**Programming Language Features**

- Many features exist simply for convenience
  - Multi-argument functions `foo(a, b, c)`
    - Use currying or tuples
  - Loops `while (a < b) ...`
    - Use recursion
  - Side effects `a := 1`
    - Use functional programming

- So what language features are really needed?

---

**Turing Completeness**

- Computational system that can
  - Simulate a Turing machine
  - Compute every Turing-computable function

- A programming language is **Turing complete** if
  - It can map every Turing machine to a program
  - A program can be written to emulate a Turing machine
  - It is a superset of a known Turing-complete language

- Most powerful programming language possible
  - Since Turing machine is most powerful automaton

**Programming Language Theory**

- Come up with a “core” language
  - That’s as small as possible
  - But still Turing complete

- Helps illustrate important
  - Language features
  - Algorithms

- One solution
  - Lambda calculus
Lambda Calculus (\(\lambda\)-calculus)

- Proposed in 1930s by
  - Alonzo Church
    (born in Washington DC!)
- Formal system
  - Designed to investigate functions & recursion
  - For exploration of foundations of mathematics
- Now used as
  - Tool for investigating computability
  - Basis of functional programming languages
    - Lisp, Scheme, ML, OCaml, Haskell...

Lambda Expressions

- A lambda calculus expression is defined as
  
  \[
  e ::= x \quad \text{variable} \\
  \ | \ \lambda x. e \quad \text{function} \\
  \ | \ e e \quad \text{function application}
  \]

  - Note that this is CFG is ambiguous, but that's not a problem for defining the terms in the language – we are not using it for parsing (i.e., different parse trees = different expressions)

  - \(\lambda x. e\) is like \((\text{fun } x \rightarrow e)\) in OCaml

  - That’s it! Nothing but higher-order functions

Three Conveniences

- “Syntactic sugar” for local declarations
  - let x = e1 in e2 is short for \((\lambda x. e2) \ e1\)

- Scope of \(\lambda\) extends as far right as possible
  - Subject to scope delimited by parentheses
  - \(\lambda x. \lambda y. x\ y\) is same as \(\lambda x. (\lambda y. (x \ y))\)

- Function application is left-associative
  - \(x \ y \ z\) is \((x \ y) \ z\)
  - Same rule as OCaml

OCaml implementation

```ocaml
type id = string

type exp = Var of id
| Lam of id * exp
| App of exp * exp

y Var "y"
\(\lambda x. x\) Lam ("x", Var "x")
\(\lambda x. \lambda y. x\ y\) Lam ("x", (Lam ("y", App (Var "x", Var "y")) App (Lam ("x", (Var "x", Var "y")))))
\(\lambda x. \lambda y. x\ y\) App (Lam ("x", Lam ("y", App (Var "x", Var "y")))))
```

CMSC 330 5

CMSC 330 6

CMSC 330 7

CMSC 330 8
Lambda Calculus Semantics

- All we’ve got are functions
  - So all we can do is call them
- To evaluate \( (\lambda x.e_1) e_2 \)
  - Evaluate \( e_1 \) with \( x \) replaced by \( e_2 \)
- This application is called \textit{beta-reduction}
  - \( (\lambda x.e_1) e_2 \rightarrow e_1[x:=e_2] \)
  - This operation is called \textit{substitution}
    - Replace formals with actuals
    - Instead of using environment to map formals to actuals
  - We allow reductions to occur \textit{anywhere} in a term
    - Order reductions are applied does not affect final value!

Lambda Calculus Examples

- \( (\lambda x.x) z \rightarrow z \)
- \( (\lambda x.y) z \rightarrow y \)
- \( (\lambda x.x y) z \rightarrow z y \)
  - A function that applies its argument to \( y \)

Lambda Calculus Examples (cont.)

