# CS107: Functions

**Guannan Wei**
guannan.wei@tufts.edu

Sept 24, 2026

Fall 2026

Tufts University

# Recap

We have been building a growing language with more features and robust semantic analysis. Last time:

- Introduce type systems for semantic analysis
- Implementation of type checking/inference
- Interpretation and compilation with types

# Compilation With Types

Assembly code does not have types.

In assembly, how to "represent" values of these types?

- Boolean: 0 or 1, but we still use the full register.
- Unit: full register with uninitialized bits (for now).

# Compilation With Types

Compiling comparison and boolean operations:

```
val x = 1 == 4;
```

We could use jumps: one branch sets 0, the other sets 1.

# Compilation With Types

Compiling comparison and boolean operations:

```
val x = 1 == 4;
```

We could use jumps: one branch sets 0, the other sets 1.

but X86 offers us a shortcut:

```
    set<op> %al          # set %al if flags validate <op>
                         # like jump, there are: sete, setne, setl, etc.
    movbq %al, %rax      # transform the byte into the full register
```

Note: %al is a 8-bit register.

# Compilation With Types

We also have to modify our compilation for the `If` statements.

```
def trans(exp: Exp, sp: Int)(env: Env) = exp match {
  case If(cond, tBranch, eBranch) =>
    trans(cond, sp)(env)  // now reg at sp will contain 0 or 1
    transJumpIfTrue(sp)("if_then")
    // translate else branch, then branch, etc.
```

What code would `transJumpIfTrue` generate?

# Compilation With Types

We also have to modify our compilation for the `If` statements.

```
def trans(exp: Exp, sp: Int)(env: Env) = exp match {
  case If(cond, tBranch, eBranch) =>
    trans(cond, sp)(env)  // now reg at sp will contain 0 or 1
    transJumpIfTrue(sp)("if_then")
    // translate else branch, then branch, etc.
```

What code would `transJumpIfTrue` generate?

```
cmp ${regs(sp)}, $1    # INVALID syntax, only registers allowed.
je $label
```

# Compilation With Types

We also have to modify our compilation for the `If` statements.

```
def trans(exp: Exp, sp: Int)(env: Env) = exp match {
  case If(cond, tBranch, eBranch) =>
    trans(cond, sp)(env)  // now reg at sp will contain 0 or 1
    transJumpIfTrue(sp)("if_then")
    // translate else branch, then branch, etc.
```

What code would `transJumpIfTrue` generate?

```
cmp ${regs(sp)}, $1    # INVALID syntax, only registers allowed.
je $label
```

```
test ${regs(sp)}, ${regs(sp)}
jnz $label
```

Note: `test S, T` sets the flags accordingly to `S & T`: So if `sp` contains 1: `1 & 1` $\neq$ `0` so we jump (jnz). If `sp` contains 0: `0 & 0` $=$ `0`, then we don't jump.

# Growing the Language

What is still missing in our language?

# Growing the Language

What is still missing in our language?

```
def f(x: Int) = x + 3

def g() = 2

def h(x: Int, y: Boolean): Int = {
  val z = if (y) {
    x + 1
  } else {
    x - 1
  };
  z * x
}

def k(f: Int => Int): Int = f(0)
```

# Let's Add Functions - Syntax

```
<type>   ::= <ident> | <type> '=>' <type>  // '=>' is right associative
           | '('[<type>[','<type>]*]')' '=>' <type>
<atom>   ::= <number> | <bool> | <ident> | '()'
           | '('<simp>')'
<tight>  ::= <atom>['('[<simp>[','<simp>]*]')']* // application
           | '{'<exp>'}'
<uatom>  ::= [<op>]<tight>                  // Previously atom
<simp>   ::= ...                            // same as before
<exp>    ::= ...                            // same as before
<arg>    ::= <ident>':'<type>
<prog>   ::=                                // function definitions
   ['def'<ident>'('[<arg>[','<arg>]*]')'[':' <type>] '=' <simp>';']* <exp>
```

- Function type `=>` is right associative, i.e., `Int => Int => Int` is `Int => (Int => Int)`.
- but function application is left associative, i.e. `f(1)(3)` is `((f(1))(3))`.

# Let's Add Functions - AST

```
case class FunType(args: List[(String,Type)], rte: Type) extends Type
```

# Let's Add Functions - AST

```
case class FunType(args: List[(String,Type)], rte: Type) extends Type

case class Arg(name: String, atp: Type, pos: Position)
case class FunDef(name: String, args: List[Arg], rte: Type, fbody: Exp)
  extends Exp
```

# Let's Add Functions - AST

```
case class FunType(args: List[(String,Type)], rte: Type) extends Type

case class Arg(name: String, atp: Type, pos: Position)
case class FunDef(name: String, args: List[Arg], rte: Type, fbody: Exp)
  extends Exp

case class LetRec(funs: List[Exp], body: Exp) extends Exp
case class App(fun: Exp, args: List[Exp]) extends Exp
```

# Example

```
                    // LetRec(List(
def f(x: Int) = { //   FunDef("f", List(Arg("x", IntType)), UnknownType,
  x + 1           //     Prim("+", Ref("x"), Lit(1))
};                //   )),
f(1)              //   App(Ref("f"), List(Lit(1)))
                  // )
```

# Let's Add Functions - Interpreter

What does a function call mean?

