CS 252:

Advanced Programming Language Principles

Lambda Calculus

Prof. Tom Austin

San José State University

Minimum complete programming language?

WARNING: I expect you to remember every construct of this language for exams

Lambda Calculus expressions

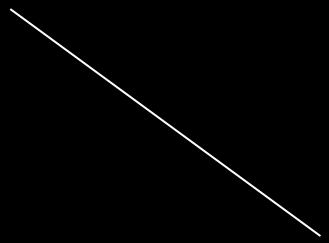
`e`	`::=`	*expressions:*
	`x`	variables
|	`(λx.e)`	lambda abstractions
|	`e e`	function application

We could have just said "function", but we want to sound cool

Lambda Calculus values

`v ::=` *values:*

`(λx.e)` lambda abstractions



When our program finishes running, it returns some complex function as its "value"

Function application

Suppose we have a function:

$(\lambda x.E)$

Where E is some complex expression.

How do we evaluate:

$(\lambda x.E)\ v \rightarrow E[x->v]$

v replaces x wherever it occurs in E

Small step semantics for λ-calculus

(in-class)

Operational Semantics

[Ctxt1]

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2}$$

[Ctxt2]

$$\frac{e2 \rightarrow e2'}{(\lambda x.e)\ e2 \rightarrow (\lambda x.e)\ e2'}$$

[Call]

$$(\lambda x.e)\ v \rightarrow e[x \rightarrow v]$$

Example: Identity Function

```
    (λx.x)  (λa.λb.a)
→       x[x->(λa.λb.a)]
→       (λa.λb.a)
```

When should we evaluate function arguments?

Strict Evaluation Strategies

Evaluate function arguments first

- *Call-by-value:*
  copy of the parameter is passed
- *Call-by-reference:*
  implicit reference is passed

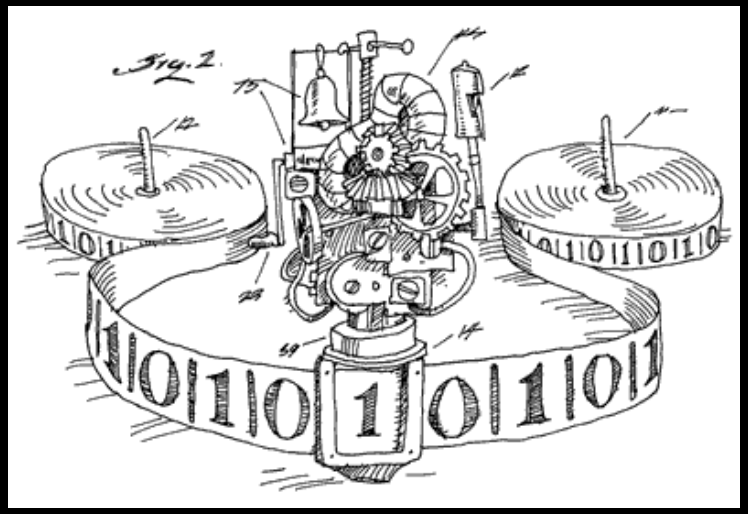
Lazy Evaluation Strategies

Substitute arguments in function body

- *Call-by-name:*
  re-evaluate the argument each time
- *Call-by-need:*
  memoizes parameter value after use

How powerful is this language?

The lambda-calculus is *Turing complete*.



You can also implement the λ-calculus w/ a Turing machine

In other words, the two are **equal in power**

Translating λ-calc to Haskell

Lambda-calculus	Haskell
`x`	`x`
`(λx.e)`	`(\x -> e)`
`e e`	`e e`

Extending the lambda calculus
(in class)

Boolean Values

- `true = λx.λy.x`
  - Get first argument
- `false = λx.λy.y`
  - Get second argument
- `test = λcond.λthn.λels.cond thn els`

Pairs

- `pair = λf.λs.(λb.b f s)`
  - ‘b’ is a boolean.
  - Remember: ‘true’ and ‘false’ act as ‘get first arg’ and ‘get second arg’, respectively.

- `first = λp.p true`

- `second = λp.p false`

Church Numerals

(AKA do something N times)

- Essentially, instructions for producing numbers
- Inputs:
  - `z`: Representation of zero
  - `s`: Successor function that works with `z`
- Examples:
  - `zero  = λs.λz.z`
  - `one   = λs.λz.s z`
  - `two   = λs.λz.s (s z)`
  - `three = λs.λz.s (s (s z))`

Functions for Numbers

- `succ = λn.(λs.λz.s (n s z))`
  - `n s z` — "apply `s` to `z` `n` times"

- `plus = λm.λn.(λs.λz.m s (n s z))`

- `plus' = λm.λn.m succ n`
  - Alternate definition of `plus` that uses `succ`.
  - "Apply `succ` to `n` `m` times."

Omega (Ω)

```
        (λx.x x) (λx.x x)
→       x x[x->(λx.x x)]
→       (λx.x x) (λx.x x)
→       …
```

Y Combinator

- Similar to omega
- Also called the ‘fix’ combinator
- Produces recursive functions
- `fix = λf.(λx.f (λy.x x y)) (λx.f (λy.x x y))`

Lab: Develop new features in the Lambda Calculus using Haskell

Details on Canvas.

Starter code is available on the course website.