COMP/CMPE 314 - Principles of Programming Languages - Notes

Chris Stephenson, Istanbul Bilgi University, Department of Mathematics, and course students

May 28, 2016

WEEK 1

Answers of the quiz #1

Question #1

Language

- A finite alphabet
- A set of strings of the symbols in the alphabet (usually infinitive)

Grammar

- A set of productions (rewriting rules)
- A finite alphabet of terminal symbols
- A finite set of non-terminal symbols
- A sentence symbol (S)

Simple language: [a]

- $S \to a$
- $S \to aS$ (infinite)

$S \to number$
$S \to S + S$
$S \to S * S$
$S \to (S)$
alphabet = [+, *, (), number]

Question #2

```
(define (interp [a : ArithC]) : number
    (type-case ArithC a
        [numC (n) n]
        [plusC (l r) (+ (numC-n l) (numC-n r))]
        [multC (l r) (* (numC-n l) (numC-n r))]))


In this code;
(+ 4 3) works but,
(+ (* 3 7) (+ 4 2)) crashes
```

In order to make it work we need to change numC to interp like this

```
(define (interp [a : ArithC]) : number
    (type-case ArithC a
        [numC (n) n]
        [plusC (l r) (+ (interp l) (interp r))]
        [multC (l r) (* (interp l) (interp r))]))
```

Programming Languages

Programming Languages		
Java	Assembly	Swift
Python	C	Prolog
Ruby	C#	Fortran
Objective-C	Racket	R
Lisp	Haskell	Whitespace
Javascript	Pascal	Giuseppe
Scala	HTML	Ada
Pick	Perl	XML
SQL	BASH	XSDL
Scratch		

Kinds of Programming Languages

Markup Language
Functional
Object oriented
Procedural
Scripting
Graphical
Experimental
Declarative
Machine

The code below is valid in Java and also valid in C.
What value does the function/method `funny2` returns?

```
int funny(int a, int b){
   return a + 2 * b;
}

int funny2(){
   int a = 2;
   return funny(a++, a++);
}
```

In this example the value returned in Java and C are different. Java returns 8 but C returns 7.
This situation caused by the compilers of this languages. In `funny(a++, a++)` statement Java starts from left `a++` but C starts from right `a++`.

Let us consider this now
1000000 + 2000000
"1000000" + "2000000"
In the first example Java returns a negative integer value. Because it assigns numbers default by int and maximum int value is 2,147,483,647.
In the second one Java concatenates the two strings like "10000002000000" (But some languages adds them)

All programming languages are data formats for data input to other programs.

$$\text{Program text} \xrightarrow{parse} \text{Data Structure} \xrightarrow{interp/eval} \text{Answer}$$

```
(eval (parse '(+ 2 (* 3 4))))
```

Parsing is common to all compilers/interpreters

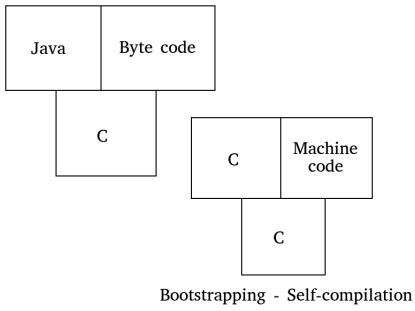

Figure 1: Bootstrapping - self compilation

WEEK 2

Functions/Methods

Robert's on Methods;

- "Hiding complexity"
- "Tools for programmers"
- "Method calls as expressions"
- "Method calls as messages"

"The calling mechanism is that the actual parameters are copied to the formal parameters and the code is executed."
it's a lie!

How methods are used?

Within a function/method you can call another method (We have certainly done that! Example: `Math.min()`)
Missing: You can call a method in the expression for the actual parameters;

`myMethod(Math.max(a, b));` ← Piping methods

⋆ Linking codes is very powerful.

- Used for decomposition - Break a problem into pieces
- Used for algorithms - Abstraction
  (Example: Binary search)

In the below java code which is taken from "The Art and Science of java" lets calculate the following equation:

$$\binom{n}{r} = \frac{n!}{(n-r)!r!}$$

```
private int factorial(int n) {
  int result = 1;
  for (int i = 1; i <= n; i++) {
    result *= i;
  }
```

```
  return result;
}

public int combinations(int n, int k) {
  return factorial(n) / (factorial(k) * factorial(n - k));
}
```

If we try to calculate `combinations(15, 2)` java will return `-4` which is incorrect and there is no error information. It should return 105. This is because of the max integer limits.

The calculation should be done like this;

```
public int computeBinomialCoefficient(int n, int k) {
  int ans = 1;
  k = Math.min(k, n - k);
  for (int i = 0; i < k; i++) {
    ans = ans * (n - k + 1 + i);
    ans = ans / (i + 1);
  }
  return ans;
}
```

We wrote a calculator to make a programming language, we need to add abstraction. (= generalisation. In place of numbers we will use identifiers in our calculations.)
Example:

```
int square(int x){ // Here x is formal parameter and bound identifier
 return x * x;
}
```

⋮

application

```
println(" " + square(7)); // Here "7" is the actual parameter
```

We evaluate the body of the function by substituting the actual parameter for the formal parameter in the body of the function.

