Function Definitions

Functions are not only about organizing code or making it easier to reuse code, but provide a powerful means of **abstraction**!

Iffy Lang

$$b ::= \mathsf{T} \mid \mathsf{F} \mid \text{if } b_1 \text{ then } b_2 \text{ else } b_3$$

Iffy Lang

$$p ::= \text{func}\, name(x_1, \ldots, x_i)\{body\}$$

$$body ::= \text{return}\, b$$

$$b ::= x \mid name(b_1, \ldots, b_i) \mid \text{T} \mid \text{F} \mid \text{if}\, b_1\, \text{then}\, b_2\, \text{else}\, b_3 \mid (b)$$

Example Program: Negation

```
func not(x) {
  ?
}
```

Example Program: Negation

```
func not(x) {
  return (if x
            then F
            else T)
}
```

Example Program: And

```
func and(x1, x2) {
  ?
}
```

Example Program: And

```
func and(x1, x2) {
  return (if x1
           then x2
           else F)
}
```

Example Program: And

```
func or(x1, x2) {
  ?
}
```

Example Program: And

```
func or(x1, x2) {
  return (if x1
          then T
          else x2)
}
```

Example Program: Implication

```
func implies(x1, x2) {
  ?
}
```

Example Program: Implication

```
func implies(x1, x2) {
  return or(not(x1),x2)
}
```

Example Program: Necessary and Sufficient

```
func iff(x1, x2) {
  ?
}
```

Example Program: Necessary and Sufficient

```
func iff(x1, x2) {
  return and(implies(x1,x2),implies(x2,x1))
}
```

Example Program: Exclusive-Or

```
func xor(x1, x2) {
  ?
}
```

Example Program: Exclusive-Or

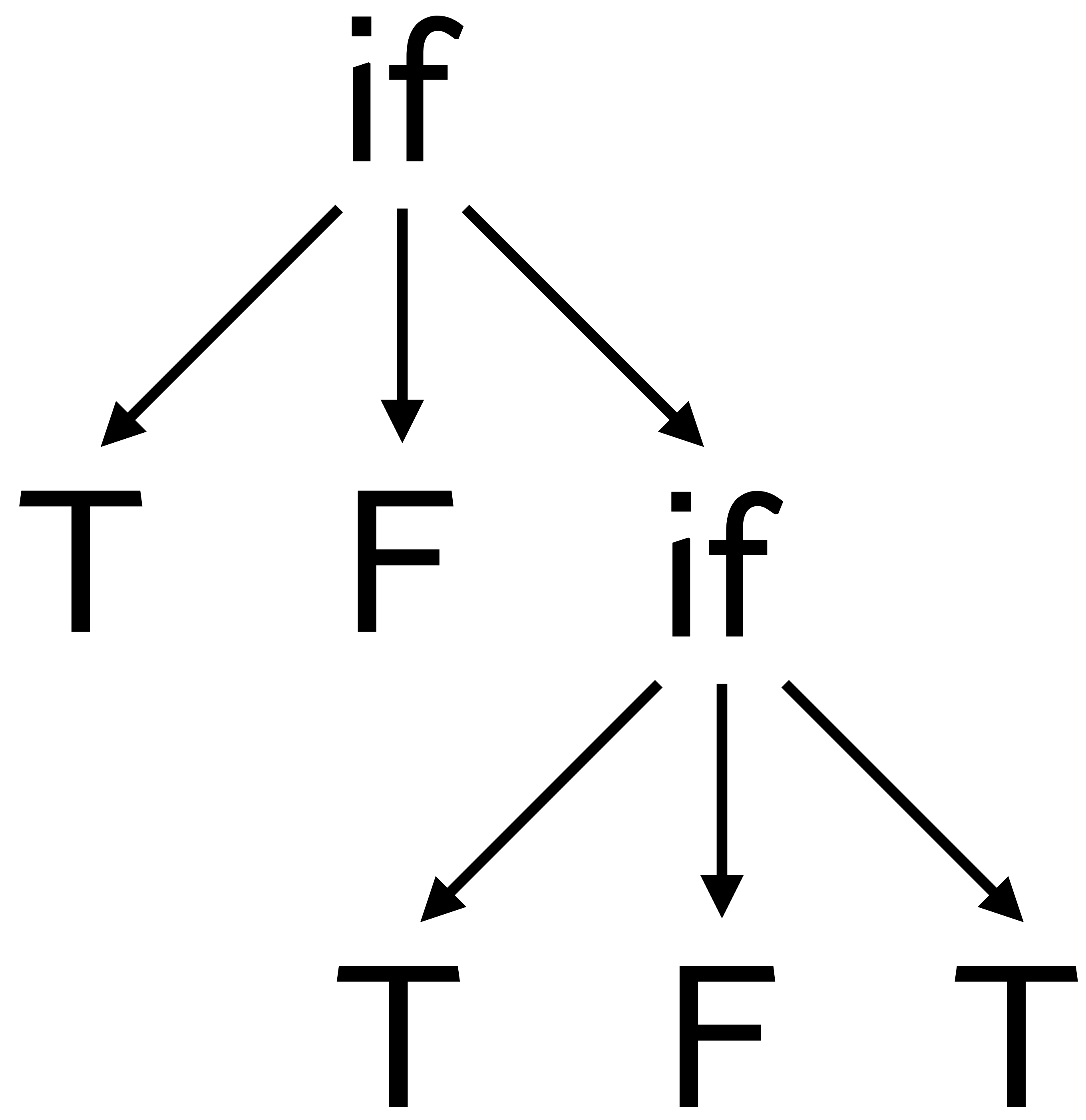
```
func xor(x1, x2) {
  return or(and(x1,not(x2)),and(not(x1),x2))
}
```

Syntax Trees

Syntax trees are a tree representation of a syntactical expression.

