1 Binary Operations

We'll now look at adding functionality for binary operations.

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>),
    }
```

1.1.1 Creating the Parser

Modifying the parser is simple. 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"),
    }
}
```

Because we know that addition will always be of the form `(+ <expr> <expr>)`, we simply need to add a match condition to the inner `match` checking if the operation is `+`.

1.1.2 Modifying the Compiler: Part 1

Suppose we update the `compile_expr` function so it looks like

```
    fn compile_expr(e: &Expr) -> 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) => {
            let e1_instrs = compile_expr(e1);
            let e2_instrs = compile_expr(e2);
            e1_instrs + "\n mov rbx, rax" + &e2_instrs + "\n add rax, rbx"
        }
    }
}
```

(Exercise.) Using the above code for compiling, 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.

1.1.3 Modifying the Compiler: Part 2

Clearly, the first attempt at modifying the compiler didn't work. In fact, while this may work for simple addition operations, this won't work for more complex addition operations.

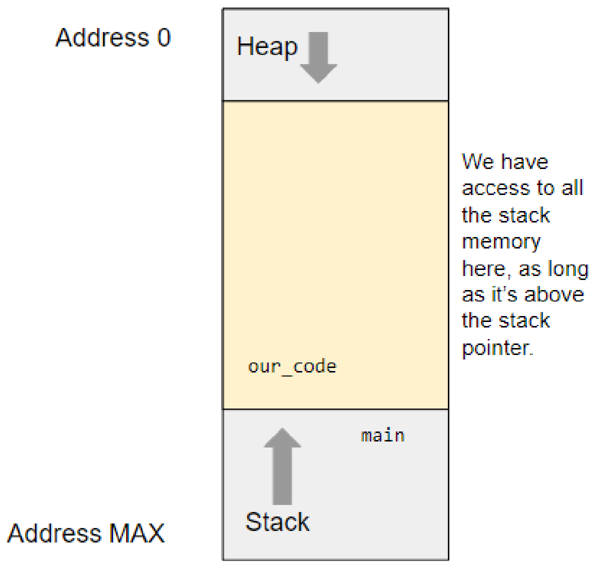
One solution is to essentially store important values in the stack, and then refer to the values in the stack when doing addition. This gives us the following code:

```
fn compile_expr(e: &Expr, si: i32) -> String {
    match e {
        Expr::Num(n) => format!("mov rax, {}", *n),
        Expr::Add1(subexpr) => compile_expr(subexpr, si) + "\nadd rax, 1",
        Expr::Sub1(subexpr) => compile_expr(subexpr, si) + "\nsub rax, 1",
        Expr::Plus(e1, e2) => {
            let e1_instrs = compile_expr(e1, si);
            let e2_instrs = compile_expr(e2, si + 1);
            let stack_offset = si * 8;
            format!("
                {e1_instrs}
                mov [rsp - {stack_offset}], rax
                {e2_instrs}
                add rax, [rsp - {stack_offset}]
            ")
        }
    }
}
```

Some technical remarks:

- In our current setting, we do not need to allocate any memory. All we're doing is using is the stack memory, which the operating system gives us. The highest address is where the stack begins, and grows in decreasing address[1]. If we need to worry about creating a lot of objects or storing a lot of information on the heap, then we need to allocate memory.

  Roughly speaking, this might look like


Address 0
Heap
our_code
main
Stack
Address MAX
We have access to all the stack memory here, as long as it's above the stack pointer.

- `rsp` is the register representing the stack pointer. `rsp` points to the very bottom of the stack frame that we get to use for all of our generated code.
- In the line `si * 8`, the `8` represents the size of a Word. This will be `8` for a 64-bit machine and `4` for a 32-bit machine.
- The `si` parameter is the **stack index**. The stack index should initially be a small positive integer (e.g., 1 or 2). The stack index simply represents how deep in nested expressions or temporary variables we need to save. In this class, we'll prefer using the value `2` (i.e., 2 words up from where `rsp` is) because the bottom of the stack is where information like the return pointer (how we can use `ret`) is stored. We will generally use information one other word at the bottom of the stack.

[1]All the operating system will do is, if we somehow use the entire stack space, we'll probably end up with a segfault.

In this sense, we'll initially call `compile_expr` with the arguments `e` (our expression to compile) and `2` (the stack index).

