1 Local Variables

In this lecture, we want to add support for local variables through `let` expressions.

(Exercise.) Consider the following code.

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

Using code discussed in the previous section, along with some intuition, what assembly do you think should be produced?

The assembly that could possibly be produced is as follows:

```
mov rax, 10
mov [rsp - 16], rax
mov rax, 10
mov [rsp - 24], rax
mov rax, [rsp - 16]
mov [rsp - 32], rax
mov rax, [rsp - 24]
add rax, [rsp - 32]
```

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

```
Expr::Num(n) => format!("mov rax, {}", *n),
    .
    .
    .
Expr::Plus(e1, e2) => {
    let e1_instrs = compile_expr(e1, si, env);
    let e2_instrs = compile_expr(e2, si + 1, env);
    let stack_offset = si * 8;
    format!("
        {e1_instrs}
        mov [rsp - {stack_offset}], rax
        {e2_instrs}
        add rax, [rsp - {stack_offset}]
    ")
},
```

At a high level, in terms of variable declaration,

- We defined a local variable `x` with value `10`. So, it would make sense to store the value somewhere (e.g., at location `rsp - 16` in the stack). This corresponds to the first two lines of assembly, which are highlighted yellow.
- In the body of the first `let`-expression, we defined a local variable `y` with value `10`. So, again, it would make sense to store this value somewhere (e.g., at location `rsp - 24` in the stack, since we wouldn't want to overwrite the value at `rsp - 16`). This corresponds to the second two lines of assembly, highlighted orange.

Next, we're performing the addition. Note that we're working with the expression `(+ x y)`. The assembly generated by the addition is found under the `Expr::Plus` branch.

- It would make sense to put the value corresponding to `x` into our register where we're storing the answer, `rax`. Since the value corresponding to `x` is stored in the stack (at location `rsp - 16`), we need to *move* the value over to `rax`. This corresponds to the fifth line in the assembly (highlighted blue). In this sense, we can assume that `e1_instrs` returns just that line: `mov rax, [rsp - 16]`.

- Next, according to how we defined the instructions for addition, we need to move the value stored in `rax` to the stack memory, `[rsp - 32]`. This corresponds to the sixth line in the assembly (highlighted orange). Note that `32` is picked since we don't want to write this value to `rsp - 24` (this would overwrite `y`'s value.)
- Similarly, for variable `y`, we need to store its value into the register `rax`. Remember that `y`'s value is stored in the stack at location `rsp - 24`. So, we can move the value from this location to `rax`. This corresponds to the seventh line in the assembly (highlighted pink).
- Finally, from the last line in the `Expr::Plus` branch, we need to add the value stored at `rsp - 32` to `rax`. Recall that `[rsp - 32]` has `x`'s value (since we moved `x`'s value to `[rsp - 32]` from the previous two steps). This corresponds to the last line in the assembly (highlighted orange).

This gives us the desired result in `rax`.

There are several things we want to consider here.

- How do we modify the grammar and our code to account for these changes?
- How do we store the identifiers and their stack offsets?

1.1 Grammar and AST

Our expression now takes on two new forms:

- A `let` expression, which takes a *binding* consisting of an identifier and an associated expression, and additionally a corresponding body to be executed.
- An identifier expression itself (this is how we refer to an identifier).

Our grammar will look something like this:

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

The `Expr` `enum`, our AST representation, might look like

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

To reiterate, in `Let(String, Box<Expr>, Box<Expr>)`,

- The `String` and first `Box<Expr>` represents the *binding*, where the `String` is the identifier and the first `Box<Expr>` is the expression. Here, we're associating the expression to the identifier.
- The last `Box<Expr>` is the *body* that follows the `let`-expression.

1.2 Modifying the Parser

We need to modify our parser to account for the two different expressions, the `let` expression and the identifier expression.

1.2.1 The `let`-Binding

Remember that, in the s-expression, the `let`-binding will look like

```
(let (<name> <expr>) <expr>)
 [a]   [b]     [c]     [d]
```

Here, this corresponds to having a List of atoms and expressions. Namely, we have

- (a) an atom with a `String` value equal to `let` (this is how we know this is a `let`-binding),
- an *expression*, represented as a `List`, with (b) an atom representing the identifier and (c) the expression to bind the identifier with.
- (d) the body of the `let`-statement, also an expression.

