Programming Languages

Scheme part 1

2020

Instructor: Odelia Schwartz

Using Scheme interpreter

- We will run code using Chicken Scheme
- Installing on your computer:

  https://wiki.call-cc.org/platforms

  See also manual:
  http://wiki.call-cc.org/man/5/The%20User%27s%20Manual

- Can also run online with different interpreter, works on simple examples I have tested:
  https://repl.it/languages/scheme

Using Scheme interpreter

Using Chicken Scheme:

- Type csi in the terminal. It will open the chicken interpreter.
- ,q to quit
- Chicken interpreter uses lower case for reserved words (book and some other interpreters use upper case)

Using Scheme interpreter

Our department computer also has Chicken Scheme:

- Log onto Johnston
- Then log onto one of the computers, such as wilderness etc.
- Type csi in the terminal. It will open the chicken interpreter

Primitive numeric functions

- Basic arithmetic: +, -, *, /
- Open csi for the following expressions

```
(* 3 7)

(- 5 6)

(- 15 7 2)

(- 24 (* 4 3))

(- 24 * 4 3)
```

Primitive numeric functions

- Basic arithmetic: +, -, *, /
- Open csi for the following expressions

(-5 6)

‘(-5 6)

‘(-5 6)

Introduction

- Other built in math functions:

  modulo, round, max, min, log, sin, sqrt

(sqrt 5)

(sqrt (round 5.1))

Remember: Chicken scheme, reserved words lower case

Lambda functions

- Nameless function:

```
(lambda (x) (* x x))
```

- Evaluate for parameter:

```
((lambda (x) (* x x)) 3)
```

- Can have multiple params:

```
((lambda (a b) (+ a b)) 4 5)
```

- With map:

```
(map (lambda (num) (* num num num)) '(3 4 2 6))
```

Define

- define used in two ways:

(1) Binds a name to a value:

```
(define pi 3.14159)
(eval pi)

(define two-pi (* 2 pi))
(eval two-pi)
```

Define

- define used in two ways:

(1) Binds a name to a value:

```
(define pi 3.14159)
(eval pi)

(define two-pi (* 2 pi))
(eval two-pi)
```

Equivalent to:
Java:
final float pi = 3.14159
final float two-pi = 2.0 * pi

Equivalent to:
C/C++:
const float pi = 3.14159
const float two-pi = 2.0 * pi

Define

- define used in two ways:

(2) Binds a name to a lambda expression:

Format:

```
(define (function_name parameters)
        (expression)
)
```

"

Define

- define used in two ways:

(2) Binds a name to a lambda expression:

Example:

```
(define (square number) (* number number))

(square 5)

(square 5.1)
```

Define

- define used in two ways:

(2) Binds a name to a lambda expression:

Another example: hypotenuse: length (longest side) of right triangle given two other sides

```
(define (hypotenuse side1 side2)
  (sqrt (+ (square side1) (square side2)))
)

(hypotenuse 3 4)
```

Define

- define used in two ways:

(2) Binds a name to a lambda expression:

Another example:

```
(define (hypotenuse side1 side2)
   (sqrt (+ (square side1) (square side2)))
)

(hypotenuse 3 4)
```

returns 5

Numeric predefined predicate functions

- =
- <>
- >
- <
- >=
- <=
- even?
- odd?
- zero?

Numeric predefined predicate functions

- =
- <>
- >
- <
- >=
- <=
- even?
- odd?
- zero?

