

CSCI 334:  
Principles of Programming Languages

Lecture 4: Fundamentals II

Instructor: Dan Barowy  
**Williams**

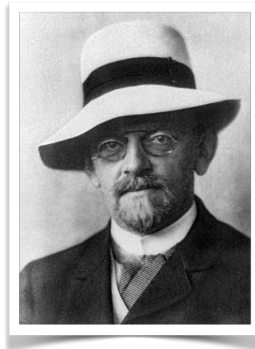
Activity

Write a function `firsts` that, when given a list of `cons` cells, returns a list of the left element of each `cons`.

```
( (a . b) (c . d) (e . f) (g . h) )  
      ↓  
firsts  
      ↓  
(a c e g)
```

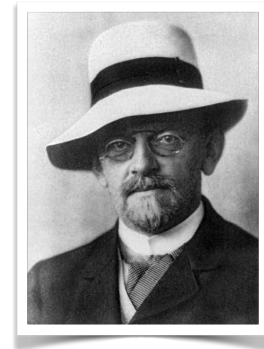
What is computable?

- Hilbert: Is there an algorithm that can decide whether a logical statement is valid given axioms?
- "Entscheidungsproblem" (literally "decision problem")
- Leibniz thought so!



What is computable?

- Why do we care?
- $f(x) = x + 1$
- We can clearly do this with pencil and paper.
- $\int 6x \, dx$
- Also computable, in a different manner.
- We care because the computable functions can be done on a computer.

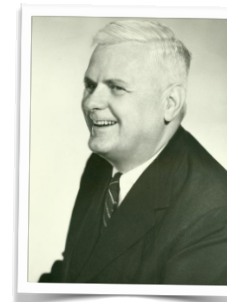




“What is the answer to the ultimate question of life, the universe, and everything?”

## Lambda calculus

- Invented by Alonzo Church in order to solve the Entscheidungsproblem.
- Short answer to Hilbert's question: no.
- Proof: No algorithm can decide equivalence of two arbitrary  $\lambda$ -calculus expressions.



## Lambda calculus is deceptively simple

- Church-Turing thesis: every computable function can be represented in the  $\lambda$ -calculus; i.e., it is “Turing complete”.
- Grammar in BNF:

$$\begin{array}{l}
 M ::= x \quad \text{variable} \\
 | \lambda x.M \quad \text{abstraction} \\
 | MM \quad \text{function application}
 \end{array}$$

$$\begin{array}{l}
 M ::= x \quad \longrightarrow \text{variable} \\
 | \lambda x.M \quad \text{abstraction} \\
 | MM \quad \text{function application}
 \end{array}$$

- “pure”  $\lambda$ -calculus doesn't say anything about the *values* of variables.
- We often extend  $\lambda$ -calculus with arithmetic, so “1”, “2”, “3”, ... are also considered terms.
- Justified by Church's own proof that arithmetic (Peano axioms) can be encoded in  $\lambda$ -calculus (see “Church encoding”).

$M ::= x$                     variable  
 |  $\lambda x.M$                  $\longrightarrow$  abstraction  
 |  $MM$                      $\longrightarrow$  function application

- Functions are at the heart of  $\lambda$ -calculus.
- Functions are "nameless":
  - Just a  $\lambda$  denoting "function".
  - A "bound variable"  $x \in \{x, y, z, \dots\}$
  - An expression  $M$  where  $x$  is "bound".
- E.g.,  $\lambda x.x + 1$  adds one to  $x$

$M ::= x$                     variable  
 |  $\lambda x.M$                  $\longrightarrow$  abstraction  
 |  $MM$                      $\longrightarrow$  function application

- $\lambda x.x + 1$
- Translation:  

```
def func(x):
    return x + 1
```
- Remember that no programming languages existed at the time.  $\lambda$ -calculus was the first!
- Why " $\lambda$ "?

$M ::= x$                     variable  
 |  $\lambda x.M$                 abstraction  
 |  $MM$                      $\longrightarrow$  function application

- How do we compute  $5 + 1 = 6$ ?
- $(\lambda x.x + 1) 5$
- Works by process of substitution
 
$$\begin{aligned}
 & ([5/x] x + 1) \\
 &= (5 + 1) \\
 &= 6
 \end{aligned}$$

### $\alpha$ -equivalence

- The chosen symbol for a bound variable does not matter.
- $\lambda x.x =_{\alpha} \lambda y.y$
- More precisely  

$$\lambda x.M = \lambda y.[y/x]M$$
- "Substitute  $y$  for occurrences of  $x$  in  $M$  (such that  $y$  does not already appear in  $M$ )."
- $M$  is the "scope" of the binding.

## Free variables

- These two are not  $\alpha$ -equivalent
- $\lambda x.x + b \neq_{\alpha} \lambda y.y + c$
- Why?  $b$  and  $c$  are "free variables"
- Proof:  
$$\lambda x.x + b$$
$$=_{\alpha} ([y/x]\lambda x.x + b)$$
$$= \lambda y.y + b$$
- $\lambda y.y + b \neq_{\alpha} \lambda y.y + c$

## $\beta$ -equivalence

- We compute function application using substitution: " $\beta$ -reduction"
- $(\lambda x.x + 1) 5 =_{\beta} 5 + 1$
- How did we get that?
- We substituted 5 for  $x$ .  
$$(\lambda x.x + 1) 5$$
$$=_{\beta} ([5/x]x + 1)$$
$$= (5 + 1)$$
$$= 5 + 1$$

## Constant function

- A constant function is one that does not depend on a variable
- $(\lambda x.1) = 1$
- Means that unnecessary variables can be eliminated
- $(\lambda x.\lambda y.y) = (\lambda y.y)$

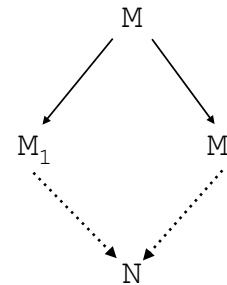
## $\eta$ -equivalence

- If an abstraction exists solely to pass its argument to another function, the abstraction can be eliminated.  
$$\lambda x.M \ x =_{\eta} M$$
  
(assuming that  $x$  does not appear in  $M$ )

## Renaming bound variables

- Some expressions are hard to evaluate unless you rename some of the variables.
- $(\lambda f. \lambda x. f(f\ x)) (\lambda y. y + x)$
- Note that the free variable  $x$  appears on the right and the bound variable  $x$  appears on the left. *These are different variables!*
- $([z/x] \lambda f. \lambda x. f(f\ x)) (\lambda y. y + x)$   
 $=_{\alpha} (\lambda f. \lambda z. f(f\ z)) (\lambda y. y + x)$   
...  $(\lambda z. z + x + x)$

## Order does not matter



If  $M \rightarrow M_1$  and  $M \rightarrow M_2$   
then  $M_1 \rightarrow^* N$  and  $M_2 \rightarrow^* N$   
for some  $N$

“confluence”

## Activity

$$\begin{aligned} & (\lambda f. \lambda x. f(f\ x)) (\lambda z. x+z) 2 \\ & \quad \dots \\ & = x + x + 2 \end{aligned}$$