

### Administrivia

There is still some flexibility in recitation assignments; if you find you need to switch sections, send mail to cis500@seas.

CIS 500, 22 September

The Lambda Calculus

The lambda-calculus
If our previous language of arithmetic expressions was the simplest nontrivial programming language, then the lambda-calculus is the simplest interesting programming language...
Turing complete
higher order (functions as data)
main new feature: variable binding and lexical scope
The e. coli of programming language research
The foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...)

# Intuitions Intuitions Suppose we want to describe a function that adds three to any number we pass it. We might write \$uppose we want to describe a function that adds three to any number we pass it. We might write plus3 x = succ (succ (succ x)) That is, "plus3 x is succ (succ (succ x))." That is, "plus3 x is succ (succ (succ x))." Case of the succ (succ (succ x))." Cls 500, 22 September 5

### Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x = succ (succ (succ x))
```

That is, "plus3 x is succ (succ (succ x))."

Q: What is plus3 itself?

A: plus3 is the function that, given x, yields succ (succ (succ x)).

### Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

plus3 x = succ (succ (succ x))

That is, "plus3 x is succ (succ (succ x))."

Q: What is plus3 itself?

A: plus3 is the function that, given x, yields succ (succ (succ x)).

plus3 =  $\lambda x$ . succ (succ (succ x))

This function exists independent of the name plus3.

5-a

### Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x = succ (succ (succ x))
```

That is, "plus3 x is succ (succ (succ x))."

```
Q: What is plus3 itself?
```

A: plus3 is the function that, given x, yields succ (succ x)).

plus3 =  $\lambda x$ . succ (succ (succ x))

This function exists independent of the name plus3.

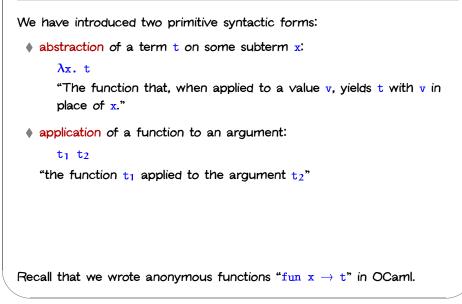
On this view, plus3 (succ 0) is just a convenient shorthand for "the function that, given x, yields succ (succ (succ x)), applied to succ 0."

plus3 (succ 0) =  $(\lambda x. \text{ succ } (\text{succ } x)))$  (succ 0)

CIS 500, 22 September

```
5-d
```

### Essentials



CIS 500, 22 September

### Abstractions Returning Functions

Consider the following variant of g:

double =  $\lambda f. \lambda y. f (f y)$ 

I.e., double is the function that, when applied to a function f, yields a function that, when applied to an argument y, yields f (f y).

### Abstractions over Functions

Consider the  $\lambda$ -abstraction

 $g = \lambda f. f (f (succ 0))$ 

Note that the parameter variable f is used in the function position in the body of g. Terms like g are called higher-order functions.

If we apply g to an argument like plus3, the "substitution rule" yields a nontrivial computation:

g plus3 =  $(\lambda f. f (f (succ 0))) (\lambda x. succ (succ (succ x)))$ i.e.  $(\lambda x. succ (succ (succ x)))$ 

 $((\lambda x. succ (succ (succ x))) (succ 0))$ 

- i.e.  $(\lambda x. \text{ succ } (\text{succ } x)))$ 
  - (succ (succ (succ (succ 0))))
- i.e. succ (succ (succ (succ (succ (succ ())))))





As the preceding examples suggest, once we have  $\lambda$ -abstraction and application, we can throw away all the other language primitives and still have left a rich and powerful programming language.

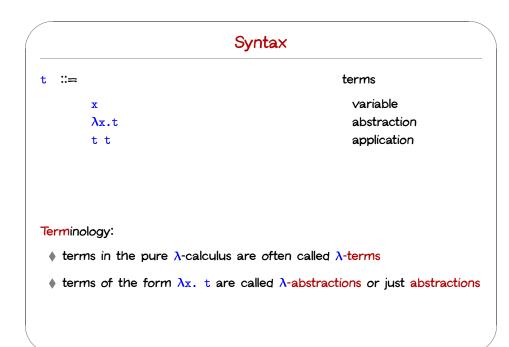
In this language - the "pure lambda-calculus"- everything is a function.

