Intro. to Scheme Programming

The fundamentals of Scheme

* symbolic programming using list expressions and recursive functions
* in a nutshell
  + everything data wise is stored in a list (recursive structure)
  + all functions use recursion to process the data, and send an adjusted list to the same function over again
* Scoping static
* Typing: dynamic (what does this mean?)
* No distinction between code and data
  + **Both functions and data are represented as lists**
    - Syntax looks like a list
  + Lists are first-class objects
    - Can be created dynamically, passed as arguments to functions, returned as results of functions and expressions
  + This requires heap allocation (why?) and garbage collection (why?)
  + Self-evolving programs

LISP vs. Scheme

* LISP is an acronym for LISt Processing language
* [Lisp](http://en.wikipedia.org/wiki/Lisp_(programming_language)) (b. 1958) is an old language with many variants
  + Fortran is only older language still in wide use
  + Lisp is alive and well today
* Most modern versions are based on Common Lisp
* Scheme is one of the major variants
  + We’ll use Scheme, *not* Lisp, in this class

LISP itself

* It’s a simple, elegant yet powerful language
* You will learn a lot about PLs from studying it
* We’ll look at how to implement a Scheme interpreter in Scheme and Python
* Many features, once unique to Lisp, are now in “mainstream” PLs: python, javascript, perl …
* It will expand your notion of what a PL can be

Why Scheme?

* clear syntax
* smallness
* Scheme is a dialect of Lisp that is favored by people who teach and study programming languages
* Why?
  + It’s simpler and more elegant than Lisp
  + It’s pioneered many new programming language ideas (e.g., continuations, call/cc)
  + It’s influenced Lisp (e.g., lexical scoping of variables)
  + It’s still evolving, so it’s a good vehicle for new ideas
* Personal Computer Scheme Programs
  + Racket/DrScheme
    - is a graphical programming environment (IDE) for Scheme
* on compute (TAMU)
  + compute.cse.tamu.edu
  + Gambit (GSI)
* on GL (UMBC)
  + MzScheme
    - is the basic scheme engine and can be called from the command line and assumes a terminal style interface
  + type mzscheme on GL to start Scheme and enter its interpreter
  + much like Python!!

Scheme the language

* is a function language
  + it is oriented around the evaluation of functions (return value)
* other types of languages
  + procedural
    - does not worry about returning a value
  + object oriented
    - oriented around an instance/object

|  |  |  |
| --- | --- | --- |
| **Comparison of Syntax Between Paradigms** | | |
| C/Cobol (procedural) | C++ (object oriented) | Scheme (functional) |
| vectorInsert(&vector);  vectorAppend(&vector);  vectorDelete(&vector); | vector <int> v;  …  v.pushback();  v.append(newElement); | what we are about to do! |

* Scheme “evaluates” the code given to it
  + some answer has to be produced!!
  + a lot of the “answer” is synthesized!!!
    - means no data is used up for the answer
      * some may have been temporarily used to evaluate
      * but data is cleaned up -> garbage collection!!
  + ***only when it evaluates, does it check data type compatibility!!***

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Scheme evaluation (using the interpreter mode)** | | | | | |
| > 4  4 | > “Lupoli”  Lupoli | > 5/3  5/3 | > 21/7  3 | > 3.33423  3.33423 | > #t  #t |

|  |  |
| --- | --- |
| **Functional Language Scheme** | |
| functional mathematics | functional Scheme |
| f(x, y) = (x2 + 3) \* y  f(6, 7) => (62 + 3) \* 7  f(6, 7) => (36 + 3) \* 7  f(6, 7) => 39 \* 7  f(6, 7) => 273 | > (+ 7 3)  10  solution synthesized,  memory used returned to heap |

Syntax of Scheme (and why)

