

Cobol
Scheme
Tcl/Tk
Lisp
Simula
Rexx
Ada
Java
Logo
Perl
SML
PostScript
Basic
C++
APL
Fortran
Modula
SmallTalk

**Introduction to the λ -calculus
(Part I)**

CS386L- Programming Languages

Dr. Greg Lavender
Department of Computer Sciences
The University of Texas at Austin

CS386L: Lecture-03 - Introduction to the λ -calculus

λ -calculus in Computer Science

- a formal notation, theory, and model of computation
 - Church's thesis \Rightarrow λ -calculus is equivalent to Turing Machines
 - equivalence was proved by Kleene
- foundation for the functional style of programming
 - ISWIM, Lisp, Scheme, ML, Haskell, ...
- natural model for many computational objects
 - higher-order functions, variables, block scopes, expressions, ordered pairs, lists, records, recursion
 - calling conventions: call-by-value, call-by-need, call-by-name
- type inferencing & polymorphic type systems
 - Roger Hindley & Robin Milner (Turing award)
- notation for Scott-Strachey denotational semantics
 - Domain theory of Dana Scott (Turing award)

2/16/04 2

Historical Origins

- **Foundations of Mathematics (1879-1936)**
 - Paradoxes of set theory
 - Cantor, Frege, Russell's paradox
 - Axiomatic systems, mathematical logic & type theory
 - Hilbert, Bernays, Brouwer, Russell, Tarski, Zermelo, Fraenkel, ...
 - Gödel's Incompleteness Theorem
 - Metamathematics \Rightarrow a theory of computable functions
 - combinators, "currying" - Moses Schönfinkel (1924)
 - combinatory logic - Haskell Curry (1930)
 - λ -calculus - Alonzo Church (1934)
 - μ -recursive functions - Stephen Kleene (1936)
 - turing machines - Alan Turing (1936)
 - others: Herbrand, Kolmogorov, Post, ...

Alonzo Church

- **PhD from Princeton University, 1927**
 - Advisor was Oswald Veblen (who advised R.L. Moore of UT)
 - Studied with David Hilbert, Paul Bernays & L.E.J. Brouwer in Germany
 - Many of his PhD students are "founding fathers" of theoretical computer science
 - Stephen Kleene, 1934 (recursive function theory)
 - J. Barkley Rosser, 1934 (logic, Church-Rosser theorem)
 - Alan Turing, 1938 (computational logic & computability)
 - Leon Henkin, 1947 (logic, completeness proofs)
 - Martin Davis, 1950 (logic, computability theory)
 - J. Hartley Rogers, 1952 (recursive function theory)
 - Michael Rabin, 1956 (probabilistic algorithms - Turing award)
 - Dana Scott, 1958 (prob. algorithms, domain theory - Turing award)
 - Raymond Smullyan, 1959 (logic, tableau method, formal systems)
 - Peter B. Andrews, 1964 (logic, type theory)

Computers and Programming

"The computer and programming languages were invented by logicians as the unexpected by-product of their unsuccessful effort to formalize [mathematical] reasoning completely. Formalism failed for reasoning, but it succeeded brilliantly for computation. In practice, programming requires more precision than proving theorems!"

- Gregory J. Chaitin, co-founder of Algorithmic Information Theory

Quoted from: *A Hundred Years of Controversy Regarding the Foundations of Mathematics*, in the *The Unknowable*, Springer-Verlag, 1999.

Computation and Information

- **What constitutes a computation?**
 - a mechanical rearrangement of symbols
 - based on well-defined syntactic transformation rules (a calculus)
 - what do the symbols mean and what is a meaningful computation? what is information anyway?
 - how do humans (or machines) interpret the result constructively
- **A simple but powerful answer**
 - symbolic term rewriting
 - **substitution** of terms according to well-defined rules
 - **reduction** of resulting intermediate expressions to a normal form
 - the result is called a **computation**

A computational point of view

- a function in set theory is a graph
 - characterized solely by an input→output relation
 - *extensional equality* - two functions f, g are equal iff they have the same graph, i.e., $\{(x, y) \mid y = f(x) = g(x)\}$
- this doesn't work too well in programming!
 - in what way are two sorting functions equivalent?
 - *intensional equality* - equivalent algorithmic complexity
 - how the function computes its result is important
 - in CS we characterize a function by its algorithm:
 - E.g., a $O(n^2)$ vs $O(n \log_2 n)$ sorting algorithm

2/16/04

7

Some functional notations

Set Theoretic:
 $\{(x, y) \mid \forall x, y \in \mathbb{N}: y = x^2\}$
Algebraic:
 $f: \mathbb{N} \rightarrow \mathbb{N}$
 $f(x) = x^2;$
Type-free λ -notation: $\lambda x. x * x$ **Typed λ -notation:** $\lambda x: \text{int}. x * x$ **Polymorphic λ -notation:** $\lambda x: \alpha. x * x$ **Scheme:**

```
(define square
  (lambda (x) (* x x)))
```

