CS 136: Elementary Algorithm Design and Data Abstraction

Official calendar entry: This course builds on the techniques and patterns learned in CS 135 while making the transition to use of an imperative language. It introduces the design and analysis of algorithms, the management of information, and the programming mechanisms and methodologies required in implementations. Topics discussed include iterative and recursive sorting algorithms; lists, stacks, queues, trees, and their application; abstract data types and their implementations.

Welcome to CS 136 (Spring 2013)

Instructors: Patrick Nicholson, Dave Tompkins, Olga Zorin

Web page: (main information source):

`http://www.student.cs.uwaterloo.ca/~cs136/`

Other course personnel: ISAs (Instructional Support Assistants), IAs (Instructional Apprentices), ISC (Instructional Support Coordinator): see website for details

Lectures: Tuesdays and Thursdays

Tutorials: Mondays

Be sure to explore the course website: *Lots* of useful info!

Textbooks:

- "C Programming: A Modern Approach" (CP:AMA) by K. N. King. **(Required)**
- "How to Design Programs" (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi (`http://www.htdp.org`)

Presentation handouts: available on web page

Advanced material and "asides" will appear in special boxes. This material will enhance your learning, but you will **not** be responsible for this material on exams.

Marking scheme

- 20% assignments (roughly weekly)
- 5% participation
- 25% midterm
- 50% final

You must pass both assignments and weighted exams in order to pass the course.

Class participation mark

- Encourage active learning and provides real-time feedback
- Based on use of “clickers” (purchase at Bookstore, register as part of Assignment 0)
- 2 marks for correct answer, 1 mark for wrong answer
- Best 75% over whole term used for 5% of final grade
- For each tutorial you attend, we’ll increase your participation mark (up to 1.2% overall, you cannot exceed 5%)

For a perfect participation mark, answer 75% of all clicker questions correctly, ***or*** *attend every lecture & tutorial and answer only 40% of all clicker questions correctly.*

Assignments

- Assignments are to be completed **individually**
- **Never share or discuss your code**
- Do not *discuss* assignment strategies with fellow students
- Each assignment may have different instructions and requirements: make sure you **read the instructions**

Getting Help

In addition to **office hours** and **tutorials**, we will be using an online discussion form (**piazza**).

Course announcements will be made on piazza and are considered mandatory reading.

Instructors, ISAs and your fellow students will monitor the forum and answer questions.

Post *clarification questions* to help understand assignments, but **do not** discuss assignment strategies, and **do not** post any of your assignment code *publicly*. You can post your code *privately*, and an ISA or Instructor *may* provide some assistance.

Languages

In CS 135 (and CS 115) the focus was on ***functional*** programming in Racket.

In this course, the focus is on ***imperative*** programming in C.

We will continue to use Racket to illustrate the similarities (and differences) between the two approaches.

This is not a "learn C" course. We will present language features and syntax only as needed. You can read the textbook (CP:AMA) for more comprehensive coverage.

What you learn in this course can be transferred to any language.

Software

We will be using our own customized "RunC" environment, which is integrated into the `gEdit` open source code editor.

RunC works with both C and Racket, provides verbose error messages, and helps to facilitate your testing.

Our testing environment (Marmoset) uses RunC to run your program. (If it does not work with RunC, it will not work with our tests).

See the website for RunC installation instructions.

Themes of the course

- Moving from functions to programs
- Modularization
- Abstraction
- Imperative programming & state
- Elementary data structures & algorithm design

Course Learning Goals

You should be comfortable programming in C and using imperative paradigms (mutation, iteration) effectively.

In Racket and C, you should be able to produce well-designed, properly-formatted, documented and tested programs of a moderate size (200 lines) that can use basic I/O.

You should understand the C memory model, including the explicit allocation and deallocation of memory.

You should understand and be able to apply the principles of modularization and abstraction.

