Part I
Expressions and Types

What is the type of the following expression?

\[
\{ \text{lambda } \{x\} \{+ x 1\}\}
\]

**Answer:** Yet another trick question; it's not an expression in our typed language, because the argument type is missing

But it seems like the answer *should* be \(\text{num} \rightarrow \text{num}\)
Type Inference

*Type inference* is the process of inserting type annotations where the programmer omits them.

We’ll use explicit question marks, to make it clear where types are omitted.

\[
\{\text{lambda} \; \{[x : ?]\} \; \{+ \; x \; 1\}\}\]

\[<\text{Type}> ::= \text{num} \]
\[\mid \text{bool} \]
\[\mid (<\text{Type}> \; -> \; <\text{Type}>) \]
\[\mid ?\]
Part 2
Type Inference

\{ \textbf{lambda} \{ [x : ?] \} \ {+ x 1} \} \quad \text{T}_1 = \text{num}

- Create a new type variable for each \( ? \)
- Change type comparison to install type equivalences

\( (\text{num} \rightarrow \text{num}) \)
Type Inference

\[
\{\text{lambda } \{[x : ?]\} \{+ x 1\}\} \\
\text{T}_1 \quad \text{num} \\
\text{num} \quad \text{T}_1 = \text{num} \\
(num \rightarrow num)
\]

\[
\{\text{lambda } \{[x : ?]\} \{\text{if } #t 1 x\}\} \\
\text{bool} \quad \text{num} \quad \text{T}_1 \\
\text{num} \quad \text{T}_1 = \text{num} \\
(num \rightarrow num)
\]
Type Inference: Impossible Cases

{\text{lambda} \{[x : ?]\} \{\text{if} \ x \ 1 \ x\}}

\[
\begin{array}{c}
\text{T}_1 \\
\hline \\
\text{num} \\
\hline \\
\text{T}_1
\end{array}
\]

\textbf{no type:} \text{T}_1 \text{ can't be both } bool \text{ and } num
Type Inference: Many Cases

\{ \text{lambda \ { \text{[y : ?]} \}} \ y \} \\
\hspace{1cm} \text{T}_1 \\
\hspace{1cm} (\text{T}_1 \rightarrow \text{T}_1)

- Sometimes, more than one type works
  - \((\text{num} \rightarrow \text{num})\)
  - \((\text{bool} \rightarrow \text{bool})\)
  - \(((\text{num} \rightarrow \text{bool}) \rightarrow (\text{num} \rightarrow \text{bool}))\)

so the type checker leaves variables in the reported type
Part 3
Type Inference: Function Calls

\[
\text{\{lambda \{[y : ?]\} y\} \text{\{lambda \{[x : ?]\} \{+ x 1\}\}\}}
\]

\[
(T_1 \rightarrow T_1) \quad (num \rightarrow num)
\]

\[
(num \rightarrow num) \quad T_1 = (num \rightarrow num)
\]
Type Inference: Function Calls

\[
\{ \text{lambda } \{ [y : ?] \} \{ y \; 7 \} \}
\]

\[
\begin{align*}
&T_1 \quad \text{num} \\
&T_2 \quad T_1 = (\text{num} \rightarrow T_2) \\
&(\text{num} \rightarrow T_2) \rightarrow T_2
\end{align*}
\]

• In general, create a new type variable record for the result of a function call.
Part 4
Type Inference: Cyclic Equations

\[
\{ \text{lambda } \{ [x : ?] \} \{ x \ x \} \} \\

T_1 \rightarrow T_1 \\

T_1 = (T_1 \rightarrow T_2) \ \text{no solution}
\]

The \textbf{occurs check}:

- When installing a type equivalence, make sure that the new type for $T$ doesn’t already contain $T$
Part 5
Type Unification

For comparing types, replace

$$\text{equal?} : (\text{Type \ Type} \rightarrow \text{Boolean})$$

with

$$\text{unify!} : (\text{Type \ Type} \rightarrow ())$$
Type Unification

For comparing types, replace

```plaintext
equal? : (Type Type -> Boolean)
```

with

```plaintext
unify! : (Type Type Exp -> ())
```

To simplify by substituting with discovered equivalences:

```plaintext
resolve : (Type -> Type)
```
Type Unification

