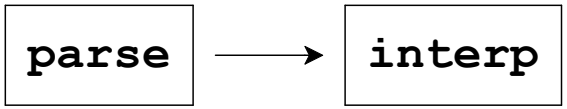
Part I

Checking Programs

We `parse` Moe programs before we `interp` them:

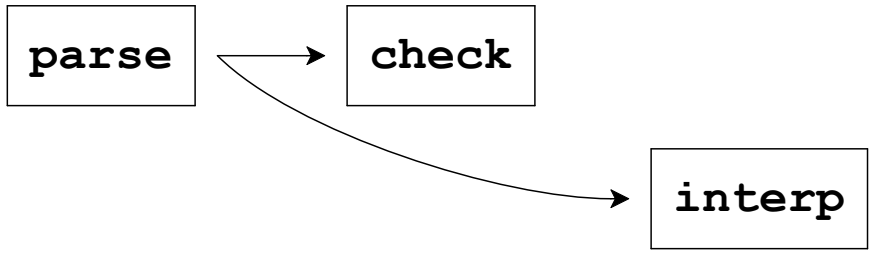


As a result, certain kinds of problems are ruled out for `interp`

```
1 + + + 2
```

Checking Programs

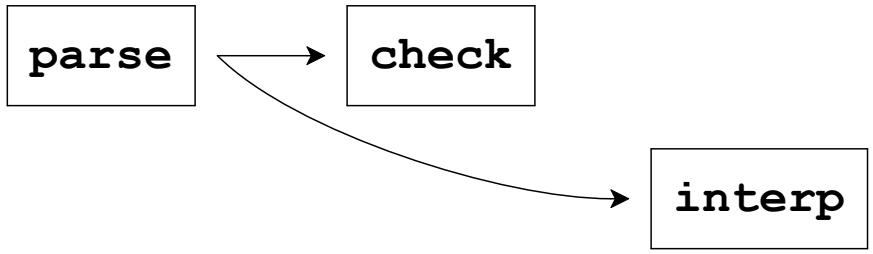
A **typed language** has more checks to rule out more problems:



- Programmer gets guarantee that `interp` can't fail in certain ways

Checking Programs

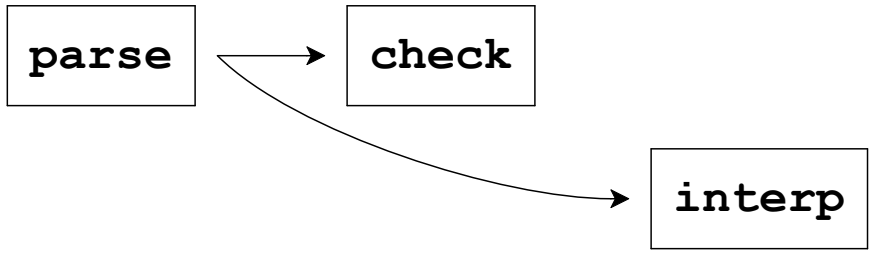
A **typed language** has more checks to rule out more problems:



- Programmer gets guarantee that `interp` can't fail in certain ways
- Implementation of `interp` gets to ignore some possibilities

Checking Programs

A **typed language** has more checks to rule out more problems:

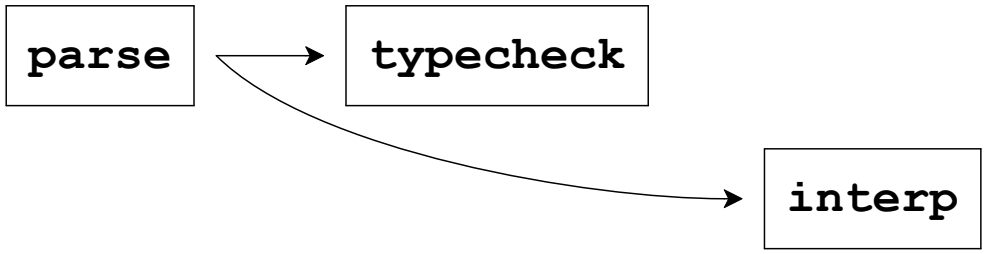


- Programmer gets guarantee that `interp` can't fail in certain ways
- Implementation of `interp` gets to ignore some possibilities

`check` must not call `interp`

Checking Programs

A **typed language** has more checks to rule out more problems:

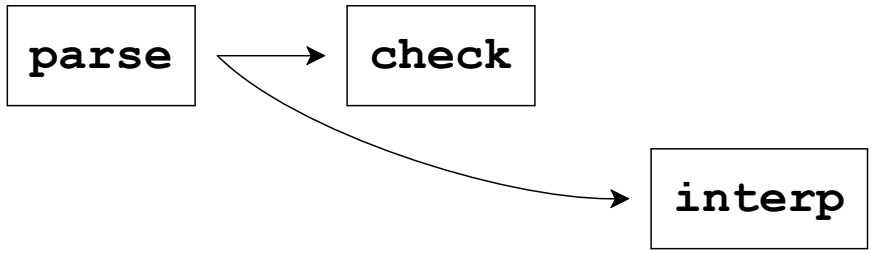


- Programmer gets guarantee that `interp` can't fail in certain ways
- Implementation of `interp` gets to ignore some possibilities

`typecheck` must not call `interp`

Checking Programs

A **typed language** has more checks to rule out more problems:

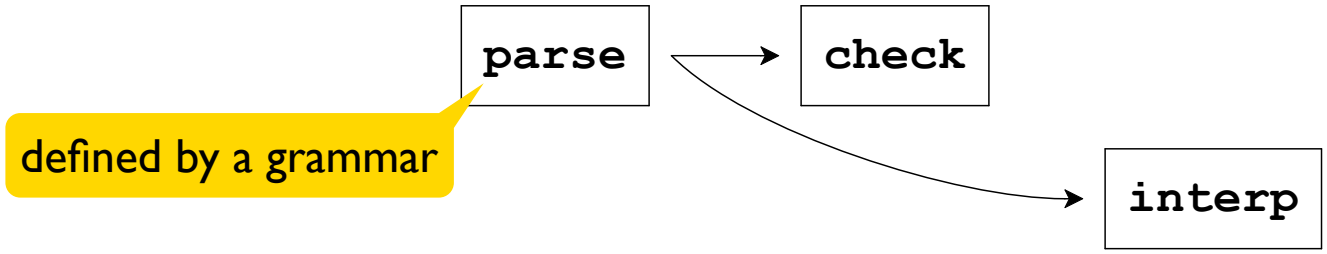


- Programmer gets guarantee that `interp` can't fail in certain ways
- Implementation of `interp` gets to ignore some possibilities

`check` must not call `interp`

Checking Programs

A **typed language** has more checks to rule out more problems:



- Programmer gets guarantee that `interp` can't fail in certain ways
- Implementation of `interp` gets to ignore some possibilities

`check` must not call `interp`

Language Grammar

```
<Exp>  ::=  <Int>
         |  #true
         |  #false
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  <Exp> == <Exp>
         |  <Symbol>
         |  fun (<Symbol>, ...): <Exp>
         |  <Exp>(<Exp>, ...)
         |  if <Exp> | <Exp> | <Exp>
         |  (<Exp>)
```

Same strategy for `check`?

Expressions and Values

Should `check` allow this expression?

```
1 + 2
```

Yes, and `interp` will produce 3

Expressions and Values

Is should the checker allow this expression?

```
(fun (x): x) + 8
```

We'd like to rule this out, since it produces an error:

```
interp: not a number
```

So, `check` should not allow a `fun` expression inside a `+`

Expressions and Values

Should the checker allow this expression?

```
(fun (x): x)(7) + 5
```

Depends on what we meant by *inside* in our most recent agreement

- *Anywhere inside* — **No**
- *Immediately inside* — **Yes**

Since our intrepreter produces 12, and since that result makes sense, let's agree on *immediately inside*

Expressions and Values

Should the checker allow this expression?

```
(fun (x): x)(fun (y): y) + 5
```

We don't want it to be!

Starts to look like we have to `interp`...

Expressions and Values

Is it possible to define ***well-formed*** (as a decidable property) so that we reject all expressions that produce errors?

Yes: reject *all* expressions!

Expressions and Values

Is it possible to define ***well-formed*** (as a decidable property) so that we reject *only* expressions that produce errors?

No

```
1 + if .... | 1 | fun (x): x
```

If we always knew whether `....` produces true or false, we could solve the halting problem

Types

We cannot reject *only* bad programs

In the process of rejecting expressions that are certainly bad, also reject some expressions that are good

```
1 + (if is_prime(131101)
     | 1
     | fun (x): x)
```

Overall strategy:

- Assign a ***type*** to each expression *without interpreting*
- Compute the type of a complex expression based on the types of its subexpressions