Example Syntax Tree

if F then T else (if F then T else T)

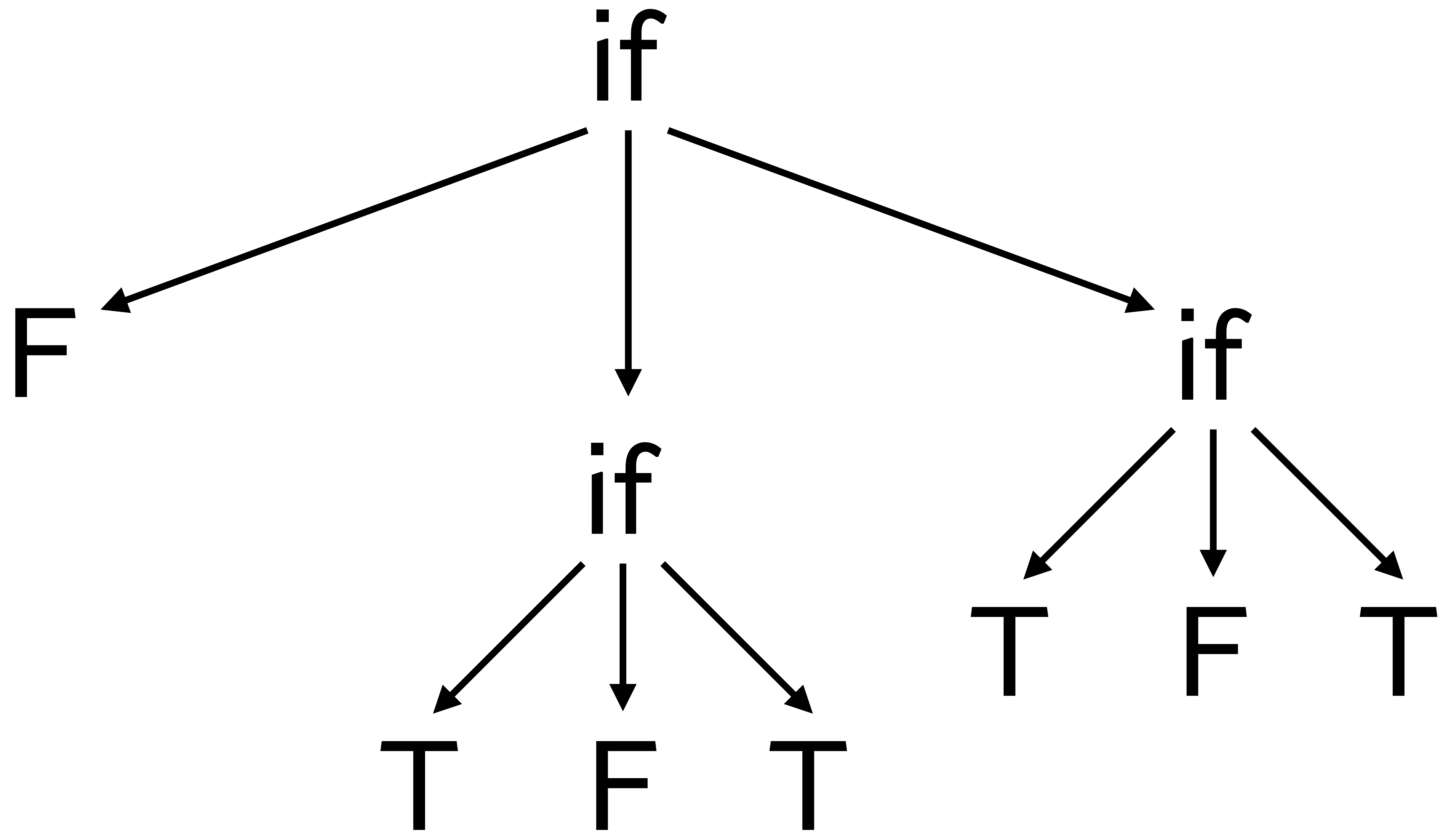


Example Syntax Tree

if (if F then T else T) then F else (if F then T else T)

