CSE 131 Notes

Compiler Construction

Spring 2023

Taught by Professor Joe Politz

Table of Contents

1 Introduction to Compilers

Some examples of compilers we've used in this class include

`gcc`	C to binary
`g++`	C++ to binary
`rustc`	Rust to binary
`javac`	Java to JVM bytecode
`ghc`	Haskell to Haskell magic
`tsc`	TypeScript to JavaScript

Essentially, a compiler takes some program and produces an output program, one that is easier for us to run in some environment. In this course, we're going to create a compiler `ucsdc` that takes in `snek` files and will produce `x86_64` binaries.

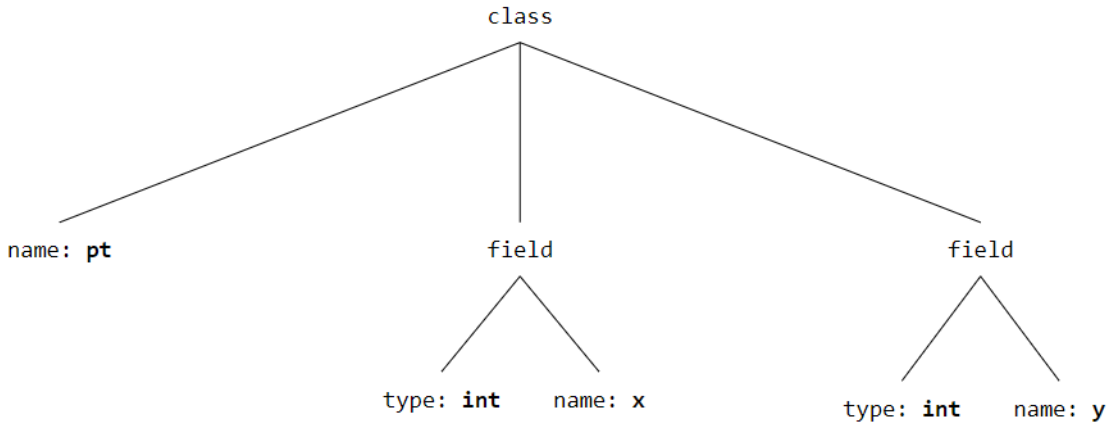
To be more specific, our compiler should do the following: given a `.snek` text file,

- parse[1] the text into an abstract syntax tree (AST),

(Example.) For example, consider the following Java class.

```
class Pt {
    int x;
    int y;
}
```

Its AST representation might look like



- and either
  - generates assembly (or some other output program), or
  - generate an error message.

We are not interested in generating the most optimized assembly, just that it works. Most of the time spent in this course will be on the code generating part (i.e., from AST to assembly). Towards the *end* of the course, we'll work with libraries/runtime.

[1] Known to be very boring.

1.1 S-Expressions

We'll be using s-expressions to represent our program's source code. That is, each `snek` file will contain s-expressions. S-expressions are defined as either

- An **atom** of the form x, or
- An **expression** of the form $(x \quad y)$, where x and y are s-expressions.

For example, `(sub1 2)` is an expression with two atoms, `sub1` and `2`.

1.1.1 The `sexp` Crate

Most programming language will have a parser for s-expressions. In Rust, we have the `sexp` crate. This crate has the following `enums`[2]:

- A `Sexp` enum, representing either an `Atom` or a vector of s-expressions. Vectors of s-expressions will contain two elements (since an expression has the form $(x \quad y)$, which has two expressions).

```
pub enum Sexp {
    Atom(Atom),
    List(Vec<Sexp>),
}
```

- An `Atom` enum, representing one of three different types of atoms: a `String`, `i64` (64-bit signed integer), or `f64` (double-precision float).

```
pub enum Atom {
    S(String),
    I(i64),
    F(f64)
}
```

(Exercise.) Why is `Vec<Box<Sexp>>` or `Box<Vec<Sexp>>` not used in the `enum` definitions above?

Remember that the reason why something like

```
enum Expr {
    Num(i32),
    Add1(Expr),
    Sub1(Expr)
}
```

wouldn't work is because `Expr`, as a recursive type, could have infinite size. That is, we could nest *many* `Expr`s, and since this value could theoretically continue infinitely, so we don't know how much space this recursive type needs. However, `Box<T>` is a pointer type that allocates memory on the heap. `Box<T>` has a *fixed* size, so Rust is fine if we have `Box<Expr>`.

The reason why we *don't* need `Vec<Box<Sexpr>>` or `Box<Vec<Sexpr>>` in the above `enums` is because `Vec<T>` allocates to the heap (when you add any elements; a vector created initially with no elements does not allocate). In other words, think of `Vec<T>` as being a resizable `Box<T>`. More accurately, a `Vec<T>` is a fixed-size type with a reference to variable-sized heap data.

[2] In Rust, `enums` are algebraic data types.

(Exercise.) Represent the following s-expression in Rust:

```
(sub1 (sub1 (add1 73)))
```

First, the s-expression itself is roughly similar to a tree, where each *atom* is a leaf node and each *list* is another s-expression. In any case, we can roughly structure the above s-expression like so:

```
(               // List
    sub1            // Atom
    (               // List
        sub1            // Atom
        (               // List
            add1            // Atom
            73              // Atom
        )
    )
)
```

In other words, we have a list whose elements are an *atom* and another *list*. With this in mind, the Rust form is

```
List(vec![
    Atom(S("sub1")),
    List(vec![
        Atom(S("sub1")),
        List(vec![
            Atom(S("add1")),
            Atom(I(73))
        ])
    ])
])
```

1.2 From S-Expressions to Rust Code

Given some s-expression string, we can use the above crate to convert the s-expression into a Rust object. However, this Rust object itself doesn't give us much information aside from how the s-expression is structured. We want to turn this Rust object into another Rust object representing the actual expressions (i.e., an abstract syntax tree). We can use the Rust structure to represent our code:

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

So, given the `Sexp`, our s-expression representation in Rust, we can use the following function to parse this representation into an abstract syntax tree:

```
fn parse_expr(s : &Sexp) -> Expr {
    match s {
        Sexp::Atom(I(n)) =>
            Expr::Num(i32::try_from(*n).unwrap()),
            Sexp::List(vec) =>
                match &vec[..] {
                    [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))),
                    _ => panic!("parse error")
                },
            _ => panic!("parse error")
        }
}
```

(Exercise.) Convert the s-expression representation in Rust of `(sub1 (sub1 (add1 73)))` to an `Expr` object.

From the code above, our object might look like:

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

1.3 From AST to Assembly

How do we convert our AST to assembly? We can use the following code to do this.

```
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"
    }
}
```

(Exercise.) Convert `(sub1 (sub1 (add1 73)))` into assembly.

The above code would produce the following assembly.

```
    mov rax, 73
    add rax, 1
    sub rax, 1
    sub rax, 1
```

2 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

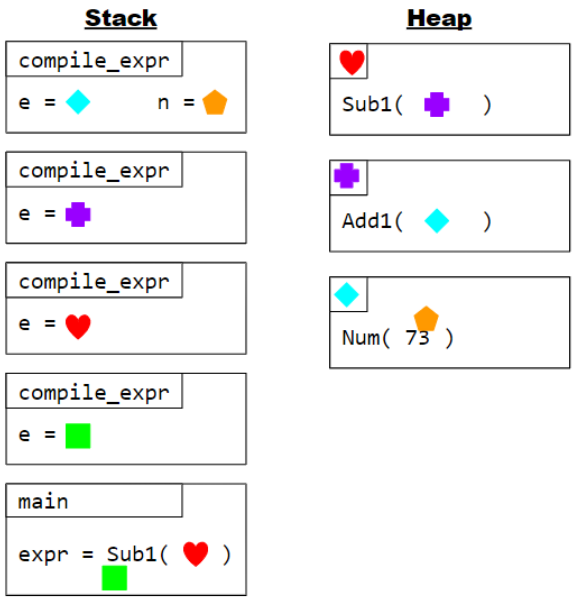
```
format!("mov rax, {}", *n)
```

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 `Enum`s 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`).



2.1 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>),
    }
```

2.1.1 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 `+`.

2.1.2 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.

2.1.3 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:

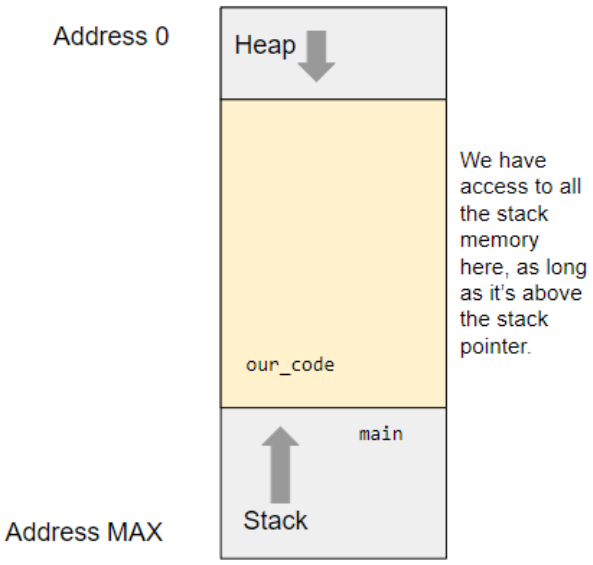
```
fn compile_expr(e: &Expr, si: i32) -> String {
    match e {
        Expr::Num(n) => format!("mov rax, {}", *n),
        Expr::Add1(subexpr) => compile_expr(subexpr, si) + "\nadd rax, 1",
        Expr::Sub1(subexpr) => compile_expr(subexpr, si) + "\nsub rax, 1",
        Expr::Plus(e1, e2) => {
            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[3]. 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



- `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.

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?

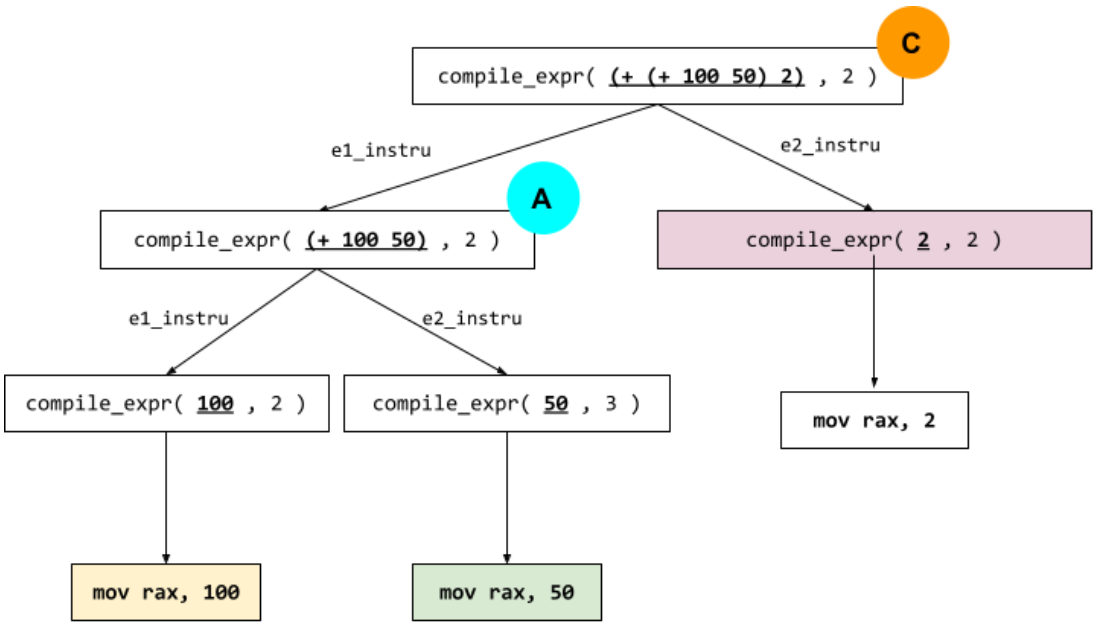
```
(+ (+ 100 50) 2)
```

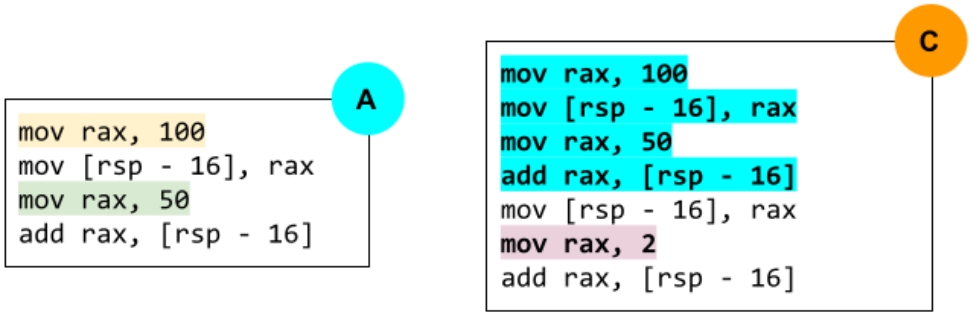
[3]All the operating system will do is, if we somehow use the entire stack space, we'll probably end up with a segfault.

