CS 135
Computer Science

Dr. Holby • Fall 2014 • University of Waterloo

Last Revision: December 6, 2014

Table of Contents

1 Introduction **1**
- 1.1 Wise words from a wise teacher . . . 1
- 1.2 Programming Language Design . . . 1
- 1.3 Racket . . . 1
- 1.4 Themes of the Course . . . 2

2 Functional Programming **2**
- 2.1 Introduction to Functional Programming . . . 2
- 2.2 Function Definition . . . 3

3 Software Design Recipe **4**
- 3.1 Programs as Communication . . . 4
- 3.2 Design Recipe Components . . . 4
- 3.3 Suggested Procedure . . . 4
- 3.4 Boolean-valued functions . . . 5
- 3.5 Symbols . . . 5
- 3.6 String manipulation . . . 5
- 3.7 Conditions . . . 6
- 3.8 Testing . . . 6

4 Syntax and Semantics **6**
- 4.1 Spelling rules . . . 6
- 4.2 Substitution . . . 7

5 Structure **8**
- 5.1 Structure . . . 8
- 5.2 Stepping . . . 8

6 Lists **9**
- 6.1 List Definition . . . 9
- 6.2 Extracting from Lists . . . 10
- 6.3 Recursion . . . 11
- 6.4 Design Recipe . . . 11
- 6.5 List Manipulation . . . 11
- 6.6 Characters . . . 11

7 Recursion **12**
7.1 Processing Two Lists 13
7.2 Types of Recursion 14

8 Trees **14**
8.1 Characteristics of Binary Trees 14
8.2 Binary Search Trees 15

9 Local Definitions **16**
9.1 Semantics 16
9.2 Function Abstraction 17
9.3 Higher Order Functions 17

10 Graphs **18**

11 History **20**

Future Modifications

1 Introduction

1.1 Wise words from a wise teacher

- Keep up with the readings
- Take notes in lecture
- Start assignments early
- Get help early
- Follow our advice on approaches to writing programs (eg. design recipe, templates)
- Keep on top of workload
- Visit office hours to get help
- Integrate exam study into your weekly routine.
- Go beyond the minimum required
- Maintain a "big picture" perspective: Look beyond the immediate task or topic
- Go over your assignments and exams, learn from mistakes
- Read your mail sent to your UW email.

1.2 Programming Language Design

Imperative: based on frequent changes to data.

- Eg: machine language, Java, C++, Turing, VB

Functional: based on the computation of new values rather than the transformation of old ones.

- Examples: Excel formulas, LISP, ML, Haskell, Erlang, F#, Mathematica, XSLT, Clojure
- More closely connected to mathematics.
- Easier to design and reason about programs

1.3 Racket

- A functional programming language
- Minimal but powerful syntax
- Small toolbox with ability to construct additional required tools
- Interactive evaluator
- Used in education and research since 1975
- A dialect of Scheme

1.4 Themes of the Course

- Design (the art of creation)
- Abstraction (finding commonality, neglecting details)
- Refinement
- Syntax, expressiveness(understandability), and semantics (meaning)
- Communication

2 Functional Programming

2.1 Introduction to Functional Programming

Definition 2.1 (function)**.** A **function** generalizes similar expressions and takes inputs to produce an output. These definitions consist of the **name** of the function, its **parameters**, and an **algebraic expression** using the parameters.

Definition 2.2 (application)**.** An **application** supplies arguments for the parameters, which are substituted into the algebraic expression.

Example 2.1. If a function is defined as $g(x, y) = g(x + y)$, an application would be $g(1, 3)$

Example 2.2. We **evaluate** each of the arguments to yield values. One example of evaluating is substitution.

Definition 2.3 (substitution)**.** Substitution must be done from innermost to outermost and then left to right. At each stage, only one substitution should be done. (If two evaluations have the same priority, the left most one is done first.)

