INTRODUCTION TO

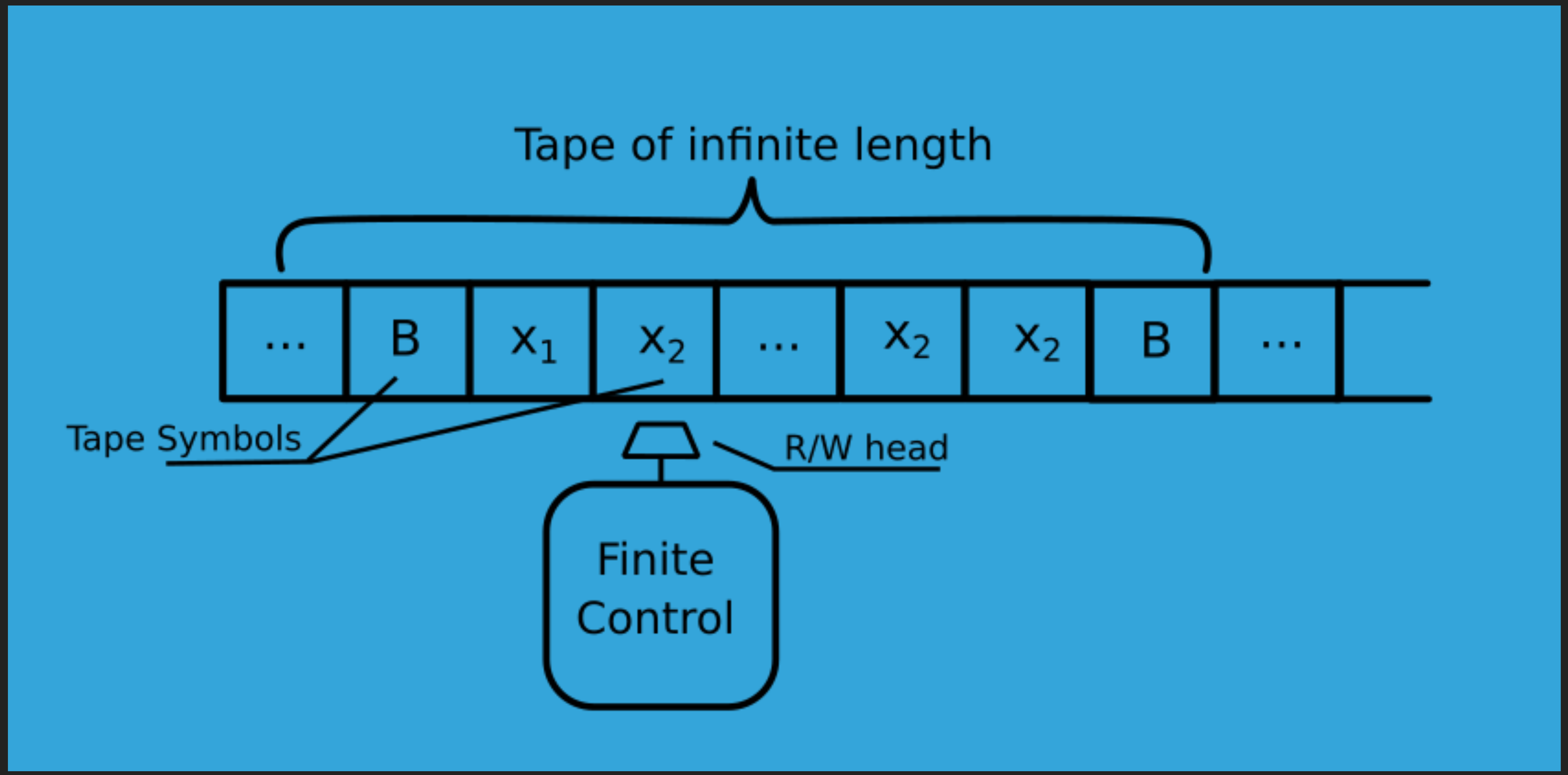
FUNCTIONAL PROGRAMMING

OUTLINE

- Functional Model of Computation
- Lambda Calculus
- Term Equivalence
- Expressing Common Data Types

