Section 8

CS160 Compilers

Junrui Liu

Plan

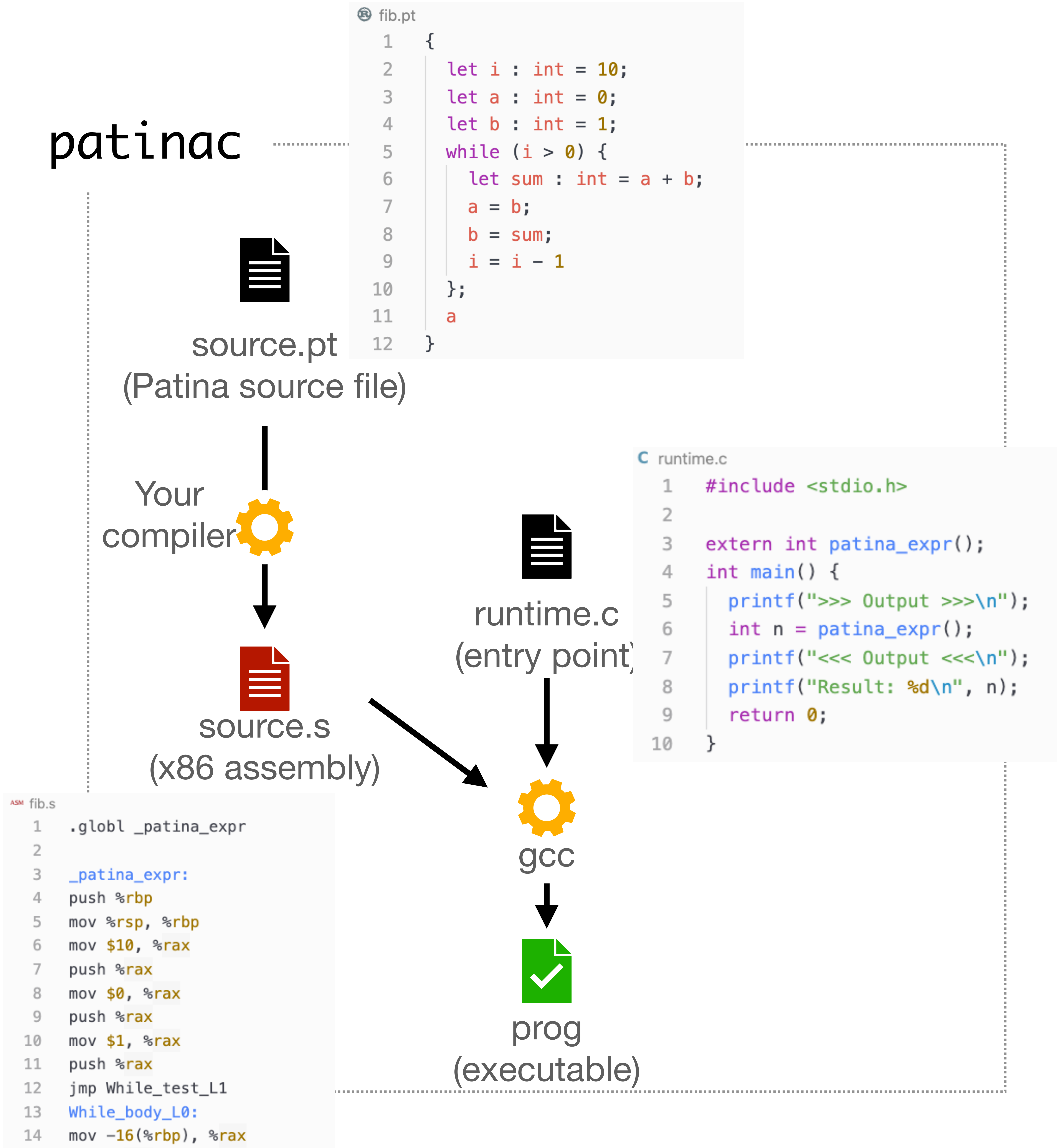
- Assignment 4 overview
- How to compile
  - Binary expressions
  - Let
  - Functions
  - Arrays

