1 Introduction to `if`-Expressions (Part 2)

(Continued from Lecture 6.)

1.1 Modifying the Runtime

In our class, we had a `runtime.rs` file which was responsible for calling our assembly code and printing out the result; this looks something like:

```
#[link(name = "our_code")]
extern "C" {
    #[link_name = "\x01our_code_starts_here"]
    fn our_code_starts_here() -> i64;
}

fn main() {
    let i: i64 = unsafe { our_code_starts_here() };
    println!("{i}");
}
```

One thing to consider here is that, with our new tagging system, we have two problems

- The code above won't print out boolean values properly (it'll either print out `3` or `1`, not `true` or `false` like we would hope).
- It also won't print out the numbers correctly. Remember that the integers have been shifted one bit to the left, and this code doesn't account for that when printing the result. So, if the code is supposed to print `5`, then this would actually print `10`.

The solution is to modify this file so that it can correctly interpret the resulting value that the assembly code produces. We might have something like the below.

```
fn main() {
    let i: i64 = unsafe { our_code_starts_here() };
    // If we have an integer (remember that, for numbers, the LSB should be 0)
    if i & 1 == 0 {
        println!("{}", i >> 1);
        return;
    }

    // Otherwise, i & 1 -> 1, so we should have a boolean value. Let b
    // be either 0 or 1 (if we have a valid boolean).
    let b = i >> 1;
    // If b is 0 or 1, then we have a boolean value.
    if b == 0 || b == 1 {
        println!("{}", b == 1);
    } else {
        println!("unknown value: {i}");
    }
}
```

1.2 The Abstract Syntax

Recall that an `if`-expression looks something like

```
(if <expr> <expr> <expr>)
```

where

- the first `<expr>` represents the condition expression; this determines which of the subsequent expressions should be executed.
- the second `<expr>` represents the "then" expression; this expression should be executed if the condition expression resolves to `true`.
- the third and last `<expr>` represents the "else" expression; this expression should be executed if the condition expression resolves to `false`.

As defined in the grammar, we also need to be able to support boolean values (`true` and `false`). This is trivially just `True` and `False` in the abstract syntax. So, the abstract syntax will look like

```
enum Expr {
    Num(i32),
    True,                                   // New!
    False,                                  // New!
    Add1(Box<Expr>),
    Plus(Box<Expr>, Box<Expr>),
    Let(String, Box<Expr>, Box<Expr>),
    Id(String),
    Eq(Box<Expr>, Box<Expr>),               // New!
    If(Box<Expr>, Box<Expr>, Box<Expr>)     // New!
}
```

1.3 Extending the Parser

Because our `if`-expression is basically a list of four expressions, we just need to match that particular pattern. So, parsing `if`-expressions should be straightforward.

```
match s {
    ...
    Sexp::List(list) => match &list[..] {
        [Sexp::Atom(S(keyword)), cond, thn, els] if keyword == "if" => Expr::If(
            Box::new(parse_expr(cond)),
            Box::new(parse_expr(thn)),
            Box::new(parse_expr(els)),
        )
        ...
    }
}
```

So, how do we represent boolean types? One way is by looking at the identifier: if the identifier happens to be `true` or `false`, we can assume that this is a `boolean` type. Otherwise, we can assume that we just have a regular identifier.

```
match s {
    Sexp::Atom(S(id)) => {
        if id == "true" {
            Expr::True
        } else if id == "false" {
            Expr::False
        } else {
            Expr::Id(id.to_owned())
        }
    }
    ...
}
```

Finally, parsing the equality operator (e.g., `(= 10 5)`) is the same as with parsing the plus operator.

1.4 Implementing the Compiler

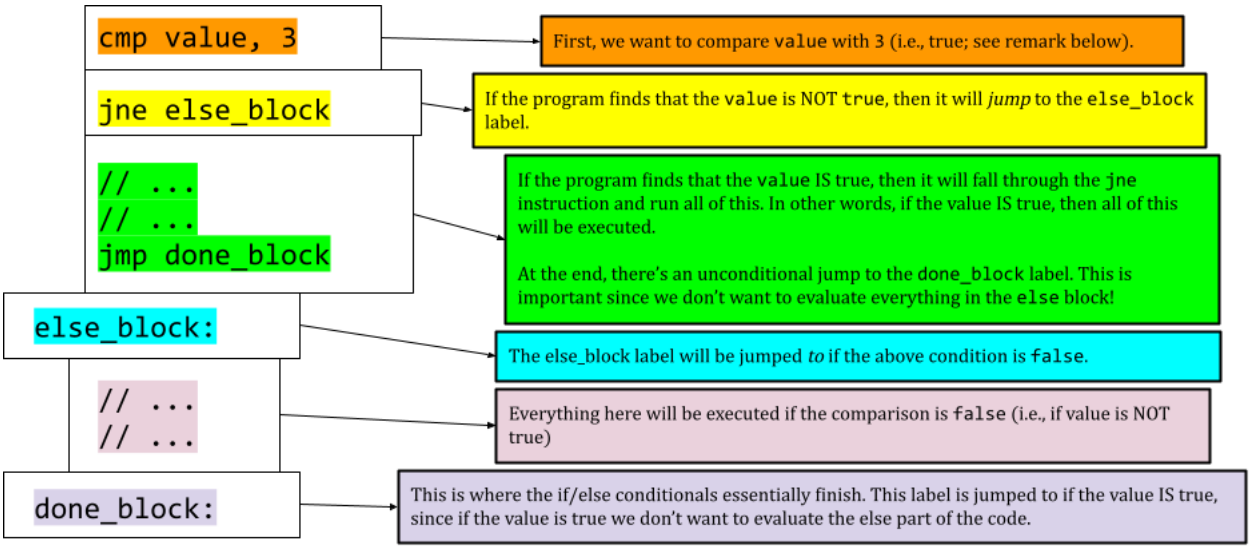
Before we start, one adjustment we need to make to account for the tagging system that we've discussed is to shift integers one bit to the left. That is, when we have an integer, we need to shift it one bit to the left so the least significant bit can be used as the tag bit.

```
match e {
    Expr::Num(n) => format!("mov rax, {}", *n << 1),
    ...
}
```

