CS4221 Foundations Of Computer Science Handouts Semester 1 (2018)

Introduction

This document contains handouts for the CS4221 lectures. Please note that these do not replace your own notes, rather they are to facilitate your own note taking by reducing the amount of writing you need to do during lectures.

To get the maximum benefit from these handouts you are recommended to take lecture notes and to reference the relevant page number in this document. In some cases there is space left in this document for you to add your own notes.

Table of Contents

1. Introductory Material

There are four things that you should achieve before the lecture on Friday of Week 7:

- Buy the book of handouts (which you've already apparently done!);
- Register for the course on Moodle;
- Get the notes from Lecture 1;
- Download the free Socrative App (we will use this on Friday).

Note: This set of notes is not the same as the set from last year's CS4221. If you have a classmate who has second hand set, feel free to lord your more comprehensive set over them.

1.1. Registering for CS4221 on Moodle

We will use Moodle throughout the semester. You should already have an account on it, but, if not, go to:

http://moodle2.csis.ul.ie/

Once you are set up, follow these instructions to register for CS4221 – anything you see in *italics* should be typed **exactly** as it looks:

- Click on **all courses** down on the bottom left of the screen
- Type *CS4221* into the search box
- Click on the module name
- In the space for **Location key** type in the module enrolment key, which is (and include the exclamation mark): *eaMM124!*

That's it! You're now registered with CS4221 Moodle.

1.2. Notes from Lecture 1

The notes are available from the first section when you log into Moodle.

1.3. Download the Socrative App

We will use *Socrative* throughout the semester, starting on Friday of Week 7. Full details are available on Moodle, but the link to download Socrative is http://socrative.com. An account has already been set up for you; use your ID number to login.

2. Expressions

- “Mathematical phrase”

- Slope of a line (X1, Y1) and (X2, Y2)

- Evaluating expressions
    - All numbers?
    - Some (or all) variables?
        - E.g. Y2-Y1, what are Y2 and Y1?
- Order of evaluation:
    - 1 + 2 * 3 = ?
        - 3 * 3 = 9
        - 1 + 6 =7
- If operators are different?
    - Give each a precedence
- Standard precedence:
    - ( )
    - *, /
    - +, -
    - Draw? Go left to right

- Note, all of these operators are associative, makes no difference

- Different kinds of notation:
    - So far, have used (mainly) infix notation
    - i.e. operators come between operands
- Other notations:
    - Prefix
        - Before operand
        - $\sqrt{4}$ -3, cos 45 etc.
    - Super-fix
        - Above (and usually after) operand
        - 3^2, x^y
    - Sub-fix
        - Underneath
        - $\log_x Y$, ⅔
- Postfix
    - After operand
        - 3++
- Why so many?
- Consider

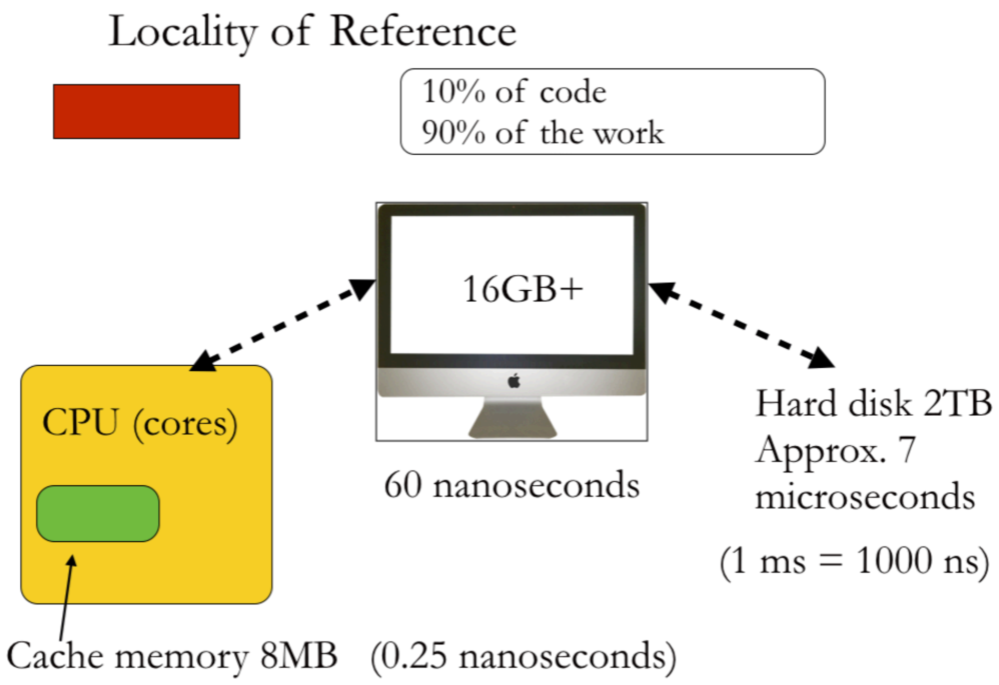
    - All four
- Notice
    - Size of square root sign
    - Length of line over 2a

    - No multiplication sign
    - ac right up together
- Reminder - what is this course about?
    - Phrasing things unambiguously
    - Start where?
    - Mix of notation
- Model Driven Development
    - Design
- Imperative Programming
    - Learn language
- Computer Organisation
    - Build hardware
- Computer Science
    - Step back and SOLVE the problem

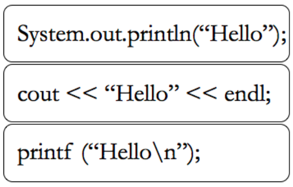
- Problems?
    - Square root sign
    - Can’t be typed
        - Variable length
        - Variable scope
            - Area in which something takes effect
    - Aside: X√ would be more convenient
    - (b^2-4ac)√
    - Still pretty ugly...
- (b^2-4ac)
- Read b
    - Meaning?
    - Not clear until after squared sign
- $(b^2-$
    - Ambiguity
        - Subtract next thing?