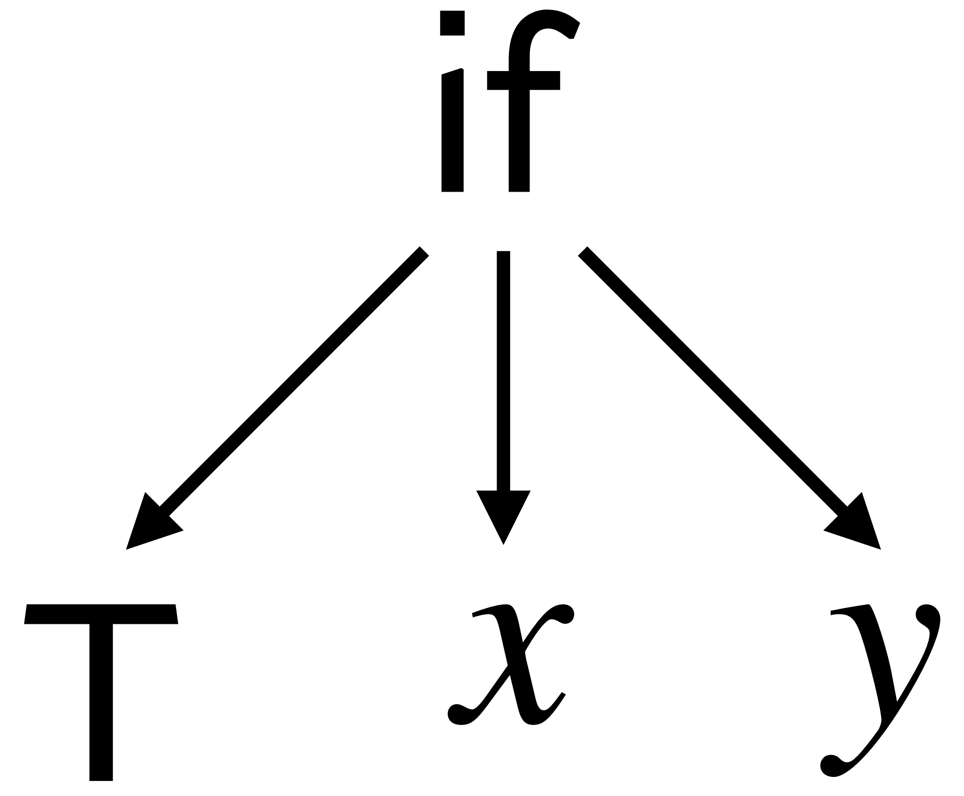
Example Syntax Tree

if (if F then T else T) then F else (if F then T else T)



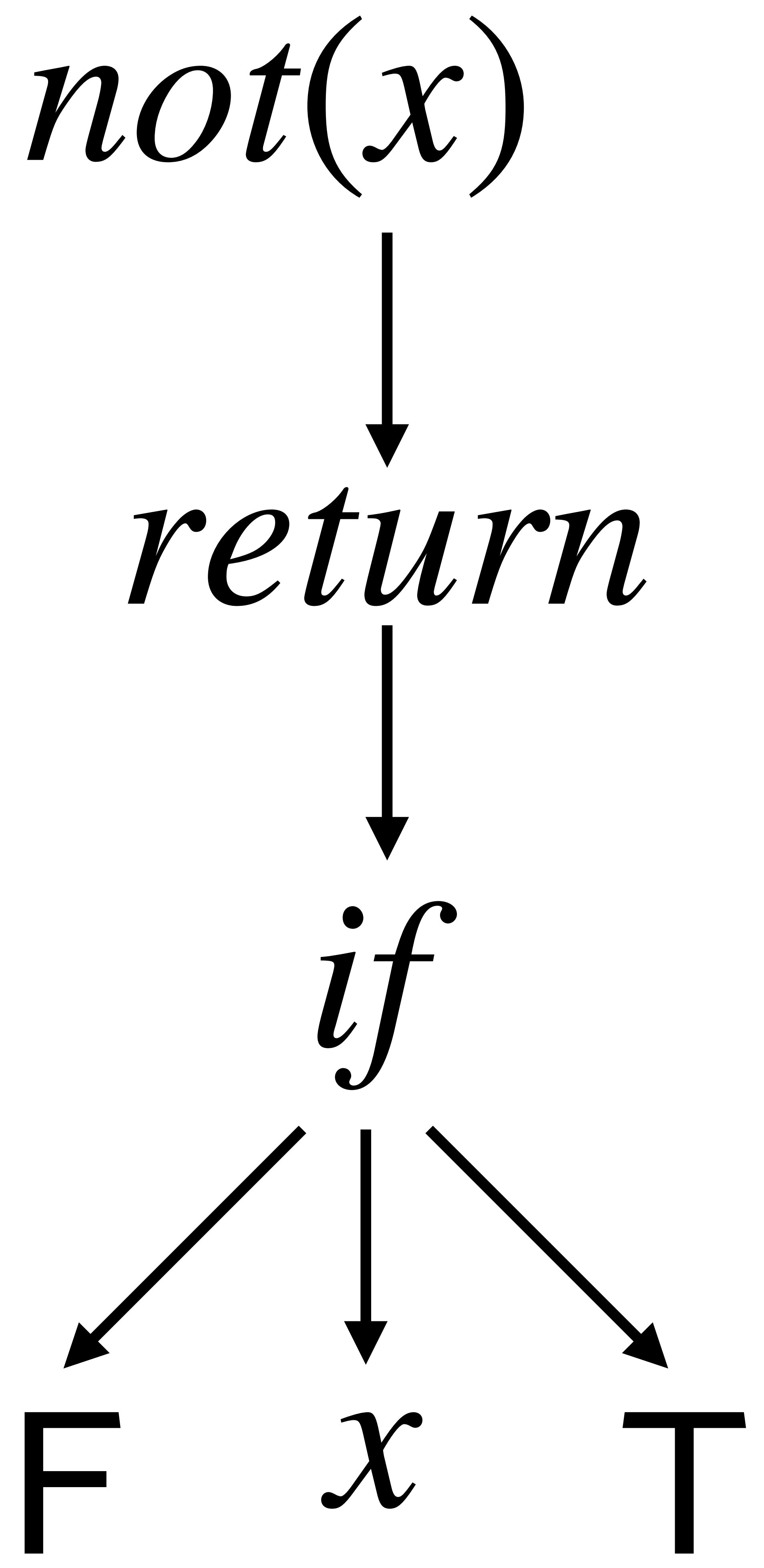
Example Syntax Tree

if x then T else y



Syntax Trees for Function Definitions

```
func not(x) {
  return (if x
             then F
             else T)
}
```

