Implementing an Interpreter

let x = 3 in
x + x

Let ("x",
Num 3,
Binop(Plus, Var "x", Var "x"))

Num 6

Parsing

Evaluation

Pretty Printing

6
Implementing an Interpreter

```
let x = 3 in
x + x
```

Type Checking

Let ("x",
Num 3,
Binop(Plus, Var "x", Var "x"))

Parsing

Evaluation

Num 6

Pretty Printing

6

3
type t = IntT | BoolT | ArrT of t * t

type x = string (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
  Const of c
  | Op of e * o * e
  | Var of x
  | If of e * e * e
  | Fun of x * typ * e
  | Call of e * e
  | Let of x * e * e
type t = IntT | BoolT | ArrT of t * t

type x = string (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan

type e =
    Const of c
    | Op of e * o * e
    | Var of x
    | If of e * e * e
    | Fun of x * typ * e
    | Call of e * e
    | Let of x * e * e

Notice that we require a type annotation here.

We'll see why this is required for our type checking algorithm later.
**Language (Abstract) Syntax (BNF Definition)**

```plaintext
t ::= int | bool | t -> t
b       -- ranges over booleans
n       -- ranges over integers
x        -- ranges over variable names
c ::= n | b
do ::= + | - | <
e ::= c
| e o e
| x
| if e then e else e
| λx:t.e
| e e
| let x = e in e
type t = IntT | BoolT | ArrT of t * t
type x = string (* variables *)
type c = Int of int | Bool of bool
type o = Plus | Minus | LessThan
type e =
    Const of c
    | Op of e * o * e
    | Var of x
    | If of e * e * e
    | Fun of x * typ * e
    | Call of e * e
    | Let of x * e * e
d =
```
When defining how evaluation worked, we used this notation:

\[
\begin{align*}
\text{e}_1 \rightarrow^* \lambda x. e & \quad \text{e}_2 \rightarrow^* v_2 \quad \text{e}[v_2/x] \rightarrow^* v \\
\text{e}_1 \text{e}_2 & \rightarrow^* v
\end{align*}
\]

In English:

“if \text{e}_1 \text{ evaluates to a function with argument } x \text{ and body } e \text{ and } \text{e}_2 \text{ evaluates to a value } v_2 \text{ and } e \text{ with } v_2 \text{ substituted for } x \text{ evaluates to } v \text{ then } \text{e}_1 \text{ applied to } \text{e}_2 \text{ evaluates to } v”

And we were also able to translate each rule into 1 case of a function in OCaml. Together all the rules formed the basis for an interpreter for the language.
This notation:

\[ e \rightarrow^* v \]

was read in English as "e evaluates to v."

It described a relation between two things – an expression e and a value v. (And e was related to v whenever e evaluated to v.)

Note also that we usually thought of e on the left as "given" and the v on the right as computed from e (according to the rules).
The typing judgement

This notation:

\[ G \vdash e : t \]

is read in English as "e has type t in context G." It is going to define how type checking works.

It describes a relation between three things – a type checking context G, an expression e, and a type t.

We are going to think of G and e as given, and we are going to compute t. The typing rules are going to tell us how.
What is the type checking context $G$?

Technically, I'm going to treat $G$ as if it were a (partial) function that maps variable names to types. Notation:

$G(x)$ -- look up $x$'s type in $G$
$G,x:t$ -- extend $G$ so that $x$ maps to $t$

When $G$ is empty, I'm just going to omit it. So I'll sometimes just write: $\vdash e : t$
Example Typing Contexts

Here's an example context:

\[ x: \text{int}, \ y: \text{bool}, \ z: \text{int} \]

Think of a context as a series of "assumptions" or "hypotheses"

Read it as the assumption that "x has type int, y has type bool and z has type int"

In the substitution model, if you assumed x has type int, that means that when you run the code, you had better actually wind up substituting an integer for x.
Typing Contexts and Free Variables

One more bit of intuition:

If an expression $e$ contains free variables $x$, $y$, and $z$ then we need to supply a context $G$ that contains types for at least $x$, $y$ and $z$. If we don't, we won't be able to type-check $e$. 
Type Checking Rules

Goal: Give rules that define the relation "\( G \vdash e : t \)".

