Scheme

Wisam Hussain
hussaiwh@mcmaster.ca
Department of Computing and Software (McMaster University)
Content

- Introduction
- Data Types
- Expressions
- Sequencing
- Procedures
- Scope Rules
- Conditionals
- Recursion
- Input / Output
- Libraries
- References
Introduction

• Scheme is a general purpose multi-paradigm programming language with support for functional, procedural and meta programming styles.

• Developed by Guy L. Steele and Gerald Jay Sussman at MIT AI lab in the 1975 and later introduced to the academic community through a series of memos known as the lambda papers.

• Standardized by the IEEE and RnRS (Revised Report on the Algorithmic Language Scheme), currently R5RS and R6RS are the most widely implemented standards
  • This presentation covers the R6RS implementation of the language

• Scheme is one of the 2 main dialects of LISP (LISt Processing) programming language
  • Scheme syntax and semantics is heavily influenced by LISP
Introduction (Characteristics)

- **Syntax**
  - Parenthesized lists in which a prefix operator is followed by its arguments (S-Expressions)

- **Typing System:**
  - Strong Dynamic typing system

- **Scope**
  - Lexical (unlike LISP which is dynamically scoped, scheme borrowed the idea of lexical scoping from ALGOL)

- **Evaluation Strategies**
  - Call by value and Call by object
  - Lazy evaluation is also available through the use of the `delay` form

- **Philosophy**
  - Minimalist with small standard core and powerful tools to extend the language
Data Types (Simple)

- Simple Data Types
- Booleans
  - True is represented by \#t
  - False is represented by \#f
- Numbers
  - Integer Numbers (e.g. 12, \#d12, \#b1100, \#o14, \#xc )
    - The \#d prefix is optional when representing integers in decimal
  - Rational Numbers (e.g. 22/7)
  - Real Numbers (e.g. 3.1416)
  - Complex Numbers (e.g. 2+3i)
  - Note: Every Integer is Rational, Every Rational is Real, Every Real is Complex and Every Complex is a Number
- Characters
  - Graphical Characters (e.g. \#\a, \#\b, \#\c)
  - Non Graphical Characters (e.g. \#newline, \#tab, \#space)
- Symbols
  - Used in scheme as an identifiers for variables
  - To use a symbol without making Scheme think it is a variable, you need to quote the symbol (e.g. 'xyz or (quote xyz)).
  - Using xyz without the quote will return the value associated with 'xyz_identifier
Data Types (Compound)

• Compound Data Types
• Strings (e.g. "Hello, World!", "Hello, World!", "123")
• Vectors (e.g. '#(0 1 #(3 4) 5 6), #(0 "Zero" #\0)
  
  • Vectors are sequences like strings but their elements can be any thing not just characters (mixed types are allowed)
• Dotted Pairs and Lists
  
  • Dotted pairs are compound values made by combining 2 values in an order couple. (e.g. '(1 . #t), '((1 . 2) . 3))
  
  • Lists are a special case of Dotted pairs where the nested dotting occurs along the second element (e.g. '(1 . (2 . (3 . (4 . ()))))) '(1 2 3 4))
  
  • Procedure (e.g. display, max, min)
Data Types (Dotted Pairs and Lists)

- Some procedures on Dotted Pair and Lists
  - Lists are a special form of dotted pairs
  - **Car** procedure: return the first element of the list
  - **Cdr** procedure: return the second element of the list (tricky!)
  - **Cons** procedure: combines 2 values into an ordered pair

```scheme
(define x (cons 1 #t))
x => (1 . #t)
(car x) => 1
(cdr x) => #t

(define y (cons (cons 1 2) 3))
y => ((1 . 2) . 3)
(car (car y)) => 1
(cdr (car y)) => 2

(caar y) => 1 ; abbreviation of (car (car y))
(cdcar y) => 2 ; abbreviation of (cdr (car y))
```
Data Types (Conversion)