        - Evaluate next sub-expression?
- Better if there was a single notation
    - No ambiguity
    - Everything evaluated the same way
- Separate the “what” from the “how”.
    - Make no comment on how to do operation
        - e.g. how to add numbers
    - Worry about implementation later
- Prefix problem
    - Don’t know how to deal with a character (or number)..
    - Until after (at least) the next one is read

2.1. Locality of reference



- Writing fast programs
    - Small (fit in the cache) “Memory Footprint”
    - Reuse functionality (stay in the cache)
    - Often faster to do one thing many times than several things once
- Prefix Notation
    - Operator goes before operands
    - (+ 2 3)
    - “Apply plus operator to 2 and 3”
    - “Apply operator to next two items”
    - “.. to the next two arguments”
- Definitions
    - Syntax
        - *Representation of data/code*
    - Semantics
        - *Meaning of syntax*
    - Abstract Syntax
        - *Representation that is* ***independent*** *of language*



```
System.out.println(“Hello”);
```

```
cout << “Hello” << endl;
```

```
printf (“Hello\n”);
```

- Design will work with any language
    - After a translation process
- Design once, deploy many times
- Abstract Syntax Tree (AST)
    - Diagram of expression
    - Shows **what** expression does.
    - (+ 2 3)

- AST
    - Convenient graphical notation
- Evaluation
    - Evaluate deepest operator
    - Repeat until no operators are left

- Nothing on level 1
- (+ 3 3) = 6
- More complex tree

- Order?
- Infix?
    - (1 * 2) + (3 * 4)
    - 1 * 2 + 3 * 4
    - (+ (* 1 2) (* 3 4))

    -
- Evaluate (+ (* 1 2) (* 3 4))
    - Read +
        - Means?
            - Get first argument
            - Get second argument – Add them
    - First argument?
        - Another expression, evaluate it first

    - Read *
        - Means?
        - Get first argument
        - Get second argument – Multiply them
- What next?

- Draw AST from prefix notation
    - First item (always an operator) in ( ) is a parent
        - Second is left child
        - Third is right child
- Notes
    - A child can be the parent of another child
    - i.e. the start of another sub-tree
    - Children often called arguments, rather than operands

- Evaluating prefix?
    - Evaluate most deeply nested first
- (+ (* (+ 2 1) 3) 4)

- (+ (* 3 3) 4)

2.2. Parsing Expressions

- Parse expression:
    - Read +
    - **Evaluate** (*..
    - Read *
    - **Evaluate** (+..
    - Read +
    - Read 2
    - Read 1
    - Add them

- Question
    - If ASTs are representation independent, can any notation or representation be converted to one?
    - Fortunately for us, yes.
    - Convert infix to AST
        - First item on left
        - Second becomes parent
        - Third on right

Take your own notes.

- How to write Sin X in infix?
  - Can’t -- must be prefix.
  - Infix often contains other representations
- What have we achieved?
  - Language independent representation for expressions (ASTs)
  - Prefix notation
    - Machine independent
    - Machine readable
    - Consistent

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

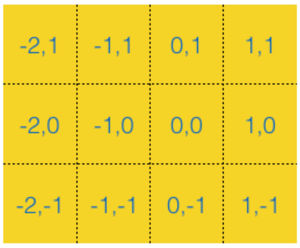
- Rewrite as prefix…
  - ± … not an operator
  - use two different expressions
- b^2
  - (sqr b)
  - Is this fair?
    - Consistent with prefix
    - Unary argument

- Square root

  - Number of arguments? One
- Treat same as sqr
  - (sqrt x)
  - (sqrt (- a b))

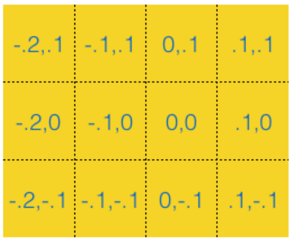
Take your own notes.

- Conway's Game of Life
  - Less than two neighbours the cell dies of loneliness
  - Two or three neighbours, the cell stays alive
  - More than three neighbours and the cell dies from overcrowding
  - Dead cell with three neighbours becomes alive
  - https://bitstorm.org/gameoflife/
- **Fractals**
- One simple rule (to colour each pixel)
  - $Z_{n+1} = Z_n^2+c$
  - c = Co-ordinates (as Complex number)
  - Z = Complex number (start with 0+0i)
- If Z goes to zero, pixel is coloured black
- If it goes to infinity, colour is the number of iterations



-2,1	-1,1	0,1	1,1
-2,0	-1,0	0,0	1,0
-2,-1	-1,-1	0,-1	1,-1

- http://hirnsohle.de/test/fractalLab/
- Fractals and self-similarity
  - Fractals are infinitely zoomable
  - Zoom in and the co-ordinates get more **precise**
  - Repeat the process again



-.2,.1	-.1,.1	0,.1	.1,.1
-.2,0	-.1,0	0,0	.1,0
-.2,-.1	-.1,-.1	0,-.1	.1,-.1

- Are they really infinitely zoomable?
  - Yes, but the hardware doesn't have infinite precision
  - 64 bit architecture goes from $\pm 2.23\times10^{-308}$ to $\pm 1.80\times10^{308}$

2.3. Converting infix to prefix

- We need a simple algorithm to convert any infix expression to corresponding prefix one

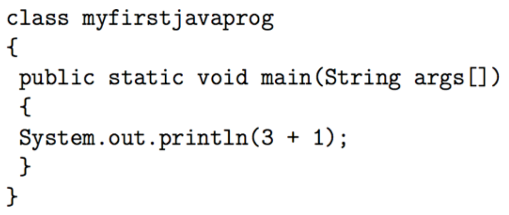
- Stacks
- *Data Structure*
    - Way of organising data in computer
- Operations
    - Add item to the data
    - Look at item
    - Remove item

Take your own notes. There will be a video of the animation from the lecture put on the class website.