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





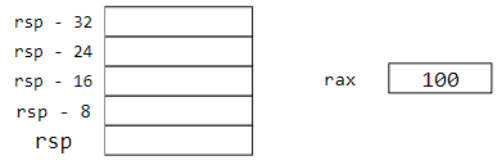
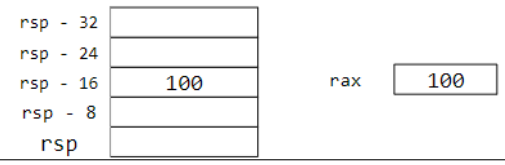
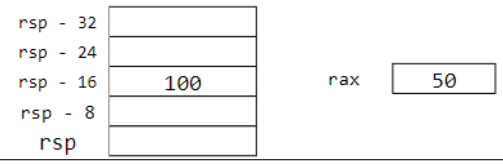
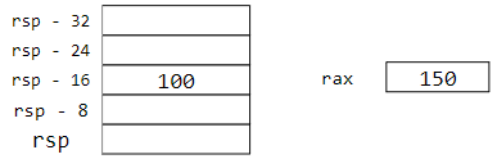
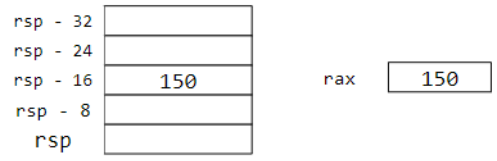
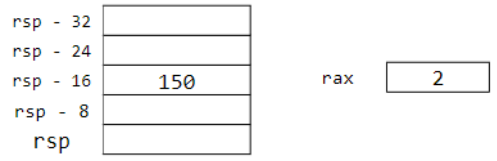
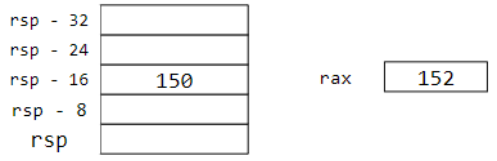
Note that the values passed in the function calls are intended to make things more clear, and will not compile otherwise.

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

3 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?

3.1 Reviewing Assembly Instructions

There are three assembly instructions you should know.

- `mov <reg>, <value>`: moves `<value>` into `<reg>`. Note that `<value>` can either be a constant, another `reg`, or a memory location (e.g., `[rsp - X]`).
- `mov [<reg> + <offset>], <value>`: moves `<value>` into memory at address `<reg> + <offset>`. The `<value>` can be a constant or `reg`, but cannot be another memory location.
- `add <reg>, <value>`: adds `<value>` to the value in `<reg>`, and stores the result in `<reg>`. The `<value>` can either be a constant, register, or memory location.

3.2 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.

3.3 Modifying the Parser

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

3.3.1 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"),
    }
}
```

3.3.2 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()),
```

3.3.3 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"),
    }
}
```

3.4 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.

3.4.1 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()),
```

3.4.2 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)
}
```

3.4.3 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()),
    }
}
```

4 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!
        | ...
*)
```

4.1 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.

4.2 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[4].

4.3 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`

[4]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?

4.4 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
```

4.5 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");
}
```

4.6 Assembly Review

There are some new assembly commands to know.

- `cmp <reg>, <val>`: computes `<reg> - <val>` and sets the appropriate condition codes[5]. 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 `cmp`ed values not equal).
- `je <label>`: Jumps to `<label>` if Zero is set (last `cmp`ed 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.

4.7 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.

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

```
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}");
    }
}
```

4.8 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!
}
```

4.9 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.

4.10 Implementing the Compiler

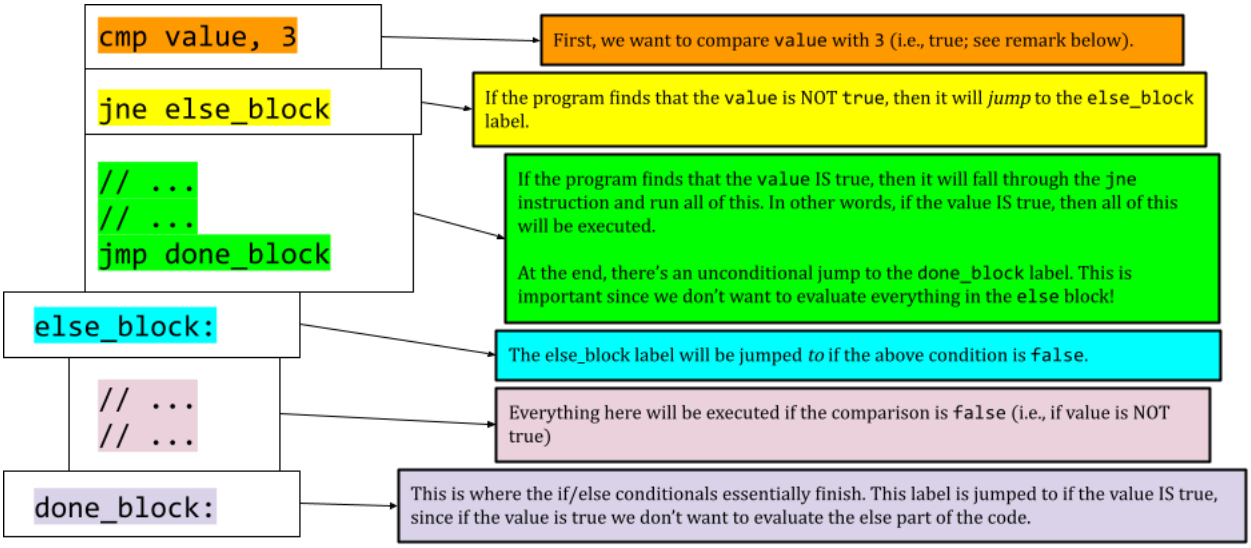
Before we start, one adjustment we need to make to account for the tagging system that we've discussed is to shift integers one bit to the left. That is, when we have an integer, we need to shift it one bit to the left so the least significant bit can be used as the tag bit.

```
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`).

4.10.1 Duplicate Labels

All `if`-expressions will follow the same pattern[6]. 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:
    // ...
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.

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

4.10.2 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}:
        ")
    }
    ...
}
```

4.10.3 `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[7] 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.

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

5 An Overview of Loops, Set, Break, Input, Blocks, and Errors

In this section, we'll go over a high-level overview of some additional features.

5.1 Loops & Break

Loop and break expressions are relatively simple; they look something like

```
(loop <expr>)
(break <expr>)
```

At its core, a `loop` will repeat an expression over and over again until it encounters a `break`, in which case the loop terminates.

5.1.1 Assembly Representation

The assembly representation of loops is straightforward:

```
loop:
                ; body of loop
    jmp loop
done:
```

Here, we defined two labels:

- `loop`, indicating the beginning of the loop.
- `done`, indicating the end of the loop (where we should "break" out).

The idea is that, as long as we aren't breaking out, we will unconditionally jump back to `loop`. If we do want to incorporate a `break` statement, we can add a jump to that label.

5.1.2 Labeling

As is the case with `if`-statements, we can have many loops! So, we need to create a unique label for each loop. We can use the `new_label` function from the last section to do this for us.

5.1.3 Implementing Break

To implement `break`, the idea is for the compiler to include an additional argument. We can call this argument `loop_label`, which will be an `Option<String>` (recall that an `Option<T>` will either be a `Some(T)` or `None`, indicating some or no value, respectively).

Before we compile the expression associated with the loop, we need to create a unique label. Once we create this label, we can compile the expression, passing that label as our argument for `loop_label`. Because of the recursive nature of the compile function, if we end up inside another loop, compiling its associated expression will result in another label for *that* function call, but not for the current function call. In that sense, we don't need to worry about the possibility of overwriting the break labels.

5.2 Set

Set is relatively straightforward: it's analogous to reassignment in most other programming languages. Its syntax looks like

```
(set! <name> <expr>)
```

Here, we're assigning the result of evaluating `<expr>` to the identifier, `<name>`. If the identifier doesn't exist, an error should be thrown.

5.3 Blocks

Blocks are just a way of writing more than one statement for an expression. Syntactically, they look like

```
(block <expr>+)
```

In other words, it takes one or more expressions, and then runs each expression in the order that they appear.

Rust	Our Language

`if x > 10 {`
`  x = x + y + z;`
`  y = x - z;`
`  z = z + 5;`
`} else {`
`  x += 10;`
`}`
	`(if (> x 10)`
	`  (block`
	`  (set! x (+ x (+ y z)))`
	`  (set! y (- x z))`
	`  (set! z (+ z 5))`
	`  )`
	`  (set! x (+ x 10)))`

5.4 Handling Inputs

Our programming language is now expected to take a single input, which can either be a number or boolean value. Syntactically, the command is just

```
input
```

In order to handle any input, we need to think about the runtime and, more importantly, the role that `start.rs` plays (recall that `start.rs` is how we call into our assembly code.)

```
fn parse_input(input: &str) -> i64 {
    if input == "true" {
        0b11
    } else if input == "false" {
        0b01
    } else if let Ok(val) = input.parse::<i64>() {
        (val << 1) as u64
    } else {
        panic!("unsupported input: `{}`", input);
    }
}

fn main() {
    let args: Vec<String> = env::args().collect();
    // This is our single input, which by default will be
    // "false" if none are provided
    let input = if args.len() == 2 { &args[1] } else { "false" };
    // Call our assembly code with the given input.
    let i: i64 = unsafe { our_code_starts_here(parse_input(&input)) };
    // Finally, determine what was returned to us.
    if i & 1 == 0 {
        // Number
        println!("{}", i >> 1);
    } else {
        // Boolean
        println!("{}", i >> 1 == 1);
    }
}
```

Note that our input will be in the register `rdi`. So, we can modify the compiler to simply move `rdi` into `rax` before using it.

```
match e {
    ...
    Expr::Id(s) if s == "input" => "mov rax, rdi"
}
```

That's it!

5.5 Errors

Recall that, in the previous sections, we didn't want things like `(+ true 1)` to work. This should cause a *runtime error*. So, how do we invoke a runtime error from our assembly code?

- First, we want to create a function in Rust that our assembly code will call.

  ```
  #[export_name = "\x01snek_error"]
  pub extern "C" fn snek_error(errcode: i64) {
      eprintln!("error code {errcode} received.");
      std::process::exit(1);
  }
  ```

- Next, in our assembly header, we want to define this function as an external function.

  ```
  section .text
  global our_code_starts_here
  extern snek_error               ; right here!
  ```

- From there, we can define a `throw_error` label which will handle calling the Rust code. It will look like

  ```
  throw_error:
      mov rdi, 7          ; rdi is the register representing the first arg
                          ; 7 is some error code we made up
      push rsp
      call snek_error     ; calls the snek_error rust function
      ret
  ```

  We'll worry about the `push` and `pop` instruction later.

6 Introduction to Calling Conventions: Print

In the previous section, we discussed throwing errors and how we can call an external (Rust) function from our generated assembly code. Note that the error case is interesting in the sense that it always terminated the program, so we didn't have to worry about coming back to our generated code.

Suppose we want to introduce a `print` expression, which takes an expression and then prints the result out. Like the error handling, the `print` expression will call an external Rust function. However, unlike the error handling, we expect the `print` expression to come back to the generated assembly code *after* calling the external function.

Syntactically, `print` looks like

```
<expr> := ... | input | (print <expr>)
```

6.1 Modifying `start.rs`

Once again, as we need to define a function in Rust to handle printing so our generated assembly code can use it, we need to modify `start.rs` to include this information. Let's define the following function to handle printing:

```
#[no_mangle]
#[export_name = "\01snek_print"]
fn snek_print(val : i64) -> i64 {
    if val == 3 { println!("true"); }
    else if val == 1 { println!("false"); }
    else if val % 2 == 0 { println!("{}", val >> 1); }
    else {
        println!("Unknown value: {}", val);
    }
    return val;
}
```

Note that

- `3` is our representation of `true` (`3` is `0b11` in binary.)
- `1` is our representation of `false` (`1` is `0b01` in binary.)

6.2 Calling Convention Idea

At a high level, when calling a function, we need to do the following:

1. "Remember where" we left off (i.e., recovering the stack).
2. Pass in the appropriate arguments.
3. Call the `snek_print` function.

Regarding "remember where" we left off: remember that our compiler function has a parameter which represents the stack index. This stack index tells us how "deep" we are in the expressions that needed temporary storage/variables. So, whatever stack index we have, we need to set up the call so that it's above the stack index; after the call is done, we need to go back to where the stack index was originally.