We need to extend our data definition.

We need to add;

- Function definition
- Function application

To start with we will keep our function definitions seperate.

Our functions will have a seperate namespace.

Our interpreter will have two inputs - a list of function definitions - as an expression. (Which may include function applications)

What does a function have?

- A formal parameter - name (identifier)
- A body - expression (in our extended version)
- A name - identifier (in the function name namespace)
- Function name - identifier in the function name namespace

- Actual parameter - identifier
  Extend our expression data definition.
  We need to add two more variants.

- Function application

- Identifier

We need to extend the interpreter to deal with the two new variants in the data definition.

- Identifier → error

- Function application
  `(apply (find-function function-list function-name) actual-parameter)`

Statement of purpose for apply, substitute the actual parameter for the formal parameter everywhere in the body of the function, then evaluate the result.

```
(define '(apply function-definition actual-parameter){
 (interp (subst (function-definition formal-parameter) actual-parameters))
})
```

What does subst do?
name expression expression → expression
Template:
number → number
arith-expression → same expression but subst the operands
identifier → if the identifier is the identifier to be substitued, we replace it
function application → function application, but subst the actual parameter

The syntax of the λ-calculus:
$\Lambda \rightarrow \vartheta$
$\Lambda \rightarrow (\lambda \vartheta \lambda)$
$\Lambda \rightarrow (\Lambda\Lambda)$
Valid sentences:
a
$(a\ b)$
$(\lambda\ x\ x)$ - informally this is the identity function
$(a\ b\ c)$ - *not valid*
$(\lambda\ f\ (\lambda\ x\ (f\ (f\ x))))$ - a function doubler
$(\lambda\ v\ (...))$

WEEK 3

A program that looks the same but evaluates to two different answers should worry us!

The following function/method is both works on Java and C, but when we compile it gives different results.

```
 int funny (int a, int b){
   return a * 2 + b;
 }

 int haha () {
   int z = 2;
   return funny (z++, z++);
}
```

This method/function is legit for both languages.

In C, we can compile it with one easy step. For this, we need to add C header files and main function.

```
#include<stdio.h>

int funny (int a, int b){
  return a * 2 + b;
}

int haha () {
  int z = 2;
  return funny (z++, z++);
}

int main(int argc, char** argv) {
  printf("\n\nThe answer in C is %d\n\n", haha());
  return 0;
}
```

The output of the program is:

```
The answer in C is 8
```

In Java we need to change the methods a little bit. We should add static at the beginning of the methods in order to compile.
We can do this without changing the main file via terminal.

```
 cat funnypre.java | cat - funnyhaha | sed -e 's/^int/static int/g' | cat - funnypost.java
```

(The files are in the course github repository.)

Output of the code is:

```
class Funny{

static int funny (int a, int b){
  return a * 2 + b;
}

static int haha () {
  int z = 2;
  return funny (z++, z++);
}

public static void main(String[] args){
  System.out.println("\n\nThe answer in Java is " + haha() + "\n\n");
}

}
```

The output of the program is:
`The answer in Java is 7`

We can write and compile codes without using Eclipse.
Integrated Development Environment (IDE) → Eclipse
Unintegrated Development Environment → Terminal

For repetitive jobs we should write bash scripts.
Example for C:

```
cat funnypre.c funnyhaha funnypost.c > funny.c
gcc funny.c -o funny
./funny
```

---

z++

The ++ post operator produces a value the same as the value of its operand. But it changes the subsequent value of the operand as a side effect.
In a program using a for loop to process an array we have (maybe) unnecessary state.
Look at Hadoop - Map reduce ← must be stateless (big data)

$$\text{text} \xrightarrow{parse} \text{first representation} \xrightarrow{desugar} \text{second representation} \xrightarrow{eval} \text{evaluation}$$
"desugaring" = "macro processing"

For example in Racket, cond is desugared into a series of nested if statements.

What does our program evaluation look like?

```
(eval (desugar (parse '(* (-3) (+ 7 8)))))
```

Rules:
$\lambda \rightarrow v$ identifier
$\lambda \rightarrow (\lambda v \Lambda)$ anonymous function definition
$\lambda \rightarrow (\Lambda\Lambda)$ function application

$(a\ (b\ c))$ ✓
a ✓
(a) x
$(a\ b)$ ✓

$(\lambda\ x\ x) \rightarrow$ identity function

```
int funny(int y){
  return y;
}
```

$((\lambda\ x\ x)\ (a\ b))$
In here x is bound, a and b are free

A "complete" program will have no "free" identifiers.
A part of that program may have free identifiers.

