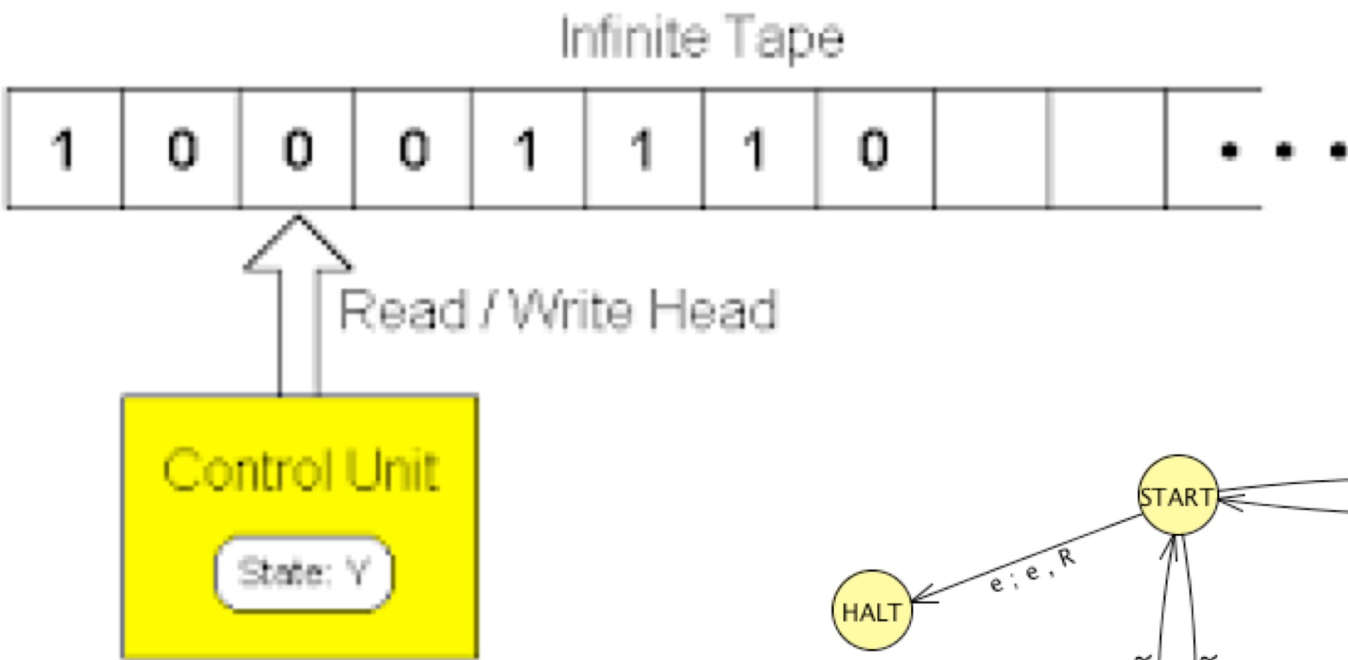
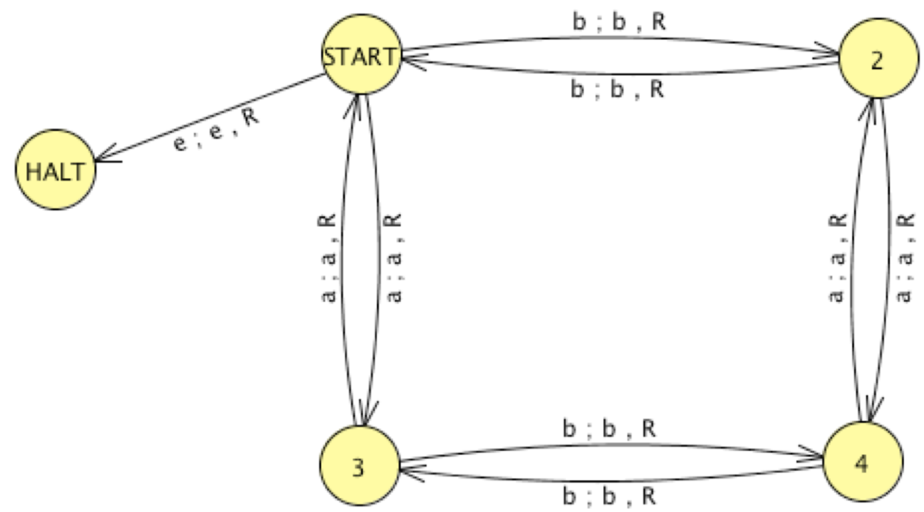
CMSC 330: Organization of Programming Languages