- Back to infix to prefix conversion
    - Reverse the expression
    - Read expression one character at a time:
    - ")": Push onto stack
    - Operator: Push onto stack
    - Operand: Push on and pop off (straight to output)
    - "(": Keep popping stack until ")" is encountered
    - Reverse the output
- *Take your own notes.*

3. Design

- Writing programs for CS4221
    - Java?
    - Too inconsistent
        - Mix of pre/post/infix (“mixfix”)
            - i++; ++i; i+i;
            - i=i+++++i;
    - Lots of extra syntax
        - public static void main(String[] args)..
    - “Syntactic Sugar”
        - Doesn’t enhance functionality
        - (Allegedly) makes program easier to read
    - Is there any place for Imperative Programming?
        - Yes, it’s just not as useful as you might believe...
- Functional Programming
- Write programs based on expressions/functions



```
class myfirstjavaprog
{
 public static void main(String args[])
 {
 System.out.println(3 + 1);
 }
}
```

- > (+ 3 1)
- Very little syntax
    - Short programs
    - Mobile Devices
    - Embedded controllers
- Relevance?
    - (Almost) anything can be expressed with an AST.
    - Good FP language copies AST with little overhead
    - Racket
        - Dialect (subset) of *Lisp*

        - http://racket-lang.org/
    - Imperative languages (Java/C++) must be compiled
    - Racket
        - Compilable
        - Interpretable

- (+ 1 2 3)
- 6
- Not all operators are binary
    - Yet another reason not to use infix
- How many operands do operators take?

- Depends..
    - Only one (sqr, sqrt)
    - Only two (divide)
    - One or more (most)
- (- 1)..
- (- 3 2 1)
- (- 5 1 1)

- Take *tail* of list of numbers from *head*
- (- 8 7 6..)

3.1. Car and Cdr

- Heads and tails
    - car = first item in list
    - cdr = rest of the list
    - (car (a b c d))
    - a
    - (cdr (a b c d))
    - (b c d)
    - (car (cdr (a b c d))
    - (car (b c d))
    - b
- Expressions vs. Lists
    - (+ 1 2) vs. ‘(+ 1 2)

    - (car ‘(a b c d))

    - (cdr ‘(a b c d)

    - (reverse ‘(a b c))

    - (list ‘a ‘b)
- Examples of ‘

```
> (+ 2 1)


> ‘(+ 2 1)


> ‘b


> b
Error: reference to undefined identifier: b
```

- (- 10 3 2 1)

- Algorithm
  - Get car and cdr
  - Subtract first item in tail from the head
  - (subtract car of the cdr from the car)
  - car: 10; cdr: (3 2 1);car of cdr: 3.
  - set head equal to answer, i.e. 7
  - Repeat while there is something in the tail

Expression	car	cdr	car(cdr)
(-10 3 2 1)			

- Divide
  - (/ 20 5 4 2)
  - Divide head by each item in the tail
  - (/ 4 4 2)
  - (/ 1 2)
  - ½
  - Note! Not 0.5.
- car and cdr for Racket...

```
> (+ 1 2)
>3
> '(+ 1 2)
> (+ 1 2)
> (car '(a b c d))
> a
  > (cdr '(a b c d))
  > (b c d)
  > (cdr (car '(a b c d))
```

  - **Error**
  - cdr: expects argument of type <pair>; given a
- car returns an ITEM

- cdr expects a LIST

  > (car '(+ 1 2))

  > +

  > (car (+ 1 2))

  cdr: expects argument of type <pair>;

  given 3

  > (car (a b c d))
- Error: a undefined

- Implications for ASTs?
- Smaller

- In Racket
  - (function arguments)
- Implication
  - prefix notation ≣ AST ≣ Racket
- So far
  - All operators are built in (+, -, * etc.)
  - All arguments/operands are numbers
  - AST gives the same answer
- Dynamic?
  - Behaviour depends on user input
  - Works the same way
  - e.g. Add 1 to X
  - Where does X come from?
  - Difference between
  - Add 1 to X
  - Given a value for X, add 1 to it.

3.2. Lambda Calculus

(λ X. + 1 X)

Take your own notes.

- Needs a value for X to do anything
- Argument names are ALWAYS one letter in λ calculus
- In Racket?
    - (lambda (x) (+ 1 x))
- Function
    - “Method” in Java
    - Usually takes one or more arguments
- Two different kinds of expressions:
    - “Reducible expression” (redex)
        - can be made simpler
        - e.g. (+ 2 1)=>3
    - Expression or λ calculus
        - can’t be simplified (yet)
        - (λx. + 1 x)
    - How to convert an expression into a redex?
    - Give it a value for variables/parameters
- Argument gets passed to parameter
- Each variable in body corresponding to parameter gets replaced

- Redexes
    - ((λ y. * 1 y) 2)
    - (* 1 2) => 2
    - ((λ xy. + x y) 2 3)
    - Take **in order**
        - x=2, y=3
    - ((λ y. + 2 y) 3)

    - (+ 2 3)
- In Racket
    - ((lambda (x y) (+ x y)) 2 3)
- Is ((λxy. * x y)2) a redex?

- ((λx. x) 2)

- What if the argument is a redex?
- ((λx. +3 x)(+ 2 2))
- Evaluate argument first (usually):
    - ((λx. +3 x) 4)
    - (+ 3 4) => 7
- Bring in "as is"
    - (+ 3 (+ 2 2))

- In general (for the moment…)
    - Evaluate the **innermost** redex first
    - i.e. the most deeply nested and furthest to the **right**
        - (λx. x)(+ 1 1)
    - Implications for ASTs?
        - Need a new node: "application" (@)
        - i.e. apply λ expression to one or more arguments.

Take your own notes.