- If M is an identifier, x. $FI(M) = X$
- If M is an application $(M1\ M2)$. $FI(M) = FI(M1) \cup FI(M2)$
- If M is a definition $(\lambda\ x\ M1)$. $FI(M1) - \{x\}$

$FI[(\lambda x(\lambda yx))] = \{\}$
$FI[(\lambda yx)] = \{x\}$

If statement

True:
$T = (\lambda x(\lambda yx))$
$(Ta) \xrightarrow{\beta} (\lambda ya)$
$((Ta)b) \xrightarrow{\beta} a$

False:
$F = (\lambda x(\lambda yy))$
$((Fa)b) \xrightarrow{\beta} b$

WEEK 4

Lies and Equality (In Java and Other Languages)

In Java;

- "`=`" does not mean "equals" (its assignment)
- "`&`" does not mean "and" (its bitwise and)
- "—" does not mean "or" (its bitwise or)
- "`!`" does not mean "surprise"
- "`int`" does not mean "integer"
- "`Integer`" does not mean "integer"
- "`BigInteger`" does not mean "BigInteger"

- We use "`==`" for "equals"
- We use "`&&`" for "logical and"
- We use "——" for "logical or"

This things are left from C which was invented as high level assembler, not high level language.
`int` means a 32 bit two's complement quantity (now most compilers are 64 bit)

If I declare `MyClass a;`
`a` is not necessarily an object of class `MyClass`.

Equality

What does == mean?

```
class Equality {

  public static void main (String[] args) {
    MyInteger a = new MyInteger(42);
    MyInteger b = new MyInteger(42);
    MyInteger c = new MyInteger(43);
    MyInteger d = c;
    String s1 = "Chris";
    String s2 = "Chris";
    System.out.println("\n a = " + a + ",b = " + b + "\n");
    System.out.println("\n a == b is " + (a == b) + "\n");
    System.out.println("\n s1 == s2 is " + (s1 == s2) + "\n");
    System.out.println("\nd = " + d + "\n");
    System.out.println("\n c == d is " + (c == d) + "\n");
    c.setV(44);
    System.out.println("\n c = " + c +  "\n");
  }
}
```

```
class MyInteger {
  int value;

  MyInteger (int v) {
    value = v;
  }

  public String toString () {
    return "" + value;
  }

  public void setV (int v) {
    value = v;
  }
}
```

Output:

```
 a = 42,b = 42


 a == b is false


 s1 == s2 is true


d = 43


 c == d is true


 c = 44
```

It does not mean "equal in value" (necessarity)
`setV` method makes our program mutable.
Java is broken. `NullPointerException` is most likely seen error and it causes programs to crash.
The best way to solve this problem is to make our programs **immutable**.

```
 String a = "Chris";
 String b = "Chris";
```

In Java Strings are immutable!

`http://javapractices.com`

Our program;

```
  '(((fundef factorial n (if2 n 1 (* n (factorial (- n 1))))) (factorial 3))
```

parse + desugar

```
(appC 'factorial (numC 3))

(ifzC (idC 'n) (numc 1) (mulC (idC 'n) (appC 'factorial (subC (idC 'n) (numC1)))

(idC 'n)'s will be changed with (numC 3)
```

α Transformation

$(\lambda\ x\ M) \equiv (\lambda\ y\ M\ [\ x{:=}y\ ])$

η Transformation (Optional)

$(\lambda\ x\ (f\ x)) \xrightarrow{\eta} f$

β Transformation

$((\lambda\ x\ M)\ N) \xrightarrow{\beta} M\ [\ x := N\ ]$

WEEK 5

The "real" world

"Evaluation Strategies"

- a strange word to use?
- we make the languages.

The problem is that most languages have design defects, so evaluating them is confusing.

What do we do?
When and What and How parameters are evaluated?

When?

In Java;

```
myfunc(a, g(b), c + d);
myFunc(a++, a++, a++); // order matters!
```

** Java compiler (javac) starts evaluating from left however C compiler (gcc) starts from right.**

Are the parameters evaluated left to right or right to left?
In Java, left to right
In C, UNDEFINED!? (Different compilers different answers)

OR We cannot evaluate the parameters at all. And simply pass the expressions to the function body. (All programming languages do this for some things)
Racket is a strict language, which means parameters are evaluated.

```
;#lang lazy
#lang racket
(define (my-if condition t f)
  (cond
    (condition t)
    (else f)))

(define (fac n)
  (my-if (< n 1) 1 (* n (fac (- n 1)))))
```

Here `my-if` looks like a function. But it isn't a function in racket. It tries to calculate parameters first. Thus it leads to infinitive loop.
If we uncomment `#lang lazy` it will work.

Except for if
In Java `f(a) && g(b)` is a boolean expression
They call it "Short circuited evaluation"
Evaluated left to right and stop when we know the answer.