# Let's Add Functions - Interpreter

What does a function call mean?

```
                    // LetRec(List(
def f(x: Int) = { //   FunDef("f", List(Arg("x", IntType)), UnknownType,
  x + 1           //     Prim("+", Ref("x"), Lit(1))
};                //   )),
f(1)              //   App(Ref("f"), List(Lit(1)))
                  // )
```

# Let's Add Functions - Interpreter

What does a function call mean?

```
                    // LetRec(List(
def f(x: Int) = { //   FunDef("f", List(Arg("x", IntType)), UnknownType,
  x + 1           //     Prim("+", Ref("x"), Lit(1))
};                //   )),
f(1)              //   App(Ref("f"), List(Lit(1)))
                  // )

def f(x: Int) = { g(x) + 1 };
def g(x: Int): Int = 2 * x;
f(1 + 1)
```

# Let's Add Functions - Interpreter

What does a function call mean?

```
                    // LetRec(List(
def f(x: Int) = { //   FunDef("f", List(Arg("x", IntType)), UnknownType,
  x + 1           //     Prim("+", Ref("x"), Lit(1))
};                //   )),
f(1)              //   App(Ref("f"), List(Lit(1)))
                  // )
```

```
def f(x: Int) = { g(x) + 1 };
def g(x: Int): Int = 2 * x;
f(1 + 1)
```

```
def f(x: Int): Int = { g(x) + 1 };
def g(x: Int): Int = if (x == 0) 0 else f(x-1);
f(1)
```

# Let's Add Functions - Interpreter

```
abstract class Val
case class Cst(x: Any) extends Val
case class Func(args: List[String], fbody: Exp, env: Env) extends Val
```

# Let's Add Functions - Interpreter

```
abstract class Val
case class Cst(x: Any) extends Val
case class Func(args: List[String], fbody: Exp, env: Env) extends Val

def eval(exp: Exp)(env: Env): Val = exp match
  case LetRec(funs, body) =>
    val funcs: List[(String, Val)] = funs.map {
      case f@FunDef(n, _, _, _) => (n, eval(f)(env))
    }
    eval(body)(env.withVals(funcs))
  case FunDef(name, args, rte, fbody) => ...
  case App(f, args) => ...
```

# Let's Add Functions - Interpreter

```
abstract class Val
case class Cst(x: Any) extends Val
case class Func(args: List[String], fbody: Exp, env: Env) extends Val

def eval(exp: Exp)(env: Env): Val = exp match
  case LetRec(funs, body) =>
    val funcs: List[(String, Val)] = funs.map {
      case f@FunDef(n, _, _, _) => (n, eval(f)(env))
    }
    eval(body)(env.withVals(funcs))
  case FunDef(name, args, rte, fbody) =>
    Func(args.map(arg => arg.name), fbody, env)
  case App(f, args) => ...
```

# Let's Add Functions - Interpreter

```
abstract class Val
case class Cst(x: Any) extends Val
case class Func(args: List[String], fbody: Exp, env: Env) extends Val

def eval(exp: Exp)(env: Env): Val = exp match
  case LetRec(funs, body) =>
    val funcs: List[(String, Val)] = funs.map {
      case f@FunDef(n, _, _, _) => (n, eval(f)(env))
    }
    eval(body)(env.withVals(funcs))
  case FunDef(name, args, rte, fbody) =>
    Func(args.map(arg => arg.name), fbody, env)
  case App(f, args) => ...
```

**What about recursive functions?**

# Let's Add Functions - Interpreter

```
abstract class Val
case class Cst(x: Any) extends Val
case class Func(args: List[String], fbody: Exp, env: Env) extends Val

def eval(exp: Exp)(env: Env): Val = exp match
  case LetRec(funs, body) =>
    val funcs: List[(String, Val)] = funs.map {
      case f@FunDef(n, _, _, _) => (n, eval(f)(env))
    }
    // reattached the whole environment to each function!
    funcs.foreach {
      case (_, f@Func(_, _, _)) => f.withVals(funcs)
    }
    eval(body)(env.withVals(funcs))
  case FunDef(name, args, rte, fbody) => ...
  case App(f, args) => ...
```

# Let's Add Functions - Interpreter

```
abstract class Val
case class Cst(x: Any) extends Val
case class Func(args: List[String], fbody: Exp, env: Env) extends Val

def eval(exp: Exp)(env: Env): Val = exp match
  case LetRec(funs, body) => ...
  case FunDef(name, args, rte, fbody) => ...
  case App(f, args) =>
    val eargs = args.map(arg => eval(arg)(env))
    val Func(fargs, fbody, fenv) = eval(f)(env)
    eval(fbody)(fenv.withVals(fargs.zip(eargs)))
```

# Let's Add Functions - Semantics

- Argument names must be distinct. Arguments behave like immutable variables.

