Chris

CS135 Lecture Notes Fall 2017 Section 002

Prof Ian Goldberg Office Hour : Thursday 3:30pm DC

Lec 01:

- Web page(not LEARN, main info source):

<https://www.student.cs.uwaterloo.ca/~cs135/>

- Other course personnel:

- ISAs (Instructional Support Assistants)

- IAs (Instructional Apprentices)

- ISC (Instructional Support Coordinator)

- Racket is used - functional programming language

- Dr. Racket

- Assignments due Tuesdays at 9:00:00pm

- Computer Labs : MC3003, 3004, 3005, 3027, 2062, 2063.

- Textbook : How to design programs ( <http://www.htdp.org> )

- Course Note : available on Web or in paperback

- Marking Scheme : 20% assignments (roughly weekly), 30% midterm (10+20), 5% participation, 45% final

- > Both assignments and weighted exams should be passed. (>=50)

- Class Participation mark

- 1 pt for answering, 1 additional pt for answering correctly

- Best 75% answers are used for the 5% mark.

- CS135 Survival Guide

- Read ASAP

- Completing assignments is the key to success

- Don’t record anything from study groups.

- Policy 71 - Student Discipline(

- Penalty : 0pt for assignment, 5% Less on Final for first offence.

Suspension for second offence.

- Suggestions:

- Keep up/ahead with the readings

- Take notes ( Online notes or paper printed notes)

- Start assignments early.

- Get help early

- Follow advice on approaches to writing programs.

- Integrate exam study into weekly routine.

- Do extra questions

- Go over assignments and exams - Learn from my mistakes.

- Programming Language Design

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

- Ex: Excel formulas, LISP, ML, Haskell, Erlang, F#, XSLT, Clojure, etc.

- More closely connected to mathematics & Easier to design and reason

- Imperative: based on frequent changes to data

- Racket

- A dialect of Scheme

- Evaluate from left-to-right.

- Only evaluate from left-to-right fashion

- Only apply a function when all of its arguments are values.

- Infix operators cause ambiguity, so we treat them as functions

- 3-2 becomes -(3,2)

- (6-4)/(5+7) becomes /(-(6,4), +(5,7))

- For racket, the function name moves inside the parentheses, and the commas are changed to spaces

- g(1,3) becomes (g 1 3)

- (6-4)/(5+7) becomes (/ (- 6 4) (+ 5 7))

- 3-2+4/5 becomes (+ (- 3 2) (/ 4 5))

- (6-4)(3+2) becomes (\* (- 6 4) (+ 3 2))

\* Only use parentheses when necessary.

- Racket can handle any rational number exactly

- “#i” for inexact numbers.

Lec 02:

- DrRacket language level:

- Helps to eliminate the chance of accidentally using advanced features when they are not introduced.

- Trick for recognizing incomplete expression:

- Count the parens, start with 0, add 1 if open paren, subtract 1 if close paren. Then the expression must end with 0 at the end of the expression.

- Special form

- looks like a Racket function, but not all of its arguments are evaluated.

- define is a special form

- Evaluating a Racket expression

- Process of substitution, using the ‘yields’ symbol =>

- (\* (- 6 4) (+ 3 2)) => (\* 2 (+ 3 2)) => (\* 2 5) => 10

- Hormer’s Rule

- Each parameter name has meaning only within the body of its function, function-scoped.

- In (define p (\* k k)), the expression (\* k k) is first evaluated to give 9, then p is bound to that value.

- In Racket, constants can not be redefined and changed.

- In functional programming languages, variables are not necessary.

Lec03:

- The expressions are evaluated, using substitution, to produce values

- functions will not evaluate until all the arguments are values

- special forms, unlike functions, do not require the evaluation of all arguments before the expression is processed.

\* values are expressions

\* Definitions are not expressions, and vice versa

The relationship b/w definition & expression?

Not all expressions can be evaluated to produce values (/ 1 0)

Module 2

- It is as important to explain your program well to others as to write the program.

\* Important : Understand & write down what you want your program to do before actually coding it.

Design recipe

- A development process that produces a written explanation of the development

- Five parts of design recipe

1. Purpose: Describes what the function is going to do. (General, not specific steps)
2. Contract: Describe what type of arguments and what type of value the function consumes and produces
3. Examples: Illustrating the typical use of the function (using check-expect)
4. Definition of the function (header and body)
5. Tests: A number of trials that attempt to evaluate every part of the program.

