Contracts,
Type Systems,
and Type Checkers

CS401 — Spring 2019

contract noun

con·tract | \ˈkän-ˌtrakt \

Definition of *contract* (Entry 1 of 3)

1 **a** **:** a binding agreement between two or more persons or parties

especially **:** one legally enforceable

// If he breaks the *contract*, he'll be sued.

An agreement between multiple parties for mutual benefit.

contract noun

con·tract | \ˈkän-ˌtrakt \

Definition of *contract* (Entry 1 of 3)

1 a : a binding agreement between two or more persons or parties

especially : one legally enforceable

// If he breaks the *contract*, he'll be sued.

The agreement is enforced and violations are blamed on an offending party.

```
    }

    return e;
}


void insert(const T& ele, u64 index = 0)
{
    assert(length >= index);

    if (length+1 > buff_length)
    {
        // reallocate buffer
        T* oldbuff = buff;

        buff_length *= 2;
        buff = allocator.alloc(buff_length);

        // copy old data
        for (u64 i = 0; i < length; ++i)
        {
```

A reallocating array<T> class in C++

```
    }

    return e;
}
```

A reallocating array<T> class in C++

```
void insert(const T& ele, u64 index = 0)
{
    assert(length >= index);

    if (length+1 > buff_length)
    {
        // reallocate buffer
        T* oldbuff = buff;

        buff_length *= 2;
        buff = allocator.alloc(buff_length);

        // copy old data
        for (u64 i = 0; i < length; ++i)
        {
```

A reallocating array<T> class in C++

```
    }

    return e;
}


void insert(const T& ele, u64 index = 0)
{
    // Precondition:
    assert(length >= index);
    assert(length <= buff_length);

    // ... insert, possible reallocation ...


    // Postcondition:
    assert(length <= buff_length);
}
```

Meyer’s “Design by Contract”

Implemented in Meyer’s **Eiffel** programming language,
a typed, object-oriented language with contracts at the center.

“A contract carries mutual **obligations** and **benefits**.”

“Design by contract”. **Bertrand Meyer. 1986.**

Note that contracts are checked at **runtime**

(**Not** compile time!)

Applying “Design by Contract”

Precondition

Postcondition

```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

“Applying design by contract”. **Bertrand Meyer. 1992.**

Preconditions

To call `put_child`, calling code must satisfy its **obligations**

`put_child(n)`


```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

Postconditions

To return `put_child`, must ensure it provides **benefits**

`put_child(n)`


```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

If client **breaks** contract,

`put_child` is not obligated to provide benefits

`put_child(Void)`