* uses a list to parse tokens in the code
  + ( [0] [1] [2] [3] …)
  + starts and ends with ( )
  + can have nested lists
  + only [0] is an operator or function
* ( ) are the killers

|  |
| --- |
| **Evaluation of numeric functions** |
| ( + 1 2 3) => 6  ( \* ( + 4 4) ( + 5 5)) => 80  ( + 32 ( \* 1.8 temp)) |

REPL

* Lisp and Scheme are interactive and use what is known as the “read, eval, print loop”
  + **While true**
    - **Read** one expression from the open input
    - **Evaluate** the expression
    - **Print** its returned value
* Lisp’s interpreter essentially does:
  1. Read an expression
  2. Evaluate it
  3. Print the resulting value
  4. Goto 1
* Understanding the rules for evaluating an expression is key to understanding lisp.

|  |
| --- |
| **Lisp’s/Scheme’s Interpreter interaction** |
|  |

Basics when using a Unix OS

* use <CTRL><Z> or type (exit) to break out of the Scheme interpreter
* every “program/script” evaluates to an answer of some sort
  + E => v
* input
  + use the (read) command in the interpreter
* Linux interpreter
  + create .scm file with code first
  + two versions

|  |  |
| --- | --- |
| **Options for interacting with the Interpreter** | |
| Interpretive Mode | Batch mode |
| mzscheme | mzscheme –r helloWorld.scm |
| in interpreter  have to “load” file (hello…)  when completed, still in interpreter, so if you need to exit you have to <CTRL><Z> or type (exit) | runs, loads, and exits!!! |
| gsi | gsi helloWorld.scm |
| in interpreter have to “load” file (hello…) when completed, still in interpreter. To exit, type (exit)  ; add code below  (load “extra.scm”) | same as above  watch video [here](https://youtu.be/lszu6ZXAQr4) |

Learning the setup of a program

* the setup is the same for many of your programs
* the setup must be in the SAME order as shown as below
* please notice that the ( ) are necessary

|  |  |
| --- | --- |
| **Setup Example** | |
| ; File: helloWorld.scm  ; Written by: Prof. Lupoli  ; Date: 11/9/10  ; TAMU email: slupoli@cse.tamu.edu  ; Class: CSCE 314  ; Description: This is the first program in Scheme  (display "Hello World\n")  (display "This is Lupoli's first Scheme Program!!\n")  (exit) | Place name here  Place what the program is going to be done  \*\*Where most of your code goes!!\*\*  end with exit |

1. If you are unfamiliar with any Unix Text editor, pair up with someone who does
2. Create the file above in Unix, and get it running.
3. After getting it to run, make 3 individual, intentional mistakes and note the error messages you receive.
4. Add the lines
   1. (< 1 3)
   2. (

API For Scheme

* thankfully there is an API for Scheme
* <http://download.plt-scheme.org/doc/html/index.html>
* can be cumbersome, but useful!!

Data types

|  |  |
| --- | --- |
| **Type** | **Examples/Definitions** |
| **numbers (int)** | **(define score 42)** |
|  | *a WHOLE number, ranging from -2147483647 to 2147483647* |
| **numbers (complex)** | **(define value 2+3i) (define check )** |
|  | *a WHOLE number, ranging from -32767 to 32767* |
| **numbers (real)** | **(define pie 3.1416)**  **(define fraction 22/7)**  **(define score 42)** |
|  | *a WHOLE number, ranging from -2^63 to 2^63-1* |
| **numbers (float)** | **float GPA = 3.99** |
|  | *a real number in a floating po(define representation* |
| **numbers (rational)** | **(define pie 3.1416)**  **(define fraction 22/7)** |
|  | *a double-precision real number in a floating po(define representation* |
| **numbers (binary)** | **(define binaryValue #b1100)** |
|  |  |
| **numbers (octal)** | **(define octalValue #o1100)** |
|  |  |
| **numbers (hex)** | **(define hexValue #xFFF)** |
|  |  |
| **char** | **(define choice 'x')** |
|  | *ONE character, or a small INTEGER in the range from -128 to 127* |
| **Boolean** | **(define found #f)** |
|  | *a Boolean value can either AND ONLY be true(#t) or false(#f)* |
| **String** | **(define name “Prof. Lupoli”)** |
|  | *Holds a number of characters together* |
| **Functions (yeah)** | **(define ….)** |
|  | *can be a recursive variety of code* |
| **Symbols** |  |
|  | *Includes special Boolean symbols #t and #f* |

