Module 6
Lambda Calculus

CS 332 Organization of Programming Languages
Embry-Riddle Aeronautical University
Daytona Beach, FL

M06 Outcomes

At the end of this module you should be able to ...

1. Identify the values, identifiers, operators, and operands in an expression.
2. Recognize two expressions as structurally equivalent or not.
3. Identify the forms of expressions: variable, function, application
4. Identify the free and bound variables in a function.
5. Identify the scope of any bound variables in a function.
6. Identify where inadvertent capture may occur in a function.
7. Reduce an expression using Alpha and Beta reduction.
8. Reduce Boolean and recursive functions.

A Reminder Before we begin ….

- Course Meta-Lesson: What is computation?

- Don’t confuse the thing with the usage
  - Use computation to solve numeric/logical problems
  - These things have meaning

- Computation: Manipulation of symbols using formal rules
  - There is no meaning in the symbols or the manipulations (operations)
  - Formal (well defined, unambiguous) allows for autonomous operation
  - Mindless and autonomous → computers!

- Manipulation of symbols / expressions is also called rewriting.

Motivation

- Functional Programming
  - Functional Language like LISP, Scheme, Racket mimic λ calculus
  - Python supports fully, Java contains functional elements
- Computation
  - λ calculus defines and can represent all computations
  - Provide clarity on what computation is (and isn't)
- Formal Systems
  - Course meta-lesson: nature of formal systems
  - Understanding formal systems helps one write formal specifications

λ Calculus Background

- Devised by Alonzo Church (and others) in the 1930's
- Wanted to create a system expressing all computations
- Similar time frame to Turing's work (Turing Machine, circa 1936)
- Both approaches came to the same conclusions
    - Some things are not computable (Church-Turing Thesis)
    - We can define computability and required characteristics
    - Turing defined in terms of *decidability* and *termination*.
    - Church defined in terms of *expressiveness*.

λ Calculus Basic Definitions

- All formal systems have:
  - A small number of things (operands)
  - A small number of operators to manipulate those things
  - Complexity through repetition (iteration or recursion)
- λ Calculus has only one thing – the expression
  - But there are three types of expressions: variables, functions, application
  - So, okay, three things.

- λ Calculus has only one operator – the function
- The BNF definition:
  - $\text{expr} \rightarrow \text{var} \mid \lambda\text{var.exp} \mid \text{exp}_1\ \text{exp}_2$

variable function application

Expressions

- The BNF definition: expr → var | λ var.exp | exp_1 exp_2
  - Variables are simply names: x, y, z, p, q, radius, area, chucky
  - λ var.exp is a function having variable var as input
  - Think of λx.exp as a function of x, defined in exp = f(x)
  - exp_1 exp_2 is an application, applying exp_1 to exp_2
  - Typically, exp_1 is a function, and exp_2 is the input to the function

- Highly abstract: No numerals, no math operators
  - We will pretend they exist at first (easier to make examples)
  - We will prove they can be derived later (justifies the pretending)

Expressions

- The BNF definition: expr → var | λ var.exp | exp_1 exp_2
- The third BNF rule (application) can make long expressions
  - exp_1 exp_2 exp_3 exp_4 exp_5 exp_6 … exp_n
  - Is this one expression or n expressions? (Answer: Yes.)
  - "Expression" is a rubbery term.
- Some expressions:
  - x (a variable)
  - x y (application: exp_1 = x and exp_2 = y)
  - λx.(x + 1) (a function: We're pretending that "+" and "1" exist.)
  - λx.(x + 1) 7 (application, applying "x+1" to 7)

More About Expressions

- Symbols have no inherent meaning
  - R = Resistance? Radius? Gas Constant?