To do that, we are going to give one rule for every sort of expression.

(We can turn each rule into a case of a recursive function that implements it pretty directly.)
Rule for constant booleans:

\[ G \vdash b : \text{bool} \]

English:

"boolean constants b \textit{always} have type bool, no matter what the context G is"
Typing Contexts and Free Variables

Rule for constant integers:

\[ G \vdash n : \text{int} \]

English:

“integer constants \( n \) *always* have type \( \text{int} \), no matter what the context \( G \) is"
Typing Contexts and Free Variables

\[ t ::= \text{int} | \text{bool} | t \rightarrow t \]
\[ c ::= n | b \]
\[ o ::= + | - | < \]
\[ e ::= \]
\[ c | e \circ e \]
\[ x \]
\[ \text{if } e \text{ then } e \text{ else } e \]
\[ \lambda x : t . e \]
\[ e \ e \]
\[ \text{let } x = e \text{ in } e \]

**Rule for operators:**

\[
\begin{align*}
G \vdash e_1 : t_1 & \quad G \vdash e_2 : t_2 & \quad \text{optype}(o) = (t_1, t_2, t_3) \\
G \vdash e_1 \circ e_2 : t_3
\end{align*}
\]

where

- \text{optype (+)} = (\text{int}, \text{int}, \text{int})
- \text{optype (-)} = (\text{int}, \text{int}, \text{int})
- \text{optype (<)} = (\text{int}, \text{int}, \text{bool})

**English:**

“\( e_1 \circ e_2 \) has type \( t_3 \), if \( e_1 \) has type \( t_1 \), \( e_2 \) has type \( t_2 \) and \( o \) is an operator that takes arguments of type \( t_1 \) and \( t_2 \) and returns a value of type \( t_3 \)"
Typing Contexts and Free Variables

Rule for variables:

Note: this is rule explains (part) of why the context needs to provide types for all of the free variables in an expression.
Typing Contexts and Free Variables

t ::= \text{int} | \text{bool} | t \rightarrow t

c ::= n | b

\text{e} ::= \begin{align*}
& c \\
& | e \circ e \\
& | x \\
& | \text{if } e \text{ then } e \text{ else } e \\
& | \lambda x: t. e \\
& | e e \\
& | \text{let } x = e \text{ in } e
\end{align*}

e ::= \begin{align*}
& \text{c} \\
& | e \circ e \\
& | x \\
& | \text{if } e \text{ then } e \text{ else } e \\
& | \lambda x: t. e \\
& | e e \\
& | \text{let } x = e \text{ in } e
\end{align*}

Rule for if:

\[
G \vdash e_1 : \text{bool} \quad G \vdash e_2 : t \quad G \vdash e_3 : t
\]

\[G \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t\]

English:

“if e1 has type bool
and e2 has type t
and e3 has (the same) type t
then e1 then e2 else e3 has type t "
Typing Contexts and Free Variables

### Types
- \( t ::= \text{int} | \text{bool} | t \rightarrow t \)

### Expressions
- \( c ::= n | b \)
- \( o ::= + | - | < \)
- \( e ::= c \)
- \( | e \ o \ e \)
- \( | x \)
- \( | \text{if} \ e \ \text{then} \ e \ \text{else} \ e \)
- \( | \lambda \ x : t . e \)
- \( | e \ e \)
- \( | \text{let} \ x = e \ \text{in} \ e \)

### Rule for functions:

\[
\begin{align*}
    \text{If } G,x:t \vdash e : t2 \\
    \text{then } G \vdash \lambda x:t.e : t -> t2
\end{align*}
\]

**English:**

"If the context \( G \) extended with \( x:t \) proves that \( e \) has type \( t2 \)
then \( \lambda x:t.e \) has type \( t -> t2 \) "

**Notice:**

Notice that to know how to extend the context \( G \), we need the typing annotation on the function argument.
Typing Contexts and Free Variables

\[ t ::= \text{int} | \text{bool} | t \to t \]

\[ c ::= n | b \]

\[ o ::= + | - | < \]

\[ e ::= \\
\quad c \\
\quad e \circ e \\
\quad x \\
\quad \text{if } e \text{ then } e \text{ else } e \\
\quad \lambda x:t.e \\
\quad e \; e \\
\quad \text{let } x = e \text{ in } e \]