6.3 Implementing `print`

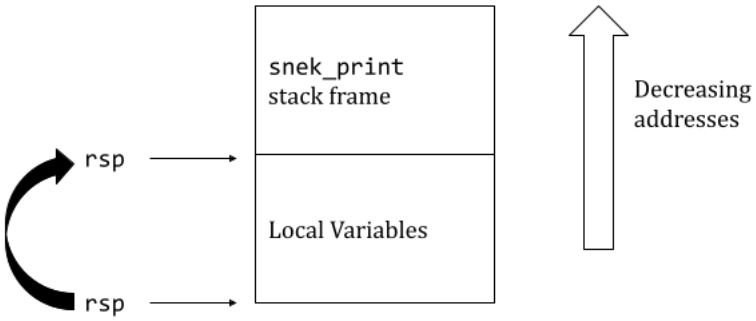
Let's now attempt to implement `print` into our compiler.

6.3.1 Attempt 1

Our initial implementation of `print` might look like

```
match e {
    ...
    | Expr::Id(s) if s == "input" => "mov rax, rdi".to_string(),
    | Expr::Print(val) => {
        let offset = si * 8;
        let v_is = compile_expr(val, si, env, l);
        format!("
            {v_is}
            sub rsp, {offset}
            mov rdi, rax
            call snek_print
            add rdi, {offset}
        ")
    }
}
```

In terms of what's going on, what `sub rsp, {offset}` is doing is moving `rsp`, the stack pointer, so that it's pointing *above* where all the local variables and temporaries are.



So, the stack frame for the `snek_print` is where all the work for this function will happen. So, being its own function, it will rely on `rsp` being "kind of" like at bottom of its stack frame, since it gets its own local variable space whatnot.

(Exercise.) Consider the following code:

```
(block
    (print 37)
    (print input)
)
```

Using our implementation above, what would be printed if `input` was `true`?

This would print `37` twice. To see why, let's consider the generated assembly.

```
mov rax, 74
sub rsp, 16
mov rdi, rax
call snek_print
add rsp, 16
mov rax, rdi
...
```

The issue is in the third line, `mov rdi, rax`. We're overwriting `rdi`, which contains the result of `input`, with a different value! More specifically, we used `rdi` to pass in the argument for `print`, but this means we lose the result of `rdi` when we passed in the argument.

So, we just need to remember to store data from registers somewhere else before we do the call. One thing we can do is store `rdi` somewhere on the stack before we make the function call. To do this, we can make use of `push` and `pop`. In fact, `rdi` is an example of a **caller-saved register**; that is, before we make a function call, we should save this register if we want to restore it afterwards.

6.3.2 Attempt 2

With what we just mentioned in mind, we have

```
match e {
    ...
    | Expr::Id(s) if s == "input" => "mov rax, rdi".to_string(),
    | Expr::Print(val) => {
        let offset = si * 8;
        let v_is = compile_expr(val, si, env, l);
        format!("
            {v_is}
            sub rsp, {offset}
            push rdi            ; added this line
            mov rdi, rax
            call snek_print
            pop rdi             ; added this line
            add rdi, {offset}
        ")
    }
}
```

Remark: Remember that we want `rsp` to point to the spot where the pop is going to be, which is why we call `pop rdi` immediately after the `call` call.

Note that

- `push` pushes a value (register, immediate, etc.) to the stack. `push` will subtract 1 from `rsp`.
- `pop` pops whatever is on top of the stack into a register. `pop` will add 1 to `rsp`.

(Example.) For example,

```
push 17
push 23
pop rax
pop rcx
```

will put `23` into `rax`, and `17` into `rcx`.

So, in terms of what the generated assembly would do, it would

- Push `rsp` into the stack.
- Then, move `rax` into `rsp`.
- Then, call the `snek_print` function.
- Then, put the most recently added value from stack (i.e, the old value of `rsp`) to the register `rsp`.

It just so happens that, in our example here, we only really care about `rdi`. But, if we have other registers that we care about (e.g., we use them a lot or need to save them between expressions), we need to save them.

6.4 Alignment Issues

To summarize some of the things we've said about the x86_64 calling convention[8]:

- Arguments go into `rdi`, and then 6 other registers, and then the stack.
- `[rsp]` (i.e., the value stored at location `rsp`) should be the return pointer.
- 16-byte alignment constraint: `rsp` (mod 16) = 0.

So, in our generated assembly, the third point may cause some problems with the stack pointer. One way we can resolve this is to check what `stack_offset` (mod 16) is, or equivalently `stack_index` (mod 2) is. For the latter, if we have an odd number, we can add 1 to the stack index so we have an even stack index.

[8] These are calling conventions specific to x86_64, and is the reason why we had to do this when calling `snek_print`. However, when we define our own functions in our own language, we can use whatever calling convention we want (including making one up) as long as it's consistent.

7 Calling Conventions

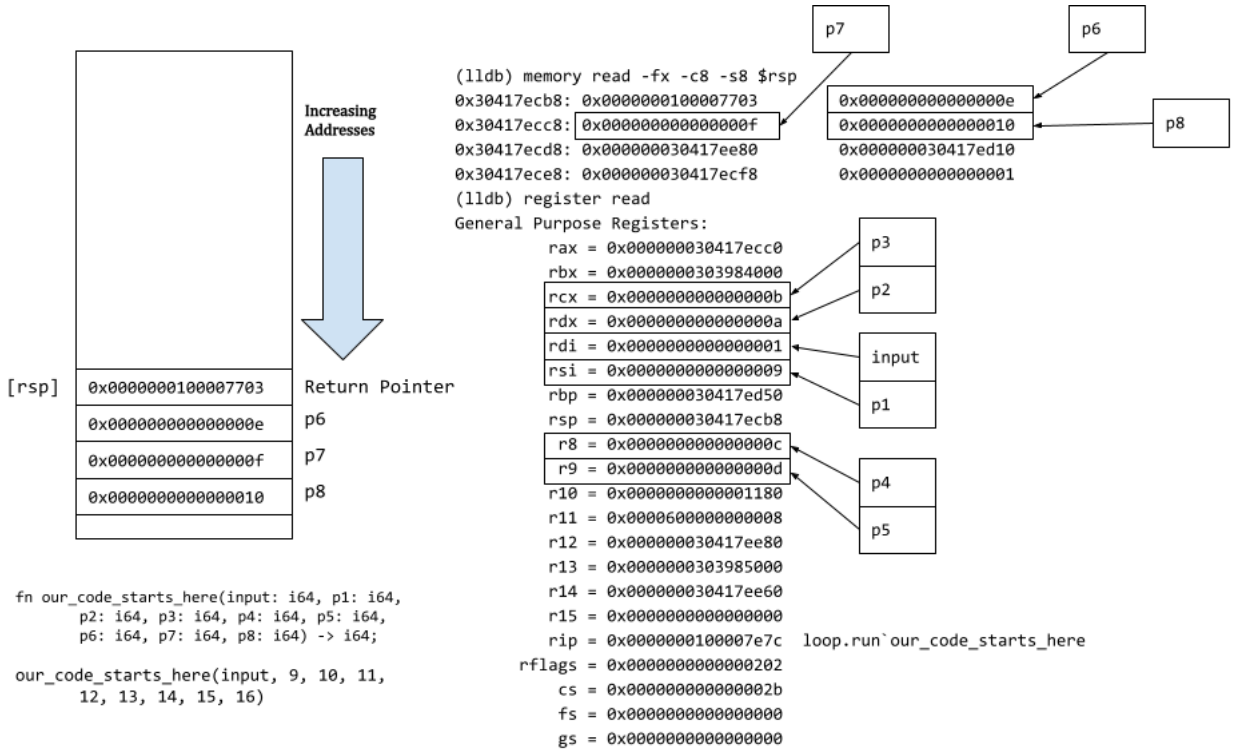
In this section, we'll talk more about calling conventions.

7.1 Argument Conventions in x86_64

Consider the following code (assuming that only this part of the code has changed.)

```
#[link(name = "our_code")]
extern "C" {
#[link_name = "\x01our_code_starts_here"]
    fn our_code_starts_here(input : i64, p1 : i64, p2 : i64, ..., p8 : i64) -> i64;
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let input = parse_arg(&args);
    let i : i64 = unsafe {
        our_code_starts_here(input, 9, 10, 11, 12, 13, 14, 15, 16)
    };
    snek_print(i);
}
```

Here, we added 8 parameters to the `our_code_starts_here` function so we can see how the arguments are represented in memory. Using `rust-lldb`, we can debug this code and, more importantly, see the memory layout of our program.



The main takeaways to get are, in x86_64,

- The first 6 arguments are `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`.
- Any remaining arguments will go in order in increasing addresses **after** `[rsp]`, where `[rsp]` holds the return pointer.

7.2 Functions & Their Conventions for Us

How do we add functions to our programming language? We need to restructure our programming language a bit. First, our program will have the following definition:

```
<prog> := <defn>* <expr>

<defn> := (fun (<name> <name>) <expr>)
    | (fun (<name> <name> <name>) <expr>)

<expr> := ...
    | (<name> <expr>)
    | (<name> <expr> <expr>)
```

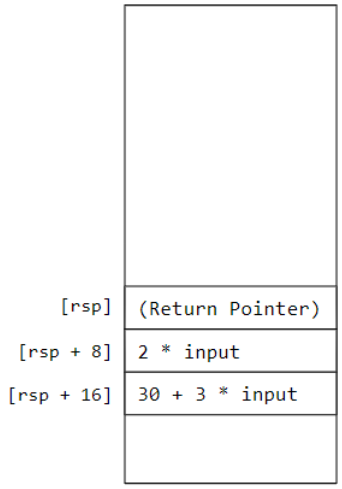
In particular,

- `<prgm>` is what our `snek` file will look like. `<prgm>` is saying that it takes zero or more function definitions, and then the expression. So, past `snek` files had zero function definitions. Now, `snek` files can have more than zero function definitions.
- In our version of `<defn>`,
  - `(fun (<name> <name>) <expr>)` means that the function declaration has a *name*, and then one argument after it, and then the following expression.
  - `(fun (<name> <name> <name>) <expr>)` means that the function declaration has a *name*, and then *two arguments* after the name, and then the following expression.
- In `<expr>`, not only do we have the usual language features (e.g., numbers, identifiers, binary operations, etc.), but we also have support for calling the above functions. Specifically,
  - `(<name> <expr>)` calls the function `<name>` with a single argument.
  - `(<name> <expr> <expr>)` calls the function `<name>` with two arguments.

Consider the following code.

```
(fun (sumsquare x y)
    (+ (* x x) (* y y)))

(sumsquare (* 2 input) (+ 30 (* 3 input)))
```

When `sumsquare` starts, we should expect the memory and stack to look something like:



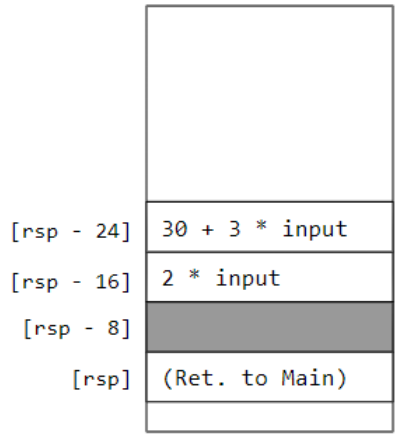
In other words, the value of the first argument, $2 \times$ `input`, when we called the function is in `[rsp + 8]`; likewise, the value of the second argument, $3 \times$ `input` $+ 30$, when we called the function is in `[rsp + 16]`. So, in general, how should we generate functions in our assembly code?

- For each function definition, we will generate a label.
- Arguments will be located in the following places:
  - The value of the first argument is located in `[rsp + 8]`.
  - The value of the second argument is located in `[rsp + 16]`.
  - In general, the value of the ith argument is located in `[rsp + 8(i + 1)]`.

  This is controlled by the environment used for the body.

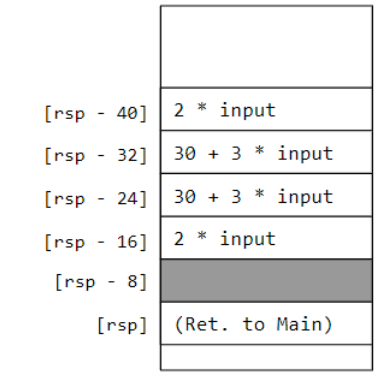
Let's try to sketch out the assembly of the above code, where `sumsquare` is defined and where it is being called, and the corresponding memory model after running it.

```
; [rsp + 8] is x
; [rsp + 16] is y
sumsquare:
    ; TODO
    ret

; basically, our main function
our_code_starts_here:
    ; code for (* 2 input) -> rax
    mov [rsp - 16], rax
    ; code for (+ 30 (* 3 input))
    mov [rsp - 24], rax
```



Notice that the order of the arguments isn't correct, so we can't use the arguments as they appear right here. So, we need to do some more moving.

```
; [rsp + 8] is x
; [rsp + 16] is y
sumsquare:
    ; TODO
    ret

; basically, our main function
our_code_starts_here:
    ; code for (* 2 input) -> rax
    mov [rsp - 16], rax
    ; code for (+ 30 (* 3 input))
    mov [rsp - 24], rax
    ; additional moves for args
    sub rsp, 48 ; 8 * 6
                ; put at "top" of stack
    call sumsquare
    add rsp, 48
```



We also may want to **save** any values that may be caller-saved (e.g., any registers like `rdi`). This should be done before doing any additional moves. Basically, we don't want to overwrite anything from a previous stack frame.