- Focus on structure

- Expression $\lambda x.x$ is the same as $\lambda y.y$ or $\lambda q.q$, and …

- $\lambda x.(x + x)$, $\lambda y.(y + y)$, $\lambda z.(z + z)$ are the same, but …
  - Common form: $\lambda var_1.(var_1 + var_1)$

- $\lambda x.(x + y)$ is not the same as $\lambda y.(y + y)$,
  - $\lambda x.(x + y)$ is adding two separate variables -- $\lambda var_1.(var_1 + var_2)$
  - $\lambda y.(y + y)$ is adding a variable to itself -- $\lambda var_1.(var_1 + var_1)$

Functions

- Reminder: Pretending that numerals and math operations exist.
- $\lambda x.(x + x)$ is a function, where x is the input
  - λ → Tells us this is a function
  - x → The independent variable in the function (using algebra/calc language)
  - $(x + x)$ → The expression that serves as the body of the function
- This is almost the same as what you learned in algebra/calculus
  - $y = x + x$ is a function of x, $y = f(x)$.
  - $y = x + x$ sets up a relation between x and y
  - the function input has a name: x
  - the function output has a name: y
  - The output of $\lambda x.(x + x)$ does not have a name – otherwise very similar

Functions, Bound and Free Variables

- Variables are either bound or free in a function
- The variable named right after the λ is bound, all others are free
  - λx.(x + x) – Here, x is bound, and there are no free variables
  - λx.(x + y) – Here, x is bound, and y is free

- This can get interesting – scoping rules apply
  - λx.(λy.(y + y) + x) – Here y is bound by the inner λ, x bound by the outer λ
  - λx.(λy.(y + x)) – Again, y is bound by the inner, and x bound by the outer λ
  - λx.(λy.(y + x) + y) – Here, y occurs both bound and free – don't do this!
  - λx.(λy.(y x)) – Same as before, but what is (x y)? Application: exp_1 exp_2

Functions, Bound and Free Variables

- This can also get ugly, so introduce cleaner notation
  - $\lambda x.(\ \lambda y\ (\lambda z.(\ x\ y\ z))) \rightarrow \lambda xyz.(\ x\ y\ z)$
  - We'll use this shorter notation

Rewriting Rules (Named, not Explained)

- Computation: rewriting expressions through formal rules

- λ calculus applies functions to expressions: How?
  - Using formal rewriting rules
  - Often called reductions (they tend to reduce the size of the expression)
  - Naming them here; details later

- λ calculus has three rewriting rules (called reductions)
  - α reduction – changes the variables named in an expression
  - β reduction – Applies a function to input
  - η reduction – Provides equivalence between functions (we're ignoring)

α Reductions

- An α reduction renames one or more variables in an expression.
  - Given $\lambda x.x$, we can change it to $\lambda y.y$
  - Given $\lambda x.(x + y)$, we can change it to $\lambda x.(x + s)$
  - Given y (yes, that's an expression), we can change it to z (or any variable)
- Why can we do this? Variable names don't matter – structure does.
- When can't we do this? When we change the structure
  - Given $\lambda x.(x + y)$, we cannot change it to $\lambda x.(x + x)$
  - $(x + y)$ adds two distinct variables, $(x + x)$ adds a variable to itself.
  - The structure changed, so cannot use the α reduction
- Why so we do this? We'll see ….

β Reductions

- Used for application, exp_1 exp_2, where exp_1 is a function

- Given $\lambda x.exp_1$ exp_2 the β reduction does the following:
  - Remove the “$\lambda x.$”
  - Substitute every occurrence of “x” in exp_1 with “exp_2“
  - Remove exp_2

β Reduction Example

- Simple example $\lambda x.x \quad 7$
  - $exp_1 = \lambda x.x$
  - $exp_2 = 7$
  - Since exp_1 is a function, we can use a β Reduction
  - Result $\lambda x.x \quad 7 \rightarrow 7$ (removed the "λx." and replaced "x" with "7")
- $\lambda x.x$ is the identity function, as it returns the input it receives
- λ calculus is abstract; so relate to less abstract examples:
  - λ calculus: $\lambda x.x \quad 7 \rightarrow 7$
  - Algebra: $f(x) = x$, and $f(7) = 7$
  - Java: `public int identity(int x) { return x; }`, and `identity(7)` returns 7.