You should be comfortable using elementary data structures (structures, arrays, lists and trees) and abstract data type collections (stacks, queues, sequences, sets, dictionaries).

You should have an introductory understanding of algorithms and time complexity.

Full Racket language

In this course, we will continue to use Racket, but we will be using the "full Racket" language (`#lang racket`), not one of the "teaching languages". There will be some minor differences, which we will highlight.

The first line of your Racket files should be `#lang racket`

In DrRacket, you should also set your language to:
"Determine language from source".

Even though you now have the full `#lang racket` available, you should not "go crazy" and start using every advanced function and language feature available to you.

For your assignments and exams you should stick to the language features discussed in class.

If you find a function that's "too good to be true", consult the course staff to see if you are allowed to use it.

Functions

Racket defines ***functions*** with the `define` special form, which *binds* the function body to the name.

In Racket, we ***apply*** a function, which ***consumes arguments*** and ***produces*** a ***value***. In the following example, `x` and `y` are ***parameters***.

Racket uses *prefix* notation (instead of *infix* notation).

$$f(x, y) = (x + y)^2$$

```
(define (f x y)
  (sqr (+ x y)))
```

Constants

You can define your own ***constants*** to make your code easier to read.

Constants also give you *flexibility* to make changes in the future.

You should avoid using “magic numbers” in your code.

```
(define ontario-hst .13) ; effective July 1, 2010

(define (add-tax price)
  (* price (add1 ontario-hst)))
```

Functions without parameters

In full Racket, we can have functions without any parameters. Even though they may "look" like a constant, they are still functions, and you should use parentheses.

```
(define magic-constant 7)

(define (magic-function) 42)

(define (use-magic x)
  (* x magic-constant (magic-function)))
```

Parameter-less functions might seem awkward now, but later we will see how they can be quite useful.

Running in Racket

A Racket program is a sequence of definitions and ***top-level expressions*** (expressions that are not inside of a definition).

When a program is "run", it starts at the top of the file *binding* each definition and *evaluating* each expression. Racket will also "display" the final value of each top-level expression.

```
#lang racket ; myfile.rkt

(define (f x) (sqr x)) ; function definition
(define c (f 3))       ; constant definition

; top-level expressions:
(+ 2 3)       ; => 5
(f (+ 1 1))   ; => 4
(f c)         ; => 81
```

Booleans

In full Racket, the constants `#t` and `#f` are used instead of `true` and `false`. Full Racket uses a wider interpretation of "true": **Any value** that is not `#f` is considered true.

Most computer languages consider *zero* (`0`) to be false, and any *non-zero* value is considered true. This is how C behaves.

Because Racket uses `#f`, it is one of the few languages where zero is considered true.

Logical operators and & or

The special forms `and` and `or` behave a little differently in full Racket:

`and` will produce: `#f` if any of the arguments are `#f`, `#t` if there are no arguments, otherwise the *last* non-false argument:

```
(and 5 6 7)   ; => 7
```

`or` will produce either `#f` or the *first* non-false argument:

```
(or #f #f 5 6 7) ;=> 5
```

Conditionals

```
(cond
  [q1 a1]
  [q2 a2]
  [else a3])
```

The `cond` special form produces the first "answer" for which the "question" is true (not `#f`). The questions are evaluated in order until a true question is encountered.

In full Racket, `cond` will not produce an error if all questions are false: it will produce `#<void>`, which we will discuss later.

If conditional

The `if` function can be used if there are only two possible answers:

```
(cond
  [q1 a1]
  [else a2])
```

is equivalent to:

```
(if q1 a1 a2)
```

Usually `cond` is preferred over `if` because it is more flexible and easier to follow, but we will use `if` because there is an equivalent in C.

Elementary data types

In addition to numbers and Booleans, Racket also supports the elementary data types `'symbols` and `"strings"`.

`'symbols` are used when a small, fixed number of labels are needed.

`"strings"` are used when the values are indeterminate, or when computation is needed (e.g. sorting).