7.3 Improvements and Approaches

Some small ideas we could think about are:

1. The last argument can avoid copying. This is mainly so we can avoid doing some unnecessary minor work.
2. Evaluate arguments in reverse order (last to first). Then, they are in the right place on the stack.

Both of these run into issues with saving and restoring registers! So, a **big idea** is to **pre-allocate stack frames**[9]. In other words, we move `rsp` at the *start* of a function to accommodate *all* local variables. Then, all references are `[rsp + ___]`. The interesting thing to think about is how much we need to move `rsp` to accommodate all local variables. Once we figure this out, then we can adjust `rsp` as needed.

Introducing, the `depth` function. This calculates the number of local variables (named or unnamed) are in this expression. This is basically just tracking stack indices.

```
fn depth(e: &Expr) -> i32 {
    match e {
        Plus(e1, e2) => {
            max(depth(e1), depth(e2) + 1)
        }
        Let(id, val, body) => {
            max(depth(val), depth(body) + 1)
        }
        ...
    }
}
```

In a `Plus` expression, which has `e1` and `e2`, we're taking the maximum of

- `depth(e1)` (Because we don't increase the stack index when we go into `e1`), and
- `depth(e2) + 1` (Remember that the plus expression makes space for one word, and then reserves that space while it's working on `e2`. This corresponds to how we did `si + 1` in our compiler.)

Analogously, we can say the same for the `Let` expression.

So, going back to compiling `sumsquare`, the first thing we can do is

```
sumsquare:
    sub rsp, depth(body * 8)
    .
    .
    .
    add rsp, depth(body * 8)
    ret
```

Note that all local variables use `[rsp + ___]` in this function. This means less calculations and moving `rsp` at each call.

[9] This is what Rust and C does.

8 Our Calling Convention

We'll now talk more about the calling conventions we aim to use for our compiler, along with functions in general. Recall, from the previous section, that a `snek` program is defined by

```
<prog> := <defn>* <expr>

<defn> := (fun (<name> <name>) <expr>)
    | (fun (<name> <name> <name>) <expr>)

<expr> := ...
    | (<name> <expr>)
    | (<name> <expr> <expr>)
```

where `<defn>` means that our function declarations can either take one argument, or two arguments, respectively.

8.1 Caller-Managed Stack Pointer

An approach for calling functions is to have the caller (i.e., the function that is *calling* a function) manage the stack pointer, `rsp`.

8.1.1 Compiling the Definition

Let's consider the following Rust code:

```
fn compile_definition(d: &Definition, labels: &mut i32) -> String {
    match d {
        Fun1(name, arg, body) => {
            ...
        }
        Fun2(name, arg1, arg2, body) => {
            let body_env = hashmap! {
                arg1.to_string() => -1,
                arg2.to_string() => -2
            };
            let body_is = compile_expr(body, 2, &body_env,
                &String::from(""), labels);
            format!(
                "{name}:
                {body_is}
                ret"
            )
        }
    }
}
```

This function is designed to compile a function declaration, specifically a function with one and two arguments. Some things to think about:

- Compiling a function is straightforward with this calling convention. All there is to the actual function is
  - The label (perhaps, the name of the function).
  - The body of the function.
  - And then, `ret` for returning.

- Note that the *environment* is set up so that negative stack indexes are used. This way, the compiler ends up accessing `[rsp + X]`, i.e., accessing memory downwards as opposed to upwards.
- In this approach, all the work of manipulating `rsp` is done by the **caller**.

8.1.2 Compiling the Function Calls

Now, let's look at the code that's responsible for compiling *function calls*.

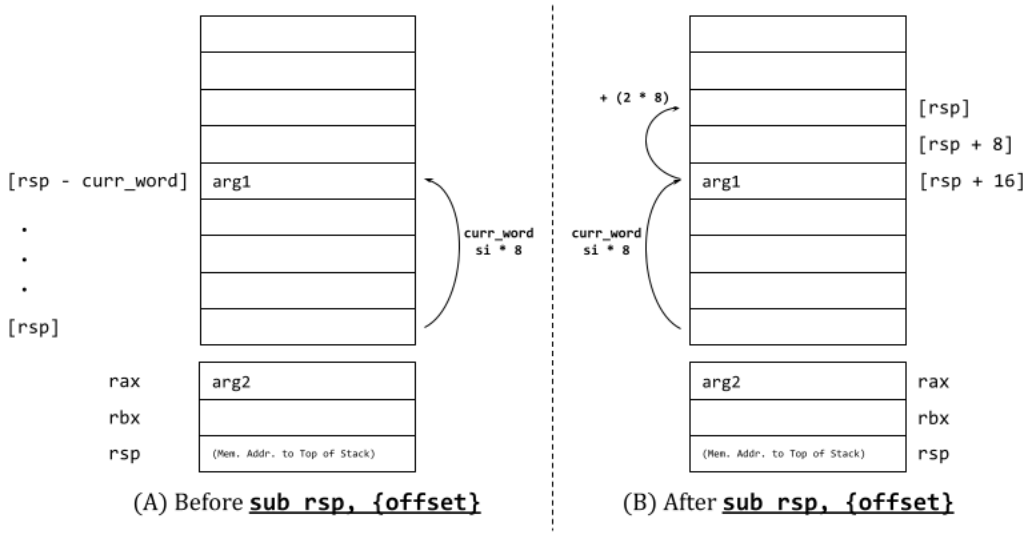
```
fn compile_expr(
    e: &Expr,
    si: i32,
    env: &HashMap<String, i32>,
    brake: &String,
    l: &mut i32,
) -> String {
    match e {
        ...
        Expr::Call2(name, arg1, arg2) => {
            let arg1_is = compile_expr(arg1, si, env, brake, l);
            let arg2_is = compile_expr(arg2, si + 1, env, brake, l);
            let curr_word = si * 8;
            let offset = (si * 8) + (2 * 8);
            // With this setup, the current word will be at [rsp+16],
            // which is where arg1 is stored. We then want to get rdi
            // at [rsp+16], arg2 at [rsp+8], and arg1 at [rsp], then call
            format!(
                "
                {arg1_is}
                mov [rsp-{curr_word}], rax
                {arg2_is}
                sub rsp, {offset}
                mov rbx, [rsp+16]
                mov [rsp], rbx
                mov [rsp+8], rax
                mov [rsp+16], rdi
                call {name}
                mov rdi, [rsp+16]
                add rsp, {offset}
            "
            )
        }
    }
}
```

There are some things we need to make sure:

- We need to make sure we set `rsp` high enough so that it doesn't interfere with any of the temporary variables.
- We also need to make sure we store the variables (arguments) in the right place before we actually call the function. `offset` is defined so that it will be used to move `rsp` above where we are, and then subtracting some more words, so we can make room for `rdi` and **two** arguments.
- Note that, even though we have three items (`rdi` and the two arguments), we only need to move the stack index up by an additional 16 spaces, hence why we're adding `2 * 8`.

8.1.3 Memory Layout

With the above code for function *calling*, let's look at a memory diagram of what's going on prior to calling a function. We know that `curr_word` is defined as `si * 8`.



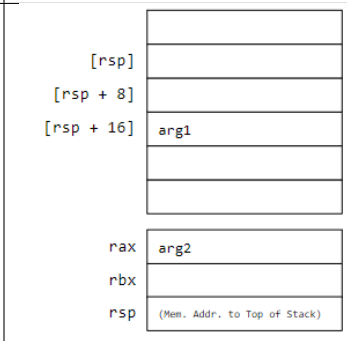
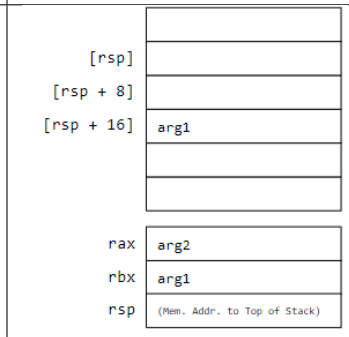
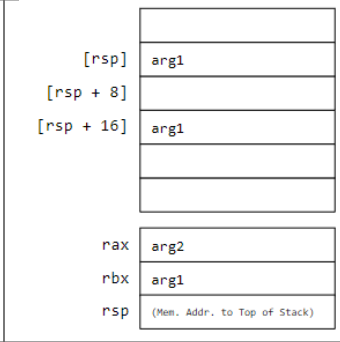
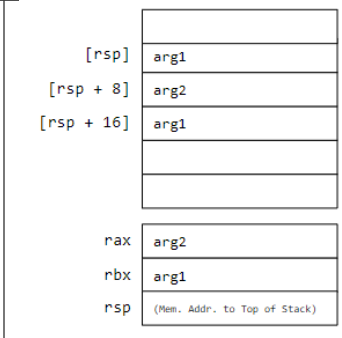
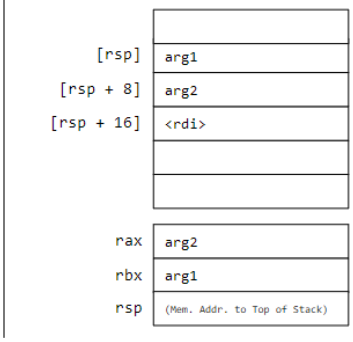
So, what's going on?

- Before we call `sub rsp, {offset}`, i.e., in diagram (A),
  - `[rsp]` is pointing to some initial return pointer (e.g., at a main expression or some function).
  - `[rsp - curr_word]` is where we stored `arg1` in the stack. Equivalently, `arg1` is stored `curr_word` space "above" `rsp`.
  - Remember that the result of `arg2` is stored in `rax`, since that was the last expression that was compiled.
- After we call `sub rsp, {offset}`, i.e., in diagram (B),
  - We moved `[rsp]` up by `(si * 8) + (2 * 8)` spaces, as seen by the two "jumps" in the diagram.

Our goal is to put `rdi` into `[rsp + 16]`, `arg2` into `[rsp + 8]`, and `arg1` into `[rsp]`. To make this happen, we need to move a few things around. That's where the following four assembly instructions,

```
mov rbx, [rsp+16]
mov [rsp], rbx
mov [rsp+8], rax
mov [rsp+16], rdi
```

come from. To see how this works, let's visualize each line.

After Running	Diagram
(Initial)	
`mov rbx, [rsp + 16]`	
`mov [rsp], rbx`	
`mov [rsp+8], rax`	
`mov, [rsp+16], rdi`	

Remarks:

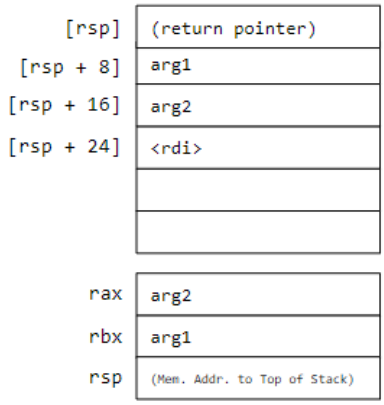
- Remember that the above only works for 2 arguments. For arbitrary arguments, the same idea still holds, but you need to generalize it.

  > Recall: *Before* we executed `sub rsp, ...`, we can imagine that `arg1` is at `[rsp - si * 8]`, `arg2` is at `[rsp - si * 8 - 8]`, `arg3` is at `[rsp - si * 8 - 16]`, and so on until `argN-1` is at `[rsp - si * 8 - 8(N - 1)]`. As usual, `rax` will hold `argN`.

  After we move `[rsp]`, we can expect `argN-1` to be at `[rsp + 16]`, `argN-2` to be at `[rsp + 24]`, and so on, with `arg1` being at `[rsp + 8(N + 1)]`. As usual, `rax` holds `argN`.

- `rdi` is a caller-saved register, hence why we're purposely saving it.
- In this calling convention, as one might have guessed, we're putting everything on the stack.
- While function arguments need positive offsets from `rsp`, **local variables** (temporaries) still use negative offsets from `rsp`.

