Advanced Topics in Programming Languages **Spring Semester, 2012**

Lecture 3: March 20, 2012

Lecturer: Mooly Sagiv *Scribe: Tomer Weiss and Yotam Feldman*

3.1 Lambda Calculus

3.1.1 Introduction to the Untyped Lambda Calculus

Lambda calculus is a mathematical system that illustrates some important programming lanuage concepts. Lambda calculus is a computational model, just like a Turing Machine. Alonzo Church developed Lambda Calculus in the 1930's, as a theory of functions that provides rules for manipulating functions in a purely syntactic manner. Church proposed an important principle, called The *Church Thesis*, and provided proof that Lambda Calculus and Turing Machines are equivalent, hence Lambda Calculus is Turing-complete.

It is common to think of Lambda calculus as a kind of assembler for programming languages. In the untyped Lambda calculus, function application has no restrictions (so the notion of the domain of a function is not built into the system). There are also typed versions of lambda calculus, but the additional type-checking constraints rule out certain forms of expressions.

Untyped Lambda calculus consists of 3 basic types of terms, $t ::=$

- variables x
- abstraction $\lambda x.\ t$
- application $t\ t$

and one type of value, $v ::=$

- abstraction value $\lambda x.\ t$

Example Some examples of lambda terms:

- $\lambda x.\ x$ a lambda abstraction called the identity function
- $\lambda x.\ (f\ (g\ x))$ another lambda abstraction
- $(\lambda x.\ x)\ (\lambda s.\ \lambda z.\ s\ z)$ an application

The syntactic convention is for application to associate from left to right, because it is thought to be more natural for functions, and also decreases the use of parentheses. In addition, the convention is that the body of the abstraction (an expression containing a λ) extends as far to the right as possible. For example, $\lambda x.\ x\ y$ should be read as $\lambda x.\ (x\ y)$, not $(\lambda x.\ x)\ y$.

3.1.2 Variables

We distinguish between free and bound variables. x is said to be *bound* when it occurs in the body M of an abstraction, for example in $\lambda x.\ M$, we can say that λx is a *binder* whose scope is M. An occurrence of x is *free* if it appears in a position where it is not bound by an enclosing abstraction on x. For example, the occurrences of x in $x\ y$ and $\lambda y.\ x\ y$ are free. It is possible to change the name of a bound varibale x to another name, say y. This means the variable x is like a placeholder, and the meaning of $\lambda x.\ M$ does not depend on x.

3.1.3 Operational semantics

We denote by $[x \mapsto t]S$ the replacement of all the unbound appearences of the variable x with t in S.

An expression of the form $(\lambda x.\ S)\ T$ where x is a variable and S, T are lambda expressions is called a redex (reducible expression). An expression that does not contain any redexes is called irreducible.

The semantics is defined by the β-reduction rule. β-reduction is an operation that transforms a redex to another lambda expression by plugging in the parameter T into S instead of x. This can be expressed by the substitution:

$$(\lambda x.\ S)\ T \to [x \mapsto T]S$$

Note of caution: we should be careful not to turn a free variable to a bound one, or otherwise, and keep the scope rules.

3.1.4 Evaluation order

Here we deal with a semantic question: how do we evaluate the expression tree, in which order. There are a number of possible schemes:

1. Normal order: We peel the expression from the outside, left-to-right.

   let us denote *id* to be the identity function: $\lambda x.x$

   $\underline{\text{id (id } (\lambda\text{z.id z))}} \to \underline{\text{id } (\lambda\text{z.id z)}} \to \lambda\text{z.}\underline{\text{id z}} \to \lambda\text{z. z}$

2. Call-by-name: Same peeling of expression evaluation, from left-to-right, but with lazy evaluation. We don't evaluate terms inside a λ abstraction.

   $\underline{\text{id (id } (\lambda\text{z.id z))}} \rightarrow \underline{\text{id } (\lambda\text{z.id z)}} \rightarrow \lambda\text{z.}\underline{\text{id z}}$

3. Call-by-value: Only outermost redexes are reduced, and a redex is reduced only when its right-hand side has already been reduced to a value.

