

CSE 231: Boa (Binary Operators)

Ranjit Jhala April 7, 2026

$(+ e_1 e_2)$

$(+ 2 3)$

Binary Operators in Boa

Next, let's build on the machinery for scoping and stacks to add **binary operators** (+, -, *)

$(+ x y)$

Recall the three pieces to define the language

1. **Concrete syntax** — the text the programmer writes
2. **Abstract syntax** — a Rust datatype our compiler uses
3. **Semantics** — what values programs produce

Concrete Syntax

```
<expr> :=
| <number>
| (add1 <expr>)
| (sub1 <expr>)
| (let (<ident> <expr>) <expr>)
| <ident>

| (+ <expr> <expr>)      -- binary addition
| (- <expr> <expr>)      -- binary subtraction
| (* <expr> <expr>)      -- binary multiplication
```

s-expression parenthesized prefix notation, so +, -, * are binary operators in **prefix** form, e.g. (+ 1 2)

QUIZ: Abstract Syntax

```
enum Expr {
  Number(i32),
  Add1(Box<Expr>),
  Sub1(Box<Expr>),
  Let(Ident, Box<Expr>, Box<Expr>),
  Var(Ident),
  // Fill in the cases for +, -, *
}
```

Recall `Box<T>` is a heap pointer — always a fixed 64-bit size, enabling recursive types.

QUIZ: Semantics

Program

Result

```
(+ 1 2)
```

3 $(+ (+ 1 2) 3)$

```
(+ (+ 1 2) 3)
```

6 $(let (tmp (+ 1 2)) (+ tmp 3))$

```
(+ (+ 1 2) (+ 3 4))
```

10 $(+ e_1 e_2)$

```
(let (x 10)
  (let (y 10)
    (+ x y)))
```

20 $(let (tmp, e_1)$

```
(let (x 10)
  (let (x (add1 x))
    (+ x 10)))
```

21 $(let (tmp, e_1) (let (tmp, e_2) (+ tmp, rax)))$

```
(+ (let (x 10) (add1 x))
  (let (y 7) (+ x y)))
```

UNBOUNDED UNDEFI

```
(+ (let (x 10) (add1 x))
  (let (y 7) (+ 10 y)))
```

28

Recall: Stack and Local Variables

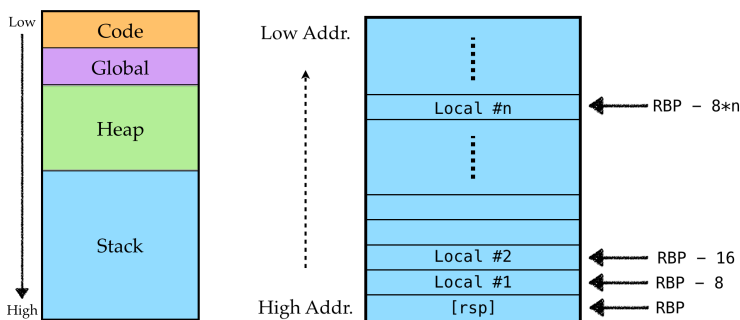


Figure 1: Store i^{th} stack-variable at address $RBP - 8 * i$.

Required

Map from vars (x, y, z, ...) to stack positions (1, 2, 3 ...)

Problem

But what about the **sub-expressions** (+ e1 e2)?

Compilation Strategy: Binary Addition

To compile (+ e1 e2)

1. *Compile* e1 into rax
2. *Save* the value of e1
3. *Compile* e2 into rax
4. *Add* the two values.

But **where** do we stash the value of e1?

QUIZ: Where to store?

Program

Stack Layout

```
(+ 1 2)
```

```
(+ (+ 1 2) 3)
```

```
(+ (+ 1 2) (+ 3 4))
```

```
(let (x 10)
  (let (y 10)
    (+ x y)))
```

```
(let (x 10)
  (let (x (add1 x))
    (+ x 10))))
```

```
(+ (let (x 10) (add1 x))
  (let (y 7) (+ x y)))
```

```
(+ (let (x 10) (add1 x))
  (let (y 7) (+ 10 y)))
```

QUIZ: Assembly*Program*

```
(+ (+ 1 2) (+ 3 4))
```

Assembly

```
mov rax, 1
```

```
(let (x 10)
  (let (y 10)
    (+ (+ x y) 99)))
```

```
mov rax, 10
```

QUIZ: Compilation Code

Fill in the cases for `compile_expr` for `let` and `+`

```
fn compile_expr(expr: &Expr, env: &Env) -> String {
  match expr {
    // cases for Number, Add1, Sub1, Var
    Expr::Let(x, e1, e2) => {
      // 1. compute e1 into RAX
      // 2. save RAX at the position for x
      // 3. compute e2

    }
    Expr::Plus(e1, e2) => {
      // 1. compute e1 into RAX
      // 2. save RAX at TMP position
      // 3. compute e2 into RAX
      // 4. ADD RAX and TMP
    }
  }
}
```