Quiz

What type is inferred for $?$ in the following expression?

```
{with {f : (? -> ?) {fun {x : ?} x}} {f 10}}
```

**Answer:** `num`
Quiz

What type is inferred for ? in the following expression?

```
(with {f : (? -> ?) {fun {x : ?} x}}
    {f {fun {x : num} x}})
```

**Answer:** \((\text{num} \rightarrow \text{num})\)
Quiz

What type is inferred for ? in the following expression?

```plaintext
{with {f : (? → ?) {fun {x : ?} x}}
  {if0 ...
    {f 10}
    {f {fun {x : num} x} 8}}}
```

**Answer:** None; no single τ works — but it’s a perfectly good program for any ... or type num
### Polymorphism

We’d like a way to write a type that the caller chooses:

```coffeescript
{with {f : ?
    [tyfun [alpha]
        {fun {x : alpha} x}]}
{if0 ...
    {[@ f num] 10
    {[@ f (num -> num)] {fun {x : num} x} 8}]]

This f is **polymorphic**

- The `tyfun` form parameterizes over a type
- The `@` form picks a type
Polymorphic Types

What is the type of this expression?

\[
[\text{tyfun} \ [\alpha]
  \{\text{fun} \ \{x : \alpha\} \ x\}]
\]

It should be something like (\(\alpha \rightarrow \alpha\)), but it needs a specific type before it can be used as a function.
Polymorphic Types

What is the type of this expression?

\[
\text{tyfun } \alpha \text{ tyfun } \beta \{
\text{fun } \{x : \alpha \} \ x\}\}
\]

It should be something like \((\alpha \to \alpha)\), but picking \(\alpha\) gives something that still needs another type

New type form: \(\forall <\text{tyid}>. <\text{TE}>\)

\(\forall \alpha. (\alpha \to \alpha)\)

\(\forall \alpha. \forall \beta. (\alpha \to \alpha)\)
TPFAE Grammar

<TPFAE> ::= <num>
    | {+ <TPFAE> <TPFAE>}
    | {- <TPFAE> <TPFAE>}
    | <id>
    | {fun {<id> : <TE>} <TPFAE>}
    | {<TPFAE> <TPFAE>}
    | {if0 <TPFAE> <TPFAE> <TPFAE>}
    | [tyfun [<tyid>] <TPFAE>]
    | [@ <TPFAE> <TE>]

<TE> ::= num
    | (<TE> -> <TE>)
    | (forall <tyid> <TE>)
    | <tyid>
TPFAE Type Checking

\[ \Gamma[\langle tyid\rangle] \vdash e : \tau \]

\[ \Gamma \vdash [\text{tyfun } \langle tyid\rangle e] : \forall \langle tyid\rangle.\tau \]

\[ \Gamma \vdash \tau_0 \quad \Gamma \vdash e : \forall \langle tyid\rangle.\tau_1 \]

\[ \Gamma \vdash [\@ e \tau_0] : \tau_1[\langle tyid\rangle \leftarrow \tau_0] \]

\[ [\ldots \langle tyid\rangle \ldots] \vdash \langle tyid\rangle \]

\[ \Gamma[\langle tyid\rangle] \vdash \tau \]

\[ \Gamma \vdash \forall \langle tyid\rangle.\tau \]
If we mix `tyfun` with `withtype`, then we can write

```plaintext
{with {f : (forall alpha (alpha -> num))}
  [tyfun [alpha]
    {fun {v : alpha}
      {withtype {list {empty num}
        {cons (alpha * list)}}}
      {rec {len : (list -> num)
        {fun {l : list}
          {cases list l
            {empty {n} 0}
            {cons {fxr}
              {+ 1 {len {snd fxr}}}}}}}}
      {len {cons {pair v
        {cons {pair v
          {empty 0}}}}}}}}
  {+ {[@ f num] 10}
    {[@ f (num -> num)] {fun {x : num} x}}}}}
```

This is a kind of polymorphic list definition

**Problem:** everything must be under a `tyfun`
Solution: build \texttt{tyfun}-like abstraction into \texttt{withtype}

\begin{verbatim}
{withtype {{\alpha} list} {empty num}
  {cons (\alpha \ast {\alpha} list))}}
{rec {len : (forall \alpha ({{\alpha} list} \to num))
    [tyfun [\alpha]
      {fun {l : {\alpha} list}}
      {cases {\alpha} list} l
      {empty {n} 0}
      {cons {fxr}
        [+ 1 {len {snd fxr}}]}}}}}
{+ {[@ len num] {[@ cons num] {pair 1 {[@ empty num] 0}}}}}
{[@ len (num \to num)] {[@ empty (num \to num)] 0}}}
\end{verbatim}
Polymorphism and Inference

{\textbf{with} \{ f : (\forall\alpha (\alpha \to \alpha)) \}
 [\textit{tyfun} [\alpha]
 [\textit{fun} \{ x : \alpha \}
 x]]
 [[[\land f (\text{num} \to \text{num})] \{ \textit{fun} \{ y : \text{num} \} y \}]]

The type application [[[\land f (\text{num} \to \text{num})] \{ \textit{fun} \{ y : \text{num} \} y \}]] is obvious, since we can get the type of \{ \textit{fun} \{ y : \text{num} \} y \}

With polymorphism, type inference is usually combined with type-application inference:

{\textbf{with} \{ f : (\forall\alpha (\alpha \to \alpha)) \}
 [\textit{tyfun} [\alpha]
 [\textit{fun} \{ x : \alpha \}
 x]]
 [f \{ \textit{fun} \{ y : \text{num} \} y \}]}
Polymorphism and Inference

{with {f : {?}}
{fun {x : {?}}
 x}}
{f {fun {y : num} {f 10}}} }}

How about inferring a tyfun around the value of f?

Yes, with some caveats...
Polymorphism and Inference

Does the following expression have a type?