   $\underline{\text{id (id } (\lambda\text{z.id z))}} \rightarrow \text{ id } \underline{(\text{id } (\lambda\text{z.id z))}} \rightarrow \underline{\text{id } (\lambda\text{z.id z)}} \rightarrow \lambda\text{z.id z}$

In the above example, all strategies ended up with the same lambda expression. It is, of course, not a coincidence. If call-by-name and call-by-value can be reduced to an irreducible expression, also called "β-normal form", then that form is the same up to α-reduction (that is, changing the names of bound variables). In fact, the Church-Rosser theorem states that the "confluence property" (or "diamond property") holds for the lambda calculus: any order of β-reductions which terminates with an irreducible expression leads to the same lambda expression (up to α-reduction).

However, it is not always the case that both strategies terminate with an irreducible expression: sometimes neither terminates, such is in the *omega* example below. Sometimes call-by-value doesn't terminate whereas call-by-name avoids some sort of unnecessary "infinite loops".

An important example is the *fixed-point combinator* operator as defined for the call-by-name strategy,

$$Y' = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)).$$

(In section 3.1.5 we define a fixed-point combinator for the call-by-value strategy.)

For every lambda abstraction f,

$$\begin{aligned} & Y'\ f = (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ f \\ \rightarrow\ & \underbrace{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}_{R} \\ \rightarrow\ & f\ \Big(\big(\lambda x.\ f\ (x\ x)\big)\ \big(\lambda x.\ f\ (x\ x)\big)\Big) \\ =\ & f\ R \end{aligned}$$

Therefore $Y'\ f \rightarrow^* R \rightarrow^* f\ R \rightarrow^* f\ (f\ R) \rightarrow^* \ldots$ and the evaluation diverges.

3.1.5 Programming in the Lambda Calculus

We observe that the lambda calculus provides no built-in support for multiple function input arguments. We can have only one input for a function. This doesn't mean that we cannot use functions that receive more than one argument as an input. It is possible to achieve the same effect using higher-order functions that yield functions as a result. For example, a function of the following type:

$$f : A \times B \to C$$
$$f = \lambda x, y.\ s$$

can be simulated using:

$$f : A \to B \to C$$
$$f = \lambda x.\ \lambda y.\ s$$
$$f\ v\ w = (f\ v)w = ((\lambda x.\lambda y.s)\ v)w \to (\lambda y.[x \mapsto v]s)w \to [x \mapsto v][y \mapsto w]s$$

The transformation of multi-argument functions into higher-order functions is called *currying*.

We can define booleans in the lambda calculus, these are called Church Booleans:

$$tru = \lambda t.\lambda f.t$$
$$fls = \lambda t.\lambda f.f$$

Following the above, we can define an *if-then-else* clause:

$$test = \lambda l.\lambda m.\lambda n.\ l\ m\ n$$

The test combinator does not actually do much: $test\ b\ v\ w$ just reduces to $b\ v\ w$. The boolean b is a conditional: it takes two arguments and chooses the first one, if it is *tru*, or the second one, if it is *fls*.

We can also implement logical operators, for example:

$$and = \lambda b.\lambda c.\ b\ c fls$$

Example *and fls tru* reduces to *fls*:

$and\ fls\ tru = (\lambda b.\lambda c.\ b\ c\ fls)fls\ tru = (\lambda c.\ fls\ cfls)tru = fls\ tru\ fls =$
$(\lambda t.\lambda f.t)tru\ fls = (\lambda f.tru)fls = tru\ fls = (\lambda t.\lambda f.t)fls = \lambda f.fls = fls$

Similarly for *or* and *not*.

Representing numbers in λ-terms can be done in the following way:

$$C_0 = \lambda s.\ \lambda z.\ z$$
$$C_1 = \lambda s.\ \lambda z.\ s\ z$$
$$C_2 = \lambda s.\ \lambda z.\ s\ (s\ z)$$
$$\ldots$$

Each number n is represented by a combinator C_n (for Church Numeral), that takes two arguments s and z (for successor and zero), and applies s, n times to z.

$$scc = \lambda n.\ \lambda s.\lambda z.s\ (n\ s\ z)$$

Example $scc\ C_0$ reduces to C_1 :

$scc\ C_0 = (\lambda n.\ \lambda s.\lambda z.s\ (n\ s\ z))\ C_0 =$
$\lambda s.\lambda z.\ s\ (C_0\ s\ z) = \lambda s.\lambda z.\ s\ ((\lambda s.\ \lambda z.\ z)\ s\ z) =$
$\lambda s.\lambda z.\ s\ ((\lambda z.\ z)\ z) = \lambda s.\lambda z.\ s\ z \equiv C_1$

The term *scc* is a combinator that takes a Church numeral n and returns another Church numeral, the successor of its argument. The term *plus* can be defined in the following fashion:

$$plus = \lambda m.\ \lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z)$$

Example $plus\ C_1\ C_1$ reduces to C_2:

$plus\ C_1\ C_1 = (\lambda m.\ \lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z))\ C_1\ C_1 =$
$(\lambda n.\lambda s.\lambda z.\ C_1\ s\ (n\ s\ z))\ C_1 = \lambda s.\lambda z.\ C_1\ s\ (C_1\ s\ z) =$
$\lambda s.\ \lambda z.\ (\lambda w.\ \lambda t.\ w\ t)\ s\ ((\lambda f.\ \lambda y.\ f\ y)\ s\ z) =$
$\lambda s.\ \lambda z.\ (\ \lambda t.\ s\ t)\ ((\lambda f.\ \lambda y.\ f\ y)\ s\ z) =$
$\lambda s.\ \lambda z.\ (\ \lambda t.\ s\ t)\ (s\ z) =$
$\lambda s.\ \lambda z.\ s\ (s\ z) \equiv C_2$

We can use plus for the definition of multiplication term:

$$times = \lambda x.\lambda y.\lambda z.\ x\ (y\ z)$$

Example $times\ C_2\ C_2$ reduces to C_4:

$times\ C_2\ C_2 = (\lambda x.\lambda y.\lambda z.\ x\ (y\ z))\ C_2\ C_2 =$
$\lambda z.\ C_2\ (C_2\ z) = \lambda z.\ (\lambda x.\ \lambda y.\ x\ (x\ y))\ ((\lambda w.\ \lambda t.\ w\ (w\ t))\ z) =$
$\lambda z.\ (\lambda x.\ \lambda y.\ x\ (x\ y))\ (\lambda t.\ z\ (z\ t)) =$
$\lambda z.\ (\lambda y.\ (\ (\lambda t.\ z\ (z\ t))\ (\ (\lambda t.\ z\ (z\ t)\ )\ y)\ )\ )\ =$
$\lambda z.\ (\lambda y.\ (\ (\lambda t.\ z\ (z\ t))\ (\ z\ (z\ y)\ )\ )\ )\ =$
$\lambda z.\ (\lambda y.\ (z\ (z\ (\ z\ (z\ y)\ )\ )\ )\ )\ =$
$\lambda s.\ \lambda z.\ s\ (s\ (s\ z))) \equiv C_4$

alternativley, multiplication can be defined as $times = \lambda m.\ \lambda n.\ m\ (plus\ n)\ C_0$, which means "apply the function that adds n to its argument, iterated m times, to zero".

We can simulate recursive functions using the following, called the *fixed-point combinator*:

$Y = (\lambda y.\ (\lambda x.\ y\ (x\ x))(\lambda x.\ y\ (x\ x)))$

This function applied to the function R yields:

$YR = (\lambda x.R\ (x\ x))\ (\lambda x.R\ (x\ x))$

Which further yields:

$R\ ((\lambda x.R\ (x\ x))\ (\lambda x.R\ (x\ x))))$

But this means that $YR = R(YR)$, meaning that the function R is evaluated using the recursive call YR as its first argument.

3.1.6 Enriching the Calculus

We have seen that booleans, numbers and operations on them can be encoded in the pure lambda calculus. However, it is often convenient to include primitive booleans and numbers, or possible other data types as a kind of syntatic sugar, in order to simplfy usage. So we can extend pure λ-calculus by adding a variety of other constructs. This of course does not add to the *power* of the calculus.

3.1.7 Issuses with the Untyped Lambda Calculus

A problem with the untyped lambda calculus is the inability to distinguish between different types of data. For instance, we may want to write a function that only operates on numbers. However, in the untyped lambda calculus, there's no way to prevent our function from being applied to truth values, or strings, for instance. When this happens by accident, the mistakes can be difficult to trace.

Another related issue, is that the representation of *false*(*fls*) and *zero*(C_0) is the same, so there is no way to destinguish between them, except in the relevant context.

Since the Lambda calculus is Turing-complete, there have to be expressions that do not terminate. This means that they do not have a β-normal form and no irreducible expression can be obtained through a finite number of reductions. The expression may even grow indefinitely as reduction progresses. Because of the Turing-completeness, the question whether a Lambda expression has a normal form is undecidable as it is equivalent to the halting problem. For example, the *divergent* combinator:

$omega = (\lambda x.\ x\ x)(\lambda x.\ x\ x)$

$(\lambda x.\ x\ x)(\lambda x.\ x\ x) \rightarrow (\lambda x.\ x\ x)(\lambda x.\ x\ x)(\lambda x.\ x\ x) \rightarrow (\lambda x.\ x\ x)(\lambda x.\ x\ x)(\lambda x.\ x\ x)(\lambda x.\ x\ x) \rightarrow$ ...

omega contains just one redex, and reducing the redex yields *omega* again.

