Programming Language Design & Semantics

- Scheme
  - Syntax
  - Sample Code
    - Addition
    - Factorial
    - Fibonacci Sequence
    - Append one list to another
    - Absolute Value of a number
    - Sum of all list elements
    - Replace element in a list
- Haskell
  - Lists
  - Types
  - Pattern Matching
    - Lists
    - As-patterns
  - Sample Code
    - Fibonacci Sequence
    - Remove Non Uppercase Characters
    - Factorial
    - Get the head of a list
    - Get the length of a list
    - Add the elements of a list
    - Get the larger of two items
    - Reverse a list
    - Take a Num and repeat it Num times
    - Check if item is in list
    - Sort a list
    - Reading arguments from prompt
- Prolog
  - Sample Code
- Lambda Calculus
  - Syntax
  - Church Numerals
  - Arithmetic
    - Addition
    - Multiplication
    - Exponentiation
    - Predecessor
    - Subtraction
    - Is Zero?

        - Boolean Operators
        - Pairs
    - Video Lectures
- General Concepts
    - Currying
    - Types
        - Role of types
        - Type Safety
        - Type Inference
        - Subtyping
    - Definitions:

Scheme

A toy language is a term for a computer programming language that is not considered to fulfill the robustness or completeness requirement of a computer programming language. Some people would argue today that Scheme is a toy language, as it is mostly used in academia.

Functional programming language, dialect of LISP. Syntax is based on s-expressions.

Syntax

Functions in Scheme are written as:
(function arg1 arg2 arg3 argn)   where n is the number of arguments.

An "s-expression" or "symbolic expression" aka "sexpr" is:
<sexpr> can be:
<atom>, ie, (number or symbol, ... and maybe function object, boolean, whatever)
<list> is:
(<sexpr>...)
<symbol> is:
non-empty sequence of chars from set a..z (not case sensitive)
0..9, +, -, _, :, ., =, <, >, $, @, ...
WHICH CANNOT BE INTERPRETED as a NUMBER, and which does not start with #. Also "." is not allowed.

Lists

A list can be created by using:
(list (1 2 3 4))
or similarly:
‘(1 2 3 4)

car - returns the first element(head) in the list
(car ‘(add 1 2))
-> add

cdr - returns the tail of the list
(cdr ‘(add 1 2))
-> (1 2)

cadr - returns the car of the cdr of the list, i.e. the second element
(cadr ‘(add 1 2))
-> 1

Sample Code

Addition

```
        (define add
                (lambda (x y)
                        (+ x y)))
```

can also be written as:

```
        (define
                (add x y)
                (+ x y))
```

At runtime:

```
        -> (add 4 5)
        -> 9
```

Factorial

```
        (define factorial
                (lambda (n)
                        (if (= n 0)
                        1
                        (* n (factorial (- n 1))))))
```

At runtime:

```
        -> (factorial 4)
        -> 24
```

Fibonacci Sequence

```
(define (fib n)         ;;Note: (fib n) is syntactic sugar for the lambda thing! Works the same!
        (if (<= n 2) 1  ;;Note2:  Slow version of fib!
        (+ (fib (- n 1))
           (fib (- n 2)))))
```

Append one list to another

```
(define (my-append x y)
        (if (null? x) y
        (cons (car x)
              (my-append (cdr x) y))))
```

Absolute Value of a number

```
(define (absolute-value x)
        (if (< x 0) (- x)
           x))
```

Sum of all list elements

```
(define (sum-of-list-elements lst)
          (if (= 1 (length lst))
             (car lst)
             (+ (car lst)
                (sum-of-list-elements (cdr lst)))))

(define (my-list-ref lst n)
          (if (zero? n)
             (car lst)
             (my-list-ref (cdr lst) (- n 1))))

;;; Map : Which is like (my-map + (1,2,3,4,5)) and adds them all up! so the answer would be 15
(define (my-map fn lst)
          (if (null? lst) null
             (cons (fn (car lst))
                     (my-map fn (cdr lst)))))
```