A practical example:

```
int i = 0;
while(i < a.length && a[i] != 0) {
  ...
  i++;
}

while(a[i]!=0 && i<a.length){ // after a swap, what goes wrong?
  ...
  i++;
}
```

If the condition goes to `i > a.length`, the `a[i]` throws an error. So the order is important!

Is evaluation in this language "Strict?" "Non Strict?" "Sometimes Strict?"
A different way of binding formal parameters: _____

Environments

- Substitution is not how the real world works (For reasons of efficiency)
- An environment is an ordered store of identifier/value pairs
- For function application we add formal parameter/actual parameter value pairs to the environment and use that environment to evaluate the function body.
- Our evaluator takes the environment as an additional parameter when the evaluator evaluates an identifier, it looks up the value in the environment.

Data Definitions

An environment is empty OR an identifier/value pair plus an environment.

- look up Env. environment identifier → value or Error
- extend Env. environment identifier → environment

```
Example:
(fundef myFunc (a b c) (+ a (* b c))) //function definition

(myFunc 3 7 (+ 2 9)) //function application
```

Evaluate a function application

```
(eval (extend Env (extend Env (extend Env (empty Env, 'a, 3) 'b, 7) 'c, 11) '(+ a (* b c))))

(fundef f1 x (f2 4))
(fundef f2 y (+ x y))
```

Whats wrong with f2?
It's a free identifier!
In Java:

```
int f2(int y){
  return x + y;
}
```

evaluate (f1 42)
x gets bound to 42
y gets bound to 4
(+ x y) looks up these values in the environment
The answer is 46 **Wrong!!**

Identifier capture!
This is called "dynamic scope" and it is a bug.

The solution (for now)
When we apply a function, do not extend the current environment with formal parameter/value bindings. Extend the empty environment.

Grammar

1) $\Lambda \rightarrow \upsilon$
2) $\Lambda \rightarrow (\lambda\ \upsilon\ \Lambda)$
3) $\Lambda \rightarrow (\Lambda\ \Lambda)$
Data structure and therefore definitions and programs will reflect this.

M[x:=N]

1) M is an identifier
   a) M=x $\Rightarrow$ M[x :=N] is N
   b) M$\neq$x $\Rightarrow$ M[x:=N] is M

2) M is an application (just do the two parts)

3) M is a function definition
   Case (a) is easy λyM1, y=x
   Case (b) is hard λyM1, y$\neq$x

Church Numerals

n $\equiv$ (λ f (λ x (f (f ($\underbrace{\text{f...x}}_{nf's}$)))))

ChurchB(n) $\equiv$ (f (ChurchB(n -1)))
ChurchB(1) $\equiv$ (λ f (λ x (f x)))

Church(n) = (λ f (x x (ChurchB(n))))

TWO=(λ f (λ x (f (f x))))
ZERO=(λ f (λ x x))

A successor function "SUCC"
SUCC=(λ n (λ f (λ x (f ((n f) x))))
onSUCC=(λ n (λ f (λ x ((n f) (f x)))))

To add m and n;
(λ m (λ n ((m succ) n)))

WEEK 7

Evaluation "Strategies"

All about function application - which is the central concept in programming **IMHO** an oxymoron

We looked at sequence. (2 weeks ago)

- Strict vs. non-strict
- Eager vs. lazy
- Left to right vs. right to left

What is passed when we apply a function to actual parameter.
Example (in Java)

```
{
 a = 2;
 myFunc(a);
}

void myFunc(int a){
 ...
 a = a + 1;
}
```

What is the value of `a`?
`System.out.println("a is: " + a);` → The answer is 2.

Why?

Because Java/C/C#/Ruby/Python/C++ are, call by value
For simple types, most common programming languages are call by value.
The alternative is call by preference (PHP can do this)

But what about complex types?

Example: array, structure, object
In Java:

```
{
 ...
 a[k] = 2;
 myFunc(a);
 System.out.println("a[k] is " + a");
}
```

```
void myFunc(int[] a){
 ...
 a[1] = a[1] + 1;
 ...
}
```

Call by reference for arrays/Structures/Strings/Objects
(Except C - C structures are naturally call by value)
Java Strings can be considered call by value. (Because Java Strings are immutable)

SUPER ACCELERATED CMPE313 (SICP - The Purple Book)

What is a list? A list of numbers x?

- Empty
- A pair of a number x and a list of number x.

```
;; add-up the numbers in a list
(define (sum-lon lon)
  (cond
    ((empty? lon) 0)
    (else (+ (first lon) (sum-lon (rest lon))))))


(define (mul-lon lon)
  (cond
    ((empty? lon) 1)
    (else (* (first lon) (mul-lon (rest-lon))))))


(define (fold combiner null-value lox)
  (cond
    ((empty? lox) null-value)
    (else (combiner (first lox) (fold combiner null-value (rest lox))))))
```

fold is a powerful abstraction.