Strings are composed of ***characters*** (e.g. `#\c #\h #\a #\r`).

Structures

In full Racket, the `struct` syntax is more compact:

- `define-struct` is now simply `struct`
- the `make-` prefix is omitted (`make-posn` is now simply `posn`)

```
(struct posn (x y)) ; defines posn, posn?, posn-x & posn-y
(define p (posn 3 4))
(posn-y p) ; => 4
```

For now, you should include `#:transparent` in your `struct` definitions (this will be explained later):

```
(struct posn (x y) #:transparent)
```

Lists

```
(define a1 (cons 1 (cons 2 (cons 3 empty))))
(define a2 (list 1 2 3))
(define a3 '(1 2 3))
```

You should be familiar with `cons`, `list`, `empty`, `first`, `rest`, `list-ref`, `length`, `append`, and `reverse`.

Two additional Racket list functions we will use later are `last` and `drop-right`.

Full Racket will not enforce that the second argument of `cons` is a list, so it will allow `(cons 1 2)`, but it's not a valid list, so don't do it!

member

The `(member v lst)` function behaves differently in full Racket. It will produce `#f` if `v` does not exist in `lst`. If if `v` does exist in `lst`, it will return the tail of `lst`, starting with the first occurrence `v`.

```
(member 2 (list 1 2 3 4)) ; => '(2 3 4)
```

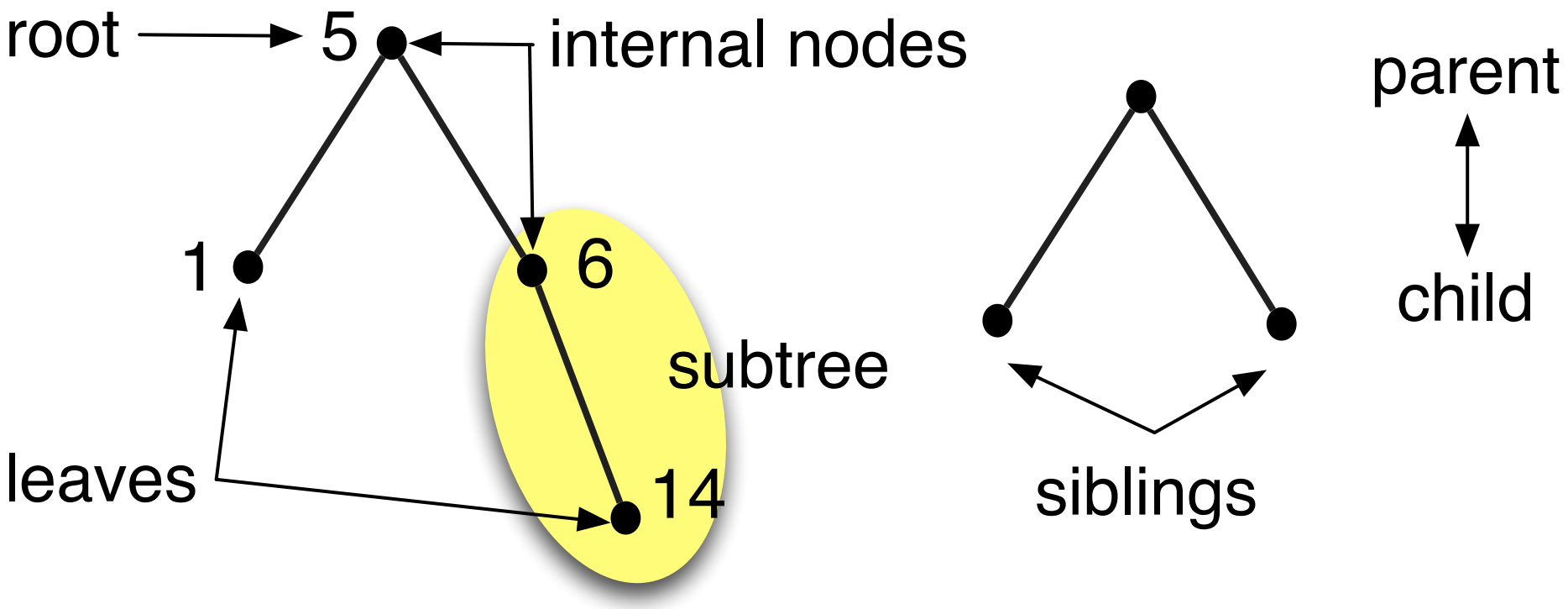
Recall, that `'(2 3 4)` is "true" (not false), so it will still behave the same in most contexts.

