# Simple Static Typechecking

Geoffrey Matthews

Professor Emeritus

March 31, 2022

# A Little Bit About Me

- Born in California, grew up in the beachtown of Santa Barbara.

# A Little Bit About Me

- Born in California, grew up in the beachtown of Santa Barbara.
- Ph.D. in the History and Philosophy of Science from Indiana University

# A Little Bit About Me

- Born in California, grew up in the beachtown of Santa Barbara.
- Ph.D. in the History and Philosophy of Science from Indiana University
- Worked briefly in the defense industry for Hughes Aircraft

# A Little Bit About Me

- Born in California, grew up in the beachtown of Santa Barbara.
- Ph.D. in the History and Philosophy of Science from Indiana University
- Worked briefly in the defense industry for Hughes Aircraft
- Professor of Computer Science at Western Washington University for 35 years

# A Little Bit About Me

- Born in California, grew up in the beachtown of Santa Barbara.
- Ph.D. in the History and Philosophy of Science from Indiana University
- Worked briefly in the defense industry for Hughes Aircraft
- Professor of Computer Science at Western Washington University for 35 years
- Research in Data Exploration, Visualization, Procedural Generation

# A Little Bit About Me

- Born in California, grew up in the beachtown of Santa Barbara.
- Ph.D. in the History and Philosophy of Science from Indiana University
- Worked briefly in the defense industry for Hughes Aircraft
- Professor of Computer Science at Western Washington University for 35 years
- Research in Data Exploration, Visualization, Procedural Generation
- Retired 2020 and moved to Virginia

# A Little Bit About Me

- Born in California, grew up in the beachtown of Santa Barbara.
- Ph.D. in the History and Philosophy of Science from Indiana University
- Worked briefly in the defense industry for Hughes Aircraft
- Professor of Computer Science at Western Washington University for 35 years
- Research in Data Exploration, Visualization, Procedural Generation
- Retired 2020 and moved to Virginia
- Hablo Español

# A Little Bit About Me

- Born in California, grew up in the beachtown of Santa Barbara.
- Ph.D. in the History and Philosophy of Science from Indiana University
- Worked briefly in the defense industry for Hughes Aircraft
- Professor of Computer Science at Western Washington University for 35 years
- Research in Data Exploration, Visualization, Procedural Generation
- Retired 2020 and moved to Virginia
- Hablo Español
- Eight cats

# Type Checking

The idea is to *analyze* a program *statically* to find errors that might occur when you run the program. For example:

- Ensure that $exp_1$ and $exp_2$ are both type Int in every case of

$$exp_1 + exp_2$$

- Ensure that $exp_1$ and $exp_2$ are both type Bool in every case of

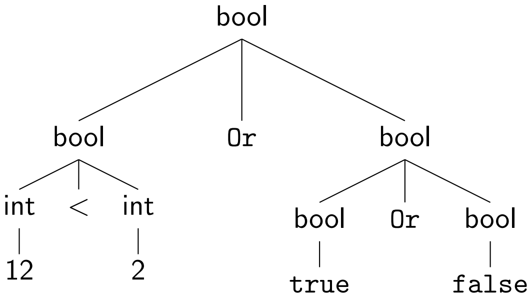
$$exp_1 \text{ Or } exp_2$$

- Ensure that $exp_1$ is type Bool in every case of

$$\text{If } exp_1 \text{ Then } exp_2 \text{ Else } exp_3$$

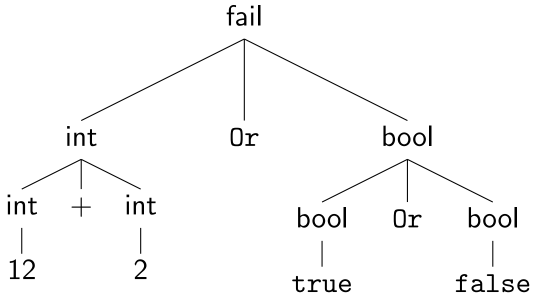
# Recursive type checking

`(12 < 2) Or (true Or false)`



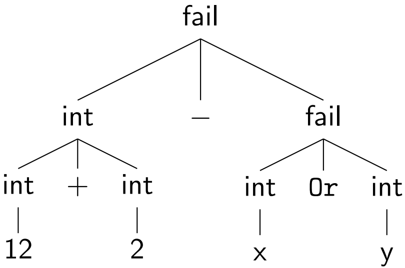
# Recursive type checking

`(12 + 2) Or (true Or false)`



# Recursive type checking

`(12 + 2) - (x Or y)`



# The language of Homework 5 has nothing to check!

- The **parser** ensures that only Bool expressions occur in `IF` and `WHILE`
- Only Int expressions can occur in assignments to variables
- Only numbers, variables and arithmetic operators can occur on the RHS of assignments
- No program with a type error can even be parsed!
- My original idea was to add boolean variables,
  but I got a much better idea.