- Racket
    - > (lambda(x) (+ 1 x))
    - #<procedure>
    - Meaning?
        - Expression it can't reduce

```
> ( (lambda(x) (+ 1 x)) (read))
> ( (lambda(x) (+ 1 x)) 5)
```

6

- Variables and Types
    - Racket uses Type Inference
        - Guesses at what the type should be e.g. expects an integer
        - Why? “+ 1”
            - However, (λ x. x) can take anything
- “Operator overloading”
    - Operator behaves the same regardles of type
        - e.g. (+ x y) adds two things
    - Doesn’t matter what type they are.
    - Another view of +:
        - ((λxy. + x y)A B)

- Too long winded to write λ expression each time
    - Not every operator is built in...
    - sqr: (λx. * x x)
    - but, (sqr x) is more useful
- This is called a “function”
    - Difference between this and supplied operators?
    - None.
    - (define sqr (lambda (x) (* x x)))
    - > (sqr 3)
    - (define add (lambda (x y) (+ x y)))
    - > (add 2 1)
    - 3
- > (define x 3)
- > x
- 3
- > (+ x 3)
- 6

- > x
- 3
- > (define x (+ x 3))
- > x
- 6
- > (define cube (lambda (x) (* x (sqr x))))
- Strictly speaking, λ calculus doesn't name functions
- **How does Racket know it's not a fn?**
- Type inference
    - No lambda part
- Functions are λ expressions with names
- λ expressions are expressions with params

	Functional	**Imperative**
# funs	Large	Small
Size funs	Small	Large
Params?	Yes	Yes
Return	Always	Usually

- Which is better?
    - Easier to debug if functions are "tightly coupled"
        - i.e. all instructions in a function are related.
    - Much easier to manipulate functions in functional language….
    - Possible to program imperative programs in a functional style
- Aim of this course?
    - Figure out what problem is
    - Break up problem (into functions)
    - Code up problem (in either Racket or Java)
- Scope
    - where something takes effect
    - scope depends on length of line
    - variables have scope
    - (λ x. x)
    - (λ x. y)

    - > (define x 2)
    - > (define add1 (lambda (y) (+ 1 y)))
    - > x
    - 2
    - > y
    - reference to undefined identifier: y

3.3. Scope

- Scope
    - where something takes effect
    - scope depends on length of line
    - variables have scope
    - (λ x. x)
    - (λ x. y)
    - > (define x 2)
    - > (define add1 (lambda (y) (+ 1 y)))
    - > x
    - 2
    - > y
    - reference to undefined identifier: y

```
> (add1 x)
3
> x
2
```

add1 returns value of 1 + arg

Doesn’t modify argument.

```
>y
Error: undefined variable
```

y only exists while fun is running, disappears after that

- Details
    - (define add1 (lambda (y) (+ 1 y)))
    - (add1 x)
    - add1 is called
    - variable y is created
    - argument (contents of x) copied in
    - body executed
    - y is thrown away
    - Limited “lifetime”
    - Why does x still exist?
    - Defined in its own right

Take your own notes.

- Reentrant Code
    - Code that has no side effects
    - Virtually every example we will see in this module
    - Can be stopped, restarted, executed multiple times

    ```
    > (define sqr  (lambda (x) (* x x)))
    ```

    - Copies parameter into local storage, so no side effects
    - Efficient parallel code is almost always reentrant
    - Reentrant code is more likely to fit in CPU cache
    - No side effects means other code isn’t impacted by it
- Map

    ```
    > (define sqr  (lambda (x) (* x x)))
    > (define nums ‘(1 2 3))
    > (map sqr nums)
    > (1 4 9)
    ```

    Execute function as many times as needed

    ```
    > (pmapf sqr nums)
    ```

- Graphical Processing Units (GPUs)
    - Thousands of cores
    - Little or no communication between them

  - Image data spread across cores
  - Each core updates its part of the image

4. Local and Global Variables

- Examples have been relatively straightforward
  - e.g. Only dealing with single function, no global variables
  - assuming single processing core
- From lecture 1:
  - *How do I design a program that can't be tested?*
- Necessary Tools
- Hundreds of functions
- Millions of copies of the **same** function running at the same time
- Functions taking other functions as parameters
- We need the ability to keep all these moving parts straight
- Local and Global Variables

```
> (define x 2)
> (define addx (lambda (y) (+ y x)))
```

  - y is local, x is global

```
> (addx 4)
6
> x
2
> (define x (addx 4))
> x
6
```

- Local and global variables in λ calculus
- Some shorthand...
  - λx. E

  - Function that takes one argument
  - Don't care what function does
  - λx. (E F)
  - Same as above, but two distinct parts to function
- Examples:
  - λx. + x y ≡ λx. E
  - E = + x y
- λx. + x y ≡ λx. E F
  - E = + x, F = y
  - OR, E = +, F = x y
- λx. x ≡ λx. E
- E = x
- But, λx. x <> λx. E F
- Terminology:
  - Global variable ≈ free variable
    - y is free in (λx. + x y)
  - Local variable ≡ bound variable
    - x is bound in (λx. + x y)

- **Free variables**
- X is free in (E F) if X is free in E or in F.
- e.g from above, is y free in (+ x y)?
  - E = + x, F = y.
  - Not in E, but is in F.
  - It DOES occur free in (E F).
- Notice:
  - E = +, F = x y.
  - Not in E, but is in F.
  - It DOES occur free in (E F).
- **Bound variables**
- X is bound in (E F) if X is bound in E or in F.
- x occurs bound in (λ*y*. E) if
  - *x*=*y* AND *x* occurs free in E