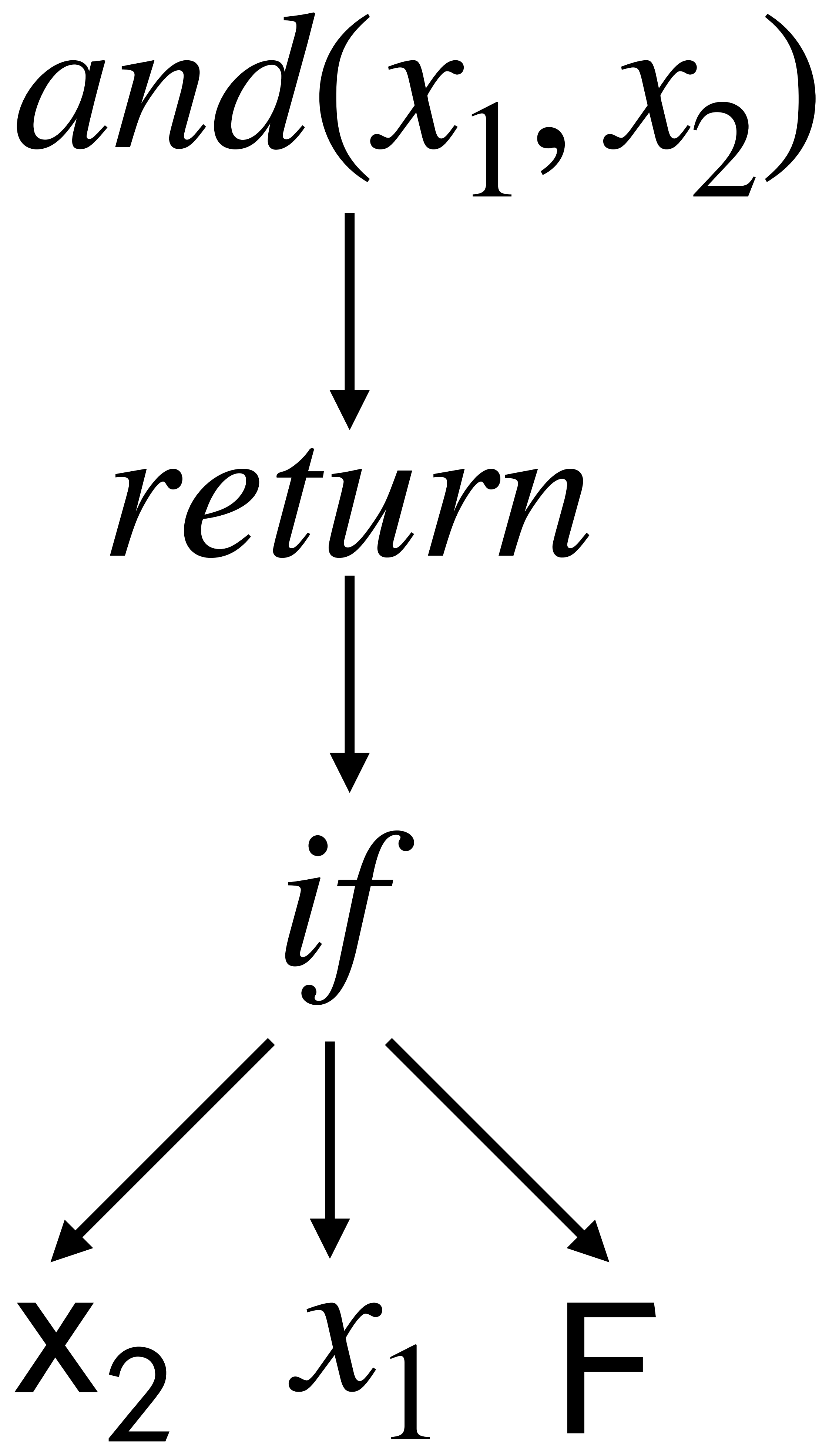


Syntax Trees for Function Definitions

```
func and(x1, x2) {
  return (if x1
            then x2
            else F)
}
```

Syntax Trees for Function Definitions

```
func and(x1, x2) {
  return (if x1
             then x2
             else F)
}
```