Algol:

```
integer procedure square(x); integer x;
begin square := x * x end;
```

Pascal:

```
function square (x:integer) : integer;
begin square := x * x end;
```

K&R C:

```
square(x) int x; { return (x * x); }
```

StdC/C++/Java:

```
int square(int x) { return (x * x); }
```

ML97:

```
fun square x = x * x;
fun square (x:int) = x * x;
val square = fn x => x * x;
```

Haskell:

```
square :: Int->Int
square x = x * x
map (\x -> x * x) [0..]
[(x,y) | x <- [0..], y <- [x * x]]
```

2/16/04

8

Definitions

- λ -calculus is a formal notation for defining functions
 - expressions in this notation are called λ -expressions
 - every λ -expression denotes a function that is "out there" in the platonistic sense
 - a λ -expression consists of 3 kinds of terms:
 - **variables**: x, y, z , etc.
 - we use V, V_1, V_2 , etc., for arbitrary variables
 - **abstractions**: $\lambda V.E$
 - Where V is some variable and E is another λ -term
 - **applications**: $(E_1 E_2)$
 - Where E_1 and E_2 are λ -terms
 - applications are sometimes called combinations

Formal Syntax in BNF

```

< $\lambda$ -term> ::= <variable>
           |  $\lambda$  <variable> . < $\lambda$ -term>
           | (< $\lambda$ -term> < $\lambda$ -term>)
<variable> ::= x | y | z | ...

```

Or, more compactly:

```

E ::= V |  $\lambda V.E$  | ( $E_1 E_2$ )
V ::= x | y | z | ...

```

Where V is an arbitrary variable and E_i is an arbitrary λ -expression.

We call λV the **head** of the λ -expression and E the **body**.

Variables

- variables can be bound or free
- the λ -calculus assumes an infinite universe of free variables
- they are bound to functions in an *environment*
- they become bound by usage in an abstraction
 - for example, in the λ -expression:
 $\lambda x. x * y$
 x is bound by λ over the body $x * y$, but y is a free variable. I.e., lexically scoped. Compare this to scheme:

```
(define z 3)
(define x 2)
(define y 2)
(define multi-by-y (lambda (x) (* x y)))
(multi-by-y z) => 6
```

2/16/04

11

Abstractions

- if $\lambda V.E$ is an abstraction
 - V is a bound variable over the body E
 - it denotes the function that when given an actual argument 'a', evaluates to the function E' with all occurrences of V in E replaced with 'a', written $E[a/V]$
 - For example the abstraction:

$$\lambda x. x$$

is the identity function

$$(\lambda x. x) 1 \Rightarrow 1$$

$$(\lambda x. x) a \Rightarrow a$$

$$(\lambda x. x) (\lambda x. x) \Rightarrow (\lambda x. x)$$

2/16/04

12

Applications

- If E_1 and E_2 are λ -expressions, so is $(E_1 E_2)$
 - application is basically function evaluation
 - apply the function E_1 to the argument E_2
 - E_1 is called the **rator** (ope-rator)
 - E_2 is called the **rand** (ope-rand)
- For example:

$(\lambda x.xx) 1 \Rightarrow 11$

$(\lambda x.xx) a \Rightarrow aa$

$(\lambda x.xx) (\lambda x.xx) \Rightarrow (\lambda x.xx) (\lambda x.xx)$

this last example is a quine, and it doesn't terminate. It keeps duplicating itself ad infinitum. In this example, we don't care, but in real programming we do care about non-terminating evaluations.

Conversion/reduction rules

- **α -conversion**
Any abstraction $\lambda V.E$ can be converted to $\lambda V'.E[V'/V]$ iff $[V'/V]$ in E is valid
- **β -conversion (β -reduction)**
Any application $(\lambda V.E_1) E_2$ can be converted to $E_1[E_2/V]$ iff $[E_2/V]$ in E_1 is valid
- **η -conversion**
Any abstraction $\lambda V.(E V)$ where V has no free occurrences in E can be converted to E

Conversion rule notation

$E_1 \xrightarrow{\alpha} E_2$	bound variable renaming to avoid naming conflicts
$E_1 \xrightarrow{\beta} E_2$	like a function call evaluation
$E_1 \xrightarrow{\eta} E_2$	elimination of irrelevant information

2/16/04

15

α -redex

$$E_1 \xrightarrow{\alpha} E_2$$

α -reduction is bound variable renaming applied to an α -redex iff no naming conflicts

redex = reducible expression

$$\lambda x.x \xrightarrow{\alpha} \lambda y.y \quad (\lambda x.x)[y/x]$$

$$\lambda x.f x \xrightarrow{\alpha} \lambda y.f y \quad (\lambda x.f x)[y/x]$$

$$\lambda x.\lambda y.x+y \xrightarrow{\alpha} \lambda y.\lambda y.f y+y \quad \text{not valid since } y \text{ is already a bound variable in } E_1$$