```
{fun {x : ?} {x x}}
```

Yes, if we infer `forall` types and type applications:

```
{fun {x : (forall alpha (alpha -> alpha))} {
  [@ x (num -> num)] [@ x num]}
```

Inferring types like this is arbitrarily difficult (i.e., undecidable), so type systems generally don’t
Let-Based Polymorphism

Inference constraint: only infer a polymorphic type (and insert `tyfun`) for their right-hand side of a `with` or `rec` binding

- This works:

  ```
  {with {f : ?
       {fun {x : ?}
        x}}
  {f {fun {y : num} {f 10}}}]
  ```

- This doesn’t:

  ```
  {fun {x : ?} {x x}}
  ```

**Note:** makes `with` a core form

**Implementation:** check right-hand side, add a `forall` and `tyfun` for each unconstrained `new` type variable
Polymorphism and Inference and Type Definitions

All three together make a practical programming system:

```ml
{withtype {{alpha list} {empty num}}
    {cons (alpha * {alpha list})}}
{rec {len : ?
    {fun {l : {alpha list}}
        {cases {alpha list} l
            {empty {n} 0}
            {cons {fxr}
                [+ 1 {len {snd fxr}}]}}
        [+ {len {cons {pair 1 {empty 0}}}}]
    {len {cons {pair {fun {x : num} x} {empty 0}}}}}}}
```

Caml example:

```ml
type 'a tree = Leaf of 'a
            | Fork of 'a tree * 'a tree
```
Polymorphism and Values

A *polymorphic function* is not quite a function:
- A function is applied to a value to get a new value
- A polymorphic function is applied to a type to get a function

What happens if you write the following?

``` With {f : ?} {fun {g : ?} 
  {fun {v : ?} 
    {g v}}}} 
  {with {g : ?} {fun {x : ?} x}} 
  {{f g} 10}}
```

A type application must be used at the function call, not in \( f \):

``` {{[@ [@ f num] num] 10} [@ g num]}
```
Polymorphism and Values

A *polymorphic function* is not quite a function:

- A **function** is applied to a value to get a new value
- A **polymorphic function** is applied to a type to get a function

What happens if you write the following?

```
{with {f : ? {fun {v : ?}}
  {fun {g : (forall alpha (alpha -> alpha))}
    {g v}}}}
{with {g : ? {fun {x : ?} x}}
  {{f 10} g}}}
```

One type application must be used inside **f**:

```
[tyfun {beta} {fun {v : beta}
  {fun {g : (forall alpha (alpha -> alpha))}
    {[[@ g beta] v]}}]]
```
Polymorphism and Values

An argument that is a polymorphic value can be used in multiple ways:

```{fun \{g : (\forall \alpha. \alpha \rightarrow \alpha)\}\}
{if \{g \ false\}
{g 0}
{g 1}\}```

but due to inference constraints,

```{fun \{g : ?\}
{if \{g \ false\}
{g 0}
{g 1}\}}```

would be rejected!
ML prohibits polymorphic values, so that

```
{fun {g : (forall alpha (alpha -> alpha))} {if {g false} {g 0} {g 1}}} is not allowed
```

is not allowed

• Consistent with inference

• Every `forall` appears at the beginning of a type, so

```
(forall alpha (forall beta (alpha -> beta)))
```

can be abbreviated

```
(alpha -> beta)
```

without loss of information