Example 2.3. $f(x) = x^2 \qquad g(x, y) = x + y$
Incorrect: $g(g(1, 3), f(2)) = g(1, 3) + f(2)$ **Correct**: $g(g(1, 3), f(2)) = g(4, f(2))$

Example 2.4. If we treat infix operators (+,−) like functions, parentheses are not required to specify order of operations. $3 - 2$ becomes $-(3, 2)$
$(6 - 4)/(5 + 7)$ becomes $/(-(6, 4), +(5, 7))$

Example 2.5. In racket, the functions move inside the parentheses so $g(1, 3)$ becomes $(g\ 1\ 3$
$3 - 2 + 4/5$ becomes $(+(-\ 3\ 2)(/\ 4\ 5))$

Note. Extra paratheses are harmful in Racket because they are used to signal a function application or other Racket syntax.

2.2 Function Definition

Functions

A function definition consists of a name for the function, a list of parameters, and a single "body" expression.

Example 2.6. Let $f(x) = x^2$ and let $g(x, y) = x + y$
In racket:

```
(define (f x) (* x x))
(define (g x y) (+ x y))
```

Each parameter name for the function has meaning only within the body if that function (local variable)

Example 2.7. For the following code:

```
(define (f x y) (+ x y))
(define (g x z) (* x z))
```

the two uses of x are independent and the two function definitions below define the same function.

```
(define (f x y) (+ x y))
(define (f a b) (+ a b))
```

Constants

Advantages:

- Give meaningful names to useful values
- Reduces typing and errors when such values are modified
- Enhances readability
- Can be used in any expression, including the body of function definitions.
- Sometimes referred to as variables, but value may not be changed.

Example 2.8. There are brackets around a function and its parameters to indicate that it is a function. To define a constant, no brackets are required around it.
Let $k = 3$ and $p = 3^2$

```
(define k 3)
(define p ( * k k))
```

3 Software Design Recipe

3.1 Programs as Communication

Communication occurs between:

- You and the computer
- You and yourself in the future (;; Comments)
- You and others

For software design, programs should be: compatible, composable, correct, durable, efficient, extensible, flexible, maintainable, portable, readable, reliable, reusable, scalable, usable, and useful.

3.2 Design Recipe Components

1. **Purpose:** Describes what the function is to compute.
2. **Contract:** Describes what type of arguments the function consumes and what type of value it produces. (Num, Int, Nat (includes 0), Any, Boolean)
3. **Examples:** Illustrating the typical use of the function.
4. **Definition:** The Racket definition (header and body) of the function.
5. **Tests:** A representative set of inputs and expected outputs.

3.3 Suggested Procedure

Draft → Examples → Definition Header & Contract → Parameters → Definition → Tests.

Example 3.1. An example of a well formatted design recipe:

```
;; (sum-of-squares p1 p2) produces the sum of
;; the squares of p1 and p2
;; sum-of-squares: num num -> num
;; examples
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)

(define (sum-of-squares p1 p2)
  (+ (* p1 p1) (* p2 p2)))

;;Tests
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
```

If there are additional constraints on the inputs, the contract section may be extended to add a requires section.

```
;; (my-function a b c)
;; my-function: num num num -> num
;; requires: 0 < a < b
;;       c > 0
```

3.4 Boolean-valued functions

Definition 3.1. A **predicate** is a function that produces a boolean result. They are written in the form

```
;;(operator a b ...)
;;examples
(> a b)
(= a b)
```

Standard racket outputs #t and #f and these values can be used in cond statements and other functions that take in boolean values.

Example 3.2. You can use equal? to test the equality of two values which may not be of the same type.

Note. A predicate can also be user-defined and can use compound operators such as AND, OR, NOT.

Example 3.3. An example of a predicate that outputs if a number is between two numbers

```
(define (between? low high numb)
  (and (< low numb) (< numb high)))
```

3.5 Symbols

Definition 3.2. A **symbol** can be used to represent something. It is defined using a single apostrophe, ‘blue. To compare symbols, use symbol=?.

```
(define something ‘blue)
(symbol=? something ‘blue)
```

3.6 String manipulation

There are many string manipulation techniques in the manual that are not covered in the lectures. Consult the manual for more information.

```
(string? "Warbear") ;determines whether argument is string (#t)
(string-append a b) ;merges strings (ab)
(string-length x) ; yield the length of string x(1)
(string<? "Apple" "apple") -> #t
;;determines whether first string is lexicographically less than the
   second.
```

```
(string<? "apple" "Apple") -> #f ;;lower case comes after upper case, so
   a > A
(string<? "a" "b" "c") -> #t
```

Definition 3.3. Substring returns a new string and must be less than or equal to the length of the original string. The end value must be greater than or equal to start. Returns the substring from the start character to the end character, but not including the end character. The end character is an optional parameter. If none specified, assume end is last character.