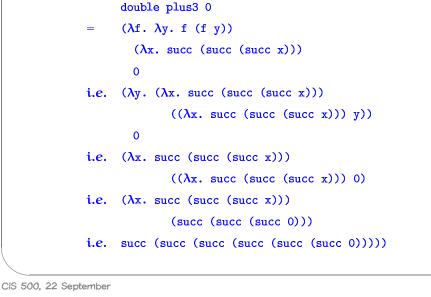
Variables always denote functions

• Functions always take other functions as parameters

• The result of a function is always a function

CIS 500, 22 September





Formalities

9

### Scope

The  $\lambda$ -abstraction term  $\lambda x.t$  binds the variable x.

The scope of this binding is the body t.

Occurrences of x inside t are said to be bound by the abstraction.

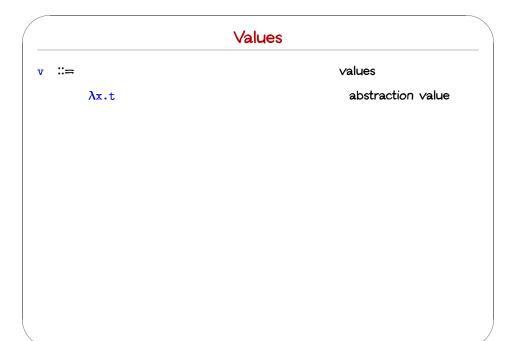
Occurrences of x that are not within the scope of an abstraction binding x are said to be free.

 $\lambda x. \lambda y. x y z$ 

CIS 500, 22 September

\_\_\_\_\_

13



Scope

The  $\lambda$ -abstraction term  $\lambda x.t$  binds the variable x.

The scope of this binding is the body t.

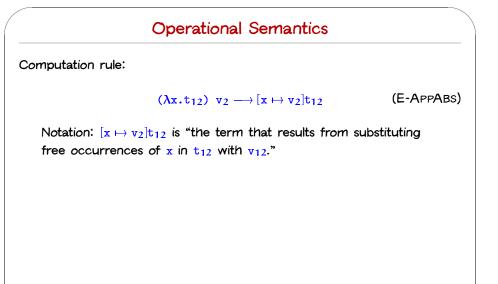
Occurrences of x inside t are said to be bound by the abstraction.

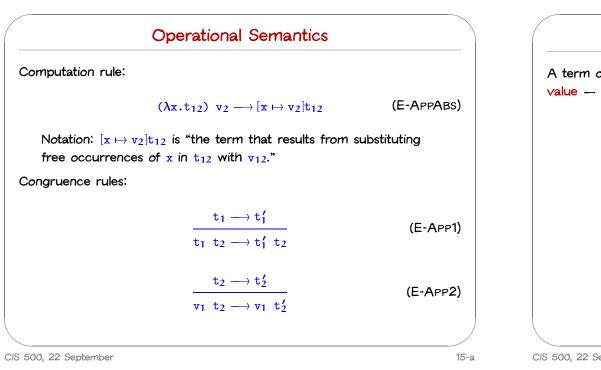
Occurrences of x that are not within the scope of an abstraction binding x are said to be free.

λx. λy. x y z λx. (λy. z y) y

CIS 500, 22 September

13-a





### Terminology

A term of the form  $(\lambda x.t) v$  — that is, a  $\lambda$ -abstraction applied to a value - is called a redex (short for "reducible expression").

CIS 500, 22 September

Alternative evaluation strategies

Strictly speaking, the language we have defined is called the pure, call-by-value lambda-calculus.

The evaluation strategy we have chosen - call by value - reflects standard conventions found in most mainstream languages.

Some other common ones:

- Call by name (cf. Haskell)
- Normal order (leftmost/outermost)
- Full (non-deterministic) beta-reduction

Programming in the Lambda-Calculus

### Multiple arguments

Above, we wrote a function double that returns a function as an argument.

double =  $\lambda f. \lambda y. f (f y)$ 

This idiom — a  $\lambda$ -abstraction that does nothing but immediately yield another abstraction — is very common in the  $\lambda$ -calculus.