```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

If client **breaks** contract,

`put_child` is not obligated to provide benefits

`put_child(Void)`

`not (Void /= Void)`

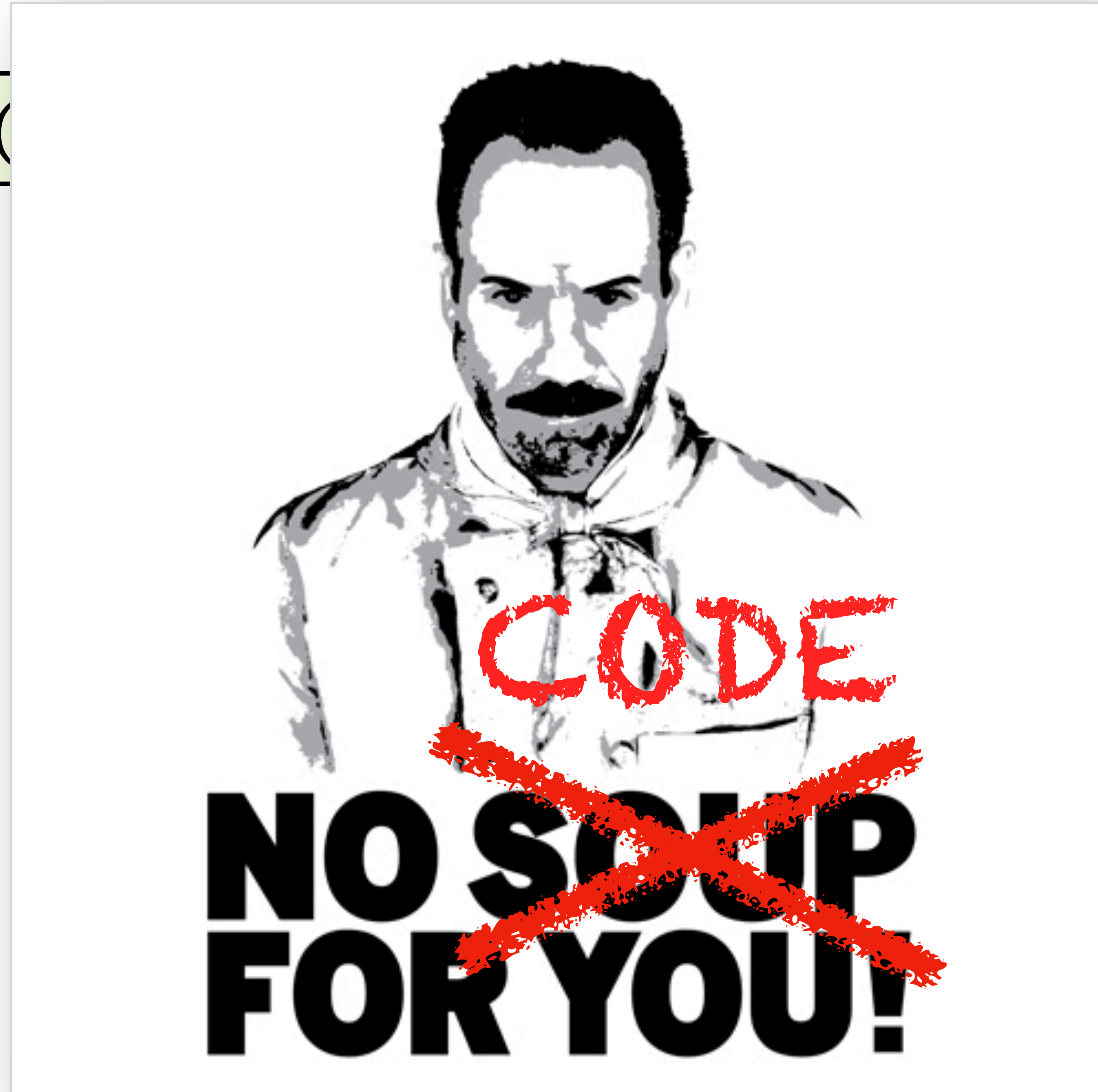

```
put_child (new: NODE) is
  require
    new /= Void
  do

    -- Insertion algorithm

  ensure
    new.parent = Current
    child_count = old child_count + 1
  end
```

If client **breaks** contract,

`put_child` is not obligated to provide benefits




```
put_child(
```

```
Void)
```

```
ensure
  new.parent = Current
  child_count = old child_count + 1
end
```

Contracts are a ***linguistic mechanism***

implemented as a *built-in feature* of the language, using *source-to-source translation*, or *using a macro system*.

factorial in Java (e.g., using jContract)

```java
/**
 *  @pre n >= 0
 *  @post return >= 1
 */
public static int fact(int n) {
    if (n <= 1) return 1;
    else return n * fact(n-1);
}
```

factorial in Java (e.g., using jContract)

```
public static int fact(int n) {
    assert n >= 0;
    if (n <= 1) {
        assert 1 >= 1;
        return 1;
    } else {
        int rv = n * fact(n-1);
        assert rv >= 1;
        return rv;
    }
}
```

The contract bakes dynamic checks into the source code, executed at every evaluation of fact(n)!

“Contracts for higher-order functions”. **Findler, Felleisen. 2002.**

Higher-order contract systems track program labels alongside contracts to ***properly assign blame*** when failure occurs.

"Correct blame for contracts". **Dimoulas. 2011.**

“I take in a positive and produce a positive.”

```
(define/contract (fib x)
  (-> positive? positive?)
  (cond
    [(= x 0) 1]
    [(= x 1) 1]
    [else (+ (fib (- x 1)) (fib (- x 2)))]))
```

```
Welcome to DrRacket, version 7.2 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (fib 2)
2
>
```

When I mess up

```
(define/contract (fib x)
  (-> positive? positive?)
  (cond
    [(= x 0) 1]
    [(= x 1) 1]
    [else (+ (fib (- x 1)) (fib (- x 2)))]))
```

```
> (fib -2)
fib: contract violation
expected: positive?
given: -2
in: the 1st argument of
    (-> positive? positive?)
contract from: (function fib)
blaming: anonymous-module
 (assuming the contract is correct)
at: unsaved-editor:3.18
>
```