If you really want a predicate, you can define your own function:

```
(define (member? v lst) (not (false? (member v lst))))
```

Tree Terminology

A tree is defined recursively: a parent can have one or more children of the same type as the parent. Typically, data is stored in the nodes of the tree.



Binary Search Trees

```
(struct bst-node (key val left right))
```

The most useful tree we have seen is the Binary Search Tree (BST), where each node stores a `key` and a `value`. BSTs maintain the **ordering property**, where for any given node with key `k`:

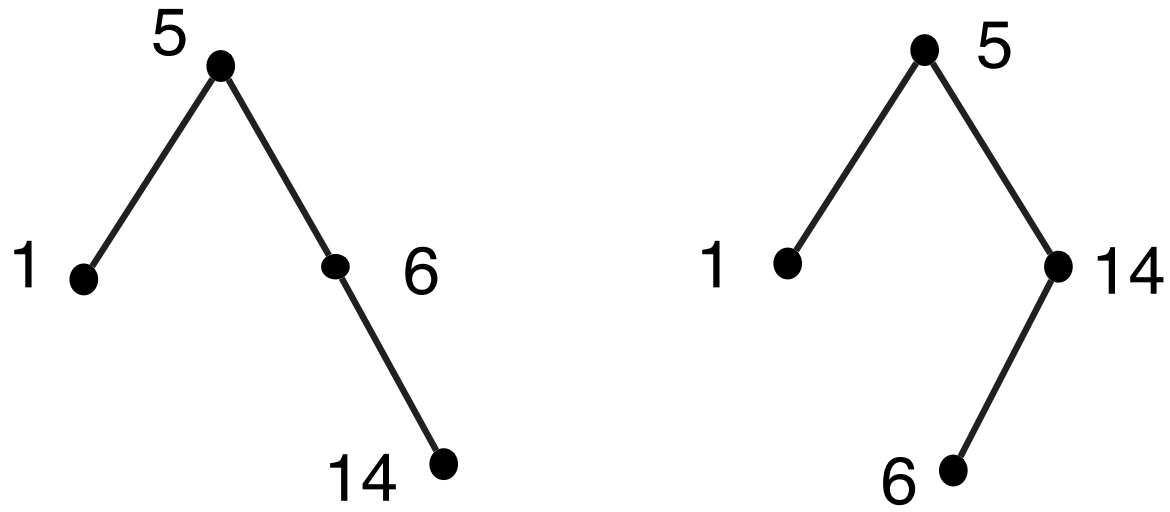
- Every `key` in the left subtree is less than `k`
- Every `key` in the right subtree is greater than `k`

We use the list function `empty` to represent an empty tree because it makes the code easy to read, but any ***sentinel value*** is fine. A popular alternative (used in the HtDP) is `#f`.

BST Example

There can be several possible BSTs holding the same set of keys:

(we often only show the keys in a BST diagram)




```
(define bst1 (bst-node 5 "" (bst-node 1 "" empty empty)
                 (bst-node 6 "" empty (bst-node 14 "" empty empty))))

(define bst2 (bst-node 5 "" (bst-node 1 "" empty empty)
                 (bst-node 14 "" (bst-node 6 "" empty empty) empty)))
```

BST Review

You should be comfortable inserting key/value pairs into a BST.