Part 2

Type Checker

```
check :: Exp -> Boolean
```

Type Checker

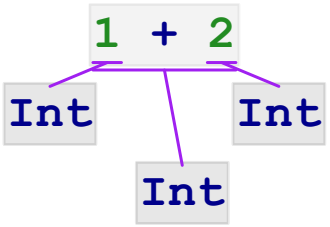
```
typecheck :: Exp -> Type
```

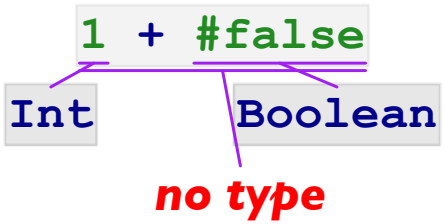
```
<Exp> : <Type>
```

Types

```
1 : Int
```

```
#true : Boolean
```





Type Rules

`<Int>` : `Int`

`#true` : `Boolean`

`#false` : `Boolean`

$$\frac{\texttt{<Exp>}_1 : \texttt{Int} \qquad \texttt{<Exp>}_2 : \texttt{Int}}{\texttt{<Exp>}_1 \texttt{ + } \texttt{<Exp>}_2 : \texttt{Int}}$$

`1` : `Int`

`#true` : `Boolean`

$$\frac{\texttt{1} : \texttt{Int} \qquad \texttt{2} : \texttt{Int}}{\texttt{1 + 2} : \texttt{Int}}$$

$$\frac{\texttt{1} : \texttt{Int} \qquad \texttt{\#false} : \texttt{Boolean}}{\texttt{1 + \#false} : \textit{no type}}$$

Type Rules

$$\texttt{<Int>} : \texttt{Int}$$

$$\texttt{\#true} : \texttt{Boolean}$$

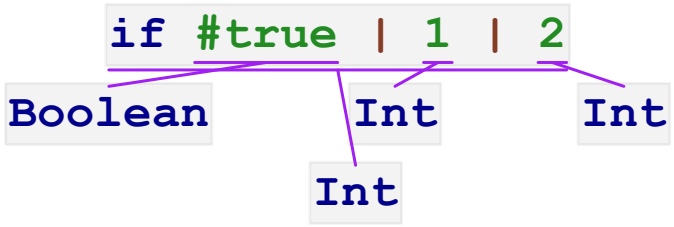
$$\texttt{\#false} : \texttt{Boolean}$$

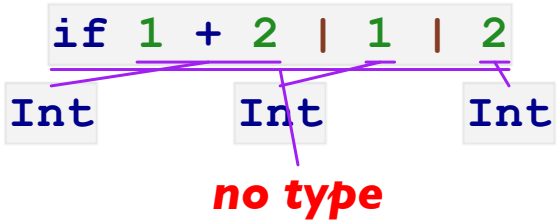
$$\frac{\texttt{<Exp>}_1 : \texttt{Int} \qquad \texttt{<Exp>}_2 : \texttt{Int}}{\texttt{<Exp>}_1 \texttt{ + } \texttt{<Exp>}_2 : \texttt{Int}}$$

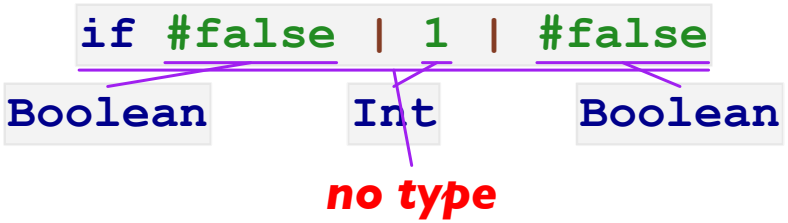
$$\frac{\dfrac{\texttt{1} : \texttt{Int} \qquad \texttt{2} : \texttt{Int}}{\texttt{1 + 2} : \texttt{Int}} \qquad \texttt{3} : \texttt{Int}}{\texttt{1 + 2 + 3} : \texttt{Int}}$$

Part 3

Types: Conditionals







Conditional Type Rules

$$\frac{\texttt{<Exp>}_1 \texttt{ : Boolean} \qquad \texttt{<Exp>}_2 \texttt{ : <Type>}_0 \qquad \texttt{<Exp>}_3 \texttt{ : <Type>}_0}{\texttt{if <Exp>}_1 \texttt{ | <Exp>}_2 \texttt{ | <Exp>}_3 \texttt{ : <Type>}_0}$$

$$\frac{\texttt{\#true : Boolean} \qquad \texttt{1 : Int} \qquad \texttt{2 : Int}}{\texttt{if \#true | 1 | 2 : Int}}$$

$$\frac{\texttt{1 + 2 : Int} \qquad \texttt{1 : Int} \qquad \texttt{2 : Int}}{\texttt{if 1 + 2 | 1 | 2 : } \textit{\textbf{no type}}}$$

$$\frac{\texttt{\#false : Boolean} \qquad \texttt{1 : Int} \qquad \texttt{\#false : Boolean}}{\texttt{if \#false | 1 | \#false : } \textit{\textbf{no type}}}$$