**Rule for function call:**

\[
G \vdash e_1 : t_1 \to t_2 \quad G \vdash e_2 : t_1 \\
G \vdash e_1 \; e_2 : t_2
\]

**English:**

“if \( e_1 \) has type \( t_1 \to t_2 \) and \( e_2 \) has type \( t_1 \) then \( e_1 \; e_2 \) has type \( t_2 \)”
Typing Contexts and Free Variables

\[
t ::= \text{int} \mid \text{bool} \mid t \rightarrow t
\]
\[
c ::= n \mid b
\]
\[
o ::= + \mid - \mid <
\]
\[
e ::= c \\
| e \ o \ e \\
| x \\
| \text{if } e \text{ then } e \text{ else } e \\
| \lambda x : t . e \\
| e \ e \\
| \text{let } x = e \text{ in } e
\]

Rule for let:

\[
\frac{G \vdash e_1 : t_1 \quad G,x:t_1 \vdash e_2 : t_2}{G \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}
\]

English:

"if e_1 has type t_1 and G extended with x:t_1 proves e_2 has type t_2 then let x = e_1 in e_2 has type t_2 "
A typing derivation is a "proof" that an expression is well-typed in a particular context.

Such proofs consist of a tree of valid rules, with no obligations left unfulfilled at the top of the tree.

notice that “int” is associated with x in the context

\[
G, x : \text{int} \vdash x : \text{int} \quad G, x : \text{int} \vdash 2 : \text{int} \\
G, x : \text{int} \vdash x + 2 : \text{int} \\
G \vdash \lambda x : \text{int}. x + 2 : \text{int} \rightarrow \text{int}
\]
Key Properties

Good type systems are *sound*.

- ie, well-typed programs have "well-defined" evaluation
  - ie, our interpreter should not raise an exception part-way through because it doesn't know how to continue evaluation
  - colloquial phrase: “sound type systems do not go wrong”

Examples of OCaml expressions that go wrong:

- true + 3 (addition of booleans not defined)
- let (x,y) = 17 in ... (can’t extract fields of int)
- true (17) (can’t use a bool as if it is a function)

Sound type systems *accurately* predict run time behavior

- if e : int and e terminates then e evaluates to an integer
Soundness = Progress + Preservation

Proving soundness boils down to two theorems:

**Progress Theorem:**
If $\vdash e : t$ then either:
1. $e$ is a value, or
2. $e \rightarrow e'$

**Preservation Theorem:**
If $\vdash e : t$ and $e \rightarrow e'$ then $\vdash e' : t$

See COS 510 for proofs of these theorems.
But you have most of the necessary techniques:
Proof by induction on the structure of ...
... various inductive data types. :-)}
The typing rules also define an algorithm for
... type checking ...

If you view G and e as inputs,
the rules for “G ⊢ e : t” tell you how to compute t
Recall the OCaml Definition of Our Syntax

type t = IntT (* type int *)
   | BoolT (* type bool *)
   | ArrT of t * t (* type t -> t *)

type x = string (* variables *)

type c = Int of int | Bool of bool (* integer and boolean constants *)

type o = Plus | Minus | LessThan (* operators *)

type e = (* expressions *)
   Const of c
   | Op of e * o * e
   | Var of x
   | If of e * e * e
   | Fun of x * t * e (* t gives type of argument *)
   | Call of e * e
   | Let of x * e * e
Signature for Context Operations

(* abstract type of contexts *)

type ctx

(* empty context *)

val empty : ctx

(* update ctx x t: updates context ctx by binding variable x to type t *)

val update : ctx -> x -> t -> ctx

(* look ctx x: retrieves the type t associated with x in ctx
* raises NotFound if x does not appear in ctx *)

exception NotFound

val look : ctx -> x -> t
Auxiliary Functions

(* const c is the type of constant c *)
let const (c : c) : t =
  match c with
  | Int i -> IntT
  | Bool b -> BoolT

(* op o = (t1, t2, t3) when o has type t1 -> t2 -> t3 *)
let op (o : o) : t =
  match o with
  | Plus -> (IntT, IntT, IntT)
  | ... 

