs*** fail then it returns true

(OR *expr1* … *exprn*)

- Evaluates each ***expr*** in order until one is true, then it returns true
- If no ***exprs*** are true then it returns false

Scheme Control Structures

(COND

(test1 expr11 ... expr1j)

...

(testn exprn1 ... exprnk)

(ELSE exprn+11 ... exprn+1m)

)

- Evaluates ***tests*** in order until one is true
- Evaluates the corresponding ***exprs***, returning the last
- If no ***tests*** are true then evaluate the ***exprs*** of the (optional) else clause, returning the last

Scheme Control Structures

Example

```
(DEFINE (compare x y)
  (COND
    ( (> x y)
      (DISPLAY "x is greater than y") )
    ( (< x y)
      (DISPLAY "y is greater than x") )
    (ELSE (DISPLAY "x and y are equal") )
  )
)
```


Scheme Control Structures

(APPLY *func args*)

- Apply function ***func*** to the list of ***args*** and return the result

(MAP *func list1* … *listn*)

- Apply function ***func*** to each ***list*** and return the list of results
- Guarantees ***func*** will be called on the lists in order
- Does not guarantee the order of evaluation within each list

Scheme Control Structures

Other control structures (not covered)

FOR-EACH

- Like **MAP** – guarantees argument order but does not return results

CASE

- Like a C/Java switch

DO

- Like a C/Java for-loop – better support for multi-variables loops

Scheme Predicates

Scheme has 3 versions of equivalence…

(EQ? *obj1 obj2*)

- True if ***obj1*** and ***obj2*** are equal simple *constants* or references to the same object

(EQV? *obj1 obj2*)

- True if ***obj1*** and ***obj2*** are equal simple *values* or references to the same object

(EQUAL? *obj1 obj2*)

- *Recursively* applies **EQV?** to the elements of complex types ***obj1*** and ***obj2***

Scheme Predicates

Membership predicates

(MEMQ *obj list*)

- Returns the first sublist of ***list*** whose car **EQ?** ***obj***
- If no match is found then return **#f**
- Similarly **MEMV** using **EQV?**
- Similarly **MEMBER** using **EQUAL?**

49

Scheme Predicates

Type checking predicates…

- `( NUMBER? obj )`
- `( COMPLEX? obj ) ( REAL? obj )`
- `( RATIONAL? obj ) ( INTEGER? obj )`
- `( EXACT? num ) ( INEXACT? num )`
- `( ZERO? num )`
- `( POSITIVE? num ) ( NEGATIVE? num )`
- `( ODD? num ) ( EVEN? num )`

50

Scheme Predicates

Ordering predicates

- `( = num1 num2 num3 … )`
  - Are the arguments all (numerically) equal?
- `( < num1 num2 num3 … )`
- `( > num1 num2 num3 … )`
- `( <= num1 num2 num3 … )`
- `( >= num1 num2 num3 … )`

51

Scheme Predicates

String comparison

- `( STRING=? str1 str2 )`
  - Case sensitive
- `( STRING-CI=? str1 str2 )`
  - Case insensitive

Also `< > <= >=`

52

Scheme Predicates

Character comparison

- `CHAR=? CHAR-CI=?` and `< > <= >=`
- `(CHAR? expr)`
- `CHAR-ALPHABETIC? CHAR-NUMERIC? CHAR-WHITESPACE?`
- `CHAR-UPPER-CASE? CHAR-LOWER-CASE?`
- `CHAR->INTEGER INTEGER->CHAR`
- `CHAR-UPCASE CHAR-DOWNCASE`

53

Scheme Numbers

Useful functions on lists of numbers

- `( + num1 num2 num3 … )`
- `( * num1 num2 num3 … )`
- `( - num1 num2 num3 … )`
- `( / num1 num2 num3 … )`
  - Left associative

Also `MAX MIN GCD LCM`

Remainders, rounding, trigonometry, exponentials, rectangular- and polar-coordinates

54

Scheme Lists (Again)

- `( CONS car cdr )`
  - Creates a pair with the given car and cdr
- `( SET-CAR! pair expr )`
- `( SET-CDR! pair expr )`
  - Change the car/cdr of a pair

55

Scheme Lists (Again)

`( LIST` ***item1*** … ***itemn*** `)`

- Creates a list of the given items

`( APPEND` ***list1*** … ***listn*** `)`

- Concatenates the lists into a single list
- Newly allocates items for ***list1*** … ***listn-1*** but **not** ***listn***

`( REVERSE` ***list*** `)`

- Returns the reversed list

56

Scheme I/O

`( CURRENT-INPUT-PORT )`

`( CURRENT-OUTPUT-PORT )`

- Return the current input/output *ports*
- Default to standard input/output

`( OPEN-INPUT-FILE` ***filename*** `)`

`( OPEN-OUTPUT-FILE` ***filename*** `)`

- Return a port opened for input/output on the given file

`( CLOSE-INPUT-FILE` ***port*** `)`

`( CLOSE-OUTPUT-FILE` ***port*** `)`

- Close an open input/output port

57

Scheme I/O

`( READ` ***port*** `)`

- Returns the object whose external representation is found on ***port***

`( READ-CHAR` ***port*** `)`

- Removes the next character on ***port*** and returns it

`( PEEK-CHAR` ***port*** `)`

- Returns the next character on ***port*** without removing it

`( EOF-OBJECT?` ***obj*** `)`

- Return true if ***obj*** is the end-of-file object

port defaults to `current-input-port`

58

Scheme I/O

`( WRITE` ***obj port*** `)`

- Writes a machine representation of ***obj*** to ***port***
- It appears as it would if written in Scheme source code

`( DISPLAY` ***obj port*** `)`

- Writes a machine representation of ***obj*** to ***port***
- Strings have no double-quotes or escaped characters, and characters appears as per `WRITE-CHAR`

`( WRITE-CHAR` ***char port*** `)`

- Writes a single character to ***port***

`( NEWLINE` ***port*** `)`

- Writes a newline character to ***port***

port defaults to `current-output-port`

59

Scheme I/O

Functions for converting between strings and numbers

`( NUMBER->STRING` ***number*** `)`

`( NUMBER->STRING` ***number radix*** `)`

`( STRING->NUMBER` ***string*** `)`

`( STRING->NUMBER` ***string radix*** `)`

60

References

R. W. Sebesta, “Concepts of Programming Languages”, 9th Edn, Addison-Wesley, 2010 (Chapter 15) (also Edn.10)

R. K. Dybvig, “The Scheme Programming Language”, 3rd Edition, MIT Press, 2003. http://www.scheme.com/tspl3/