# Quick review of `Lambda` expressions

My syntax:

$$\underbrace{\texttt{[ }\underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}\texttt{ ]}}_{\text{application}}$$

# Quick review of Lambda expressions

My syntax:

$$\underbrace{\texttt{[ }\underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}\texttt{ ]}}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` $\Rightarrow$

# Quick review of Lambda expressions

My syntax:

$$\underbrace{\texttt{[}\ \underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ \texttt{]}}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` $\Rightarrow$ 16

# Quick review of Lambda expressions

My syntax:

$$\underbrace{\texttt{[}\ \underbrace{\texttt{\\x \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ \texttt{]}}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` $\Rightarrow$ 16
- `[ \f { [f 3] } \x { 2 * x } ]` $\Rightarrow$

# Quick review of Lambda expressions

My syntax:

$$\underbrace{[\ \underbrace{\backslash x\ \{\ x\ +\ 1\}}_{\text{function}}\underbrace{(3\ *\ 5)}_{\text{argument}}\ ]}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` $\Rightarrow$ 16
- `[ \f { [f 3] } \x { 2 * x } ]` $\Rightarrow$ 6

# Quick review of Lambda expressions

My syntax:

$$\underbrace{[\ \underbrace{\backslash\mathtt{x}\ \{\ \mathtt{x} + 1\}}_{\text{function}}\ \underbrace{(3 * 5)}_{\text{argument}}\ ]}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` $\Rightarrow$ 16
- `[ \f { [f 3] } \x { 2 * x } ]` $\Rightarrow$ 6
- `[ \x { \y { x + y} } 3 ]` $\Rightarrow$

# Quick review of Lambda expressions

My syntax:

$$\underbrace{\texttt{[ } \underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}} \underbrace{\texttt{(3 * 5)}}_{\text{argument}} \texttt{ ]}}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` ⇒ 16
- `[ \f { [f 3] } \x { 2 * x } ]` ⇒ 6
- `[ \x { \y { x + y} } 3 ]` ⇒ `\y { 3 + y }`

# Quick review of Lambda expressions

My syntax:

$$\underbrace{\texttt{[ } \underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}} \underbrace{\texttt{(3 * 5)}}_{\text{argument}} \texttt{ ]}}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` ⇒ 16
- `[ \f { [f 3] } \x { 2 * x } ]` ⇒ 6
- `[ \x { \y { x + y} } 3 ]` ⇒ `\y { 3 + y }`
- `\x { x + 1 }` ::

# Quick review of Lambda expressions

My syntax:

$$\underbrace{[\ \underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}}\ \underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ ]}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` $\Rightarrow$ 16
- `[ \f { [f 3] } \x { 2 * x } ]` $\Rightarrow$ 6
- `[ \x { \y { x + y} } 3 ]` $\Rightarrow$ `\y { 3 + y }`
- `\x { x + 1 }` :: `(int -> int)`

# Quick review of Lambda expressions

My syntax:

$$\underbrace{\texttt{[}\ \underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}}\ \underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ \texttt{]}}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` $\Rightarrow$ 16
- `[ \f { [f 3] } \x { 2 * x } ]` $\Rightarrow$ 6
- `[ \x { \y { x + y} } 3 ]` $\Rightarrow$ `\y { 3 + y }`
- `\x { x + 1 }` :: `(int -> int)`
- `\x { \y { x + y } }` ::

# Quick review of Lambda expressions

My syntax:

$$\underbrace{\texttt{[}\ \underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ \texttt{]}}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` $\Rightarrow$ 16
- `[ \f { [f 3] } \x { 2 * x } ]` $\Rightarrow$ 6
- `[ \x { \y { x + y} } 3 ]` $\Rightarrow$ `\y { 3 + y }`
- `\x { x + 1 }` :: `(int -> int)`
- `\x { \y { x + y } }` :: `(int -> (int -> int))`