- Since scheme has a Strong Dynamic typing system, we need
  - A way to determine variable types
  - A way to convert from one type to another
- Scheme provides a wide range of procedure to achieve that
- Type checking Examples:
  - (boolean? #t)  =>  #t
    (complex? 2+3i)  =>  #t
    (integer? 42)   =>  #t
    (symbol? 'xyz)  =>  #t
    (list? '(1 3))  =>  #t
  - Note that the ? Character is part of the procedure name
- Type conversion Examples:
  - (number->string 16)  =>  "16"
    (string->number "16")  =>  16
    (char->integer #\d)    =>  100
    (integer->char 100)    =>  #\d
  - Note that the -> symbols are part of the procedure name and that they are not pointers
Naming Conventions

• Procedure naming convention
• The name of procedures that always return a boolean value usually ends with ?
  • Examples (boolean?, integer?, list?, empty?)
• The name of procedures that always stores values in previously allocated locations usually ends with !
  • Examples (set!, vector-set!, string-set!)
• The name of procedures that convert an object from one type to another usually contains ->
  • Examples (integer->string, integer->complex)
• Identifiers can contain letters, digits and (! $ % & * + - . / : <= > ? @ ^ ~)
  • Identifiers can not start with a digit
  • Identifiers are case insensitive (Foo is the same as foo)
• The ; keyword is used to create comments
  • (Example: ;this is a comment)
  • Only single line comments are supported
Expressions

• Expressions are the main building block in scheme
• Expressions can be evaluated, producing a value
• Expression in scheme can be
  • Literal Expressions
    • \#t => \#t
      23 => 23
  • Compound Expressions
    • Have the following format
      • (Operator Operand-1 ... Operand-N)
        • where operands can be simple or compound expressions
      • (+ 23 42) => 65
      • (+ 14 (* 23 42)) => 980
  • Note that the parenthesis are not optional
Sequencing

- We use the begin form to bunch together a group of sub forms that needs to be evaluated in a sequence

```
(begin
    (display "Hello")
    (display " ")
    (display "World")
    (display " ")
    (display "!")
    (newline))
```

Hello World !
Procedures

• User defined procedures can be created using the special form `lambda`
• The following example defines a procedure that adds 2 to a number
• \( (\text{lambda } (x) (\text{+ } x \text{ 2})) \)
• To apply this function to an argument

\[
((\lambda x (\text{+ } x 2)) 5)
\]

7
Procedures

• To reuses the same procedure in our code, we can use a variable to hold the procedure value

\[
\text{(define add2}
\text{ (lambda (x) (+ x 2)))}
\]

(\text{add2 4})

6
Procedures

- Procedures can have multiple arguments
- Procedure arguments are local to the body of the procedure

```
(define area
  (lambda (length breadth)
    (* length breadth)))

(area 5 10)

50
```
Procedures

• Procedures can have variable number of arguments
• To achieve that replace the parameters list by a single symbol that will bind to a list of arguments

```lisp
(define sum1
  (lambda args
    (apply + args)))
```

```lisp
(sum1 5)
```

```
5
```

```lisp
(sum1 5 10 15)
```

```
30
```
Scope Rules

• Scheme variables have lexical scope
  • Global Variables have the program text as their scope
  • Local variables
    • Lambda parameters have the lambda body as their scope

```
(define x 9)
(define add2 (lambda (x) (+ x 2)))

x
9
(add2 3)
5
(add2 x)
11
```
Scope Rules

- The form `set!` modifies the lexical binding of a variable.
  
  ```lisp
  (set! x 20)
  ```

- The above modifies the global binding of `x` from 9 to 20, because that is the binding of `x` that is visible to `set!`.

- If the `set!` was inside `add2`'s body, it would have modified the local `x`

  ```lisp
  (define add2
    (lambda (x)
      (set! x (+ x 2))
      x))

  (add2 x)
  ```

  22
  
  x
  
  20
Scope Rules

• Local variables can be created without creating a procedure using the special form let.