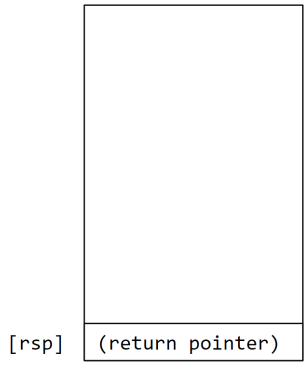
Now, after we do the `call` instruction, the memory diagram looks like



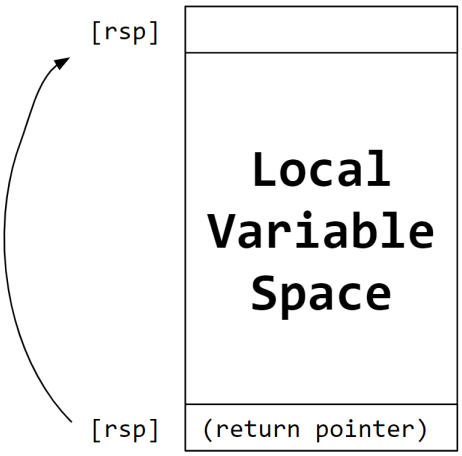
In other words, the `call` instruction will move `rsp` up and write the return pointer into that word. At that point, it's clear that `arg1` and `arg2` are in the correct offsets.

8.2 Callee-Managed Stack Pointer

Suppose, instead, we want to have the function itself manage the stack pointer by moving it sufficiently enough. Then, at the start of a function call, suppose the memory diagram looks like this:



Under this assumption where the callee manages the stack pointer, the first thing the function will do is `sub rsp` so that there's enough space for all local variables. That is, we should move `[rsp]` so that we end up with a memory diagram that looks like



Then, all lookups, including for argument and local variables, will have its location in the stack be positive offsets from `rsp`. There are some advantages to doing this, especially in relation to garbage collection.

8.2.1 The Depth Function

First, we need to know how many local variables are needed in the body of a function so we know how much we need to move `rsp` by. We can use the `depth` function to calculate the maximum stack index needed in an expression to store all local variables and temporaries.

Using the power of ChatGPT, along with some corrections, we have the following implementation:

```
fn depth(e: &Expr) -> i32 {
    match e {
        Expr::Num(_) => 0,
        Expr::True => 0,
        Expr::False => 0,
        Expr::Add1(expr) => depth(expr),
        Expr::Plus(expr1, expr2) => depth(expr1).max(depth(expr2) + 1),
        Expr::Let(_, expr1, expr2) => depth(expr1).max(depth(expr2) + 1),
        Expr::Id(_) => 0,
        Expr::Eq(expr1, expr2) => depth(expr1).max(depth(expr2) + 1),
        Expr::If(expr1, expr2, expr3) => {
            depth(expr1).max(depth(expr2)).max(depth(expr3))
        },
        Expr::Loop(expr) => depth(expr),
        Expr::Block(exprs) => exprs.iter().map(|expr| depth(expr)).max().unwrap_or(0),
        Expr::Break(expr) => depth(expr),
        Expr::Print(expr) => depth(expr),
        Expr::Set(_, expr) => depth(expr),
        Expr::Call1(_, expr) => depth(expr),
        Expr::Call2(_, expr1, expr2) => depth(expr1).max(depth(expr2) + 1),
    }
}
```

8.2.2 Compiling the Definition

With this in mind, we have

```
    fn compile_definition(d: &Definition, labels: &mut i32) -> String {
        match d {
```

```
            Fun1(name, arg, body) => {
                ...
            }
            Fun2(name, arg1, arg2, body) => {
                let depth = depth(body);
                let offset = depth * 8;
                let body_env = hashmap! {
                    arg1.to_string() => depth + 1,
                    arg2.to_string() => depth + 2
                };
                let body_is = compile_expr(body, 0, &body_env,
                    &String::from(""), labels);
                format!(
                    "{name}:
                    sub rsp, {offset}
                    {body_is}
                    add rsp, {offset}
                    ret"
                )
            }
        }
    }
```

We also need to do the same thing with the main expression (the main program):

```
sub rsp, {offset}
{main}
add rsp, {offset}
```

where `offset` is defined by:

```
let depth = depth(&p.main); // p.main -> main program
let offset = depth * 8;
```

8.2.3 Changing Offsets in Code

Recall how, before this section, any offsets we used were negative offsets (e.g., `mov [rsp - {offset}], rax`). With this change, we now can use **positive offsets** (e.g., `mov [rsp + {offset}], rax`). This scheme is similar to what most compilers like Rust, `g++`, and so on use.

9 Introduction to Recursion

In this section, we'll talk about recursion. Note that, in our examples, we'll assume that the callee manages (i.e., moves) the stack pointer. In particular, this means everything will have a positive offset from `rsp`.

9.1 Recursive Sum Example

Let's consider the following code:

```
(fun (sumrec num)
    (if (= num 0)
        0
        (+ num (sumrec (+ num -1)))
    )
)
```

This program simply performs $1+2+3+\ldots+$`num`. The generated assembly would look something like what is shown below.

```
sumrec:
    sub rsp, 16
    mov rax, [rsp + 24]
    ... if (= num 0)
    cmp rax, 1
    je ifelse_1
         mov rax, 0
         jmp ifend_0
    ifelse_1:
         ... put temp num on stack for LHS
         mov [rsp + 0], rax
         mov rax, [rsp + 24]
         ... (+ num -1) stored in rax
         ... now do 1-arg calling conv
         sub rsp, 16
         mov [rsp], rax
         mov [rsp+8], rdi
         call sumrec
         mov rdi, [rsp+8]
         add rsp, 16
         ... do addition on the waiting num ...
         add rax, [rsp + 0]
    ifend_0:

    add rsp, 16
    ret
```

Note that only relevant assembly is shown. Some things to point out:

- In the second assembly line, `sub rsp, 16`, the `16` is the *depth* that we calculated.
- In the lines before the recursive call, i.e.,

```
sub rsp, 16
mov [rsp], rax
mov [rsp+8], rdi
call sumrec
```

we're moving the arguments into the correct position in memory so the recursive call can make use of them.

- When we run a `call` instruction, `rsp` is moved up one word and the return pointer to the next line of instruction (program counter) is put in that location in memory (where `rsp` is pointing to).

To see how the memory looks when each line of assembly is executed, see `Lec12Trace.pdf`.

9.2 Second Recursive Sum Example

Let's rewrite the recursive sum example a bit.

```
(fun (sumrec num sofar)
    (if (= num 0)
        sofar
        (sumrec (+ num -1) (+ sofar num))
    )
)
```

The generated assembly might look like

```
sumrec:
    sub rsp, 16

    mov rax, [rsp + 24]
    mov [rsp + 0], rax
    ... if (= num 0)
    cmp rax, 1
    je ifelse_1
        mov rax, [rsp + 32]
        jmp ifend_0
    ifelse_1:
        mov rax, [rsp + 24]
        ... add -1 to num, store on stack as tmp ...
        mov [rsp + 0], rax

        mov rax, [rsp + 32]
        ... add sofar to num, store in rax ...
        add rax, [rsp + 8]

        ... 2-arg calling convention from class ...
        sub rsp, 24
        mov rbx, [rsp+24]
        mov [rsp], rbx
        mov [rsp+8], rax
        mov [rsp+16], rdi
        call sumrec             ; (A)
        mov rdi, [rsp+16]       ; (B)
        add rsp, 24             ; (C)

    ifend_0:
    add rsp, 16                 ; (C)
    ret                         ; (D)
```

An interesting thing to note is that, after reaching the base case, there's no additional calculation that needs to be made. In particular, the steps after returning is

(a) Move `rsp` back. Remember that, after `call` is done (i.e., when `ret` is executed), `rsp` is moved back one word.

(b) Restore `rdi`.

(c) Move `rsp` back more.

(d) Return!

No local variables or arguments were accessed.

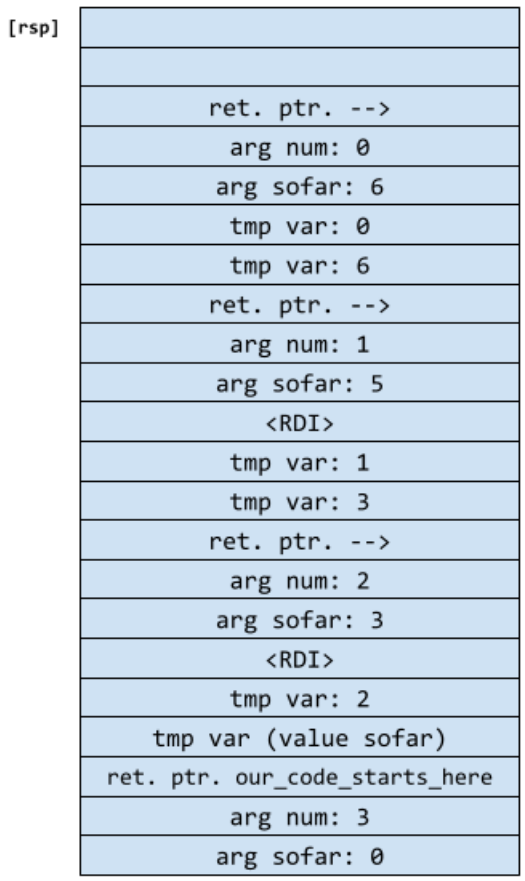
10 Introduction to Tail Recursion

Let's consider the following function:

```
(fun (sumrec num sofar)
    (if (= num 0)
        sofar
        (sumrec (+ num -1) (+ sofar num))
    )
)
```

This function is in tail position: that is, after the recursive call, we don't need to do any additional computations. The assembly representation, is shown to the left. On the right, the stack frame when `(subrec 3 0)` is evaluated is shown, at the point when the base case is about to be executed.

```
sumrec:
    sub rsp, 16

    mov rax, [rsp + 24]
    mov [rsp + 0], rax
    ... if (= num 0)
    cmp rax, 1
    je ifelse_1
        mov rax, [rsp + 32]
        jmp ifend_0
    ifelse_1:
        mov rax, [rsp + 24]
        ... add -1 to num,
        ... store on stack as tmp
        mov [rsp + 0], rax

        mov rax, [rsp + 32]
        ... add sofar to num,
        ... store in rax
        add rax, [rsp + 8]

        ... 2-arg calling
        ... convention from class
        sub rsp, 24
        mov rbx, [rsp+24]
        mov [rsp], rbx
        mov [rsp+8], rax
        mov [rsp+16], rdi
        call sumrec
    --> mov rdi, [rsp+16]
        add rsp, 24

    ifend_0:
    add rsp, 16
    ret
```



(Exercise.) What will `RAX` be after each return pointer?

`RAX` will be **6**. There are two ways you can tell:

- In the code itself, the base case just returns the `sofar` argument. Intuitively, this means that this function should return 6 for all subcalls, implying `rax` has 6 at every return pointer.
- Additionally, when looking at the generated assembly, notice the `move rax, [rsp + 32]` instruction. `[rsp + 32]` points to the value `sofar`, 6. Pair this with when we look at the instructions after the `call` instruction (which is where the program returns to after `ret` is executed), notice that there's no additional move instructions to `rax`.

Remark: There are several things to notice:

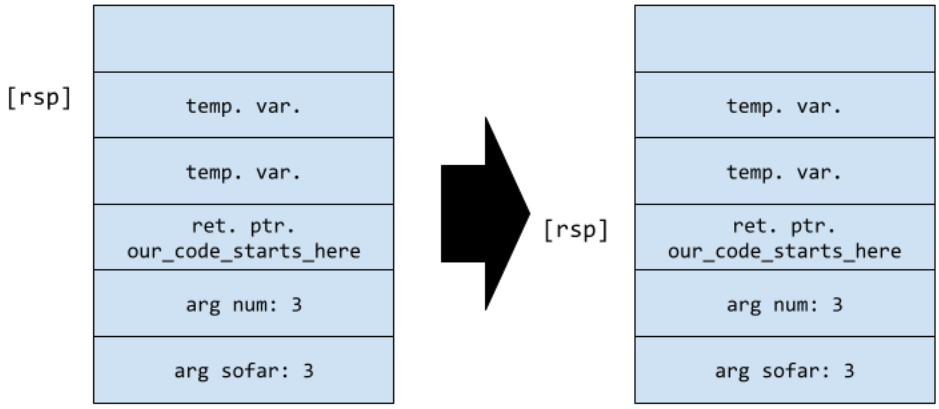
- there's no use of local variables or arguments following the `call` instruction.
- there are no changes to `rax` after the `call` instruction.

These observations means that we can reuse the space that we set up for the function call to perform all operations. In other words, we can use one stack frame's worth of space for the *entire* computation! Basically, we're using more space than we need.

10.1 Restructuring the Assembly

Instead of the `call` instruction above, what should we do to replace the instructions so that the new instructions overwrite the current arguments with the new arguments and "re-use" the stack frame?