When **I** mess up

```
(define/contract (fib x)
  (-> positive? positive?)
  (cond
    [(= x 0) 1]
    [(= x 1) 1]
    [else (+ (fib (- x 1)) (fib (- x 2)))]))
```

```
> (fib -2)
fib: contract violation
  expected: positive?
  given: -2
  in: the 1st argument of
      (-> positive? positive?)
  contract from: (function fib)
  blaming: anonymous-module
   (assuming the contract is correct)
  at: unsaved-editor:3.18
>
```

Racket blames **me**
(anonymous-module)

When **fib** messes up

```
(define/contract (fib x)
  (-> positive? positive?)
  (cond
    [(= x 0) -200]
    [(= x 1) 1]
    [else (+ (fib (- x 1)) (fib (- x 2)))]))
```

```
Welcome to DrRacket, version 7.2 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (fib 20)
fib: broke its own contract
  promised: positive?
  produced: -829435
  in: the range of
      (-> positive? positive?)
  contract from: (function fib)
  blaming: (function fib)
   (assuming the contract is correct)
  at: unsaved-editor:3.18
```

Racket blames **fib**

Earlier...

Note that contracts are checked at **runtime**

(**Not** compile time!)

But sometimes we want to know our program won’t break **before** it runs!

Type Systems

A **type system** assigns each source fragment with a given **type**: a specification of how it will behave

Type systems include **rules**, or **judgements** that tells us how we compositionally build types for larger fragments from smaller fragments

The **goal** of a type system is to **rule out** programs that would exhibit run time type errors!

A type system for STLC

(Simply-Typed Lambda Calculus)

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | x
    | n

prim ::= + | * | …
```

Term Syntax

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | x
    | n

prim ::= + | * | …
```

Type Syntax

```
t ::= nat
    | bool
    | t -> t
```

Term Syntax

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | x
    | n

prim ::= + | * | …
```

Type Syntax

```
t ::= nat
    | bool
    | t -> t
```

Function Types

Term Syntax

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | x
    | n

prim ::= + | * | …
```

Type Syntax

```
t ::= nat
    | bool
    | t -> t
```

Examples…

```
(int -> int) -> int

bool -> int

bool -> (int -> bool)
```

A type system for STLC

Type rules are written in natural-deduction style

(Like our big-step operational semantics.)

Assumptions above the line (No assumptions here.)

$$\frac{}{n : \textbf{num}} \quad \textbf{Const}$$

Conclusions below the line

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | x
    | n

prim ::= + | * | …
```

A type system for STLC

Type rules are written in natural-deduction style

(Like our big-step operational semantics.)

Assumptions above the line

(No assumptions here.)

$$\frac{}{n : \mathbf{num}} \textbf{ Const}$$

Conclusions below the line

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | x
    | n
```

“We may conclude any number n has type **num**”

```
prim ::= + | * | …
```

Variable Lookup

We assume a **typing environment** which maps variables to their types

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t} \quad \textbf{Var}$$

If x maps to type t in Γ, we may conclude that x has type t under the type environment Γ

Const revisited…

"We may conclude any constant n is of type **num** under **any** typing environment."

$$\frac{}{\Gamma \vdash n : \textbf{num}} \; \textbf{Const}$$

Functions…

If you conclude that e has type t' with Gamma **plus** assuming x has type t,…

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

Then you can conclude that the entire lambda has type `t -> t'`

Functions…

If you conclude that e has type t’ with Gamma **plus** assuming x has type t,…

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

Then you can conclude that the entire lambda has type `t -> t’`

Note

Variables (x) must be **tagged** with a type (e.g., by programmer)

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'} \quad \textbf{Lam}$$

$$\frac{}{\texttt{(lambda (x : num) 1)}}$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

Start with the empty environment (since this term is closed)

$$\frac{}{\Gamma = \{\} \vdash \texttt{(lambda (x : num) 1)} : ? \to ?}$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

$$\Gamma = \{\} \vdash \frac{}{\texttt{(lambda (x : num) 1)}} : t \to t'$$

We **suppose** there are two types, t and t',
which will make the derivation work.

