School of Electrical Engineering and Computing

SEENG2200/6220 PROGRAMMING LANGUAGES & PARADIGMS

(S1, 2020)

Functional Programming I

Dr Nan Li
Office: ES222
Nan.Li@newcastle.edu.au

THE UNIVERSITY OF NEWCASTLE AUSTRALIA

Outline

- 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

- 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!***

Pure Function

- 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*

Brief History

- 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*

Brief History

- Pure LISP
  - *Purely functional*
  - *Only supported two data types: atoms and lists*
- Scheme (1975)
  - *A "teaching" version of LISP*
  - *Statically scoped*
  - *First-class functions*

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*
  - *E.g., (**+** 1 2) and (**max** 5 8)*
- 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

Example

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

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.*
  - *E.g., #\a, #\b, #\?*
  - **`#\space`** *and* **`#\newline`**

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)*

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*

Scheme Lists

- Atoms can be combined into a **list**

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

- 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 )
```

Scheme Lists

- To create a list:
  - *Use single-quote (indeed* **`quote`** *function):* `'(1 2)`
  - *Use list function:* `(`**`list`**` 1 2)`
- 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!*

Scheme Lists

- The **`quote`** function suppresses evaluation of its *single* argument, for example

```
> ( + 2 3 )
    5
> ( quote (+ 2 3) )
    (+ 2 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 ) ) )
  (1 5)
```

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

```
> `( 1 ( unquote-splicing '(+ 2 3) ) )
  (1 + 2 3)
```

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

Scheme Pairs

- A pair is written as two expressions separated by a dot
  - *E.g.,* `(1 . 2)`
- A pair can be created by using *cons* function

```
> (cons 1 2)
  (1 . 2)
```

- The first element of a pair is the **car**
  - E.g., `(car (cons 1 2) )= 1`
- The second element of a pair is the **cdr**
  - E.g., `(cdr (cons 1 2) )= 2`

Pairs and Lists

- A list is actually a sequence of *pairs*
- A list can be written/represented by using nested pairs
  - `(1 2 3)` *can be considered as*

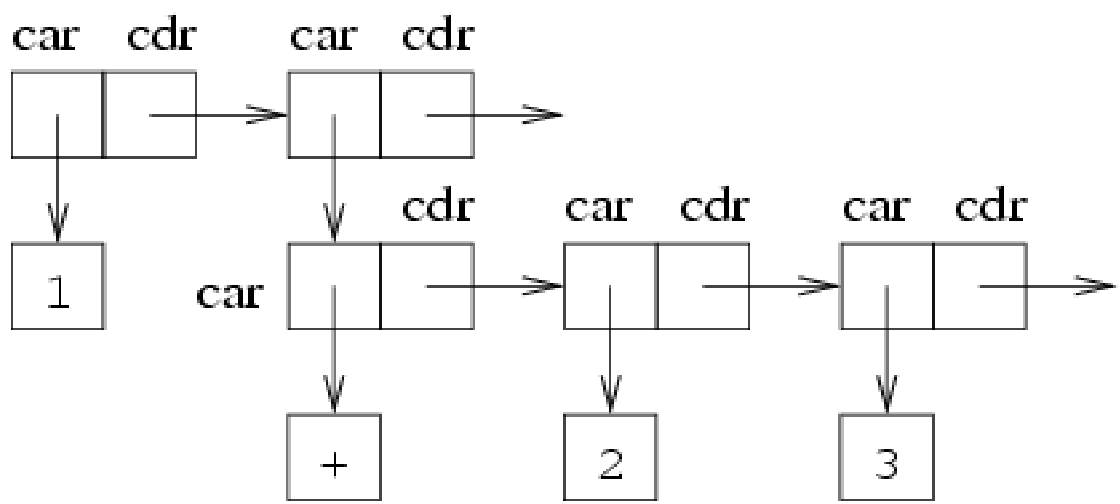
    `( 1 . ( 2 . ( 3 . () ) ) )`
  - *The **cdr** of the last element is the empty list ( )*
  - *If the last element of the list is not an empty list then it is an improper list, e.g.,*

    `( 1 . ( 2 . ( 3 . 4 ) ) )`

    $\Rightarrow$ `( 1 2 3 . 4 )`

Pairs and Lists

- Considering the pairs of a list
  - `car` is a “pointer” to the value of the element
  - `cdr` is a “pointer” to the next element of the list
- For example, the list `(1 + 2 3)` looks like



Scheme Variables and Functions

- `( 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 and 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 Variables and 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 and Functions

(**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*)

)

Example

- Swap function

```
( define ( swap a b )
   ( define temp a)
   ( set! a b )
   ( set! b temp )
   (list a b)
 )

> ( swap 5 6 )
   (6 5)
```

The `let` Function

- `( 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*

The `let*` Function

- (**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*

The `letter` Function

- `( 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*

Examples

```
> 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
```

Examples

```
> ( 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)
```

Some Functions

- (**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

Some Functions

- (**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*

Some Functions

- (**list** item1 … itemn)
  - *Creates a list of the given items*
- (**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*

Some Functions

- (**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*
- (**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-tall** list k))*

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

Example

- Factorial function

```
( define ( factorial n )
    ( cond
      ( ( = n 0 ) 1 )
      ( ( > n 0 )
        ( * n ( factorial ( - n 1 ) ) ) )
      )
      ( ( < n 0 ) 'error )
    )
 )
```

Example

- Output

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

Example

- Factorial function

```
(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 <u>equivalence</u>…
- (**`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***

Examples

```
> ( equal? 1 2 )
  #f
> ( equal? (list 1) (list 2) )
  #f
> ( equal? (list 1) (list 1) )
  #t
> ( eqv? (list 1) (list 1) )
  #f
> ( eq? (list 1) (list 1))
  #f
> ( eq? '() '())
  #t
```

Scheme Predicates

- Scheme has 3 versions of <u>membership</u> 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?**

Examples

```
> ( memq 1 '(1) )
  (1)
> ( memq 1 '(2 1 4) )
  (1 4)
> ( memv '(1) '(2 1 4) )
  #f
> ( memv '(1) '(2 (1) 4) )
  #f
> ( member '(1) '(2 1 4) )
  #f
> ( member '(1) '(2 (1) 4) )
  ((1) 4)
> ( memv 1 '(2 (1) 4) )
  #f
```

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 )
```

Scheme Predicates

- Ordering predicates

```
( = num1 num2 num3 … )
( < num1 num2 num3 … )
( > num1 num2 num3 … )
( <= num1 num2 num3 … )
( >= num1 num2 num3 … )
```

Scheme Predicates

- String comparison
  - *Case sensitive*

  (**string=?** str1 str2)

  - *Case insensitive*

  (**string-ci=?** str1 str2)

- Also **< > <= >=**

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

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*

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`**

Scheme I/O

- (**`write`** `obj port`)
  - *Writes a machine representation of **obj** to **port***
- (**`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`**

Scheme I/O

- Functions for converting between strings and numbers

```
( number->string number )
( number->string number radix )
( string->number string )
( string->number string radix )
```

  - Radix must be an exact integer. By default, radix=10. E.g.,

```
>( number->string 5)
  "5"
>( number->string 5 2)
  "101"
```

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/