Part 4

Types: Variables and Functions

```
x : no type
```

```
fun (x :: Boolean): x
```

```
Boolean
```

```
Boolean -> Boolean
```

```
fun (x :: Boolean): if x | 1 | 2
```

```
Boolean     Int     Int
```

```
Int
```

```
Boolean -> Int
```

Variable and Function Type Rules

$$[\ \ldots\ \texttt{<Symbol>}\leftarrow\tau\ \ldots\ ] \vdash \texttt{<Symbol>} : \tau$$

$$\frac{\Gamma[\ \texttt{<Symbol>}\leftarrow\tau_1\ ] \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{fun (<Symbol> :: } \tau_1 \texttt{) : e} : \tau_1 \texttt{ ->} \tau_2}$$

Abbreviations: τ = **<Type>**
e = **<Exp>**
Γ = **<Env>**

Variable and Function Type Rules

$$[\ \ldots\ \texttt{<Symbol>}\leftarrow\tau\ \ldots\ ] \vdash \texttt{<Symbol>} : \tau$$

$$\frac{\Gamma[\texttt{<Symbol>}\leftarrow\tau_1] \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{fun (<Symbol> :: } \tau_1 \texttt{) : e} : \tau_1 \texttt{ -> } \tau_2}$$

$$\varnothing \vdash \texttt{x} : \textit{no type}$$

$$\frac{[\texttt{x}\leftarrow\texttt{Boolean}] \vdash \texttt{x} : \texttt{Boolean}}{\varnothing \vdash \texttt{fun (x :: Boolean): x} : \texttt{Boolean -> Boolean}}$$

$$\frac{\dfrac{[\texttt{x}\leftarrow\texttt{Boolean}] \vdash \texttt{x} : \texttt{Boolean} \quad [\texttt{x}\leftarrow\texttt{Boolean}] \vdash \texttt{1} : \texttt{Int} \quad [\texttt{x}\leftarrow\texttt{Boolean}] \vdash \texttt{2} : \texttt{Int}}{[\texttt{x}\leftarrow\texttt{Boolean}] \vdash \texttt{if x | 1 | 2} : \texttt{Int}}}{\varnothing \vdash \texttt{fun (x :: Boolean): if x | 1 | 2} : \texttt{Boolean -> Int}}$$

Revised Rules

$$\Gamma \vdash \texttt{<Num>} : \texttt{Int}$$

$$\Gamma \vdash \texttt{\#true} : \texttt{Boolean}$$

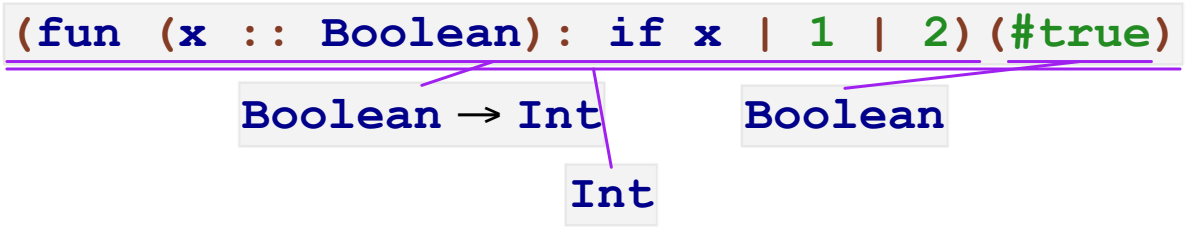
$$\Gamma \vdash \texttt{\#false} : \texttt{Boolean}$$

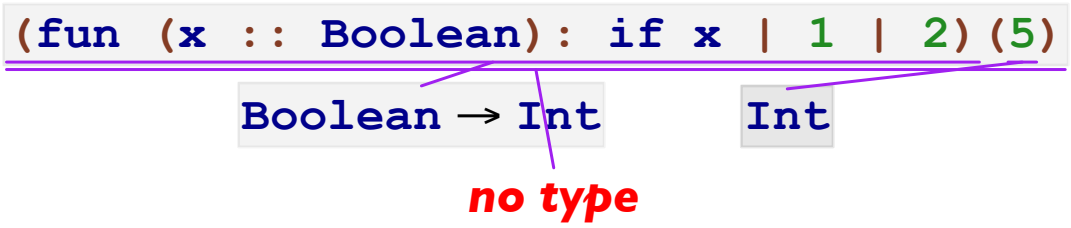
$$\frac{\Gamma \vdash \texttt{e}_1 : \texttt{Int} \qquad \Gamma \vdash \texttt{e}_2 : \texttt{Int}}{\Gamma \vdash \texttt{e}_1 \texttt{ + } \texttt{e}_2 : \texttt{Int}}$$