Input and Output

I/O stream

keyboard

🡨 stream 🡪

Computer

(read ) (printf)

Printing a Line of Text

* like many other languages you have been introduced to, Scheme also comes with a standard set of libraries

|  |
| --- |
| Print Example |
| ; File: helloWorld.scm  ; Written by: Prof. Lupoli  ; Date: 11/9/10  ; TAMU email: slupoli@cse.tamu.edu  ; Class: CSCE 314  ; Description: This is the first program in Scheme    (display "Hello World\n”)  (display "This is Lupoli's first Scheme Program!!\n”)    Draw a sizeable square on your paper. If it was a monitor what would like after the code above completed. |

Use the code above to display YOUR name on line ONE, and your town and state on line TWO

|  |
| --- |
| Other built-in commands (libraries) |
| |  | | --- | | > ([min](http://download.plt-scheme.org/doc/html/reference/numbers.html?q=min&q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._min%29%29) 1 3 2) | | 1 | | > ([min](http://download.plt-scheme.org/doc/html/reference/numbers.html?q=min&q=filter&q=min#%28def._%28%28quote._%7E23%7E25kernel%29._min%29%29) 1 3 2.0) | | 1.0 | |

Using display with variables

* only works with printf!
* and ONLY in some flavors of Scheme
  + your “flavor” may not have printf
    - if not, print individually using “display”
    - ***Gambit Scheme does not have this!!***

(define x 0) ; MUST DECLARE ALL VARIABLE BEFORE USING

(define y 8)

|  |  |
| --- | --- |
| Displaying multiple variables | |
| mzscheme | (printf “X is: ~s and Y is: ~s \n“ x y) |
| Gambit (gsi) | (display “X is:”)  (display x)  (display “Y is:”)  (display y) |

literal escape constants/command constants

|  |  |
| --- | --- |
| **Scheme Escape Sequences** | |
| Escape Sequence | Description |
| \t | tab |
| \r | carriage return, go to beg. of next line |
| \\ | backslash |
| \” | (define quote |
| \’ | single quote |
| \n | new line |
| \b | back space |
| \f | form feed |

What will these statements below printf??

(display “Hi Class!”)

(display “Good Luck!!! \n”)

(display “\t \t You’ll need it!!!”)

Other output commands

* there are several USEFUL output commands
  + (newline)
    - prints a new line, (display “\n”) does the same thing
  + (format “….” var1 var2 …)
    - exactly like C/Python placeholder syntax
    - exactly the same as printf
    - **again, may not be offered in some “flavors”**
* there are several output commands that ***are the same***
  + display/write/print
    - only useful for printing text with ***NO variables***
    - “display” ***DOES NOT*** add a newline at end
    - printf is not used in all Scheme flavors

Introduction to the Reader (Scanner)

## How to gather INPUT from the keyboard

## The scanner class is a STANDARDIZED class that uses different methods for READING in values from either the KEYBOARD or a FILE

