Lisp, Scheme, and Racket

The Lisp programming language was created by John McCarthy in 1958, making it one of the oldest high-level programming languages. It was designed to meet the needs of early artificial intelligence researchers, and it was widely used in that field for decades. In addition to being powerful, Lisp is an elegant, expressive language with a minimal amount of syntax. To be only slightly unfair, compare these versions of the classic “Hello world” program:

Lisp:

"Hello world"

Naming no names:

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

Because of this simplicity, a dialect of Lisp, called Scheme, became a popular choice for teaching computer science. After the PLT Scheme implementation diverged from the original Scheme language, it was renamed Racket. Given this history, the DrRacket software we use in this class supports using several related languages, which you can select from the Language menu.

If you used Prof. Hunsberger’s textbook, *Introduction to Computer Science via Scheme*, for CMPU 101, you used “Full Swindle”, which is (essentially) the traditional Scheme language.

If you used the *How to Design Programs*, you actually used multiple specially designed teaching languages: Beginning Student, Beginning Student with List Abbreviations, Intermediate Student, and Intermediate Student with λ. These are essentially Racket, but with some simplifications of syntax and behavior to help new programmers.

For this class, we will use the modern Racket language. However, most of what we write will also be perfectly correct Scheme. Whichever textbook you used for 101, almost everything will work the way you remember, but where there are differences, I’ll try to point them out during the semester.

---

1 At present, the most popular programming language for artificial intelligence is Python, which borrows many ideas from Lisp.
Interacting with DrRacket

The DrRacket window is divided into the definitions pane and the interactions pane. You can type expressions in the interactions pane and they will be immediately evaluated, e.g.,

```scheme
> 5
5
> "I love lambdas!"
"I love lambdas!"
```

These expressions are values, which evaluate to themselves. Other expressions call the functions that are provided by the language or that you define, e.g.,

```scheme
> (random 3)
1
```

In this case, the argument to the random function tells it the upper bound \( u \) for generating a random integer in the range \([0, u - 1]\). So, \((\text{random } 1)\) will generate a random integer between 0 and 0, and \((\text{random } 5)\) will generate a random integer between 0 and 4.

If you omit the argument or give an invalid argument, e.g., 0, DrRacket will show an error.

Definitions

While the interactions pane is ideal for experimenting, when you want to design a program, you don’t want to re-enter it into the interactions pane every time. Instead, you write the program in the definitions pane, which can be saved.

We can use `define` to give names to values:

```scheme
(define x 3)
```

And to define our own functions:

```scheme
(define (my-square x) (* x x))
```

But there’s nothing special about defining functions; this is a shorthand for giving a name to a function value, i.e., a lambda expression:

```scheme
(define my-square (lambda (x) (* x x)))
```

We can tell DrRacket to evaluate the contents of the definitions pane and then we can call our function: `(my-square 5) \rightarrow 25`.

When we want to describe what an expression evaluates to more compactly, we just draw an arrow, e.g., \((\text{random } 3) \rightarrow 0\).

If you want to repeat the previous line in the interactions panel, type Ctrl-↑ or Esc-p.
Local Definitions

In general, we want to avoid using global variables. Instead, declare local variables inside your functions using `let`:

```lisp
(let [(var1 val1)
       (var2 val2)
       ...
       (varn valn)]
  <body using those variables>)
```

This corresponds with the `local` expressions you may have used in the teaching languages:

```lisp
(local [(define up "down")
         (define down "up")]
       (list up down))
```

That syntax will still work in Racket, but we prefer writing it this way, where the defines are implicit:

```lisp
(let [(up "down")
       (down "up")]
       (list up down))
```

Data, Symbols, and Quoting

While Racket includes data types seen in most languages, such as strings, Boolean values, and numbers, it also allows you to use symbols. When we give a name to variable or a function, we are using symbols:

```lisp
(define (foo bar) bar)
(define baz 1)
```

Here `foo` is a symbol whose value is a function, `bar` is a symbol that only has a value inside the body of the `foo` function when it is called, and `baz` is a symbol with the value `1`.

In mathematics, we can talk about “the variable `x` in the equation”, without `x` having a value. Likewise in Racket we can use a symbol without it having an associated value. To do this, we quote the symbol, telling Racket not to try to evaluate it:

```lisp
> (define x 42)
> x
42
> (quote x)
'x
> 'x
'x
```

Using a single quote before an expression is a shortcut for calling the special `quote` operator.
Lists

Lists are the fundamental data structure in any derivative of Lisp, the list processor language. Lists are defined recursively, using the cons function to construct a cell with a first element, which can be anything, and the rest, which is also a list:

- **cons**: Construct a new cons cell – a part of a list – consisting of the specified element followed by the rest, e.g.,
  
  `(cons 'x '(a b)) → '(x a b)
  `(cons '(a b) '(c d e)) → '(((a b) c d e)

Lists can contain any kind of value, including other lists, and they can contain any number of values. If a list has zero values, it’s called the empty list, written as `()` or created by calling list with no arguments: (list).