# Quick review of Lambda expressions

My syntax:

$$\underbrace{\texttt{[}\ \underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}}\ \underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ \texttt{]}}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` $\Rightarrow$ 16
- `[ \f { [f 3] } \x { 2 * x } ]` $\Rightarrow$ 6
- `[ \x { \y { x + y} } 3 ]` $\Rightarrow$ `\y { 3 + y }`
- `\x { x + 1 }` :: `(int -> int)`
- `\x { \y { x + y } }` :: `(int -> (int -> int))`
- `\f { [f 3] }` ::

# Quick review of Lambda expressions

My syntax:

$$\underbrace{[\ \underbrace{\backslash x\ \{\ x + 1\}}_{\text{function}}\ \underbrace{(3 * 5)}_{\text{argument}}\ ]}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` ⇒ 16
- `[ \f { [f 3] } \x { 2 * x } ]` ⇒ 6
- `[ \x { \y { x + y} } 3 ]` ⇒ `\y { 3 + y }`
- `\x { x + 1 }` :: `(int -> int)`
- `\x { \y { x + y } }` :: `(int -> (int -> int))`
- `\f { [f 3] }` :: `((int -> ??)  -> ??)`

# Quick review of Lambda expressions

My syntax:

$$\underbrace{\texttt{[}\ \underbrace{\texttt{\textbackslash x \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ \texttt{]}}_{\text{application}}$$

- `[ \x { x + 1 } (3 * 5) ]` ⇒ 16
- `[ \f { [f 3] } \x { 2 * x } ]` ⇒ 6
- `[ \x { \y { x + y} } 3 ]` ⇒ `\y { 3 + y }`
- `\x { x + 1 }` :: `(int -> int)`
- `\x { \y { x + y } }` :: `(int -> (int -> int))`
- `\f { [f 3] }` :: `((int -> ??) -> ??)`

To do static recursive typechecking, we need **typed lambda expressions**.

# Typed Lambda expressions

My syntax:

$$\texttt{[}\ \underbrace{\underbrace{\texttt{\textbackslash x:int \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}}_{\text{application}}\ \texttt{]}$$

# Typed Lambda expressions

My syntax:

$$\underbrace{\texttt{[}\ \underbrace{\texttt{\textbackslash x:int \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ \texttt{]}}_{\text{application}}$$

- `\f:(int -> int) { [f 3] }` ::

# Typed Lambda expressions

My syntax:

$$\underbrace{\texttt{[}\ \underbrace{\texttt{\textbackslash x:int \{ x + 1\}}}_{\text{function}}\ \underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ \texttt{]}}_{\text{application}}$$

- `\f:(int -> int) { [f 3] }` :: `((int -> int) -> int)`
- `\f:(int -> (int -> int)) { [f 3] }` ::

# Typed Lambda expressions

My syntax:

$$\underbrace{\texttt{[ }\underbrace{\texttt{\textbackslash x:int \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}\texttt{ ]}}_{\text{application}}$$

- `\f:(int -> int) { [f 3] }` :: `((int -> int) -> int)`
- `\f:(int -> (int -> int)) { [f 3] }` ::
  `((int -> (int -> int)) -> (int -> int))`
- `\f:((int -> int) -> int) { [f x] }` ::

# Typed Lambda expressions

My syntax:

$$\underbrace{\texttt{[}\ \underbrace{\texttt{\textbackslash x:int \{ x + 1\}}}_{\text{function}}\underbrace{\texttt{(3 * 5)}}_{\text{argument}}\ \texttt{]}}_{\text{application}}$$

- `\f:(int -> int) { [f 3] }` :: `((int -> int) -> int)`
- `\f:(int -> (int -> int)) { [f 3] }` :: `((int -> (int -> int)) -> (int -> int))`
- `\f:((int -> int) -> int) { [f x] }` :: `(((int -> int) -> int) -> int)`

# Typed lambda expressions and applications in our language

```
f := \x:int {x+1};  x := [f 3]

g := [\x:int {\y:int {x+y}} 4];  x := [g 5]

j := \f:(int->int){\y:int{ [f y] };
k := [j \x:int { 2*x }];
x := [k 5]

x := [[\x:int { \y:int {x + y} } 13] 12]
y := [\x:int { [\y:int {x + y} 13] } 12]
```