  - OR, *x* is bound in E.
- Examples:
  - Is x bound in (λx. + x y)?
  - x is in the parameter list (λx) and it does appear free in (+x y)
  - Thus, x **is** bound in (λx. + x y).
  - **x** is a local variable in (λx. + **x** y).
- Is y bound in (λx. + x y)?
  - It doesn't occur in parameter list
  - Other possibility? Bound in E?
    - E = + x y
  - Occurs free in E, so is NOT bound.
- More Examples:
  - e1: + x 3 x is free in e1
  - e2: (+ x) 3 (consider e2 = E F)
    - Free in E = (+ x), NOT free in F = 3
    - Therefore, free in e2
- Is y bound in (λx. + (λy. + 3 y) x 2)?
  - It doesn't occur in parameter list
  - Other possibility? Bound in E?
    - E = (+ (λy. + 3 y) x 2)
    - It does appear in the parameter list
    - It **is** free in the body (+ 3 y)
      - Therefore, it is bound in E.
  - So, yes, y is bound in (λx. + (λy. + 3 y) x 2)
- Nested function
- ((λx. + (λy. + 3 y) x 2) 7)
- (+ (λy. + 3 y) 7 2)
- (+ (+ 3 7) 2)
- Different order of evaluation
- ((λx. + (λy. + 3 y) x 2) 7)
- ((λx. + (+ 3 x) 2) 7)
- (+ (+ 3 7) 2)

- (λx. + x 3) x
- x free or bound?
- Bound in E, free in F
- They are two different x's. The same name does **not** always mean same variable.

- (define x 2)
- ((λf. f (* 1 3) (+ 4 5) x) ((λxyz. (* 2 x)))
- Which x is free and which is bound?
- ((λf. f (* 1 3) (+ 4 5) x) ((λxyz. (* 2 x)))
    - x is free and x is bound.

4.1. β reduction

- Naming variables
- Variables with the same name are not necessarily the same variable
- Does not imply that two variables are the same
- To avoid confusion better to keep different names
    - > (define x 3)
    - > (define add1 (lambda (x) (+ x 1)))
    - > (add1 x)
    - 4
    - > x
    - 3
- **Formally in λ calculus**
- β reduction means passing arguments to a lambda.
- Remove the λ and parameters list (e.g. λxy.) and in the *resulting* body, replace the free variables with the arguments.

  - (λx. + x 1) 4
  - The body without λx. is (+ x 1)
  - In the *absence* of λx. part, x is free in (+ x 1)
  - Replace x with 4
  - => (+ 4 1) (redex)
  - =5
  - (λx. + x 1) 4 $\overrightarrow{\beta}$ (+ 4 1) = 5
- Nested functions
- ASTs/Prefix expressions can have multiple levels of nesting
  - E.g. (λx. * (+ 2 3) (- 2 (* 3 4))) 5
- But also:
  - (λx. + (λy. + 2 y) x 4) 3
  - Beta reduction replaces **free** occurrences in the body.
  - [x is free, *after* removing the (λx) part].
  - x=3 $\overrightarrow{\beta}$ (+ (λy. + 2 y) 3 4)
  - y=3 $\overrightarrow{\beta}$ (+ (+ 2 3) 4)
  - = 9
- A more confusing but *identical* example:
  - (λx. + (λx. + 2 x) x 4) 3
- Replace only FREE occurrences, after removing λx.
  - x=3 $\overrightarrow{\beta}$ (+ (λx. + 2 x) 3 4)
- Note x is not replaced, because it is *still* bound (to lambda starting with λx).
  - x=3 $\overrightarrow{\beta}$ (+ (+ 2 3) 4)
- =9

- Passing lambdas as arguments
- (λf. f 3) (λx. + x 1)
- Argument is a function (lambda)
  - β reduction replaces free occurrences of f
  - So we get:
    - (λx. + x 1) 3

    - Another β reduction follows:
        - + 3 1 = 4
- Passing lambdas as arguments
- Is this a strange thing to do?
    - No, it is an ENORMOUSLY powerful thing in programming
    - Usually modify functionality by passing **data**
    - Can modify functionality by passing **code**
    - Parallel processors are often programmed in this way
    - Programmer can efficiently move code between processors
- Extremely difficult to do in imperative programming
- Simple to do in functional programming

Take your own notes.

4.2. α-conversion and δ-conversion

- Remember: Functions in λ calculus and ASTs (usually) don't have names
- Racket can use them
    - Useful for reusing functions
    - Useful for debugging
    - Slightly more longwinded
- > (define t (lambda (f) (f 3)))
    - > (t   (lambda (x) (+ x 1))   )
    - 4
- > (define t2 (lambda (f) (f 2 3)))
    - > (t2 +)
    - 5
    - > (t2 7)
    - Error: attempt to call a non-procedure
- Lesson?
    - Anything can be passed as a parameter: numbers, variables, functions, operators

  - Syntax the same in lambda calculus, AST and Racket
  - Not consistent in imperative programming
    - Very different when passing a function to a function
- Formal Notation for β reduction:
  - (λx. E) a $\overset{\rightarrow}{\beta}$ E[a/x]
  - Meaning: in E, replace free occurrences of x with a
- Consider: (λx. + x 1) and (λy. + y 1)
  - Are they the same?
  - Yes - names don't matter.
  - Converting one into another: **<u>α-conversion</u>**

    E.g. (λx. + x 1) $\overset{\leftrightarrow}{\alpha}$ (λy. + y 1)
  - **<u>Note:</u>** bi-directional arrow: two way process
- However, if we replace x with y in:
  - (λx. + x y)
  - We get: (λy. + y y)
  - Not correct. Why?
    - Because y is free in (λx. + x y)
  - What about:
    - (λx. + x (λy. + y 1) 2) $\overset{\leftrightarrow}{\alpha}$ (λy. + y (λy. + y 1) 2)
    - This is fine
    - y is NOT free in the body of the lambda on left side.
  - Formal Definition:
    - λx. E $\overset{\leftrightarrow}{\alpha}$ λy. E[y/x], **IF** y does not already exist free in **E**.
- Utility of α-conversion
- (λf. (λx. f (f x))) x
- β-reduction=>(λx. x (x x))
- Erroneous.

