Lambda Calculus Introduction

CIS352 — Spring 2021

Kris Micinski

The Lambda Calculus (1930s)

- Variables
- Function application
- Lambda abstraction

Just these three elements form a *complete* computational system

Original Syntax

e	$::=$	x	Variables
	$\mid$	$\lambda x\,.\,e$	Lambdas
	$\mid$	$e_0\ e_1$	Applications

Scheme Syntax

e	$::=$	x	Variables
		$\mid (\lambda\ (x)\ e)$	Lambdas
		$\mid (e_0\ e_1)$	Applications

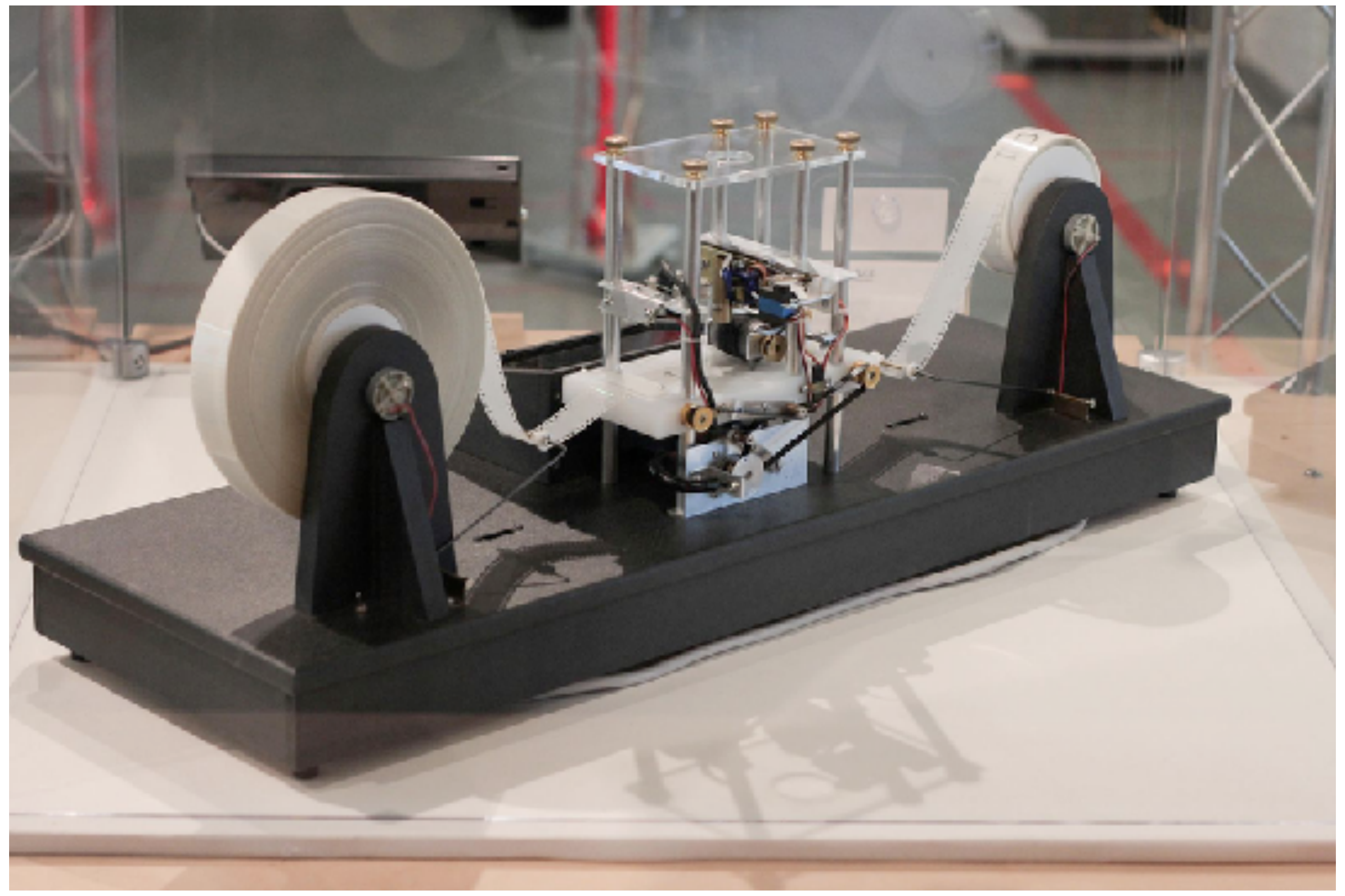
```
(define (expr? e)
  (match e
    [(? symbol? x) #t]
    [`(lambda (,(? symbol? x)) ,(? expr? e-body)) #t]
    [`(,(? expr? e0) ,(? expr? e1)) #t]
    [_ #f]))
```