• **Let** introduces a list of local variables that have the body of let as its lexical scope.

```
(let ((x 1)
      (y 2)
      (z 3))
  (list x y z))
```

```
(1 2 3)
```

```
(define x 20)
(let ((x 1)
      (y x))
  (+ x y))
```

```
21
```
Scope Rules

Sometimes, it is convenient to have let's list of lexical variables be introduced in sequence, so that the initialization of a later variable occurs in the *lexical scope* of earlier variables.

```
(define x 20)
(let ((x 1)
      (y x))
  (+ x y))
```

```
(define x 20)
(let* ((x 1)
       (y x))
  (+ x y))
```
Conditional

• If statement
  • If the test condition evaluates to \texttt{#t} (any value other than \texttt{#f}) then the “then” branch is evaluated otherwise the else branch is evaluated.
  • The else branch is optional in Scheme

```
(define pressure 80)
(if (> pressure 70)
    'safe
    'unsafe)

safe
```

```
(define pressure 80)
(if (> pressure 70)
    'safe)

safe
```
Conditionals

• Cond statement
  • The cond form is convenient for expressing nested if-expressions.

\[
\text{cond } ((\text{char}<? \ c \ #\c) \ -1) \\
\quad ((\text{char}=? \ c \ #\c) \ 0) \\
\quad (\text{else} \ 1))
\]

• Can be written as

\[
\text{cond } ((\text{char}<? \ c \ #\c) \ -1) \\
\quad ((\text{char}=? \ c \ #\c) \ 0) \\
\quad (\text{else} \ 1))
\]

• Begin is added implicitly to the condition actions
Conditionals

• Case statement
  • Case is a special form of cond

(case c
  ((#\a) 1)
  ((#\b) 2)
  ((#\c) 3)
  (else 4))

Recursion

• A procedure body can contain calls to other procedure including itself.

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

• Mutual recursion is also possible in Scheme

```
(define is-even?
  (lambda (n)
    (if (= n 0) #t
      (is-odd? (- n 1)))))

(define is-odd?
  (lambda (n)
    (if (= n 0) #f
      (is-even? (- n 1)))))
```
Recursion

- If you want to use the is-even? And is-odd? procedures as local variable use the letrec keyword

```scheme
(letrec ((local-even? (lambda (n)
                           (if (= n 0) #t
                               (local-odd? (- n 1)))))
        (local-odd? (lambda (n)
                        (if (= n 0) #f
                            (local-eve? (- n 1))))))
  (display (list (local-even? 23) (local-odd? 23))))

(#f #t)
```

- Note: Looping is achieved in Scheme using recursion
Input / Output

• Scheme has input / output procedures that will let you read from an input port or write to an output port
  • If no port is specified, Scheme uses the console for input and output.
  • We can read one character at a time, one line at a time or one expression at a time using the read-char, read-line, and read procedures respectively.
  • Assume we have a text file called hello.txt and it contains the “hello” string

```
(define i (open-input-file "hello.txt"))
(read-char i)
(close-input-port i)
```

#\h
Input / Output

• Writing can be done 1 character at a time or 1 expression at a time using write-char and write respectively.
  • display procedure can be used instead of the write to output in a non machine readable format
  • (write “CAS 706”) will write “CAS 706” on the console with quotation
  • (display “CAS 706”) will display CAS 706 on the console without quotation

(define o (open-output-file "greeting.txt"))
(display "hello" o)
(write-char #:\space o)
(display 'world o)
(newline o)
(close-output-port o)

hello world
Libraries

• Scheme code can be organized into libraries
  • Libraries can import other libraries
  • Libraries can import all or some of their content

```scheme
(library (hello)
 (export hello-world)
 (import (rnrs base)
 (rnrs io simple))
 (define (hello-world)
 (display "Hello World")
 (newline))

(import (hello))
```

• To import a library use the import procedure. (example below)
References

Thank You