- \( \text{resolve } T_1 \Rightarrow T_1 \)

- **unify!** \( T_1 \) with \( \text{num} \)
  
  Then, \( \text{resolve of } T_1 = \text{num} \)

- So far, \( \text{resolve of } (T_1 \Rightarrow T_2) = (\text{num} \Rightarrow T_2) \)

  **unify!** \( T_1 \) with \( T_2 \)

  Then, \( \text{resolve of } T_2 = \text{num} \)
Part 6
Type Grammar, Again

\[ \langle \text{Type} \rangle \ ::= \ \text{num} \]
\[ \quad | \ \text{bool} \]
\[ \quad | \ (\langle \text{Type} \rangle \rightarrow \langle \text{Type} \rangle) \]
\[ \quad | \ ? \]
Representing Type Variables

(define-type Type
  (numT)
  (boolT)
  (arrowT [arg : Type]
    [result : Type])
  (varT [is : (Boxof (Optionof Type))]))

(varT (box (none)))
Representing Type Variables

```
(define-type Type
  (numT)
  (boolT)
  (arrowT [arg : Type]
    [result : Type])
  (varT [is : (Boxof (Optionof Type))]))

(varT (box (some (numT)))))
```
Representing Type Variables

(define-type Type
  (numT)
  (boolT)
  (arrowT [arg : Type]
    [result : Type])
  (varT [is : (Boxof (Optionof Type))]))

(define (unify! [t1 : Type] [t2 : Type] [expr : Exp])
  ....
  (type-case Type t1
    ...
      [(varT b)
        .... (set-box! b (some (resolve t2))) ....]
    ...)
  ....)
Part 7
Unification Examples

(test (unify! (numT)
        (numT))
(values))
Unification Examples

(test (unify! (boolT) (boolT)) (values))
Unification Examples

(test/exn (unify! (numT) (boolT)) "no type")
Unification Examples

(test (unify! (varT (box (none))))
    (numT))
(values))
Unification Examples

(test (unify! (varT (box (some (numT)))))
 (numT))
 (values))
Unification Examples

```
(test/exn (unify! (varT (box (some (boolT))))
  (numT))

"no type")
```
Unification Examples

(test/exn (let ([t (varT (box (none)))]))
  (begin
    (unify! t (numT))
    (unify! t (boolT)))
  "no type")
Unification Examples

(test (let ([t (varT (box (none)))]))
  (begin
    (unify! t
      (numT))
    (unify! t
      (numT)))
  (values))
Unification Examples

(test (let ([t (varT (box (none)))]))
(begin
  (unify! (arrowT t (boolT))
    (arrowT (numT) (boolT)))
  (unify! t
    (numT)))
(values))
Unification Examples

(test/exn (let ([t (varT (box (none)))]
              (unify! (arrowT t (boolT))
                       t))
          "no type")
Unification Examples

(test (let ([t1 (varT (box (none)))]
            [t2 (varT (box (none)))]
            (unify! t1
t2))
(values))
Unification Examples

(test/exn (let ([t1 (varT (box (none)))]
   [t2 (varT (box (none)))]
  (begin
   (unify! t1
      t2)
   (unify! t1
      (numT))
   (unify! t2
      (boolT)))
"no type")
Unification Examples

(test/exn (let ([t1 (varT (box (none)))]
                [t2 (varT (box (none)))]
  (begin
    (unify! t1 t2)
    (unify! t2 (boolT))
    (unify! t1 (numT)))
"no type")
Unification Examples

(test/exn (let ([t1 (varT (box (none)))]
                 [t2 (varT (box (none)))]
                 (begin
                  (unify! t1
                           (arrowT t2 (boolT)))
                  (unify! t1
                           (arrowT (numT) t2))))
             "no type")
Type Unification

**unify**! a type variable \( T \) with a type \( \tau \):
- If \( T \) is set to \( \tau_1 \), \( \text{unify!} \ \tau_1 \) with \( \tau_2 \)
- If \( \tau_2 \) is already equivalent to \( T \), succeed
- If \( \tau_2 \) contains \( T \), then fail
- Otherwise, set \( T \) to \( \tau_2 \) and succeed