Because x is tagged, it must be **num**

$$\frac{\{x \mapsto \textbf{num}\} \vdash 1 : t'}{\Gamma = \{\} \vdash \texttt{(lambda (x : num) 1)} : \textbf{num} \rightarrow t'}$$

We **suppose** there are two types, t and t',
which will make the derivation work.

The **Const** rule allows us to conclude 1 : **num**

$$\frac{\overline{\{x \mapsto \textbf{num}\} \vdash 1 : t'}}{\Gamma = \{\} \vdash \texttt{(lambda (x : num) 1)} : \textbf{num} \to t'}$$

We **suppose** there are two types, t and t’, which will make the derivation work.

So t' = **num**

$$\frac{\{x \mapsto \textbf{num}\} \vdash 1 : \textbf{num}}{\Gamma = \{\} \vdash \texttt{(lambda (x : num) 1)} : \textbf{num} \rightarrow \textbf{num}}$$

Function Application

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

Function Application

If (under Gamma), e has type `t -> t'`

And e' (its argument) has type t

$$\frac{\Gamma \vdash e : t \to t' \qquad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

Then the application of e to e' results in a `t'`

Our type system so far…

$$\frac{}{\Gamma \vdash n : \textbf{num}}\ \textbf{Const} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x : t}\ \textbf{Var}$$

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

$$\frac{\Gamma[x \mapsto t] \vdash e : t'}{\Gamma \vdash (\lambda\,(x : t)\ e) : t \to t'}\ \textbf{Lam}$$

Almost everything! Just need builtin functions

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | x
    | n

prim ::= + | * | …
```

Trick! Just **assume** they're part of Γ!

$$\Gamma_\iota = \{ + : \textbf{num} \rightarrow \textbf{num} \rightarrow \textbf{num}, \ldots \}$$

Almost everything! Just need builtin functions

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | x
    | n

prim ::= + | * | …
```

Then write in curried form…

Trick! Just **assume** they're part of Γ!

$$\Gamma_\iota = \{ + : \mathbf{num} \rightarrow \mathbf{num} \rightarrow \mathbf{num}, \ldots \}$$

Practice Derivations

Write derivations of the following expressions…

```
((λ (x : int) x) 1)
```

$$\frac{}{\Gamma \vdash n : \textbf{num}}\ \textbf{Const} \qquad \frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t}\ \textbf{Var}$$

$$\frac{\Gamma \vdash e : t \rightarrow t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda\, (x : t)\ e) : t \rightarrow t'}\ \textbf{Lam}$$

```
((λ (f : num -> num) (f 1)) (λ (x : num) x))
```

$$\frac{}{\Gamma \vdash n : \textbf{num}}\ \textbf{Const} \qquad \frac{x \mapsto t \in \Gamma}{\Gamma \vdash x : t}\ \textbf{Var}$$

$$\frac{\Gamma \vdash e : t \to t' \quad \Gamma \vdash e' : t}{\Gamma \vdash (e\ e') : t'}\ \textbf{App}$$

$$\frac{\Gamma, \{x \mapsto t\} \vdash e : t'}{\Gamma \vdash (\lambda\, (x : t)\ e) : t \to t'}\ \textbf{Lam}$$

Typability in STLC

Not all terms can be given types…

```
(λ (f : num -> num) (f f))
```

It is impossible to write a derivation for the above term!

f is `num->num` but would **need** to be `num`!

Typability

Not all terms can be given types…

```
((λ (f) (f f))
 (λ (f) (f f)))
```

It is **impossible** to write a derivation for Ω!

Consider what would happen if `f` were:

- `num -> num`
- `(num -> num) -> num`

Always just out of reach…

```
(λ (f : num -> num -> num) (((f 2) 3) 4))

((λ (f : num -> num) f) (λ (x:num) (λ (x:num) x)))
```

Type **Checking**

Type **checking:** verifying the derivation of a **fully-typed** term

```
((λ (x:num) x:num) : num -> num)
```

Notice that each subterm is assigned a "full" type

```
((λ (x:num) x:num) : num -> num)
```

Type checking tells us which rules we **must** apply **if there is** to be a derivation

Type **Inference**

Allows us to leave some **placeholder** variables that will be "filled in later"

```
((λ (x:t) x:t') : num -> num)
```

The `num->num` type then **forces** `t = num` and `t' = num`

Type **Inference**

Type inference can **fail**, too…

```
(λ (x) (λ (y:num->num) ((+ (x y)) x)))
```