3.1.8 Summary

In summary, λ-calculus is a mathematical system with some syntactic and computational properties of a programming language. There is a general notion for functions that includes a way of treating an expression as a function of some variable that it contains. There is an equational proof system that leads to calculation rules, and these calculation rules are a simple form of symbolic evaluation. Moreover, Every computable function can be represented in pure λ-calculus, so λ-calculus is Turing-complete.

In general, calculations in typed Lambda calculus are unnecessarily convoluted, intricate and impractical. Just like the Turing machine it is merely an abstract model. Applying it for actual computations is not very sensible.

These problems can be solved by introducing native types for Booleans, numbers, lists and so on instead of using the rather pedestrian encodings. If some other modifications are applied, this *syntactic sugar* is added to make the calculus more usable, and the result is a complete functional programming language such as Lisp or Haskell. Because of their tremendously useful properties, Lambda expressions and derived functional concepts have also been introduced to imperative and object-oriented languages such as Python, Scala and indirectly even C++.

3.2 Haskell

3.2.1 Introduction

Haskell is a high-level general-purpose functional programming language, designed by a committee in the 1980's and 1990's to unify research efforts in lazy languages. It is named after the mathematician Haskell B. Curry, who invented the idea of currying (discussed above in lambda calculus).

Haskell is strongly typed and supports automatic type inference, with a rapidly evolving type system. Haskell uses higher-order functions and is purely functional (unlike ML or Scheme, for example).

3.2.2 Why Study Haskell?

The Haskell language demonstrates many important concepts in the world of programming languages, including:

- Types: type checking and type inference, parametric polymorphism and ad-hoc polymorphism (overloading)

- Control: lazy and eager evaluation, tail recursion and continuations, and precise management of effects.

Also, Haskell is an important language in the field of the functional programming paradigm, which deals with *values* rather than state, and is an important way of thinking about programming.

3.2.3 Types

Haskell is purely functional, strongly-typed and type-safe (without breaches such as casting). Therefore, a function having a type $f :: A \to B$ means that for every $x \in A$, if $f(x)$ terminates, than $f(x) \in B$. In contrast, in ML functions can throw an exception or have side effects, as it is not purely functional.

Builtin types in Haskell include:

1. Booleans:
   `True, False`

2. Integers and Floats, including builtin operations:
   `0, 1, 2, 1.2, 3.14159... +, *, ...`

3. Strings (which are lists of characters):
   `"Ron Weasley"...`

4. Tuples, containing a fixed number of elements, possibly of different types:
   `(4, 5, "Griffendor") :: (Integer, Integer, String)`

5. Lists, containing a variable number of elements of the *same* type (possibly polymorphic):
   `[] :: [a]` (an empty list, of polymorphic type)
   `1 : [2, 3, 4] :: [Integer]` (“`:`” is the equivalent of `cons` in Lisp or Scheme)

6. Records: collections of different attributes:

```
-- definition of a record type
data Person = Person {firstName :: String, lastName :: String}
-- definition of an element of the same type
hg = Person { firstName = "Hermione", lastName = "Granger"}
```

For more advanced types, see the sections on datatypes and polymorphism.

Another example of the language being strongly-typed is the fact that in an *if-then-else* expression the type of the returned expression in either case must be the same.

Defining a function with multiple arguments can be done in two ways:

1. Through currying:

```
f x y = x + y
```

   In this case the type of `f` is:

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

2. Using tuples:

```
f (x y) = x + y
```

   In this case the type of `f` is:

```
(Integer, Integer) -> Integer
```

3.2.4 Higher order functions

Functions are first class objects: they can be passed as parameters and returned as results of functions (similar to the lambda calculus).

Example The differential operator is naturally defined as a higher order function.

```
diff f = f'
        where
        f' x = (f (x + h) - f x) / h
        h = 0.0001
```

The compiler infers the type of the function:

```
diff :: (float -> float) -> (float -> float)
```

Execution examples:

```
(diff square) 0 = 0.0001
(diff square) 0.0001 = 0.0003
(diff (diff square)) 3 = 2
```

Examples of higher-order functions from the Haskell libraries include the functions map, filter, and fold (similar to Scheme's accumulate), along with curry and uncurry.

