CS107: Functions and Arrays

Guannan Wei
guannan.wei@tufts.edu

February 3, 2026

Spring 2026

Tufts University

Recap

What did we learn last time?

- Type checking/inference
- Adding functions to our language

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))
    }
    funcs.foreach {
      case (_, f@Func(_, _, _)) => f.withVals(funcs)
    }
    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))
    }
    funcs.foreach {
      case (_, f@Func(_, _, _)) => f.withVals(funcs)
    }
    eval(body)(env.withVals(funcs))
  case FunDef(name, args, rte, fbody) =>
    Func(args.map(arg => arg.name), fbody, env)
  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).
- Function application is left associative, i.e. `f(1)(3)` will be parsed as `App(App("f", 1), 3)`.

Let's Add Functions - Semantics

- Argument names must be distinct. Arguments behave like immutable variables.
- Functions can be recursive (even mutually recursive).
- Function application is left associative, i.e. `f(1)(3)` will be parsed as `App(App("f", 1), 3)`.
- 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).
- Function application is left associative, i.e. `f(1)(3)` will be parsed as `App(App("f", 1), 3)`.
- We don't allow overloading, i.e. a function cannot have the same name that another one even with different arguments.
- We allow functions to 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!

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

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

Let's Add Functions - Type Checking

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

Compilation - Calling Conventions

One of the main concepts we have been using so far is **convention**.

Our compiler only generates code that uses registers `sp` and above and which puts the result in `sp`.

Thus, we can keep intermediate results and know which memory locations are available at any given point.

Compilation - Calling Conventions

How should we call a function from any point in the program without losing data?

Example:

```
def f(x: Int) = 1+x;

val y = f(1);
val z = f(2);
y + z
```

Compilation - Calling Conventions

Example:

```
def f(x: Int) = 1+x;

val y = f(1);
val z = f(2);
y + z
```

```
# main program
main:
  ...
  call f
  ...
  ...
  call f
  ...
  ...
# generated function
f:
  ...
  ...
```

Compilation - Calling Conventions

Example:

```
def f(x: Int) = 1+x;

val y = f(1);
val z = f(2);
y + z
```

```
# main program
main:
  ...
  call f
  ...
  ...
  call f
  ...
  ...
# generated function
f:
  mov $1, reg0    where is value of x?
  mov $2, reg1
  ...
  ...
```

Compilation - Calling Conventions

Example:

```
def f(x: Int) = 1+x;

val y = f(1);
val z = f(2);
y + z
```

```
# main program
main:
  ...
  call f
  ...
  ...
  call f
  ...
  ...
# generated function
f:
  mov $1, reg0    where is value of x?
  mov $2, reg1
  ...
  ...             where should we return?
```

Compilation - Calling Conventions

Example:

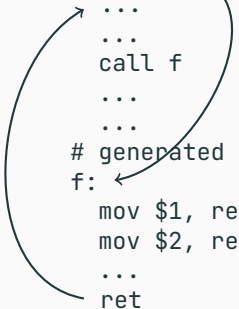
```
def f(x: Int) = 1+x;

val y = f(1);
val z = f(2);
y + z
```

```
# main program
main:
  ...
  call f
  ...
  ...
  call f
  ...
  ...
# generated function
f:
  mov $1, reg0
  mov $2, reg1
  ...
  ret
```

where the value is stored?

where is value of x?

where should we return?

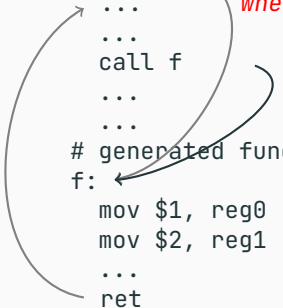


Compilation - Calling Conventions

Example:

```
def f(x: Int) = 1+x;

val y = f(1);
val z = f(2);
y + z
```

```
# main program
main:
  ...
  call f
  ...          where the value is stored?
  ...
  call f
  ...
  ...
# generated function
f:
  mov $1, reg0    where is value of x?
  mov $2, reg1
  ...
  ret             where should we return?
```



Compilation - Calling Convention

Calling convention: how functions receive parameters from their caller and how they return a result.

Mostly the System V AMD64 ABI (used on Linux, Mac OS, BSD, etc.) calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`

Compilation - Calling Convention

Calling convention: how functions receive parameters from their caller and how they return a result.

Mostly the System V AMD64 ABI (used on Linux, Mac OS, BSD, etc.) calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- Return: use **`call`** and **`ret`**

Compilation - Calling Convention

Calling convention: how functions receive parameters from their caller and how they return a result.

Mostly the System V AMD64 ABI (used on Linux, Mac OS, BSD, etc.) calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the instruction pointer on the stack, and jumps to the label

Compilation - Calling Convention

Calling convention: how functions receive parameters from their caller and how they return a result.