No **possible** type for x! Used as fn and arg to +

Extending STLC…

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | x
    | n

prim ::= + | * | …
```

Let's add **if, and, or**

Extending STLC…

```
e ::= (lambda (x) e)
    | (e e)
    | ((prim e) e)
    | (if e e e)
    | (and e e)
    | (or e e)
    | x
    | n | #t | #f

prim ::= + | * | …
```

Now we need typing rules for if!

If needs guard to be a boolean…

Shouldn't be valid for guard to be, e.g., `(+ 1 2)`

```
(if guard
    t
    f)
```

If needs guard to be a boolean…

Shouldn't be valid for guard to be, e.g., `(+ 1 2)`

```
(if guard
    t
    f)
```

$$\frac{\Gamma \vdash e_g : \textbf{bool} \quad \Gamma \vdash e_t : t \quad \Gamma \vdash e_f : t}{\Gamma \vdash (\textbf{if}\ e_g\ e_t\ e_f) : t}\ \textbf{If}$$

If needs guard to be a boolean…

Shouldn't be valid for guard to be, e.g., `(+ 1 2)`

```
(if guard
    t
    f)
```

e_t/e_f must be same type!

$$\frac{\Gamma \vdash e_g : \textbf{bool} \quad \Gamma \vdash \boxed{e_t : t} \quad \Gamma \vdash \boxed{e_f : t}}{\Gamma \vdash (\textbf{if}\ e_g\ e_t\ e_f) : t}\ \textbf{If}$$

Exercise

Can you come up with the type rules for and/or?

$$(\texttt{and}\ e_1\ e_2)$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash (\textbf{and}\ e_1\ e_2) : \textbf{bool}}\ \textbf{And}$$

Completeness of STLC

- **Incomplete**: Reasonable functions we can't write in STLC
  - E.g., any program using recursion
- Several useful **extensions** to STLC
  - **Fix operator** to allow typing recursive functions
  - **Algebraic data types** to type structures
  - **Recursive types** to allow typing recursive structures
    - `tree = Leaf (int) | Node(int,tree,tree)`

Typing the Y Combinator

$$\frac{\Gamma \vdash f : t \to t}{\Gamma \vdash (Y f) : t}\ \textbf{Y}$$

Typing the Y Combinator

Think of how this would look for **fib**

$$\frac{\Gamma \vdash f : t \to t}{\Gamma \vdash (Y f) : t} \ \textbf{Y}$$

What would `t` be here?

```
(let ([fib
 (Y (λ (f) (λ (x)
             (if (= x 0)
                 1
                 (* x (fib (- x 1))))))))]))
```

Typing the Y Combinator

Think of how this would look for **fib**

```
(int -> int) -> (int -> int)
```

$$\frac{\Gamma \vdash f : t \rightarrow t}{\Gamma \vdash (Y f) : t} \ \mathbf{Y}$$

```
int -> int
```

```
(let ([fib
 (Y (λ (f) (λ (x)
              (if (= x 0)
                  1
                  (* x (fib (- x 1))))))))]))
```

Error States

A program steps to an **error state** if its evaluation reaches a point where the program has not produced a value, and yet cannot make progress

```
((+ 1) (λ (x) x))
```

Gets “stuck” because + can’t operate on λ

Error States

A program steps to an **error state** if its evaluation reaches a point where the program has not produced a value, and yet cannot make progress

```
((+ 1) (λ (x) x))
```

Gets “stuck” because + can’t operate on λ

(Note that this term is **not typable** for us!)

Soundness

A type system is **sound** if no typable program will ever evaluate to an error state

"Well typed programs cannot go wrong."
— Milner

(You can **trust** the type checker!)

Proving Type Soundness

Theorem: if e has some type derivation, then it will evaluate to a value.

Relies on two lemmas

Progress	Preservation
If e typable, then it is either a value or can be further reduced	If e has type `t`, any reduction will result in a term of type `t`

Progress

If e typable, then it is either a value or can be further reduced

Assume we proved this…

Preservation

If e has type `t`, any reduction will result in a term of type `t`

Assume we proved this…

Theorem: if e has some type derivation, then it will evaluate to a value.

Proof Sketch: If we have that `e : t`, and that e steps to `e'`, then `e' : t` as well (by preservation, multiple times). If `e'` is a value, then our proof is done (we have stepped to a value of type `t`). If it is not a value, progress tells us we can make a step.