2/16/04

16

β -redex

$$E_1 \xrightarrow{\beta} E_2$$

$$(\lambda x. f x) E \xrightarrow{\beta} f E$$

$$(\lambda x. (\lambda y. x + y)) \underline{3} \xrightarrow{\beta} \lambda y. \underline{3} + y$$

$$(\lambda y. \underline{3} + y) \underline{4} \xrightarrow{\beta} \underline{3} + \underline{4}$$

2/16/04

17

Is the λ -calculus Turing complete?

- Can we represent the class of Turing computable functions?
 - yes, we can represent
 - Booleans and conditional functions
 - numerals and arithmetic functions
 - data structures, such as ordered pairs, lists, etc.
 - recursion
 - doing so however is syntactically tedious!
 - actual programming languages use syntactic sugar for functions
 - lambda expressions exist in Scheme, ML, Haskell, and even Python
 - λ -calculus is more suitable as an abstract model of a programming language rather than as a practical programming language
 - it is used to study programming languages semantics from a mathematical perspective called Denotational Semantics
 - see the appendix in the *Revised Report(5) on the Algorithmic Language Scheme* on course website for the denotational semantics of Scheme

2/16/04

18

Example of Syntactic Sugar in Scheme

Scheme has **let** and **let*** operators for establishing local bindings of variables to values over a lexically scoped block. In a let expression, all values in the list of let bindings are evaluated, and then bound to the local variables. In a let* expression, the values are evaluated and bound to the variables sequentially:

```
(define x 2)
(let ((x 3) (y x)) (* x y)) => 6
(let* ((x 3) (y x)) (* x y)) => 9
```

Which are just syntactic sugar for:

```
((lambda (x y) (* x y)) 3 x) => 6
((lambda (x) ((lambda (y) (* x y)) x)) 3) => 9
```

Church Booleans

- We define booleans and logical operators in the λ -calculus as functions:

```
True = T =  $\lambda t.\lambda f.t$  =  $\lambda t f.t$ 
False = F =  $\lambda t.\lambda f.f$  =  $\lambda t f.f$ 
AND =  $\lambda xy.xy(\lambda t f.f)$  =  $\lambda xy.xyF$ 
OR =  $\lambda xy.x(\lambda t f.t)y$  =  $\lambda xy.xTy$ 
NEG =  $\lambda x.x(\lambda uv.v)(\lambda ab.a)$  =  $\lambda x.xFT$ 
```

- Example:

```
NEG True = ( $\lambda x.x(\lambda uv.v)(\lambda ab.a)$ ) ( $\lambda t f.t$ )
=> ( $\lambda t f.t$ ) ( $\lambda uv.v$ ) ( $\lambda ab.a$ )
=> ( $\lambda uv.v$ )
=>  $\lambda t f.f$ 
=> False
```

Church Booleans

- We define *true* and *false* as λ -expressions:

$\text{true} = \lambda t.\lambda f.t$
 $\text{false} = \lambda t.\lambda f.f$

- We then define a "conditional" λ -expression:

$\text{test} = \lambda c.\lambda x.\lambda y.c \ x \ y$
 where $\text{test } \underline{\text{true}} \ v \ w = (\lambda c.\lambda x.\lambda y.c \ x \ y) \ \text{true} \ v \ w$
 $\Rightarrow^* \text{true} \ v \ w$
 $= (\lambda t.\lambda f.t) \ v \ w$
 $\Rightarrow^* v$

Church Numerals

- The natural numbers may be defined using zero and the successor function:
 - $\underline{0}$, $\underline{1} = \text{succ}(\underline{0})$, $\underline{2} = \text{succ}(\text{succ}(\underline{0}))$, ..., etc.
- In the λ -calculus, we only have functions, so we define the natural numbers as functions:
 - $\underline{0} = \lambda s.(\lambda z.z)$, but we will write this as $\lambda sz.z$,
 - then the rest of the natural numbers can be defined as:
 - $\underline{1} = \lambda sz.s(z)$, $\underline{2} = \lambda sz.s(s(z))$, $\underline{3} = \lambda sz.s(s(s(z)))$, ..., etc.