Assignment 4 Overview

- Three parts:
    1. (Required) Patina language - (functions + arrays)
    2. (Bonus) functions
    3. (Bonus) arrays

Assignment 4 Overview

- demo



patinac
fib.pt
{
let i : int = 10;
let a : int = 0;
let b : int = 1;
while (i > 0) {
let sum : int = a + b;
a = b;
b = sum;
i = i - 1
};
a
}
source.pt
(Patina source file)
Your compiler
source.s
(x86 assembly)
runtime.c
#include <stdio.h>
extern int patina_expr();
int main() {
printf(">>> Output >>>\n");
int n = patina_expr();
printf("<<< Output <<<\n");
printf("Result: %d\n", n);
return 0;
}
runtime.c
(entry point)
gcc
prog
(executable)
fib.s
.globl _patina_expr
_patina_expr:
push %rbp
mov %rsp, %rbp
mov $10, %rax
push %rax
mov $0, %rax
push %rax
mov $1, %rax
push %rax
jmp While_test_L1
While_body_L0:
mov -16(%rbp), %rax

Assignment 4 Overview

- Patches

x86-64
Operands

- Full x86-64 is extremely complex, but we'll only use a tiny subset
- Immediate values: `$1, $−3`
- General-purpose registers:
    - `%rax, %rbx, %rcx, %rdx,%rdi, %rsi`, and a few more
- Special-purpose register:
    - `%rsp`: stack pointer
    - `%rbp`: frame pointer/base pointer
    - `%rip`: instruction pointer
- Memory dereference: "`offset(reg)`"
    - E.g. `−8(%rbp)` dereferences the address "%rbp - 8 bytes"

