CSE 413: Programming Languages and their Implementation

Luke McDowell
Autumn Quarter 2004

2

Today's Outline

- Administrative Info
- Survey
- Overview of the Course
- Introduction to Scheme

3

Course Information

- Instructor: Luke McDowell, CSE 214
  lucasm@cs.washington.edu
  Office hours: 1:30-2:20 Mon, 3:30-4:20 Wed
- Teaching Assistant: Lincoln Ritter
  lritter@cs.washington.edu
  Office hours: See web page
- Text: *Concepts of Programming Languages*, Robert W. Sebesta, Sixth Edition
  » Fifth edition is fine.
- Other references available from web page
  » *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*
  » Link to *Structure and Interpretation of Computer Programs*

4

Course Policies

- Homeworks
  » Turned in electronically before 11:59pm on due date
  » Late homework not accepted
- Work in teams only on explicit team projects
  » Appropriate *discussions* encouraged – see website
  » Must give *credit* for any such discussions on your homework
- Approximate Grading

» Homework:	50%	
» Midterm:	25%	Wed November 3, in class
» Final:	25%	Tues December 14, 2:30-4:20

5

Course Mechanics

- 413 Web page:
  `http://www.cs.washington.edu/413`
- 413 mailing list
  » cse413a_au04@u.washington.edu
  » You should automatically be included if enrolled
- Course labs : Math Science Computing Labs
  » Basement of Communications building: B-022/027
  » Or work from home – all software available on course web

6

Course Paper

- Slide handouts will be provided
  » Also available on the web page
  » Not…
- Homeworks not handed out, see the web page

7

That Survey Thing

- Why are you taking my picture?
- What if I forgot everything?
- What if I know this all already?
- What if I'm the famous one?

8

References

- Section 15.5, *Concepts of Programming Languages*
- Section 2, *Revised⁵ Report on the Algorithmic Language Scheme (R5RS)*
- For more help:
  - » Sections 1-1.1.5, *Structure and Interpretation of Computer Programs* (Abelson, Sussman, & Sussman)

9

Elements of Programming

- Primitive expressions
  - » simplest entities of the language
- Means of combination
  - » by which compound elements are built
- Means of abstraction
  - » by which compound elements can be named and manipulated as units

10

There are many "languages"

- Computer programming
- Shell and scripting languages
- Applications
- Sciences

11

Training and Education

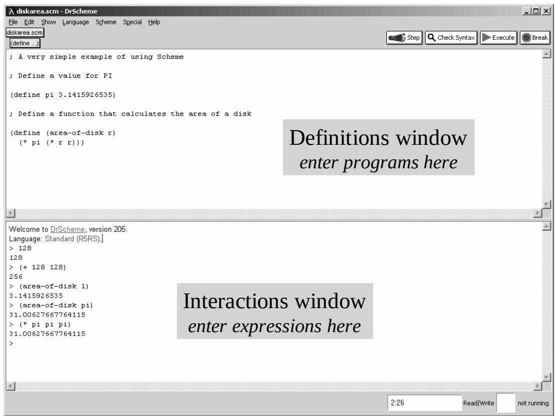
- Training
  - » learn the specifics of a known language
  - » build up a "tool chest" so that you can perform specific tasks in a particular field
- Education
  - » learn how to recognize valid abstractions and synthesize them in new and useful ways in many different knowledge domains
- We'll do some of both in this class

12

Why Scheme?

- The simplicity of the language lets us work on problem solving, rather than just syntax issues
- Flexibility of the language lets us see that the structure of C/Java/Basic is not the only way to express problem solutions
- Variety is the spice of life
  - » study more than one language paradigm and study the relationship between design paradigms
  - » professional programmers switch languages every few years anyway, so start practicing now

13



Definitions window

- Define programs in the Definitions window
  - » save the contents of the window to a file using menu item File - Save Definitions As …
  - » load existing files with menu item File - Open
  - » execute the contents of the definitions window by clicking on the "Execute" (or “Run”) button
  - » check and highlight syntax by clicking on the "Check Syntax" button
  - » Re-indent all with control-i

15

Interactions Window

- Evaluate simple expressions directly in the Interactions window
- Position the cursor after the ">", then type in your expression
  - » DrScheme responds by evaluating the expression and printing the result
  - » recall previous expression with escape-p
- Expressions can reference symbols defined when you executed the Definitions window

16

Think functionally

- Programming that makes extensive use of assignment is known as
  - » The order of assignments changes the operation of the program because the state is changed by assignment
- Programming without the use of assignment statements is known as
  - » In such a language, all procedures implement well-defined mathematical functions of their arguments whose behavior does not change
- Scheme is heavily oriented towards *functional* style

17

Primitive Expressions

- constants
  - » integer :
  - » rational :
  - » real :
  - » boolean :
- variable names (symbols)
  - » Names can contain almost any character except white space and parentheses
  - » Stick with simple names like `value, x, iter,` …

18

Compound Expressions

- Either a *combination* or a *special form*

1. Combination : (operator operand operand …)
   - » there are quite a few pre-defined operators
   - » We can define our own operators
2. Special form
   - » keywords in the language
   - » eg, define

19

Combinations

- (operator operand operand …)
- this is *prefix* notation, the operator comes first
- a combination always denotes a procedure application
- the operator is a symbol or an expression, the applied procedure is the associated value
  - » +, -, abs, my-function, foop?
  - » characters like * and + are not special; if they do not stand alone then they are part of some name

20

Evaluating Combinations

- To evaluate a combination
  - » Evaluate the subexpressions of the combination
  - » Apply the procedure that is the value of the leftmost subexpression (the operator) to the arguments that are the values of the other subexpresions (the operands)
- For example