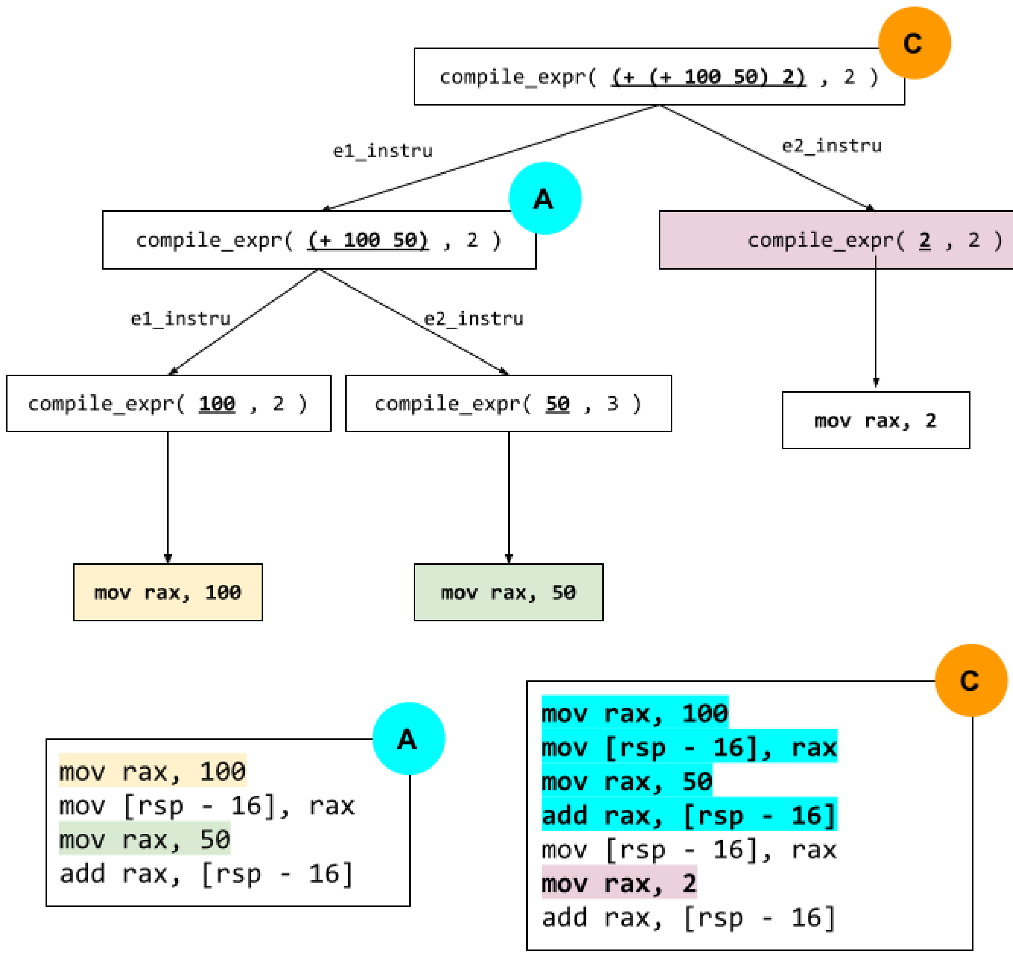
(Exercise.) Using the newly revised code for compiling, what is the assembly code generated after compiling the following code?

```
(+ (+ 100 50) 2)
```

Roughly speaking, the AST representation of the above code is

```
Plus(
    Plus(
        Num(100),
        Num(50)
    ),
    Num(2)
)
```

Drawing out the recursion tree gives us



```
mov rax, 100
mov [rsp - 16], rax
mov rax, 50
add rax, [rsp - 16]
```

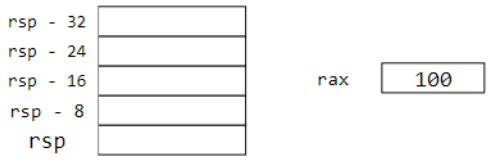
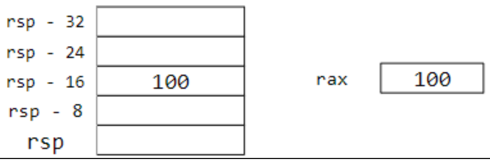
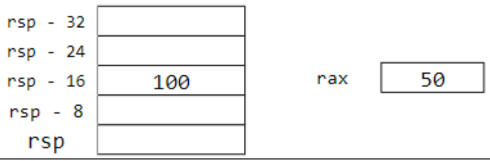
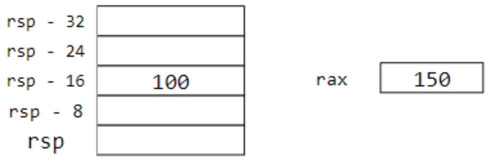
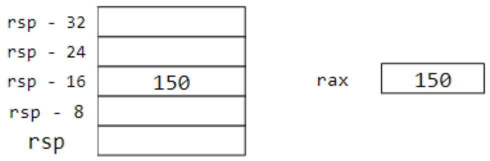
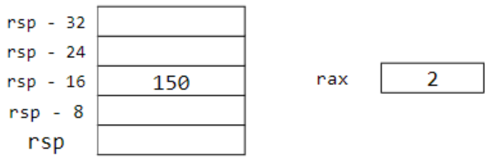
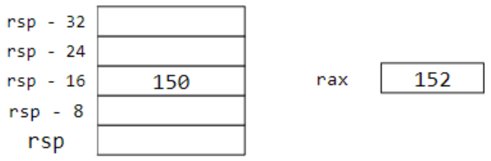
```
mov rax, 100
mov [rsp - 16], rax
mov rax, 50
add rax, [rsp - 16]
mov [rsp - 16], rax
mov rax, 2
add rax, [rsp - 16]
```

Note that the values passed in the function calls are intended to make things more clear, and will not compile otherwise.

Here's the resulting assembly.

```
mov rax, 100
mov [rsp - 16], rax
mov rax, 50
add rax, [rsp - 16]
mov [rsp - 16], rax
mov rax, 2
add rax, [rsp - 16]
```

Suppose we run through each line of this assembly. This is what the resulting stack memory might look like:

Executed Line	Result After Running
`mov rax, 100`	 rsp - 32 rsp - 24 rsp - 16 rsp - 8 rsp rax 100
`mov [rsp - 16], rax`	 rsp - 32 rsp - 24 rsp - 16 100 rsp - 8 rsp rax 100
`mov rax, 50`	 rsp - 32 rsp - 24 rsp - 16 100 rsp - 8 rsp rax 50
`add rax, [rsp - 16]`	 rsp - 32 rsp - 24 rsp - 16 100 rsp - 8 rsp rax 150
`mov [rsp - 16], rax`	 rsp - 32 rsp - 24 rsp - 16 150 rsp - 8 rsp rax 150
`mov rax, 2`	 rsp - 32 rsp - 24 rsp - 16 150 rsp - 8 rsp rax 2
`add rax, [rsp - 16]`	 rsp - 32 rsp - 24 rsp - 16 150 rsp - 8 rsp rax 152