1 Introduction to Binary Operations

Consider the following s-expression: `(sub1 (sub1 (add1 73)))`. Looking at the code discussed in the lecture handout, and assuming `main` runs, what does the stack and heap look like when

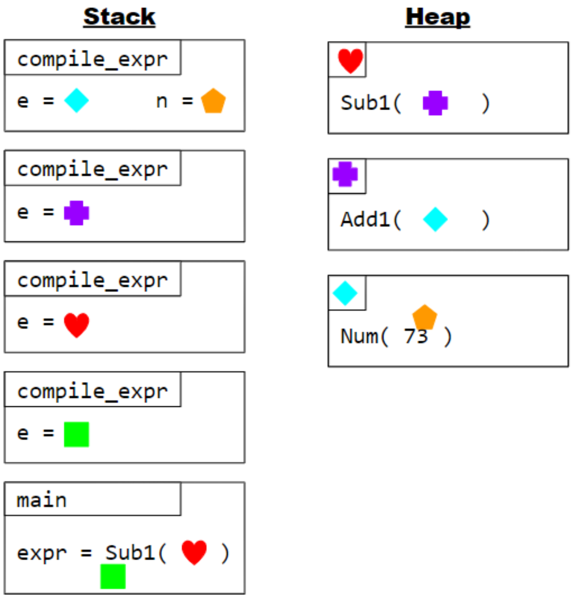
```
format!("mov rax, {}", *n)
```

evaluates?

We'll take a look at the function calls of `compile_expr(&expr)`. First, note that Rust will store objects on the stack unless you allocate it on the heap. Recall, from the previous lecture, we have the AST representation

```
Expr::Sub1(
    Box::new(Expr::Sub1(
        Box::new(Expr::Add1(
            Box::new(
                Expr::Num(73)
            )
        ))
    ))
)
```

Our code initially calls `compile_expr(&expr)`, where `&expr` is a reference to the above object. Note that the outer `Expr::Sub1` is in the stack, but the data in each of the `Enums` will be allocated in the heap. In any case, after calling the function initially, it makes a recursive call with the argument being the held data of the inner object. This repeats until we reach the end (when we have the `Num`).



1.1 Adding Binary Operation Support

Let's suppose we want to add `(+ <expr> <expr>)` to our compiler. Our grammar for our language might look like

```
(*
        expr := <number>
            | (add1 <expr>)
            | (sub1 <expr>)
            | (+ <expr> <expr>)
    *)
```

The Expr enum might look like

```
    enum Expr {
        Num(i32),
        Add1(Box<Expr>),
        Sub1(Box<Expr>),
        Plus(Box<Expr>, Box<Expr>),
    }
```

The parse_expr function might look like

```
pub fn parse_expr(s: &Sexp) -> Expr {
    match s {
        Sexp::Atom(I(n)) => Expr::Num(i32::try_from(*n).unwrap()),
        Sexp::List(list) => match &list[..] {
            [Sexp::Atom(S(op)), e] if op == "add1" =>
                Expr::Add1(Box::new(parse_expr(e))),
            [Sexp::Atom(S(op)), e] if op == "sub1" =>
                Expr::Sub1(Box::new(parse_expr(e))),
            [Sexp::Atom(S(op)), e] if op == "negate" =>
                Expr::Negate(Box::new(parse_expr(e))),
            [Sexp::Atom(S(op)), e1, e2] if op == "+" => {
                Expr::Add(Box::new(parse_expr(e1)), Box::new(parse_expr(e2)))
            }
            _ => panic!("parse error"),
        },
        _ => panic!("parse error"),
    }
}
```

Then, our compile_expr function might look like

```
    fn compile_expr(e: &Expr, si: i32) -> String {
        match e {
            Expr::Num(n) => format!("mov rax, {}", *n),
            Expr::Add1(subexpr) => compile_expr(subexpr) + "\nadd rax, 1",
            Expr::Sub1(subexpr) => compile_expr(subexpr) + "\nsub rax, 1",
            Expr::Plus(e1, e2) => {
                // ?
            }
        }
    }
```

There are two ways we can implement the Plus part of the function.

A

```
let e1_instrs = compile_expr(e1);
let e2_instrs = compile_expr(e2);
e1_instrs + "\n mov rbx, rax"
    + e2_instrs + "\n add rax, rbx"
```

B

```
let e1_instrs = compile_expr(e1, si);
let e2_instrs = compile_expr(e2, si + 1);
let stack_offset = si * 4;
format!("
    {e1_instrs}
    mov [rsp - {stack_offset}], rax
    {e2_instrs}
    add rax, [rsp - {stack_offset}]
")
```

(Exercise.) With option (a), what is the assembly code generated after compiling the following code? What is the result of running the assembly?

(a) `(+ (+ 100 30) 4)`

```
mov rax, 500
mov rbx, rax
mov rax, 30
mov rbx, rax
mov rax, 9
add rax, rbx
add rax, rbx
ret
```

The result is `134`, as expected.

(b) `(+ 500 (+ 30 9))`

```
mov rax, 500
mov rbx, rax
mov rax, 30
mov rbx, rax
mov rax, 9
add rax, rbx
add rax, rbx
ret
```

The result is `69`, which isn't what we were expecting. Notice how, in the second line, we effectively put `500` into `rbx`. In the fourth line, we overwrite `500` with `30`. In any case, this isn't what we were expecting, so option (a) will not work.