Replace element in a list

```
(define (replace lst find repl)
          (if (pair? lst)
             (cons
                  (replace (car lst) find repl)
                  (replace (cdr lst) find repl)
             (if (equal? find lst)
                repl
                lst)))
```

Haskell

Lists

A list can be created by using

    [1,2,3,4,5]

or similarly

    1:2:3:4:5:[ ]

head takes a list and returns its head,

```
->head [1,2,3,4,5]
1
```

tail takes a list and returns its tail

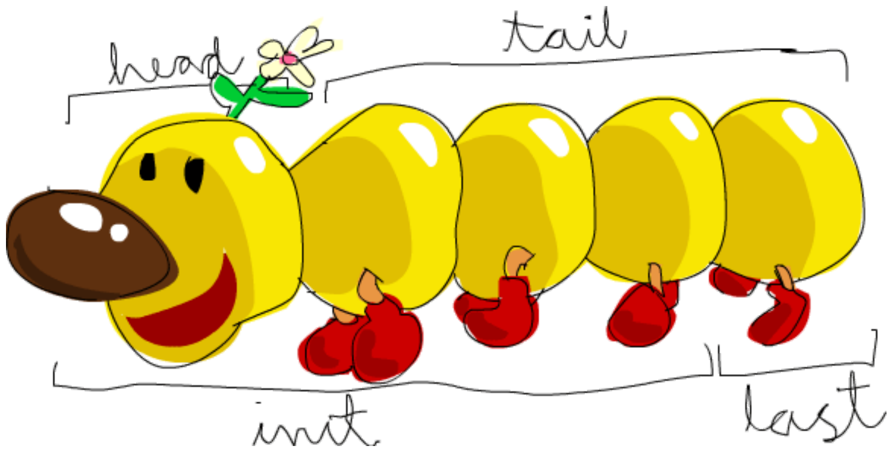
```
->tail [1,2,3,4,5]
2,3,4,5
```

last takes a list a returns its last element

```
->last [1,2,3,4,5]
5
```

init takes a list and returns everything except the last element

```
->init [1,2,3,4,5]
1,2,3,4
```



Types

In Haskell, we declare types for functions as:

someFunction :: someType -> someOtherType

This means for someFunction, it takes a variable of type someType and returns a variable of type someOtherType.

Pattern Matching

Lists

Patterns in Haskell are defined in functions as, for example, in the function *tell*, which takes a list and depending on the list returns a different string:

```
tell :: (Show a) -> [a] -> String
tell [] = “The argument given is an empty list”
tell (x:[]) = “The argument passed to this function is a list with one element:” ++ show x
tell (x:y:[]) = “The arg passed to this function is a list with two elements:” ++ show x ++ “
and ”++ show y
tell (x:y:_ )= “This list is longer than two elements”

->tell [1]
“The argument passed to this function is a list with one element: 1”
->tell [a, b]
“The arg passed to this function is a list with two elements: a and b”
->tell []
“The argument given is an empty list”
->tell [alpha, beta, delta, gamma]
“This list is longer than two elements”
```

As-patterns

These allow you to break up an item according to a pattern, just precede a regular pattern by the @ symbol, after the argument:

```
firstLetter :: String -> String
firstLetter “” = “That’s an empty string!”
firstLetter str@(x:xs) = “The first letter of “ ++ str ++ “ is “ ++ [x]
```

NOTE: Remember, a String is just a list of Chars!

```
->firstLetter “Barak”
“The first letter of Barak is B”
```

Sample Code

Fibonacci Sequence

```
fib1 = fib
        where
                fib 0 = 1
                fib 1 = 1
                fib n = fib(n-1) + fib(n-2)
```

Remove Non Uppercase Characters

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [c | c <- st, elem c [‘A’..’Z’]]
```

Factorial

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Get the head of a list

```
head :: [a] -> a
head [] = error “Can’t take head of empty list”
head(x:_) = x
```

Get the length of a list