**unify**! a type \( \tau_1 \) to type \( \tau_2 \):
- If \( \tau_2 \) is a type variable \( T \), then \( \text{unify!} \ T \) and \( \tau_1 \)
- If \( \tau_1 \) and \( \tau_2 \) are both \textit{num} or \textit{bool}, succeed
- If \( \tau_1 \) is \( (\tau_3 \to \tau_4) \) and \( \tau_2 \) is \( (\tau_5 \to \tau_6) \), then
  - \( \text{unify!} \ \tau_3 \) with \( \tau_5 \)
  - \( \text{unify!} \ \tau_4 \) with \( \tau_6 \)
- Otherwise, fail
Type Unification

(define (unify! [t1 : Type] [t2 : Type] [expr : Exp])
  (type-case Type t1
    [(varT is1)
      ...]
    [else
      (type-case Type t2
        [(varT is2) (unify! t2 t1 expr)]
        [(numT) (type-case Type t1
          [(numT) (values)]
            [else (type-error expr t1 t2)]
          [else (type-error expr t1 t2))]]
        [(boolT) (type-case Type t1
          [(boolT) (values)]
            [else (type-error expr t1 t2)]
          [else (type-error expr t1 t2))]]
        [(arrowT a2 b2) (type-case Type t1
          [(arrowT a1 b1)
            (begin
              (unify! a1 a2 expr)
              (unify! b1 b2 expr)]
            [else (type-error expr t1 t2))]]))))
Type Unification

(define (unify! [t1 : Type] [t2 : Type] [expr : Exp])
  (type-case Type t1
    [(varT is1) (type-case (Optionof Type) (unbox is1)
      [(some t3) (unify! t3 t2 expr)]
      [(none) (local [(define t3 (resolve t2))]
        (if (eq? t1 t3)
          (values)
          (if (occurs? t1 t3)
            (type-error expr t1 t3)
            (begin
              (set-box! is1 (some t3))
              (values)))))])
    [else ...]))
Type Unification Helpers

(define (resolve [t : Type]) : Type
  (type-case Type t
    [(varT is)
      (type-case (Optionof Type) (unbox is)
        [(none) t]
        [(some t2) (resolve t2)]
      )
    [else t]])

(define (occurs? [r : Type] [t : Type]) : Boolean
  (type-case Type t
    [(numT) #f]
    [(boolT) #f]
    [(arrowT a b)
      (or (occurs? r a)
          (occurs? r b))]
    [(varT is) (or (eq? r t)
      (type-case (Optionof Type) (unbox is)
        [(none) #f]
        [(some t2) (occurs? r t2)]))])))
Part 10
Type Checker with Inference

(define typecheck : (Exp Type-Env -> Type)
  (lambda (a tenv)
    (type-case Exp a
      ...
      [(numE n) (numT)]
      ...))))
Type Checker with Inference

(define typecheck : (Exp Type-Env -> Type)
  (lambda (a tenv)
    (type-case Exp a
      ...
      [(plusE l r)
        (begin
          (unify! (typecheck l env) (numT) l)
          (unify! (typecheck r env) (numT) r)
          (numT))]
      ...))))
Type Checker with Inference

(define typecheck : (Exp Type-Env -> Type)
 (lambda (a tenv)
   (type-case Exp a
     ...
     [(idE name) (get-type name env)]
     [(lamE n arg-type body)
       (arrowT arg-type
         (typecheck body (aBind name
                           arg-type
                           env)))]]
     ...)))}
Type Checker with Inference

(define typecheck : (Exp Type-Env -> Type)
  (lambda (a tenv)
    (type-case Exp a
      ...
      [(appE fn arg)
        (local [(define result-type (varT (box (none)))))
          (begin
            (unify! (arrowT (typecheck arg env) result-type)
              (typecheck fn env)
              fn)
              result-type))]
      ...))))
Part II
Type Errors

Checking — report that an expression doesn’t have an expected type (expressed as a string):

\[
\text{type-error : (Exp String -> ...)}
\]

Inference — report that, near some expression, two types are incompatible:

\[
\text{type-error : (Exp Type Type -> ...)}
\]