We can preemptively add to `rsp` to "undo" the `sub` at the beginning of the function. That is,

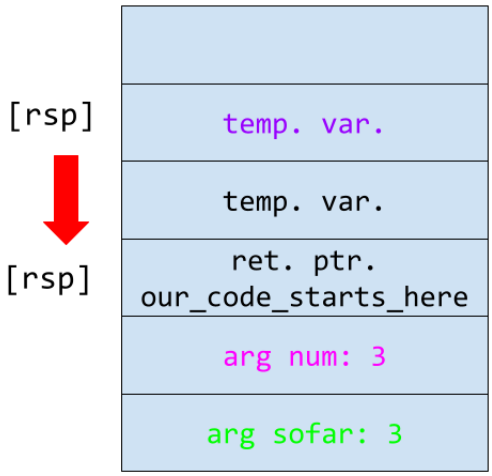


So, we can write the following assembly:

```
add rsp, 16
mov rbx, [rsp - 16]
mov [rsp + 8], rbx
mov [rsp + 16], rax
jmp sumrec
```

Notice how we have an *unconditional jump* to `sumrec`, as opposed to a `call` instruction. This is important, because this effectively means we have a **loop**!

```
add rsp, 16
mov rbx, [rsp - 16]
mov [rsp + 8], rbx
mov [rsp + 16], rax
jmp sumrec
```



This process is known as **proper tail calls**. We effectively re-use the stack from when the "last thing" is a function call (what would have been a separate function call with its own stack frame is now a separate "function call" re-using the same stack frame from the current "function call.").

10.2 Tail Call Positions

When is an expression in tail call position? How does the compiler know when it is in tail call position? A few things:

- Anything where we generate instructions that work with the result of a subexpression is not in tail call.
- Essentially, if you work with `rax` or store the value after a recursive call to the compile function, that recursive function call cannot be in tail call position. If you don't do anything with it and rely on the answer being in `rax` after making the recursive call, then it's in tail position.

So, with this in mind:

- `Add1`: not in tail call position (adds 1 to `rax`).
- `Eq`: not in tail call position (needs to store `rax` somewhere before compiling the second expression so we can compare them).
- `Plus`: not in tail call position (needs to store `rax` somewhere so we can add to `rax` later).
- `Let`: the value associated with the binding cannot be in tail position, but the body of the `let`-expression *can* be in tail position (provided that any preceding calls are in tail position as well).
- `If`: the conditional expression cannot be in tail position, but the then/else expressions can be in tail position (provided that any preceding calls are in tail position as well).
- `Block`: only the last expression in a block can be in tail position; the other expressions cannot be in tail position.
- `Break`/`Loop`: they *generally* cannot be in tail call position (there are special cases, though).

In a function call, the arguments cannot be in tail-call position. Note that the body of the expression is where we compile tail call position code or regular code based on whether we're in tail call position.

In other words, at the beginning of compilation, we assume that we're in tail call position. As we compile each expression, we might "lose" that tail call position. We can never get it back once lost in that subexpression.

11 Structured Data: Pairs

In this section, we'll introduce **structured data** to our programming language. In particular, we'll introduce **pairs**, which is essentially a two-element tuple where both elements can be anything – numbers or pairs.

11.1 Pair Expressions

Our language now has the following syntax:

```
expr := ... | (pair <expr> <expr>) | (fst <expr>) | (snd <expr>) | nil
```

Here, `pair` defines a pair of expressions. `fst` and `snd` returns the first and second element of a pair[10], respectively.

(Exercise.) What will the following program evaluate to?

```
(fun (inc lst)
    (if (= lst nil)
        nil
        (pair (+ (fst lst) 1) (inc (snd lst)))
    )
)

(inc (pair 70 (pair 800 nil)))
```

The answer is `(71, (801, nil))`. In this function, we first check if the given pair is `nil`; if it is, return `nil`. Otherwise, we create a new pair where the first expression is just the first element of the original pair incremented by 1, and the second element is the result of recursively calling `inc` on the second element of the original pair.

We can think of the `(snd lst)` as the *rest of the list*.

(Exercise.) What will the following program evaluate to?

```
(fun (sum lst)
    (let (total 0)
        (loop
            (if (= lst nil)
                (break total)
                (block
                    (set! total (+ total (fst lst)))
                    (set! lst (snd lst))
                )
            )
        )
    )
)

(sum (pair 70 (pair 800 nil)))
```

[10] Although `fst` and `snd` takes any expressions, it expects a pair expression.

The answer is 870. This program iterates through each element of the pair, getting its value and adding it to `total`. In particular, if we ran the program, we see that

Expression	(fst lst)	Total
(pair 70 (pair 800 nil))	70	70
(pair 800 nil)	800	870
nil	-	-

11.2 Representing Pairs

Recall that we used a tagging system, where we dedicated one bit, to differentiate numbers and booleans. However, with a new type, we need to rethink the tagging system.

Our tagging system will now consist of the following:

- **Numbers** will still use `0` as its tag value.
- **Booleans** will use `11` as its tag value.
  - `true` will be represented in binary as `111` (7).
  - `false` will be represented in binary as `011` (3).
- **Pairs** will use `01` as its tag value.
- **Nil** will use `1` as its tag value.

With a tagging system in hand, how do we represent pairs themselves? One approach is as follows:

- An idea is to store each of the pair's value as 31-bits. For example, to represent 2 numbers, we would represent the first number as 31 bits, and the next number as another 31 bits, with the tag value being 2 bits.
- **However**, this won't really work if we have nested pairs. For example, if we have a pair with pairs as its element, then how do we represent this?

Another thing we can think about is heap allocation.

11.3 Compiler Design

As implied, we'll have to allocate pair element on the **heap** (we'll need to work with the Rust runtime for this). So, our representation is that the pair's value will be a 62-bit address on the heap.

How do we know *where* to allocate pair elements on the heap? An idea is to dedicate a register that just keeps track of the current heap location. In our class, we'll use `r15` for our purposes. With this in mind, here are a few assumptions we will be making:

- `r15` is expected to keep growing for now; it's not like `rsp` where it can increase or decrease depending on how stack space is used.
- `r15` only refers to available memory, never used memory.
- `r15` will be 16-byte aligned (it will end with `0000`)

With this in mind, how do we modify our compiler to support pairs? A sketch of an implementation we'll use is as follows:

```
Pair(e1, e2) => {
    let fst = compile_expr(e1, ...);
    let snd = compile_expr(e2 ...);
    // e1 will be somewhere in [rsp], e2 in rax
    format!("
        {fst}
        {snd}
        mov [r15 + 8], rax
        mov rbx, [rsp + offset]
        mov [r15], r15
        mov rax, r15
        add rax, 1
        add r15, 16
    ")
}
```

Remarks:

- We should probably first check and see if we have space left before allocating. We didn't do this part yet.
- Note that we move `rax` into `[r15 + 8]` (and not `[r15]`) because `rax` has the value of the *second* element of the pair, not the first.
- `mov rax, r15` and `add rax, 1` is designed to put the location in heap of the pair's values into `rax` and then add 1 to `rax` for tagging purposes. Note that we can add `1` to `rax` like this because we know that `r15` will end with `0000` (this is one of the assumptions we made).
- `add r15, 16` moves `r15` by 2 words, thus ensuring that it's always pointing to free memory in the heap.

As one might have suspected, once we execute a pair, it should return the memory location to that pair.

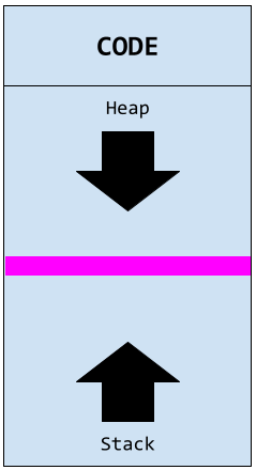
11.4 Heap Allocation

We actually have two things we need to do here:

- We need to create some sort of a *heap* that our compiler can use to store pair values. Once we create
- We need to put the heap pointer into `r15`.

Let's think about some ideas for how we can set the heap up.

- One idea is to set `r15` to `rsp - {a lot}`. In other words, if we move `rsp` very high up, then we can use `r15` as the "heap" pointer. **However**, let's think about the process layout:



The typical convention is that the heap grows downwards and the stack grows upwards. In around the middle of memory (denoted by the pink rectangle), there's some special addresses where touching them will result in an error (heap overflows and stack overflows). So, if we set `rsp` high enough, then we might either hit that special address, or if we have a depth-heavy recursion[11], or if we somehow point `r15` to the heap space[12], then we'll be in trouble.

- We can also call Rust's equivalent of `malloc`, and use its value[13]. In particular,
  - Call `malloc` for each pair[14], or
  - One *big* `malloc`.

  The suggestion we'll use is similar to having a **big** `malloc`. This has the added benefit of being easy to `free` at the end.

11.5 Modifying Our Rust Code

Now, we need to modify our Rust code to account for these changes.

11.5.1 Modifying the Runtime

In `runtime.rs`, we'll essentially do the following:

```
fn main() {
    let args = ...;
    let input = parse_arg(&args);
    let mut memory = Vec::with_capacity(100_000);       // New!
    let buffer: *mut u64 = memory.as_mut_ptr();         // New!
    let i: i64 = unsafe {
        our_code_starts_here(input, buffer);
    };

    snek_print(i);
}
```

Essentially, we'll create a vector with an initial capacity of `100_000` elements. This will represent our heap. Then, we can get a pointer to that vector, and then pass that pointer into our generated assembly. This means our signature for `our_code_starts_here` will look like:

```
extern "C" {
    #[link_name = "\x01our_code_starts_here"]
    fn our_code_starts_here(input: i64, buffer: *mut u64) -> i64;
}
```

11.5.2 The Generated Assembly

In our generated assembly, we'll have

```
our_code_starts_here:
    mov r15, rsi
    ...
```

Here, `rsi` represents the *second* argument[15].

[11] Since this can hit our proposed "heap" space.
[12] Since this is where Rust might allocate memory, so we might end up overwriting memory that Rust needs.
[13] `malloc` gives us the address to some allocated memory in the heap.
[14] Note that heap size is *unknowable* iin general **statically**.
[15] Recall the x86_64 calling convention.

11.5.3 Printing Values

Finally, we need to adjust the `snek_print` function. In particular, our function now needs to account for the fact that it can either receive

- a number (with tag `0`).
- a boolean (with tag `11`).
- a pair (with tag `01`).
- `nil` (with tag `1`).

With this in mind, we have

```
fn snek_str(val: i64) -> String {
    if val == 7 { "true".to_owned()}
    else if val == 3 { "false".to_owned() }
    else if val % 2 == 0 { format!("{}", val >> 1) }
    else if val == 1 { "nil".to_owned() }
    else if val & 1 == 1 {
        let addr = (val - 1) as *const i64;
        let fst = unsafe { *addr };
        let snd = unsafe { *addr.offset(1) };
        format!("(pair {} {})", snek_str(fst), snek_str(snd))
    } else { format!("unknown value: {val}") }
}

#[export_name = "\x01snek_print"]
fn snek_print(val: i64) -> i64 {
    println!("{}", snek_str(val));
    val
}
```

Note that `offset` is used so we don't need to do direct pointer arithmetic[16]. Note that we'll need to do some work printing parentheses and whatnot.

11.6 Memory Representation

Suppose we have the following program:

```
(pair 5 (pair 6 (pair 7 nil)))
```

The way we compile this is to first compile the left-most part of the pair (`5`), and then the right-most part of the pair (the rest of the pair, in this case). In particular,

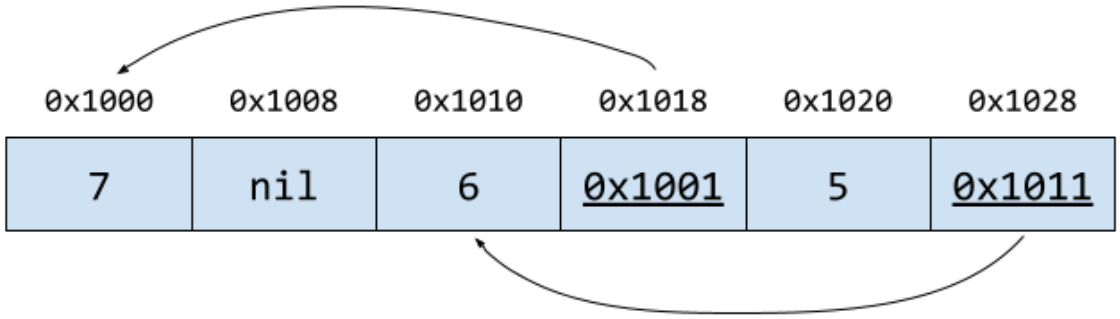
- The instructions for any nested expressions get evaluated before we move those values onto the heap for the current pair.

  > In this program's case, we evaluate `5` first, and then `6`, and then `7`. We put all of them on the stack first. *Then*, we put `7` onto the heap first (while `6` and `5` wait).

- The first pair that gets allocated is the innermost pair.

If we assume left-to-right evaluation order, then `7` goes on the heap first. In this sense, allocation happens inside-out. In any case, in the heap, we expect the memory layout to look like

[16] We're looking at the memory location directly next to the current memory location since pairs are contiguous.