## by default, the port assigned is usually the console

Intro. To Methods in scanner class

## Just remember to first

## IDENTIFY what exactly you wish to read in (or get from the user!!)

## HOW you want to use it.

### Remember a numeric value CAN be read in as a String!!

|  |  |  |
| --- | --- | --- |
| [Char](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html) | [(read](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html#next())-char )            Finds and returns the next character token from this scanner. | |
| **(define charVar #\ ) ; the space after #\ is important**  (display "Enter an CHAR value\n")  (set! charVar (read-char)) ; set! is used AFTER the definition  (display "the CHAR value is ~s \n" charVar) | | |
| String | [(](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html#nextDouble())read-line)            Scans the next token of the input as a String. If any input is taken BEFORE this string, you must clear the buffer by using a garbage line. | |
| (define garbageBuffer 0)  (display "Enter an LINE value\n")  (set! garbageBuffer(read-line)) ; get garbage out of buffer IF previous reading  (set! line (read-line)) | | |
| Numbers (int, floats, doubles) | | (read)  Finds and returns the next number token from this scanner. |
| (define floatVar 0)  (display "Enter an FLOAT value\n")  (set! floatVar (read))  (display "the FLOAT value is ~s \n" floatVar)  (display (+ (read) (read))) **; also works gathering input from user** | | |

Create a file named average.scm. Grab 3 (numeric) values from the user and display the average. Answerb:

What is a token??

See if you can figure it out from these examples??

|  |  |
| --- | --- |
|  | Token count |
| Lupoli | 1 |
| 98 | 1 |
| Prof. Lupoli! | 2 |
| 123.012 | 1 |
| Lupoli needs a vacation | 4 |
| ! ! ! | 3 |

|  |
| --- |
| Which Scanner method would you use? |

|  |  |  |
| --- | --- | --- |
|  | Datatype needed | Scanner method needed |
| Lupoli |  |  |
| 98 |  |  |
| Prof. Lupoli! |  |  |
| 123.012 |  |  |
| Lupoli needs a vacation |  |  |
| ! ! ! |  |  |

REMEMBER WE HAVE NO IDEA WHAT VALUE THE USER WILL ENTER!!!

A quick word on variables

* creating variables causes side effects
  + take up memory, allocation, speed, etc…
  + heaven forbid we had to do any casting!!
  + will learn more later
* we really just want an answer with little to so side effects

|  |  |
| --- | --- |
| The simple power of Functional Languages | |
| What we are used to | What Scheme (Functional) can do |
| (define x 0 )  (define y 0 )  (set! x (read))  (set! y (read))  (display (+ x y)) | (display (+ (read) (read)))  ; THIS is what we should be  ; doing!!! |

Use of Comments

|  |  |
| --- | --- |
| **Multi-line** | **Single line** |
| #|  File: helloWorld.scm  Written by: Prof. Lupoli  Date: 11/9/10  TAMU email: slupoli@cse.tamu.edu  Class: CSCE 314  Description: This is grabbing values  and displaying their average  |# | ; File: helloWorld.scm  ; Written by: Prof. Lupoli  ; Date: 11/9/10  ; TAMU email: slupoli@cse.tamu.edu  ; Class: CSCE 314  ; Description: This is grabbing values  ; and displaying their average  (define x 100) ; can comment after |

Why use comments?

* For notes
  + to yourself
  + to me!!
* For commenting out unfinished lines of code
  + skipping unfinished functions
* To understand what the code is doing!!
* watch where you put them!!

Reserved Words

* case sensitive
* can not be used as variable or function names
* each are a “form” (function!!!)
  + remember Functional Programming??
  + the have a name
  + parameters
  + and return something!

**case-lambda**

**call/cc**

**class**

**define-class**

**exit-handler**

**field**

**import**

**inherit**

**init-field**

**interface**

**let\*-values**

**let-values**

**let/ec**

**mixin**

**opt-lambda**

**override**

**protect**

**provide**

**public**

**rename**

**require**

**require-for-syntax**

**syntax**

**syntax-case**

**syntax-error**

**unit/sig**

**unless**

**when**

**with-syntax**

Style (like me!!! Stylin’)