```
length :: (Num b) => [a] -> b
length [] = 0
length (_:xs) = 1 + length xs
```

Add the elements of a list

```
sum :: (Num a) => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

Get the larger of two items

```
max :: (Ord a) => a -> a -> a
max a b
        | a > b = a
        | otherwise = b

;;;Check if its GT (Greater than) EQ (Equal to) or LT (Less than)
;;; GT EQ and LT are base Ordering types in Haskel
myCompare :: (Ord a) => a -> a -> Ordering
myCompare a b
        | a > b = GT
        | a == b = EQ
        | otherwise = LT

;;;Returns the maximum value in a list
maximum :: (Ord a) => [a] -> a
maximum [] = error “empty list”
maximum [x] = x
maximum (x:xs) = max x (maximum xs)
```

Reverse a list

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Take a Num and repeat it Num times

```
;;; It accepts a Num and returns a list:
;;;eg repeat 5 = (5,5,5,5,5) or repeat 2 = (2,2)
repeat :: a -> [a]
repeat x = x:repeat x
```

Check if item is in list

```
elem :: (Eq a) => a -> [a] -> Bool
elem a [] = False
elem a (x:xs)
        | a == x = True
        | otherwise = elem a xs
```

Sort a list

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs)
        let smallsort = quicksort [a | a <- xs, a <= x]
            bigsort = quicksort [a | a <- xs, a>x]
        in smallsort ++[x] ++bigsort

;;;Implementation of zipWith function in Haskel
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

;;;Implementation of map function in Haskel
myMap :: (a -> b) -> [a] -> [b]
myMap _ [] = []
myMap f (x:xs) = f x : myMap f xs
```

Reading arguments from prompt

```
;;;This is just an example to take arguments in etc…. found cool so decided to add in I guess…
main = do
        putStrLn “What your first name?”
        firstName <- getLine
        putStrLn “What is your last name?”
```

```
lastName <- getLine
let bigLastName = map toUpper lastName
     bigFirstName = map toUpper firstName
putStrLn $ “Hey “ ++bigFirstName ++” “ ++bigLastName ++”, how you duin? *wink*”
```

[illegible]

Prolog

Prolog is a general purpose logic programming language. It consists of rules, queries and facts. Facts are declared in Prolog as follows:

```
        man(barak).
        man(john).
```

Logic Variables are written using uppercase characters. It we run the code above and query:

```
        man(X).
```

Prolog will try and find a suitable value for X:

```
        X = barak ?
        X = john ?
```

I was using this site to learn bits of prolog and actually found it very useful to understand theorems and concepts, it takes a while if you go through each step but no need for that, it could get you out of a bad situation, few marks at least.. http://www.learnprolognow.org/

Sample Code

Sample 1.

```
%Sorry guys, the code below is called KB2 and this is the “knowledge base”
happy(yolanda).
listensToMusic(mia).
listensToMusic(yolanda) :- happy(yolanda).
playsAirGuitar(mia) :- listensToMusic(mia).
playsAirGuitar(yolanda) :- listensToMusic(mia).