This gives us the following branch for `let`-bindings:

```
[Sexp::Atom(S(op)), Sexp::List(binding), body] if op == "let" => {
    match &binding[..] {
        [Sexp::Atom(S(id)), expr] => {
            Expr::Let(id.to_owned(), Box::new(parse_expr(expr)),
                Box::new(parse_expr(body)))
        }
        _ => panic!("parse error"),
    }
}
```

1.2.2 The Identifier Case

Our identifier, like a number, is just by itself. For example, (+ 10 x) evaluates to $10 + x$, where x is an atom. One thing to note is that identifiers are *Strings*, just like how numbers are *Integers*. So, this gives us the following branch for identifiers:

```
Sexp::Atom(S(id)) => Expr::Id(id.to_owned()),
```

1.2.3 Putting it Together

Our parser now looks something like the below.

```
pub fn parse_expr(s: &Sexp) -> Expr {
    match s {
        Sexp::Atom(I(n)) => Expr::Num(i32::try_from(*n).unwrap()),
        Sexp::Atom(S(id)) => Expr::Id(id.to_owned()),
        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)), e1, e2] if op == "+" => {
                Expr::Plus(Box::new(parse_expr(e1)), Box::new(parse_expr(e2)))
            }
            [Sexp::Atom(S(op)), Sexp::List(binding), body] if op == "let" => {
                match &binding[..] {
```

```
                    [Sexp::Atom(S(id)), expr] => {
                        Expr::Let(id.to_owned(), Box::new(parse_expr(expr)),
                            Box::new(parse_expr(body)))
                    }
                    _ => panic!("parse error"),
                }
            }
            _ => panic!("parse error"),
        },
        _ => panic!("parse error"),
    }
}
```

1.3 Modifying the Compilers

There are some things we need to consider.

- We need to store all the identifiers and their stack offsets (where in the stack their values are stored in). For this, we can make use of a `HashMap<String, i32>`, which we'll call our **environment** (`env`).
- Like with the parsing, we need to create two new branches for the `Let` case and the `Id` case.

In this course, we'll make use of the `HashMap` implementation from the `im` crate (i.e., `im::HashMap`). The difference between this `HashMap` implementation and the one in the standard library is that `im::HashMap` will create a brand new `HashMap` object when you update the map, whereas the standard library version will update the original map. The reason why we're using `im::HashMap` is because we don't need to worry about removing the identifier from the map once we're done recursively calling the `compile_expr` function.

1.3.1 The Identifier Case

The identifier is relatively straightforward. Notice how, in the exercise at the beginning of this section, whenever we refer to an identifier, we simply *move* the value (stored in the stack) corresponding to the identifier to `rax`.

Remember that our map, `env`, has the identifier and its offset. So, we can *get* the offset from the map and use that.

Therefore, the identifier case for the compiler looks like

```
Expr::Id(id) => format!("mov rax, [rsp - {}]", env.get(id.as_str()).unwrap()),
```

1.3.2 The `let`-Binding

For the `Let` case, we have three associated values: the identifier, expression associated with the identifier, and the body associated with the binding itself.

At a high level, the idea is as follows:

- First, we want to *compile* the expression associated with the identifier. At the end, the value should be stored in `rax`. You can observe the other branches within the `compile_expr` function (from the previous sections) to confirm this; in the branches, the last assembly instruction always involves moving or adding something *to* `rax`.
- Once we compiled the expression and have our result in `rax`, we need to do two things.
  - First, we need to store the result somewhere! We can make use of the current stack index to get the appropriate stack offset. Once we have this offset, we can *move* the result in `rax` to that location in the stack.

  - Next, we should probably store the identifier and where its value is stored in the stack (i.e., stack offset) in our environment `env`. So, we can just update the map to include this information.

- Now that we've done this, we can compile the body. Note that we want to increment the stack index by 1 when compiling the body – otherwise, there's a real possibility that we'll overwrite the value stored in the stack (you know, the value corresponding to the identifier) with a different value.

This gives us the branch for the compiler,

```
Expr::Let(id, ex, body) => {
    let ex_instr = compile_expr(ex, si, env);
    let new_env = env.update(id.to_owned(), si * 8);
    let body_instr = compile_expr(body, si + 1, &new_env);
    format!("
        {ex_instr}
        mov [rsp - {}], rax
        {body_instr}
    ", si * 8)
}
```

1.3.3 Putting it Together

Our compiler should now look something like the below.

```
fn compile_expr(e: &Expr, si: i32, env: &HashMap<String, i32>) -> String {
    match e {
        Expr::Num(n) => format!("mov rax, {}", *n),
        Expr::Add1(subexpr) => compile_expr(subexpr, si, env) + "\nadd rax, 1",
        Expr::Sub1(subexpr) => compile_expr(subexpr, si, env) + "\nsub rax, 1",
        Expr::Plus(e1, e2) => {
            let e1_instrs = compile_expr(e1, si, env);
            let e2_instrs = compile_expr(e2, si + 1, env);
            let stack_offset = si * 8;
            format!("
                {e1_instrs}
                mov [rsp - {stack_offset}], rax
                {e2_instrs}
                add rax, [rsp - {stack_offset}]
            ")
        },
        Expr::Let(id, ex, body) => {
            let ex_instr = compile_expr(ex, si, env);
            let new_env = env.update(id.to_owned(), si * 8);
            let body_instr = compile_expr(body, si + 1, &new_env);
            format!("
                {ex_instr}
                mov [rsp - {}], rax
                {body_instr}
            ", si * 8)
        }
        Expr::Id(id) => format!("mov rax, [rsp - {}]", env.get(id.as_str()).unwrap()),
    }
}
```