21

Percolate values up a tree

Evaluate

```
(* (+ 2 (* 4 6))
   (+ 3 5 7))
```

22

Evaluating Special Forms

- Special forms have unique evaluation rules
- **`(define x 3)`** is an example of a special form; it is not a combination
  - » the evaluation rule for a simple define is "associate the given name with the given value"
- There are more special forms which we will encounter, but there are surprisingly few of them compared to other languages

23

Procedures

24

References

- Section 15.5, *Concepts of Programming Languages*
- Section 4.1, *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*
- For more help:
  - » Sections 1.1.6-1.1.8, *Structure and Interpretation of Computer Programs* (Abelson, Sussman, & Sussman)

25

Recall the *define* special form

- Special forms have unique evaluation rules
- **`(define x 3)`** is an example of a special form; it is not a combination
  - » the evaluation rule for a simple define is "associate the given name with the given value"

26

Define and name a variable

- **`(define`** ⟨*name*⟩ ⟨*expr*⟩**`)`**
  - » **`define`** - special form
  - » *name* - name that the value of *expr* is bound to
  - » *expr* - expression that is evaluated to give the value for *name*
- **`define`** is valid only at the top level of a <program> and at the beginning of a <body>

27

Define and name a procedure

- **`(define (`**⟨*name*⟩ ⟨*formal params*⟩**`)`** ⟨*body*⟩**`)`**
  - » **`define`** - special form
  - » *name* - the name that the procedure is bound to
  - » *formal params* - names used within the body of procedure
  - » *body* - expression (or sequence of expressions) that will be evaluated when the procedure is called.
  - » The result of the last expression in the body will be returned as the result of the procedure call

28

Example definitions

```
(define pi 3.1415926535)

(define (area-of-disk r)
  (* pi (* r r)))

(define (area-of-ring outer inner)
  (- (area-of-disk outer)
     (area-of-disk inner)))
```

29

Defined procedures are "first class"

- Compound procedures that we define are used exactly the same way the primitive procedures provided in Scheme are used
  - » names of built-in procedures are not treated specially; they are simply names that have been pre-defined
  - » you can't tell whether a name stands for a primitive (built-in) procedure or a compound (defined) procedure by looking at the name or how it is used

30

Evaluation example

- **`(area-of-ring 4 1)`**

31

Booleans

- Recall that one type of data object is boolean
  - » **#t** (true) or **#f** (false)
- We can use these explicitly or by calculating them in expressions that yield boolean values
- An expression that yields a true or false value is called a predicate
  - » **#t** =>
  - » **(< 5 5)** =>
  - » **(> pi 0)** =>

32

Conditional expressions

- As in all languages, we need to be able to make decisions based on inputs and do something depending on the result

Predicate **Consequent**

33

Special form: **cond**

- **(cond** ⟨*clause*$_1$⟩⟨*clause*$_2$⟩ **...** ⟨*clause*$_n$⟩**)**
- each clause is of the form
  - » **(**⟨*predicate*⟩ ⟨*expression*⟩**)**
- the last clause can be of the form
  - » **(else** ⟨*expression*⟩**)**

34

Example: sign.scm

```
; return the sign of x as -1, 0, or 1

(define (sign x)
  (cond
    ((< x 0) -1)
    ((= x 0) 0)
    ((> x 0) +1)))
```

35

Special form: **if**

- **(if** ⟨*predicate*⟩⟨*consequent*⟩⟨*alternate*⟩**)**
- **(if** ⟨*predicate*⟩⟨*consequent*⟩ **)**

36

Examples : abs.scm

```
; absolute value function
(define (abs a)
```

37

Examples : true-false.scm

```
; return 1 if arg is true, 0 if arg is false
(define (true-false arg)
```

38

Logical composition

- **(and** $\langle e_1 \rangle \langle e_2 \rangle \ldots \langle e_n \rangle$**)**
- **(or** $\langle e_1 \rangle \langle e_2 \rangle \ldots \langle e_n \rangle$**)**
- **(not** $\langle e \rangle$**)**

- Scheme interprets the expressions e_i one at a time in left-to-right order until it can tell the correct answer

39

in-range.scm

```
; true if val is lo <= val <= hi

(define (in-range lo val hi)
  (and (<= lo val)
       (<= val hi)))
```

40

Newton's method for square root

- Guess a value y for the square root of x
- Is it close enough to the desired value $\sqrt[2]{x}$?
  - » ie, is y^2 close to x?
- If yes, then done. Return recent guess.
- If no, then new guess is average of current *guess* and $\frac{x}{guess}$
- Repeat with new guess

41

sqrta.scm

```
; Square root using Newton's method

(define (average a b)
  (/ (+ a b) 2.0))

(define (good-enough? guess x)
  (< (abs (- (* guess guess) x)) 0.001))

(define (improve guess x)
  (average guess (/ x guess)))

(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x )))

(define (sqrta x)
  (sqrt-iter 1.0 x))
```

42

auxiliary functions

```
; Square root using Newton's method

(define (average a b)
  (/ (+ a b) 2.0))

(define (good-enough? guess x)
  (< (abs (- (* guess guess) x)) 0.001))


(define (improve guess x)
  (average guess (/ x guess)))
```

43

iterator and main functions

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x )))


(define (sqrta x)
  (sqrt-iter 1.0 x))
```

44

sqrt-iter

- Our first example of recursion
- Note that this recursion is used to implement a loop (an iteration)
  - » We will see this over and over in Scheme
- Iteration is calling the same block of code with a changing set of parameters
- The syntax of the procedure is recursive but the resulting process is iterative
  - » more on this later

45