SEQUENTIAL MODEL OF COMPUTATION

- Computation is given as a sequence of instructions that changes the state
- Instructions are done in sequence: s1; s2; s3
- Assignments x = 42
- Conditional statements: if, switch, while, for
- Computation terminates when the final instruction is reached



FUNCTIONAL MODEL OF COMPUTATION

- Program is an expression
- Computation is a reduction of the expression

```
1 * 2 + 3

let f x = x * 2 in f (13 + 42)
```

FUNCTIONAL MODEL OF COMPUTATION

- Computation
  - Find a redex – a subexpression that can be computed directly
  - Reduce the redex according to the rules expressed in terms of a substitution
  - Terminate when there aren't any redexes left
- Only one redex at each step
- New redexes may be created

```
1 * 2 + 3 -*→ 3 + 3 -+-> 6


let f x = x * 2 in f (13 + 42) -+->

let f x = x * 2 in f 55 -$→

let f x = x * 2 in 55 * 2 -*→

let f x = x * 2 in 110
```

VARIABLE BINDING

- Binding creates a new computation rule
- A bound variable becomes a redex

```
z = 5 * 2 + 3

f x = x * 2


f (z + 42) -z->

f (13 + 42) -+->

f 55 -f->

55 * 2 -*->

110
```

RECURSIVE VARIABLE BINDING

- We can use variable in the body of a binding

```
x = 2 * x + 3
```

RECURSIVE VARIABLE BINDING

- We can use variable in the body of a binding
- Computation can diverge in this case

```
x = 2 * x + 3


x - 1 -x→

2 * (2 * x + 3) + 3 - 1 -x→

2 * (2 * (2 * x + 3) + 3) - 1 -x→ …
```

ANONYMOUS FUNCTIONS

- An anonymous function is a value (no redexes)
- Regex only appear when there is application
  - *β-redex*
- *β-reduction*: computation rule that substitute the bound variable with the argument

```
\x → x * 2 + 3


(\x → x * 2 + 3) 42 -β→

42 * 2 + 3 -*→

84 + 3 -+->

87
```

REDUCTION STRATEGY

- What to do when there are two redexes?

```
(\x → 2 * x + 3) (13 + 42)
```

REDUCTION STRATEGY

- What to do when there are two redexes?
- Eager evaluation (ML):
  - first compute arguments, then substitute
- Lazy evaluation (Haskell):
  - reduce the leftmost redex

```
(\x → 2 * x + 3) (13 + 42)

Eager: (\x → 2 * x + 3) (13 + 42) -+->
(\x → 2 * x + 3) 55 -β→ 2 * 55 + 3 …

Lazy: (\x → 2 * x + 3) (13 + 42) -*→
(2 * (13 + 42) + 3) -+-> 2 * 55 + 3 …
```

RECURSIVE FUNCTION EVALUATION

```
fact = \n → if n == 0 then 1 else n * fact (n - 1)


fact 3 -fact→
```

RECURSIVE FUNCTION EVALUATION

```
fact = \n → if n == 0 then 1 else n * fact (n - 1)


fact 3 -fact→

(\n → if n == 0 then 1 else n * fact (n - 1)) 3 -β→
```

RECURSIVE FUNCTION EVALUATION

```
fact = \n -> if n == 0 then 1 else n * fact (n - 1)


fact 3 -fact->

(\n -> if n == 0 then 1 else n * fact (n - 1)) 3 -β->

if 3 == 0 then 1 else 3 * fact (3 - 1) -==->
```

