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

```
```

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

```
```

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

```

```

```
(print e)
```

```

```

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?

```

```

Saving RDI

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

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 {



}
```

QUIZ: Semantics

Fill in the result of evaluating the following programs.

Program *Result*

```
(fun (incr n)
  (add1 n)
)

(incr 100)
```

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

(fac 5)
```

Evaluation

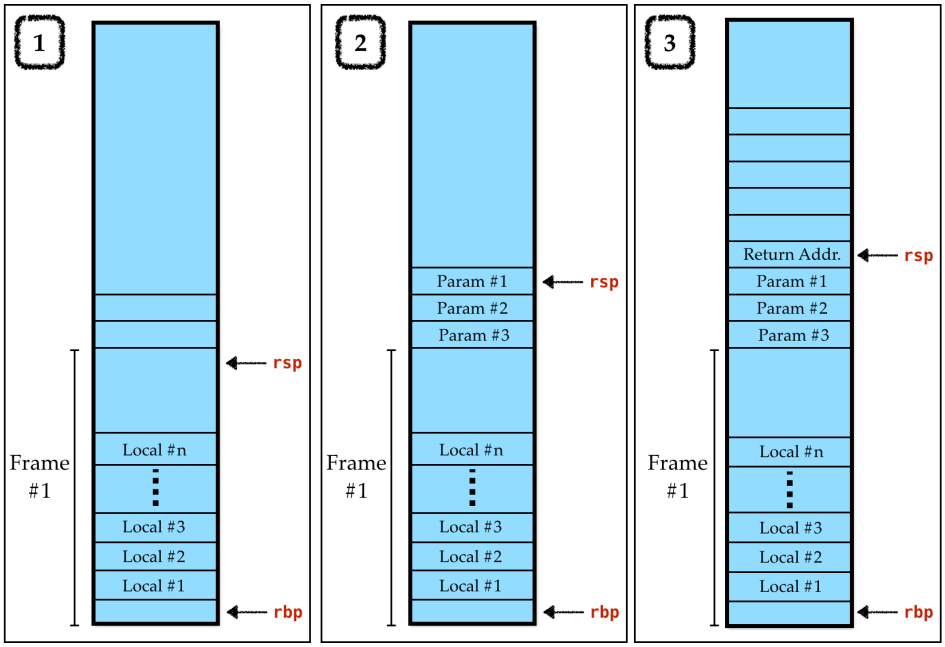
```
fn eval(e: &Expr, env: &mut Env,                    )
   -> Result<Val, Err>
{
  match e { // ...
    Call1(f, arg) => {



    }
  }
}
```

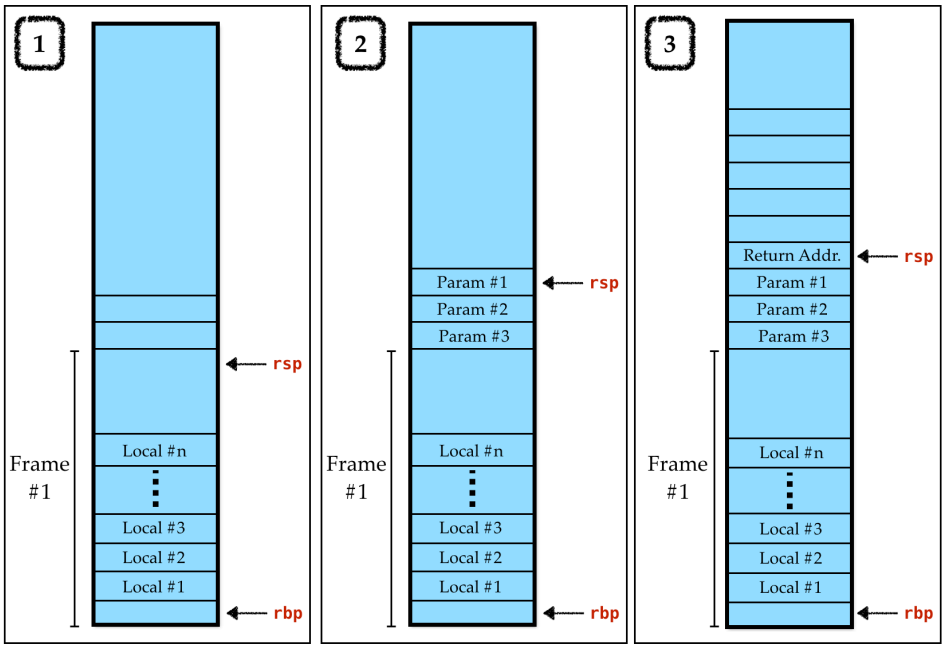
Callers, Callees, and Frames

Caller



,

Callee



,

QUIZ: Assembly: Caller

Program	*Assembly*
`(incr 100)`	`mov rax, 200`
`(f e)`	`; << e >>`

QUIZ: Assembly: Callee

Program	*Assembly*

`(fun (incr n)`
`  (add1 n)`
`)`
	`; setup frame`
	
	`; body`
	
	`; teardown frame`

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 {









}
```