- \( (\lambda x.x y)(\lambda z.z) \rightarrow (\lambda z.z) y \rightarrow y \)
- \( (\lambda x.y.x y) z \rightarrow \lambda y.z y \)
  - A curried function of two arguments
  - Applies its first argument to its second
- \( (\lambda x.y.x y)(\lambda z.z z) x \rightarrow (\lambda y.(\lambda z.z z)y)x \rightarrow (\lambda z.z z)x \rightarrow xx \)

Beta Reduction Example

- \( (\lambda x.\lambda z.x z) y \)
  - \( \rightarrow (\lambda x.(\lambda z.(x z))) y \) \hspace{1cm} // since \( \lambda \) extends to right
  - \( \rightarrow (\lambda x.(\lambda z.(x z))) y \) \hspace{1cm} // apply \( (\lambda x.e_1) e_2 \rightarrow e_1[x:=e_2] \)
  - \( \rightarrow \lambda z.(y z) \) \hspace{1cm} // final result

Parameters
- Formal
- Actual

Equivalent OCaml code
- \( (\text{fun } x -> (\text{fun } z -> (x z))) y \rightarrow \text{fun } z -> (y z) \)
Static Scoping & Alpha Conversion

- **Lambda calculus uses static scoping**

- **Consider the following**
  - $(\lambda x. (\lambda x.) z) \rightarrow ?$
    - The rightmost “$x$” refers to the second binding
  - This is a function that
    - Takes its argument and applies it to the identity function
  - This function is “the same” as $(\lambda x. (\lambda y.) y)$
    - Renaming bound variables consistently is allowed
    - This is called alpha-renaming or alpha conversion
    - *Ex.* $\lambda x.x = \lambda y.y = \lambda z.z$ $\lambda y.\lambda x.y = \lambda z.\lambda x.z$

Defining Substitution

- **Use recursion on structure of terms**
  - $x[x:=e] = e$ // Replace $x$ by $e$
  - $y[x:=e] = y$ // $y$ is different than $x$, so no effect
  - $(e_1 e_2)[x:=e] = (e_1[x:=e]) (e_2[x:=e])$
    // Substitute both parts of application
  - $(\lambda x.e')[x:=e] = \lambda x.e'$
    - In $\lambda x.e'$, the $x$ is a parameter, and thus a local variable that is different from other $x$'s. Implements static scoping.
    - So the substitution has no effect in this case, since the $x$ being substituted for is different from the parameter $x$ that is in $e$
  - $(\lambda y.e')[x:=e] = ?$
    - The parameter $y$ does not share the same name as $x$, the variable being substituted for
    - Is $\lambda y.(e'[x:=e])$ correct? No...

Completing the Definition of Substitution

- **Recall:** we need to define $(\lambda y.e')[x:=e]$
  - We want to avoid capturing (free) occurrences of $y$ in $e$
  - Solution: alpha-conversion!
    - Change $y$ to a variable $w$ that does not appear in $e'$ or $e$
      - (Such a $w$ is called fresh)
    - Replace all occurrences of $y$ in $e'$ by $w$.
    - Then replace all occurrences of $x$ in $e'$ by $e$!
  - **Formally:**
    - $(\lambda y.e')[x:=e] = \lambda w.((e'[y:=w])[x:=e])$ ($w$ is fresh)
Beta-Reduction, Again

- Whenever we do a step of beta reduction
  - \((\lambda x.e_1) \ e_2 \rightarrow e_1[x:=e_2]\)
  - We must alpha-convert variables as necessary
  - Usually performed implicitly (w/o showing conversion)

- Examples
  - \((\lambda x.\lambda y.x \ y) \ y = (\lambda x.\lambda z.x \ z) \ y \rightarrow \lambda z.y \ z \quad // \ y \rightarrow z\)
  - \((\lambda x.(\lambda x.x)) \ z = (\lambda y.(\lambda x.x)) \ z \rightarrow z \ (\lambda x.x) \quad // \ x \rightarrow y\)

OCaml Implementation: Substitution

(* substitute e for y in m *)

let rec subst m y e =
  match m with
  | Var x ->
    if y = x then e (* substitute *)
    else m (* don’t subst *)
  | App (e1,e2) ->
    App (subst e1 y e, subst e2 y e)
  | Lam (x,e0) -> ...

OCaml Impl: Substitution (cont’d)

(* substitute e for y in m *)

let rec subst m y e = match m with ...
  | Lam (x,e0) ->
    if y = x then m (* Shadowing blocks substitution *)
    else if not (List.mem x (fvs e)) then
      Lam (x, subst e0 y e) (* Safe: no capture possible *)
    else
      let z = newvar() in (* fresh *)
      let e0’ = subst e0 x (Var z) in
      Lam (z,subst e0’ y e)

OCaml Impl: Reduction

let rec reduce e =
  match e with
  | App (Lam (x,e), e2) -> subst e x e2 (* Straight \(\beta\) rule *)
  | App (e1,e2) ->
    let el’ = reduce el in
    if el’ != el then App(el’,e2)
    else App (el,reduce e2) (* Reduce lhs of app *)
  | Lam (x,e) -> Lam (x, reduce e) (* Reduce rhs of app *)
  | _ -> e (* nothing to do *)
**Encodings**

- The lambda calculus is Turing complete
- Means we can **encode** any computation we want
  - If we’re sufficiently clever...