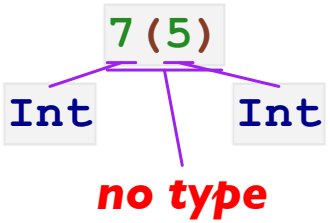
$$\frac{\Gamma \vdash \texttt{e}_1 : \texttt{Boolean} \qquad \Gamma \vdash \texttt{e}_2 : \tau_0 \qquad \Gamma \vdash \texttt{e}_3 : \tau_0}{\Gamma \vdash \texttt{if } \texttt{e}_1 \texttt{ | } \texttt{e}_2 \texttt{ | } \texttt{e}_3 : \tau_0}$$

Part 5

Types: Function Calls







Function Call Type Rule

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ \tau_2 \ \texttt{->} \ \tau_3 \qquad \Gamma \vdash \mathbf{e}_2 \ : \ \tau_2}{\Gamma \vdash \mathbf{e}_1 \ \texttt{(} \ \mathbf{e}_2 \ \texttt{)} \ : \ \tau_3}$$

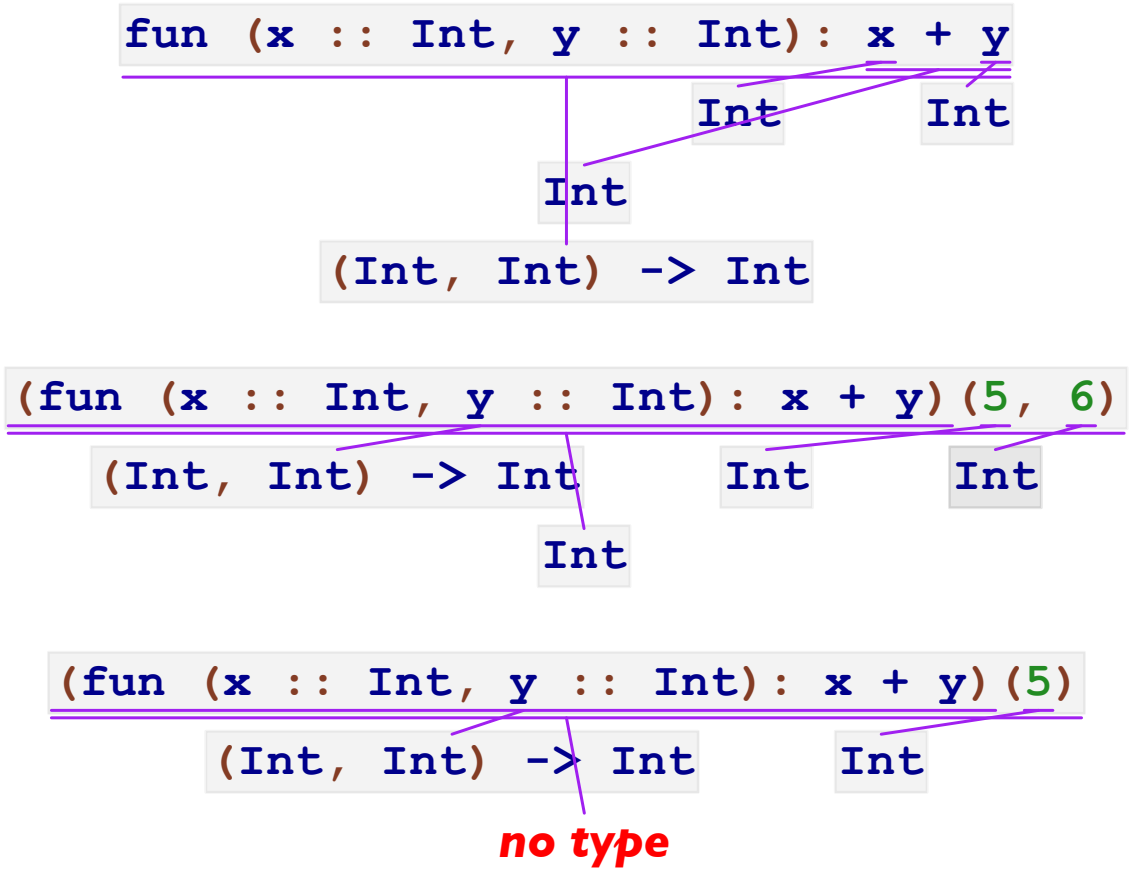
$$\frac{\varnothing \vdash \texttt{(fun (x :: Boolean): if x | 1 | 2)} \ : \ \texttt{Boolean} \rightarrow \texttt{Int} \qquad \varnothing \vdash \texttt{\#true} \ : \ \texttt{Boolean}}{\varnothing \vdash \texttt{(fun (x :: Boolean): if x | 1 | 2)(\#true)} \ : \ \texttt{Int}}$$