Lambda Calculus

Turing Machine





Turing Completeness

- A language L is Turing complete if it can compute any function computable by a Turing Machine
- Show a language L is Turing complete if
  - We can map every Turing machine to a program in L
    - I.e., a program can be written to emulate a Turing machine
  - Or, we can map any program in a known Turing-complete language to a program in L
- Turing complete languages the "most powerful"
  - *Church-Turing thesis* (1936): Computability by a Turing Machine *defines* "effectively computable"

Programming Language Expressiveness

- So what language features are needed to express all computable functions?
  - What's a minimal language that is Turing Complete?

- Observe: some features exist just 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 pass "heap" as an argument to each function, return it when with function's result

Lambda Calculus (λ-calculus)

- Proposed in 1930s by
  - Alonzo Church

    (born in Washingon 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 Calculus Syntax

- A lambda calculus expression is defined as

e ::= x	**variable**
| λx.e	**abstraction** (func def)
| e e	**application** (func call)

  - This grammar describes ASTs; not for parsing (ambiguous!)
  - Lambda expressions also known as lambda **terms**

- λx.e is like `(fun x -> e)` in OCaml

That's it! Nothing but (higher-order) functions

Why Study Lambda Calculus?

- It is a "core" language
  - Very small but still Turing complete
- But with it can explore general ideas
  - Language features, semantics, proof systems, algorithms, …
- Plus, higher-order, anonymous functions (aka *lambdas*) are now very popular!
  - C++ (C++11), PHP (PHP 5.3.0), C# (C# v2.0), Delphi (since 2009), Objective C, Java 8, Swift, Python, Ruby (Procs), … (and functional languages like OCaml, Haskell, F#, …)

Two Conventions

- Scope of λ extends as far right as possible
  - Subject to scope delimited by parentheses
  - λx. λy.x y is same as λx.(λy.(x y))
- Function application is left-associative
  - x y z is (x y) z
  - Same rule as OCaml

OCaml Lambda Calc Interpreter

- e ::= x
  | λx.e
  | e e

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

y	`Var "y"`
λx.x	`Lam ("x", Var "x")`
λx.λy.x y	`Lam ("x",(Lam("y",App (Var "x", Var "y"))))`
(λx.λy.x y) λx.x x	`App (Lam("x",Lam("y",App(Var"x",Var"y"))), Lam ("x", App (Var "x", Var "x")))`

Quiz #1

`λx.(y z)` and **`λx.y z`** are equivalent

A. True
B. False

Quiz #1

`λx.(y z)` and **`λx.y z`** are equivalent

A. **True**
B. False

Quiz #2

What is this term's AST?

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

λx.x x

A. `App (Lam ("x", Var "x"), Var "x")`
B. `Lam (Var "x", Var "x", Var "x")`
C. `Lam ("x", App (Var "x",Var "x"))`
D. `App (Lam ("x", App ("x", "x")))`

Quiz #2

What is this term's AST?

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

λx.x x

A. `App (Lam ("x", Var "x"), Var "x")`
B. `Lam (Var "x", Var "x", Var "x")`
C. `Lam ("x", App (Var "x",Var "x"))`
D. `App (Lam ("x", App ("x", "x")))`

Quiz #3

This term is equivalent to which of the following?

`λx.x a b`

A. `(λx.x) (a b)`
B. `(((λx.x) a) b)`
C. `λx.(x (a b))`
D. `(λx.((x a) b))`

Quiz #3

This term is equivalent to which of the following?

