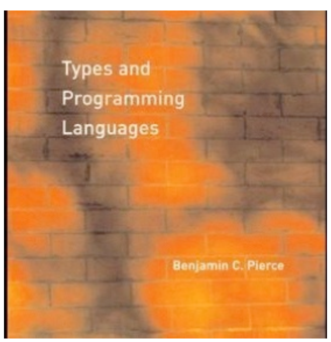
Natural and Contextual Semantics

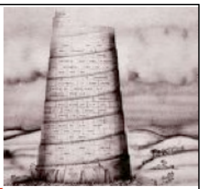
Lecture 4
CS 565



Natural Semantics

- The semantics given previously is known as "small-step"
  - Evaluation relation shows how each individual step in the computation takes place
  - Closely mirrors how an interpreter might evaluate a program
  - Apply a multi-step evaluation relation $\rightarrow^*$ on top to talk about terms evaluating (in many steps) to values
- An alternative style called "natural semantics" directly formulates the notion of "this term evaluates to this value"
  - (Details omitted)

Evaluation Contexts



- Both styles of semantics address two concerns:
  - order of evaluation

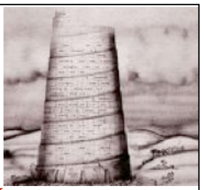
    explicit in small-step semantics

    implicit in natural semantics
  - meaning of terms
- Can we separate out these two notions?
  - Decompose a term into two parts:

    - the part of the term that is to be evaluated

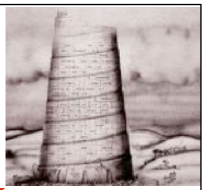
    - the remaining portion of the term that should be examined after the subterm evaluates; call this part of the term a “context”

Contextual semantics



- Small-step semantics where the atomic execution step is a rewrite of the program
  - Evaluation terminates when program has been rewritten to a terminal program
  - For IMP terminal command is “skip”
- Need to define
  - What constitutes an atomic reduction step
  - How to select the next reduction step

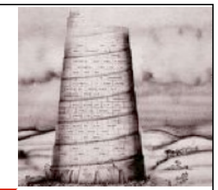
Redex



- A redex is a term that can be transformed in a single step
  - A redex has no antecedents

```
r ::= x | x := int | int + int' | skip; c |
        if true then c1 else c2 |
        if false then c1 else c2 |
        true and b | false or b |
        ....
```

Evaluation Contexts



- An evaluation context is a term with a "hole" in the place of a subterm
  - Location of the hole points to the next subexpression that should be evaluated
  - If $\mathtt{E}$ is a context then $\mathtt{E[r]}$ is the expression obtained by replacing redex $\mathtt{r}$ for the hole defined by context $\mathtt{E}$
  - Now, if $\mathtt{r},\sigma \rightarrow \mathtt{t},\sigma'$ then $\mathtt{E[r]},\sigma \rightarrow \mathtt{E[t]},\sigma'$

Global reduction rule + Local reduction rules for individual $\mathtt{r}$

Contexts

- Can define evaluation context via a grammar:

```
E ::= [] | n + E | E + e | x := E |
       if E then c1 else c2 |
       E; c | ...
```

- The grammar fixes the order of evaluation, allowing us to simplify the number and structure of the rules used in the semantics

Evaluation Contexts

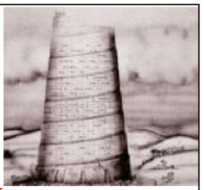
- A context has exactly one hole
- Redexes that are substituted for a context are never values
- A context uniquely identifies the next redex to be evaluated

- Consider `e1+e2` and its decomposition as `E[r]`

  - If `e1=n1` and `e2=n2` then `E=[]` and `r=n1+n2`
  - If `e1=n1` and `e2≠n2` then `E=n1+E'` and `e2=E'[r]`
  - If `e1≠n1` then `E=E'+e2` and `e1=E'[r]`

  Last two cases are evaluated recursively

Evaluation Contexts

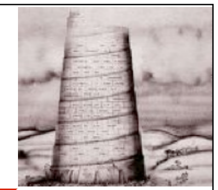


- Consider `c = c1;c2`
  - Suppose `c1 = skip`.

    Then, `c=E[skip;c2]` with `E=[ ]`
  - Suppose `c1 ≠ skip`.