$$\frac{\varnothing \vdash \texttt{(fun (x :: Boolean): if x | 1 | 2)} \ : \ \texttt{Boolean} \rightarrow \texttt{Int} \qquad \varnothing \vdash \texttt{5} \ : \ \texttt{Int}}{\varnothing \vdash \texttt{(fun (x :: Boolean): if x | 1 | 2)(5)} \ : \ \textit{no type}}$$

$$\frac{\varnothing \vdash \texttt{7} \ : \ \texttt{Int} \qquad \varnothing \vdash \texttt{5} \ : \ \texttt{Int}}{\varnothing \vdash \texttt{7(5)} \ : \ \textit{no type}}$$

Part 6

Types: Multiple Arguments



Revised Function and Call Rules

$$\frac{\Gamma[\,\texttt{<Symbol>}_1 \leftarrow \tau_1 \ldots \texttt{<Symbol>}_n \leftarrow \tau_n\,] \vdash \texttt{e} : \tau_0}{\Gamma \vdash \texttt{fun(<Symbol>}_1 \texttt{ :: } \tau_1\texttt{, ..., <Symbol>}_n \texttt{ :: } \tau_n\texttt{): e : (}\tau_1\texttt{, ..., }\tau_n\texttt{) -> } \tau_0}$$

$$\frac{\Gamma \vdash \texttt{e}_0 \texttt{ : (}\tau_1\texttt{, ..., }\tau_n\texttt{) -> } \tau_0 \qquad \Gamma \vdash \texttt{e}_1 \texttt{ : } \tau_1 \qquad \ldots \qquad \Gamma \vdash \texttt{e}_n \texttt{ : } \tau_n}{\Gamma \vdash \texttt{e}_0\texttt{(e}_1\texttt{, ..., e}_n\texttt{) : } \tau_0}$$

Part 7

Typed Language

```
<Exp>   ::=  <Int>
           |  <Exp> + <Exp>
           |  <Exp> * <Exp>
           |  <Symbol>
           |  fun (<Symbol> :: <Type>): <Exp>
           |  <Exp>(<Exp>)
           |  (<Exp>)

<Type>  ::=  Int
           |  Boolean
           |  <Type> -> <Type>
           |  (<Type>)
```

Expressions

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
        r :: Exp)
| multE(l :: Exp,
        r :: Exp)
| funE(n :: Symbol,
       arg_type :: Type,
       body :: Exp)
| appE(fn :: Exp,
       arg :: Exp)
```

Types and Type Bindings

```
type Type
| intT()
| boolT()
| arrowT(arg :: Type,
          result :: Type)

type TypeBinding
| tbind(name :: Symbol,
         type :: Type)

type TypeEnv = Listof(TypeBinding)
```

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | ....
    | ....
```

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | intE(n): ....
    | ....
```

$\Gamma \vdash$ **<Num>** : `Int`

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | intE(n): intT()
    | ....
```

$\Gamma \vdash$ **<Num>** : **Int**

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | plusE(l, r):
        ....
    | ....
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \mathtt{Int} \qquad \Gamma \vdash \mathbf{e}_2 : \mathtt{Int}}{\Gamma \vdash \mathbf{e}_1 \ \mathbf{+} \ \mathbf{e}_2 : \mathtt{Int}}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | plusE(l, r):
        .... typecheck(l, tenv) ....
        .... typecheck(r, tenv) ....
    | ....
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \mathtt{Int} \qquad \Gamma \vdash \mathbf{e}_2 : \mathtt{Int}}{\Gamma \vdash \mathbf{e}_1 \mathbf{+} \mathbf{e}_2 : \mathtt{Int}}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | plusE(l, r):
        match typecheck(l, tenv)
        | intT():
            .... typecheck(r, tenv) ....
        | ~else: type_error(l, "Int")
    | ....
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \mathtt{Int} \qquad \Gamma \vdash \mathbf{e}_2 : \mathtt{Int}}{\Gamma \vdash \mathbf{e}_1 \ \mathbf{+} \ \mathbf{e}_2 : \mathtt{Int}}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | plusE(l, r):
        match typecheck(l, tenv)
        | intT():
            match typecheck(r, tenv)
            | intT(): intT()
            | ~else: type_error(r, "Int")
        | ~else: type_error(l, "Int")
    | ....
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \mathtt{Int} \qquad \Gamma \vdash \mathbf{e}_2 : \mathtt{Int}}{\Gamma \vdash \mathbf{e}_1 \ \mathtt{+} \ \mathbf{e}_2 : \mathtt{Int}}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | idE(name): ....
    | ....
```

$$[\ldots\ \texttt{<Symbol>} \leftarrow \tau\ \ldots] \vdash \texttt{<Symbol>} : \tau$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | idE(name): type_lookup(name, tenv)
    | ....
```