- Use α-conversion to avoid confusion:
  - convert x into y inside the nested lambda.
  - (λf. (λy. f (f y))) x
  - β-reduction=> (λy. x (x y))
  - Correct

- δ-conversion and Normal Form
    - (λx. (+ x 1)) 2

      $\vec{\beta}$ (+ 2 1)
    - $\vec{\delta}$ 3
- (F a1 a2) $\vec{\delta}$ result, where F is a built in operator
- β-reduction puts values in, δ-conversion evaluates them
- The result after full evaluation is said to be in **Normal form**
    - E.g. (+2 1) = 3 is in Normal form
    - No more redexes left.

Example of δ-conversion

- β-reduction – an interesting example
- (λf. (λx. f 4 x)) (λyx. + x y) 3
- (λf. (λx. f 4 x)) (λyx. + x y) 3
- $\vec{\beta}$ (λx. (λyx. + x y) 4 x) 3
- $\vec{\beta}$ (λyx. + x y) 4 3
- $\vec{\beta}$ (+ 3 4)
- $\vec{\delta}$ 7
- Racket Code
- (define Lf (lambda (f) (lambda (x) (f 4 x))))
- (define Lyx (lambda (y x) (+ x y)))
- ((Lf Lyx) 3)

`

When to evaluate arguments – the effect.

- Consider function
- D: (λx. x x)
- In Racket: (define D (lambda(x) (x x)))
- Evaluate D D

- (λx. x x) (λx. x x)
- $\vec{\beta}$ (λx. x x) (λx. x x)
- $\vec{\beta}$ (λx. x x) (λx. x x)
- Infinite calls
- Try it in Racket using (D D)

- Consider (λx. 3) 7
- $\vec{\beta}$ 3
- Result is 3; no matter what the argument is.
- Evaluating the argument is needless.
- Consider (λx. 3) (D D)
- Evaluate the argument first? Infinite calls.
- Otherwise, the answer is just 3.

4.3. Order of evaluation

- How do we evaluate simple expressions?
  - So far “innermost”
  - e.g. (+ (* 2 3) 4)
- Applicative Order (Eager Evaluation):
  - “leftmost innermost”.
  - i.e. try to evaluate the leftmost redex;
  - Immediately go to the innermost level of nesting
  - (λxy. + x y) (+ 1 2) (+ 3 4)
  - =(λxy. + x y) 3 (+ 3 4)
  - =(λxy. + x y) 3 7
- Normal Order (Lazy Evaluation):
- Back to (λx. 3) (D D):
  - Applicative Order forces evaluation of (D D) even though it is **not** needed
  - Arguments are evaluated EXACTLY ONCE
- Another Strategy: Normal Order
  - Reduce “leftmost outermost”. i.e. work with the outermost bracket level whenever possible.

  - (λx. + x 1) (+ 2 3)
  - $\vec{\beta}$ (+ (+ 2 3) 1)
  - Cannot work at the outermost level now. So reduce the inner (nested) redex.
  - =(+ 5 1) = 6
- + is a "strict" function:
  - Requires all its arguments before proceeding further
  - Forces evaluation of arguments even in lazy evaluation
- (λx. 3) (D D) with Normal Order
  - 3
  - (D D) not evaluated

- Implications
- Applicative Order *can* cause infinite calls, and evaluate arguments needlessly
- It evaluates arguments **exactly** once
  - regardless of whether or not they are needed
- Normal Order only evaluates arguments when necessary
- It evaluates arguments **zero or more** times
  - this **might** be more inefficient
- Fully Lazy Evaluation
  - Evaluate arguments **zero or one** times
- Final Example
- (λx. + x x) (* 6 2)
- Normal Order β reduction:
  - + (* 6 2) (* 6 2)
  - + 12 (* 6 2)
  - + 12 12 = 24
- Applicative Order β reduction:
- Evaluate argument *before* β reduction; we get 12
  - + 12 12
  - =24

5. Boolean Algebra and Recursion

- True or False
    - (IF-THEN-ELSE)
- Charles Babbage (1791 - 1871)
    - Differential Engine (1822)
        - Solve Polynomial Functions
    - Faraday's electric engine (1821)
    - Analytical Engine (1830)
        - Programmable, memory, printer, CPU
        - First built 153 years later!
- George Boole (1815-1864)
- First Professor of Mathematics in UCC
- Formalised logic
- Lets us reason about unseen cases
    - Enables scaling in modern computers — hyperscale
- "The Joy Of Logic"
    - https://vimeo.com/199831792
- Boolean Operators
    - (AND, OR…)
- Relational Operators
    - (<, >, =…)
- Prefix notation?
    - (> 2 1)
    - (< 4 2)
- Racket?

```
>  (> 2 1)
> (= 2 1)
> (< (+ 3 1) (* 4 5))
> (+ 2 (> 3 1))
```

- Conditionals
- General view of conditional:
  - if E then C1 else C2
- Meaning:
  - if condition E is true
    - THEN execute command(s) C1
    - ELSE execute command(s) C2
- λ calculus / Racket view:
  - if condition E is true
    - THEN return C1
    - ELSE return C2
- Examples
- > (if (> 2 0) “first”   “second”)
- “first”
- Return the larger of two numbers:
- (λxy. if (> x y) x   y)

- IF *can* have only one part as well:
  - (λxy. if (> x y) x)
- **Notice**: λ calculus can use all the classical boolean constructs:
  - and, or, not
  - (or #t #f)
  - #t

- (not #t)
- (or #f #t)
- (and #f #t)

- (or (and #t #f) (or #f #t))

- All numbers are considered #t
- (if 1 “first” “second”)
- **‘or’** returns the first TRUE value it can find; otherwise it returns FALSE.
- (or 1 0)
- (or 0 1)
- (or 31 #t)
- (or #t 31)

- Sometimes the first TRUE value is **not** a boolean!