**Examples**
- Booleans
- Pairs
- Natural numbers & arithmetic
- Looping

**Booleans**

- Church’s encoding of mathematical logic
  - **true** = \( \lambda x. \lambda y. x \)
  - **false** = \( \lambda x. \lambda y. y \)
  - if \( a \) then \( b \) else \( c \)
    - Defined to be the \( \lambda \) expression: \( a \ b \ c \)

**Examples**
- if **true** then \( b \) else \( c \) = \( (\lambda x. \lambda y. x) \ b \ c \rightarrow (\lambda y. y) \ c \rightarrow c \)
- if **false** then \( b \) else \( c \) = \( (\lambda x. \lambda y. y) \ b \ c \rightarrow (\lambda y. y) \ c \rightarrow c \)

**Booleans (cont.)**

- Other Boolean operations
  - **not** = \( \lambda x. ((x \ false) \ true) \)
    - **not** \( x \) = if \( x \) then false else true
    - **not** \( true \rightarrow (\lambda x. (x \ false) \ true) \ true \rightarrow (true \ false) \ true \rightarrow false \)
  - **and** = \( \lambda x. \lambda y. ((x \ y) \ false) \)
    - **and** \( x \ y \) = if \( x \) then \( y \) else false
  - **or** = \( \lambda x. \lambda y. ((x \ true) \ y) \)
    - **or** \( x \ y \) = if \( x \) then true else \( y \)

**Pairs**

- Encoding of a pair \( a, b \)
  - \( (a,b) = \lambda x. \text{if } x \text{ then } a \text{ else } b \)
  - **fst** = \( \lambda f. f \ true \)
  - **snd** = \( \lambda f. f \ false \)

**Examples**
- \( \text{fst} \ (a,b) = (\lambda f. f \ true) \ (\lambda x. \text{if } x \text{ then } a \text{ else } b) \rightarrow (\lambda x. \text{if } x \text{ then } a \text{ else } b) \ true \rightarrow \text{if } true \text{ then } a \text{ else } b \rightarrow a \)
- \( \text{snd} \ (a,b) = (\lambda f. f \ false) \ (\lambda x. \text{if } x \text{ then } a \text{ else } b) \rightarrow (\lambda x. \text{if } x \text{ then } a \text{ else } b) \ false \rightarrow \text{if } false \text{ then } a \text{ else } b \rightarrow b \)
Natural Numbers (Church* Numerals)

- Encoding of non-negative integers
  - 0 = λf.λy.y
  - 1 = λf.λy.f y
  - 2 = λf.λy.f (f y)
  - 3 = λf.λy.f (f (f y))

  i.e., n = λf.λy.<apply f n times to y>
  - Formally: n+1 = λf.λy.f (n f y)

*(Alonzo Church, of course)

Operations On Church Numerals

- Successor
  - succ = λz.λf.λy.f (z f y)
  - 0 = λf.λy.y
  - 1 = λf.λy.f y

- Example
  - succ 0 =
    (λz.λf.λy.f (z f y)) (λf.λy.y) →
    λf.λy.f ((λf.λy.y) f y) →
    λf.λy.f ((λy.y) y) →
    Since (λx.y) z → y
    λf.λy.f y
    = 1

Operations On Church Numerals (cont.)