% happy is a fact, aswell is listensToMusic
% the last three items it contains are rules
% the ":-" should be read as "if" or "is implied by"
% the part to the left hand side of the rule is the head and to the right, the body
% Any fact produced by an application of modus ponens can be used as input to further rules
% the facts and rules contained in a knowledge base are called clauses.
% Thus, KB2 contains five clauses, namely three rules and two facts.
% Another way to look at it is that KB2 contains three predicates, which are:
%       - listensToMusic
%       - happy (defined using a single clause)
%       - playsAirGuiter
% listensToMusic and playsAirGuiter are defined using two clauses (two rules and one rule and
a fact).
% Atoms
% 1. A string made up of upper-case letters, lower-case letters, digits, and underscore
% 2. A sequence of characters enclosed in quotes
```

% 3. A string of special characters

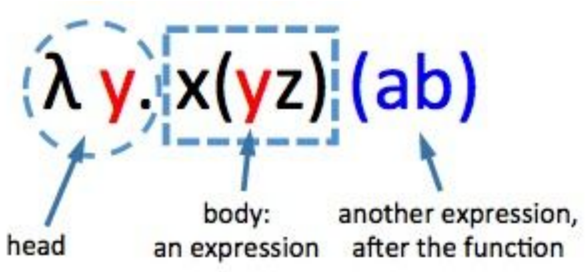
% Numbers - Integers

Lambda Calculus

Lambda calculus is a formal system for expressing computation through function abstraction and application using variable binding and substitution.

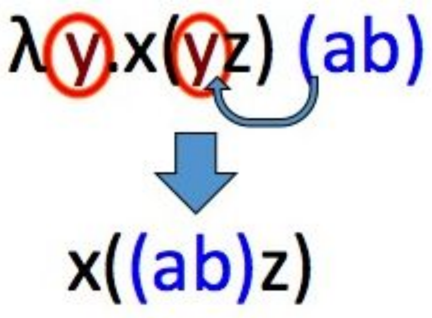
Syntax

Single letters are in LC are variables, with λ used to denote functions.



A function.

The lambda tells us what variable to replace in our function with, for example, (ab) above.



Functions such as:	λx.λy.xzy
can be abbreviated as:	λxy.xzy

Church Numerals

Numbers in LC are represented as Church Numerals as follows:
0 := λf.λx.x
1 := λf.λx.f x
2 := λf.λx.f (f x)
3 := λf.λx.f (f (f x))

Arithmetic

Addition

The addition function **plus**(*m*,*n*)=*m*+*n* uses the identity $f^{(m+n)}(x) = f^m(f^n(x))$.

plus ≡ λm.λn.λf.λx. m f (n f x)

plus 1 2 = (λmnfx.mf(nfx)) (λfx.f(x)) (λfx.f(fx))
=λfx.λfx.f(x)f(λfx.f(fx))fx)
=λfx.λfx.f(x)f(f(fx))
=λfx.f(f(fx))

plus = λm.λn. iter n (succ m)

if we take an example plus (2 4) we get:

m = 2, n = 4

→ iter 4 (succ 2)
which means apply succ 4 times to 2
which is 6

Multiplication

The multiplication function **mult**(*m*,*n*)=*m***n* uses the identity $f^{(m*n)}(x) = (f^n)^m(x)$.

mult ≡ λm.λn.λf. m (n f)

mult 1 2 = λmnf.m(nf)(λfx.f(x) (λfx.f(f(x)))
= λf.λfx.f(x)(λfx.f(f(x)) f)
= λf.(λfx.f(x))(λx.f(f(x)))
=λf.(λx.(λx.f(f(x))(x))
=λf.(λx.f(f(x))

Exponentiation

The exponentiation function **exp**(*m*,*n*)=*m*^*n* is straightforward by the definition of Church numerals.

exp ≡ λm.λn. n m

Predecessor

The predecessor function:

pred ≡ λn.λf.λx. n (λg.λh. h (g f)) (λu. x) (λu. u)

Subtraction

The subtraction function can be written based on the predecessor function.

sub ≡ λm.λn. (n **pred**) m

Is Zero?

The zero predicate can be written as:

zero? ≡ λn. n (λx.F) T

```
zero? 0 = (λn.n(λx.λa.λb.b)λa.λb.a) (λfx.x)
         = (λfx.x) (λx.λa.λb.b) λa.λb.a
         = (λx.x)λa.λb.a
         = λa.λb.a
         =T
```