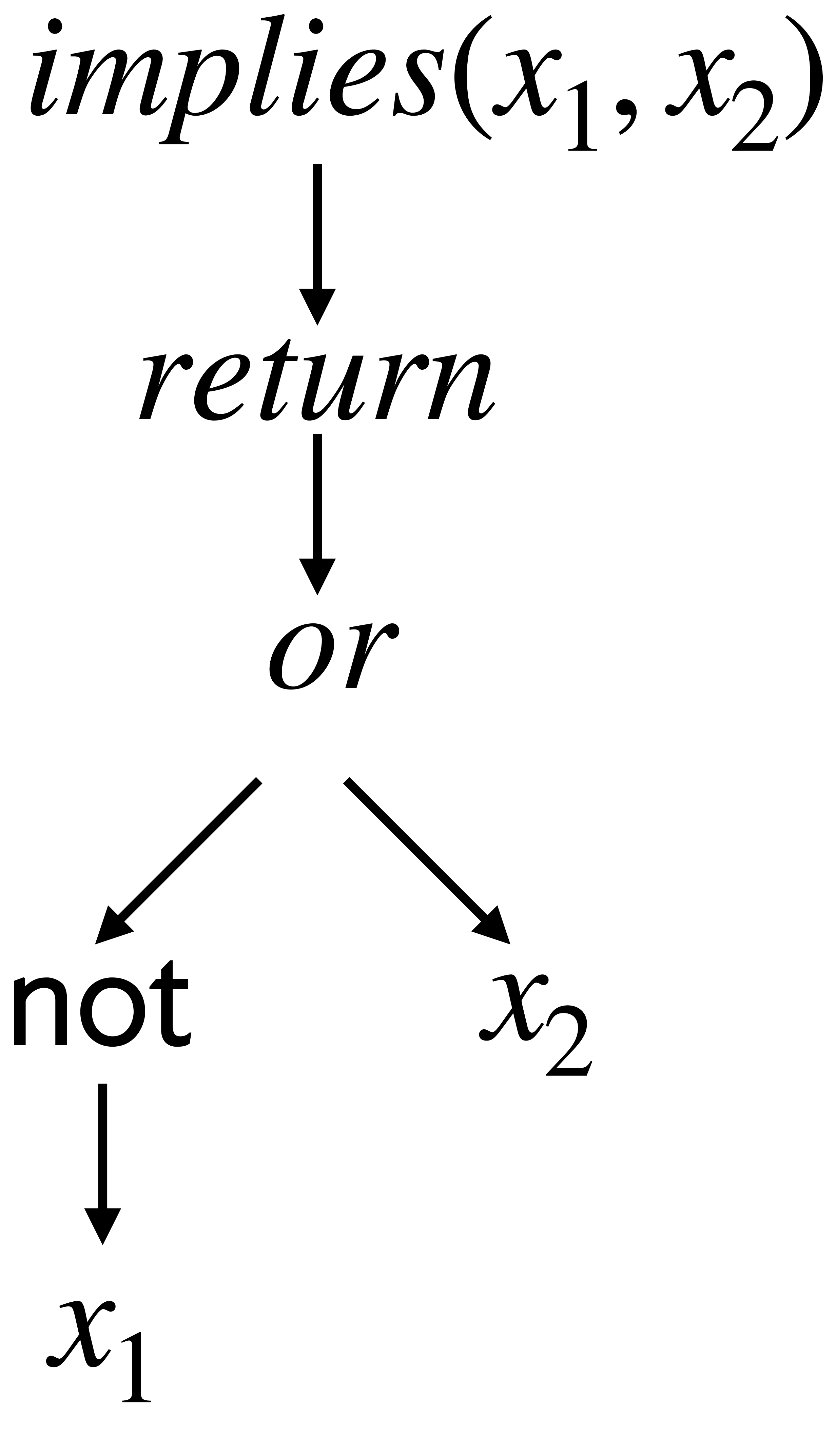
Syntax Trees for Function Application

$$name(b_1, \ldots, b_i)$$

$$\begin{tikzcd}
 & name \arrow[dl] \arrow[d] \arrow[dr] & \\
b_1 & \cdots & b_i
\end{tikzcd}$$

Syntax Trees for Function Application

```
func implies(x1, x2) {
  return or(not(x1),x2)
}
```



Free and Bound Variables

A **binder** is a piece of syntax that delimits the scope of a variable.

Free and Bound Variables

In IffyLang, the only binder, currently, is the function signature.