- **Why return the first item?**
- Efficiency: This can save unnecessary evaluations…
- (or (f1 a) (f2 a) (f3 a)…(f1000 a))
- Stop evaluating as soon as possible

- AND is the opposite to OR
- (and 3 -1)
- (and -1 3)
- (and 1 #f)
- (and #f 2)
- (or 1 #f)
- (not -1)
- As with OR, the TRUE item could be non-boolean
- **Efficiency of AND vs OR**

- AND requires everything to be evaluated for true
- (and (f1 a) (f2 a) (f3 a)…(f1000 a))
- Extra arguments?
- (not 1 2)
- **not**: *arity mismatch…*
  expected: 1
  given: 2
- (and 1 2 3 4)
- (or 1 2 3 4)
- Strings are always true
    - (and “hello” “goodbye”)
    - “goodbye”
    - (or “hello” “goodbye”)
    - “hello”
- Using conditionals
    - Decision making
    - Give appearance of intelligence
    - (define pass?
    -         (lambda (x)
    -             (if (>= x 40) “pass” “fail”)))
    - (define pass2?
    -         (lambda (x)
    -                 (if (>= x 40)   #t   #f)))
- (pass? 25)
- (pass2? 25)
- **Which is better?**
    - pass2? because it returns a boolean
    - It is more *tightly coupled*
- pass2? returns either #t or #f
- pass? returns a string each time
- A string *has* a boolean value: **#t.**

(if

```
     (and (pass? 35) (pass? 45))
   “Passed both”
   “Didn’t pass both”)
```

- (define passBoth (lambda(x y)

  ```
  (if
     (and (pass2? x) (pass2? y))
      “Passed Both”
     “Didn’t pass both”)
      ))
  ```

Another example

```
(define scrape
  (lambda (x)
    (if   (and (< x 45) (pass2? x)
   #t   #f )))
```

- (scrape 42):
  - (< *42* 45)
  - (pass2? *42)*
- → (and (< x 45) (pass2? 42))
- (define pass3? (lambda (x) (>= x 40)))
  - Evaluates (>= x 40)
  - Returns the boolean value.
- More examples:
- Write two functions
  - (1) Check if a number is even.
  - (2) Checks if a number is high-even, that is, if the <u>number is greater than 20 ***and*** even</u>.
- Built in Racket function:
- > (integer? **x**)

```
> (define even
    (lambda (x)
            (integer? (/ x 2))
    )
)

> (define high-even
    (lambda (x)
     (and (> x 20) (even x))
    )
  )

> (define high-even2
     (lambda (x)
      (if (> x 20) (even x) #f)
    )
)
```

- Function call overhead
- Which is better? high-even or high-even2?
- high-even2 executes a function call first, incurs “overhead”
- high-even relies on short-circuiting behaviour of AND.
- When **(> x 20)** returns **#f,** execution stops
- **Remember:** AND returns the first FALSE item it finds
- Therefore, high-even is better.

5.1. Recursion

- Solve a problem with a function that calls itself
- For example, how do you calculate Factorial *n*?
- 3! = 3 * 2 * 1
- 4! = 4 * 3 * 2 * 1
- Answer: n * Factorial (n-1)
- .... kind of
- Prove for simple case
- Prove for case i+1
- Assume true for all
- **Inductive proof for dominoes:**
- *Informal*
    - The first domino knocks over the second
    - which knocks the third
    - and so on ....
- *Classic*
    - The first domino falls
    - Whenever the ***i***th domino falls, it knocks the ***i+1***th domino
    - Therefore, all the dominoes fall.
- Idea
    - Can prove something for a simple case
    - Prove it for a general case
    - Assume proven for all cases
- Important because?
    - Numbers go to infinity
    - Impossible to prove for every case
- Recursion is similar to induction
- Solve *simple case* of a problem
- Figure out how complex (*general*) case can be solved
- ....using the simple case
- Magically solves all cases

- Example: Compute Factorial
- Factorial 1 = 1
- Factorial n = n * (n-1) * (n-2) *… * 1
- Factorial n-1 = (n-1) * (n-2) *… * 1
- Factorial n = n * Factorial (n-1)

Execution of factorial

- Factorial 1 = 1 [SIMPLE CASE]
- Factorial n = n * Factorial (n-1) [GENERAL CASE]
- Fact 3: *(shorthand for Factorial 3)*
  - Fact 3 = 3 * Fact 2
  - Fact 2 = 2 * Fact 1
  - Fact 1 = 1
- Go back up:
  - Fact 2 = 2 * 1
  - Fact 3 = 3 * 2 * 1
- Answer = 6.
- Each line:
  - Does ONE thing
- Passes on the rest of the problem (to itself)
- **Using Lambda Calculus**
- fact : (λn. if (= n 1)

  1

  (* n (fact (- n 1))))
- Execute (fact 3):
  - if (= 3 1) 1 (* 3 (fact (- 3 1)))
  - (* 3 (fact (- 3 1))) = (* 3 (fact 2))
- Execute (fact 2):
  - if (= 2 1) 1 (* 2 (fact (- 2 1)))
  - (* 2 (fact (- 2 1))) = (* 2 (fact 1))
- Execute (fact 1):
  - if (= 1 1) 1 (* 1 (fact (- 1 1)))

    - 1
- Go back:
    - (* 2 (fact 1)) = (* 2 1)
    - 2
- Go back:
    - (* 3 (fact 2)) = (* 3 2)

Take your own notes for AST.