Note. Strings start at 0 in computer science.

```
(substring "Semantics" 2 5) -> "man"
(substring "Semantics" 2) -> "mantics"
(substring "Syntax" 3 3) -> "" (empty string)
```

3.7 Conditions

Similar to if statements in other languages, and are eventuated from top to bottom. They can also be nested provided that the required parameters are filled.

```
(cond
  [boolean action]
  [boolean action]
  [else action])
```

3.8 Testing

Definition 3.4. Black-box testing is testing done when the content of the code is unknown or not required.

Definition 3.5. White-box testing is testing done when content of code is known and required.

4 Syntax and Semantics

4.1 Spelling rules

Identifier names are made up of letters, numbers, hyphens, and underscores. Must contain at least one non-number and no brackets or apostrophes. Symbols start with a single quote followed by a valid identifier name.

Definition 4.1. Syntax is the way that code is allowed to be written. Usually code will not compile without proper syntax.

Definition 4.2. Semantics is the meaning of what is said.

Definition 4.3. A valid programming statement must have exactly one meaning to avoid **ambiguity**

4.2 Substitution

Racket evaluates functions through substitution. Only one substitution can be made at a time.

Definition 4.4. A step-by-step reduction in accordance to the rules is called **tracing** a program.

Example 4.1. cond statements skip false statements and goes to the first true statement.

```
(cond [false exp] ...) -> (cond ...)
(cond [true exp]... -> exp
(cond [else exp]... -> exp
```

Example 4.2. and statements evaluate to false if one argument is false. True skips to the next argument, and a stand alone (and) must be reached for the statement to be true.

```
(and false...) -> false
(and true...) -> (and ...)
(and) -> true
```

Example 4.3. or statements are the opposite of and statements. Evaluates to true if one argument is true, and false skips to the next argument. A stand alone (or) is substituted with false.

```
(or true ...) -> true
(or false ...) -> (or...)
(or) -> false.
```

Constants are directly replaced by their actual value even if other constants are called. It is all done in one substitution step.

Definition 4.5. A **syntax error** occurs when a sentence cannot be interpreted using the programming language's grammar and convention.

Definition 4.6. A **run-time error** occurs when an expression cannot be reduced to a value, but the evaluation steps are still incomplete.

```
(cond [(> 3 4) x] ;no else statement, cond is false
```

5 Structure

5.1 Structure

Definition 5.1. Structures bundle data together, and are similar to objects and classes in other languages.

Posn

Posn is a predefined structure. Various functions are included when a structure is defined.

```
;;A Posn is a (make-posn x y)
(make-posn 1 1) ;;makes a posn object with the given arguments
(posn-x posnname) ;; returns x value of a posn
(posn-y posnname) ;; returns y value of a posn (can be extended when
   structure has more arguments)
```

Note. Since posn is a data structure, (make-posn 8 1) cannot be reduced any further and is considered a value.

Definition 5.2. The **scale** function scales the posn by a given factor.

```
(check-expect (scale (make-posn 6 9 2)) (make-posn 12 18))
```

Definition 5.3. Racket uses **dynamic typing** and one cannot enforce a parameter to be a specific value type.

```
(make-posn ‘shadow ‘hawker) ;valid, racket does not enforce contracts
```

Structure Definition

Structures can be defined using define-struct. Various functions are also created when the structure is defined.

```
(define-struct card (suit num))  ;A Card is a (make-card Sym Nat)
(make-card ‘Hearts 7)         ;makes a card
(card-num (make-card ‘Hearts 7)) ;returns num parameter of card (7)
(card? (make-card ‘Spades 10))  ;checks whether argument is a card
```

5.2 Stepping

Using code from the previous example,

```
(card-num (make-card ‘Hearts 7)) -> 7
(card? 5) -> false
```

Definition 5.4. A **template** is a function that consumes a structure and outputs every argument within the structure.

```
;; my-mp3info-fn: Mp3Info -> Any
(define (my-mp3info-fn info)
  (... (mp3info-performer info)...
      (mp3info-title info)...
      (mp3info-length info)...
      (mp3info-genre info)...))
```