- Order of execution

1. Write a draft of the Purpose
   1. Begins with an application of the function, then explain what it will do.
   2. Parameters should always appear in the purpose
2. Write Examples (by hand, then code)
   1. Use (check-expect n1 n2) to let Racket check if the actual answer matches the expected answer provided
3. Write Definition Header & Contract
   1. Name of the function: ParamTypes.... -> ResultType
4. Finalize the purpose with parameter names
5. Write Definition Body
6. Write Tests

- Types of parameters in Contracts

- Num: any Racket exact / rational number

- Int: restriction to integers

- Nat: restriction to natural numbers (including 0) (Integer >= 0)

- Any: any Racket value

- Tests

- Tests can handle complexities encountered while writing the body, ie the corner cases.

- It is not valid to pass in a value that violates what the contract specifies.

- Tests don’t need to be ‘big’, they should be small and direct.

- Work out the correct answers of tests by hand. Do not figure out by running the program.

- Testing methods

- (check-expect (sum-of-squares 3 4) 25)

- (check-within (sqrt 2) 1.414001)

- (check-error (/ 1 0) “/: division by zero”)

\* These testing methods are actually special because they are allowed to be executed before the function definition

- Additional contract requirements

- If there are important constraints about the parameters not specified in the contract, add an additional “requires” section in the contract section

- Boolean-valued functions

- For example, (= x y) determines if the statement “x=y” is true or false

- Other functions include (< x y), (<= x y), (>= x y)

- ;; =: Num Num -> Bool

**Lec 04:**

- **and**, **or** are special forms

- **not** is a function ; not: Bool -> Bool

- DrRacket only evaluates as many arguments of and and or as is necessary to determine the value. -> Reason why they are special forms.

- Short-circuiting

- Examples: ( These will never be divided by 0 )

- (and (not (= x 0) (<= (/ y x) c ))

- (or (= x 0) (> (/ y x) c))

\* (or p q) != (or q p)

- Predicate

- A function that produces a Boolean result.

- Symbol

- Defined using an apostrophe: i.e. ‘blue

- Allows a programmer to avoid using represent names of colours, or of planets, or of types of music

\* Acts like a constant string

- symbol=?: Sym Sym -> Bool

- String (Differences b/w strings and symbols)

- Strings are really compound data -> a sequence of characters.

- Symbols can’t have certain characters in them (such as spaces)

- More efficient to compare two symbols and two strings

- More built-in functions for strings

- (string-append Str Str): Str Str -> Str

- (string-length Str): Str -> Nat

- (string<? Str Str): Str Str -> Bool

- General equality testing:

- (equal? Any Any): Any Any -> Bool

- Do not use general (equal?)

- Conditional expressions

- Uses the special form cond

- General form of a conditional expression

(cond

[question1 answer1]

[question2 answer2]

...

[questionk answerk])

Where questionk could be else -> special form

- Short-circuiting also applies to cond

- Testing cond:

1. Test every conditional outcome.
2. Testing the boundaries.
3. DrRacket highlights unused code.

- **Black-box tests** are defined before the body of the function was written, not based on details of the code

- **White-box tests** depend on the code, for example, to check specific answers in conditional expressions.

**Lec 05:**

- Program is a sequence of definitions and expressions

- Three problems we need to address:

1. **Syntax**: The way we’re allowed to say things.
2. **Semantics**: the meaning of what we say
3. **Ambiguity**: valid sentences have exactly one meaning

- Grammar:

- <sentence> = <subject> <verb> <object>

- <def> = (define ( <var> <var> ... <var> ) <exp> ) for defining functions

- Semantic Model

- To express substitution rules, we use ellipsis.

- An ellipsis (...) is used in English to indicate an **omission**.

- In mathematics, an ellipsis is often used to indicate a **pattern**.

- Application of built-in functions:

- Turn the expression into value in **one step** directly.

- General substitution rule:

- (f v1 ... vn) => exp’

- where (define (f x1 ... xn) exp) occurs to the left, and exp’ is obtained by substituting into the expression exp, with all occurrences of the formal parameter xi replaced by the value vi (for i from 1 to n)

**Lec06**

- Function definition is always in its simplest form.

- Substitution rule for defining constants: evaluate all arguments until they are values, then the define is executed.

- Substitution rule for cond expressions

- (cond [false exp]...) => (cond ...)

- (cond [true exp] ...) => exp.

- (cond [else exp] ...) => exp.

- Errors: (cond [(> 3 4) x]) => (cond [false x]) => (cond) => Error: empty cond

- Substitution for and and or

- (and false ...) => false

- (and true ...) => (and ...)

- (and) => true

- (or true ...) => true

- (or false ...) =>(or ...)

- (or) => false

- Condensed trace : trace by skipping several steps in order to include the important info.

- Racket Structures (Take posn as example)

- Constructor : ;; make-posn: Num Num -> Posn

- Selector : ;; posn-x: Posn -> Num

;; posn-y: Posn -> Num

- A constructor with all parameters as values is itself a value.

