1 Introduction to if-Expressions

In this section, we'll discuss how to implement `if`-expressions. Our concrete syntax for this extension will look like

```
(*
    expr :=
        | <number>
        | true                              // New!
        | false                             // New!
        | <name>
        | (add1 <expr>)
        | (sub1 <expr>)
        | (+ <expr> <expr>)
        | (let (<name> <expr>) <expr>)
        | (if <expr> <expr> <expr>)         // New!
        | ...
*)
```

1.1 Structure of if-Expressions

An `if`-expression looks like

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

- Here, 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`.

Before we talk more about how `if`-expressions should be evaluated, we need to figure out how `if`-expressions should work in the first place in terms of what is allowed and what isn't.

1.2 Boolean Values

Let's begin by figuring out what some sample programs should resolve to.

(Exercise.) What should the following programs evaluate to?

a.
```
(let (x 5)
    (if (= x 10) (+ x 2) x))
```

This should evaluate to `5`. We first defined $x = 5$ and then used that in our `if`-expression to see if $x = 10$; since it doesn't, we just return `x`, which is 5.

b. `(if 5 true false)`

There are several reasonable answers we can consider here.

- `true`: since `5` is a truthy value (i.e., evaluates as a true expression), it would make sense for this program to return `true`.
- `false`: since `5` isn't a boolean expression, we could just have the program return `false`.
- an error: since `5` isn't a boolean expression, we can throw an error telling the user that this isn't an boolean expression.

In our class, we'll say `true`. In other words, we're allowing truthy and falsy values.

c. `(+ 7 true)`

There are two answers we can have for this.

- `8`: if `true` implicitly resolves to `1`, then this is just $7 + 1 = 8$.
- an error: since `7` and `true` are different types, it wouldn't make sense to add them.

In our class, we'll say that this expression should throw an error at compile-time.

d. `(= true 1)`

There are two answers we can have for this.

- `true`: for the same reason as above.
- an error: for the same reason as above.

In our class, we'll say that this expression should throw an error at compile-time.

Based on our discussion above, in this class,

- We'll allow truthy and falsy values to be resolved to boolean types (`true` and `false`, respectively) when used as conditions in `if`-expressions.
- We won't allow the mixing of types when doing comparison (e.g., `=`, `<`, etc.) or arithmetic (e.g., `+`, `-`) operations. These should throw an error during runtime[1].

1.3 Boolean Representation

With the above discussion in mind, how should we best represent boolean values in assembly?

(Exercise.) *Intuitively* (i.e., using our knowledge from previous sections), what should be in `RAX` after these are done evaluating?

a. `1`

Trivially, we just move `1` into `RAX`.

b. `5`

Trivially, we just move `5` into `RAX`.

c. `-3`

[1] In our class, we won't be working on a type checker. However, if we did work on a type checker, then we could make mixing of types when doing these operations a compile-time (parse) error.

Trivially, we just move `-3` into `RAX`.

d. `true`

There's not exactly a way to directly represent `true` in assembly, so the best we can do is a truthy value. For now, let's just move `1` into `RAX`.

e. `false`

Same idea as before: the best we can do is a falsy value. For now, let's just move `0` into `RAX`.

f. `(= 3 5)`

Since $3 = 5$ is false, we can just put `0` into `RAX`.

g. `(+ 4 7)`

Trivially, we just move the result of $4 + 7$, or `11`, into `RAX`.

Remember that we didn't want the mixing of types when doing any comparison or arithmetic operations, e.g., `(+ 5 true)` should throw an error. However, with our answers above, how do we know that the `1` in `RAX` isn't actually a `true` value?

1.4 A New Representation of Numbers: Tagging

We will now represent numbers as **64-bits** instead of 32-bits like in previous sections. With this in mind, this is 64 bits:

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

The number 5 can be represented like so:

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0101
```

Let's suppose we *shift* 5 one to the left to get the number 10:

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1010
```

If we're okay with 63-bit numbers, we can use the **least significant bit** (i.e., the right-most bit) as a **tag**. This tag tells us if the value is a number or a boolean value. In this class,

- Numbers will have a least significant bit of `0`.
- Booleans will have a least significant bit of `1`.