Definition 5.5. **anyof** is helpful in the contract section if a function can return different data structure outputs based on the input.

```
;;divide: Num Num -> (anyof Num Boolean)
(define (divide x y)
  (cond [(= y 0) false]
        [else (/ x y)]))
```

6 Lists

6.1 List Definition

Definition 6.1. A **list** is a recursive structure defined in terms of a smaller list. A list includes one item in the list, and a sublist not including that item. It is very similar to a linked list in other CS languages. The data definition is:

```
;; A List is one of:
;; * empty
;; * (cons Any List)
```

The template is:

```
;; my-list-fn: (listof Any) -> Any
(define (my-list-fn list)
  (cond
    [(empty? list) ...]
    [else ... (first list) ...
    ... (my-list-fn (rest list))]))
```

Definition 6.2. The function **cons** is used to take an element and a list and produces a new, longer list.

```
(cons a b)
```

where a represents an item in the list and must be a value (Str, Num, Sym, etc), and b must be a list (empty is a list). Similar to posn, (cons a b) is in simplest form.

Example 6.1. A few examples of defining lists.

```
(define clst empty) ;;empty list ; Empty list
(define clst1 (cons ‘Heavy empty)) ; A list containing one item.
(define clst2 (cons ‘Light clst1)) ;A list containing two items, Light
   and Heavy
(define clst3 (cons ‘Light (cons ‘Heavy empty))) ;c2 and c3 are the same
```

6.2 Extracting from Lists

Definition 6.3. The **first** function returns the first element in a given list.

```
(first (cons a b)) --> a ;substitution process
;;Returns first element in the list (a).
```

Definition 6.4. The **rest** function returns the list of elements without the first.

```
(rest (cons a b)) -> b ;substitution process
;;Returns the list b
```

Note. With a very long list of items, first and rest can be used together to produce a chosen element.

```
(define somelist (cons a (cons b (cons c (cons d (cons e empty))))))
(first (rest (rest somelist))) ;produces c
```

Definition 6.5. The **empty** function returns true and false depending on whether a given list is empty. For substitution, #t or #f is substituted in one step.

```
(empty? (cons ‘Food empty)) ; #t if list empty, #f if not. (#f)
```

Definition 6.6. The **member** function returns true and false depending on whether a given element in in a given list.

```
(member? ‘Food (cons ‘Food empty)) ; (#t)
```

Definition 6.7. The **length** function returns the length of a given list.

```
(length los) ;returns length of los

;;Without the built-in function, this would be the method to count the
   length.
(define (count-concerts los)
  (cond [(empty? los) 0]
    [else (+ 1 (count-concerts (rest los)))]))
```

6.3 Recursion

Definition 6.8. A function is **recursive** if the body of a function requires the application of the same function.

Definition 6.9. In **structural recursion** all parameters to the recursive call are either unchanged or one step closer to a base case.

6.4 Design Recipe

The design recipe for lists will contain a self-referential data definition.

```
;; A List-Of-Symbols is one of:
;; * empty
;; * (cons Sym List-Of-Symbols)

;; my-los-fn: (listof Sym) -> Any
(define (my-los-fn los)
  (cond [(empty? los)... ]
        [else (... (first los)...(my-los-fn (rest los))...)]))
```

6.5 List Manipulation

Example 6.2. A function that negates every number in a list, and produces a new list.

```
;; (negate-list lon) negates every number in lon
;; negate-list: (listof Num) -> (listof Num)
;; examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
(cons -2 (cons 12 empty)))
(define (negate-list lon)
  (cond [(empty? lon) empty]
        [else (cons (- (first lon)) (negate-list (rest lon)))]))
```

Note. Sometimes, functions only make sense if it is given a nonempty list. An additional statement can be added to the requires section of the design recipe to indicate this.