RECURSIVE FUNCTION EVALUATION

```
fact = \n → if n == 0 then 1 else n * fact (n - 1)


fact 3 -fact→

(\n → if n == 0 then 1 else n * fact (n - 1)) 3 -β→

if 3 == 0 then 1 else 3 * fact (3 - 1) -==→

if False then 1 else 3 * fact (3 - 1) -if→
```

RECURSIVE FUNCTION EVALUATION

```
fact = \n → if n == 0 then 1 else n * fact (n - 1)


fact 3 -fact→

(\n → if n == 0 then 1 else n * fact (n - 1)) 3 -β→

if 3 == 0 then 1 else 3 * fact (3 - 1) -==→

if False then 1 else 3 * fact (3 - 1) -if→

3 * fact (3 - 1) -fact→
```

RECURSIVE FUNCTION EVALUATION

```
fact = \n -> if n == 0 then 1 else n * fact (n - 1)


fact 3 -fact->

(\n -> if n == 0 then 1 else n * fact (n - 1)) 3 -β->

if 3 == 0 then 1 else 3 * fact (3 - 1) -==->

if False then 1 else 3 * fact (3 - 1) -if->

3 * fact (3 - 1) -fact->

3 * if (3 - 1) == 0 then 1 else (3 - 1) * fact ((3 - 1) - 1) ---> …
```

RECURSIVE FUNCTION EVALUATION

```
fact = \n → if n == 0 then 1 else n * fact (n - 1)


fact 3 -fact→

(\n → if n == 0 then 1 else n * fact (n - 1)) 3 -β→

if 3 == 0 then 1 else 3 * fact (3 - 1) -==→

if False then 1 else 3 * fact (3 - 1) -if→

3 * fact (3 - 1) -fact→

3 * if (3 - 1) == 0 then 1 else (3 - 1) * fact ((3 - 1) - 1) ---> …

3 * (2 * (1 * if (((3 - 1) - 1) - 1) == 0 then 1 else …)) ---> …
```

RECURSIVE FUNCTION EVALUATION

```
fact = \n → if n == 0 then 1 else n * fact (n - 1)


fact 3 -fact→

(\n → if n == 0 then 1 else n * fact (n - 1)) 3 -β→

if 3 == 0 then 1 else 3 * fact (3 - 1) -==→

if False then 1 else 3 * fact (3 - 1) -if→

3 * fact (3 - 1) -fact→

3 * if (3 - 1) == 0 then 1 else (3 - 1) * fact ((3 - 1) - 1) ---> …

3 * (2 * (1 * if (((3 - 1) - 1) - 1) == 0 then 1 else …)) ---> …

3 * (2 * (1 * 1)) -*→ … -*→ 6
```

EXERCISE

- Find redexes in the following terms
  - `1+2*3/4`
  - `let f = \x → x * x in f (f (5 - f 2))`
  - `let f = \x → g (g x) in let g = \y → (-1) * y in 42`

OUTLINE

- Functional Model of Computation
- Lambda Calculus
- Term Equivalence
- Expressing Common Data Types

SYNTAX

- Lambda term is recursively defined from variables $V = \{x, y, z, \ldots\}$ by abstraction and application
- Abstract syntax:
  - $\Lambda ::= V \,|\, (\Lambda\Lambda) \,|\, (\lambda V . \Lambda)$

$$x \in V \Rightarrow x \in \Lambda$$
$$M \in \Lambda, x \in V \Rightarrow (\lambda x . M) \in \Lambda$$
$$M, N \in \Lambda \Rightarrow (M\,N) \in \Lambda$$