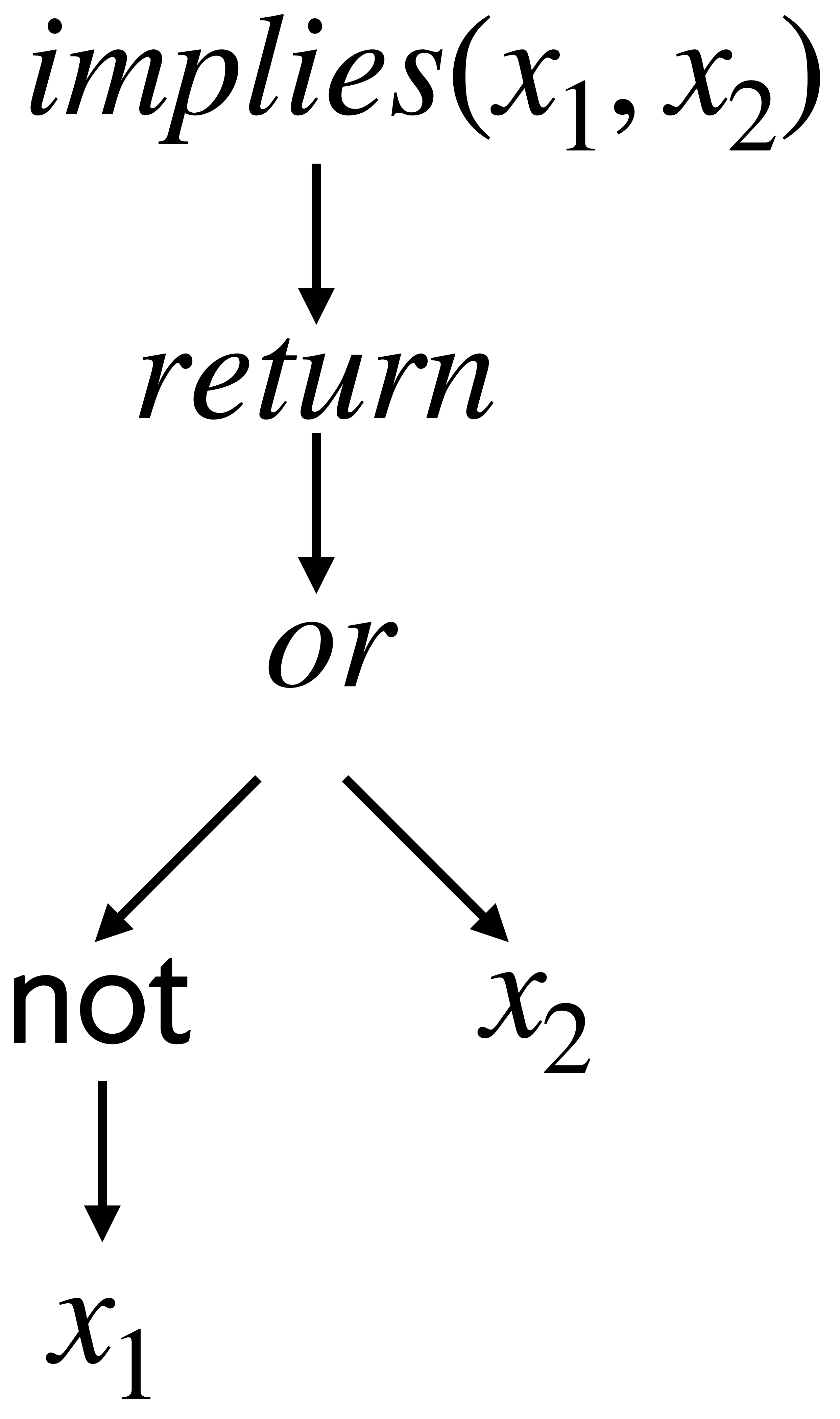
Here:

$$p ::= \text{func}\, name(x_1, \ldots, x_i)\{body\}$$

The binder is "name$(x_1, \ldots, x_i)$" and is said to "bind" the variables x_1 through x_i to the function name.

Free and Bound Variables

```
func implies(x1, x2) {
  return or(not(x1),x2)
}
```