    Then, `c1=E[r]` and `c=E'[r]` with `E'=E;c2`
- Consider `c = if b then c1 else c2`
  - If `b=true` then `c=E[r]` where `r` is a redex in `c1` and `E` defines its context
  - If `b=false` then `c=E[r]` where `r` is a redex in `c2` and `E` defines its context
  - Otherwise, `b=E[r]`, so `c =E'[r]` where `E'= if E then c1 else c2`

Evaluation Contexts



- Decomposition theorem:
  - If `c≠skip` then there exists unique `E`, `r` such that `c=E[r]`

exists ⇒ progress

unique ⇒ determinism

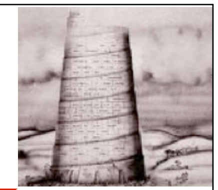
Example

Consider the evaluation of:

`x:=1; x:=x+1 with σ = [x -> 0]`

State	Context	Redex
`x:=1;x:= x+1,σ`	`[];x:=x+1`	`x:=1`
`skip;x:=x+1,[x->1]`	`[]`	`skip; x:= x+1`
`x:= x+1,[x->1]`	`x := [] + 1`	`x`
`x:= 1+1,[x->1]`	`x: = []`	`1 + 1`
`x:= 2,[x->1]`	`[]`	`x := 2`
`skip, [x->2]`		

IMP



$$
\begin{array}{lcll}
E & ::= & & \text{Contexts} \\
 & | & [\,] & \\
 & | & E = a_2 & \\
 & | & int = E & \\
 & | & E < a_2 & \\
 & | & int < E & \\
 & | & E \textbf{ and } b_2 & \\
 & | & bool \textbf{ and } E & \\
 & | & E + a_2 & \\
 & | & int + E & \\
 & | & E * a_2 & \\
 & | & int * E & \\
 & | & E - a_2 & \\
 & | & int - E & \\
 & | & \mathbf{x} := E & \\
 & | & E \,;\, c_2 & \\
 & | & \textbf{if } E \textbf{ then } c_1 \textbf{ else } c_2 &
\end{array}
$$

IMP

$$\boxed{c\,,\,\sigma \;\Longrightarrow\; c'\,,\,\sigma'}$$

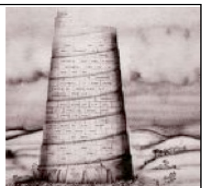
$$\frac{\begin{array}{l} c = \mathbf{E}[\mathbf{r}] \\ r\,,\,\sigma \;\longrightarrow\; r'\,,\,\sigma' \\ c' = \mathbf{E}[\mathbf{r}'] \end{array}}{c\,,\,\sigma \;\Longrightarrow\; c'\,,\,\sigma'}\ \textsc{Ctxt}$$

$$\frac{}{\mathbf{skip}\,;\,c\,,\,\sigma \;\Longrightarrow\; c\,,\,\sigma}\ \textsc{Skip}$$

IMP

$$\boxed{r\ ,\ \sigma\ \longrightarrow\ r'\ ,\ \sigma'}$$

change r' to t'



$$\frac{\sigma\,(\,\mathbf{x}\,) = \mathit{int}}{\mathbf{x}\ ,\ \sigma\ \longrightarrow\ \mathit{int}\ ,\ \sigma}\ \textsc{AexpVar}$$

$$\frac{\mathit{int}_1\ +\ \mathit{int}_2 = \mathit{int}_3}{\mathit{int}_1\ +\ \mathit{int}_2\ ,\ \sigma\ \longrightarrow\ \mathit{int}_3\ ,\ \sigma}\ \textsc{AexpPlus}$$

$$\frac{\mathit{int}_1\ *\ \mathit{int}_2 = \mathit{int}_3}{\mathit{int}_1\ *\ \mathit{int}_2\ ,\ \sigma\ \longrightarrow\ \mathit{int}_3\ ,\ \sigma}\ \textsc{AexpTimes}$$

$$\frac{\mathit{int}_1\ -\ \mathit{int}_2 = \mathit{int}_3}{\mathit{int}_1\ -\ \mathit{int}_2\ ,\ \sigma\ \longrightarrow\ \mathit{int}_3\ ,\ \sigma}\ \textsc{AexpSub}$$

$$\frac{}{\mathit{int}_1 = \mathit{int}_2\ ,\ \sigma\ \longrightarrow\ \texttt{true}\ ,\ \sigma}\ \textsc{BexpEq}$$

$$\frac{\mathit{int}_1\ \neq\ \mathit{int}_2}{\mathit{int}_1 = \mathit{int}_2\ ,\ \sigma\ \longrightarrow\ \texttt{false}\ ,\ \sigma}\ \textsc{BexpNEq}$$