Mostly the System V AMD64 ABI (used on Linux, Mac OS, BSD, etc.) calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the instruction pointer on the stack, and jumps to the label
  - **`ret`** pops the return address from the stack and jumps to it

Compilation - Calling Convention

Calling convention: how functions receive parameters from their caller and how they return a result.

Mostly the System V AMD64 ABI (used on Linux, Mac OS, BSD, etc.) calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the instruction pointer on the stack, and jumps to the label
  - **`ret`** pops the return address from the stack and jumps to it
  - Corollary: we need to reset the stack before calling `ret`

Compilation - Calling Convention

Calling convention: how functions receive parameters from their caller and how they return a result.

Mostly the System V AMD64 ABI (used on Linux, Mac OS, BSD, etc.) calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the instruction pointer on the stack, and jumps to the label
  - **`ret`** pops the return address from the stack and jumps to it
  - Corollary: we need to reset the stack before calling `ret`
- Return value will be saved in `%rax`

Compilation - Calling Convention

Calling convention: how functions receive parameters from their caller and how they return a result.

Mostly the System V AMD64 ABI (used on Linux, Mac OS, BSD, etc.) calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the instruction pointer on the stack, and jumps to the label
  - **`ret`** pops the return address from the stack and jumps to it
  - Corollary: we need to reset the stack before calling `ret`
- Return value will be saved in `%rax`
- Before calling a function, all intermediate values will be saved on the stack

Compilation - Calling Convention

Calling convention: how functions receive parameters from their caller and how they return a result.

Mostly the System V AMD64 ABI (used on Linux, Mac OS, BSD, etc.) calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the instruction pointer on the stack, and jumps to the label
  - **`ret`** pops the return address from the stack and jumps to it
  - Corollary: we need to reset the stack before calling `ret`
- Return value will be saved in `%rax`
- Before calling a function, all intermediate values will be saved on the stack
- After the call, they need to be restored from the stack

Compilation - Calling Convention

Example:

```
def f(x: Int) = 1+x;

val y = f(1);
val z = f(2);
y + z
```

Your compiler in project 3 should generate code similar to this:

```
  # main program
  main:
    pushq %rbp
    movq %rsp, %rbp
    movq $1, %rdi
    call f
    movq %rax, %rdi
    movq $2, %rsi
    pushq %rdi
    movq %rsi, %rdi
    call f
    popq %rdi
    movq %rax, %rsi
    movq %rdi, %rdx
    movq %rsi, %rcx
    addq %rcx, %rdx
    movq %rdx, %rsi
    movq %rsi, %rdi
    movq %rdi, %rax
    movq %rbp, %rsp
    popq %rbp
    ret
# generated function
f:
    pushq %rbp
    movq %rsp, %rbp
    movq $1, %rsi
    movq %rdi, %rdx
    addq %rdx, %rsi
    movq %rsi, %rax
    movq %rbp, %rsp
    popq %rbp
    ret
```

Let's Add Arrays

We are going to use Scala syntax, but we are not (yet) going to handle objects.

The array will behave more like a C array; the length needs to be remembered.

```
val arr = new Array[Int](4 + 5);
```

Let's Add Arrays

We are going to use Scala syntax, but we are not (yet) going to handle objects.

The array will behave more like a C array; the length needs to be remembered.

```
val arr = new Array[Int](4 + 5);
```

```
val arr: Array[Int] = new Array[Int](4 + 5);
```

Let's Add Arrays - Syntax

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

Let's Add Arrays - Syntax

Scala array read syntax:

```
arr(1)
```

Wait a minute! Is this a function application?

Let's Add Arrays - Syntax

Scala array read syntax:

```
arr(1)
```

Wait a minute! Is this a function application?

Array write syntax:

```
arr(1) = 5
```

Let's Add Arrays - Syntax

Scala array read syntax:

```
arr(1)
```

Wait a minute! Is this a function application?

Array write syntax:

```
arr(1) = 5
```

The operations on arrays are primitive operations:

- `"block-get"`
- `"block-set"`

Let's Add Arrays - AST

```
case class Prim(op: String, args: List[Exp]) extends Exp
```

Let's Add Arrays - AST

```
case class Prim(op: String, args: List[Exp]) extends Exp
case class ArrayDec(size: Exp, tp: Type) extends Exp
```

Let's Add Arrays - AST

```
case class Prim(op: String, args: List[Exp]) extends Exp
case class ArrayDec(size: Exp, tp: Type) extends Exp
```

It is the parser's job to translate `arr(i) = v` to `Prim("block-set", List(arr, i, v))`, and the semantic analyzer's job to translate `arr(i)` to `Prim("block-get", List(arr, i))`.

Let's Add Arrays - Semantic Analysis

```
case class ArrayType(tp: Type) extends Type
```

How do we know if an array is well-formed?

Let's Add Arrays - Semantic Analysis

```
case class ArrayType(tp: Type) extends Type
```

How do we know if an array is well-formed?

If `tp` is well-formed!