β Reduction Examples

- $\lambda x.(x + 1)\ \ 7 \rightarrow 7 + 1 = 8$ (still pretending numerals/ops exist)
- $\lambda x.(x + 1)\ \ y \rightarrow y + 1$ (this is as far as we can go)
- $\lambda x.(x + y)\ \ 7 \rightarrow 7 + y$ (as far as we can go)
- $\lambda x.(x\ \ y)\ \ z \rightarrow z\ y$ (as far as we can go)

- Functions having more than one variable:
  - $\lambda xy.(x + y)$ -- substitute left to right, x first, then y
  - $\lambda xy.(x + y)\ \ 8\ \ 6 \rightarrow \lambda y.(8 + y)\ \ 6 \rightarrow 8 + 6 = 14$
  - $\lambda xyz.(z + x + y*x)\ \ 2\ \ 3\ \ 4 \rightarrow \lambda yz.(z + 2 + y*2)\ 3\ \ 4 \rightarrow \lambda z.(z + 2 + 3*2)\ 4 \rightarrow 4 + 2 + 3*2 = 12$

- Reminder: This is exactly the syntax for functional languages.

β Reduction Examples

- A little more complicated; the second expression can be anything
  - $\lambda x.(x\ \ 12)\quad \lambda t.(t + 5) \rightarrow \lambda t.(t + 5)\ \ 12 \rightarrow 12 + 5 = 17$
  - $\lambda x.(x\ \ y)\quad \lambda t.(t + 5) \rightarrow \lambda t.(t + 5)\ \ y \rightarrow y + 5$ (as far as we can go)

- A bad example: “accidental capture”
  - $\lambda xy.(x + y)\ \ t \rightarrow \lambda y.(t + y)$: y is bound, t is free
  - $\lambda xy.(x + y)\ y \rightarrow \lambda y.(y + y)$: no free variables
  - the substituted “y” was “captured”
  - structure of the expression was modified – not good!

- Fix accidental capture with the α reduction:
  - $\lambda xy.(x + y)\ y \rightarrow \lambda xy.(x + y)\ z \rightarrow \lambda y.(z + y)$ – structure is preserved

Numerals

- All formal systems have things and operators

- λ calculus does not include numerals as one of the things

- But, λ calculus can represent numerals:
  - Find some expression and call it 0 (zero)
  - Let 1 be the successor to zero
  - Let 2 successor to 1 …. and so on (forever)
  - This has been done for us
  - It will not be pretty

- λ calculus is ugly, cleaner to assign names to expressions

Numerals

- Numerals are represented as follows:
  - $0 \equiv \lambda sz.z$
  - $1 \equiv \lambda sz.s(z)$
  - $2 \equiv \lambda sz.s(s(z))$
  - $3 \equiv \lambda sz.s(s(s(z)))$
  - $4 \equiv \lambda sz.s(s(s(s(z))))$
  - And so on …. told you it was ugly. Try 1,000,427!

- "z" and "s" have no meaning, often used as a mnemonic
  - "z" reminds you of zero
  - "s" reminds you of the successor function

Math at Last!

- Successor Function: $S = \lambda wyx.y(w\ y\ x)$

- Apply S to 0 (zero):
  - $0 \equiv \lambda sz.z$
  - $1 \equiv \lambda sz.s(z)$
  - $2 \equiv \lambda sz.s(s(z))$
  - $3 \equiv \lambda sz.s(s(s(z)))$
  - $4 \equiv \lambda sz.s(s(s(s(z))))$

$S\ 0 = \lambda wyx.y(w\ y\ x)\quad \lambda sz.z$	
$S\ 0 = \lambda yx.y(\lambda sz.z\ y\ x)$	β Reduction
$S\ 0 = \lambda yx.y(\lambda z.z\ x)$	β Reduction
$S\ 0 = \lambda yx.y(x)$	β Reduction
-- OR –	
$S\ 0 = \lambda sz.s(z)$	= 1

1 + 1 = 2 (The successor to 1 is 2)

- Successor Function: S = λwyx.y(wyx)

- Apply S to 0 (zero):
  - 0 ≡ λsz.z
  - 1 ≡ λsz.s(z)
  - 2 ≡ λsz.s(s(z))
  - 3 ≡ λsz.s(s(s(z)))
  - 4 ≡ λsz.s(s(s(s(z))))

- Reminder: All math is derived from the +1 function.