`λx.x a b`

A. `(λx.x) (a b)`
B. `(((λx.x) a) b)`
C. `λx.(x (a b))`
D. `(λx.((x a) b))`

Lambda Calculus Semantics

- Evaluation: All that's involved are function calls (λx.e1) e2
  - Evaluate e1 with x replaced by e2
- This application is called beta reduction
  - (λx.e1) e2 → e1{e2/x}
    - e1{e2/x} is e1 with occurrences of x replaced by e2
    - This operation is called *substitution*
      - **Replace** formal parameters with actual arguments
      - Instead of using environment to map formals to actuals
  - We allow reductions to occur *anywhere* in a term
    - Order reductions are applied does not affect final value!
- When a term cannot be reduced further it is in beta normal form

Beta Reduction Example

- (λx.λz.x z) y
  - → (λx.(λz.(x z))) y   // since λ extends to right
  - → (λx.(λz.(x z))) y   // apply (λx.e1) e2 → e1{e2/x}
    // where e1 = λz.(x z), e2 = y
  - → λz.(y z)   // final result

Parameters
- Formal
- Actual

- Equivalent OCaml code
  - (fun x -> (fun z -> (x z))) y → fun z -> (y z)

Beta Reduction 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

Beta Reduction Examples (cont.)

- $(\lambda x.x\ y)\ (\lambda z.z) \to (\lambda z.z)\ y \to y$

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

- $(\lambda x.\lambda y.x\ y)\ (\lambda z.zz)\ x \to (\lambda y.(\lambda z.zz)y)x \to (\lambda z.zz)x \to xx$

Beta Reduction Examples (cont.)

(λx.x (λy.y)) (u r) → (u r) (λy.y)

(λx.(λw. x w)) (λz.z) → (λw. (λz.z) w) → (λw.w)

Quiz #4

`(λx.y) z` can be beta-reduced to

A. **`y`**
B. **`y z`**
C. **`z`**
D. cannot be reduced

Quiz #4

`(λx.y) z` can be beta-reduced to

A. **`y`**
B. **`y z`**
C. **`z`**
D. cannot be reduced

Quiz #5

Which of the following reduces to $\lambda z.\ z$?

a) $(\lambda y.\ \lambda z.\ x)\ z$
b) $(\lambda z.\ \lambda x.\ z)\ y$
c) $(\lambda y.\ y)\ (\lambda x.\ \lambda z.\ z)\ w$
d) $(\lambda y.\ \lambda x.\ z)\ z\ (\lambda z.\ z)$

Quiz #5

Which of the following reduces to λz. z?

a) (λy. λz. x) z
b) (λz. λx. z) y
c) (λy. y) (λx. λz. z) w
d) (λy. λx. z) z (λz. z)

Static Scoping & Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - (λx.x (λx.x)) z → ?
    - 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 (λx.x (λy.y))
  - Renaming **bound** variables consistently preserves meaning
    - This is called alpha-renaming or alpha conversion
  - Ex. λx.x = λy.y = λz.z   λy.λx.y = λz.λx.z

Terminology: Free and Bound Variables

- A free variable is one that doesn't have a surrounding lambda that binds it
  - In (λy.y z x), the variables z and x are free
  - In (λy.λz.y z x), the variable x is free
  - In (λy.λz.y z), there are no free variables
- A bound variable is one that does have a corresponding binder
  - In (λy.y z x), the variable y is bound (but not z and x)
  - In (λy.λz.y z x), the variables y and z are bound (not x)
  - In (λy.λz.y), the variable y is bound (z does not appear)

Quiz #6

Which of the following expressions is alpha equivalent to (alpha-converts from)

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

a) $\lambda y.\ y\ y$

b) $\lambda z.\ y\ z$

c) $(\lambda x.\ \lambda z.\ x\ z)\ y$

d) $(\lambda x.\ \lambda y.\ x\ y)\ z$

Quiz #6

Which of the following expressions is alpha equivalent to (alpha-converts from)

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

a) $\lambda y.\ y\ y$

b) $\lambda z.\ y\ z$

c) $(\lambda x.\ \lambda z.\ x\ z)\ y$

d) $(\lambda x.\ \lambda y.\ x\ y)\ z$

Defining Substitution

- Use recursion on structure of terms
  - x{e/x} = e // Replace x by e
  - y{e/x} = y // y is different than x, so no effect
  - (e1 e2){e/x} = (e1{e/x}) (e2{e/x})
    // Substitute both parts of application
  - (λx.e'){e/x} = λx.e'
    - In λx.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'
  - (λy.e'){e/x} = ?
    - The parameter y does not share the same name as x, the variable being substituted for
    - Is λy.(e' {e/x}) correct? No…