(* use err s to signal a type error with message s *)
exception TypeError of string
let err s = raise (TypeError s)
Simple Rules

(* type check expression e in ctx, producing t *)
let rec check (ctx : ctx) (e : e) : t =
  match e with
  | Const c -> const c
  | Op (e1, o, e2) ->
    let (t1, t2, t) = op o in (* op : t1 -> t2 -> t *)
    let t1' = check ctx e1 in
    let t2' = check ctx e2 in
    if (t1 = t1') && (t2 = t2') then
      t
    else
      err "bad argument to operator"

const(c) = t
  G ⊢ c : t

optype(o) = (t1, t2, t3)
  G ⊢ e1 : t1
  G ⊢ e2 : t2
  G ⊢ e1 o e2 : t3
let rec check (ctx : ctx) (e : e) : t =
  match e with
  | Var x ->
    begin
      try look ctx x with
        NotFoundException -> err ("free variable: " ^ x)
      end
  end
Function Typing

(* type check expression e in ctx, producing t *)

let rec check (ctx : ctx) (e : e) : t =
    match e with
    | Fun (x, t, e) ->
      check (update ctx x t) e

Notice that if we did not have the type t as a typing annotation we would not be able to make progress in our type checker at this point. We need to have a type for the variable x in our context in order to recursively check the expression e

\[ G, x : t \vdash e : t_2 \]
\[ G \vdash \lambda x : t. e : t \rightarrow t_2 \]
let rec check (ctx : ctx) (e : e) : t =
    match e with
    | Call (e1, e2) ->
      begin
        let t1 = check ctx e1 in
        match t1 with
        | ArrT (targ, tresult) ->
          let t2 = check ctx e2 in
          if targ = t2 then tresult
          else err "bad argument to function"
        | _,_ -> err "not a function in call position"
      end
    end

G ⊨ e1 : targ -> tresult
G ⊨ e2 : targ
G ⊨ e1 e2 : tresult
(* type check expression e in ctx, producing t *)

let rec check (ctx : ctx) (e : e) : t =
  match e with
  | If (e1, e2, e3) -> ...
  | Let (x, e1, e2) -> ...

Exercise: Other Rules
TYPE INFERENCE
Robin Milner

For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.

We will be studying Hindley-Milner type inference. Discovered by Hindley, rediscovered by Milner. Formalized by Damas. Broken several times when effects were added to ML.
The ML language and type system is designed to support a very strong form of type inference.

```plaintext
let rec map f l =
  match l with
  [ ] -> [ ]
| hd::tl -> f hd :: map f tl
```

It’s very convenient we don’t have to annotate \( f \) and \( l \) with their types, as is required by our type checking algorithm.
The ML language and type system is designed to support a very strong form of type inference.

```
let rec map f l =
  match l with
  [ ] -> [ ]
| hd::tl -> f hd :: map f tl
```

ML finds this type for map:

```
map : ('a -> 'b) -> 'a list -> 'b list
```
The ML language and type system is designed to support a very strong form of type inference.

```ml
let rec map f l =  
  match l with  
  [ ] -> [ ]  
  | hd::tl -> f hd :: map f tl
```

ML finds this type for map:

```
map : ('a -> 'b) -> 'a list -> 'b list
```

which is really an abbreviation for this type:

```
map : forall 'a,'b.('a -> 'b) -> 'a list -> 'b list
```
Language Design for Type Inference

We call this type the *principal type (scheme)* for map.

Any other ML-style type you can give map is an instance of this type, meaning we can obtain the other types via substitution of types for parameters from the principle type.

E.g.:

\[
\text{map} : (\texttt{('a -> 'b)} \rightarrow \texttt{'a list} \rightarrow \texttt{'b list})
\]

\[
(\texttt{bool -> int)} \rightarrow \texttt{bool list} \rightarrow \texttt{int list}
\]

\[
(\texttt{'a -> int)} \rightarrow \texttt{'a list} \rightarrow \texttt{int list}
\]

\[
(\texttt{'a -> 'a)} \rightarrow \texttt{'a list} \rightarrow \texttt{'a list}
\]
Principal types are great:

- the type inference engine can make a *best choice* for the type to give an expression
- the engine doesn't have to guess (and won't have to guess wrong)