For example, we can represent the number `13` as

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1010
```

We can represent the boolean `false` as

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
```

We can represent the boolean `true` as

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0011
```

We can represent the number `0` as

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

1.5 Consequences of Tagging

Because we effectively have to shift everything one to the left to introduce tagging, we need to think about a few things.

- **Addition**
  - When we are performing any binary (or unary) operations, we need to check if the least significant bit is correct for the given input. For example, when we are doing addition, we should check if the least significant bit of both inputs are `0` (implying that both are numbers). If any one of them isn't, then we should throw an error.
  - Otherwise, addition is pretty straightforward. Assuming we have two numbers, adding two numbers should not change since the addition of both least significant bits will be 0 (since $0 + 0 = 0$). In other words, something like `(+ 3 5)` will generate the assembly

    ```
    mov rax, 6             ; 3 / 11 (decimal / binary)
                           ; -> 6 /   110 (decimal / binary) accounting for tag
    mov [rsp - 16], rax
    mov rax, 10            ; 5 / 101 (decimal / binary)
                           ; -> 10 / 1010 (decimal / binary) accounting for tag
    add rax, [rsp - 16]    ; 6 + 10 -> 10000 (with tagging)
    ```

    Notice how `10000` in binary is `16` in decimal. However, if we shift this answer by one to the *right*, we get `8`, the answer to $3 + 5$.

- **Multiplication**
  - Multiplication is similar to addition, except we want to make sure we shift one of the two inputs by 1 to the right before we perform the actual multiplication. So, `(* 3 5)` should produce the assembly

    ```
    mov rax, 6             ; 3 ->  1100 accounting for tag
    mov [rsp - 16], rax
    mov rax, 10            ; 10 -> 1010 accounting for tag
    sar rax, 1             ; 5  ->  101 (remove tagging)
    imul rax, [rsp - 16]   ; 6 * 5 -> 11110000
    ```

    `11110` in binary is `30` in decimal. Shifting this by 1 to the right gives us the desired answer of `15`. The reason why we choose to shift one of the two values to the right by 1 is so we can avoid potential overflow errors.

- **Errors**
  - If we get an error (e.g., a type mismatch error), then from our assembly code, we want to call a *Rust* function that handles the error.
  - For example, we might have something like

    ```
        and rax, 1
        cmp rax, 1      ; if we get a boolean value instead of number
        je error_label  ; jump to the error_label error
                        ; otherwise, fall through label
        ...
    error_label:
        call rust_error
    ```

    with corresponding Rust code

    ```
    fn error() {
        panic!("error");
    }
    ```

1.6 Assembly Review

There are some new assembly commands to know.

- cmp <reg>, <val>: computes <reg> - <val> and sets the appropriate condition codes[2]. This does not modify <reg>.
- <label>: : Sets this line as a label for jumping to later.
- and <reg>, <value>: Performs bitwise AND on <reg> and <value>.
- jmp <label>: Unconditionally jumps to <label>.
- jne <label>: Jumps to <label> if Zero is not set (last cmped values not equal).
- je <label>: Jumps to <label> if Zero is set (last cmped values equal).
- jge <label>: Jumps to <label> if Overflow is the same as Sign (corresponds to >= for last cmp).
- jle <label>: Jumps to <label> if Zero is set or Overflow is not equal to Sign (corresponds to <= for last cmp).
- shl <reg>: Shifts <reg> to the left by 1, filling in least-significant bit with zero.
- sar <reg>: Shifts <reg> to the right by 1, filling in most-significant bit to preserve sign
- shr <reg>: Shifts <reg> to the right by 1, filling in most-significant bit with zero.

[2] The only ones that matter to us are Overflow, Sign, and Zero