Examples:

(even? 5)
(>= 7 6)

Numeric predefined predicate functions

- Two Boolean values:

#t

#f

Numeric predefined predicate functions

- Two Boolean values:

#t

#f

- Empty list evaluates as false
- Non empty list evaluates as true

Numeric predefined predicate functions

- Two Boolean values:

#t

#f

- Empty list evaluates as false
- Non-empty list evaluates as true

Similar to C integers as Boolean…

Control flow

- If expression

1. (if predicate then_expression else_expression)

Control flow

- If expression

1. (if predicate then_expression else_expression)

Example:

Write a function for computing factorial

- Use define for defining the function name
- Use if statement for control

Control flow

- If expression

1. (if predicate then_expression else_expression)

Example:

(define (factorial n)

if statement in here…

)

Control flow

- If expression

1. (if predicate then_expression else_expression)

Example:

```
(define (factorial n)
   (if (<= n 1)
    1
   (* n (factorial (- n 1)))
   )        ;this is a comment. end if
)           ;end define
```

Control flow

- If expression

1. (if predicate then_expression else_expression)

Example:

```
(define (factorial n)
   (if (<= n 1)
    1
   (* n (factorial (- n 1)))
   )         ;this is a comment. end if
)            ;end define

(factorial 4)
```

Control flow

- If expression

1. (if predicate then_expression else_expression)

Note: We can create a file called factorial.scm
with this code

(load “factorial.scm”)
(factorial 4)

Control flow

- Cond statement

2. Multiple selection via cond:

```
( cond
   (predicate_1 expression_1)
   (predicate_2 expression_2)
   …
   (predicate_n expression_n)
   [ (else expression_n+1) ]   ;optional
)
```

Control flow

- Cond statement

2. Multiple selection via cond:

```
( cond
   (predicate_1 expression_1)
   (predicate_2 expression_2)
   …
   (predicate_n expression_n)
   [ (else expression_n+1) ]   ;optional
)
```

Predicates evaluated one at a time from first line, until one evaluates to #t. The corresponding expression is then evaluated and returned. If none evaluate #t then else is evaluated and value returned…

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

Write a function (compare x y) that returns:

“x is greater than y” if x>y

“y is greater than x” if y>x

“x and y are equal” otherwise

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (compare x y)
  (cond
   ((> x y) "x is greater than y")
    ((< x y) "y is greater than x")
    (else  "x and y are equal")
  )
)
```

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (compare x y)
  (cond
   ((> x y) "x is greater than y")
    ((< x y) "y is greater than x")
    (else  "x and y are equal")
  )
)
```



```
(compare 5.1 5.1)
```

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (leap? year)
   (cond
   ((zero? (modulo year 400)) #t)


)) ;ends define and cond
```

If can be divided by 400 evenly then leap year (evaluates to #t)

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (leap? year)
   (cond
   ((zero? (modulo year 400)) #t)
   ((zero? (modulo year 100)) #f)

)) ;ends define and cond
```

If can be divided by 100 evenly then NOT leap year (evaluates #f)

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (leap? year)
   (cond
   ((zero? (modulo year 400)) #t)
   ((zero? (modulo year 100)) #f)
   (else (zero? (modulo year 4)))
)) ;ends define and cond
```

Otherwise if divisible by 4 then leap year is #t and if not divisible by 4 leap year is #f

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (leap? year)
  (cond
  ((zero? (modulo year 400)) #t)
  ((zero? (modulo year 100)) #f)
  (else (zero? (modulo year 4)))
)) ;ends define and cond
```

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (leap? year)
  (cond
  ((zero? (modulo year 400)) #t)
  ((zero? (modulo year 100)) #f)
  (else (zero? (modulo year 4)))
)) ;ends define and cond
```

Returns value of last expression in line that evaluates to true

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (leap? year)
   (cond
   ((zero? (modulo year 400)) #t)
   ((zero? (modulo year 100)) #f)
   (else (zero? (modulo year 4)))
)) ;ends define and cond
```

Try leap? On 2020 and 2021

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (leap? year)
   (cond
   ((zero? (modulo year 400)) #t)
   ((zero? (modulo year 100)) #f)
   (else (zero? (modulo year 4)))
)) ;ends define and cond

(leap? 2020)
(leap? 2021)
```

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (leap? year)
   (cond
   ((zero? (modulo year 400)) #t)
   ((zero? (modulo year 100)) #f)
   (else (zero? (modulo year 4)))
)) ;ends define and cond

(leap? 2020)
(leap? 2021)
```

