Part 1

Quiz

What is the type of the following expression?

```
{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 (***num*** $\rightarrow$ ***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

```
{lambda {[x : ?]} {+ x 1}}
```

```
<Type>  ::=  num
          |  bool
          |  (<Type> -> <Type>)
          |  ?
```

Part 2

Type Inference

```
{lambda {[x : ?]} {+ x 1}}
```

T_1 *num*

num $T_1 = num$

$(num \rightarrow num)$

- Create a new type variable for each **?**
- Change type comparison to install type equivalences

Type Inference

```
{lambda {[x : ?]} {+ x 1}}
```

T_1 *num*

num T_1 = *num*

(*num* → *num*)

```
{lambda {[x : ?]} {if true 1 x}}
```

bool *num* T_1

num T_1 = *num*

(*num* → *num*)

Type Inference: Impossible Cases

```
{lambda {[x : ?]} {if x 1 x}}
```

$\mathbf{T}_1$ *num* $\mathbf{T}_1$

no type: $\mathbf{T}_1$ can't be both *bool* and *num*

Type Inference: Many Cases

```
{lambda {[y : ?]} y}
```

$\mathbf{T}_1$

$(\mathbf{T}_1 \rightarrow \mathbf{T}_1)$

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

  so the type checker leaves variables in the reported type

Part 3

Type Inference: Function Calls

```
{{lambda {[y : ?]} y} {lambda {[x : ?]} {+ x 1}}}
```

$(\mathbf{T}_1 \rightarrow \mathbf{T}_1)$ $(num \rightarrow num)$

$(num \rightarrow num)$

$\mathbf{T}_1 = (num \rightarrow num)$

Type Inference: Function Calls

`{lambda {[y : ?]} {y 7}}`

T_1 $\quad$ *num*

T_2 $\quad$ $T_1 = (num \to T_2)$

$((num \to T_2) \to T_2)$

- In general, create a new type variable record for the result of a function call

Part 4

Type Inference: Cyclic Equations

```
{lambda {[x : ?]} {x x}}
```

$\mathbf{T}_1$ $\mathbf{T}_1$

no type: $\mathbf{T}_1$ can't be ($\mathbf{T}_1 \rightarrow$ ...)

- $\mathbf{T}_1$ can't be ***num***
- $\mathbf{T}_1$ can't be ***bool***
- Suppose $\mathbf{T}_1$ is ($\mathbf{T}_2 \rightarrow \mathbf{T}_3$)
  - $\mathbf{T}_2$ must be $\mathbf{T}_1$
  - So we won't get anywhere!

Type Inference: Cyclic Equations

```
{lambda {[x : ?]} {x x}}
```

T_1 T_1

no type: T_1 can't be ($T_1 \rightarrow$ ...)

The ***occurs check***:

- When installing a type equivalence, make sure that the new type for **T** doesn't already contain **T**

Part 5

Type Unification

Unify a type variable **T** with a type τ_2:

- If **T** is set to τ_1, unify τ_1 and τ_2
- If τ_2 is already equivalent to **T**, succeed
- If τ_2 contains **T**, then fail
- Otherwise, set **T** to τ_2 and succeed

`(resolve` τ_2`)` is **T**?

`(occurs? T (resolve` τ_2`))`

Unify a type τ_1 to type τ_2:

- If τ_2 is a type variable **T**, then unify **T** and τ_1
- If τ_1 and τ_2 are both ***`num`*** or ***`bool`***, succeed
- If τ_1 is $(\tau_3 \rightarrow \tau_4)$ and τ_2 is $(\tau_5 \rightarrow \tau_6)$, then
  - unify τ_3 with τ_5
  - unify τ_4 with τ_6
- Otherwise, fail

Type Errors

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

```
type-error : (ExprC string -> ...)
```

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

```
type-error : (ExprC Type Type -> ...)
```

Part 6

Type Grammar, Again

```
<Type>  ::=  num
          |  bool
          |  (<Type> -> <Type>)
          |  ?
```

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 (numT)))
```

Representing Type Variables

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


 (type-case Type t
   ...
   [varT (b)
         ...
         (set-box! b (numT))
         ...]
   ...)
```

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 (numT)))
              (numT))
      (values))
```

Unification Examples

```
(test/exn (unify! (varT (box (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")
```

Part 8

Type Unification

```
(define (unify! [t1 : Type] [t2 : Type] [expr : ExprC])
  (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)])]
       [boolT () (type-case Type t1
                   [boolT () (values)]
                   [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 : ExprC])
  (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 () false]
    [boolT () false]
    [arrowT (a b)
            (or (occurs? r a)
                (occurs? r b))]
    [varT (is) (or (eq? r t)
                   (type-case (optionof Type) (unbox is)
                     [none () false]
                     [some (t2) (occurs? r t2)]))]))
```

Part 9

Type Checker with Inference

```
(define typecheck : (ExprC TypeEnv -> Type)
  (lambda (a tenv)
    (type-case ExprC a
      ...
      [numC (n) (numT)]
      ...)))
```

Type Checker with Inference

```
(define typecheck : (ExprC TypeEnv -> Type)
  (lambda (a tenv)
    (type-case ExprC a
      ...
      [plusC (l r) (begin
                     (unify! (typecheck l env) (numT) l)
                     (unify! (typecheck r env) (numT) r)
                     (numT))]
      ...)))
```

Type Checker with Inference

```
(define typecheck : (ExprC TypeEnv -> Type)
  (lambda (a tenv)
    (type-case ExprC a
      ...
      [idC (name) (get-type name env)]
      [lamC (n arg-type body)
            (arrowT arg-type
                    (typecheck body (aBind name
                                           arg-type
                                           env)))]
      ...)))
```

Type Checker with Inference

```
(define typecheck : (ExprC TypeEnv -> Type)
  (lambda (a tenv)
    (type-case ExprC a
      ...
      [appC (fn arg)
            (local [(define result-type (varT (box (none))))]
              (begin
                (unify! (arrowT (typecheck arg env)
                                result-type)
                        (typecheck fn env)
                        fn)
                result-type))]
      ...)))
```