I want to evaluate a polynomial.
$a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_nx^n$
Data definition a polynomial number $\rightarrow$ number

Horner's rule;
$a_0 + x(a_1 + x(a_2 + x(a_3...a_n))))$

To evaluate a polynomial

```
(define (eval-poly p x)
  (fold (lambda (l r) (+ l (+ l (* x r)) non-empty p))))
```

map

```
(define (map f lox)
  (fold (lambda (l r) (cons (f l) (cons (f l) r)) empty lox)))
```

filter

```
(define (filter p? lox)
   (fold (lambda (l r) (if (p? l) (cons l r) empty lox))))
```

Suppose I am doing a map to add 2 to every item in a list:

```
(define (add2 lon)
   (map (lambda (x) (+ x 2)) lon))

(define (mul 7 lon)
   (map (lambda (x) (* x 7)) lon))

(define (curry f x)
   (lambda (y) (f x y)))
```

Generalizing:

```
(define (arith-map operator constant)
   (map (curry operator constant) lon))
```

The λ-calculus

Summary:

$\Lambda \to \upsilon$ (identifier)
$\Lambda \to (\lambda\ \upsilon\ \Lambda)$ (definition)
$\Lambda \to (\Lambda\ \Lambda)$ (application)

α rewriting rule:
$(\lambda\ \upsilon\ \Lambda) \equiv (\lambda\ u\ \Lambda[\upsilon{:=}u])$ (plagiarizing student rule)

η rule (optional):
$(\lambda\ \upsilon\ (\mathrm{f}\ \upsilon)) \equiv \mathrm{f}$

β rule:
$((\lambda\ \upsilon\ \Lambda_1)\ \Lambda_2 \xrightarrow{\beta} \Lambda_1\ [\upsilon{:=}\Lambda_2]$

defining substitution is more complicated
$\Lambda[\upsilon{:=}\Lambda_0]$ - see the slides

(a) $\Lambda = u$

if $u = \upsilon \to \Lambda_0$

else Λ

(b) $\Lambda = (\Lambda_1\ \Lambda_2)$

$(\Lambda_1[\upsilon{=}\Lambda_0]\ \Lambda_2[\upsilon{=}\Lambda_0]$

(c) $\Lambda = (\lambda\ u\ \Lambda_1)$

(i) $u = \upsilon \to \Lambda$

(ii) $u \neq \upsilon \to$ if necessary re-letter before substituting in the body.

Church Numerals

$0 \equiv (\lambda\ f\ (\lambda\ x\ x))$
$1 \equiv (\lambda\ f\ (\lambda\ x\ (f\ x)))$
$2 \equiv (\lambda\ f\ (\lambda\ x\ (f\ (f\ x))))$

Successor:
SUCC=$(\lambda$ n $(\lambda\ f\ (\lambda\ x\ (\ f\ ((\text{n}\ f\ )\ x\ ))))$
onSUCC=$(\lambda$ n $(\lambda\ f\ (\lambda\ x\ ((\text{n}\ f\ )\ (\ f\ x\ ))))$

$((\lambda$ n $(\lambda\ f\ (\lambda\ x\ (\ f\ ((\text{n}\ f\ )\ x\ ))))))\ \lambda\ f\ (\lambda\ x\ (\ f\ x\ ))))$

$\downarrow \beta$

$(\lambda\ f\ (\lambda\ x\ (\ f\ ((\lambda\ f\ (\lambda\ x\ (\ f\ x\ )))\ f\ )\ x\ ))))$

$\downarrow \beta$

$(\lambda\ f\ (\lambda\ x\ (\ f\ ((\lambda\ x\ (\ f\ x\ )))\ x\ ))))$

$\downarrow \beta$

$(\lambda\ f\ (\lambda\ x\ (\ f\ (\ f\ x\ )))) \equiv 2!$

WEEK 8

Haskell

- Purely functional - no state of programs
- Statically typed
- Lazy

Functional Programming

- Functions are first class values.
- No state / no side effects

How can we do this in our interpreter?
A function application `(name expr)` will no longer be.
It will be `(expr expr)`
This means we have more than one type of expression. So runtime types. So we can have booleans.

1 Types

2 How will functions be named?

- Just like any other expression.
- Identifiers (which are always formal parameters) and are then applied to actual parameters.
- This will also be true for functions.

What would we like to have?

```
(let
   (myfunc (lambda (x) (+ x 3)))      function definition
   (myfunc2 (lambda (y) (* y 4)))     function definition
   ...<program>                       expressions
   )
```

let is sugar.
Desugaring the let;

```
((lambda (myfunc myfunc2)
   <program>)
   (lambda (x) (+ x 3)) (lambda (y) (* y 4)))
```

Example:

```
(let
     ((double (lambda (x) (* x 2)))
   (double 7))

((lambda (double)
   (double 7))
     (lambda (x) (* x 2)))

(let
   ((multiply-list-by-n (lambda (l n) (map (lambda ((x) (* x n)) l))
     (multiply-list-by-n '(4 5) 3)
     (multiply-list-by-n '(5 6) 7)))))
```

3 Closure

We need to distinguish function definitions from function values when we evaluate a function definition we get a function value.

function definition $\xrightarrow{one->many}$ function value

A function value is a function definition plus the environment at the point where the function value was evaluated. This is called closure.