```
zero? 1 = (λn.n(λx.λa.λb.b)λa.λb.a) (λfx.f(x))
         = (λfx.f(x)) (λx.λa.λb.b) λa.λb.a
         = (λx.(λx.λa.λb.b)(x)) λa.λb.a
         = (λx.λa.λb.b)(λa.λb.a)
         = (λa.λb.b)
         = F
```

Boolean Operators

Booleans:

true ≡ λa.λb. a
false ≡ λa.λb. b

and ≡ λm.λn. m n m
or ≡ λm.λn. m m n
not1[1] ≡ λm.λa.λb. m b a
not2[2] ≡ λm. m (λa.λb. b) (λa.λb. a)
xor ≡ λa.λb. a (**not**2 b) b
if ≡ λm.λa.λb. m a b

Pairs

Pairs:

pair ≡ λx.λy.λz.z x y
fst ≡ λp.p (λx.λy.x)
snd ≡ λp.p (λx.λy.y)

An example:

fst (**pair** a b) ≡ (λp.p (λx.λy.x)) ((λx.λy.λz.z x y) a b) ≡ (λp.p (λx.λy.x)) (λz.z a b) ≡ (λz.z a b) (λx.λy.x)
≡ (λx.λy.x) a b ≡ a

Video Lectures

Basics: http://www.youtube.com/watch?v=S_WzF6BHadc

General Concepts

Comparisons between Scheme, Haskell, Prolog

(Taken from the board in the lab)

Haskell	Scheme	Prolog
pure	impure	impure
lazy	eager	lazy
static typing	dynamic typing	dynamic typing
Polymorphic typed LC	Untyped LC	
Curried	Non-curried	
Monads	Macros	Unification
Algebraic datatypes	Explicit accessors	Structure Objects Simple Objects -variables -constants -numbers -atoms
Compiler(with interpreter)	Interpreter(with compiler)	Interpreter(with compiler)
Highly sugared	Minimal sugar	Minimal sugar

Currying

Currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument

- Currying
  - Given a function (x,y) which takes a pair (v,w) and returns s when x is mapped to v, and y is mapped to w.
  - Applying the function gives fvw, which reduces to [y|->w] [x|->v]s.

Currying is transforming multi argument functions to higher order functions

- Inference Rule
- is the act of drawing a conclusion based on the form of the premises
- usually only rules that are recursive are important.

- usually takes form of:
Premise#1
Premise#2
...
Premise#n
Conclusion

Types

Role of types

Types exist so that developers can represent entities from the real world in their programs. However, types are not the entities that they represent. For instance, the integer type, in Java, represents numbers ranging from -231 to 231 - 1. Larger numbers cannot be represented. If we try to assign, say, 231 to an integer in Java, then we get back -231. This happens because Java only allows us to represent the 31 least bits of any binary integer.

Types are useful in many different ways. Testimony of this importance is the fact that today virtually every programming language uses types, be it statically, be it at runtime. Among the many facts that contribute to make types so important, we mention:

- Efficiency: because different types can be represented in different ways, the runtime environment can choose the most efficient alternative for each representations.
- Correctness: types prevent the program from entering into undefined states. For instance, if the result of adding an integer and a floating point number is undefined, then the runtime environment can trigger an exception whenever this operation might happen.
- Documentation: types are a form of documentation. For instance, if a programmer knows that a given variable is an integer, then he or she knows a lot about it. The programmer knows, for example, that this variable can be the target of arithmetic operations. The programmer also knows much memory is necessary to allocate that variable. Furthermore, contrary to simple comments, that mean nothing to the compiler, types are a form of documentation that the compiler can check.

Type Safety

Type safety requires progress and type preservation to be maintained is the extent to which a programming language discourages or prevents type errors.
Type safety prevent illegal operations (e.g. divide an int by a string)

Type Inference

Type inference refers to the automatic deduction of the type of an expression in a programming language.

aSubtyping

In programming language theory, subtyping (also subtype polymorphism or inclusion polymorphism) is a form of type polymorphism in which a subtype is a datatype that is related to another datatype (the supertype) by some notion of substitutability.

- covariant: converting from a specialized type (Cats) to a more general type (Animals): Every cat is an animal.
- contravariant: converting from a general type (Shapes) to a more specialized type (Rectangles): Is this shape a rectangle?