6.6 Characters

Definition 6.10. A **string** is just a sequence of characters. Racket's notation for a character 'a' is #\a

Definition 6.11. The function **string->list** converts a given string into a list of characters.

```
(string->list "test")
;;The code below is the output of the above function.
(cons #\t (cons #\e (cons #\s (cons #\t empty))))
```

Example 6.3. An example of where this would be useful is counting characters in a string.

```
(define (vowel? c)
  (member c(string->list "aeiouAEIOU")))

(define (count-vowels-list loc)
  (cond [(empty? loc) 0]
        [(vowel? (first loc)) (+ 1 (count-vowels-list (rest loc)))]
        [else (count-vowels-list (rest loc))]))

(define (count-vowels str)
  (count-vowels-list (string->list str)))
```

Note. Lists and structures can be used together to create lists of structures which can make things pretty complicated.

7 Recursion

Example 7.1. An example of a countdown function using recursion.

```
(define (countdown n)
  (cond [(= n 0) ...]
        [else (cons n (countdown (sub1 n)))]))
```

If it is needed for the countdown to stop at a desired number, another parameter can be added. The parameter will "go along for the ride" and be called in every iteration.

Example 7.2. Using recursion, a list can be easily sorted. **insert** is a recursive function which takes in a number and a sorted list, and inserts the number into the list.

```
(define (sort lon)
  (cond [(empty? lon) empty]
      [else (insert (first lon) (sort(rest lon)))]))

(define (insert n slon)
  (cond[(empty? slon) (cons n empty)]
       [(<= n (first slon)) (cons n slon)]
       [else (cons (first slon) (insert n (rest slon)))]))
```

Definition 7.1. Lists can be abbreviated. However, lists have fixed sizes while cons can contain a list of any arbitrary size.

```
(cons exp1 (cons exp2 empty))
;;becomes
(list exp1 exp2).
```

Definition 7.2. The commands **(second my-list)** is equivalent to (first (rest my-list)). This pattern goes all the way up to **(eighth (my-list))**. Use these sparingly to improve readability.

Definition 7.3. Lists can be further abbreviated by **quoting**. For instance (cons ‘red (cons ‘blue empty)) can be written as ‘(red blue) with the apostrophe being outside the list. Similarly, numbers can use the quoted list format too because quoted numbers evaluate to numbers.

Definition 7.4 (AL)**.** An **association list** is a list that contains a list of (key, value) pairs. An example would be a **dictionary**.

```
;; my-al-fn: AL -> Any
(define (my-al-fn alst)
 (cond
  [(empty? alst) ...]
  [else ... (first (first alist))...
  ... (second (first alst))...
  ... (my-al-fn (rest alist))]))
```

Example 7.3. Example of AL:

```
;; (lookup-al k alst) produces the value corresponding to key k,
;; or false if k not present
;; lookup-al: Num AL -> (anyof Str false)
(define (lookup-al k alst)
  (cond [(empty? alst) false]
        [(equal? k (first (first alst))) (second (first alst))]
        [else (lookup-al k (rest alst))]))
```

7.1 Processing Two Lists

Example 7.4. Processing **one list** is simple because the other list will just "go along for a ride".

Example 7.5. Processing in **lockstep** requires that the lists are the same length and consumed at the same rate.

```
;; dot-product: (listof Num) (listof Num) -> Num
(define (dot-product lon1 lon2)
  (cond [(empty? lon1) 0]
        [else (+ (* (first lon1) (first lon2))
              (dot-product (rest lon1) (rest lon2)))]))
```

Example 7.6. Processing at **different rates** is more complicated and requires at least four conditions.

```
(define (merge lon1 lon2)
  (cond [(and (empty? lon1) (empty? lon2)) empty]
        [(and (empty? lon1) (cons? lon2)) lon2]
        [(and (cons? lon1) (empty? lon2)) lon1]
        [(and (cons? lon1) (cons? lon2))
        (cond [(< (first lon1) (first lon2))
              (cons (first lon1) (merge (rest lon1) lon2))]
              [else (cons (first lon2) (merge lon1 (rest lon2)))])]))
```

Note. In some cases, conditions can be combined if they produce the same output.

7.2 Types of Recursion

Definition 7.5. Structural recursion, all parameters to the recursive call are either unchanged or one step closer to a base case.

Definition 7.6. Structural recursion with an accumulator is when a parameter called the **accumulator** is passed down. An example would be finding the largest element in a list.

Definition 7.7. When the arguments in the recursive application are **generated** and the function does not follow an inductive approach (breaking out of a recursion by reaching a base case), then a function is following **generative recursion**. Parameters are freely calculated at each step.