In general,  $\lambda x$ .  $\lambda y$ . t is a function that, given a value v for x, yields a function that, given a value u for y, yields t with v in place of x and u in place of y.

That is,  $\lambda x$ .  $\lambda y$ . t is a two-argument function.

(Recall the discussion of currying in OCaml.)

CIS 500, 22 September





Since  $\lambda$ -calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

The following conventions make the linear forms of terms easier to read and write:

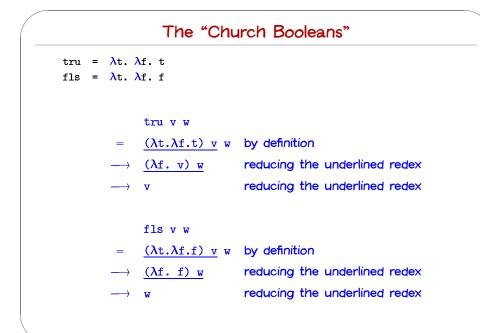
Application associates to the left

E.g., t u v means (t u) v, not t (u v)

 $\blacklozenge$  Bodies of  $\lambda$ - abstractions extend as far to the right as possible

E.g.,  $\lambda x$ .  $\lambda y$ . x y means  $\lambda x$ . ( $\lambda y$ . x y), not  $\lambda x$ . ( $\lambda y$ . x) y

CIS 500, 22 September



### Functions on Booleans

### not = $\lambda b. b fls tru$

That is, not is a function that, given a boolean value v, returns fls if v is tru and tru if v is fls.

### Functions on Booleans

and =  $\lambda b. \lambda c. b c fls$ 

That is, and is a function that, given two boolean values v and w, returns w if v is tru and fls if v is fls

Thus and  $v \in y$  yields tru if both v and w are tru and fls if either v or w is fls.

### Pairs

pair =  $\lambda f.\lambda s.\lambda b.$  b f s fst =  $\lambda p.$  p tru snd =  $\lambda p.$  p fls

That is, pair  $v \in v$  is a function that, when applied to a boolean value b, applies b to v and w.

By the definition of booleans, this application yields v if b is tru and w if b is fls, so the first and second projection functions fst and snd can be implemented simply by supplying the appropriate boolean.

CIS 500, 22 September

23

Example fst (pair v w) fst (( $\lambda f. \lambda s. \lambda b. b f s$ ) v w) by definition = fst (( $\lambda$ s.  $\lambda$ b. b v s) w) reducing the underlined redex fst ( $\lambda$ b. b v w) reducing the underlined redex  $\rightarrow$  $(\lambda p. p tru) (\lambda b. b v w)$ by definition =  $(\lambda b. b v w)$  tru reducing the underlined redex reducing the underlined redex tru v w as before.

### Church numerals

ldea: represent the number n by a function that "repeats some action n times."

```
c_0 = \lambda s. \lambda z. z
c_1 = \lambda s. \lambda z. s z
c_2 = \lambda s. \lambda z. s (s z)
c_3 = \lambda s. \lambda z. s (s (s z))
```

That is, each number n is represented by a term  $c_n$  that takes two arguments, s and z (for "successor" and "zero"), and applies s, n times, to z.

CIS 500, 22 September

Functions on Church Numerals				
uccessor:				

### Functions on Church Numerals

### Successor:

CIS 500, 22 September

scc =  $\lambda$ n.  $\lambda$ s.  $\lambda$ z. s (n s z)

CIS 500, 22 September

27

### Functions on Church Numerals

Successor:

scc =  $\lambda$ n.  $\lambda$ s.  $\lambda$ z. s (n s z)

Addition:

# Functions on Church Numerals Successor: scc = $\lambda$ n. $\lambda$ s. $\lambda$ z. s (n s z) Addition: plus = $\lambda m$ . $\lambda n$ . $\lambda s$ . $\lambda z$ . m s (n s z)

