

# CS107: More Functions and Arrays

---

**Guannan Wei**

guannan.wei@tufts.edu

September 29, 2026

Spring 2026

Tufts University

Introducing functions into our tiny language:

- Syntax for defining and calling functions
- Type checking and inference of functions

Today:

- Compiling functions to assembly
- Heap allocated arrays

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`.

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

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 ?, 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?
```

```
...
```

```
...
```

```
...
```

*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
```

```
mov ? reg0
```

*where the return value is stored?*

```
...
```

```
call f
```

```
...
```

```
...
```

```
# generated function
```

```
f: ←
```

```
mov $1, reg0
```

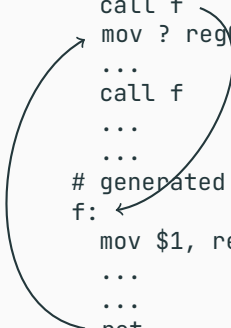
*where is value of x?*

```
...
```

```
...
```

```
ret
```

*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 $2, reg0
```

```
...
```

```
...
```

```
ret
```

*where the value is stored?*

*where is value of x?*

*where should we return?*

Calling convention:

- the caller and the function must agree on how to exchange values (arguments and returned values)
- specify what state survives the call, even if they were compiled separately.

Calling convention:

- the caller and the function must agree on how to exchange values (arguments and returned values)
- specify what state survives the call, even if they were compiled separately.

Different architectures/OS have different calling conventions:

- System V AMD64: x86-64 on Unix-like systems, including Linux, Mac OS, and BSD
- AAPCS64: Arm Linux, macOS, etc.
- Microsoft x64

## System V AMD64 ABI

Mostly the System V AMD64 ABI calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`; additional arguments are passed on the stack

## System V AMD64 ABI

Mostly the System V AMD64 ABI calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`; additional arguments are passed on the stack
- Return: use `call` and `ret`

## System V AMD64 ABI

Mostly the System V AMD64 ABI calling convention:

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

# System V AMD64 ABI

Mostly the System V AMD64 ABI calling convention:

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

Mostly the System V AMD64 ABI calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`; additional arguments are passed on the stack
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the next instruction pointer on the stack, and jumps to the label
  - **`ret`** pops the return address from the stack and jumps to it
  - Need to reset the stack (e.g. `%rsp` / `%rbp`) before calling `ret` (callee-saved registers)

# System V AMD64 ABI

Mostly the System V AMD64 ABI calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`; additional arguments are passed on the stack
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the next instruction pointer on the stack, and jumps to the label
  - **`ret`** pops the return address from the stack and jumps to it
  - Need to reset the stack (e.g. `%rsp` / `%rbp`) before calling `ret` (callee-saved registers)
- Return value will be saved in `%rax`

# System V AMD64 ABI

Mostly the System V AMD64 ABI calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`; additional arguments are passed on the stack
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the next instruction pointer on the stack, and jumps to the label
  - **`ret`** pops the return address from the stack and jumps to it
  - Need to reset the stack (e.g. `%rsp` / `%rbp`) before calling **`ret`** (callee-saved registers)
- Return value will be saved in `%rax`
- Before a call, the caller saves any values it still needs that the callee may overwrite (caller-saved registers)

# System V AMD64 ABI

Mostly the System V AMD64 ABI calling convention:

- Argument passing: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`; additional arguments are passed on the stack
- Return: use **`call`** and **`ret`**
  - **`call`** pushes the next instruction pointer on the stack, and jumps to the label
  - **`ret`** pops the return address from the stack and jumps to it
  - Need to reset the stack (e.g. `%rsp` / `%rbp`) before calling `ret` (callee-saved registers)
- Return value will be saved in `%rax`
- Before a call, the caller saves any values it still needs that the callee may overwrite (caller-saved registers)
- 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
```

```
...
```

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

## Let's Add Arrays - Syntax

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

What is missing? We don't have a way to access or change the array elements.

## 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
```

In our compiler, 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
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
```

The parser translates `arr(i) = v` to `Prim("block-set", List(arr, i, v))`, because solely looking at the form, we can decide it is an array update.

The semantic analyzer translates `arr(i)` to `Prim("block-get", List(arr, i))`, because the form could be either function application or array access, and we need *types* to disambiguate.

## Let's Add Arrays - Semantic Analysis

Semantic analysis:

- Well-formedness
- Type conformance
- Typing rules

## Let's Add Arrays - Semantic Analysis

Semantic analysis:

- Well-formedness
- Type conformance
- Typing rules

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

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

## Let's Add Arrays - Semantic Analysis

Semantic analysis:

- Well-formedness
- Type conformance
- Typing rules

```
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:

$$\frac{\Gamma \vdash \textit{size} : \textit{Int}}{\Gamma \vdash \text{ArrayDec}(\textit{size}, T) : \text{Array}[T]} \text{ARRAYDEC}$$

$$\frac{\Gamma \vdash \textit{arr} : \text{Array}[T] \quad \Gamma \vdash \textit{i} : \textit{Int}}{\Gamma \vdash \text{Prim}(\text{"block-get"}, \text{List}(\textit{arr}, \textit{i})) : T} \text{ARRAYGET}$$

$$\frac{\Gamma \vdash \textit{arr} : \text{Array}[T] \quad \Gamma \vdash \textit{i} : \textit{Int} \quad \Gamma \vdash \textit{v} : T}{\Gamma \vdash \text{Prim}(\text{"block-set"}, \text{List}(\textit{arr}, \textit{i}, \textit{v})) : \text{Unit}} \text{ARRAYSET}$$

## 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))           // Use meta-language's heap allocation
  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 a persistent storage, i.e., not erased once a function call is over (unlike the stack and local variables).

## 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 |

Compiling `ArrayDec` generates code to update the heap pointer to point to the next available 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?

## 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 ..., %rax        # load the base address of the array  
movq $3, %rcx         # load the index to read  
movq (%rax, %rcx, 8), %rax # read the 3rd  
                        # 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.

## A Taste Of MiniScala

A MiniScala function to compute  $x$  to the power of  $y$ :

```
def pow(x: Int, y: Int): Int =  
  if (y == 0) 1 // pow(x, 0) == 1  
  else if (even(y)) {  
    val t = pow(x, y / 2) // pow(x, 2z) = pow(pow(x, z), 2)  
    t * t  
  } else {  
    x * pow(x, y - 1) // pow(x, z + 1) = x * pow(x, z)  
  }
```

## A Taste Of MiniScala

Say "Hello":

```
val arr = new Array[Int](5);  
arr(0) = 'H'; arr(1) = 'e'; arr(2) = 'l'; arr(3) = 'l'; arr(4) = 'o';
```

```
var i = 0;  
while (i < 5) {  
    putchar(arr(i));  
    i = i + 1  
};
```

0