# Typed `lambda` expressions and applications in our language

```
f := \x:int {x+1};  x := [f 3]

g := [\x:int {\y:int {x+y}} 4];  x := [g 5]

j := \f:(int->int){\y:int{ [f y] };
k := [j \x:int { 2*x }];
x := [k 5]

x := [[\x:int { \y:int {x + y} } 13] 12]
y := [\x:int { [\y:int {x + y} 13] } 12]
```

Are all of these type safe?

# Typed `lambda` expressions and applications in our language

```
f := \x:int {x+1};  x := [f 3]

g := [\x:int {\y:int {x+y}} 4];  x := [g 5]

j := \f:(int->int){\y:int{ [f y] };
k := [j \x:int { 2*x }];
x := [k 5]

x := [[\x:int { \y:int {x + y} } 13] 12]
y := [\x:int { [\y:int {x + y} 13] } 12]
```

Are all of these type safe?
You will learn how to add `lambdas` to your interpreters very soon.
Today we'll just do a typechecker for them.

# Two new types of Arithmetic Expressions for Parsing

```
data AExp =
    Var String
  | Num Int
  | Plus AExp AExp
  | Times AExp AExp
  | Neg AExp
  | Div AExp AExp
  | Lambda String Type AExp  --NEW
  | App AExp AExp             --NEW
   deriving (Show, Eq)
```

- We won't talk about the parsing today.
- AExp are not always numbers anymore!

# Example typechecking

Expressions

```
[\x:int {99} 22] + 3                          --accept
[\x:(int->int) {99} 22]                       --reject
[\x:(int->int) {[x 2]} 9]                     --reject
[\x:(int->int) {[x 2]} \x:int{9}]             --accept
[\x:(int->int) {x + 2} \x:int{9}]             --reject
```

Programs

```
x := \x:int{x+1};
IF x < 2 THEN y := 3 ELSE y := 4;             --reject

x := 3;
z := [\y:int {x + y} 5];
w := x + y                                    --reject
```

# A type for types

```
data Type =
     BoolType
   | IntType
   | LambdaType Type Type
   | FailureType
    deriving (Show, Eq)

type TypeStore = Map.Map VarName Type
```

```
x := 5; y := \x:int {x+x};  z := \x:(int->int) { [x 4] }
```

# Typechecking Statements

```
typeCheckStmt :: (Stmt AExp BExp) -> TypeStore
                                  -> (Bool, TypeStore)
typeCheckStmt program store =
 case program of
  ...
  Seq s1 s2 ->
    let (s1Good, store') = typeCheckStmt s1 store
    in if not s1Good
       then (False, store')
       else let (s2Good, store'')  = typeCheckStmt s2 store'
            in (s2Good, store'')
  ...
```

```
x := 5; y := \w:int {x+w};  z := [y x]
```

# Typechecking Statements

```
typeCheckStmt :: (Stmt AExp BExp) -> TypeStore
                                   -> (Bool, TypeStore)
typeCheckStmt program store =
  case program of
    ...
    Skip -> (True, store)
    ...
```

# Typechecking Statements

```
typeCheckStmt :: (Stmt AExp BExp) -> TypeStore
                           -> (Bool, TypeStore)
typeCheckStmt program store =
 case program of
  ...
  If b s1 s2 ->
    let (bType, store') = findTypeBExp b store in
    if bType /= BoolType
    then (False, store')
    else let (s1Good, store'') = typeCheckStmt s1 store' in
       if not s1Good
       then (False, store'')
       else let (s2Good, store''') = typeCheckStmt s1 store''
       in (s2Good, store''')
       -- Do you see a problem here?
```

# Typechecking Statements

```
typeCheckStmt :: (Stmt AExp BExp) -> TypeStore
                              -> (Bool, TypeStore)
typeCheckStmt program store =
 case program of
  ...
  If b s1 s2 ->
    let (bType, store') = findTypeBExp b store in
    if bType /= BoolType
    then (False, store')
    else let (s1Good, store'') = typeCheckStmt s1 store' in
       if not s1Good
       then (False, store'')
       else let (s2Good, store''') = typeCheckStmt s1 store''
       in (s2Good, store''')
       -- Do you see a problem here?
IF a < b THEN z := 3 ELSE z := \x:int{x+1} END; w := z + 1
```