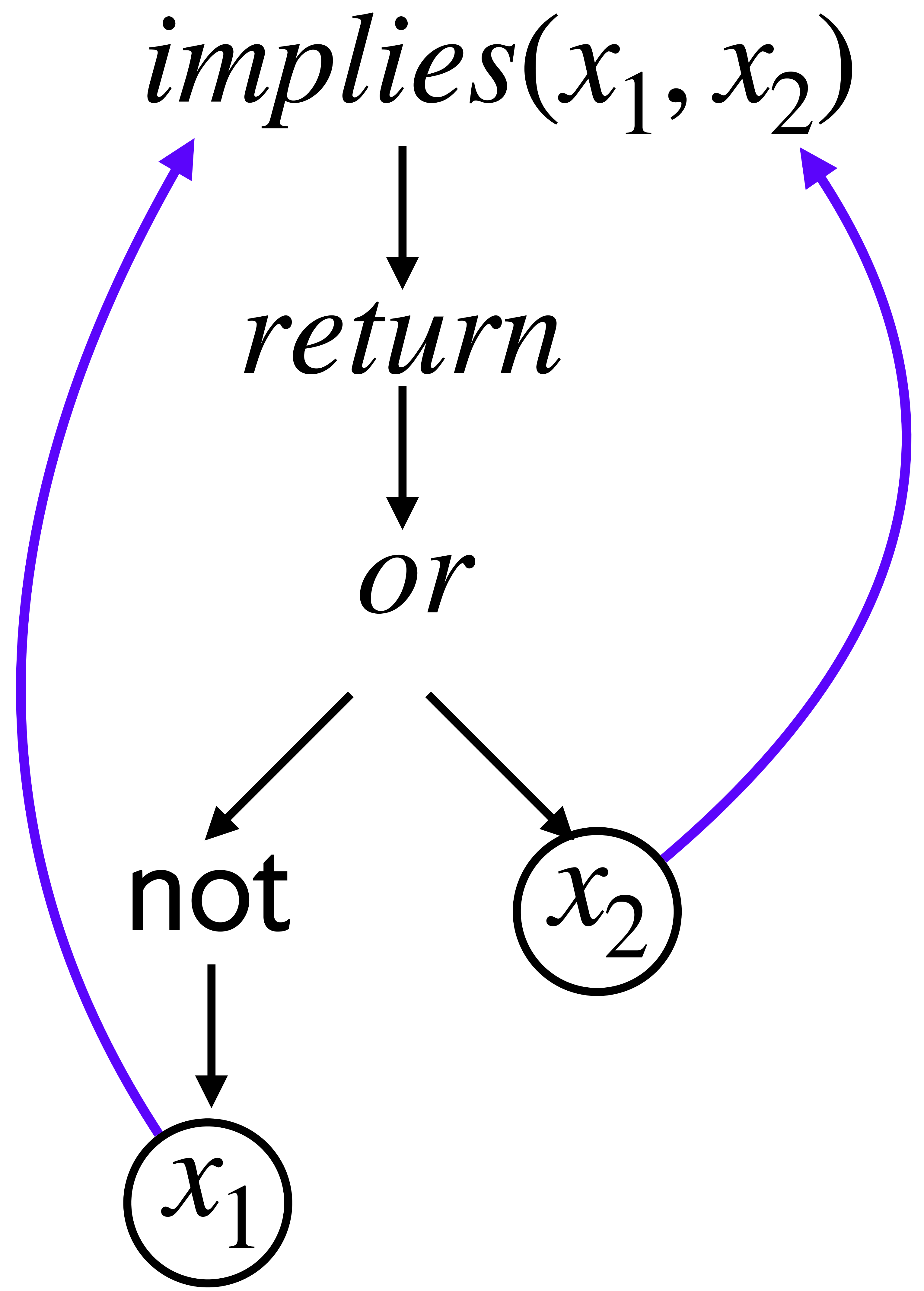
Free and Bound Variables

A **bound** variable is one associated with a **binder.**

Free and Bound Variables

```
func implies(x1, x2) {
  return or(not(x1),x2)
}
```

The variables x_1 and x_2 are said to be **bound variables**, because they have a binder above them in their syntax tree.



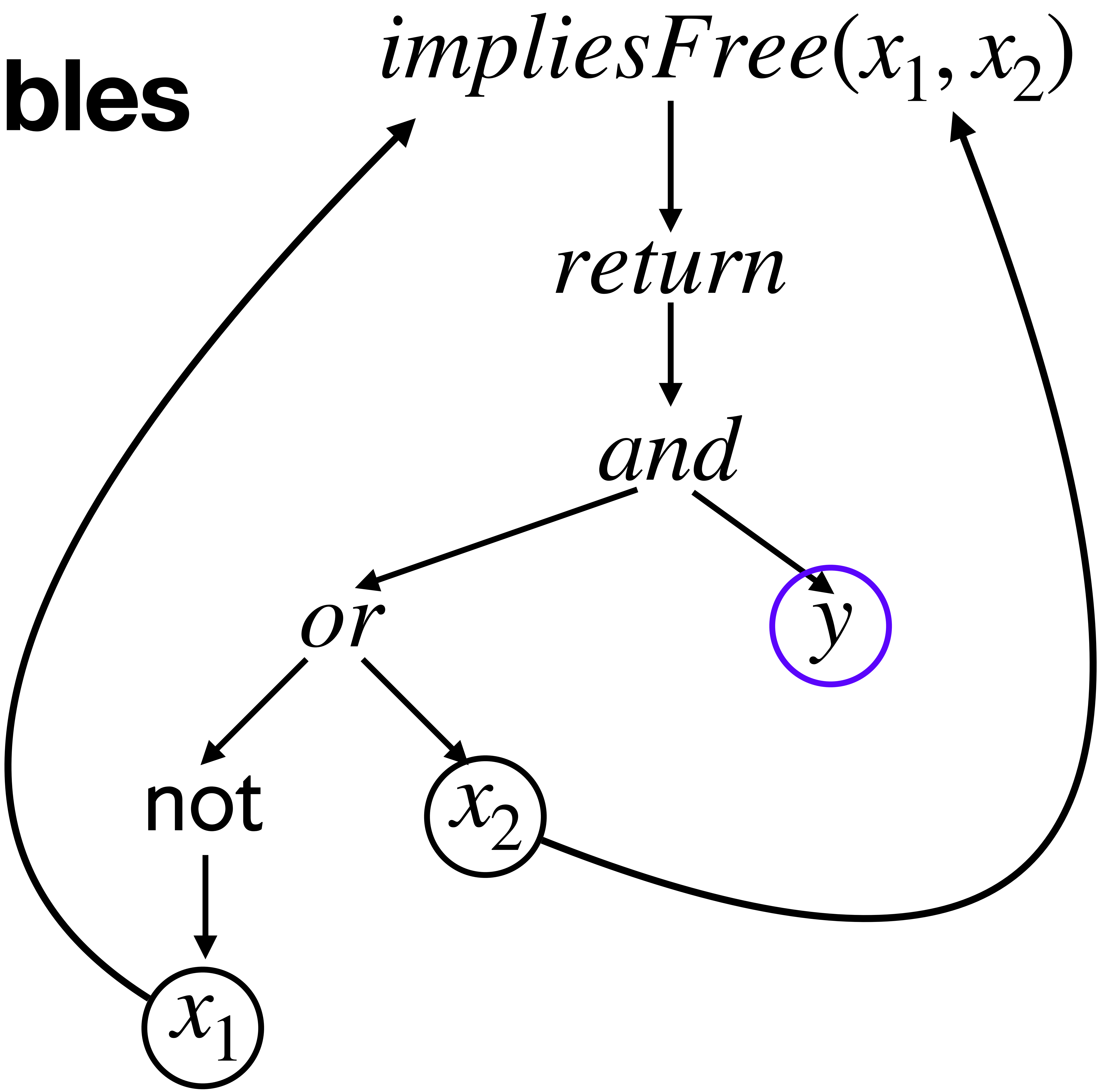
Free and Bound Variables

Any other variables are called **free**.

Free and Bound Variables

```
func impliesFree(x1, x2) {
  return and(or(not(x1),x2),y)
}
```

A **free** variable has no binder above it in the syntax tree.



Closed Programs

A program is called **closed** if and only if it has no free variables.

Closed Programs

How many programs have you ever written with a free variable?