EXAMPLES OF TERMS

$$z$$

$$(xz)$$

$$(\lambda x\,.\,(xz))$$

$$((\lambda x\,.\,(xz))y)$$

$$(\lambda y\,.\,((\lambda x\,.\,(xz))y))$$

$$((\lambda y\,.\,((\lambda x\,.\,(xz))y))w)$$

$$(\lambda z\,.\,(\lambda w\,.\,((\lambda y\,.\,((\lambda x\,.\,(xz))y))w)))$$

CONCRETE SYNTAX

- No outer parentheses
  - $(x\,z) = x\,z$
- Application is left-associative
  - $F\,X\,Y\,Z = (((F\,X)\,Y)\,Z)$
- Abstraction is right-associative
  - $\lambda x\,y\,z\,.\,M = (\lambda x\,.\,(\lambda y\,.\,(\lambda z\,.\,M))$
- Abstraction's body lasts until EOF
  - $\lambda x\,.\,F\,X\,Y = \lambda x\,.\,(F\,X\,Y)$

$$
\begin{aligned}
z &= z \\
(xz) &= xz \\
(\lambda x\,.\,(xz)) &= \lambda x\,.\,xz \\
((\lambda x\,.\,(xz))y) &= (\lambda x\,.\,xz)y \\
(\lambda y\,.\,((\lambda x\,.\,(xz))y)) &= \lambda y\,.\,(\lambda x\,.\,xz)y \\
((\lambda y\,.\,((\lambda x\,.\,(xz))y))w) &= (\lambda y\,.\,(\lambda x\,.\,xz)y)w \\
(\lambda z\,.\,(\lambda w\,.\,((\lambda y\,.\,((\lambda x\,.\,(xz))y))w))) &= \lambda zw\,.\,(\lambda y\,.\,(\lambda x\,.\,xz)y)w
\end{aligned}
$$

β-REDUCTION

- $[x \mapsto N]\,M$ – substitution of N for x in M
- $(\lambda x\,.\,M)N$ – β-redex

$$(\lambda x\,.\,M)N \to_\beta [x \mapsto N]M$$

FREE AND BOUND VARIABLES

- Abstraction $\lambda x\,.\,M[x]$ binds variable x in term M
- The scope of a bound variable is the body of the abstraction
- The same variable name can be bound multiple times

$$(\lambda y\,.\,(\lambda w\,.\,w\,z)\;y)\;x$$

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

FREE AND BOUND VARIABLES

- Abstraction $\lambda x \,.\, M[x]$ binds variable x in term M
- The scope of a bound variable is the body of the abstraction
- The same variable name can be bound multiple times

bound

$$(\lambda y \,.\, (\lambda w \,.\, w\, z)\, y)\, x$$

free

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

different

FREE AND BOUND VARIABLES

Free variables

$$
\begin{aligned}
FV(x) &= \{x\} \\
FV(M\,N) &= FV(M) \cup FV(N) \\
FV(\lambda x\,.\,M) &= FV(M) \backslash \{x\}
\end{aligned}
$$

Bound variables

$$
\begin{aligned}
BV(x) &= \varnothing \\
BV(M\,N) &= BV(M) \cup BV(N) \\
BV(\lambda x\,.\,M) &= BV(M) \cup \{x\}
\end{aligned}
$$

COMBINATORS

- M is a combinator (or closed lambda-term) if it has no free variables: $FV(M) = \varnothing$
- Some common combinators have names

$$
\begin{aligned}
\boldsymbol{I} &= \lambda x\,.\,x \\
\boldsymbol{\omega} &= \lambda x\,.\,x\,x \\
\boldsymbol{\Omega} &= \omega\,\omega = (\lambda x\,.\,x\,x)\,(\lambda x\,.\,x\,x) \\
\boldsymbol{K} &= \lambda xy\,.\,x \\
\boldsymbol{K_*} &= \lambda xy\,.\,y \\
\boldsymbol{C} &= \lambda fxy\,.\,f\,y\,x \\
\boldsymbol{B} &= \lambda fgx\,.\,f\,(g\,x) \\
\boldsymbol{S} &= \lambda fgx\,.\,f\,x\,(g\,x)
\end{aligned}
$$