# Let's Add Functions - Semantics

- Argument names must be distinct. Arguments behave like immutable variables.
- Functions can be recursive (even mutually recursive).

# Let's Add Functions - Semantics

- Argument names must be distinct. Arguments behave like immutable variables.
- Functions can be recursive (even mutually recursive).
- We don't allow overloading, i.e. a function cannot have the same name that another one even with different arguments.

# Let's Add Functions - Semantics

- Argument names must be distinct. Arguments behave like immutable variables.
- Functions can be recursive (even mutually recursive).
- We don't allow overloading, i.e. a function cannot have the same name that another one even with different arguments.
- Functions are first-class: functions can be stored in variables, used as parameters and returns from other functions.

# Let's Add Functions - Type Conformance

```
case class FunType(args: List[(String,Type)], rte: Type) extends Type
```

A function type is well-formed if all of its argument types and its return type are well-formed.

# Let's Add Functions - Type Conformance

```
case class FunType(args: List[(String,Type)], rte: Type) extends Type
```

A function type is well-formed if all of its argument types and its return type are well-formed.

A function type `tp` conforms to type `pt` if all of the following hold:

1) `pt` is a function type *or* UnknownType
2) `pt` has the same number of arguments as `tp`
3) the type of `pt` argument #n conforms to the type of `tp` argument #n (note the inversion)
4) the return type of `tp` conforms to the return type of `pt`

# Let's Add Functions - Type Conformance - Examples

Example:

- $(\texttt{Int}, \texttt{Boolean}) \Rightarrow \texttt{Int}$ conforms to **???** (result: $(\texttt{Int}, \texttt{Boolean}) \Rightarrow \texttt{Int}$)
- $\texttt{Int} \Rightarrow \texttt{Int}$ conforms to $\texttt{Int} \Rightarrow \texttt{Int}$
- $\texttt{Int} \Rightarrow \texttt{Int}$ does not conform to Boolean - rule #1
- **???** $\Rightarrow \texttt{Int}$ conforms to $\texttt{Int} \Rightarrow \texttt{Int}$ (result: $\texttt{Int} \Rightarrow \texttt{Int}$)
- $\texttt{Int} \Rightarrow \texttt{Boolean}$ does not conform to $\texttt{Int} \Rightarrow \texttt{Int}$ - rule #4
- **???** $\Rightarrow \texttt{Boolean}$ conforms to $\texttt{Int} \Rightarrow$ **???** (result: $\texttt{Int} \Rightarrow \texttt{Boolean}$)
- $\texttt{Int} \Rightarrow \texttt{Int}$ does not conform to **???** $\Rightarrow \texttt{Int}$ - rule #3

# Let's Add Functions - Type Checking

Now let's talk about inference rules!

# Let's Add Functions - Type Checking

Now let's talk about inference rules!

$$\frac{\Gamma, x_1 : T_1, \ldots, x_n : T_n \vdash fb : T}{\Gamma \vdash \texttt{FunDef}(f, x_1, \ldots, x_n, T, fb) : (T_1, \ldots, T_n) \Rightarrow T}\ \text{FUNDEF}$$

$$\frac{\begin{array}{c}\Gamma, f_1 : FT_1, \ldots, f_n : FT_n \vdash f_1 : FT_1 \\ \vdots \\ \Gamma, f_1 : FT_1, \ldots, f_n : FT_n \vdash f_n : FT_n \\ \Gamma, f_1 : FT_1, \ldots, f_n : FT_n \vdash b : T\end{array}}{\Gamma \vdash \texttt{LetRec}(f_1, \ldots, f_n, b) : T}\ \text{LETREC}$$

# Let's Add Functions - Type Checking

$$\frac{\begin{array}{l}\Gamma \vdash f : (T_1, \ldots, T_n) \Rightarrow T \\ \Gamma \vdash a_1 : T_1 \\ \quad \vdots \\ \Gamma \vdash a_n : T_n\end{array}}{\Gamma \vdash \texttt{App}(f, \texttt{List}(a_1, \ldots, a_n)) : T}\ \text{APP}$$

# Where Are We?

Today, we learned the following:

- Compilation with types
- Introduce functions
- Type checking functions
- Interpreting functions

Next time:

- Compiling functions
- Arrays