Remarks:

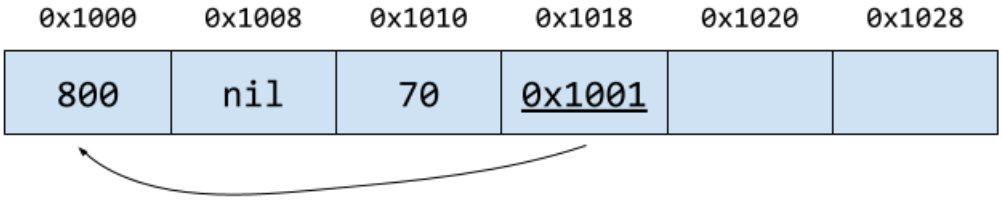
- We have `0x1001` and `0x1011` (instead of `0x1000` and `0x1010`) because we use `0x01` as the tag value for the pair.
- Note that, in our implementation of the compiler, we're actually going to store the integer multiplied by 2 (e.g., 14, 12, and 10, respectively), since in memory we're storing the *tagged* value. In this example, we're just showing the integers as is (with no tagging).
- This program would evaluate to `0x1021`, since this is the memory address to the first element in the pair.

> Another way to think about this is as follows: if we wanted to translate this program into something like Python, syntactically this would look like
>
> ```
> p1 = (7, nil);
> p2 = (6, p1);
> p3 = (5. p2);
> ```
>
> We had to *allocate* memory for `p1` (the innermost pair), and then allocate memory for `p2`, and then finally for `p3`.

Let's now suppose we have the following program:

```
(fun (inc lst)
    (if (= lst nil)
        nil
        (pair (+ (fst lst) 1) (inc (snd lst)))
    )
)

(inc (pair 70 (pair 800 nil)))
```

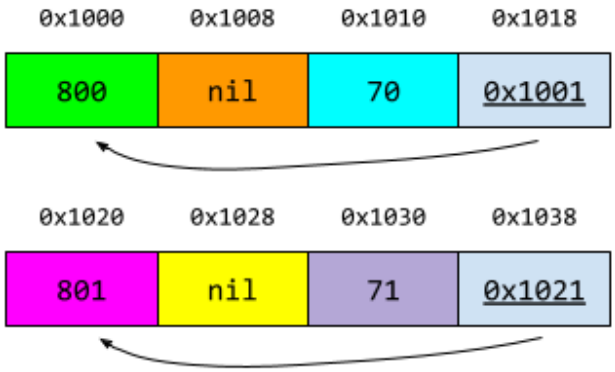
The heap diagram might look something like



Here, the result of `(pair 70 (pair 800 nil))` is `0x1011`, so `0x1011` is passed into the `inc` function. If we look at the function itself, we can see that the function itself will return the same pair, pair, nil structure.

```
(fun (inc lst)
    (if (= lst nil)
        nil
        (pair (+ (fst lst) 1) (inc (snd lst)))
    )
)

(inc (pair 70 (pair 800 nil)))
```

Result
(pair 71 (pair 801 nil))



And, in this exmaple. this function would return `0x1031`, the address to the newly created pair.

11.7 Equality

Let's consider the following program:

```
(let (point1 (pair 6 5))
    (let (point2 (pair 6 5))
        (block
            (print (= point1 point1))                       // A
            (print (= point1 point2))                       // B
            (let (pointpair1 (pair point1 point2))
                (let (pointpair2 (pair point1 point2))
                    (block
                        (print (= pointpair2 pointpair2))   // C
                        (print (= pointpair1 pointpair2))   // D
                    )
                )
            )
        )
    )
)
```

We now need to decide what this program should print. More specifically, however, we need to figure out how equality of pairs will work. This introduces two types of equalities:

- **Reference equality:** are the two operands referring to the same memory address? For example, in Python, this is `==`.
- **Structural equality:** are the two operands equal when considering their structures? For example, in Python, this is `is`.

With this in mind, we have

Statement	Reference Equality	Structural Equality
A	`true`	`true`
B	`false`	`true`
C	`true`	`true`
D	`false`	`true`

Note statements (C) and (D); if we want structural equality, we probably want to do *recursive structural equality!*

11.8 Revisiting Print

Let's suppose we have the following program:

```
(let (p (pair 1 2))
    (block
        (setfst! p p)
        (print p)
    )
)
```

One thing we should note is that we have a **cycle** in the sense that the pair is referring to itself. Therefore, if we tried to `print` the pair, we would end up with infinite recursion since we would constantly recurse through the first element of the pair.

To fix this, we should consider checking if we've *seen* the pair before. If we've seen it, we can print something indicating that a cycle is detected. Otherwise, we can print out the pair as normal.

```
fn snek_str(val: i64, seen: &mut Vec<i64>) -> String {
    if val == 7 { "true".to_owned()}
    else if val == 3 { "false".to_owned() }
    else if val % 2 == 0 { format!("{}", val >> 1) }
    else if val == 1 { "nil".to_owned() }
    else if val & 1 == 1 {
        if seen.contains(&val) { return "...".to_owned() }
        seen.push(val);
        let addr = (val - 1) as *const i64;
        let fst = unsafe { *addr };
        let snd = unsafe { *addr.offset(1) };
        let v = format!("(pair {} {})", snek_str(fst), snek_str(snd));
        seen.pop();
        v
    } else { format!("unknown value: {val}") }
}

#[export_name = "\x01snek_print"]
fn snek_print(val: i64) -> i64 {
    println!("{}", snek_str(val, &mut vec![]));
    val
}
```

11.9 A Brief Sketch of Equality

Given two pairs, how can we check if they are equal? We can use the following Rust implementation,

```
fn snek_eq_helper(val1: i64, val2: i64, seen: &mut Vec<(i64, i64)>) -> bool {
    if seen.contains(&(val1, val2)) { return true }

    seen.push((val1, val2));
    // continue on.
}
```

The idea is that if we come across the same two cycles, we can assume that they're equal and return. Otherwise, we can evaluate the pairs as usual.

12 Garbage Collection

Now that we're working with memory in the heap, let's suppose we *don't* have a lot of memory to work with. In this case, we need to think about *garbage collection* as a way to get rid of unused memory so we can allocate memory for useful things.

12.1 Motivation

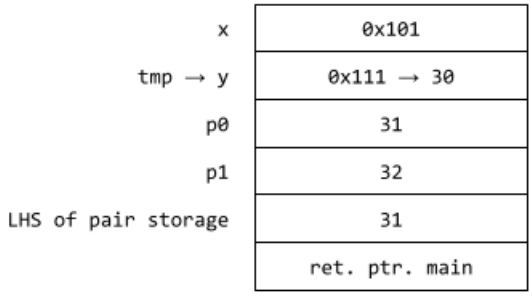
Let's take a look at two examples to get an idea of what we're working with.

12.1.1 Motivation 1: Basic Garbage

Recall the following code from class:

```
(let (x (pair 1 2))
    (let (y (let (tmp (pair 10 20)) (+ (fst tmp) (snd tmp))))
        (let (p0 (+ (fst x) y))
            (let (p1 (+ (snd x) y))
                (pair p0 p1)
            )
        )
    )
)
```

After creating the final pair, the memory diagram looks like[17][18]



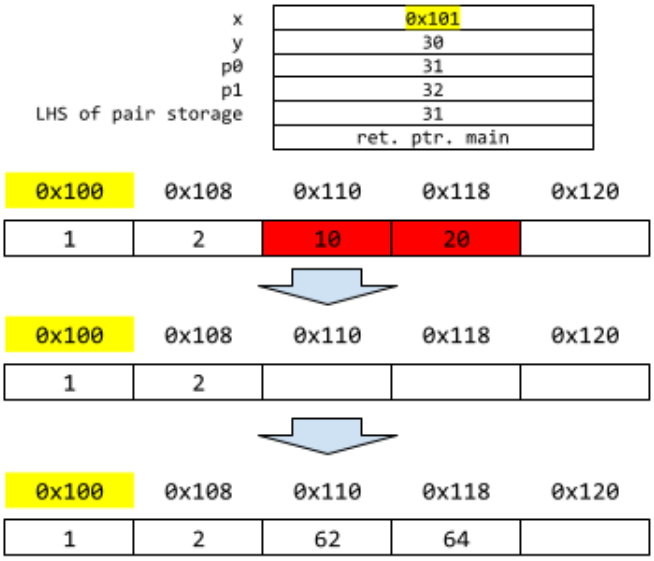
Here, the stack is the top diagram while the heap is the bottom diagram. Note that `rax`[19] is `0x121`, and `r15`[20] is `0x130`.

Now, let's suppose our heap only has five available words. `rax` will hold the result of `(+ (snd x) y)` (i.e., result of addition, which is a number). Immediately, we should notice that

- We don't have enough memory to allocate for the final pair!
- More importantly, however, the values in the heap at location `0x110` and `0x118` are **garbage**. Nothing in the stack (or a register) is referring to these values!

[17] For the sake of conciseness, we're showing the numbers without their tagged representation.
[18] Also, the `tmp` to `y` arrow indicates that we're *reusing* that space.
[19] Which is storing the answer to our program
[20] Recall that `r15` will always point to the next available word in the heap

To clarify, the idea is that any memory that is not reachable from the stack (or any registers) is considered garbage and can be reused. So, our goal is to get rid of the garbage so we have enough memory to allocate for the final pair. With this said, a high-level implementation of a basic garbage collector would look something like this.



So here's what's going on:

- We've determined that the stuff at `0x110` and `0x118` are garbage, so we can get rid of them.
- After that, we can *compact the heap*, essentially moving `r15` back to `0x110`. After that, we can allocate memory for our next pair.
- This gives us the desired result, with `rax` being `0x110`.

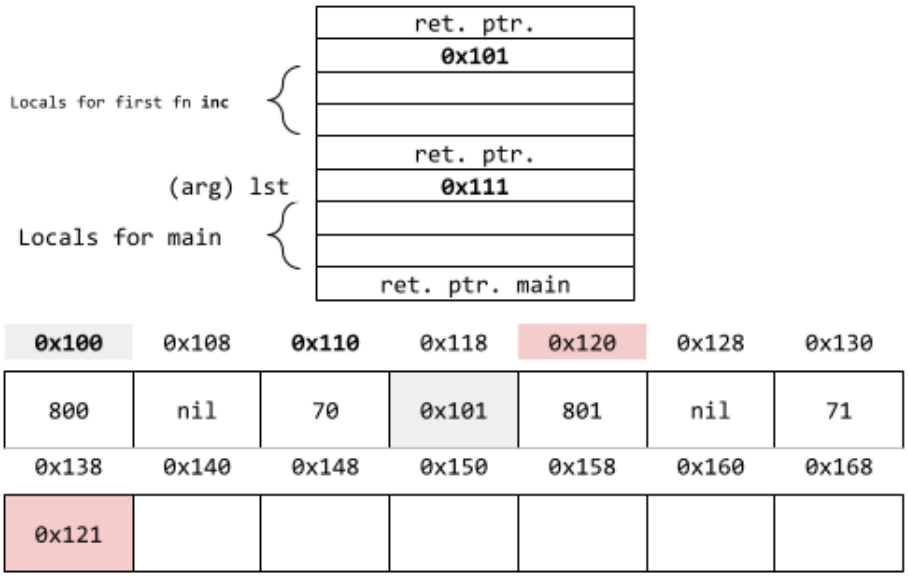
A key observation here is that we didn't need to "fix" any memory addresses stored in the stack or in the heap itself.