In Racket

```
(define fact
  (lambda (x)
     (if (= x 1)
         1
         (* x (fact (- x 1))))
     )
   )
```

- (fact 3)
    - 6
- (fact 5)
    - 120
- (fact -1)

- **Infinite Recursion**
    - Each function call incurs overhead,
    - including local variables…
    - eventually computer runs out of memory
- **Racket-specific**
    - If a call exceeds maximum *allowed* memory (default 240MB)
    - …it is terminated.
- **Error checking**
    - Change condition to
    - (if (<= x 1)…

- Fibonacci's assumptions about rabbits
    - Start with one pair
    - Rabbits can mate at the age of one month
    - Gestation period is one month
    - Two rabbits produced each time
    - Equal number of male and female rabbits
    - Rabbits never die

Golden Ratio

- Ratio of sum of **(a + b) to a** is the same as the ratio of **a to b**
- Appears all over nature, art, music

Sacred Geometry

- *Plato said god geomtrises continually*
    - Plutarch (45AD – 127 AD)

Back to Fibonacci

```
(define fib
   (lambda (x)
         (if (<= x 2)
         1
         ( + (fib (- x 1))
         (fib (- x 2))
    )))
```

- Additional reading on recursion
- Given in Reference Material section
    - Structure and Interpretation of Computer Programs
- On the class website
- **<u>Section 1.2 Procedures and the Processes They Generate</u>**
- Implement and understand two different implementations of **factorial.**

- Iteration *may sometimes* replace recursive function

  int fact=1;

  for (**int** j=arg; j>1; j--)

  fact = fact * j;

- But not always!
- Sometimes not trivial to replace a recursive function.
- For example browsing a tree of item categories on argos.ie or amazon.com
- Useful exercise: implement Fibonacci in Java

```
public static int fibonacciLoop(int number) {
          if (number == 1 || number == 2) {
               return 1;
          }
          int fibo1 = 1, fibo2 = 1, fibonacci = 1;
          for (int i = 3; i <= number; i++) {
               fibonacci = fibo1 + fibo2;
               fibo1 = fibo2;
               fibo2 = fibonacci;

          }
          return fibonacci;
}
```

number	**fibonacci**	**fibo1**	**fibo2**
(initial)	1	1	1

5.2. Solving problems recursively

- Identify the base case; then identify the general case
- Not always easy
    - General case may be difficult to formulate
- Example: Add numbers from *0* … *n*.
    - Base Case/Terminating Case/Simple case
    - *0* … nothing to add
    - i.e. sum(0) = 0
- General case:
- sum(n) = n + (n – 1) + (n – 2) + …. + 0
- sum(n-1) = (n – 1) + (n – 2) + …. + 0

Solving Problems Recursively

- Putting it together:

sum: λn. if (= 0 n) 0
(+ n (sum (- n 1)))

- Another view: recognise a sequence
- n 0 1 2 3 4 ...
- sum(n) 0 1 3 6 10 ...

- n 0 1 2 3 4 ...
- fact(n) 0 1 2 6 24 ...
- Write a recursive function that generates the sequence
- i.e. for a given value of n, it produces sum(n) or fact(n).

Generating Functions from Sequences

- Using λ calculus & recursion for design
    - Try to describe what is happening with sequence
- Example: explain the following sequence

  - n 1 2 3 4 ...
  - f (n) 1 5 9 13 ...
- Base?
  - f(1) = 1
- General?
  - No easy way to spot; however, *usually* f(n) is somehow related to f(n-1)
  - **Here**, each number is 4 bigger than the previous one.
  - Therefore, f(n) = f(n-1) + 4

Mathematically

- Mathematically:
- f(1) = 1
- f(n) = f(n-1) + 4
- Recursive λ calculus function:

  f : λn. if (= n 1) 1

  (+ (4 (f (- n 1))))
- Another example:
- n 1 2 3 4 ...
- f (n) 1 5 13 29 ...
- f(1) = 1. (won't always be ....)
- f(n) = ?
- *Usually* f(n) = calc(n) + f(n-1)
- (but not always...)
- Write out:
- n 1 2 3 4 ...
- f (n) 1 5 13 29 ...
  - f(1) = 1
  - f(2) = 5 = f(1) + 4
  - f(3) = 13 = f(2) + 8
  - f(4) = 29 = f(3) + 16
  - f(5) = 61 = f(4) + 32

  - 4, 8, 16... powers of 2.
- Aside:
- Power of two in λ calculus?
  - λx. (* x x)
  - Only squares; need to generate higher powers of two
- A more useful function:
- (pow x y) (i.e. x^y)
- Base case: $x^0 = 1$, thus (pow x 0) = 1
- General case: (* x (pow x (- y 1))
- Because $x^y = x * x^{y-1}$
- Notice:
  - Two variables
  - Only one controls recursive call
- Recursive λ calculus function:

```
pow : λxy. if (= 0 y) 1
           (* x  (pow x (- y 1)))
```

- Evaluate 3^2
- (pow 3 2)
- if (= 0 2) 1 (* 3 (pow 3 1))
- if (= 0 1) 1 (* 3 (pow 1 0))
- if (= 0 0) 1 (* 3 (pow 3 -1))
- (* 3 (* 3 1)) = 9

Back to original question

- f(1) = 1
- f(n) = f(n-1) + 2^n
- Recursive λ calculus function:

```
f : λn. if (= n 1)        1
                          (+ (f (- n 1))  (pow 2 n) )
```

- Procedures and Processes
- *Procedures*: another term for functions.
- Function call generates a computational *process*
    - i.e. a set of steps required to execute the code
- Important to understand this process to become an expert programmer
    - i.e. not all code is executed
    - sometimes code is executed multiples times
- Possible to examine the *shape* it generates.

Reminder: **factorial**

- (fact n) = (* n (fact (- n 1)) (General Case)
- Example execution: (fact 4)

```
(fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2  1)))
(* 4 (* 3  2))
(* 4  6)
24
```

- Factorial with a non-recursive process
- Avoid deferred operations:
- Keep a running product with every recursive call
- Much like with loops/iterations. Recall:

```
int product=1;
int counter=1;
while (counter<=n) {
     product=product*counter;
     counter++;
}
```

Factorial: non-recursive process with a recursive function

- (facto counter product ***n***):

  (facto 1 1 ***4***)
  (facto 2 2 ***4***)
  (facto 3 6 ***4***)
  (facto 4 24 ***4***)

- Constant Memory/space required
  - Because no deferred operations.
  - Much like with loops.
  - Hence, an ***<u>iterative process</u>***.

```
(define facto (lambda (counter product n)
        (if (> counter n)
        product
        (facto  (+ counter 1)
                 (* counter product)
                n
))))
```