The fact that principal types exist is surprisingly brittle. If you change ML's type system a little bit in either direction, it can fall apart.
Suppose we take out polymorphic types and need a type for id:

```ocaml
let id x = x
```

Then the compiler might guess that id has one (and only one) of these types:

```ocaml
id : bool -> bool
id : int -> int
```
Suppose we take out polymorphic types and need a type for `id`:

\[
\text{let id x = x}
\]

Then the compiler might guess that `id` has one (and only one) of these types:

\[
\text{id : bool -> bool}
\]

\[
\text{id : int -> int}
\]

But later on, one of the following code snippets won't type check:

\[
\text{id true}
\]

\[
\text{id 3}
\]

So whatever choice is made, a different one might have been better.
We showed that removing types from the language causes a failure of principal types.

Does adding more types always make type inference easier?
We showed that removing types from the language causes a failure of principle types.

Does adding more types always make type inference easier?

Nope!
OCaml has universal types on the outside ("prenex quantification"): 

\[
\text{forall 'a,'b. } (('a \to 'b) \to 'a \text{ list } \to 'b \text{ list})
\]

It does not have types like this:

\[
(\text{forall 'a.'a } \to \text{ int }) \to \text{ int } \to \text{ bool}
\]

argument type has its own polymorphic quantifier
Consider this program:

```ocaml
let f g = (g true, g 3)
```

notice that parameter g is used inside f as if:

- 1. its argument can have type bool, **AND**
- 2. its argument can have type int
Consider this program:

```haskell
let f g = (g true, g 3)
```

notice that parameter g is used inside f as if:

1. its argument can have type bool, \textit{AND}
2. its argument can have type int

Does the following type work?

```haskell
('a -> int) -> int * int
```
Consider this program:

```ocaml
let f g = (g true, g 3)
```

notice that parameter g is used inside f as if:

1. it’s argument can have type bool, **AND**
2. it’s argument can have type int

Does the following type work?

```ocaml
f: ('a -> int) -> int * int
```

**NO**: this says g’s argument can be any type ‘a (it could be int or bool)

Consider g is (fun x -> x + 2) : int -> int.
Unfortunately, \( f \ g \) goes wrong when g applied to true inside f.
Consider this program again:

```
let f g = (g true, g 3)
```

We might want to give it this type:

```
f : (forall a. a -> a) -> bool * int
```

Notice that the universal quantifier appears left of ->
System F is a lot like OCaml, except that it allows universal quantifiers in any position. It could type check \( f \).

\[
\text{let } f \ g = (g \text{ true}, \ g \text{ 3})
\]

\[
f : (\forall a.a\rightarrow a) \rightarrow \text{bool} \times \text{int}
\]

Unfortunately, type inference in System F is undecidable.
System F is a lot like OCaml, except that it allows universal quantifiers in any position. It could type check f.

\[
\text{let } f \ g = (g \text{ true}, \ g \text{ 3})
\]

\[
f : (\forall a. a \to a) \to \text{bool} \times \text{int}
\]

Unfortunately, type inference in System F is undecidable.

Developed in 1972 by logician Jean Yves-Girard who was interested in the consistency of a logic of 2\textsuperscript{nd}-order arithmetic.

Rediscovered as programming language by John Reynolds in 1974.
Language Design for Type Inference

Even seemingly small changes can effect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```
let f x = x + x
```
Even seemingly small changes can effect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```
let f x = x + x
```

\( f : \text{int} \rightarrow \text{int} \) ?

\( f : \text{float} \rightarrow \text{float} \) ?
Even seemingly small changes can affect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```
let f x = x + x
```

- \( f : \text{int} \rightarrow \text{int} \) ?
- \( f : \text{float} \rightarrow \text{float} \) ?
- \( f : 'a \rightarrow 'a \) ?
Language Design for Type Inference

Even seemingly small changes can affect type inference.

Suppose "+" operated on both floats and ints. What type for this?

```ocaml
let f x = x + x
```

- `f : int -> int` ?
- `f : float -> float` ?
- `f : 'a -> 'a` ?

No type in OCaml's type system works. In Haskell:

```haskell
f :: Num 'a => 'a -> 'a
```