Now, let's think about `if`-expressions. `if`-expressions imply that only a subset of code will be executed. For example, if we have an `if`-statement and the condition resolves to `false`, the code inside the `if`-statement will not be executed. In assembly, we can represent this behavior using **branches**. Roughly speaking, the assembly will look like

```
    cmp value, 3        ; if value == true {
    jne else_block      ;       // your code
    // code here        ;       // here
    jmp done_block      ;
else_block:             ; } else {
    // code here        ;       // your code here
done_block:             ; }
```

What's going on?



Remark: `3` is represented as `true` (the binary representation of `3` is `11`; notice how the tag bit is `1`).

1.4.1 Duplicate Labels

All `if`-expressions will follow the same pattern[1]. However, if we just use this same exact template, we'll end up with multiple duplicate label declarations. In other words, we might end up with something like

```
    cmp value 3
    jne else_block
    // ...
    jmp done_block
else_block:
```

[1] Maybe not the exact pattern, but the same behavior

```
    // ...
done_block:
    // ...
    cmp value 3
    jne else_block
    // ...
    jmp done_block
else_block:
    // ...
done_block:
```

This code would fail to run since there are, for example, duplicate `done_block` label declarations. So, we need to create a unique label whenever we do declare a label. Let's declare a function that does just this:

```
fn new_label(l: &mut i32, s: &str) -> String {
    let current = *l;
    *l += 1;
    format!("{s}_{current}")
}
```

At a high level, `new_label` takes in a mutable reference to an integer and a string, and creates a label using those inputs. The integer is incremented – this is important since this guarantees that every label created from this function will be unique.

1.4.2 The Equals Comparison Operator

How do we check if two expressions are equal? The process is relatively similar to adding two expressions. The only difference is at the end, when instead of actually *adding* the values, we put either 3 (`true`) or 1 (`false`) into `rax`.

The way we do this (putting either `3` or `1` into `rax`) is literally just another `if`-statement! At a high level, this might look like:

```
if rax == [rsp - stack_offset] {
    rax = 3
} else {
    rax = 1
}
```

So, compilation might look something like this:

```
match e {
    Expr::Eq(a, b) => {
        let a_instr = compile_expr(a, si, env, counter);
        let b_instr = compile_expr(b, si + 1, env, counter);
        let else_label = new_label(counter, "ifelse");
        let end_label = new_label(counter, "ifend");
        let stack_offset = si * 8;

        format!("
        {a_instr}
        mov [rsp - {stack_offset}], rax
        {b_instr}
        cmp rax, [rsp - {stack_offset}]
        jne {else_label}
          mov rax, 3
          jmp {end_label}
```

```
            {else_label}:
              mov rax, 1
            {end_label}:
            ")
        }
        ...
    }
```

1.4.3 `if`-Expressions

With everything in mind, our final implementation of `if`-expressions will look something like the below.

```
match e {
    Expr::If(cond, thn, els) => {
        let end_label = new_label(counter, "ifend");
        let else_label = new_label(counter, "ifelse");
        let cond_instrs = compile_expr(cond, si, env, counter);
        let thn_instrs = compile_expr(thn, si, env, counter);
        let els_instrs = compile_expr(els, si, env, counter);
        format!(
            "
           {cond_instrs}
           cmp rax, 3
           jne {else_label}
             {thn_instrs}
             jmp {end_label}
           {else_label}:
             {els_instrs}
           {end_label}:"
        )
    }
    ...
}
```

At a high level,

- First, we should evaluate the conditional part of the `if`-statement. We should[2] end up with a boolean value in `rax`.
- With the boolean value in `rax`, we can determine which code (either the "then" or "else" blocks) to run.

[2] We'll talk more about type validation later.