**Lec07**

- Typing:

- “If you know what your data looks like, you know what your functions look like”

- Data definition for posn: “A Posn is a (make-posn Num Num)”

- Structure definition: “define-struct sname (fname1 ... fnamen)”

- Structure definition is always in simplest form

- Produces: make-sname, sname-fname1 ... sname-fnamen, sname?

- Substitution rule for ith selector:

- (sname-fnamei (make-sname v1 ... vi ... vn)) => vi.

- Another Example : mp3info

- (define-struct mp3info (performer title length genre) -> Structure definition

- ;; An Mp3Info is a (make-mp3info Str Str Num Sym) -> Data definition

- Template: Derived from a data definition

- Ex: ;; my-mp3info-fn: Mp3Info -> Any

(define (my-mp3info-fn info)

(... (mp3info-performer info) ...

(mp3info-title info) ...

(mp3info-length info) ...

(mp3info-genre info) ... ))

- Do this once per new structure type:

1. Data Analysis and Definition : Structure Definition + Data Definition
2. Template

- Another example: MmInfo (not a structure, it is a concept)

- (define-struct movieinfo (director title duration genre))

- Template for MmInfo

;; my-mminfo-fn: MmInfo -> Any

(define (my-mminfo-fn mminfo)

(cond [(mp3info? mminfo) template-for-mp3info]

[(movieinfo? mminfo) template-for-movieinfo]))

- (define (mminfo-artist info)

(cond [(mp3info? info) (mp3info-performer info)]

[(movieinfo? info) (movieinfo-director info)]))

- anyof types ( An alternative contract to the above approach )

- ;; mminfo-artist: (anyof MovieInfo Mp3Info) -> Str

**Lec 08**

Lists:

- Basic list constructs

- empty: A value representing a list with 0 items

- cons: Consumes an item and a list and produces a new, longer list.

- first: Consumes a nonempty list and produces the first item

- rest: Consumes a nonempty list and produces the same list without the first item

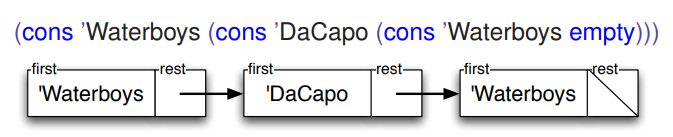
- empty?: Consumes a value and produces true if it is empty and false otherwise

- cons?: Consumes a value and produces true if it is a cons value and false otherwise.

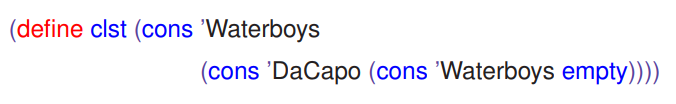
- A list is either empty, or consists of a first value followed by a list (the rest of the list)

;; A list is one of: empty (Base Case) | (cons value list) (Recursive case)

- Box-and-pointer visualization



- Example of list selectors : first | rest



- First concert: (first clst) => ‘Waterboys

- Concert after the first: (rest clst) => (cons ‘DaCapo (cons ‘Waterboys empty))

- Second concert: (first (rest clst)) => ‘DaCapo

- We will use (listof X) in contracts, where X may be replaced with any type.

- Exs: (listof Num), (listof Bool), (listof (anyof Num Sym)), (listof MP3Info), (listof Any)

- Stepping rules for list functions

- (cons a b) is a value iff a is a value | b is a list

- (first (cons a b)) => a

- (rest (cons a b)) => b

- (empty? empty) => true

- (empty? a) => false where a is not empty

- Template for lists

- Data definition: ;; A list of symbols, (listof Sym), is one of: empty | (cons Sym (listof Sym))

- Template:

- Contract: my-los-fn: (listof Sym) -> Any

- (define (my-los-fn los)

(cond [(empty? Los) ... ]

[else ( ... (first los) ...

(my-los-fn (rest los)) ... )]))

- Recursive list function test cases:

- 1+ test cases for base case

- Condensed trace: Shows the important steps and skips over the trivial details.

**Lec 09**

- Negate-list

(define (negate-list lon)

(cond [(empty? lon) empty]

[else (cons (- (first lon)) (negate-list (rest lon)))]))

- Nonempty lists

;; A NENumList is either:

;; \* (cons Num empty)

;; \* (cons Num NENumList)

- Template:

;; my-neln-fn: NENumList -> Any

(define (my-neln-fn neln)

(cond [(empty? (rest neln)) ( ... (first neln) ... )]

[else

(... (first nelst) ... )

(... (my-neln-fn (rest nelst)) ... )