Example Another natural example of higher order functions in mathematics arise from numerical analysis. Demonstrated here is code for a fixed-point operator on a real function, used for calculation of the square root function (this can be used for other fixed-point methods of finding roots as well, such as Netwon's method).

```
fixed_point f guess0 =
  let tolerance = 0.000001 in
  let close_enough v1 v2 = abs (v1 - v2) < tolerance;
      try guess =
        let next = f guess in
          if close_enough guess next then next else try next
  in try guess0

-- defining sqrt using fixed points
average x y = (x + y)/2

sqrt1 x = fixed_point (\y -> average y (x / y)) 1

average_damp f = \x -> average x (f x)

sqrt2 x = fixed_point (average_damp (\y -> x / y)) 1
```

Run examples (for explanation of `\x -> cos x` see anonymous functions, under Miscellaneous):

```
> fixed_point (\x -> cos x) 1
0.7390855263619245
> sqrt2 2
1.414213562373095
```

Compare this with the Scheme code (taken from *"Structure and Interpretation of Computer Programs"*):

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
```

```
  (try first-guess))

;;; (fixed-point cos 1.0)

(define (sqrt1 x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1.0))

(define (average-damp f)
  (lambda (x) (average x (f x))))
(define (sqrt2 x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

3.2.5 Pattern matching

Pattern matching is a concise way to define functions, in a manner similar to how one would define a mathematical function. Instead of using cases regarding the input, the function is defined several times on different inputs through general patterns. On execution, the correct definition of the function is invoked according to the pattern that the input matches, or an excpetion is thrown if no pattern matched the input.

Although the resulting definition using this method is usually very clear and concise, there are cases in which the resulting definition is confusing, so caution is required.

Example A simple example is the definition of the length of a list.

```
length [] = 0
length (x:xs) = 1 + length(xs)
```

Note that to apply the second definition, the list must match the expressions `(x:xs)`, meaning that the list must not be empty. The binding is implicit; the first element is bound to `x` implicitly and the rest of the list to `xs` (as it is many times in mathematical definitions and notations).

Polymorphism note. The function `length` as defined above accepts any type of list; indeed, this is desired. The type of the function is, therefore, `[a] -> int`. Note that in Haskell all the elements in a list must have the same type.

Example Here is a concise definition of the important map function, which applies a function on every element of the given list.

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Compare this to a Scheme implementation of the same function (taken from *"Structure and Interpretation of Computer Programs"*):

```
(define map
        (lambda (f xs)
                (if (eq? xs '()) '()
                        (cons (f (car xs)) (map f (cdr xs))))))
```

The general form for pattern definitions is:

```
<name> <pat1> = <exp1>
<name> <pat2> = <exp2>
...
<name> <patn> = <expn>
```

3.2.6 Datatypes

Datatypes are compound user defined types. They are defined through several possible constructors, typically of different sub-datatypes. Operations on a datatype can be performed using the case expression, to distinguish between the different "representations" of the datatype. A natural tool for function definitions over datatypes is the pattern matching mechanism.

The general form of defining a datatype is as follows:

```
data <name> = <clause> | ... | <clause>
<clause> ::= <constructor> | <contructor> <type>
```

Example Evaluation of numerical expressions. Note the polymorphic definition, for each numeric type `a` an appropriate `Exp a` is defined.

```
data Num a => Exp a = Const a |
                      Plus (Exp a, Exp a) |
                      Mult (Exp a, Exp a)

eval (Const x) = x

eval (Plus (x, y)) = (eval x) + (eval y)

eval (Mult (x, y)) = (eval x) * (eval y)

> eval (Mult (Plus (Const 1, Const 2), Const 4))
12
```

The definition of `eval` used pattern matching. Using the case expression, the function would be:

```
eval x = case x of
        Const x -> x
        Plus (x, y) -> (eval x) + (eval y)
        Mult (x, y) -> (eval x) * (eval y)
```

Example Recursive definitions of datatypes are possible as well:

```
data Tree = Leaf Int | Node (Int, Tree, Tree)

-- usage
Node(4, Node(3, Leaf 1, Leaf 2),
Node(5, Leaf 6, Leaf 7))

-- tree mapping
sum (Leaf n) = n
sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2)
```