Variable capture

- How about the following?
  - (λx.λy.x y) y → ?
  - When we replace y inside, we don't want it to be captured by the inner binding of y, as this violates static scoping
  - I.e., (λx.λy.x y) y ≠ λy.y y
- Solution
  - (λx.λy.x y) is "the same" as (λx.λz.x z)
    - Due to alpha conversion
  - So alpha-convert (λx.λy.x y) y to (λx.λz.x z) y first
    - Now (λx.λz.x z) y → λz.y z

Completing the Definition of Substitution

- Recall: we need to define $(\lambda y.e')\{e/x\}$
  - 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:

$$(\lambda y.e')\{e/x\} = \lambda w.(e'\{w/y\})\{e/x\} \quad \text{(w is fresh WRT e and e')}$$

Beta-Reduction, Again

- Whenever we do a step of beta reduction
  - (λx.e1) e2 → e1{e2/x}
  - We alpha-convert variables as necessary
  - Sometimes performed implicitly (w/o showing conversion)

- Examples
  - (λx.λy.x y) y = (λx.λz.x z) y → λz.y z // y → z
  - (λx.x (λx.x)) z = (λy.y (λx.x)) z → z (λx.x) // x → y

OCaml Implementation: Free variables

```
(* compute free variables in e *)
let rec fvs e =
  match e with
    Var x -> [x]
```
"Naked" variable is free

```
  | App (e1,e2) -> (fvs e1) @ (fvs e2)
```
Append free vars of sub-expressions

```
  | Lam (x,e0) ->
    List.filter (fun y -> x <> y) (fvs e0)
```
Filter x from the free variables in e0

OCaml Implementation: Substitution

```
(* substitute e for y in m--   m{e/y}       *)
let rec subst e y m =
  match m with
      Var x ->
        if y = x then e (* substitute *)
        else m          (* don't subst *)
    | App (e1,e2) ->
        App (subst e y e1, subst e y e2)
    | Lam (x,e0) -> …
```

OCaml Impl: Substitution (cont'd)

```
(* substitute e for y in m--   m{e/y}      *)
let rec subst e y m = match m with …
    | Lam (x,e0) ->
      if y = x then m
      else if not (List.mem x (fvs e)) then
        Lam (x, subst e y e0)
      else
        let z = newvar() in (* fresh *)
        let e0' = subst (Var z) x e0 in
        Lam (z,subst e y e0')
```

Shadowing blocks substitution

Safe: no capture possible

Might capture; need to α-convert

OCaml Impl: Reduction

```
let rec reduce e =
  match e with
      App (Lam (x,e), e2) -> subst e2 x e
    | App (e1,e2) ->
      let e1' = reduce e1 in
      if e1' != e1 then App(e1',e2)
      else App (e1,reduce e2)
    | Lam (x,e) -> Lam (x, reduce e)
    | _ -> e
```

Straight β rule

Reduce lhs of app

Reduce rhs of app

Reduce function body

nothing to do

Quiz #7

Beta-reducing the following term produces what result?

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

A. y (λz.z y)
B. z (λy.y z)
C. y (λy.y y)
D. y y

Quiz #7

Beta-reducing the following term produces what result?

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

A. **$y\ (\lambda z.z\ y)$**
B. $z\ (\lambda y.y\ z)$
C. $y\ (\lambda y.y\ y)$
D. $y\ y$

Quiz #8

Beta reducing the following term produces what result?

$$\lambda x.(\lambda y.\ y\ y)\ w\ z$$

a) $\lambda x.\ w\ w\ z$

b) $\lambda x.\ w\ z$

c) $w\ z$

d) Does not reduce

Quiz #8

Beta reducing the following term produces what result?

$$\lambda x.(\lambda y.\ y\ y)\ w\ z$$

a) λx. w w z

b) λx. w z

c) w z

d) Does not reduce