- **list?**: Test if something is a list.
- **empty?** (or null?): Test if something is an empty list, e.g.,

  `(empty? '()) → #t
  `(empty? '(x)) → #f

- **length**: Get the number of elements in a list. Note that this requires traversing the entire list. Therefore, the longer the list is, the slower it will be to compute its length. It’s a very bad idea to compute the exact length of a list if all you want to know is whether it’s empty!

There are built-in functions to access the two parts of a cons cell, letting you traverse a list:

- **first** (or **car**): Return the first element of a given list, e.g.,

  `(first '(a b)) → 'a

- **rest** (or **cdr**): Return the list of all elements except the first, e.g.,

  `(rest '(a b)) → '(b)

Other convenient functions to know for working with lists include append, reverse, list-ref, last, and member.

Functions and Values

Functions are values, just like numbers, strings, or images. As such, we can pass a function as an argument to another function – or have a function return another function.

For example, this function consumes a function and uses it to generate a list:

```
When the second argument is not a list, cons makes a “dotted pair”, e.g.,

`(cons 'a 'b) → '(a . b)
```

However, usually you’ll only do this by mistake!

The names car and cdr are **historical**, in the original Lisp interpreter, car was an abbreviation for “contents or the address part of the register” and cdr for “contents of the decrement part of the register” – terminology specific to the IBM 704 the interpreter ran on. Today we tend to use first and rest for clarity, but it wouldn’t be a Lisp-derived language without car and cdr.
(define (one-two-three fn)
  (list (fn 1) (fn 2) (fn 3)))
(one-two-three sqr) → '(1 4 9)

We often pass a function as an argument when we're writing a general function whose behavior we want to control. E.g., the built-in map function consumes the name of a function that it applies to each element of a list:

(map sqr '(1 -2 3 -4)) → '(1 4 9 16)
(map abs '(1 -2 3 -4)) → '(1 2 3 4)

Writing a function to return a function is less common, but it can be convenient. The following function returns a function that adds the specified value to its own argument, i.e., it returns a more specific function:

(define (make-adder x)
  (lambda (y) (+ x y)))
(make-adder 5) → #<procedure>
((make-adder 5) 10) → 15

If we plan to re-use it, we can give a name to the function that make-adder returns:

(define add-5 (make-adder 5))
(add-5 10) → 15

Calling Functions on Lists

- apply: Applies a function to the arguments found in a list.
  (apply + '(3 3)) → 6

- map: Apply a function to each top-level element in a list, one by one, and return a list of the resulting values. If the function accepts multiple values, multiple lists of the same length are provided:
  (map odd? '(1 2 3)) → (#t #f #t)
  (map (lambda (x) (* 2 x)) '(1 2)) → '(2 4)
  (map + '(1 2 3 4) '(10 20 30 40)) → '(11 22 33 44)

- andmap: Check if mapping a function over a list returns #t each time.
  (andmap odd? '(1 2 3)) is equivalent to
  (and (odd? 1) (odd? 2) (odd? 3))

- ormap: Check if mapping a function over a list ever returns #t:

sqr squares a number, i.e., multiplies it by itself.
abs takes the absolute value of a number, i.e., its distance from 0 on a number line.
(ormap even? '(1 2 3)) is equivalent to
(or (even? 1) (even? 2) (even? 3))

- **filter**: Return only the values from a list that match a predicate, e.g.,

  (filter odd? '(1 2 3 4 5 6 7 8 9)) → '(1 3 5 7 9)

**Branches**

It's good style to choose the most specific conditional:

- **when** or **unless** when there's only one branch, e.g.,

  (when (> x 3)
    (print "That's too big."))

  (unless (list? x)
    (print "Expected a list!"))

- **if**: For at most two branches, e.g.,

  (if (= x 3)
    (print "x is 3")
    (print "x is not 3!"))

- **cond**: For multiple branches. Prefer this over nesting if statements; it's easier to read.