Example The classical dual representation of complex numbers, with addition and multipication.

```
-- constructors
data Complex = Complex_rect (Float, Float) |
                    Complex_polar (Float, Float)
             deriving Show  -- so it can be printed

-- selectors

real_part (Complex_rect (x, y)) = x

real_part (Complex_polar (magnitude, angle)) = magnitude * (cos angle)

imag_part (Complex_rect (x, y)) = y

imag_part (Complex_polar (magnitude, angle)) = magnitude * (sin angle)

magnitude (Complex_rect (x, y)) = sqrt (x * x + y * y)

magnitude (Complex_polar (magnitude, angle)) = magnitude

angle (Complex_rect (x, y)) = atan2 y x
```

```
angle (Complex_polar (magnitude, angle)) = angle

as_rect z@(Complex_rect _) = z
-- @ is a syntactic sugar that binds the expression to the variable z

as_rect z@(Complex_polar (magnitude, angle)) =
   Complex_rect (real_part z, imag_part z)

add x y =
  Complex_rect (((real_part x) + (real_part y)),
                          ((imag_part x) + (imag_part y)))

mult x y =
  Complex_polar (((magnitude x) * (magnitude y)),
                                    ((angle x) + (angle y)))
```

Some run examples:

```
> add (Complex_polar(1, pi / 2)) (Complex_rect(1, 0))
Complex_rect (1.0,1.0)
> mult (Complex_polar(1, pi / 2)) (Complex_rect(0, 1))
Complex_polar (1.0,3.141592653589793)
> as_rect (mult (Complex_polar(1, pi / 2)) (Complex_rect(0, 1)))
Complex_rect (-1.0,1.2246063538223773e-16)
```

Compare this to the Scheme code (taken from *"Structure and Interpretation of Computer Programs"*):

```
;;; constructors
(define (make-from-real-imag (real-part z) (imag-part z)))
(define (make-from-mag-ang (magnitude z) (angle z)))

;;; selectors
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type -- REAL-PART" z))))
(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
```

```
         (imag-part-polar (contents z)))
        (else (error "Unknown type -- IMAG-PART" z))))
(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))
        (else (error "Unknown type -- MAGNITUDE" z))))
(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))
        (else (error "Unknown type -- ANGLE" z))))

(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))
(define (mult-complex z1 z2)
  (make-mag-ang (* (magnitude z1) (magnitude z2))
                       (+ (angle z1) (angle z2))))
```

Another method to define the above selectors is through message passing. However, the conciseness of the Haskell definition is clear in this example.

Polymorphism note. To allow complex numbers with any numeric type of the real part and complex part etc., the only thing we need to change is the definition of the datatype itself, in the following way:

```
data Fractional a => (Complex a) = Complex_rect (a, a) |
                                   Complex_polar (a, a)
```

The appropriate type `a` is automatically inferred, as usual.

3.2.7 Polymorphism

Polymorphism is a language element having a non-specific type. It allows definitions of datatypes and functions that can be used with many different types. The polymorphic type can be instantiated by any type, or just by a limited set of types (numeric types, for example).

Polymorphism is a very important and useful tool. It allows better code reuse because the programmer does not need to define the same function many times for different types.

It also gurantees consistency between implementations for different types, and therefore is less error-prone.

Many of the above examples have in fact included polymorphic aspects, such as polymorphic functions (as in the `length` and `complex` examples) and polymorphic datatypes (as in the `complex` example, where the valid types for instantiation are limited to `Fractional`).

3.2.8 Lazy evaluation

Lazy evaluation is an evaluation strategy in which arguments of functions are not evaluated before the function is called. Haskell uses the "call by need" strategy, which is similar to "call by name" but also includes *memoization* of the arguments, so they are evaluated only once, even if accessed many times. In a purely function setting, such as in Haskell, "call by name" and "call by need" are semantically equivalent, with "call by need" usually more efficient.

Lazy evaluation is sometimes more efficient and may result in lower asymptotic complexity. It allows efficient handling of very large elements, and even defining and operating over infinite structures. It is also convenient for defining domain-specific languages. It has some disadvantages, such as creating problems for the compiler to optimize code, and it sometimes makes language extensions tricky to get right.

Example Control-flow operators that must be builtin in eager languages, can be defined by the user. Here is a short-circuiting *or*:

```
(||) :: Bool -> Bool -> Bool
True || x = True
False || x = x
```

In eager languages, `x` would be evaluated in any case.