- Character in racket : #\ + character

- ‘ ‘ <=> #\space

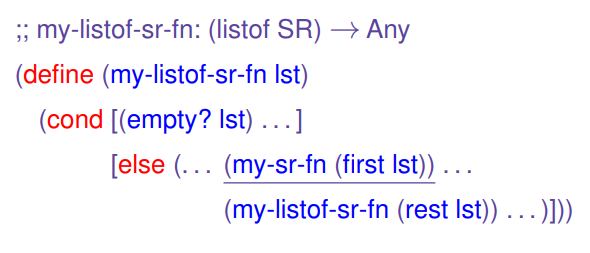
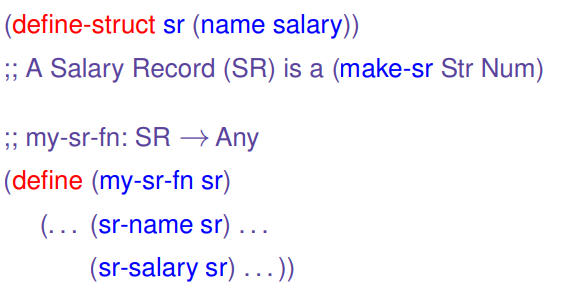
- ‘ \n ’ <=> #\newline

- Wrapper function

- Wrapper function calls the main function

- Ex: Main function requires a list of characters, use a wrapper function to act on (string->list str)

- List of structures



- Natural number recursive definition

;; A Nat is either:

;; \* 0

;; \* (add1 Nat)

**Lec 11:**

- Lists of lists

- Dictionaries: contains a number of **keys**, each with an associated **value**.

- Three main operations:

1. **lookup**: given key, produce corresponding value
2. **add**: add a (key,value) pair to the dictionary
3. **remove**: given key, remove it and associated value

- Association Lists

- A list of (key, value) pairs

**;; An association list (AL) is one of:**

**;; \* empty**

**;; \* (cons (list Num Str) AL)**

- Sentinel value: Different type from the normally produced type to indicate errors

- Do **add** and **remove** as HMWK

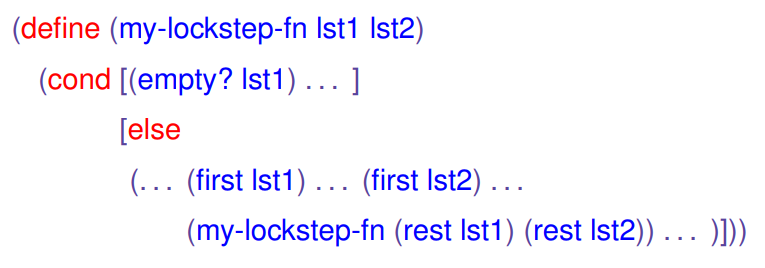
- Lists of lists can also represent a two-dimensional table.

- Processing two lists simultaneously

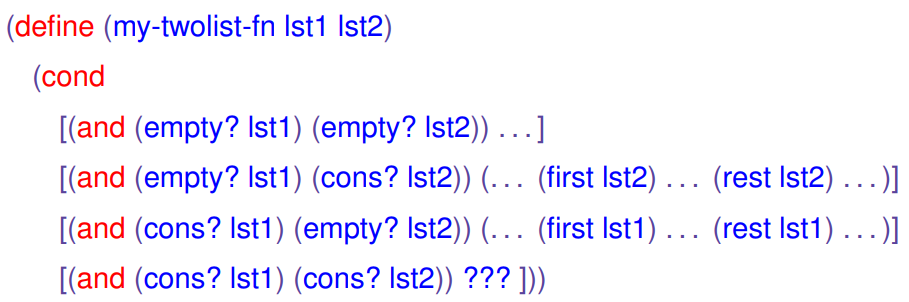
- Case1: processing just one list

- Case2: processing in lockstep

- lst1, lst2 must be the same length, and are consumed at the same rate.



- Case3:



**Lec 12**

- Non-structural recursion

- Accumulative recursion

- Computationally, we can pass down the largest value see n so far as a parameter called an **accumulator**.

- The accumulator accumulates the result of prior computation.

- Generative recursion

- Parameters are freely calculated at each step.

- Easier to get wrong, harder to debug, harder to reason about.

**Lec 13**

- BINode: ;; **(define-struct binode (op arg1 arg2))**

**;; A BINode is a (make-binode (anyof ‘\* ‘+ ‘/ ‘-**

- BinExp: **;; A BinExp is one of**

**;; \* a Num**

**;; \* a BINode**

- BINode and BinExp refers to each other

- This is called **mutual recursive.**

**- The base case of a tree is always the leaves**

- **;; my-binexp-fn : BinExp -> Any**

**(define (my-binexp-fn binexp)**

**(cond [(number? binexp) ...]**

**[else**

**( .... (binode-op binexp) ...**

**... (my-binexp-fn (binode-arg1 binexp)) ...**

**... (my-binexp-fn (binode-arg2 binexp)) ... )**

- **Depth of node** : The number of edges from the node to the root node

- **Evolution tree** :