  (cond [(test1) (form1)]
        [(test2) (form2)]
        ...
        [(testn) (formn)])

  The conditions are evaluated sequentially. The default, "otherwise" condition is else or #t, since it's always true. e.g.,

  ;; Sign function: Return -1 if x is negative, 1 if positive, and 0 otherwise.
  (define sign
    (lambda (x)
      (cond [(< x 0) -1]
            [(> x 0) 1]
            [else 0])))

- You don't need a conditional when the results are Boolean values:

  (if (and (number? x) (> 0 x)) #t #f)

  is redundant; just say (and (number? x) (> 0 x)).
Common Predicates

- **equal?**: Determines equality between the ordered contents of two given lists or strings. It also works on numbers and characters. Prefer this over `eq?`, `eqv?`, and `=` unless you *know* that you want to use one of those or speed is a concern.

- **null?**: Determines whether a given argument is the empty list.

- **and, or, not**: Logical predicates. Note that `or` isn't exclusive-or.

- **list?, number?, integer?, string?, etc.**: Type-checking predicates.

- **even?, odd?, >, <, >=, <=**: Numeric predicates.

Example Problem: Transforming Lists

```scheme
(define office-data '((Mary 221) (John 308) (Jane 221)))
```

Given a list of people's offices, above, let's get a list of the names of everyone who has an office. Since a person's name is the first element of each "entry" sublist, we need to apply the `first` function to each list element in turn and return the result:

```scheme
;; Returns the list of people's names included in the given office list.
(define get-names
  (lambda (officelist)
    (map first officelist)))

(get-names office-data) → '(Mary John Jane)
```

What if we want to get the list of floors everyone is on? Well, the floor is the first digit of the office number, so we break the problem down: We write a function `get-floor` that turns an office number into a floor number, and we apply it to each element of the office list:

```scheme
;; Given an entry with the name and office number, returns the name and
;; the floor where the office is.
(define get-floor
  (lambda (entry)
    (list (first entry) (floor (/ (second entry) 100)))))

;; Find out the floors on which people are located.
(define get-floors-wrapper
  (lambda (officelist)
    (map get-floor officelist)))
```

Since we're only going to use `get-floor` when we're using `get-floors-wrapper`, we don't need two separate functions. We can "inline" it by using an unnamed lambda function:
;; Find out the floors on which people are located.
(define get-floors
  (lambda (officelist)
    (map (lambda (entry)
           (list (first entry) (floor (/ (second entry) 100))))
         officelist)))
(get-floors office-data) → '((Mary 2) (John 3) (Jane 2))

And if we just want a list of all the floors people are on?

(define get-floors-a
  (lambda (officelist)
    (remove-duplicates
     (map second (get-floors officelist))))))
(get-floors-a office-data) → '(2 3)

Exercises

Always include test cases!

1 Write a function to convert Fahrenheit to Celsius. The formula for this is usually given as $c = \frac{5}{9}(f - 32)$, where $f$ is the temperature in deg. F. and $c$ is the temperature in deg. C.

2 Define a recursive function, every-other-one, that takes a list as its only input and returns a list containing “every other” element of the list.

3 Define a function called toss-die that takes no input and returns a randomly generated value between 1 and 6.

   Define a function sum-n-die-tosses that takes a single input $n$. It should return as its output the sum of $n$ tosses of a six-sided die.

   Define a function, called avg-n-die-tosses, that takes a single input $n$ and returns as its output the average of $n$ tosses of a six-sided die.

Acknowledgments

This tutorial adapts descriptions and examples from Matthew Butterick (Beautiful Racket), Matthew Flatt, Luke Hunsberger, Peter Norvig, George Springer & Daniel Friedman (Scheme and the Art of Computer Programming), and Jennifer Walter.
Exercise Solutions

1 (define fahr-to-cels
   (lambda (f)
     (* 5/9 (- f 32.0)))
(fahr-to-cels 212) \rightarrow 100.0
(fahr-to-cels 32) \rightarrow 0.0
(fahr-to-cels -40) \rightarrow -40.0
(fahr-to-cels 0) \rightarrow -17.77777777777778

2 (define every-other-one
   (lambda (l)
     (cond ((or (null? l) (null? (rest l)))
           l)
           (else
             (cons (first l)
                   (every-other-one (rest (rest l))))))))
(every-other-one '(a b c d e f g h)) \rightarrow '(a c e g)
(every-other-one '(a b c d e f g h i)) \rightarrow '(a c e g i)
(every-other-one '(1)) \rightarrow '(1)
(every-other-one '()) \rightarrow '()

3 (define toss-die
   (lambda ()
     (+ 1 (random 6)))

(define sum-n-die-tosses
   (lambda (n)
     (if (<= n 0)
         0
         (+ (toss-die)
            (sum-n-die-tosses (- n 1)))))

(define avg-n-die-tosses
   (lambda (n)
     (/ (sum-n-die-tosses n) n 1.0)))