Lambda Calculus vs. Turing machines

Lambda Calculus equivalent (in expressivity) to Turing

machines.

The **Church-Turing Thesis** states that turing machines / lambda calculus can encode any computable function.



In fact, it is possible to encode (most of) any Scheme program as a lambda calculus expression via a **Church/Boehm encoding**.

Now let's look at the three lambda calculus forms in detail…

An expression, *abstracted* over all possible values for a formal parameter, in this case, x.

$$(\lambda\ (x)\ e)$$

Formal parameter

Function body

An expression, *abstracted* over all possible values for a formal parameter, in this case, x.

$$(\lambda\ (x)\ e)$$

Formal parameter

Function body

In fact, you can read lambdas *mathematically* as "**for all**." This observation forms the basis for universal quantification in higher-order logics implemented using typed lambda calculus variants!

Next we have **applications**

$$(e \quad e)$$

Expression in ***function position***

Expression in ***argument position***

Variables are only defined/assigned when a function is applied and its parameter bound to an argument.

$$x$$

Variable reference

How do we compute with the lambda calculus..?

Answer: via **reductions**, which define equivalent / transformed terms.

The **most important** reduction is β, which applies a function by substituting arguments

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
```

The **most important** reduction is β, which applies a function by substituting arguments

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
```

$$\downarrow \beta$$

```
((λ (x) x) ((λ (x) x) (λ (x) x)))
```

The **most important** reduction is β, which applies a function by substituting arguments

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
        ↓ β
((λ (x) x) ((λ (x) x) (λ (x) x)))
        ↓ β
((λ (x) x) (λ (x) x))
```

The **most important** reduction is β, which applies a function by substituting arguments

```
((λ (f) (f (f (λ (x) x)))) (λ (x) x))
```

↓ β

```
((λ (x) x) ((λ (x) x) (λ (x) x)))
```

↓ β

```
((λ (x) x) (λ (x) x))
```

↓ β

```
(λ (x) x)
```

Textual substitution. This says:
replace every x in E_0 with E_1.

```
((λ (x) E₀) E₁)     →β     E₀[x ← E₁]
```

redex

(**red**ucible **ex**pression)

Next lecture: carefully defining substitution!

```
((λ (x) x) (λ (x) x))
```

$$\downarrow \beta$$

```
x[x ← (λ (x) x)]
```

```
((λ (x) x) (λ (x) x))
```

$$\downarrow \beta$$

```
(λ (x) x)
```

Can you beta-reduce the following term more than once…?

```
((λ (x) (x x)) (λ (x) (x x)))
```

```
((λ (x) (x x)) (λ (x) (x x)))
```

β reduction may continue indefinitely (i.e., in non-terminating programs)

↓ β

```
((λ (x) (x x)) (λ (x) (x x)))
```

↓ β

```
((λ (x) (x x)) (λ (x) (x x)))
```

↓ β

```
((λ (x) (x x)) (λ (x) (x x)))
```

↓ β

```
(λ (x) x x) (λ (x) x x)
```

↓ β

```
((λ (x) (x x)) (λ (x) (x x)))
```

β

```
((λ (x) (x x)) (λ (x) (x x)))
```

This specific program is known as Ω (Omega)

β

```
((λ (x) (x x)) (λ (x) (x x)))
```

β

```
((λ (x) (x x)) (λ (x) (x x)))
```

β

```
(λ (x) x x) (λ (x) x x)
```

β

Ω is the smallest non-terminating program!

Note how it reduces to itself in a single step!

```
((λ (x) (x x)) (λ (x) (x x)))
```

$$\Big\downarrow \beta$$

```
((λ (x) (x x)) (λ (x) (x x)))
```