- Data definition : **;; A Taxon is one of:**

**;; \* a Modern**

**;; \* an Ancient**

**(define-struct modern (name pop))**

**;; A Modern is a (make-modern Str Nat)**

**(define-struct ancient (name age left right))**

**;; An Ancient is a (make-ancient Str Num Taxon Taxon)**

- Template : **;; my-taxon-fn: Taxon -> Any**

**(define (my-taxon-fn t)**

**(cond [(modern? t) (my-modern-fn t)]**

**[(ancient? t) (my-ancient-fn t)]))**

**;; my-ancient-fn: Ancient -> Any**

**(define (my-ancient-fn t)**

**( ... (ancient-name t) ...**

**(ancient-age t) ...**

**(my-taxon-fn (ancient-left t)) ...**

**(my-taxon-fn (ancient-right t)) ...))**

- Binary Tree

- Template: **;; my-bt-fn: BT -> Any**

**;; (define (my-bt-fn tree)**

**(cond [(empty? tree) ... ]**

**[else**

**... (node-key tree) ...**

**... (node-val tree) ...**

**... (my-bt-fn (node-left tree)) ...**

**... (my-bt-fn (node-right tree)) ...))**

**Lec 13**

- Binary Search Trees (BST)

- Data definition: **;; A Binary Search Tree (BST) is one of:**

**;; \* empty**

**;; \* Node**

**(define-struct node (key val left right)**

**;; A Node is a (make-node Num Str BST BST)**

**;; requires: key > every key in left BST**

**;; key < every key in right BST**

- Key is greater than every key in left

- Key is less than every key in right

- Reduces the problem size by half after each computation.

- O(logn), worse case O(N).

- General Trees

- Arbitrary number of children

- Data definition: (define-struct node (key val (listof Node)))

**TUT 06**

- Recognize pure structural, accumulative and generative recursion

- If not one step to the base case, then its not pure structural

- For Nat3, (- n 3) is one step closer

- For Nat, (sub1 n) is one step closer

- If not pure structural, then look for accumulator

- If at the base case, the function produces not the sentinel value, but a value as an accumulation of previous result, then it’s accumulative

- If not either of these, then it’s generative

**Lec 14**

- Midterm review session: AL 116 Wed Nov|7-9 pm

- Nested lists

- Data definition: **;; A Nest-List-X is one of:**

**;; \* empty**

**;; \* (cons X Nest-List-X)**

**;; \* (cons Nest-List-X Nest-List-X)**

- (cons ‘(1 2) ‘(3 4)) => ‘((1 2) 3 4)

- (list ‘(1 2) ‘(3 4)) => ‘((1 2) (3 4))

- (append ‘(1 2) ‘(3 4)) => ‘(1 2 3 4)

- **Chapter 9: Local definitions and lexical scope**

- local: contains a series of local definitions plus an expression using them

- (local [(define x1 exp1) ... (define xn expn)] bodyexp)

- A define within a local expression may shadow a name which has already been defined to another value.

\* - Substitution rules of local:

- To get fresh constant names, we will define unused names and promote the definition to the top of the file.

- Subpart1: Change the name of local definitions to fresh names

- Subpart2: Promote all local definitions to the top of the file

- Subpart3: Remove the local special form, and sub it with the expression

- Evaluate constant definition

- Do normal substitution

**Lec 15**

**Functional Abstraction**

- Ex: Abstract function for apple and odd

(define (my-filter pred? lst)

(cond [(empty? lst) empty]

[(pred? (first lst))

(cons (first lst) (my-filter pred? (rest lst)))]

[else (my-filter pred? (rest lst))]))

**\* Function are values ( !IMPORTANT )**

**- Can be consumed as function arguments**

**- Can be produced as function results**

**- Can be bound to identifiers**

**- Can be put in structures and lists**

- Type of function is just its contract

- Say for my-filter:

;; my-filter: (Int->Bool) (listof Int) -> (listof Int)

- To generalize the contract for my-filter (Parametric types)

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

**Lec 16 The most important day!**

- Anonymous functions: has no name.

- Steps to simplify eat-apples

1. Use local to define helper functions since not-symbol-apple? is used only once

2. Only put the predicate not-symbol-apple? in the local, not the whole filter function

3. Lambda (Most important slide, Module 10 Slide 37)

\* (local [(define (name-used-once x1 ... xn) exp)]

name-used-once)

\* Can be rewritten as:

\* (lambda (x1 ... xn) exp)

\* Can be thought of as “make-function”

- Eat-apple turns into:

(define (eat-apples lst)

(filter (lambda (item) (not (symbol=? item ‘apple))) lst))

- Function definition is the same as constant definition

- (define (f x y) (\* x y) is the same as...

