CSE 231: Printing using the Runtime

Ranjit Jhala *April 23, 2026*

Printing

Lets add support for **printing** expressions

A stepping stone to *user-defined functions* and *calls*

```
<expr> := ... | (print <expr>)
```

QUIZ: Semantics

Program

```
(print 42)
```

Expected Output

```
42
```

```
(let (x1 100)
(let (x2 200)
  (block
    (print x1)
    (print x2))))
```

Expected Output

```
100
200
200
```

Abstract Syntax

```
enum Expr { // ...
  Print(Box<Expr>),
}
```

Evaluation

```
fn eval(expr: &Expr, env: &mut Env) -> Value {
  match expr { // ...
    Expr::Print(e) => {

    }
  }
}
```

Strategy

1. Expose `snek_print` in the **runtime** (`start.rs`),
2. Compile `print` to **call** `snek_print`.

Similar to how we exposed `snek_error` for tag errors

Exposing `snek_print`

In the Runtime

```
#[export_name = "\x01snek_print"]
fn snek_print(val: i64) -> i64 { ... }
```

In the Assembly

```
global our_code_starts_here
extern snek_print
extern snek_error
...
```

QUIZ: Assembly for `print`

Program

```
(print 42)
```

Assembly

```
mov rax, <42>
mov rdi, rax
call snek_print
```

```
(print e)
```

```
<e>
mov rdi, rax
Call snek_print
```

QUIZ: Passing parameters with `rdi`

Like with `snek_error`

- We used `rdi` to pass the param to `snek_print`

Unlike with `snek_error`

- We're coming back!

But *unlike* with `snek_error`, we're coming back!

Can you write a test that will *break* our compiler?

```
; INPUT = 200
(block
   (print 100)
   input)
```

Saving RDI

What were we using `rdi` for before? How can we save it?

```
<see code>
      Stash RDI on Stack
      call snek_print
      RESTORE RDI from Stack.
```

CSE 231: Functions

Ranjit Jhala *April 23, 2026*

User-defined Functions

Program

- A list of *function definitions*
- A *main expression*

```
<prog> := <func>* <expr>
```

Function Definitions

- A *name*,
- A *parameter* (for now),
- A *body* expression.

```
<func> := (fun (<name> <ident>) <expr>)
```

Expressions

- ... as before
- plus *function calls*

```
<expr> := ... | (<name> <expr>)
```

QUIZ: Abstract Syntax

```
struct Prog {



}

struct Defn {

}

enum Expr {



}
```

MIDTERM
CENTER 115
3:00-3:50PM
5/1

NO parsing

e := n | x | (add1 ..)
| (f e1)
defn := (fun (f x) e)
prog := <defn>*
<expr>

QUIZ: Semantics

Fill in the result of evaluating the following programs.

Program | *Result*

```
(fun (incr n)
  (add1 n)
)
"main"
(incr 100)
```

{ name: "incr" param: "n" body: (add1 n) }

v = 100

(incr (+ 90 10))

101

```
(fun (fac n)
  (if (= n 0)
    1
    (* n (fac (sub1 n)))
  )
)
"main"
(fac 5)
```

5 × 4 × 3 × 2 × 1 × 1

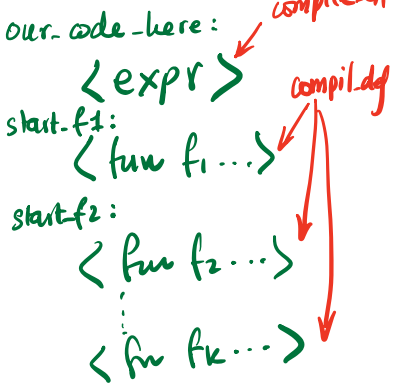
120

PROG

(fun f1 ...)
(fun f2 ...)
⋮
(fun fk ...)

expr

ASM



Evaluation

type FunEnv = Map<String, Defn>

snek_error

call snek_error
↳NEXT
jmp snek_error

```
fn eval(e: &Expr, env: &mut Env, fenv: &FunEnv ) 
   -> Result<Val, Err>
{
  match e { // ...
    Call1(f, arg) => {
      let f_def = fenv.get(f);
      let val = eval(arg, env, funenv);
      let new_env = Env::new().insert(f.param, val);
    }  eval(f.body, new_env)
  }
}
```

Find Defn of 'f'
eval arg to get sow

(let (x1 ...)
  (let (x2 ...)
    (... (foo 94) ...)))