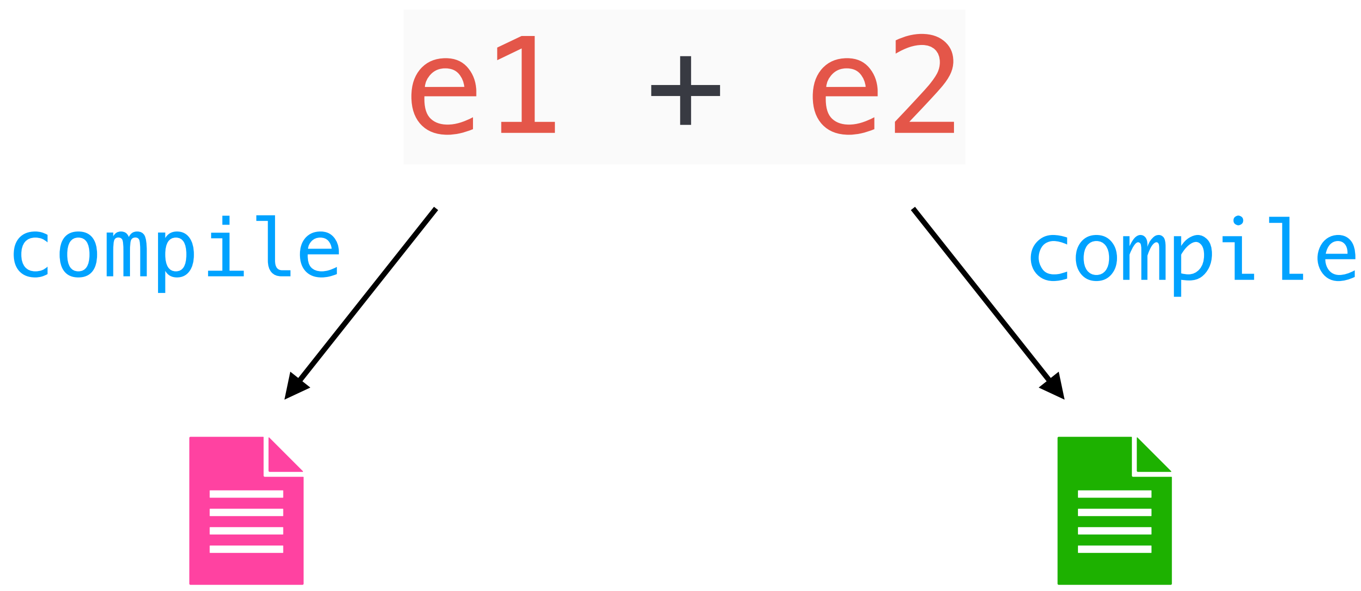
x86-64
Instructions

- Data movement: `mov src, dst`
- Two-operand arithmetic: `add, sub, imul, and, or`
  - Second operand is usually destination: “`OP X, Y`” means “Y := (Y op X)”
- Comparison “`cmp X, Y`” is like “`sub X, Y`”, but instead of storing (Y-X) in Y, it sets conditional flags (==, !=, <, <=, >, >=) that are true
  - E.g. “`mov $1, %rax;  cmp $-3, %rax`”
- Unconditional jump `jmp <label>`, conditional jump `j<cond> <label>`
- Stack starts at a high address, and grows toward **lower addresses**
  - e.g. “`push %rax`” is equivalent to “`mov %rax, -8(%rsp); sub $8, %rsp`”

Compiling `Binary`

- Function `compile`
  - takes a Patina expr `e`
  - outputs assembly instructions that compute `e`

- Invariant: instructions always store the value of `e` into `%rax`



Compiling `Let`

- Each function, when called, gets a new frame on the stack
- Frame = local variables + intermediate values
- How to layout the frame?
  - Approach A: Partition the frame into two regions, one for vars and the other for intermediate values
  - Approach B: No partitioning

Compiling `Let`

Approach A

- Two regions:
  - 1: all local variables in this function (fixed size)
  - 2: intermediate values
- When a function is called, allocate enough space for region 1 by incrementing stack pointer. But how large should the space be?
- Count the number of local variables in the function:

```
let cond: bool = 1 > 0;
if cond then {
  let x: int = 5
} else {
  let y: int = 6
}
```

- If you use this approach, `env.top` will point to the newest variable in the first region

Compiling `Let`

Approach B

- A single region that contains both local variables and intermediate values
- `env.top` will simply synchronize with the stack pointer
- Only requires a single pass, but can be harder to implement correctly.