- (define f (lambda (x y) (\* x y)))

- Stepping rule:

- If the function is defined in the old way, do not change it

- If the function is defined by lambda expression, then first substitute the function name with lambda.

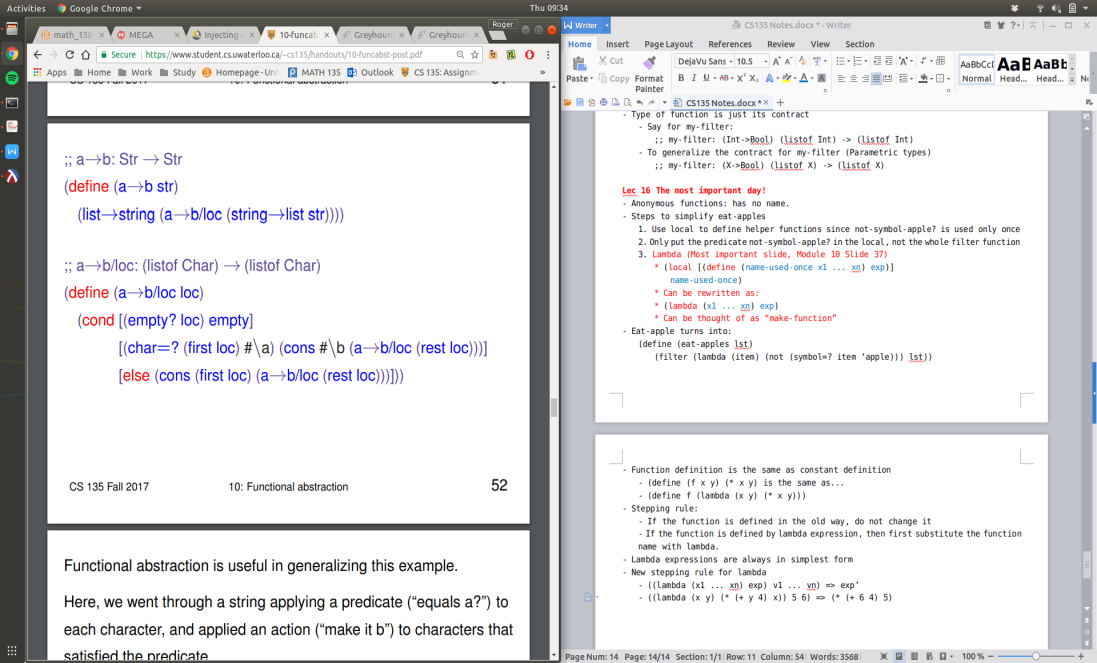
- Lambda expressions are always in simplest form

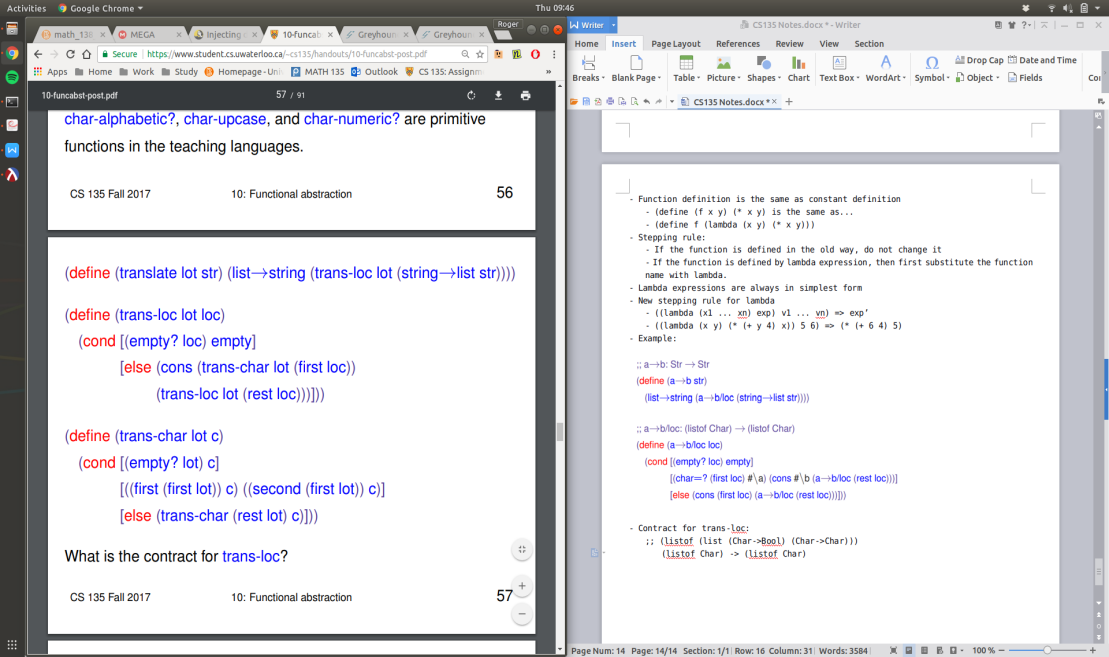
- New stepping rule for lambda

- ((lambda (x1 ... xn) exp) v1 ... vn) => exp’