static scope
v
dynamic scope

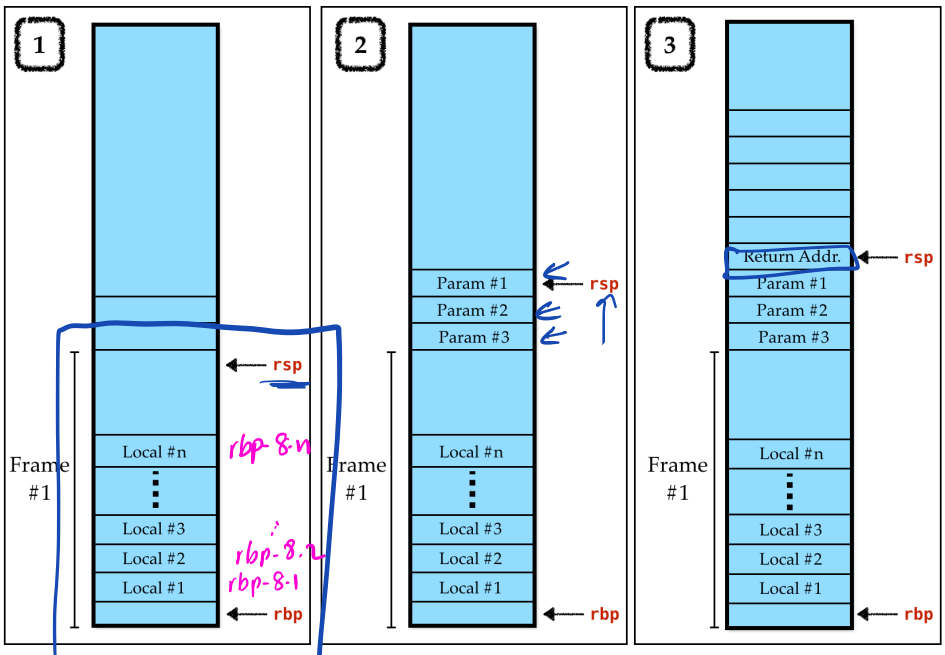
let (x ...)

let (x 10)

(let (f (fun a (+ a x)))

Callers, Callees, and Frames

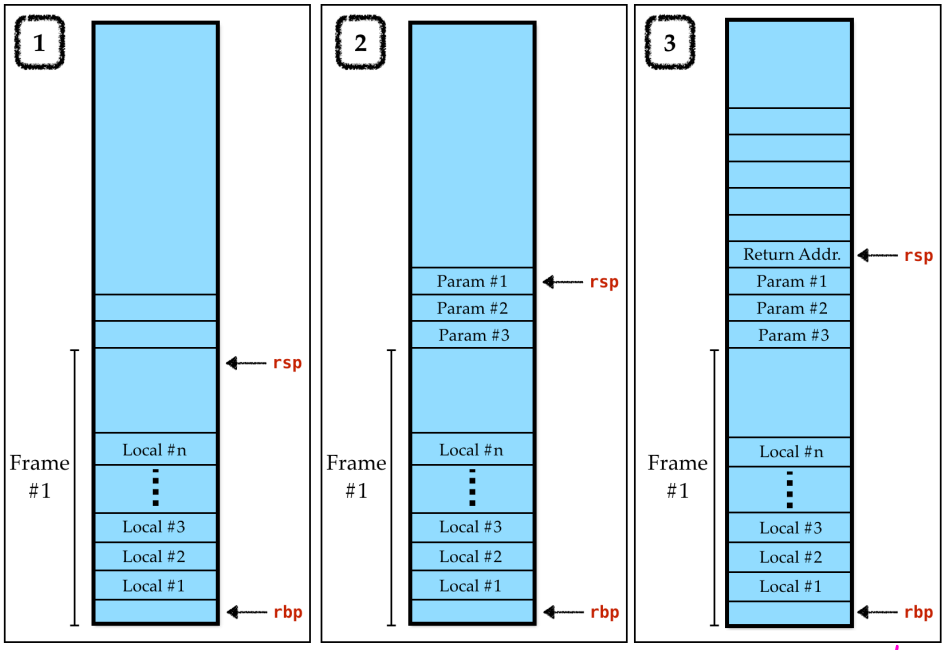
Caller Thula



i
[RBP - 8*i]
k
[RBP + 8*(k+2)]

Callee Daniel

After Daniel RET



Caller (Thula)
- push #arg 3
  push #arg 2
  push #arg 1
  Call "daniel"

Callee (Daniel) SETUP
- save (Thula's) RBP
- rbp ← rsp
- float rsp up to alloc space

callee (Daniel) CLEANUP
- float RSP DOWN
- restore Thula/call RBP
- ret

caller (Thula) [2]
float RSP DOWN

QUIZ: Assembly: Caller

Program	*Assembly*

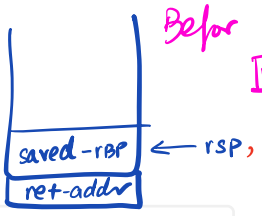
`(incr 100)`	`mov rax, 200`
	`push rax`
	`call incr_start`
	`add rsp, 8`

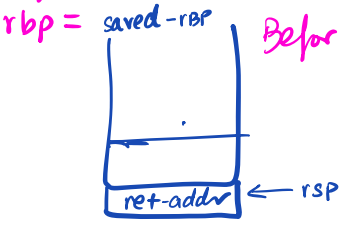
`(f e)`	`; << e >>`
	`push rax`
	`call f_start`
	`add rsp, 8*#args`

QUIZ: Assembly: Callee

Program	*Assembly*

`(fun (incr n)`
`  (add1 n)`
`)`
	`; setup frame`
	`push rbp`
	`mov rbp, rsp`
	`sub rsp, 50*8`
	`; body`
	`<add1 n>`
	`; teardown frame`
	`mov rsp, rbp`
	`pop rbp`
	`ret`





Compiling Calls

```
fn compile_expr(e: &Expr) -> String {


}
```

Compiling Definitions

```
fn compile_defn(defn: &Defn) -> String {



}
```

Frame Setup (`frame_setup`)

```
fn frame_setup(e: &Expr) -> String {




}
```

Frame Teardown (`frame_teardown`)

```
fn frame_teardown(e: &Expr) -> String {




}
```

Frame Allocation

How much *space* should we allocate for a frame?

How much should we float `rsp` up?

```
fn max_vars(e: &Expr) -> usize {









}
```