$$\frac{}{\mathbf{not}\,\texttt{true}\ ,\ \sigma\ \longrightarrow\ \texttt{false}\ ,\ \sigma}\ \textsc{BexpNotT}$$

$$\frac{}{\mathbf{not}\,\texttt{false}\ ,\ \sigma\ \longrightarrow\ \texttt{true}\ ,\ \sigma}\ \textsc{BexpNotF}$$

$$\frac{\mathit{bool}_1\ \mathbf{and}\ \mathit{bool}_2 = \mathit{bool}}{\mathit{bool}_1\ \mathbf{and}\ \mathit{bool}_2\ ,\ \sigma\ \longrightarrow\ \mathit{bool}\ ,\ \sigma}\ \textsc{BexpAnd}$$

$$\frac{\mathit{bool}_1\ \mathbf{or}\ \mathit{bool}_2 = \mathit{bool}}{\mathit{bool}_1\ \mathbf{or}\ \mathit{bool}_2\ ,\ \sigma\ \longrightarrow\ \mathit{bool}\ ,\ \sigma}\ \textsc{BexpOr}$$

$$\frac{\sigma' = \sigma\,[\,\mathbf{x}\ \mapsto\ \mathit{int}\,]}{\mathbf{x} := \mathit{int}\ ,\ \sigma\ \longrightarrow\ \mathbf{skip}\ ,\ \sigma'}\ \textsc{Assign}$$

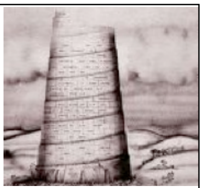
$$\frac{}{\mathbf{if}\,\texttt{true}\,\mathbf{then}\ c_1\ \mathbf{else}\ c_2\ ,\ \sigma\ \longrightarrow\ c_1\ ,\ \sigma}\ \textsc{IfT}$$

$$\frac{}{\mathbf{if}\,\texttt{false}\,\mathbf{then}\ c_1\ \mathbf{else}\ c_2\ ,\ \sigma\ \longrightarrow\ c_3\ ,\ \sigma}\ \textsc{IfF}$$

$$\frac{}{\mathbf{while}\ b\ \mathbf{do}\ c_1\ ,\ \sigma\ \longrightarrow\ \mathbf{if}\ b\ \mathbf{then}\ c_1\ ;\ \mathbf{while}\ b\ \mathbf{do}\ c_1\ \mathbf{else}\,\mathbf{skip}\ ,\ \sigma}$$

IMP

$$\boxed{r\,,\,\sigma \longrightarrow r'\,,\,\sigma'}$$



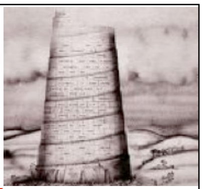
$$\frac{\sigma' = \sigma\,[\,\mathbf{x} \mapsto \mathit{int}\,]}{\mathbf{x} := \mathit{int}\,,\,\sigma \longrightarrow \mathbf{skip}\,,\,\sigma'}\ \textsc{Assign}$$

$$\frac{}{\mathbf{if}\ \mathtt{true}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\,,\,\sigma \longrightarrow c_1\,,\,\sigma}\ \textsc{IfT}$$

$$\frac{}{\mathbf{if}\ \mathtt{false}\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\,,\,\sigma \longrightarrow c_3\,,\,\sigma}\ \textsc{IfF}$$

$$\frac{}{\mathbf{while}\ b\ \mathbf{do}\ c_1\,,\,\sigma \longrightarrow \mathbf{if}\ b\ \mathbf{then}\ c_1\,;\,\mathbf{while}\ b\ \mathbf{do}\ c_1\ \mathbf{else}\,\mathbf{skip}\,,\,\sigma}$$

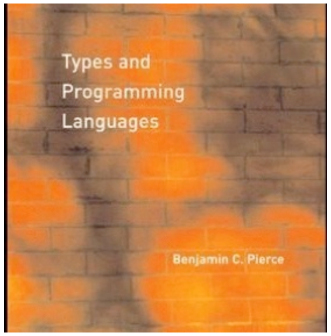
Contextual Semantics



- Summary
  - Think of a hole as representing a program counter
    - The rules for advancing holes is non-trivial
      - Must decompose entire command at every step
      - How would you implement this?
  - Major advantage of contextual semantics is that allows a mix of global and local reduction rules
  - Global rules indicate next redex to be evaluated
    - defined by contexts
  - Local rules indicate how to perform the reduction
    - one for each redex