- ((lambda (x y) (\* (+ y 4) x)) 5 6) => (\* (+ 6 4) 5)

- Example:





- Contract for trans-loc:

;; (listof (list (Char->Bool) (Char->Char)))

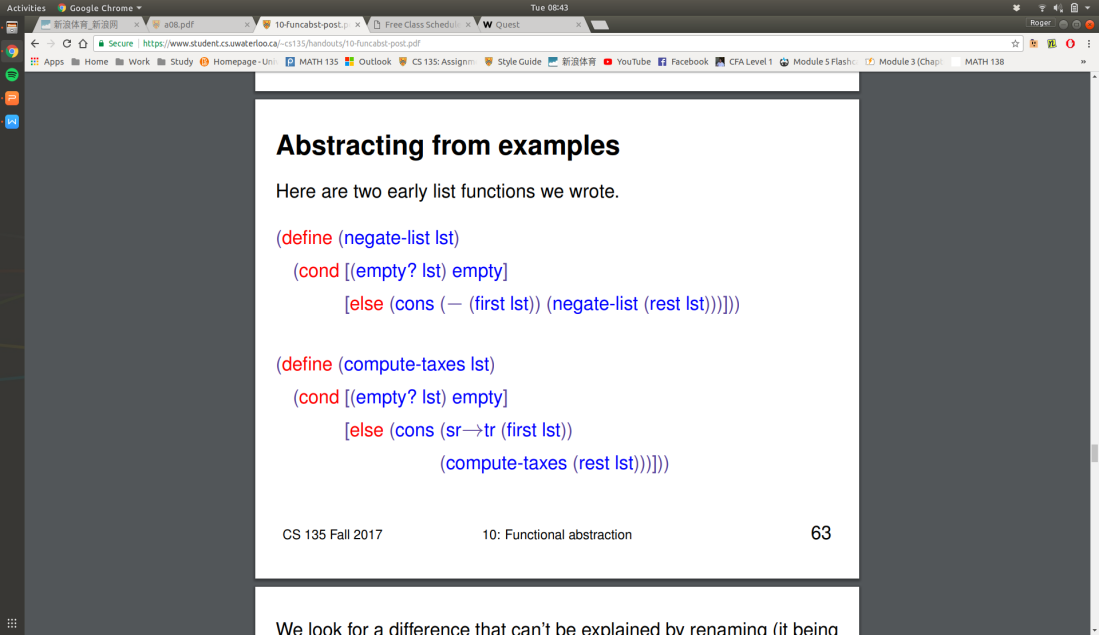
(listof Char) -> (listof Char)

- Parametric types:

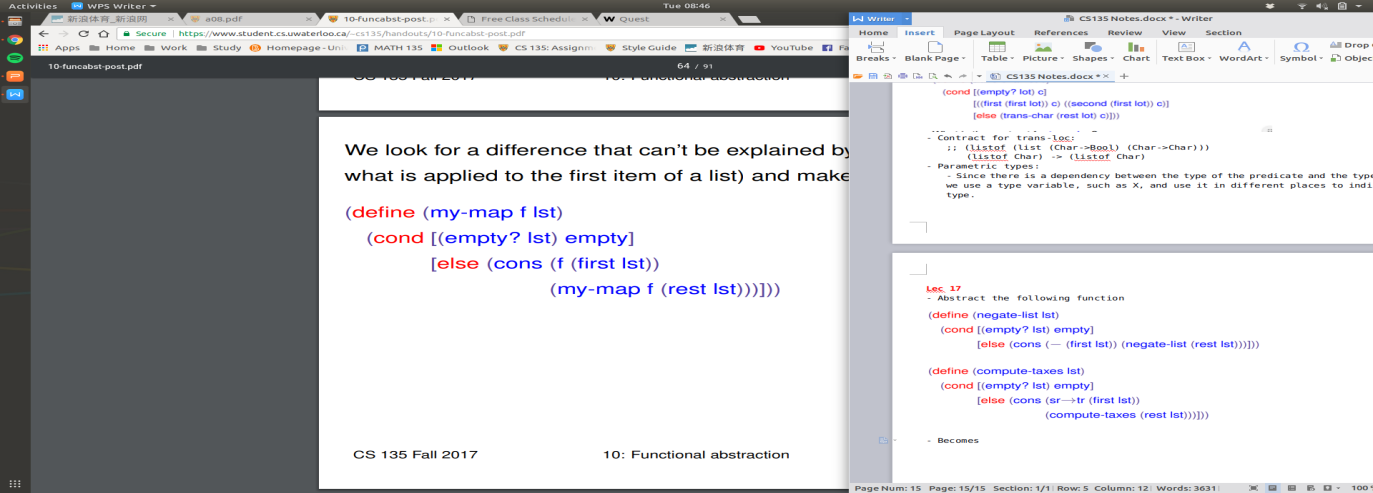
- Since there is a dependency between the type of the predicate and the type of list, we use a type variable, such as X, and use it in different places to indicate same type.