Control flow

- Cond statement

2. Multiple selection via cond:

Example:

```
(define (leap? year)
   (cond
   ((zero? (modulo year 400)) #t)
   ((zero? (modulo year 100)) #f)
   (else (zero? (modulo year 4)))
)) ;ends define and cond

(leap? 2020)
(leap? 2021)
```

List functions

- Returning an element or list

(quote a)

(quote (a b c))

List functions

- Returning an element or list

(quote a)

(quote (a b c))

Abbreviation:

‘a

‘(a b c)

List functions

- Returning an element or list

(quote a)

(quote (a b c))

Abbreviation:

‘a

‘(a b c)

Why the need for quote?

List functions

- Returning an element or list

(quote a)

(quote (a b c))

Abbreviation:

‘a

‘(a b c)

Why the need for quote?
In Scheme and some other Functional languages, data and code have same format. This tells the Interpreter it is data

List functions: car, cdr

- car takes a list and returns first element

```
(car '(a b c))

(car '((a b) c d ))

(car 'a)
```

List functions: car, cdr

- car takes a list and returns first element

(car ‘(a b c))

(car ‘((a b) c d))

(car ‘a)

We got an error…

List functions: car, cdr

- car takes a list and returns first element

(car '(a b c)) returns a

(car '((a b) c d)) returns (a b)

(car 'a) error since a is not a list

List functions: car, cdr

- car takes a list and returns first element

(car ‘(a b c))	returns a
(car ‘((a b) c d))	returns (a b)
(car ‘a)	error since a is not a list
(car ‘(a))	
(car ‘())	

List functions: car, cdr

- car takes a list and returns first element

(car ‘(a b c))	returns a
(car ‘((a b) c d))	returns (a b)
(car ‘a)	error since a is not a list
(car ‘(a))	returns a
(car ‘())	error….

List functions: car, cdr

- cdr takes a list and returns list after removing first element

(cdr ‘(a b c))

(cdr ‘((a b) c d))

(cdr ‘a)

(cdr ‘(a))

(cdr ‘())

List functions: car, cdr

- cdr takes a list and returns list after removing first element

(cdr ‘(a b c))	return (b c)
(cdr ‘((a b) c d))	returns (c d)
(cdr ‘a)	error
(cdr ‘(a))	returns ()
(cdr ‘())	error

List functions: car, cdr

- car and cdr

Names carried over from IBM 704
address and decrement parts of register

Names not intuitive…

I remember a comes before d …

List functions: car, cdr

- Define a function named second that returns the second element in a list, using car and cdr

List functions: car, cdr

- Define a function named second that returns the second element in a list, using car and cdr

```
(define (second a_list) (car (cdr a_list)))

(second '(a b c d))
```

List functions: car, cdr

- Define a function named second that returns the second element in a list, using car and cdr

```
(define (second a_list) (car (cdr a_list)))

(second '(a b c d))
```

Returns b

Other variants of car, cdr

- (caar x) equivalent to (car (car x))

Other variants of car, cdr

- (caar x) equivalent to (car (car x))

Example:

(caar ‘((a) b c d))

(car (car ’((a) b c d)))

Other variants of car, cdr

- (caar x) equivalent to (car (car x))

Example:

(caar ‘((a) b c d))

(car (car ’((a) b c d)))

Answer a

Other variants of car, cdr

- Can keep going with it…
- Any combo of a, d up to 4 legal in-between!

Other variants of car, cdr

- Can keep going with it…
- (caddar x) equiv to (car (cdr (cdr (car x))))

Other variants of car, cdr

- Can keep going with it…
- (caddar x) equiv to (car (cdr (cdr (car x))))

Example:

(caddar '((a b (c) d) e))

Other variants of car, cdr

- Can keep going with it…
- (caddar x) equiv to (car (cdr (cdr (car x))))

Example:

(caddar ‘((a b (c) d) e))

Answer (c). Why?