27-a

### Functions on Church Numerals

### Successor:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$ 

### Addition:

plus =  $\lambda m$ .  $\lambda n$ .  $\lambda s$ .  $\lambda z$ . m s (n s z)

### Multiplication:

CIS 500, 22 September

27-d

### Functions on Church Numerals

### Successor:

```
scc = \lambda n. \lambda s. \lambda z. s (n s z)
```

### Addition:

plus =  $\lambda m$ .  $\lambda n$ .  $\lambda s$ .  $\lambda z$ . m s (n s z)

### Multiplication:

times =  $\lambda m$ .  $\lambda n$ . m (plus n) c<sub>0</sub>

### Zero test:

### Functions on Church Numerals

### Successor:

 $scc = \lambda n. \lambda s. \lambda z. s (n s z)$ 

### Addition:

plus =  $\lambda m$ .  $\lambda n$ .  $\lambda s$ .  $\lambda z$ . m s (n s z)

### Multiplication:

times =  $\lambda m$ .  $\lambda n$ . m (plus n) c<sub>0</sub>

CIS 500, 22 September

 Functions on Church Numerals

 Successor:
 scc =  $\lambda n. \lambda s. \lambda z. s (n s z)$  

 Addition:
 plus =  $\lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$  

 Multiplication:
 times =  $\lambda m. \lambda n. m (plus n) c_0$  

 Zero test:
 iszro =  $\lambda m. m (\lambda x. fls) tru$ 

27-е

### Functions on Church Numerals Predecessor $zz = pair c_0 c_0$ Successor: scc = $\lambda n$ . $\lambda s$ . $\lambda z$ . s (n s z) ss = $\lambda$ p. pair (snd p) (scc (snd p)) Addition: plus = $\lambda m$ . $\lambda n$ . $\lambda s$ . $\lambda z$ . m s (n s z) Multiplication: times = $\lambda m$ . $\lambda n$ . m (plus n) c<sub>0</sub> Zero test: iszro = $\lambda$ m. m ( $\lambda$ x. fls) tru What about predecessor? CIS 500, 22 September 27-h CIS 500, 22 September 28

# Predecessor $zz = pair c_0 c_0$ $ss = \lambda p. pair (snd p) (scc (snd p))$ $prd = \lambda m. fst (m ss zz)$

# Normal forms

### Recall:

- A normal form is a term that cannot take an evaluation step.
- A stuck term is a normal form that is not a value.

Are there any stuck terms in the pure  $\lambda$ -calculus?

Prove it.

### Normal forms

### Divergence

### Recall:

- A normal form is a term that cannot take an evaluation step.
- A stuck term is a normal form that is not a value.

Are there any stuck terms in the pure  $\lambda$ -calculus?

### Prove it.

Does every term evaluate to a normal form?

Prove it.

CIS 500, 22 September

29-a

omega = $(\lambda x. x x) (\lambda x. x x)$	
Note that omega evaluates in one step to itself!	
So evaluation of omega never reaches a normal form: it div	verges.
S 500, 22 September	

Iterated Applica	tion					
Suppose f is some $\lambda$ -abstraction, and consider the following term:						
$Y_f = (\lambda x. f(x x)) (\lambda x)$	. f (x x))					

### Iterated Application

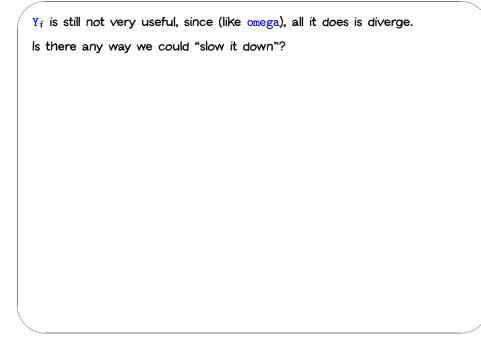
Suppose f is some  $\lambda$ -abstraction, and consider the following term:

$$Y_f = (\lambda x. f(x x)) (\lambda x. f(x x))$$

Now the "pattern of divergence" becomes more interesting:

CIS 500, 22 September

31-a

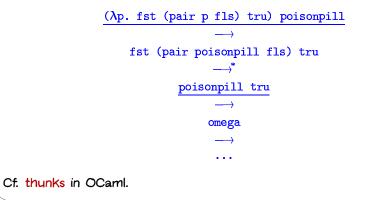


### CIS 500, 22 September

### **Delaying Divergence**

poisonpill =  $\lambda y$ . omega

Note that poisonpill is a value — it it will only diverge when we actually apply it to an argument. This means that we can safely pass it as an argument to other functions, return it as a result from functions, etc.