**Lec 17**

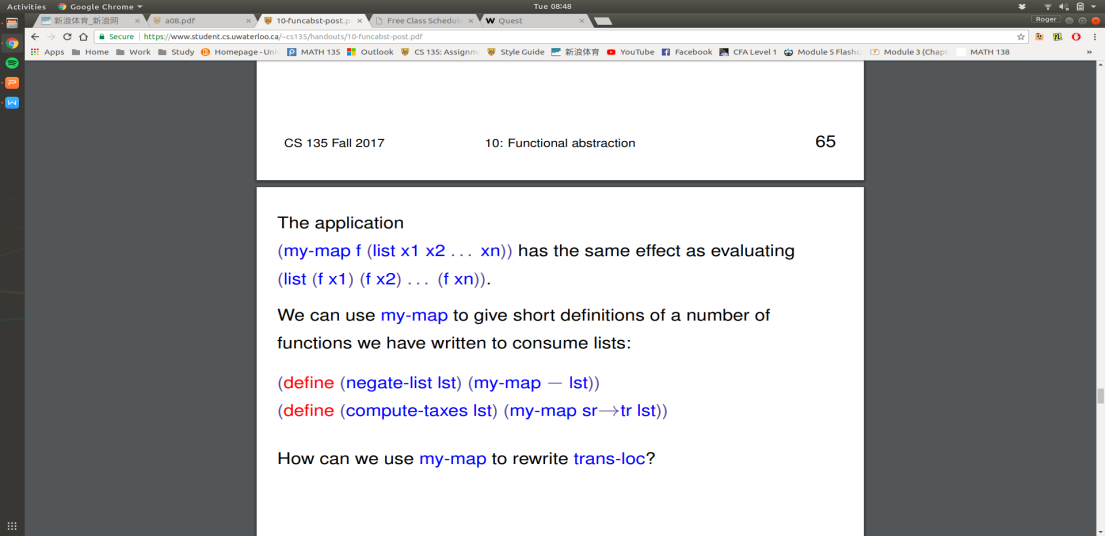
- Abstract the following function



- Becomes ...



- Application of my-map



- Translation of trans-loc using my-map

(define (trans-loc lot loc)

(my-map

(lambda (c)

(trans-char lot c))

loc)))

- Contract for my-map

;; (X -> Y) (listof X) -> (listof Y)

- Built-in Map and Filter

- Map => Applys function to each element of the list

- Filter => Keeps all elements in the list that is in conformity with the predicate.

- **my-foldr**

(define (my-foldr combine base lst)

(cond [(empty? lst) base]

[else (combine (first lst)

(my-foldr combine base (rest lst))]))

- (define (sum-of-numbers lst)

(my-foldr + 0 lst))

- (define (prod-of-numbers lst)

(my-foldr \* 1 lst))

- (define (count-symbols lst)

(my-foldr (lambda (f r) (add1 r)) 0 lst))

**\* Abstract list functions <=> Template with ... (ellipsis) filled in**

- (define (my-filter pred? lst)

(foldr (lambda (f r)

(cond [(pred? f) (cons f r)]

[else r]))

empty lst))

- (define (my-map fn lst)

(foldr (lambda (f r)

(cons (fn f) r))

empty lst))

- **Contract for my-foldr**

;; my-foldr: (X Y -> Y) Y (listof X) -> Y

- **Max-list**

;; max-list: (listof Num) -> (anyof Num false)

(define (max-list lst)

(foldr (lambda (f r)

(cond [(false? r) f]

[else (max f r)]))

false lst))

- Do (lambda (f r) ...) for the combine argument of my-foldr !

;; build-list: Nat (Nat -> X) ->

=================================================================================

- Graph Terminology

- Given an edge (v,w), we say that w is an **out-neighbour** of v, and v is an **in-neighbour** of w

- DAG: Directed Acyclic Graph

List => Tree => DAG => Graph

- Template for graphs

;; my-graph-fn: Graph -> Any

(define (my-graph-fn graph)

(cond [(empty? Graph) ...]

[else ... (first (first graph)) ...

... (my-list-node-fn (second (first graph))) ...

... (my-graph-fn (rest graph)) ... ]))

- For games:

- Board configuration is a node

- Neighbours computes the next possible configurations