[... <Symbol>←τ ...] ⊢ <Symbol> : τ

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | funE(n, arg_type, body):
        ....
    | ....
```

$$\frac{\Gamma[\texttt{<Symbol>} \leftarrow \tau_1] \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{fun (<Symbol> :: } \tau_1 \texttt{) : e : } \tau_1 \texttt{ -> } \tau_2}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | funE(n, arg_type, body):
        .... typecheck(body, ....) ....
    | ....
```

$$\frac{\Gamma[\texttt{<Symbol>} \leftarrow \tau_1] \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{fun (<Symbol> :: } \tau_1 \texttt{) : e : } \tau_1 \texttt{ -> } \tau_2}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | funE(n, arg_type, body):
        .... typecheck(body, extend_env(tbind(n, arg_type),
                                        tenv))
        ....
    | ....
```

$$\frac{\Gamma[\texttt{<Symbol>} \leftarrow \tau_1] \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{fun (<Symbol> :: } \tau_1 \texttt{) : e : } \tau_1 \texttt{ -> } \tau_2}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | funE(n, arg_type, body):
        arrowT(arg_type,
               typecheck(body, extend_env(tbind(n, arg_type),
                                          tenv)))
    | ....
```

$$\frac{\Gamma[\texttt{<Symbol>} \leftarrow \tau_1] \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{fun (<Symbol> :: } \tau_1 \texttt{) : e : } \tau_1 \texttt{ -> } \tau_2}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | appE(fn, arg):
        ....
    | ....
```

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ \tau_2 \ \texttt{->} \ \tau_3 \qquad \Gamma \vdash \mathbf{e}_2 \ : \ \tau_2}{\Gamma \vdash \mathbf{e}_1 \ \texttt{(} \ \mathbf{e}_2 \ \texttt{)} \ : \ \tau_3}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | appE(fn, arg):
        .... typecheck(fn, tenv) ....
        .... typecheck(arg, tenv) ....
    | ....
```

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ \tau_2 \ \texttt{->} \ \tau_3 \qquad \Gamma \vdash \mathbf{e}_2 \ : \ \tau_2}{\Gamma \vdash \mathbf{e}_1 \ \texttt{(} \ \mathbf{e}_2 \ \texttt{)} \ : \ \tau_3}$$

Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | appE(fn, arg):
        match typecheck(fn, tenv)
        | arrowT(arg_type, result_type):
            .... typecheck(arg, tenv) ....
        | ~else: type_error(fn, "function")
    | ....
```

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ \tau_2 \ \texttt{->} \ \tau_3 \qquad \Gamma \vdash \mathbf{e}_2 \ : \ \tau_2}{\Gamma \vdash \mathbf{e}_1 \ \texttt{(} \ \mathbf{e}_2 \ \texttt{)} \ : \ \tau_3}$$

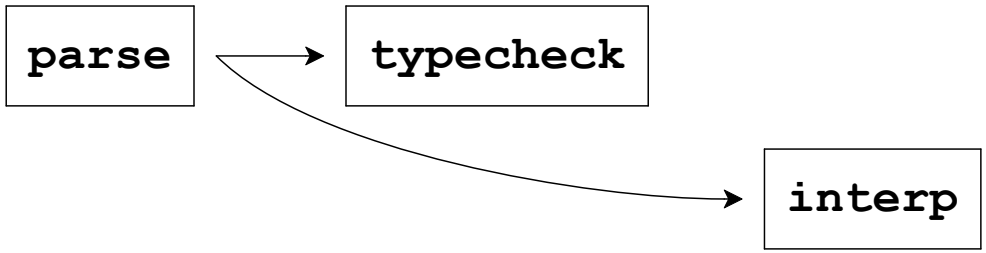
Type Checker