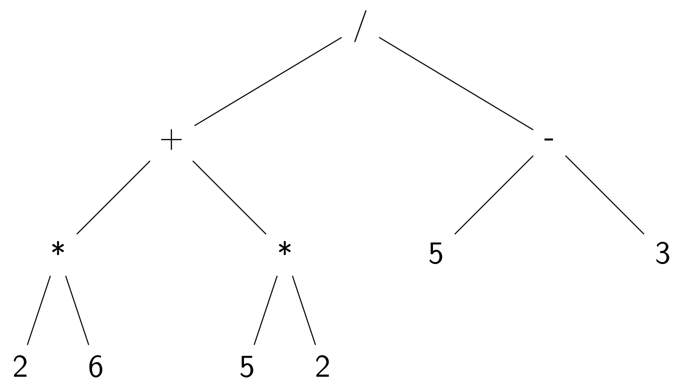
```
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

8 Trees

Definition 8.1. A **binary arithmetic expression** consists of numbers joined by binary operators.

Example 8.1.

$$((2 \times 6) + (5 \times 2))/(5 - 3)$$



8.1 Characteristics of Binary Trees

- Internal nodes have exactly two children
- Leaves have number labels while internal nodes have symbol labels
- Order of the children matter
- Structure of tree dependent on given expression

Example 8.2. Structure for Binary Trees (3rd line should be in comments):

```
(define-struct binode (op arg1 arg2))
;; A Binary arithmatic expression Internal Node (BINode)
;; is a (make-binode (anyof (+ - * /) BinExp BinExp)

;; A binary arithmatic expression (BinExp) is one of:
;; * a Num
;; * a BINode
```

Definition 8.2. Evaluation of binary trees requires the recognition of all the symbols

```
(define (eval ex)
  (cond [(number? ex) ex]
        [else (eval-binode (binode-op ex)
                   (eval (binode-arg1 ex))
                   (eval (binode-arg2 ex)))]))

(define (eval-binode op left right)
  (cond [(symbol=? op ‘-) (- left right)]
      [(symbol=? op ‘+) (+ left right)]
      [(symbol=? op ‘/) (/ left right)]
      [(symbol=? op ‘*) (* left right)]
      [else (error "Unrecognized operator")]))
```

8.2 Binary Search Trees

Definition 8.3. Binary search trees are structures that store data and makes searching for items more efficient. This requires that for each node, all the elements in its left subtree are less than it, and elements in the right subtree are greater than it. If a node has no children, it's left and right subtrees are denoted as empty.

```
(define-struct node (key val left right))
;; A Node is a (make-node Num Str BST BST)

;; A Binary Search Tree (BST) is one of:
;; * empty
;; * a Node
```

Aside. In a traditional list, to find an item, you go through each element one by one and checking equality, leading to a worst case scenario of $O(n)$. In a well-balanced binary search tree, every traversal cuts off half of the elements, leading to a time complexity of $O(\log n)$.

Example 8.3. Checking if an item is in a tree:

```
(define (in-tree x tree)
  (cond
    [(empty? tree) false] ;If tree is empty, false
    [(equal? x (node-value tree)) true] ;Element is at the node
    [else (or (in-tree x (node-left tree) (in-tree x (node-right
       tree))))])) ;; Checks both subtrees
```

Note. The above template can be modified if certain computations are required with the value.

9 Local Definitions

Definition 9.1. Local constants and variables may only be used within a local block. This is useful for larger programs where a variable name is not needed outside a certain area.

Note. When a variable name is both defined in global and local, the local version takes priority within the lexical scope. In CS 135, local functions do not require examples or tests.

```
;; Heron Formula
(define (t-area4 a b c)
  (local [(define s (/ (+ a b c) 2))]
         (sqrt (* s (- s a) (- s b) (- s c)))))
```

9.1 Semantics

When a local variable or function is defined, it is first promoted to the top level, and the new name is substituted everywhere the old name is used in the expression. In the next example, a random variable name **s_47** is generated.