S 1 = λwyx.y(w y x) λsz.s(z)	
S 1 = λyx.y(λsz.s(z) y x)	β Reduction
S 1 = λyx.y(λz.y(z) x)	β Reduction
S 1 = λyx.y(y(x))	β Reduction
-- OR –	
S 1 = λsz.s(s(z))	= 2

Boolean Values and Operators

- Consider standard if-then-else in programming
  - if P then x else y
  - If P = T, us x and throw away y
  - If P = F, throw away x and use y

- $T = \lambda xy.x$ → Take in two variables, use the first, throw out the second

- $F = \lambda xy.y$ → Take in two variables, throw out the first, use the second

- AND $\wedge = \lambda xy.xy(\lambda uv.v) = \lambda xy.xyF$
- OR $\vee = \lambda xy.x(\lambda uv.u)y = \lambda xy.xTy$
- NOT $\neg = \lambda x.x(\lambda uv.v)(\lambda ab.a) = \lambda x.xFT$

Boolean Operations - AND

$\wedge\ T\ T = \lambda xy.xy(\lambda uv.v)\ \ T\ T$

$\wedge\ T\ T = \lambda xy.xyF\ \ T\ T$ (since $F = \lambda uv.v$)

$\wedge\ T\ T = \lambda y.TyF\ T$ β Reduction

$\wedge\ T\ T = T\ T\ F$ β Reduction

$\wedge\ T\ T = T$ β Reduction

Boolean Operations - AND

$\wedge$ T F = $\lambda xy.xy(\lambda uv.v)$ T F

$\wedge$ T F = $\lambda xy.xyF$ T F (since F = $\lambda uv.v$)

$\wedge$ T F = $\lambda y.TyF$ F β Reduction

$\wedge$ T F = T F F β Reduction

$\wedge$ T F = F β Reduction

Boolean Operations - OR

$V\ F\ T = \lambda xy.x(\lambda uv.u)y\ F\ T$

$V\ T\ F = \lambda xy.xTy\ T\ F$ (since $T = \lambda uv.u$)

$V\ T\ F = \lambda y.TTy\ \ F$ β Reduction

$V\ T\ F = T\ T\ F$ β Reduction

$V\ T\ F = T$ β Reduction

Repetition Through Recursion

- Generalized computation requires some form of repetition

- Lambda calculus achieves this through recursion

- The expression creating recursion is the Y-Combinator.

- Y = λy.((λx.y(xx)) (λx.y(xx)))   R

- The (λx.y(xx)) portion is the “make a copy” portion

- The outer λy.F(y) allows you to input what you want copied.

Recursion

$Y\ R = \lambda y.(\ (\lambda x.y(xx))\ (\lambda x.y(xx))\ )\quad R$

$Y\ R = (\lambda x.R(xx))\ (\lambda x.R(xx))$ β Reduction

$Y\ R = R(\ (\lambda x.R(xx))\ (\lambda x.R(xx))\ )$ β Reduction

$Y\ R = R(R(\ (\lambda x.R(xx))\ (\lambda x.R(xx))\ ))$ β Reduction

$Y\ R = R(R(R(\ (\lambda x.R(xx))\ (\lambda x.R(xx))\ )))$ β Reduction

(and so on …)

Fully Computational?

- There are four basic statement types that make a programming language in any programming language
  - Variable definition / declaration – initial definition of variables and functions
  - Assignment – binding of values to variables
  - Selection – if-then-else and all its forms (if-then-else, switch statements)
  - Repetition – iteration (for and while loops) and/or recursion
- Lambda calculus has all of these things
  - Assignment is radically different than in imperative and OO paradigms
- When speaking at the program level, evaluation of expressions is often assumed
  - Lambda Calculus actually defines it, beginning with zero and successor
- Functional programming paradigm maps directly to Lambda calculus