```
def typecheck :: (Exp, TypeEnv) -> Type:
  fun (a, tenv):
    match a
    | ....
    | appE(fn, arg):
        match typecheck(fn, tenv)
        | arrowT(arg_type, result_type):
            if arg_type == typecheck(arg, tenv)
            | result_type
            | type_error(arg,
                         to_string(arg_type))
        | ~else: type_error(fn, "function")
    | ....
```

$$\frac{\Gamma \vdash \mathbf{e}_1 \ : \ \tau_2 \ \texttt{->} \ \tau_3 \qquad \Gamma \vdash \mathbf{e}_2 \ : \ \tau_2}{\Gamma \vdash \mathbf{e}_1 \ \texttt{(} \ \mathbf{e}_2 \ \texttt{)} \ : \ \tau_3}$$

Part 8

`typecheck` and `interp`



Only call `interp` on an expression for which `typecheck` produces a type

`typecheck` ***never*** calls `interp`

`interp` ***never*** calls `typecheck`

`parse` never calls `typecheck` or `interp`...

Part 9

Pairs

```
let pair :: Int -> Int -> Boolean -> Int = (fun (x :: Int):
                                             fun (y :: Int):
                                               fun (s :: Boolean):
                                                 if s | x | y):
  let fst :: (Boolean -> Int) -> Int = (fun (p :: Boolean -> Int):
                                         p(#true)):
    let snd :: (Boolean -> Int) -> Int = (fun (p :: Boolean -> Int):
                                           p(#false)):
      snd(pair(1)(2))
```

Pairs

```
let pair :: Boolean -> Boolean -> Boolean -> Boolean = (fun (x :: Boolean):
                                                         fun (y :: Boolean):
                                                           fun (s :: Boolean):
                                                             if s | x | y):
  let fst :: (Boolean -> Boolean) -> Boolean = (fun (p :: Boolean -> Boolean):
                                                 p(#true)):
    let snd :: (Boolean -> Boolean) -> Boolean = (fun (p :: Boolean -> Boolean):
                                                   p(#false)):
      snd(pair(#true)(#false))
```

Pairs

```
let pair :: Int -> Boolean -> Boolean -> .... = (fun (x :: Int):
                                                  fun (y :: Boolean):
                                                    fun (s :: Boolean):
                                                      if s | x | y):
  let fst :: (Boolean -> ....) -> .... = (fun (p :: Boolean -> ....):
                                            p(#true)):
    let snd :: (Boolean -> ....) -> .... = (fun (p :: Boolean -> ....):
                                              p(#false)):
      snd(pair(1)(#false))
```

No possible type for ....

Language with Pairs

```
<Exp>   ::= <Int>
          | <Exp> + <Exp>
          | <Exp> * <Exp>
          | <Symbol>
          | fun (<Symbol> :: <TE>): <Exp>
          | <Exp>(<Exp>)
          | (<Exp>)
          | pair(<Exp>, <Exp>)                NEW
          | fst(<Exp>)                        NEW
          | snd(<Exp>)                        NEW

<Type>  ::= Int
          | Boolean
          | <Type> -> <Type>
          | (<Type>)
          | <Type> * <Type>                   NEW
```

$$\frac{\Gamma \vdash \mathbf{e}_1 : \tau_1 \qquad \Gamma \vdash \mathbf{e}_2 : \tau_2}{\Gamma \vdash \mathbf{pair}(\mathbf{e}_1, \mathbf{e}_2) : \tau_1 * \tau_2}$$

Language with Pairs

```
<Exp>   ::= <Int>
          | <Exp> + <Exp>
          | <Exp> * <Exp>
          | <Symbol>
          | fun (<Symbol> :: <TE>): <Exp>
          | <Exp>(<Exp>)
          | (<Exp>)
          | pair(<Exp>, <Exp>)                NEW
          | fst(<Exp>)                        NEW
          | snd(<Exp>)                        NEW

<Type>  ::= Int
          | Boolean
          | <Type> -> <Type>
          | (<Type>)
          | <Type> * <Type>                   NEW
```

$$\frac{\Gamma \vdash \texttt{e} : \tau_1 \texttt{ * } \tau_2}{\Gamma \vdash \texttt{fst(e)} : \tau_1}$$

Language with Pairs

```
<Exp>   ::= <Int>
          | <Exp> + <Exp>
          | <Exp> * <Exp>
          | <Symbol>
          | fun (<Symbol> :: <TE>): <Exp>
          | <Exp>(<Exp>)
          | (<Exp>)
          | pair(<Exp>, <Exp>)                NEW
          | fst(<Exp>)                        NEW
          | snd(<Exp>)                        NEW

<Type>  ::= Int
          | Boolean
          | <Type> -> <Type>
          | (<Type>)
          | <Type> * <Type>                   NEW
```

$$\frac{\Gamma \vdash \mathbf{e} : \tau_1 \ \mathbf{*} \ \tau_2}{\Gamma \vdash \mathbf{snd(e)} : \tau_2}$$