Other variants of car, cdr

- Can keep going with it…
- (caddar x) equiv to (car (cdr (cdr (car x))))

Example:

(caddar ‘((a b (c) d) e))

Answer (c)

Because:
1st inner car = (a b (c) d)
Next inner cdr = (b (c) d)
Next cdr = ((c) d)
Final outer car = (c)

Creating a list

- Two ways

- cons: takes two params, the first either an atom or a list, and the second a list. Returns a new list with first param as first element, and second param as remainder of the result.

Creating a list

- Two ways
- cons: takes two params, the first either an atom or a list, and the second a list. Returns a new list with first param as first element, and second param as remainder of the result.

Example: (cons ‘a ‘(b c))

Returns?

Creating a list

- Two ways

- cons: takes two params, the first either an atom or a list, and the second a list. Returns a new list with first param as first element, and second param as remainder of the result.

Example: (cons ‘a ‘(b c))

Returns? (a b c)

Creating a list

- Two ways

- cons: takes two params, the first either an atom or a list, and the second a list. Returns a new list with first param as first element, and second param as remainder of the result.

Example: (cons ‘a ‘())

Returns?

Creating a list

- Two ways

- cons: takes two params, the first either an atom or a list, and the second a list. Returns a new list with first param as first element, and second param as remainder of the result.

Example: (cons ‘a ‘())

Returns? (a)

Creating a list

- Two ways

- cons: takes two params, the first either an atom or a list, and the second a list. Returns a new list with first param as first element, and second param as remainder of the result.

Example: (cons ‘() ‘(a b))

Returns (() a b)

Creating a list

- Two ways

- cons: takes two params, the first either an atom or a list, and the second a list. Returns a new list with first param as first element, and second param as remainder of the result.

Example: (cons ‘(a b) ‘(c d))

Returns ((a b) c d)

Taking a list apart

And putting it back together

- car and cdr take a list apart
- cons constructs a new list from two given parts

Taking a list apart
And putting it back together

- What does this function do to list parameter a_list?

(cons (car a_list) (cdr a_list))

Taking a list apart

And putting it back together

- What does this function do to list parameter a_list?

(cons (car a_list) (cdr a_list))

Answer: returns list with exact same structure as a_list

Taking a list apart
And putting it back together

- What does this function do to list parameter a_list?

(cons (car a_list) (cdr a_list))

Answer: returns list with exact same structure as a_list

Example:
(cons (car '(a b c)) (cdr '(a b c))) = (a b c)

Creating a list

- Two ways
- list: takes any number of params; returns a list with the params as elements

Creating a list

- Two ways
- list: takes any number of params; returns a list with the params as elements

Example: (list 'apple 'orange 'grape)

Creating a list

- Two ways
- list: takes any number of params; returns a list with the params as elements

Example: (list 'apple 'orange 'grape)

Answer: (apple orange grape)

Creating a list

- Two ways
- cons would be more tedious for generating the list (apple orange grape) …

Try it!

Creating a list

- Two ways

- cons would be more tedious for generating a list (apple orange grape) …

Example: start from the end

(cons 'grape '())

Results in (grape)

Then would need to add orange and then apple…

Creating a list

- Two ways
- cons would be more tedious for generating a list (apple orange grape) …

Example: (cons 'apple (cons 'orange (cons 'grape '())))

Answer: (apple orange grape)

Creating a list

- Two ways
- cons would be more tedious for generating a list (apple orange grape) …

Example: (cons 'apple (cons 'orange (cons 'grape '())))

Answer: (apple orange grape)

Why would we still want to use this?

Creating a list

- Two ways

- cons would be more tedious for generating a list (apple orange grape) …

Example: (cons 'apple (cons 'orange (cons 'grape '())))

Answer: (apple orange grape)

Why would we still want to use this?
Because of how it works with car and cdr (taking a list apart versus putting it together). We will see this later in recursions.

Creating a list

- Summary: Two ways
- cons: takes two params, the first either an atom or a list, and the second a list. Returns a new list with first param as first element, and second param as remainder of the result.
- list: takes any number of params; returns a list with the params as elements.