# Typechecking Statements

```
typeCheckStmt :: (Stmt AExp BExp) -> TypeStore
                             -> (Bool, TypeStore)
typeCheckStmt program store =
 case program of
  ...
    While b s ->
      let (bType, store') = findTypeBExp b store in
      if bType /= BoolType
      then (False, store')
      else let (sGood, store'') = typeCheckStmt s store' in
           (sGood, store'')
  ...
```

# Typechecking Statements

```
typeCheckStmt :: (Stmt AExp BExp) -> TypeStore
                             -> (Bool, TypeStore)
typeCheckStmt program store =
 case program of
  ...
    Assign x val ->
      let (valType, store') = findTypeAExp val store
      in if valType /= FailureType
         then (True, Map.insert x valType store')
         else (False, store)
  ...
```

# Typechecking Boolean Expressions

```
findTypeBExp :: (BExp AExp) -> TypeStore
                    -> (Type, TypeStore)
findTypeBExp b store =
  case b of
    ...
    Bool x -> (BoolType, store)
    ...
```

# Typechecking Boolean Expressions

```
findTypeBExp :: (BExp AExp) -> TypeStore
                    -> (Type, TypeStore)
findTypeBExp b store =
  case b of
    ...
    Or x y ->
    let (t, store') = (findTypeBExp x store) in
    if t /= BoolType
    then (FailureType, store')
    else let (t', store'') = (findTypeBExp y store') in
                if t' /= BoolType
                then (FailureType, store'')
                else (BoolType, store'')
    ...
```

# Typechecking Boolean Expressions

```
findTypeBExp :: (BExp AExp) -> TypeStore
                            -> (Type, TypeStore)
findTypeBExp b store =
  case b of
    ...
    Lt x y ->
     let (t, store') = (findTypeAExp x store) in
     if t /= IntType
     then (FailureType, store')
     else let (t', store'') = (findTypeAExp y store') in
               if t' /= IntType
               then (FailureType, store'')
               else (BoolType, store'')
    ...
```

# Typechecking Arithmetic Expressions

```
findTypeAExp :: AExp -> TypeStore -> (Type, TypeStore)
findTypeAExp a store =
 case a of
  ...
  Num x -> (IntType, store)
  ...
```

# Typechecking Arithmetic Expressions

```
findTypeAExp :: AExp -> TypeStore -> (Type, TypeStore)
findTypeAExp a store =
 case a of
  ...
  Num x -> (IntType, store)
  ...
```

Next: What if it's a variable?

```
data AExp =
  ...
  Var String
```

# Typechecking Arithmetic Expressions

```
findTypeAExp :: AExp -> TypeStore -> (Type, TypeStore)
findTypeAExp a store =
 case a of
  ...
  Var x -> (Map.findWithDefault FailureType x store, store)
  ...
```

# Typechecking Arithmetic Expressions

```
findTypeAExp :: AExp -> TypeStore -> (Type, TypeStore)
findTypeAExp a store =
 case a of
  ...
  Var x -> (Map.findWithDefault FailureType x store, store)
  ...
```

Next: What if it's a `Plus`?

```
data AExp =
  ... Plus AExp AExp
```

# Typechecking Arithmetic Expressions

```
findTypeAExp :: AExp -> TypeStore -> (Type, TypeStore)
findTypeAExp a store =
 case a of
  ...
    Plus x y ->
    let (t, store') = (findTypeAExp x store) in
      if t /= IntType
      then (FailureType, store')
      else let (t', store'') = (findTypeAExp y store') in
                     if t' /= IntType
                     then (FailureType, store'')
                     else (IntType, store'')
  ...
```

# Typechecking Arithmetic Expressions

```
findTypeAExp :: AExp -> TypeStore -> (Type, TypeStore)
findTypeAExp a store =
 case a of
  ...
    Plus x y ->
    let (t, store') = (findTypeAExp x store) in
      if t /= IntType
      then (FailureType, store')
      else let (t', store'') = (findTypeAExp y store') in
                     if t' /= IntType
                     then (FailureType, store'')
                     else (IntType, store'')
  ...
```