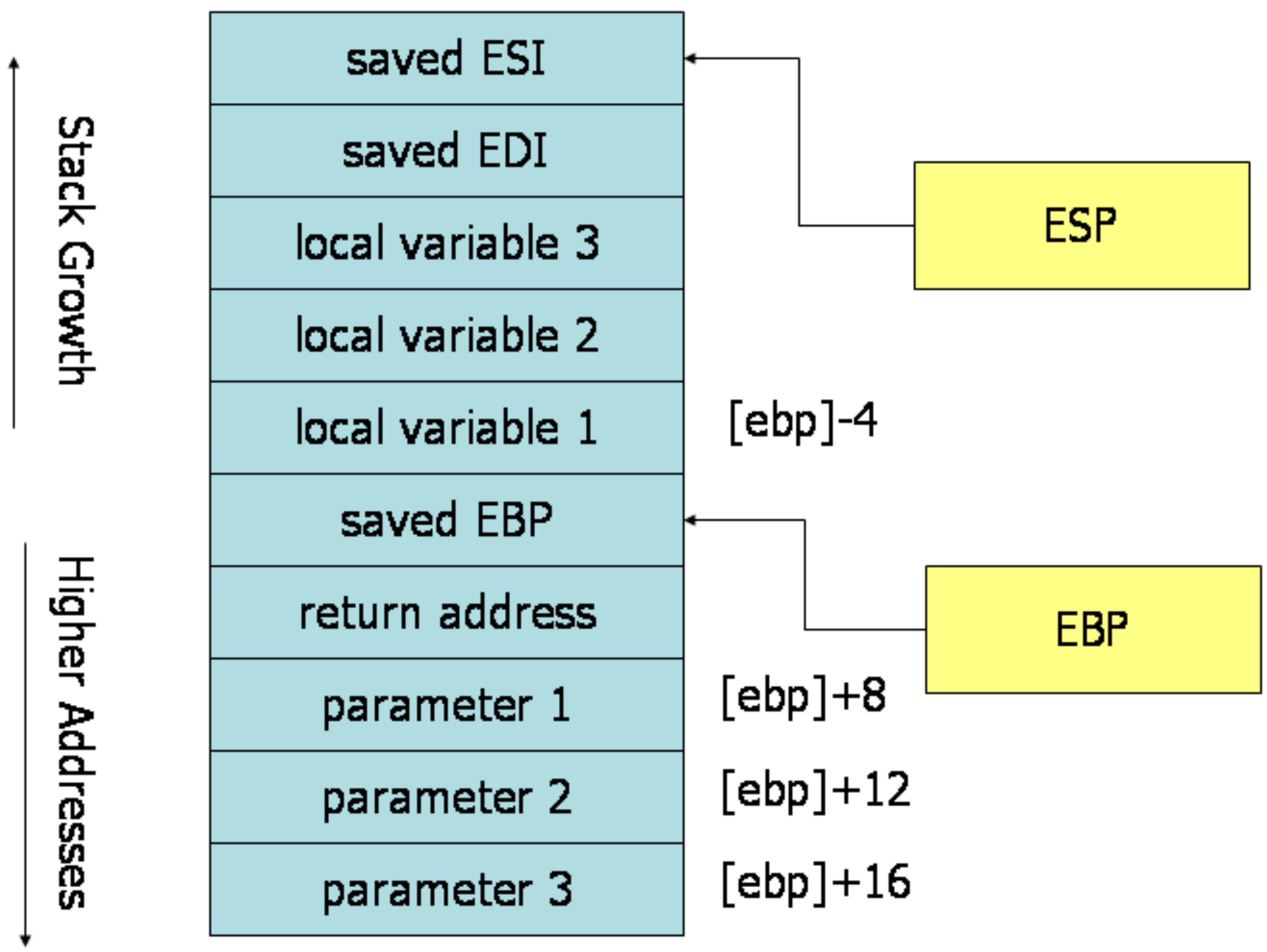
Compiling functions

- What's hard about functions?
    - Communicate where to return to using `call` and `ret`
    - Coordinate register-saving
    - Agree where the arguments should be located
- You can design your own calling convention (*)

System V 32-bit Calling Convention

https://wiki.osdev.org/System_V_ABI#i386

- Caller-saved registers (not preserved by function calls):
  - eax, ecx, edx
- Callee-saved registers (preserved by function calls):
  - ebx, esi, edi, ebp (and esp)
- Arguments are passed using the stack:



https://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html

Patina Built-in Functions

- Possible to implement in assembly, using `syscall` instruction

- Instead, they have been implemented in `runtime.c`

- Caveat: since we're interfacing functions external to Patina, need to follow their calling convention

  - System V 64-bit calling convention: https://wiki.osdev.org/System_V_ABI#x86-64

  - First six arguments are passed through registers (**`rdi, rsi, rdx,`** `rcx, r8, r9`), rest on stack

Compiling arrays

- Arrays are created via function calls to built-in function `alloc`,
- In runtime.c, `alloc` is implemented as a call to the C library function `calloc`, which returns a pointer to the allocated array
- Array reads and writes can be implemented using indirect addressing:
  - Suppose variable a is an array at memory -8(rbp)
  - Reading the second element of a can be implemented as
    - mov -8(%rbp), %rax
    - mov 16(%rax), %rax
- This treats arrays as raw pointers and offers no array bounds checking.
  - Bonus: implement bounds checking in runtime.c.

Questions?

Have a wonderful Thanksgiving :D