VARIABLE RENAMING

- Names of bound variables do not matter
- We can rename variables without affecting the computation result
- Terms that are different only in the names of their bound variables are called *α-equivalent*

$$I = \lambda x\,.\,x = \lambda y\,.\,y = \lambda z\,.\,z$$
$$B = \lambda fgx\,.\,f\,(g\,x) = \lambda uvz\,.\,u\,(v\,z)$$

$$(\lambda x\,.\,x)\,N \to_\beta N$$
$$(\lambda y\,.\,y)\,N \to_\beta N$$
$$(\lambda z\,.\,z)\,N \to_\beta N$$

EXERCISE

- Do the substitutions:

$$[x \mapsto w\,(\lambda x\,.\,w\,x)]\;(x\,y\,(\lambda xy\,.\,x\,z\,(w\,x)\,y))$$

$$[y \mapsto w\,(\lambda x\,.\,w\,x)]\;(x\,y\,(\lambda xy\,.\,x\,z\,(w\,x)\,y))$$

$$[z \mapsto w\,(\lambda x\,.\,w\,x)]\;(x\,y\,(\lambda xy\,.\,x\,z\,(w\,x)\,y))$$

- $$[x \mapsto w\,(\lambda x\,.\,y\,x)]\;(x\,y\,(\lambda xy\,.\,x\,z\,(w\,x)\,y))$$

- Functional Model of Computation
- Lambda Calculus
- Term Equivalence
- Expressing Common Data Types

VARIABLE SUBSTITUTION

- Substitution – syntactic transformation that replaces a variable with some expression
- We can only (safely) substitute for free vars
  - $[x \mapsto \lambda z\,.\,z]\,(x(\lambda x\,.\,x\;y)x) = (\lambda z\,.\,z)(\lambda x\,.\,x\;y)(\lambda z\,.\,z)$
- The issue of variable capturing:
  - $[x \mapsto y]\,(\lambda y\,.\,x\;y) = \lambda y\,.\,y\;y$
- We can always rename the bound variables:
  - $[x \mapsto y]\,(\lambda w\,.\,x\;w) = \lambda w\,.\,y\;w$

CAPTURE AVOIDING SUBSTITUTION

$$
\begin{aligned}
[x \mapsto N]\, x &= N \\
[x \mapsto N]\, y &= y \\
[x \mapsto N]\, (P\, Q) &= ([x \mapsto N]\, P)\, ([x \mapsto N]\, Q) \\
[x \mapsto N]\, (\lambda x \,.\, P) &= \lambda x \,.\, P \\
[x \mapsto N]\, (\lambda y \,.\, P) &= \lambda y \,.\, [x \mapsto N]\, P, \mathit{if}\ y \notin FV(N) \\
[x \mapsto N]\, (\lambda y \,.\, P) &= \lambda z \,.\, [x \mapsto N]\, ([y \mapsto z]\, P, \mathit{if}\ y \in FV(N), z \notin FV(N) \cup FV(P)
\end{aligned}
$$

PROPERTIES OF SUBSTITUTIONS

- Substitution is not commutative
- Substitution lemma:
  - Let $M, N, L \in \Lambda$
  - Assume $x \not\equiv y, x \notin FV(L)$
  - Then $[y \mapsto L]([x \mapsto N]M) \equiv [x \mapsto [y \mapsto L]N]([y \mapsto L]M)$

β-EQUIVALENCE

- Axiom (rule β): $(\lambda x\,.\,M)\,N =_\beta [x \mapsto N]\,M$
- Properties:

$$M =_\beta M$$
$$M =_\beta N \Rightarrow N =_\beta M$$
$$M =_\beta N, N =_\beta L \Rightarrow M =_\beta L$$

$$M =_\beta M' \Rightarrow M\,Z =_\beta M'\,Z$$
$$M =_\beta M' \Rightarrow Z\,M =_\beta Z\,M'$$
$$M =_\beta M', \lambda x\,.\,M =_\beta \lambda x\,.\,M'$$