Let's Add Arrays - Semantic Analysis

An array type `tp` conforms to type `pt` if:

- `pt` is an array type, and
- inner type `tp` conforms to `pt` inner type.

Let's Add Arrays – Semantic Analysis

An array type `tp` can be converted to a type `pt` if all of the following hold:

- `pt` is a function type with one argument
- the function argument's type conforms to Int type
- the inner type of `tp` conforms to the return type of `pt`

Let's Add Arrays – Semantic Analysis

An array type `tp` can be converted to a type `pt` if all of the following hold:

- `pt` is a function type with one argument
- the function argument's type conforms to Int type
- the inner type of `tp` conforms to the return type of `pt`

In other words, `Array`$[T]$ can be converted to `Int` $\Rightarrow T$ (i.e., how you would read an array).

Let's Add Arrays - Semantic Analysis

Inference rules:

Let's Add Arrays - Semantic Analysis

Inference rules:

$$\frac{\Gamma \vdash \mathit{size} : \texttt{Int}}{\Gamma \vdash \texttt{ArrayDec}(\mathit{size}, T) : \texttt{Array}[T]}\ \textsc{arraydec}$$

$$\frac{\Gamma \vdash \mathit{arr} : \texttt{Array}[T] \qquad \Gamma \vdash i : \texttt{Int}}{\Gamma \vdash \texttt{Prim}(\texttt{"block-get"}, \texttt{List}(\mathit{arr}, i)) : T}\ \textsc{arrayget}$$

$$\frac{\Gamma \vdash \mathit{arr} : \texttt{Array}[T] \qquad \Gamma \vdash i : \texttt{Int} \qquad \Gamma \vdash v : T}{\Gamma \vdash \texttt{Prim}(\texttt{"block-set"}, \texttt{List}(\mathit{arr}, i, v)) : \texttt{Unit}}\ \textsc{arrayset}$$

Let's Add Arrays - Interpreter

```
abstract class Val
case class Cst(x: Any) extends Val

def eval(exp: Exp)(env: Env): Val = exp match {
  case ArrayDec(size, _) =>
    val Cst(s: Int) = eval(size)(env) // Why is this safe?
    Cst(new Array[Any](s))
  case Prim("block-get", args) => ??? // left as exercise
  case Prim("block-set", args) => ??? // left as exercise
}
```

Let's Add Arrays - Compiler

Where do we want to store our arrays?

Let's Add Arrays - Compiler

Where do we want to store our arrays?

We will use the heap. The heap is permanent, i.e., not erased once a function call is over (unlike the stack and local variables).

Therefore the heap is used as persistent storage.

Let's Add Arrays - Compiler

When the (compiled) program launches, the OS maps a memory space for our stack. Thus, we can access the memory location stored in registers, e.g.:

```
mov $4, -8(%rsp)
```

To have access to the heap, we call `malloc` in `bootstrap.c` and pass the pointer to our main function as the first argument.

```
char* heap = malloc(SIZE);
entry_point(heap);
free(heap);
```

Let's Add Arrays - Compiler

Where is this pointer going to be saved?

Let's Add Arrays - Compiler

Where is this pointer going to be saved?

A global variable: `heap`. This address represents the first memory address that we are allowed to use.

So, how do we allocate an array (`ArrayDec`)?

Subsequent array allocations must not overlap!

Let's Add Arrays - Compiler

Initially:

```
  [byte0, byte1, byte2, byte3, ..., byteN]
  ^
  |
 heap
```

Let's Add Arrays - Compiler

Now we want to allocate an array `arr1` of size 4 (each element is 8 bytes):

```
[byte0, byte1, byte2, byte3, ..., byte32, byteN]
 ^                                 ^
 |                                 |
arr1                              heap
```

Let's Add Arrays - Compiler

We assume `%rax` contains the address of the array we want to access.

How to write to a memory location:

```
movq $1, (%rax)      # write one in the first element of the array
```

How to read from a memory location?

Let's Add Arrays - Compiler

We assume `%rax` contains the address of the array we want to access.

How to write to a memory location:

```
movq $1, (%rax)      # write one in the first element of the array
```

How to read from a memory location?

```
movq (%rax), %rax       # read the first element of the array
                        # and store it in %rax
```

Let's Add Arrays - Compiler

We assume `%rax` contains the address of the array we want to access.

How to write to a memory location:

```
movq $1, (%rax)      # write one in the first element of the array
```

How to read from a memory location?

```
movq (%rax), %rax       # read the first element of the array
                        # and store it in %rax
```

Shortcuts for arrays:

```
movq heap, %rax
movq $3, %rcx
movq (%rax, %rcx, 8), %rax  # read the 3rd (stored in %rcx)
                            # element of the array and store it in %rax
```

Read content `M[R[%rax] + R[%rcx]*8]` into `%rax`; see more in x64 Cheat Sheet.

Where Are We?

Today, we learned the following:

- Type checking functions
- Interpreting functions
- Calling conventions
- Adding arrays