4 When we apply a function value?

Extend the environment in the function value with the bindings of the formal parameters to the actual parameters.

4.1 Sugaring the λ-calculus

Some definitions
(F^0M) $\equiv$ M
(F^{n+1}M) $\equiv$ F(F^nM))

C_n = Church numeral n

So,
$C_n \equiv(\lambda\ f\ (\lambda\ x\ (\ f^n\ x)))$
$C_0 \equiv(\lambda\ f\ (\lambda\ x\ (\ f^0\ x))) \equiv (\lambda\ f\ (\lambda\ x\ x))$
$C_1 \equiv(\lambda\ f\ (\lambda\ x\ (f^1\ x))) \equiv (\lambda\ f\ (\lambda\ x\ (f\ x)))$
$C_2 \equiv(\lambda\ f\ (\lambda\ x\ (f^2\ x))) \equiv (\lambda\ f\ (\lambda\ x\ (f\ (f\ x))))$

LET - as we did for our interpreter

SUCC	$(\lambda\ n\ (\lambda\ f\ (\ \lambda\ x)\ (f\ ((n\ f)\ x))))$
TRUE	$(\lambda\ x\ (\lambda\ y\ x))$
FALSE	$(\lambda\ x\ (\lambda\ y\ y))$
IF	$(\lambda\ a\ (\lambda\ b\ (\lambda\ c\ ((a\ b)\ c))))$
ZERO?	$(\lambda\ n\ ((n\ (\lambda\ x\ \mathit{FALSE}))\ \mathit{TRUE}))$

4.2 Data Structures

If we can make a pair, we can make a data structure.
What are the semantics of `CONS FIRST REST` (in Racket)?

```
(FIRST (CONS A B)) = A        -> necessary semantics

(REST (CONS A B)) = B         -> necessary semantics
```

CONS	$(\lambda\ a\ (\lambda\ b\ (\lambda\ f\ ((f\ a)\ b))))$
FIRST	$(\lambda\ c\ (c\ \mathit{TRUE}))$
REST	$(\lambda\ c\ (c\ \mathit{FALSE}))$

WEEK 9

Laziness in Racket

CONS - We want it to be lazy
How can we get laziness in an eager language?

```
(lambda () <expr>) --> promise
```

to force our promise to evaluate we just write `(promise)`

```
(delay (+ 2 3))
```

`define-syntax-rule` makes a function that operates on program text. It's a way of providing syntactic sugar.

Hamming Code

Hamming's Problem
Produce a list of numbers only divisible by the prime numbers 3, 5, 7 and no other prime number in order 1 3 5 7 9 15 21 25 27 35

Georg Contor

The hotel with an infinite number of rooms is full.

-	1	2	3	4
0	/	/	/	/
1	/	/	/	
2	/	/		
3	/			

What is our interpreter now going to look like?
We have function values. So we need a different way of naming functions.
The only way we now have of creating names is as function parameters.
This is cumbersome to write in the target language.
Why don't we use the syntactic sugar of a Racket style let statement?

```
(let
   <def. list>
   expr)

<def. list> --> ((name1 expr1)
                 (name2 expr2)
                 (namen exprn)
                 ...)

(let <def. list> expr) --> (let ((name1 expr1))
                              (let ((name2 expr2))
                                 (let ...
                                    expr)))
```

desugar2

`(let (name exp) program)` $\xrightarrow{desugar}$ `((lambda name program) expr)`

Grammar for our target language:

S → identifier

S → (S S)

S → (λ identifier S)

S → (let (identifier S) S)

S → number

S → (+ S S) ...

When we want multi-parameter functions we can also do this by desugaring.

`(lambda (` v_1 v_2 v_3 `expr)` $\xrightarrow{desugar}$ `(lambda` v_1 `(lambda` v_2 `(lambda` v_3 ... `expr))))`

`(a b c d)` $\xrightarrow{desugar}$ `(((a b) c) d)`

λ-calculus

$((I\ M)\ N) \rightarrow (M\ N)$

exponentiation = identity function

The predecessor function is hard.

`(PRED n)` what we want it `n-1`

Think of a pipeline

Let us program this transformation on pairs.

$[a, b] \rightarrow [b, (SUCC\ \mathrm{B})]$

```
(lambda n
   (FIRST ((n (lambda c
                 ((CONS (REST c)) (SUCC (REST c))))))
                 ((CONS ZERO) ZERO)))
```

This is a `PRED` function.

