Imperative Programming Languages

- Most of the languages we looked at so far are imperative languages
- Tied to the von Neumann architecture
- States are represented by variables, and executing statements changes states

Pure Functional Programming

- All computations are expressed as mathematical functions: an association of input to output
- There are no external states (or variables): the output of a function call depends only on its input
- Many functional languages are not pure and provide imperative features to increase efficiency

Mathematical Functions

- A mathematical function is a mapping from the domain to the range
- The domain can be cross product of sets (multiple arguments)
- Computations is generally defined by recursion and conditional, instead of sequencing and iteration
- No side effects, operand evaluation order is irrelevant

Higher-order Functions

- A function that takes another function as a parameter, and may return a function.
- "Apply-to-all" is a common operation. e.g. map in Racket, transform in C++.

Lambda Calculus

- Lambda calculus was invented by Alonso Church (1930s)
- It can be used to model computation
- Computations are modelled as performing reductions on lambda terms.
- Basis of functional programming

Lambda Terms

- A “variable” is a lambda term.
- If M is a lambda term, so is $(\lambda x.M)$
- If M and N are lambda terms, so is $(M\ N)$

Lambda Terms

- The second rule is called abstraction—corresponds to function definition
- x is the parameter of the function
- Note that a function can have only a single parameter
- The third rule is called application—corresponds to applying function M to argument N

Lambda Terms

It is common to abbreviate:

- $(\ldots((E_1 E_2) E_3) \ldots E_n) \equiv (E_1 E_2 \ldots E_n)$
- $(\lambda x.(\lambda y.(\lambda z.M))) \equiv (\lambda xyz.M)$

Bound and Free Variables

- In $(\lambda x.M)$, each free occurrences of x is bound to the outer lambda. e.g. $(\lambda x.(xy))$
- But occurrences of x that are already bound is not bound to the outer lambda. e.g. $(\lambda x.(\lambda x.(xy)))$
- We can define recursively when x is free in E:
  - $E = x$
  - $E = (\lambda y.A)$ where $y \neq x$ and x is free in A
  - $E = (A\ B)$ where x is free in A and B
- Intuitively, how x is free or bound is similar to how local variables override variables of the same name.

Reductions

- There are three main reductions that can be applied to lambda terms.
- α-conversion: rename a variable in λx and all instances of x bound to it.
- β-reduction: apply a function to its argument. This is done by substitution: from $((\lambda x.M)A)$, we substitute A into all free instances of x in M.

$$((\lambda x.(xy))A) \to (Ay)$$

- η-conversion: if x is not free in M, then $(\lambda x.(Mx)) \to M$
- Notice that functions can be arguments to other functions, and results can be functions as well

Reductions

- From any starting lambda terms, we can apply different reductions at different points. This is how "computation" is done.
- The result of the computation is to perform reductions until we get to the "simplest" form that cannot be further reduced (other than α-conversions).
- Some lambda terms cannot be reduced and in fact β-reductions can be applied forever:
$$((\lambda x.xxx)(\lambda x.xxx))$$
- When there are multiple reductions that can be applied at some point, different choices can lead to different sequences of reductions
- Can this lead to two different simplest forms? No! (Church-Rosser Theorem)
- Leftmost reduction will always get to the the simplest form, if it exists

Currying

- Named after logician Haskell Curry
- In Lambda calculus, each function can only have one parameter
- Functions with multiple parameters are simulated by nested one parameter functions
- Applying an m-ary function to an argument results in an $(m-1)$-ary function
- Evaluating an m-ary function is the same as evaluating a sequence of m unary functions
- This can be done (kind of) in C++ and other imperative languages as well

Basic Operations

- Natural numbers can be represented as lambda terms.
- $0 \equiv (\lambda sz.z)$
- $1 \equiv (\lambda sz.sz)$
- $2 \equiv (\lambda sz.s(sz))$
- etc.
- Addition and multiplication can be done by applying the functions:

$$(\lambda wzyx.wy(zyx))$$

$$(\lambda wzy.w(zy))$$

Basic Operations

- True is represented by $(\lambda xy.x)$
- False is represented by $(\lambda xy.y)$
- Why does this make sense?
- If we want to say "if A then B else C" and A evaluates to one of the above, then (ABC) would select the correct branch.
- not: $(\lambda w.wFT)$ (F and T are from above)
- and: $(\lambda wz.wzF)$
- or: $(\lambda wz.wTz)$

Recursion

- Recursion can be modelled in Lambda calculus by applying functions that can conditional replicate itself.
- There is a "fixed-point combinator function" Y such that applying it to any other function R results in an arbitrarily long chain $R(R(\ldots R(Y R) \ldots)$
- In particular, $(Y R)A \rightarrow R(Y R)A$. If R is a binary function, it could take the first argument as a copy of itself for the recursive call.

Lisp-based Languages

- Based on Lambda calculus
- Have lists as data structures (cons, car, cdr, etc.)
- Allows more than one parameter in a function
- Let expressions: `(let ((x val)) body)` is equivalent to `((lambda (x) body) val)`
- Evaluating a let expression (or any expression in general) needs a list of current name-value pairs. Use a stack-like structure for lookups.

Lambda/Let

- Let is essentially lambda function definition followed by an application of the function
- If we know how to handle lambda definitions and applications, we can implement let "for free"

Lambda/Let

- Generally, each expression is evaluated using a list of lists containing the current environment
- Each function application can simply evaluating the function body but using an updated environment: param = values

Lambda/Let

- To implement a lambda definition, we need to return a closure.
- Closure consists of: parameter list, function body, and the current environment

Letrec

- Let does not allow for recursion. The value is evaluated from surrounding scope
- One can use the fixed-point combinator trick in Lambda calculus but it is not easy to read/write
- Letrec handles that internally
- Mutually recursive functions are more problematic but can be done by other combinators as well

Lazy Evaluation

- Sometimes, arguments to functions need not be evaluated.

  ```
  (define (f test a b)
    (if (test) a b))
  (f (...) (...) (...))
  ```

  If the test evaluates to true, the third argument does not need to be evaluated at all.

- Lazy evaluation: delay evaluation of an operand/argument until it is needed
- The evaluation of the argument has to be wrapped in a package that can be evaluated later. This is sometimes call a thunk.
- The package includes the current evaluation environment.