Example Using infinite, or huge, structures (similar to Scheme's streams)

```
main = take 100 [1 .. ] -- take first 100 of [1 .. infinity]

nextMove :: Board -> Move
        nextMove b = selectMove allMoves
        where
                allMoves = allMovesFrom b
```

Here we traverse a gigantic, perhaps infinite, tree of possible moves, very efficiently. We can view this as a lazy paradigm: generate the set of all possible solutions and traverse them to find the one required.

Example This example illustrates the use of lazy infinite structures for computing the value of the $\sqrt{x}$ function.

```
sqrt_improve guess x = average guess (x / guess)

sqrt_stream x =
  let guesses = 1 : map (\guess -> sqrt_improve guess x) guesses
  in guesses

-- (second : rest) is bound to tl by @
converge (first : tl @ (second : rest)) tolerance =
  if abs(first - second) < tolerance
    then second
    else converge tl tolerance
```

Run example:

```
> converge (sqrt_stream 2) 0.0000000001
1.414213562373095
```

Compare this to the Scheme code (taken from *"Structure and Interpretation of Computer Programs"*):

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))

(define (sqrt-stream x)
  (define guesses
    (cons-stream 1.0
                 (stream-map (lambda (guess)
                               (sqrt-improve guess x))
                             guesses)))
  guesses)
```

Example Approximating the value of π through monte-carlo simulation, whose random input is represented as an infinite stream. (The relevant mathematical fact is that the probability of two numbers smaller than N chosen at random to be relatively-prime tends to $\frac{6}{\pi^2}$ as $N \to \infty$.)

```
prefix list n =
  if n == 0
    then []
    else case list of
```

```
              (first : rest) -> first : prefix rest (n - 1)

nth (x : xs) n =
  if n == 0 then x else nth xs (n - 1)

map2 f (x : xs) (y : ys) = (f x y) : (map2 f xs ys)
map2 _ _ _ = [] -- (halting condition, everything else - one list is empty)

add_lists l1 l2 = map2 (+) l1 l2

integers_from n = n : (integers_from (n + 1))

random_modulus = 2 ^ 32 + 15

randoms seed =
  let result =
        seed : map (\x -> ((x + 101) * 3141592621) `mod` random_modulus) result
  in result

map_successive_pairs f (first : second : rest) =
  (f first second) : map_successive_pairs f rest

cesaro = map_successive_pairs (\r1 r2 -> gcd r1 r2 == 1) (randoms 27182818284)

monte_carlo (experiment : rest) passed failed =
  let next passed failed =
        passed / (passed + failed) : monte_carlo rest passed failed
  in if experiment
        then next (passed + 1) failed
        else next passed (failed + 1)

pi_stream = map (\p -> sqrt (6 / p)) (monte_carlo cesaro 0 0)
```

Some run examples of the function defined above:

```
fibs = 0 : 1 : add_lists fibs (tail fibs)
> take 10 fibs
[0,1,1,2,3,5,8,13,21,34]
> nth pi_stream 1000000
3.141126913610699
```

Compare the Haskell implementation to the Scheme code (taken from *"Structure and Interpretation of Computer Programs"*):

```
(define rand
  (let ((x random-init))
    (lambda ()
      (set! x (rand-update x))
      x)))

(define random-numbers
  (cons-stream random-init
               (stream-map rand-update random-numbers)))

(define cesaro-stream
  (map-successive-pairs (lambda (r1 r2) (= (gcd r1 r2) 1))
                        random-numbers))

(define (map-successive-pairs f s)
  (cons-stream
   (f (stream-car s) (stream-car (stream-cdr s)))
   (map-successive-pairs f (stream-cdr (stream-cdr s)))))

(define (monte-carlo experiment-stream passed failed)
  (define (next passed failed)
    (cons-stream
     (/ passed (+ passed failed))
     (monte-carlo
      (stream-cdr experiment-stream) passed failed)))
  (if (stream-car experiment-stream)
      (next (+ passed 1) failed)
      (next passed (+ failed 1))))

(define pi
  (stream-map (lambda (p) (sqrt (/ 6 p)))
              (monte-carlo cesaro-stream 0 0)))
```

3.2.9 Miscellaneous

List comprehensions

A syntactic suger for map and filter. It is similar to the mathematical notation of set comprehension.