* nested blocks
  + used for
    - compound statements
    - iteration (repeating loops of code)
    - conditional (if (x < 10)…)

|  |
| --- |
| **Style and a COMPLETE example w/ Scanner** |
| ***; complete header here***  (define (response x)  (if (boolean? x) "was boolean" "was not boolean")  )  (define (num-response x)  (if (rational? x) (\* x 10) "not rational" )  )  (define user #\Y) **; define variables**  (if (eq? user #\Y) **; will pass. eq? works with Characters**  (display "Then I will pass SCHEME, and learn some Java!")  )  (if(eq? user #\N) **; will fail**  (begin  (display "Then I will not pass Prof. Lupoli’s class.")  (display "And my parents will be upset!")  )  **; TWO LINES OR MORE**  ) |

FYI Section

|  |  |
| --- | --- |
| First Scanner Example | |
| ; File: helloWorld.scm  ; Written by: Prof. Lupoli  ; Date: 11/9/10  ; TAMU email: slupoli@cse.tamu.edu  ; Class: CSCE 314  ; Description: This is grabbing values and displaying their average    (define x -1)  (define y -1)  (define average 0)  (display "Enter two values\n")  (set! x (read)) ; set is used to change a variable  (set! y (read))  (set! average (/ (+ x y) 2))  (display "The average is ~s \n" average) ; display works with MzScheme   1. Identify where the declaration statements are located 2. Identify where the read command is located 3. Identify type of data (float, String, int) is being read in 4. Identify where the output is taking place | |
| **Inputting a Number** | Enter two values  20  40  The average is 30 |
| **Inputting a Decimal** | Enter two values  5.5  7.7  The average is 6.6 |
| **Inputting a String** | Enter two values  Lupoli  Luppold  +: expects type <number> as 1st argument, given: Lupoli; other arguments were: Luppold |

|  |
| --- |
| Input/Output Exercise |
| ; File: firstInputExercise.scm  ; Written by: You!!  ; Date: now  ; TAMU email: whateva@cse.tamu.edu  ; Class: CSCE 314  ; Description: This is grabbing values and displaying their name and address    (define name #\ )  (define address #\ )    ; 1. ) Create the CODE to LITERALLY display YOUR name, and address (No variables yet.)  ; 2.) Create the CODE to ask for user’s name and address, USE THE VARIABLES DECLARED FOR YOU ALREADY!! Hint: Which scanner functions will you need?  ; 3.) Create the code to display their name and address that THEY type in. NOT yours!! |

|  |
| --- |
| **First Input-Output Try** |
| ; read line  (display "Enter an LINE value\n")  ; THESE BELOW MAY BE CORRECT!!! BUFFER HAS STUFF IN IT  (set! garbageBuffer(read-line))  (set! line (read-line))  ;(set! line (readline ""))  (display "the LINE value is ~s \n" line) ; mscheme display  ; read int  (display "Enter an INTEGER value\n")  (set! intVar (read))  (display "the INTEGER value is ~s \n" intVar)  ; read float  (display "Enter an FLOAT value\n")  (set! floatVar (read))  (display "the FLOAT value is ~s \n" floatVar)  ; read double  (display "Enter an DOUBLE value\n")  (set! doubleVar (read))  (display "the DOUBLE value is ~s \n" doubleVar)  (exit) |

Using Dr. Racket, minor issue

* there is a secret to running all the scheme programs properly on dr. racket
* we need to add the line : #lang scheme
* at the top of every code file and then run it in dr. racket

Answers

|  |
| --- |
| average.scm |
| // display would be different in Gambit  ;File: helloWorld.scm  ;Written by:  ;Date:  ;TAMU email:  ;Class: CSCE 314  ;Description: This is the first program is Scheme  (display "Hello World\n")  (display "This is Patrick's first Scheme Program!!\n")  (define x 0)  (define y 8)  (define z 0)  (display "X is:~s and Y is: ~s\n" x y)  (display "Enter a three integer value\n")  (set! x (read))  (set! y (read))  (set! z (read))  (display "The int values are ~s , ~s , ~s\n" x y z) ; mzscheme display function  (exit) |