```
(t-area4 3 4 5)
;;Function is called
(local [(define s (/ (+ 3 4 5) 2))]
        (sqrt (* s (- s 3) (- s 4) (- s 5))))
;;Local is moved outside
(define s_47 (/ (+ 3 4 5) 2))
(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5)))
;;New variable is first simpified
(define s_47 6)
(sqrt (* s_47 (- s_47 3) (- s_47 4) (- s_47 5)))
;;Variable is substituted in
(sqrt (* 6 (- 6 3) (- 6 4) (- 6 5)))
```

Example 9.1. Sometimes local can be used for clarity. The distance function may appear longer, but it becomes easier to understand.

```
(define (distance posn1 posn2)
  (sqrt (+ (sqr (- (posn-x posn1) (posn-x posn2)))
         (sqr (- (posn-y posn1) (posn-y posn2))))))
```

```
(define (distance posn1 posn2)
  (local [(define delta-x (- (posn-x posn1) (posn-x posn2)))
        (define delta-y (- (posn-y posn1) (posn-y posn2)))]
    (sqrt (+ (sqr delta-x) (sqr delta-y)))))
```

9.2 Function Abstraction

Definition 9.2. Abstraction is the process of finding similarities and forgetting unimportant differences. Advantages of functional abstraction include:

- Reduced code size
- Avoided cut-and-paste
- Bugs may be fixed on one place

Example 9.2. (+ x y) and (* x y) are similar if you ignore the function. Furthermore, they can be abstracted into one function that consumes a function and two numbers. Xs are used in the contract to signal that the parameters are of the same type.

```
;;foo: (X X -> X) Num Num -> Num
(define (foo f x y) (f x y))

(foo + 2 3) ; 5
(foo * 2 3) ; 6
```

Example 9.3. Similarly a function can produce a function that could be applied to other arguments.

```
(define add3 (local [(define (f m) (+ 3 m))] f))

(add3 2) ;;Produces 5
```

Definition 9.3. Lambda may be used to create a function. This is a much cleaner than using a local block.

```
(define add3 (lambda (x) (+ 3 x)))
```

9.3 Higher Order Functions

Definition 9.4. The Racket built-in function **filter** consumes a predicate with contract X -¿ Bool and a list, and produces a list of all the elements in the list that satisfy that condition.

```
;; filter: (X -> Bool) (listof X) -> (listof X)

(filter (lambda (x) (= (modulo x 2) 0)) '(1 2 3 4))
(filter even? '(1 2 3 4))
;; Both filters produce (list 2 4)
```

Definition 9.5. The Racket built-in function **map** consumes a function and a list, and applies the function to each element in the list, outputted in a new list.

```
;; map: (X -> Y) (listof X) -> (listof Y)

(map add1 '(6 9)) ;; produces (list 7 10)
```

Note. map may consume an arbitiary amount of lists, as long as all the parameters are dealt with either by lambda or by another function.

Definition 9.6. The Racket built-in function **foldr** consumes a two argument function, and applies the operation to the base case, last element, second last element, and so on until the first item of the list.

```
;;foldr
(foldr (lambda (x y) (+ x y)) 7 (1 2 3)) ;;produces 7 + 3 + 2 + 1 = 13
```

Example 9.4. All duplicates may be replaced from a list using foldr and no explicit recursion.

```
(define (dedup list)
  (foldr (lambda (element newlist)
         (cond [(member? element newlist) newlist] ; Already exists
               [else (cons element newlist)])) ; Unique element
         empty list))
```

10 Graphs

Definition 10.1. A **directed graph** consists of a collection of **nodes** together witha collection of **edges**.

Definition 10.2. An **edge** is an ordered pair of nodes, represented by an arrow from the first node to the second. Given an edge (v, w), w is an **out-neighbour** of v, and v is an **in-neighbour** of w.

Definition 10.3. A graph has a **cycle** if following all the edges results in an infinite loop.

Definition 10.4. Directed graphs without cycles are called **dirct acyclic graphs**.

```
;;A Node is a Sym
;; A Graph is a (listof (list Node (listof Node)))
```

Example 10.1. A useful operation for a graph includes finding its neighbours.