```
myData = [1,2,3,4,5,6,7]
twiceData = [2 * x | x <- myData]
-- [2,4,6,8,10,12,14]
```

```
twiceEvenData = [2 * x| x <- myData, x `mod` 2 == 0]
-- [4,8,12]
```

Anonymous functions

It is very convenient to define functions that are not used many times as anonymous functions, i.e. without a binding name. In Haskell they are defined through the lambda syntax, such as in `\x -> x+1`. That expression is equivalent to the lambda expression

$$\lambda x.\ x + 1$$

This is very similar to the `lambda` syntax in Scheme and the `function` syntax in Javascript.

Actually, declaration of funcions without pattern matching at all would take the form:

```
f = \x y -> x + y
> f 3 4
7
```

The above example also demostrates the definition of anonymous function with multiple argument (curried), just like in $\lambda x.\ \lambda y.\ x + y$.

List reversing example

How one would define a function reversing a given list?

```
reverse [] = []
reverse (x:xs) = (reverse xs) ++ [x]
```

However, the time complexity of this implementation is $O(n^2)$ because the `++` operator requires $O(n)$ time. The reason is that, as in Scheme, a list is defined using nested `cons` of elements. Therefore inserting an element to the beginning of a list requires $O(1)$ for each element, whereas inserting an element to the end of a list requires $O(n)$ for each element.

A more efficient implementation, with just one pass on the list, is as follows:

```
reverse xs =
    let rev ([], accum) = accum
        rev (y:ys, accum) = rev (ys, y:accum)
    in rev (xs, [])
```

Offside rule

The offside rule means that Haskell uses white-space indentation in the context of `let`, `where`, `case` to start a new declaration, which opens a new scope. See previous examples for the indentation usage with those language constructs.

3.2.10 Compiling lazy functional programs

Functional languages lack explicit control flow and are based on expressions constructed out of values and function applications. Evaluating a non-trivial expression amounts to evaluating some sub-expressions and combining the partial results into the final answer by applying a built-in operator or user-defined function. It is usually up to the compiler and run-time system to determine which subexpressions to evaluate in which order. Imperitive languages may require evaluting the value of each of these subexpressions before calling the function, for example, in the case of function arguments. Lazy evaluation relaxes these constraints by specifying that a subexpression will only be evaluated when its value is needed for the progress of the computation.

The expressivness of the lazy evalutation burdens the functional language implementation, which must support delaying and resuming computations. This is implemented, in short, via translating a function application to a heap-allocated data structure, containing the function and its arguments, so that when needed, such a suspended function can be activated.

In Haskell, handling of the most difficult aspects of the language—higher-order functions and lazy evaluation—is deferred to the run-time system. Consequently, the run-time system of a functional language is considerably larger than that of most imperative languages and includes routines for garbage collection, dynamic function creation/invocation, and suspension/resumption of lazy computations. The run-time system takes care of handling those aspects of a functional language that cannot be handled at compile time, like lazy evaluation and higher-order functions.

3.2.11 Summary

Haskell is a powerful purely-functional programming language. By its effective type system and strong language constructs for defining functions and datatypes combined with the applicability of lazy evaluation, Haskell provides concise coding, whose results compare with the efficiency of C code.

Haskell is less popular than imperative or modern object-oriented languages, but it is successfully used in some commercial applications (through F# and Erlang) and its ideas are used in imperative programs too. However, your knowledge of Haskell is likely to help you in the way you think about programs, and might lead to better results when used as a programming language.

3.3 Sources

For the section on Lambda calculus we used the following resources:

- Types and Programming Languages—by B. C. Pierce
- Concepts in Programming Languages—by J. C. Mitchell
- Glasgow Haskell Compiler documentation—
  `www.haskell.org/ghc/docs/latest/html/users_guide/index.html`
- The untyped Lambda Calculus—by Manuel Eberl
  `pspace.org/downloads/lambda_paper.pdf`
- A Tutorial Introduction to the Lambda Calculus—by Raul Rojas
  `www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf`
- A short introduction to the Lambda Calculus—by Achim Jung
  `www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf`
- The Church-Rosser theorem, Wikipedia
  `http://en.wikipedia.org/wiki/Church%E2%80%93Rosser_theorem`

For the section on Haskell we used also

- Haskell: the Craft of Functional Programming—by Simon Thompson
- The Haskell School of Expression—by Paul Hudak
- Modern Compiler Design—by Grune, Bal, Jacobs, Langendoen
- Examples containing Scheme code are based on examples from Structure and Interpretation of Computer Programs—by Abelson and Sussman
  `http://mitpress.mit.edu/sicp/full-text/book/book.html`

The course's sources can be found in

- Advanced Topics in Programming Languages—Tel Aviv University
  `www.cs.tau.ac.il/~msagiv/courses/apl12.html`