α-EQUIVALENCE

- Axiom (rule α):
  $(\lambda x\,.\,M) =_\alpha \lambda y\,.\,[x \mapsto y]\,M, \textit{if}\ y \notin FV(M)$

$$
\begin{aligned}
\boldsymbol{\omega} &= \lambda x\,.\,xx \\
\mathbf{1} &= \lambda fx\,.\,fx \\
\boldsymbol{\omega}\mathbf{1} &= (\lambda x\,.\,xx)\,(\lambda fx\,.\,fx) \\
&=_\beta (\lambda fx\,.\,fx)\,(\lambda fx\,.\,fx) \\
&=_\beta (\lambda x\,.\,(\lambda fx\,.\,fx)\,x) \\
&=_\alpha (\lambda x\,.\,(\lambda fx'\,.\,fx')\,x) \\
&=_\beta (\lambda xx'\,.\,xx') \\
&=_\alpha (\lambda fx'\,.\,fx') \\
&=_\alpha (\lambda fx\,.\,fx) \\
&= \mathbf{1}
\end{aligned}
$$

η-EQUIVALENCE

- Axiom (rule η):
  $\lambda x \,.\, M\, x =_\eta M$
- For any term N : $(\lambda x \,.\, M\, x)\, N =_\beta M\, N$
- Pointfree Haskell is all about η rule

```haskell
((.)$(.))

((.).(.))

f >>= a .  b . c =<< g
```

FUNCTIONAL EXTENTIONALITY

- When are two functions equivalent?

FUNCTIONAL EXTENTIONALITY

- When are two functions equivalent?
- Two functions are equal if $\forall N : F\,N = G\,N$

$$y \notin FV(F) \cup FV(G) :$$

$$F\,y = G\,y$$

$$\lambda y\,.\,F\,y = \lambda y\,.\,G\,y$$

$$F = G$$

EXERCISE

- Are these terms equivalent?

$$\lambda x\,.\,x$$
$$\lambda y\,.\,y$$
$$\lambda xy\,.\,xy$$

- Show that

$$\boldsymbol{SKK} =_\beta \boldsymbol{I}$$
$$\boldsymbol{B} =_\beta \boldsymbol{S(KS)K}$$

- Functional Model of Computation
- Lambda Calculus
- Term Equivalence
- Expressing Common Data Types

BOOLEANS

$$
\begin{aligned}
true &\equiv \lambda t\, f\,.\, t \\
false &\equiv \lambda t\, f\,.\, f \\
if &\equiv \lambda b\, x\, y\,.\, b\, x\, y \\
not &\equiv \lambda b\,.\, b\ false\ true \\
and &\equiv \lambda x\, y\,.\, x\, y\ false \\
or &\equiv ??? \\
and\ true\ false &= ??? \\
or\ true\ false &= ???
\end{aligned}
$$

CHURCH NUMERALS

$$
\begin{aligned}
0 &\equiv \lambda f x \,.\, x \\
1 &\equiv \lambda f x \,.\, f x \\
2 &\equiv \lambda f x \,.\, f (f x) \\
3 &\equiv \lambda f x \,.\, f (f (f x)) \\
succ &\equiv \lambda n f x \,.\, f (n f x) \\
plus &\equiv \lambda m n f x \,.\, m f (n f x) \\
plus\ 2\ 3 &= ??? \\
mult &= ???
\end{aligned}
$$

PAIRS / LISTS

$$
\begin{aligned}
Pair &\equiv \lambda x\, y f\,.\, f\, x\, y \\
First &\equiv \lambda p\,.\, p\ \ true \\
Second &\equiv \lambda p\,.\, p\ \ false \\
Nil &\equiv \lambda x\,.\, true \\
Null &\equiv \lambda p\,.\, p\, (\lambda x\, y\,.\, false) \\
Repeat &= ???
\end{aligned}
$$