Next: What if it's a Lambda?

```
data AExp = ... Lambda String Type AExp
```

# Typechecking Arithmetic Expressions

```
findTypeAExp :: AExp -> TypeStore -> (Type, TypeStore)
findTypeAExp a store =
 case a of
  ...
    Lambda s t a ->
    let (t', store') =
       findTypeAExp a (Map.insert s t store)
       in (LambdaType t t', store)
                    -- Why not store' ?
  ...
```

# Typechecking Arithmetic Expressions

```
findTypeAExp :: AExp -> TypeStore -> (Type, TypeStore)
findTypeAExp a store =
 case a of
  ...
    Lambda s t a ->
    let (t', store') =
       findTypeAExp a (Map.insert s t store)
       in (LambdaType t t', store)
                     -- Why not store' ?
  ...
```

Next: What if it's a application of a function?

```
data AExp =
  ...
  App AExp AExp
```

# Typechecking Arithmetic Expressions

```
findTypeAExp :: AExp -> TypeStore -> (Type, TypeStore)
findTypeAExp a store =
 case a of
  ...
    App f x ->
    let (t,store') = (findTypeAExp f store) in
      case t of
        LambdaType t1 t2 ->
           let (t',store'') = (findTypeAExp x store) in
           if t1 == t'
           then (t2, store)
           else (FailureType, store)
        _ (FailureType, store)

                  -- what about store' or store''?
  ...
```

# Typed recursive functions

What type information do you need to typecheck recursive functions?

```
REC f := \x:int { [ f (x+1) ] }

f :: (int -> ??)
```

# Typed recursive functions

What type information do you need to typecheck recursive functions?

```
REC f := \x:int { [ f (x+1) ] }
```

```
f :: (int -> ??)
```

```
REC f := int \x:int { [ f (x+1) ] }
```

```
f :: (int -> int)
```

```
REC f := (int -> int) \x:int { [ f (x+1) ] }
```

# Typed recursive functions

What type information do you need to typecheck recursive functions?

```
REC f := \x:int { [ f (x+1) ] }
```

```
f :: (int -> ??)
```

```
REC f := int \x:int { [ f (x+1) ] }
```

```
f :: (int -> int)
```

```
REC f := (int -> int) \x:int { [ f (x+1) ] }
```

```
f :: (int -> (int -> int))
```

# Typed recursive functions

What type information do you need to typecheck recursive functions?

```
REC f := \x:int { [ f (x+1) ] }
```

```
f :: (int -> ??)
```

```
REC f := int \x:int { [ f (x+1) ] }
```

```
f :: (int -> int)
```

```
REC f := (int -> int) \x:int { [ f (x+1) ] }
```

```
f :: (int -> (int -> int))
```

Recursive typechecking will be one of your exercises today.

# Without typechecking, recursion totally unnecessary!

Recursive version:

```
f1 := \x { If x < 1 Then 1 Else x * [f1 (x-1)] End};
[f1 5];
```

Doing the same thing without recursion:

```
f2 := \g {\x {If x < 1 Then 1 Else x * [[g g] (x-1)] End} };
[[f2 f2] 5]
```

# Without typechecking, recursion totally unnecessary!

Recursive version:

```
f1 := \x { If x < 1 Then 1 Else x * [f1 (x-1)] End};
[f1 5];
```

Doing the same thing without recursion:

```
f2 := \g {\x {If x < 1 Then 1 Else x * [[g g] (x-1)] End} };
[[f2 f2] 5]
```

This can actually be done in **Scheme**, which has no static typechecking. It cannot be done in Haskell. Why not?

# Your Turn!

Write typecheckers for these expressions and statements.

1. `IF x < 4 THEN 8 + x ELSE x + 9 END`
2. `LET x = 2 + 2 IN x + x END`
3. `REC f := int \x:int { [f x+1] }`
4. `REC f = int \x:int { [f x+1] } IN [f 3] END`

I've done the parsing for you,
just add clauses to the `findTypeAExp` and `typeCheckStmt` function.

```
data AExp = ...
  | Let String AExp AExp
  | IfExp (BExp AExp) AExp AExp
  | RecExp String Type String Type AExp AExp
  ...
data Stmt a b =
  | RecAssign VarName Type VarName Type a
  ...
```