```
;; (neighbours v G) produces list of neighbours of v in G
;; neighbours: Node Graph -> (listof Node)
(define (neighbours v G)
  (cond [(empty? G) (error "vertex not in graph")]
        [(symbol=? v (first (first G))) (second (first G))]
        [else (neighbours v (rest G))]))
```

Example 10.2. To find a route between two points in a graph, mutual recursion may be used to generate a list of neighbours, and keep finding neighbours until either

```
;; (find-route/list los dest G) produces route from
;; an element of los to dest in G, if one exists
;; find-route/list: (listof Node) Node Graph -> (anyof (listof Node)
   false)

(define (find-route/list los dest G)
  (cond [(empty? los) false]
        [else (local [(define route (find-route (first los) dest G))]
              (cond [(false? route)
                     (find-route/list (rest los) dest G)]
                    [else route]))]))
```

```
;; (find-route orig dest G) finds route from orig to dest in G if it
   exists
;; find-route: Node Node Graph -> (anyof (listof Node) false)
(define (find-route orig dest G)
  (cond
    [(symbol=? orig dest) (list orig)]
    [else (local [(define nbrs (neighbours orig G))
                  (define route (find-route/list nbrs dest G))]
            (cond [(false? route) route]
                  [else (cons orig route)]))]))
```

Note. For **cyclic graphs**, this function will never terminate. To guarantee termination, create another parameter called **visited**, and ensure that each node has not been already visited.

```
;; find-route/list: (listof Node) Node Graph (listof Node)
;; -> (anyof (listof Node) false)
(define (find-route/list los dest G visited)
  (cond [(empty? los) false]
        [(member? (first los) visited)
        (find-route/list (rest los) dest G visited)]
        [else (local [(define route (find-route/acc (first los)
                                       dest G visited))]
              (cond [(false? route)
                    (find-route/list (rest los) dest G visited)]
                    [else route]))]))
```

The code for find-route/list does not change the accumulator, it only uses it. Adding the accomulator is done in the next code snippet.

```
;; find-route/acc: Node Node Graph (listof Node)
;; -> (anyof (listof Node) false)
(define (find-route/acc orig dest G visited)
  (cond [(symbol=? orig dest) (list orig)]
        [else (local [(define nbrs (neighbours orig G))
                      (define route (find-route/list nbrs dest G
                                       (cons orig visited)))]
              (cond [(false? route) route]
                    [else (cons orig route)]))]))
```

11 History

Example 11.1. Charles Babbage did meechanical computattion for military applications, but the computational operations were separate from the execution.

Example 11.2. Ada Augusta Byron assissted Babbage in explaining and promoting his ideas. She was possibly the first computer scientist.

Example 11.3. David Hilbert formalized the asiomatic treatment of Euclidean geometry. He believed that these three questions can be answered:

- Is mathematics complete? (For any formula ϕ, if ϕ is true, ϕ is provable.)
- Is mathematics consistent? (For any formula ϕ, thehre aren't proofs for both ϕ and $\neg\phi$.)
- Is there a procedure to, given a formula ϕ, produce a proof of ϕ, or show there isn't one?

Example 11.4. Kurt Godel proved that any non-trivla system of axioms (a system of axioms capable of describing integer arithmetic) is not complete. If it is consistent, it cannot be proved with the system. "The statement cannot be proved" would be inconsistent if the statement was false, because it would be a proof of a false statement, so it must be true but not provable.

Example 11.5. Alonzo Church and his student Kleene invented lambda calculus.

$$\lambda x, x + 2 \text{ is a function that adds 2 to its argument}$$

Example 11.6. Alan Turing defined a different model of computation and resulted in a simplier and influential proof.

Example 11.7. Grace Murray Hopper was the author of the first compiler defining an english-like data processin language, which later became COBOL.

Example 11.8. John Backus designed FORTRAN, which became the dominant language of numerical and scientific computation. He proposed a functional programming language, which led to the development of Lisp.

Example 11.9. John McCarthy was an AI researcher at MIT known for designing and implementing Lisp, based on ideas from lambda calculus and recursive function theory.

LISP eventually evolved into a general purpose programming language, and became the dominant language in AI due to encouragement of modifying the language itself. It ecame two main standards after the 1980s: Common Lisp anad Scheme.

Example 11.10. Gerald Sussman invented Scheme after an idea for a Lisp-like porgramming language had actors and lexical scoping added. He also wrote the textbook "Structures and INterpretation of Computer Programs", based on Scheme.

Example 11.11. The textbook "How to Design Programs" was written to remedy some issues with that book, namely the steep learning curve and lack of methodology.