You should also be comfortable searching for a key in a BST.

You are *not* expected to know how to delete items from a BST.

Abstract list functions

In Racket, functions are also ***first-class*** values and can be provided as arguments to functions.

The built-in abstract list functions accept functions as parameters. You should be familiar with the abstract list functions `filter`, `map`, `foldr`, `foldl`, and `build-list`.

```
(define lst '(1 2 3 4 5))

(filter odd? lst) ; => '(1 3 5)
(map sqr lst)     ; => '(1 4 9 16 25)
(foldr + 5 lst)   ; => 20
```

Lambda

In Racket, `lambda` can be used generate an anonymous function when needed.

```
(define lst '(1 2 3 4 5))

(filter (lambda (x) (> x 2)) lst) ; => '(3 4 5)

(build-list 7 (lambda (x) (sqr x)))
  ; => '(0 1 4 9 16 25 36)
```

Implicit Local

In full Racket, you will not need to explicitly use the `local` special form, as there is an *implicit* ("built-in") `local` in every function body.

```
(define (t-area a b c)
  (local
    [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

Is equivalent to:

```
(define (t-area a b c)
   (define s (/ (+ a b c) 2))
   (sqrt (* s (- s a) (- s b) (- s c))))
```

The constant `s` is implicitly `local`.

Design Recipe

In CS 135 we learned the ***design recipe***, which included: contracts, purpose statements, examples, tests, templates and data definitions.

There were two main goals with the design recipe:

- To help you **design** new functions from scratch
- To aid **communication** by providing **documentation**

In this course, you should be comfortable designing functions, and so we will focus on communication and providing documentation.

Documentation

In this course, every function you write must have a **contract** and a **purpose** statement.

Examples are *encouraged* to aid communication.

We will address **testing** strategies later. You are expected to test your own code.

> The `check-expect` special form should not be used in this course. We will discuss alternatives for testing your code.

Extended contracts

In this course we extend our contract syntax by adding sections for ***preconditions (PRE)*** and ***postconditions (POST)***.

```
;; sum-first: Int -> Int
;;   PRE:  k >= 1
;;   POST: produce an Int >= 1
;; (sum-first k) produces the sum of the first k integers
;;   from 1..k

(define (sum-first k)
  ...)
```

We will often omit documentation in these course notes because of space limitations.

Pre- and post- conditions

The **preconditions** section lists the conditions that must be **true** *before* the function can be applied, including any *restrictions on the arguments*. If there are none, the preconditions are simply “true”.

The **postconditions** section lists any conditions that will be met *after* the function is applied, including a specification of what the function produces.

These sections **strengthen** the contract and more effectively communicate what the contract does:

“If you follow the contract and the preconditions are true, the postconditions will be met.”

More contract examples

```
;; lucky?: Num -> Boolean
;;   PRE:  true
;;   POST: produces #t if n is lucky, #f otherwise
;; (lucky? n) determines if n is a lucky number

(define (lucky? n)
  ...)

;; luckiest: (listof Num) -> Num
;;   PRE:  lon is not empty
;;   POST: produces an element of lon
;; (luckiest lon) finds the luckiest number in lon

(define (luckiest lon)
  ...)
```

Some languages have built-in features to *enforce* the preconditions and *verify* that the postconditions are true.

In later CS courses you may be introduced to a more formal syntax for specifying pre- and post- conditions.

Because our pre- and post- conditions are only comments, we will be informal.

Goals of this module

You should understand the following material from CS 135 (or CS 115/116): functions, constants, structures, lists, trees (including tree terminology and binary search trees), `lambda`, abstract list functions and scope.

You should understand the highlighted differences between the Racket teaching languages and `#lang racket`.

You should understand the documentation (design recipe) we will be requiring in this course, including the changes to the contract syntax (PRE & POST conditions).