Successor function

- So how do we write a successor function?
 - $S = \lambda w y x. y(w y x)$
 - Let's test it on zero = $\lambda s z. z$
 - $S 0 = (\lambda w y x. y(w y x))(\lambda s z. z)$
 - $\Rightarrow \lambda y x. y((\lambda s z. z) y x) \Rightarrow \lambda y x. y((\lambda z. z) x)$
 - $\Rightarrow \lambda y x. y(x)$
 - $\Rightarrow \alpha \lambda s z. s(z)$
 - $= \underline{1}$
 - Note that $\lambda y x. y(x) = \lambda s z. s(z)$ under α -conversion
 - the variables names are "dummy variables" that can be changed as needed to (carefully) avoid variable name conflicts
 - all we are doing is defining syntactic patterns and rules of rewriting that mimic the semantics of Peano arithmetic

2/16/04

23

Curried Functions

- Named after Haskell Curry who used them in combinatory logic
 - but first used by Moses Schonfinkel in the 1920s
- The basic idea is that any n -ary function can be replaced by a composition of n unary functions.
 - $f(x, y) \Rightarrow (f x) y$
 - $f(x_1, x_2, \dots, x_n) \Rightarrow ((\dots ((f x_1) x_2) \dots) x_n)$

2/16/04

24

Curried Function in Scheme

- **Non-curried sum**
 - `(define sum (lambda (x y) (+ x y)))`
 - `(sum 2 3) => 5`
- **Curried sum**
 - `(define sum (lambda (x) (lambda (y) (+ x y))))`
 - `((sum 2) 3) => 5`
 - `(let ((f (sum 2))) (f 3)) => 5`

Lambda and curried functions in ML

- **A lambda expression in ML is written as**
 - `fn <args> => <body>`
 - `val sum = fn x => fn y => x + y;`
 - `val it = fn : int -> int -> int`
 - `sum 3 2;`
 - `val it = 5 : int`
- **A curried version**
 - `fun sum x = fn y => x + y;`
 - `val sum = fn : int -> int -> int`
 - `sum 3;`
 - `val it = fn : int -> int`
 - `it 2;`
 - `val it = 5 : int`
 - `(sum 3) 2;`
 - `val it = 5 : int`
 - `sum 3 2;`
 - `val it = 5 : int`

Lambda expressions in Haskell

- Lambda expressions are written as:
 - `\<arg> -> <body>`
 - `succ = \n -> n + 1`
 - `succ 0 => (\n -> n + 1) 0 => 0 + 1 => 1`
 - `succ = (\a -> \b -> a + b) 1`
 - `succ 0 => ((\a -> \b -> a + b) 1) 0`
`=> (\b -> 1 + b) 0`
`=> 1 + 0`
`=> 1`
 - note the strong syntactic similarity to the λ -calculus
 - but we usually prefer the syntactic sugar form
 - `succ n = n + 1`
 - `succ 0 => 0 + 1 => 1`
 - `succ (succ 0) => succ (0+1) => succ 1 => 1+1 => 2`

2/16/04

27

Comparing map & fold using lambda

- Map and foldl are similar across functional languages
 - but note the order of the "don't care" argument in the fold examples
 - Scheme (using DrScheme)
 - `(map (lambda (x) (* x x)) '(1 2 3 4 5)) => (1 4 9 16 25)`
 - `(foldl (lambda (_ x) (+ 1 x)) 0 '(1 2 3 4 5)) => 5`
 - ML (using SML)
 - `map (fn x => x * x) [1,2,3,4,5]; => [1,4,9,16,25]:int list`
 - `foldl (fn (_,x) => x + 1) 0 [1,2,3,4,5]; => 5 : int`
 - Haskell (using Hugs)
 - `map (\x -> x * x) [1,2,3,4,5] => [1,4,9,16,25]`
 - `foldl (\x _ -> x + 1) 0 [1,2,3,4,5] => 5`
 - `foldl (\x _ -> x + 1) 0 "Hello, world" => 12`
 - note: strings literals in Haskell are just arrays of characters

2/16/04

28

Homework

- Read the following paper:
- A Tutorial Introduction to the λ -calculus, by Rojas
- Show the following in the λ -calculus using the previous definitions for AND, OR, True and False:
 - $\text{AND True True} \Rightarrow \text{True}$
 - $\text{AND True False} \Rightarrow \text{False}$
 - $\text{OR True False} \Rightarrow \text{True}$
 - $\text{OR False False} \Rightarrow \text{False}$

Homework

- Show how to express the following in the λ -calculus where S is the successor function:
 - $S\underline{1}$, $S\underline{2}$, $S\underline{3}$
- Addition in the λ -calculus
 - let $\underline{2S3}$ represent $2+3$
 - write the λ -expression for $\underline{2S3}$
 - note that you will have to use α -conversion to avoid name conflict
 - show that $\underline{2S3}$ reduces to $S\underline{S3}$
 - show that $S\underline{S3}$ reduces to $S\underline{4}$
 - show that $S\underline{4}$ equals the λ -expression for $\underline{5}$
- Multiplication is done using the expression: $\lambda xyz.x(yz)$
 - show that $(\lambda xyz.x(yz)) \underline{2} \underline{2} \Rightarrow^* \lambda sz.s(s(s(z)))) = \underline{4}$