Observation

(λ x x)	≡	(λ y y) (α transformation)
(λ x (λ y y))	≡	(λ a (λ b b)) (α transformation)
(λ x (λ y x))	≠	(λ a (λ b b))

What is the normal form?
What is the essential difference?

de Bruijn indices

(λ x x)	→	(λ 1)
(λ (λ 1))	→	(λ (λ 1))
(λ (λ 2))	→	(λ (λ 1))

WEEK 10

HASKELL

1 Functional - function values are first class values
no assignment, no side effects

2 Lazy - example: all lists can be infinitive. No special treatment is needed for infinitive lists

3 Sophisticated Type System - That allow generic programming
variable types

4 Lots of interesting sugar

> Java is not the solution
> Java is the problem.

β Transformations

Why did we object to identifier capture?
$(\lambda\ m\ (\lambda\ n\ (((\lambda\ x\ (\lambda\ y\ (x\ y)))\ m)\ n)))\ y)\ x)$
$(\lambda\ m\ (\lambda\ n\ (((\lambda\ x\ (\lambda\ y\ (x\ y)))\ m)\ n)))\ y)\ x)$
If we have identifier capture the result of the β transformations depends on the order they are done in.

Is it the case that our definition of β transformation (including the substitution rule) avoids the problem?

The Curch-Rosser Property aka the "Diamond Property"

If a λ-sentence M can be transformed by chains of β transformations into sentences M_1 and M_2, then there exist (possibly empty) chains of β transformations that transform M and M_2 into some sentence M_3

A normal form: is a λ sentence in which no β transformation is possible.
If Church-Rosser applies every sentence has at most one normal form.
Proof: Suppose M has 2 normal forms, M_1 and M_2. But Church-Rosser tells us that there are chains of β transformations that transform M_1 M_2 into M_3. But M_1 M_2 are normal forms, so these chains must be empty. So $M_1 = M_2 = M_3$

$((\lambda\ x\ y)\ ((\lambda\ x\ (x\ x))\ (\lambda\ x\ (x\ x))))$

WEEK 11

Use map to write Haskell function that multiplies every element of a possibly infinite list of integers by n.

```
mulList :: Integer -> [Integer] -> [Integer]
mulList n l = map ((*) n) l
```

The signature of map?

```
map :: (x -> y) -> [x] -> [y]
```

Type signature of *?

```
Integer -> Integer -> Integer
```

Exercise 3.59 in SICP

Infinite series
`S =` $a_0 + a_1x + a_2x^2 + a_3x^3 + ...$
treat an infinite series as an infinite list $[a_0, a_1]$

Consider $\int_0^x Sdx$ can we write an integrate function?

Contract `[Number] -> [Number]`

$$\int_0^x (a_0 + a_1x + a_2x^2 + a_3x^3 + ...)dx = a_0x + \frac{a_1x^2}{2} + \frac{a_2x^3}{3} + ...$$

```
integrate s = 1 : zipWith (/) S [1,2..]
```

zipWith signature
`(a -> b -> c) -> [a] -> [b] -> [c]`

Consider a function such that $\int f(x)dx = f(x)$

```
funny = integrate funny

powers x = 1 : map(* x) (powers x)
```

The most beautiful function

```
(LET ((x (lambda x x)) (succ (lambda n (lambda f (lambda x (f ((n f) x))))))
       <body>)


(LETn (v <expr>) <body>)  ==>  ((lambda v <body>) <expr>)
```

we want to write factorial

```
(LET (fac (lambda n (((IF (ZERO ? n)) ONE) (fac (PRED n))))
```

all defined in LETs desugar

```
((lambda fac (fac SEVEN)) (lambda n ((IF (ZERO ? N) ONE) (fac (PRED n))))
```

Our factorial function needs to look like this

```
(lambda fac (lambda n .. fac ..))
```

Then we need to make a closure of fac applied to itself.

```
((lambda f ((lambda x (f (x x))) (lambda x (f (x x))))) F)
```

$\downarrow\beta$

```
((lambda x (F (x x)) (lambda x (F (x x)))) ≡ (YF)
```

$\downarrow\beta$

```
(F ((lambda x (F (x x)) (lambda x (F (x x)))) ≡ F(YF)
```

The Y combinator

WEEK 12

Haskell

- has typing
- and type interface

What is the right hand side cosSeries?
It is a list comprehension

The aim is;

$$1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ...$$

```
facs = 1 : zipWith (*) facs[2,3,..]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Eratosthenes Sieve

```
isPrime :: [Integer] Integer -> Bool
isPrime lp n
         | (head lp) ^ 2 > n = True
         | mod n (head lp) == 0 False
         | otherwise = isPrime(tail lp) n


primes = 2 : (filter (isPrime primes) [3,5...])
```

Type Signatures

```
isPrime :: [Integer] -> Integer -> Bool

filter :: (a -> Bool) -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
foldr :: (a-> b -> b) -> b -> [a] -> b

sum = foldr (+) 0
product = foldr (*) 1
```

Leveraging the Haskell type system - polynomial arithmetic.
$(a_0 + a_1x + a_2x^2 + ... + a_nx^n) \leftarrow$ polynomial in 1 variable.
represent as a list $[a_0, a_1, ..a_n]$
$[a_0, a_1, ..a_n] + [b_0, b_1, ..b_m] = [a_0 + b_0, a_1 + b_1, ..]$
$[a_0, a_1, ..] * [b_0, b_1, ..b_m] = [a_0 * b_0, a_1 * b_0 + a_0 * b_1, ..a_m * b_n]$

WEEK 13

Memory Management

In Java we can allocate memory with new keyword.