- IsZero?
  - iszero = λz.z ((λy.false) true)
    This is equivalent to λz.((z (λy.false)) true)
  - Example
    - iszero 0 =
      (λz.z (λy.false) true) (λf.λy.y) →
      (λf.λy.y) (λy.false) true →
      (λy.y) true →
      Since (λx.y) z → y
      true

Arithmetic Using Church Numerals

- If M and N are numbers (as λ expressions)
  - Can also encode various arithmetic operations

- Addition
  - M + N = λf.λy.(M f)((N f) y)
  - Equivalently: + = λM.λN.λf.λy.(M f)((N f) y)
    - In prefix notation (+ M N)

  - Multiplication
    - M * N = λf.(M (N f))
    - Equivalently: * = λM.λN.λf.λy.(M (N f)) y
      - In prefix notation (* M N)
**Arithmetic (cont.)**

- Prove 1+1 = 2
  - 1+1 = λx.λy.(1 x)((1 x) y)
  - 2 = λf.λy.f y
  - 1+1 = (λf.λy.f y) x)((1 x) y) →
  - λx.λy.((λf.λy.f y) x)((1 x) y) →
  - λx.λy.(λy.x y)((1 x) y) →
  - λx.λy.x ((1 x) y) →
  - λx.λy.x (((λf.λy.f y) x) y) →
  - λx.λy.x (λy.x y) →
  - λx.λy.x (x y) = 2

- With these definitions
  - Can build a theory of arithmetic

---

**The Fixpoint Combinator**

Y = λf.λx.f (x x) (λx.f (x x))

- Then
  - Y F =
    - (λf.λx.f (x x) (λx.f (x x))) F →
    - (λx.F (x x)) (λx.F (x x)) →
    - F ((λx.F (x x)) (λx.F (x x)))
    - = F (Y F)
  - Y F is a fixed point (aka “fixpoint”) of F
  - Thus Y F = F (Y F) = F (F (Y F)) = ...

- We can use Y to achieve recursion for F

---

**Looping & Recursion**

- Define D = λx.x x, then
  - D D = (λx.x x) (λx.x x) → (λx.x x) (λx.x x) = D D
  - So D D is an infinite loop
    - In general, self application is how we get looping

---

**Example**

fact = λf. λn.if n = 0 then 1 else n * (f (n-1))

- The second argument to fact is the integer
- The first argument is the function to call in the body
  - We’ll use Y to make this recursively call fact

(Y fact) 1 = (fact (Y fact)) 1
  → if 1 = 0 then 1 else 1 * ((Y fact) 0)
  → 1 * ((Y fact) 0)
  → 1 * (fact (Y fact) 0)
  → 1 * (if 0 = 0 then 1 else 0 * ((Y fact) (-1))
  → 1 * 1 → 1
Discussion

- Lambda calculus is Turing-complete
  - Most powerful language possible
  - Can represent pretty much anything in “real” language
    - Using clever encodings
- But programs would be
  - Pretty slow (10000 + 1 → thousands of function calls)
  - Pretty large (10000 + 1 → hundreds of lines of code)
  - Pretty hard to understand (recognize 10000 vs. 9999)
- In practice
  - We use richer, more expressive languages
  - That include built-in primitives

The Need For Types

- Consider the untyped lambda calculus
  - false = \( \lambda x.\lambda y.y \)
  - 0 = \( \lambda x.\lambda y.y \)
- Since everything is encoded as a function...
  - We can easily misuse terms...
    - false 0 → \( \lambda y.y \)
    - if 0 then ...
  - ...because everything evaluates to some function
- The same thing happens in assembly language
  - Everything is a machine word (a bunch of bits)
  - All operations take machine words to machine words

Simply-Typed Lambda Calculus (STLC)

- \( e ::= n | x | \lambda x: t.e | e \)
  - Added integers \( n \) as primitives
    - Need at least two distinct types (integer & function)...
    - ...to have type errors
  - Functions now include the type of their argument

Simply-Typed Lambda Calculus (cont.)

- \( t ::= \text{int} | t \rightarrow t \)
  - \( \text{int} \) is the type of integers
  - \( t1 \rightarrow t2 \) is the type of a function
    - That takes arguments of type \( t1 \) and returns result of type \( t2 \)
  - \( t1 \) is the domain and \( t2 \) is the range
  - Notice this is a recursive definition
    - So we can give types to higher-order functions
Types are limiting

- STLC will reject some terms as ill-typed, even if they will not produce a run-time error
  - Cannot type check Y in STLC
  - Or in Ocaml, for that matter!
- Surprising theorem: All (well typed) simply-typed lambda calculus terms are strongly normalizing
  - They will terminate
  - Proof is not by straightforward induction
    - Applications "increase" term size

Summary

- Lambda calculus shows issues with
  - Scoping
  - Higher-order functions
  - Types
- Useful for understanding how languages work