Introduction to Lambda Calculus

Lecture 5
CS 565

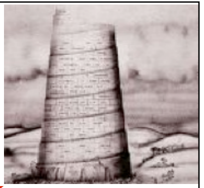


Lambda Calculus

- So far, we've explored some simple but non-interesting languages
  - language of arithmetic expressions
  - IMP (arithmetic + while loops)
- We now turn our attention to a simple but interesting language
  - Turing complete (can express loops and recursion)
  - Higher-order (functional objects are values)
  - Interesting variable binding and scoping issues
  - Foundation for many real-world programming languages

    Lisp, Scheme, ML, Haskell, ...

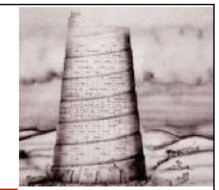
Intuition



- Suppose we want to describe a function that adds three to any input:
  - `plus3 x = succ (succ (succ x))`
  - Read "*plus3 is a function which, when applied to any number x, yields the successor of the successor of the successor of x*"
  - Note that the function which adds 3 to any number need not be named plus3; the name "plus3" is just a convenient shorthand for naming this function

```
(plus3 x) (succ 0) ≡
      ((λ x.(succ (succ (succ x)))) (succ 0))
```

Basics



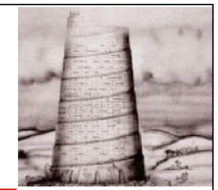
- There are two new primitive syntactic forms:
  - `λ x.t`

    "The function which when given a value `v`, yields `t` with `v` substituted for `x` in `t`."
  - `(t1 t2)`

    "the function `t1` applied to argument `t2`"
  - Key point: functions are anonymous: they don't need to be named. For convenience we'll sometimes write:

    `plus3 ≡ λ x. (succ (succ (succ x)))`
  - but the naming is a metalanguage operation.

Abstractions



- Consider the abstraction:

```
g ≡ λ f. (f (f (succ 0)))
```

- The argument `f` is used in a function position (in a call).
- We call `g` a higher-order function because it takes another function as an input.
- Now, `(g plus3)`

```
= (λ f. (f (f (succ 0)))
  (λ x .(succ (succ (succ x))))
= ((λ x.(succ (succ (succ x)))
  ((λ x.(succ (succ (succ x))))(succ 0)))
= ((λ x. (succ (succ (succ x))))
  (succ (succ (succ (succ 0)))))
= (succ(succ(succ(succ(succ(succ(succ 0)))))))
```

Abstractions

- Consider

```
double ≡ λ f. λ y. (f (f y))
```

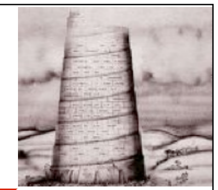
- The term yielded by applying double is another function
  `(λ y. (f (f y))`

- Thus, double is also a higher-order function because it returns a function when applied to an argument.

Example

```
(double plus3 0)

= ((λ f.λ y.(f (f y))) (λ x.(succ (succ (succ x)))) 0)

= ((λ y.((λ x.(succ (succ (succ x))))
    ((λ x. (succ (succ (succ x)))) y)))
   0)

= ((λ x. (succ (succ (succ x))))
   ((λ x. (succ (succ (succ x)))) 0))

= ((λ x. (succ (succ (succ x)))) (succ (succ (succ 0))))

= (succ (succ (succ (succ (succ (succ 0))))))
```

Key Issues



- How do we perform substitution:
  - how do we bind "free variables", the variables that are non-local in the function
  - Think about the occurrences of `f` in

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

- How do we perform application:
  - There may be several different application subterms within a larger term.
  - How do we decide the order to perform applications?

Pure Lambda Calculus

- The only value is a function
  - Variables denote functions
  - Functions always take functions as arguments
  - Functions always return functions as results
- Minimalist
  - Can express essentially all modern programming constructs
  - Can apply syntactic reasoning techniques (e.g. operational semantics) to understand behavior.

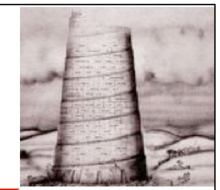
Scope

- The λ abstraction `λ x.t` binds variable `x`.
- The scope of the binding is `t`.
- Occurrences of `x` that are not within the scope of an abstraction binding `x` are said to be free:

```
λ x. λ y.(x y z)
λ x.((λ y.z y) y)
```

- Occurrences of `x` that are within the scope of an abstraction binding `x` are said to be bound by the abstraction.