```
int[] a = new int[n];
```

In C/C++ we have malloc and free.

If we have manual memory management;

(a) We can have a memory leak. (forget to free)

(b) We can have a catastrophe. (free memory still in use)

Even this is not cheap.
Especially if we think about fragmentation.

```
+-------------------------------------------------+
|      | free |      | free |      | free |      |
+-------------------------------------------------+
```

Manual is not a good idea. - Leads to problems in practice.
We want an automatic system.
It needs to detect when allocated memory can be handed back.

- We want a system that is sound. (does not hand back free memory)
- We want a system that is complete. (hands back all free memory)

Solution 1 (PHP, Swift)

Reference Counts

Increase for each new reference to a piece of memory,
decrease when a reference is deleted or overwritten.
When the count drops to zero hand back the memory.

```
class x {
   .. x a = this;    <-- Circular Reference
}
```

In this example, there is no way to fall zero! It causes memory leak.

Solution 2 (Java, C#, Racket, Haskell, ..)

Garbage Collection

Find the memory in use starting from base references,

Mark the memory in use,

Free all the rest.

Cheney (1970)

Divide memory into two,

Use A for allocating memory,

When A is "exhausted"

Copy referenced memory from A to B

Mark the old copies with "broken hearts"

Exploit

- Short term allocation
- Long term allocation

Generational Garbage Collection

```
(define (substitute [to-sub : lambda] [for : symbol] [in : lambda]) : lambda
     (type-case lambda in
          [lambda-id (id) (if (symbol=? for id) to-sub in)]
          [lambda-apply (lhs rhs) (lambda-apply (substitute to-sub for lhs)
                                                (substitute to-sub for rhs))]
          [lambda-def (bound body) (if (symbol=? for bound) in
                                       (if (or (not (set-member for (free-identifiers body)))
                                               (not (set-member bound (free-identifiers to-sub)))
                                       (lambda-def bound (substitute to-sub in body)))))]
```

WEEK 14

```
(LET ((MUL (lambda m (lambda m (lambda f (lambda x (m (n x))))))) body)

(LET ((id1 bv1) (id2 bv2) ...) <body>)

(LET (id bv) <body>) --> ((lambda id <body>) bv)

(LET (id1 bv1) (LET (id2 bv2) (LET ...) <body>)
```

this is a fold.

Quiz question answer:

```
(LET ((a (b x)) (b c)) (lambda b (c a)))

(LET ((a (b x))) (LET ((b c)) (lambda b (c a))))

((lambda a ((lambda b (lambda b (c a))) c))) (b x)))
```

To get a practical lambda calculus interpreter we need to always do the leftmost β transformation.
Example: Y combinator will loop if you don't always do the leftmost.

Consider $(\Lambda_1\ \Lambda_2)\ \Lambda_1$ is not a def
So we transform Λ_1 - if after any transformation Λ_1 becomes a def we need to apply it immediately.

Mutation + Sequence

```
a = b;                          c = a;
            not equal to
c = a;                          a = b;
```

Sequence matters!

in Racket;

```
(begin
    a1
    a2
    a3
    a4)
```

memory is a key-value store
make every value a pair mem x value
`(a4 (a3 (a2 (a1 x))))` A general solver for polynomial equations of this form

$a_0 + a_1x + a_2x^2 + ... + a_nx^n = 0$
Present polynomial as array of numbers
$(1 + 2x + 3x^2)(4 + 2x) = 4 + 10x + 16x^2 + 6x^3$
evaluate $a_0 + a_1x + ... + a_nx^n = a_0 + x(a_1 + x(a_2...a_n)))))$
$1 + 2x + 3x^2 + 4x^3$, x=2 $\rightarrow$ 49
Make the polynomial whose roots are $r_0, r_1, r_2, ...$
$(x - r_0)(x - r_1)(x - r_2)...(x - r_n)$
$(-r_0 + x)(-r_1 + x)(-r_2 + x)...(-r_n + x)$
$(x - 1)(x - 2)(x - 3)(x - 4)$
$x^4 - 10x^3 + 35x^2 - 50 + 24$
Need to differentiate a polynomial
$a_0 + a_1x + a_2x^2... \rightarrow a_1 + 2a_2x + 3a_3x^2 + ... + na_nx^{n-1}$