CMSC 330: Organization of Programming Languages

Lambda Calculus
Turing Machine
Turing Completeness

- Turing machines are the most powerful description of computation possible
  - They define the Turing-computable functions
- 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 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
    - Use currying or tuples
  - Loops
    - Use recursion
  - Side effects
    - Use functional programming pass “heap” as an argument to each function, return it when with function’s result:
      - effectful : 'a → 's → ('s * 'a)
Programming Language Expressiveness

▶ It is not difficult to achieve Turing Completeness
  - Lots of things are ‘accidentally’ TC

▶ Some fun examples:
  - x86_64 `mov` instruction
  - Minecraft
  - Magic: The Gathering
  - Java Generics

▶ There’s a whole cottage industry of proving things to be TC
▶ But: What is a “core” language that is TC?
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…
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#, …)
  - Excel, as of 2021!
Lambda Calculus Syntax

- A lambda calculus **expression** is defined as

  \[ e ::= x \quad \text{variable} \]
  \[ | \quad \lambda x.e \quad \text{abstraction (fun def)} \]
  \[ | \quad e \ e \quad \text{application (fun call)} \]

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

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

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

- 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

- As a convenience, we use the following “syntactic sugar” for local declarations
  - $\text{let } x = e1 \text{ in } e2$ is short for $(\lambda x.e2) \ e1$
Quiz #1

\[ \lambda x. (y \ z) \quad \text{and} \quad \lambda x. y \ z \] are equivalent

A. True
B. False
Quiz #1

\( \lambda x. (y \ z) \) and \( \lambda x. y \ z \) are equivalent

A. True

B. False
Quiz #2

This term is equivalent to which of the following?

\[
\lambda x.x \ a \ b
\]

A. \((\lambda x.x) \ (a \ b)\)
B. \(((\lambda x.x) \ a) \ b)\)
C. \(\lambda x. \ (x \ (a \ b))\)
D. \((\lambda x. \ ((x \ a) \ b))\)
Quiz #2

This term is equivalent to which of the following?

\[ \lambda x. x \ a \ b \]

A. \( (\lambda x. x) \ (a \ b) \)
B. \((((\lambda x. x) \ a) \ b)\)
C. \( \lambda x. \ (x \ (a \ b)) \)
D. \( (\lambda x. \ ((x \ a) \ b)) \)
Lambda Calculus Semantics

- Evaluation: All that’s involved are function calls 
  \((\lambda x.e_1)\ e_2\)
  - Evaluate \(e_1\) with \(x\) replaced by \(e_2\)
- This application is called **beta-reduction**
  - \((\lambda x.e_1)\ e_2 \rightarrow e_1[x:=e_2]\)
    - \(e_1[x:=e_2]\) is \(e_1\) with occurrences of \(x\) replaced by \(e_2\)
    - This operation is called **substitution**
      - Replace formals with actuals
      - 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

(\lambda x.\lambda z.x z) y
→ (\lambda x.(\lambda z.(x z))) y
   // since \lambda extends to right
→ (\lambda x.(\lambda z.(x z))) y
   // apply (\lambda x.e1) e2 → e1[x:=e2]
   // where e1 = \lambda z.(x z), e2 = y

→ \lambda z.(y z)
   // final result

Equivalent OCaml code

• (fun x -> (fun z -> (x z))) y → fun z -> (y z)
Beta Reductions (CBV)

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

2. $(\lambda x.y)\ z \rightarrow \ y$

3. $(\lambda x.x\ y)\ z \rightarrow \ z\ y$
   - A function that applies its argument to $y$
Beta Reductions (CBV)

1. 
   \((\lambda x. x \ y) (\lambda z. z) \rightarrow (\lambda z. z) \ y \rightarrow y\)

2. 
   \((\lambda x. \lambda y. x \ y) \ z \rightarrow \lambda y. z \ y\)
   - A curried function of two arguments
   - Applies its first argument to its second

3. 
   \((\lambda x. \lambda y. x \ y) (\lambda z. z z) \ x \rightarrow (\lambda y. (\lambda z. z z) y) x \rightarrow (\lambda z. z z) x \rightarrow x \ x\)