Free Variables



- Intuitively, the free variables of an exp are “non-local” variables
  - Define `FV(M)` formally thus:

```
FV(x)  = {x}
FV(M1 M2) = FV(M1)  U  FV(M2)
FV(λ x. M) = FV(M) – {x}
```

  - Free variables become bound after substitution.
  - But, if proper care is not taken, this leads to unexpected results:

```
(λx.λy. y x) y = λy. y y
```

  - We say that term `M` is α-congruent to `N` if `N` results from `M` by a series of changes to bound variables:

`λx.(x y)` α-congruent to `λz.(z y)`
not α-congruent to `λy.(y y)`
`λx.x (λx.x)` α-congruent to `λx'.x'(λx.x)` and
α-congruent to `λx'.x'(λx''.x'')`

Substitution

- λ`x.M` α-congruent to λ`y.M[y/x]` if `y` is not free or bound in `M`.
  - Want to define substitution s.t. `(λx.N)M → [M/x]N`
- Define this more precisely:
  - Let `x` be a variable, and `M` and `N` expressions.
    Then `[M/x]N` is the expression `N'`:

    `N` is a variable: (case 1)

    `N = x` then `N' = M` (1.1)

    `N ≠ x` then `N' = N` (1.2)

    `N` is an application `(Y Z)`: (case 2)

    `N' = ([M/x]Y) ([M/x]Z)`

Substitution (cont)

- N is $\lambda y.Y$ (then $[M/x]N$ is the expression N') (case 3)
  - $y = x$ then $N' = N$ (3.1)
  - $y \neq x$ then:

    x does not occur free in Y or if y does not occur free in M:

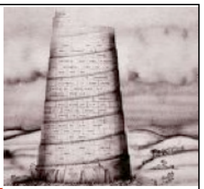
    $N' = \lambda y.[M/x]Y$ (3.2.1)

    x does occur free in Y and y does occur free in M:

    $N' = \lambda z.[M/x]([z/y]Y)$ for fresh z (3.2.2)

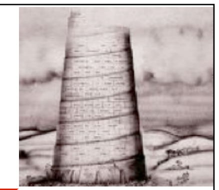
    First change bound variable y in Y to z, then perform substitution

Example



```
(λp.(λq.(λp.p( p q))(λr.(+ p r)))(+ p 4)) 2

(λq.(λp.p( p q))(λr.(+ 2 r)))(+ 2 4)

[+ 2  4/q](λp.p( p q))(λr.(+ 2 r))

(λp.p( p (+ 2 4)))(λr.(+ 2 r))

(λr.(+ 2 r))( (λr.(+ 2 r)) (+ 2 4)))
```

Operational Semantics



- Values:

  $$\lambda\ x.\ t$$

- Computation rule:

  $$((\lambda\ x.\ t)\ v) \longrightarrow [v/x]t$$

- Congruence rules

  $$\frac{t1 \longrightarrow t1'}{(t1\ t2) \longrightarrow (t1'\ t2)}$$

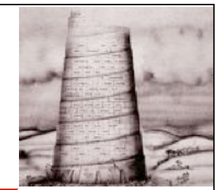
  $$\frac{t2 \longrightarrow t2'}{(v\ t2) \longrightarrow (v\ t2')}$$

The computation rule is referred to as the β-substitution or β-conversion rule. $((\lambda\ x.\ t)t')$ is called a β-redex.

Evaluation Order

- Outermost, leftmost redex first
- Arguments to application are evaluated before application is performed
  - Call-by-value
  - “Strict”
- Other orders do not evaluate arguments before application
  - E.g. normal order
  - “Lazy”

Example



$(\lambda x.x)\ ((\lambda x.x)\ (\lambda z.(\lambda x.x)\ z))$

$id\ (id\ (\lambda z.id\ z))$ (with $id \equiv \lambda x.x$)

Call-by-value (strict):

$$
\begin{aligned}
& id\ \underline{(id\ (\lambda z.\ id\ z))} \\
= {} & \underline{id\ (\lambda z.\ id\ z)}
\end{aligned}
$$

(1st id would come 1st, but arg must be evaluated)

$$= \lambda z.\ id\ z$$

Normal order (lazy):

$$
\begin{aligned}
& \underline{id\ (id\ (\lambda z.\ id\ z))} \\
= {} & \underline{id\ (\lambda z.\ id\ z)} \\
= {} & \lambda z.\ \underline{id\ z} \\
= {} & \lambda z.z
\end{aligned}
$$