Closed Programs

How many programs have you ever written with a free variable?

None!

Valid Programs

Valid programs are closed programs!

Valid Programs

Valid programs are closed programs!

Discuss: What are the valid programs in IffyLang?

Evaluation

```
func implies(x1, x2) {
 return or(not(x1),x2)
}

func comp() {
  return implies(T,F)
}
```

How do we evaluate the `comp` function?

Context of Function Definition

$\text{implies}(x_1, x_2)\{\text{return}(\text{or}(\text{not}(x_1), x_2))\}, \text{comp}()\{\text{return}(\text{implies}(\mathsf{T}, \mathsf{F})\}$

$\Uparrow$

```
func implies(x1, x2) {
 return or(not(x1),x2)
}

func comp() {
  return implies(T,F)
}
```

Context of function definitions:

$\Delta ::= \Delta, \text{name}(x_1, \ldots, x_i)\{\text{body}\} \mid \text{empty}$

Evaluation

Context of function definitions:

$$\Delta ::= \Delta, \text{name}(x_1, \ldots, x_i)\{\text{body}\} \mid \text{empty}$$

Evaluation:

$$\Delta \vdash b_1 \rightsquigarrow b_2$$

where b_1 and b_2 are closed programs.

Evaluation

Context of function definitions:

$$\Delta ::= \Delta, \texttt{name}(x_1, \ldots, x_i)\{\texttt{body}\} \mid \text{empty}$$

Evaluation:

$$\Delta \vdash b_1 \rightsquigarrow b_2$$

where b_1 and b_2 are closed programs.

$$\frac{}{\Delta, \texttt{name}()\{\texttt{body}\} \vdash \texttt{name}() \rightsquigarrow \texttt{body}}\ \beta_0$$

$$\frac{}{\Delta \vdash \texttt{return}\, b \rightsquigarrow b}\ \text{return}$$

Context of Function Definition

```
func implies(x1, x2) {
  return or(not(x1),x2)
}

func comp() {
  return implies(T,F)
}
```

$$\text{implies}(x_1, x_2)\{\mathtt{return}(\text{or}(\mathtt{not}(x_1), x_2))\}, \text{comp}()\{\mathtt{return}(\mathtt{implies}(\mathsf{T}, \mathsf{F})\} \vdash \mathtt{comp}()$$

$$\rightsquigarrow \mathtt{return}(\mathtt{implies}(\mathsf{T}, \mathsf{F}))$$

$$\rightsquigarrow \mathtt{implies}(\mathsf{T}, \mathsf{F})$$

$$\rightsquigarrow ?$$

Substitution

Substitution replaces a free variable in a program with a closed program.

Denoted:

$$[x \mapsto t_1]t_2$$

where t_1 is closed.

Substitution

$$[x \mapsto t_1]x = t_1$$

Suppose t_1 is closed.

$$[x \mapsto t_1]y = y$$

where $x \neq y$

$$[x \mapsto t_1]T = T$$

$$[x \mapsto t_1]F = F$$

$$[x \mapsto t_1](\text{if } b_1 \text{ then } b_2 \text{ else } b_3) = \text{if } [x \mapsto t_1]b_1 \text{ then } [x \mapsto t_1]b_2 \text{ else } [x \mapsto t_1]b_3$$

$$[x \mapsto t_1](\text{func}\, name(x_1, \ldots, x_i)\{body\}) = \text{func}\, name(x_1, \ldots, x_i)\{[x \mapsto t_1]body\}$$

$$[x \mapsto t_1]name(b_1, \ldots, b_i) = name([x \mapsto t_1]b_1, \ldots, [x \mapsto t_1]b_i)$$

Substitution

$$[x \mapsto t_1](\text{if } b_1 \text{ then } b_2 \text{ else } b_3) = \text{if } [x \mapsto t_1]b_1 \text{ then } [x \mapsto t_1]b_2 \text{ else } [x \mapsto t_1]b_3$$

$$[x \mapsto t_1](\text{func } name(x_1, \ldots, x_i)\{body\}) = \text{func } name(x_1, \ldots, x_i)\{[x \mapsto t_1]body\}$$

$$[x \mapsto t_1]name(b_1, \ldots, b_i) = name([x \mapsto t_1]b_1, \ldots, [x \mapsto t_1]b_i)$$

Substitution

Suppose t_1 is closed.

$$[x \mapsto b_1]x = b_1$$

$$[x \mapsto b_1]y = y$$

where $x \neq y$

$$[x \mapsto b_1]T = T$$

Explain using trees!

$$[x \mapsto b_1]F = F$$

$$[x \mapsto b_1](\text{if } b_2 \text{ then } b_3 \text{ else } b_4) = \text{if } [x \mapsto b_1]b_2 \text{ then } [x \mapsto b_1]b_3 \text{ else } [x \mapsto b_1]b_4$$

$$[x \mapsto b_1]name(b_1', \ldots, b_i') = name([x \mapsto b_1]b_1', \ldots, [x \mapsto b_1]b_i')$$

$$[x \mapsto b_1]return(b_2) = return([x \mapsto b_1]b_2)$$

Evaluating Functions

$$\frac{}{\Delta, \texttt{name}(x_1, \ldots, x_i)\{\texttt{body}\} \vdash \texttt{name}(b_1, \ldots, b_i) \rightsquigarrow [x_1, \ldots, x_i \mapsto b_1, \ldots, b_i]\texttt{body}}\ \beta$$