Width and depth subtyping are 2 ways of obtaining a new type of record which allows the same operation as the original record type

Width subtyping
Add more fields to record
Formally: every field in supertype will appear in subtype hence any operation possible on the supertype is possible on the subtype

Depth subtyping
Replace fields with their subtype
The fields in the subtype are subtypes of fields in the supertype
Only makes sense for immutable records (state cannot be changed once it is created)

Evaluation rules provide the operational semantics of the terms in our language, describing how a term is evaluated to a value.

Typing rules provide a set of inference rules assigning types to terms. A term t is well typed if there is some T such that t:T.

Typing rules question example:
 ---- prove using typing rules:
---- if isZero {succ 0} else {pred succ 0:Nat}

The basic element of type-checking is the typing relation written “t : T” meaning that (term) t has type T.
--- A term t is well-typed if t : T for some type T
 (a term t is well-typed if t has a type T for some type T)

Safety = Progress + Preservation

(Progress means that a well typed term is never “stuck”: either it is a value or it is reducib

(Preservation means that if a well-typed term is reduced then the result is also well-typed)

i.e.

If e : T and e → e' then e' : T
(first part is the preservation, second part is the progress)

preservation, says that the steps of evaluation pre- serve typing; the second, called *progress*, ensures that well-typed expressions are either values or can be further evaluated. Safety is the conjunction of preservation and progress.

We say that an expression, *e*, is *stuck* iff it is not a value, yet there is no *e′* such that *e* □→ *e′*. It follows from the safety theorem that a stuck state is

The progress theorem ensures that well-typed expressions do not "get stuck" in an ill-defined state, and the preservation theorem ensures that if a step is taken, the result remains well-typed (with the same type). Thus the two parts work hand-in-hand to ensure that the statics and dynamics are coherent, and that no ill-defined states can ever be encountered while evaluating a well-typed expression.

Fun fact about Type Safety
Type safety is usually a requirement for any toy language proposed in academic programming language research.
A toy language is a term for a computer programming language that is not considered to fulfill the robustness or completeness requirement of a computer programming language. Some people would argue today that Scheme is a toy language, as it is mostly used in academia.

Definitions

- Axiomatic
  - program logic which provides a tool for derivation of assertions of the form {precondition}statement{postcondition}
    - Advantages: Focus attention on reasoning about programs (invariants)
- Operational
  - describes behaviour of language by defining an abstract machine for it (INTERPRETER)
    - Advantages: Simple and flexible.
- Denotational
  - Maps program directly to its meaning using mathematical notation such as numbers/functions (NOT INTERPRETER)
    Advantages: Highlights essential concepts of the language.

- Call-by-name
  - No reductions inside abstractions
    - example

(λx. x+1) 5+2

λx. 5+2+1

` 8

Evaluation Rules

t1->t1’

________ (E-APP1)

t1t2->t1’t2

(λx.t1)t2 -> [x|->t2]t1 (E-APPABS) // Application Abstraction

- call-by-value
    - Only outermost redexes may be reduced. A redex may only be reduced if its right hand side has been reduced to a value.
        - Example

          (λx. x+1) 5+2

          (λx. x+1)7

          7+1

          8

    Evaluation Rules

    t1->t1’

    ____________ (E-APP1)

    t1t2-> t1’t2

    t2->t2’

    ____________ (E-APP2)

    vt2-> vt2’

    (λxt1)v2 ->[x|->v2]t1 (E-APPABS)

- Type safety = Progress+Preservation
    - Progress means no well-typed terms are “stuck”
    - Preservation means that if a well-typed term undergoes an evaluation step the result is also well typed.

- Fixed point operators
    - Point that maps to itself by a function
    - fix= λf.(λx.f(λyxxy)(λx.f(λyxxy))
    - eg: f(fix f) = fix f
- Type checking
    - Advantages:
        - Detect programming errors early
        - Documentation
        - Efficiency