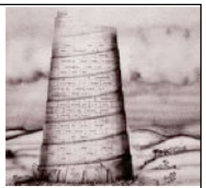
Multiple arguments

- The λ calculus has no built-in support to handle multiple arguments.
- However, we can interpret λ terms that when applied yield another λ term as effectively providing the same effect:
- Example:

```
double ≡ λ f. λ x. (f (f x))
```

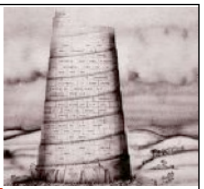
  - We can think of double as a two-argument function.
- Representing a multi-argument function in terms of single-argument higher-order functions is known as currying.

Programming Examples: Booleans



```
true ≡ λ t. λ f. t
false  ≡ λ t. λ f. f


(true v w) ≡ ((λ t.λ f. t) v) w) →
             ((λ f. v) w) →
             v
(false v w) ≡ ((λ t.λ f. f) v) w) →
              ((λ f. f) w) →
               w
```

Booleans (cont)



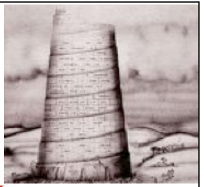
```
not ≡ λ b. b false true
```

The function that returns true if b is false, and false if b is true.

```
and ≡ λ b. λ c. b c false
```

The function that given two Boolean values (v and w) returns w if v is true and false if v is false. Thus, (and v w) yields true only if both v and w are true.

Pairs



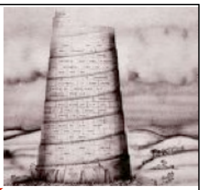
- We can encode common operations on pairs thus:

$$
\begin{array}{lll}
\texttt{pair} & \equiv & \lambda f.\ \lambda s.\ \lambda b.b\ f\ s \\
\texttt{fst} & \equiv & \lambda p.\ p\ \texttt{true} \\
\texttt{snd} & \equiv & \lambda p.\ p\ \texttt{false}
\end{array}
$$

Example:

$$
\begin{array}{ll}
\texttt{fst}\ (\texttt{pair}\ v\ w) & \equiv \\
\texttt{fst}\ (\underline{(\lambda f.\ \lambda s.\ \lambda b.b\ f\ s)v}\ w) & \rightarrow \\
\texttt{fst}\ ((\lambda s.\ \lambda b.b\ v\ s)w) & \equiv \\
\underline{(\lambda p.p\ \texttt{true})(\lambda b.(b\ v\ w)} & \rightarrow \\
\underline{(\lambda b.b\ v\ w)\ \texttt{true}} & \rightarrow \\
\texttt{true}\ v\ w & \rightarrow^{*}\ v
\end{array}
$$

Numbers (Church Numerals)

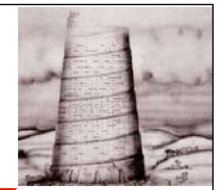


- There are no explicit operations to manipulate numbers
- Encode numbers with higher-order functions

```
zero ≡ λs. λz. z
one  ≡ λs. λz. s z
two  ≡ λs. λz. s (s z)
```

  - read s as successor and z as zero

Numbers



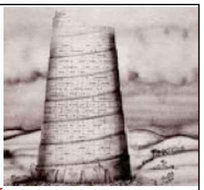
```
succ ≡ λn. λs. λz.s (n s z)
```

A function that takes s and z and applies s repeatedly to z

```
plus ≡ λm. λn. λs. λz.m s(n s z)
```

takes two Church numerals and yields another Church numeral that given s and z applies s iterated n times to z and then applies s iterated my times to the result

Example



```
(plus one two succ zero) ≡

(plus (λ s. λ z.(s z)) (λ s. λ z.(s (s z))) succ zero) ⟶
(λ s. λ z.((λ s. λ z.(s z)) s ((λ s. λ z.(s (s z))) s z)) succ zero) →
(λ s. λ z.((λ s. λ z.(s z)) s ((λ s. λ z.(s (s z))) s z)) succ zero) →

((λ s. λ z.(s z)) succ ((λ s. λ z.(s (s z))) succ zero)) ⟶

((λ s. λ z.(s z)) succ (succ ( succ zero))) ⟶

((λ s. λ z.(s z)) succ (succ ( succ zero))) ⟶

(succ (succ ( succ zero)))
```