# A delayed variant of omega Here is a variant of omega in which the delay and divergence are a bit more tightly intertwined: omegav = $\lambda y. (\lambda x. (\lambda y. x x y)) (\lambda x. (\lambda y. x x y)) y$ Note that omegav is a normal form. However, if we apply it to any argument v, it diverges: $\begin{array}{r} & & \\ &$

### Another delayed variant

Suppose f is a function. Define

 $Z_f = \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y$ 

This term combines the "added f" from  $Y_f$  with the "delayed divergence" of onegav.

CIS 500, 22 September

```
\begin{array}{rcl} & & & \\ & & & \\ \text{Let} & & \\ & & f &= & \lambda \text{fct.} & \\ & & & \lambda n. & \\ & & & & \text{if n=0 then 1} & \\ & & & & \text{else n * (fct (pred n))} \end{array}
```

f looks just the ordinary factorial function, except that, in place of a recursive call in the last time, it calls the function fct, which is passed as a parameter.

N.b.: for brevity, this example uses "real" numbers and booleans, infix syntax, etc. It can easily be translated into the pure lambda-calculus (using Church numerals, etc.).

Since  $Z_f$  and v are both values, the next computation step will be the reduction of  $f Z_f$  — that is, before we "diverge," f gets to do some computation.

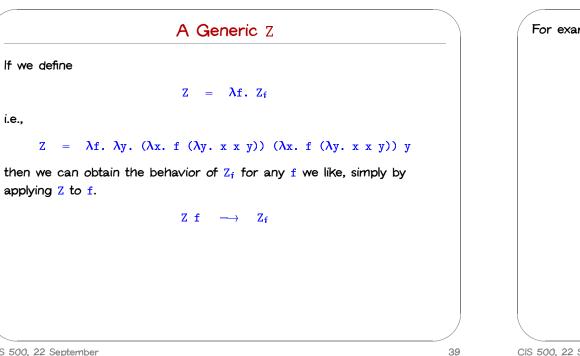
Now we are getting somewhere.

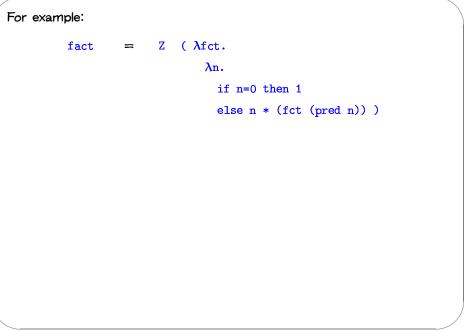
CIS 500, 22 September

36

We can use Z to "tie the knot" in the definition of f and obtain a real recursive factorial function:

```
Z_{f} 3
\longrightarrow^{*}
f Z_{f} 3
=
(\lambda fct. \lambda n. ...) Z_{f} 3
\longrightarrow \longrightarrow
if 3=0 then 1 else 3 * (Z_{f} (pred 3))
\longrightarrow^{*}
3 * (Z_{f} (pred 3)))
\longrightarrow
3 * (Z_{f} 2)
\longrightarrow^{*}
3 * (f Z_{f} 2)
\dots
```





40

CIS 500, 22 September

CIS 500, 22 September

Technical note: The term Z here is essentially the same as the fix discussed the book. Z =  $\lambda f. \lambda y. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y)) y$ fix =  $\lambda f. (\lambda x. f (\lambda y. x x y)) (\lambda x. f (\lambda y. x x y))$ Z is hopefully slightly easier to understand, since it has the property that Z f v  $\longrightarrow$  f (Z f) v, which fix does not (quite) share.