12.1.2 Motivation 2: Slightly Complicated Garbage

Consider the following code:

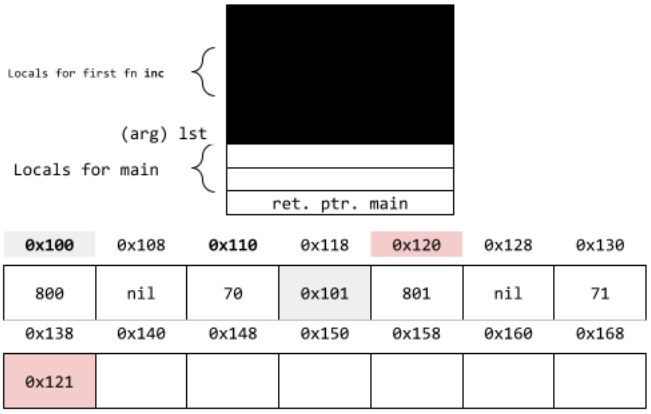
```
(fun (inc lst)
    (if (= lst nil)
        nil
        (pair (+ (fst lst) 1) (inc (snd lst)))
    )
)

(inc (inc (pair 70 (pair 800 nil))))
```

After the second call to `inc` (i.e., after the first call to `inc` finishes), we have the following rough diagram:



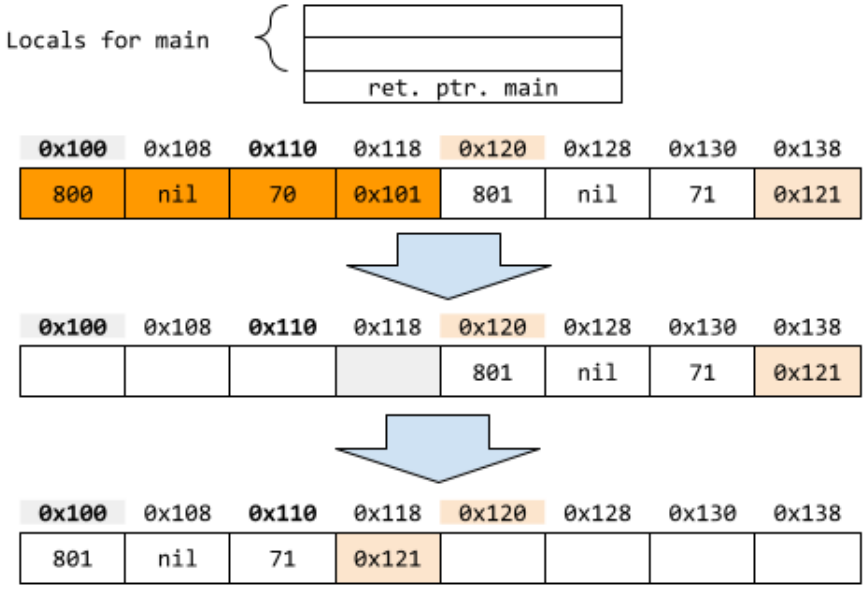
The register `rax` would have value `0x131`. What is considered garbage? The first four words – `0x100` through `0x118` – are considered **garbage** since there's no references to those words anywhere in the stack. However, why is this the case?

- We'll define the stack as everything at an address higher than `rsp` to the top of `our_code_starts_here`, but not anything lower address or above that. Therefore, we don't need to consider the following in the stack when deciding what is garbage:



- So, as long as no references to `0x111` is in main, we can prove that it's garbage.

Now, let's suppose our heap only has eight available words. In this example, `rax` is `0x131`. As one might have suspected, we don't have any memory left to allocate the remaining pairs needed for this program; in particular, after we call the `inc` function with the pair that we got from our initial call to `inc`, we don't have enough memory to allocate another pair needed for the recursive call. So, we need to collect some garbage. Here's how we might go about this.



So, here's what's going on:

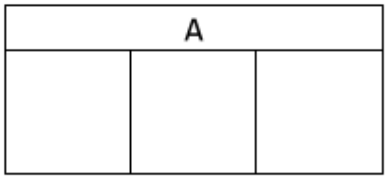
- We determined that the first four words are garbage, since no items in the stack or any registers are referring to those four words in the heap.
- We can compact the heap by moving `r15` to the beginning of our heap, thus allowing us to reuse the four words that are garbage.
- Now that we have room in the heap, we can allocate the final pair and change `rax` to point to our final result.

Well, *not quite*. Unlike the previous example, notice how we have a memory address in the heap that's referring to a memory address that's now garbage. That memory address has been relocated, so **we need to fix this.** Specifically, we need to change the value at `0x118` to point to `0x100`, not the garbage value at `0x120`! Likewise, any call to the stack that uses any memory addresses to the heap might need to be fixed before we can continue.

Therefore, we need to not only compact the heap, but also *relocate/forward* all existing references. For this, we will now discuss an algorithm that we can use to perform garbage collection. This algorithm is known as the **mark-compact algorithm**.

12.2 Layout and Notation

In the following sections, we'll make use of the following box to represent heap space. Each square box represents one word in the heap (so each group of boxes represents three words of contiguous heap memory).



The (A) represents an address to this group of memory (like `0x100`).

The first box of each group is some metadata that we'll need for garbage collection. For our algorithm here, we will need to make use of this metadata. However, there are algorithms out there that don't require metadata.

In any case, the important thing to remember is that, under this representation, **each pair** requires 3 words in the heap.

12.3 The Algorithm

At a high level, our algorithm looks like

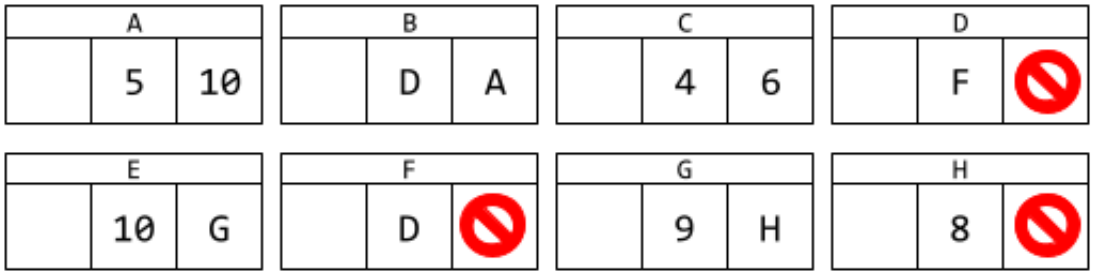
```
mark(roots):
    for ref in roots:
        mark_heap(ref)

mark_heap(r):
    if r.marked:
        return
    r.marked = true
    for (i, r') in r:
        if ispair(r')
            mark_heap(r')

fwd_headers():
    from = 0
    to = 0
    while move_from < HEAPEND:
        if from.marked:
            from.fwd = to
            to += 3 words/from.size
        from += 3 words/from.size

fwd_internal(roots):
    for ref in roots:
        update_fwd(ref)
        fwd_heap(ref)

fwd_heap(r):
    if r.fwded:
        return
    r.fwded = true
    for (i, r') in r:
        if ispair(r')
            r[i] = getfwd(r')
            fwd_heap(r')

compact():
    ...
```

Roughly speaking, we can break each of these methods into four groups:

1. `mark(roots)` and `mark_heap(r)`
2. `fwd_headers()`
3. `fwd_internal(roots)` and `fwd_heap(r)` (forwarding the internal pointers)
4. Compacting.

12.4 Motivating Example

Let's consider the following heap structure.



Note that this example doesn't correspond to any particular program.

12.4.1 Step 1: Finding the Root Set

> First, we want to find all the references that are currently on the stack (or registers). These are called the **root set**.

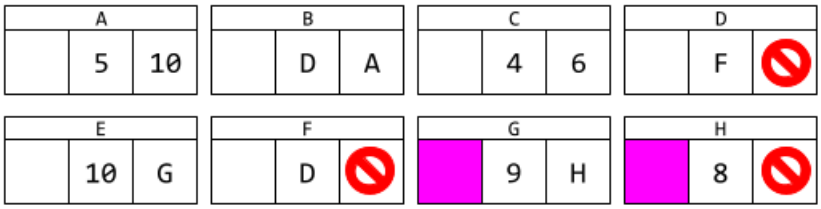
Right now, we don't know what is considered garbage (since we don't know what the stack or the registers look like). So, let's suppose G and B are the only two references on the stack (or registers, depending on implementation). We call this the root set (the roots of your traversal into the heap). Let's suppose we want to *clean* the heap.

12.4.2 Step 2: Marking Heap to Find Live Data

> Now, we want to call the `mark` function with our root set. This is where we're going to *mark* the memory in heap that are still in use (i.e., should not be garbage collected). This process is effectively depth-first search.

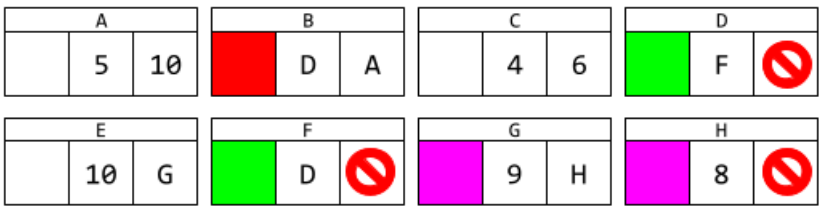
When we call `mark` with our root set, we're iterating over each root, which in our example is G and B.

- We first call `mark_heap` with G. This will mark G, and then recursively call `mark_heap` with B and thus mark B as well. The result of marking is shown below:

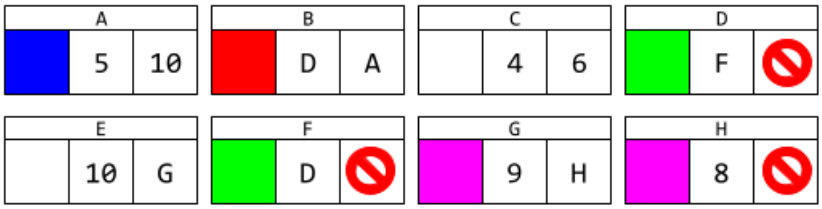


  Note that we're only considering non-`nil` pairs.

- Next, we call `mark_heap` with B. This will mark B, and then
  - recursively call `mark_heap` with D, marking it. Then, we'll recursively call `mark_heap` with F, marking it. Finally, we recursively call `mark_heap` with D, but since this has already been marked we don't need to do anything.



– after that's done, we cal recursively call `mark_heap` with A, marking it. Note that, at A, there's no other pairs (only raw numbers), so we're done.

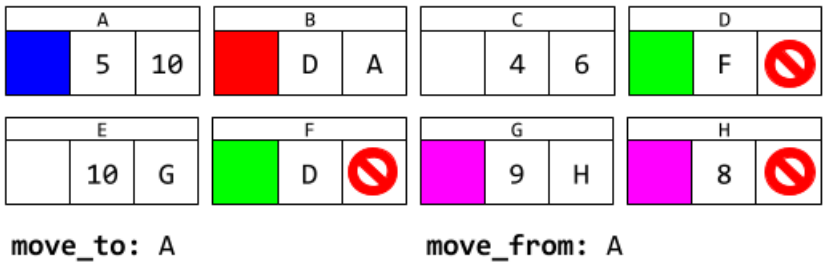


At this point, we're done. Notice how memory (C) and (E) haven't been marked; this means that they're garbage. Our goal, then, is to move `r15` to between F and G, and start allocation there! This, however, means we need to move everything back (compacting the heap).

12.4.3 Step 3: Forward Headers

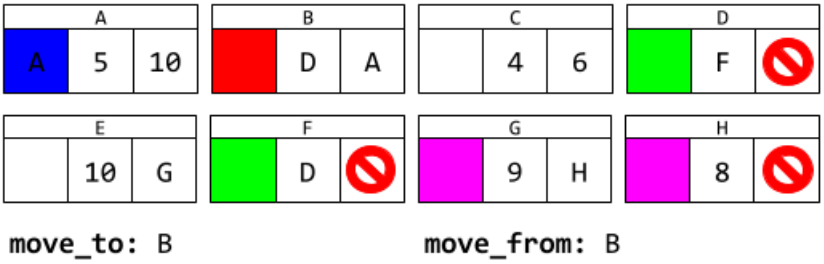
> We'll make use of `fwd_headers`. For each marked pair, this function will set the new address (to store that pair after compacting) in the pair's metadata (the first node).

Initially, we'll set `move_to` and `move_from` (`to` and `from` in the code, respectively) to `A` (the first memory location in heap).

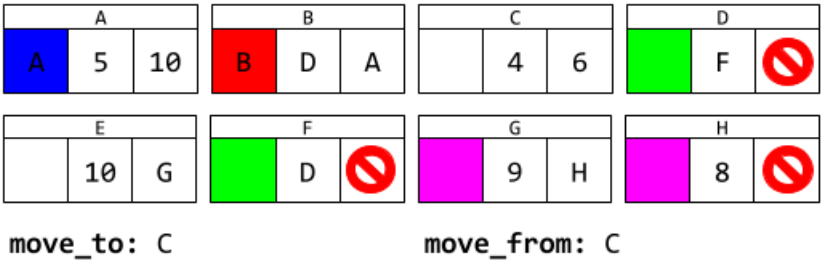


Iterating over each block of memory, we have

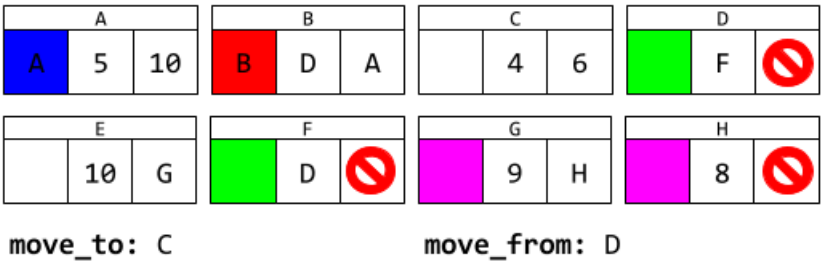
1. Since `move_from` $= A$ is marked, we set its metadata to `move_to` $= A$ (indicating that we'll move A to A after compacting). We also need to increment `move_to` and `move_from`. This gives us the following diagram:



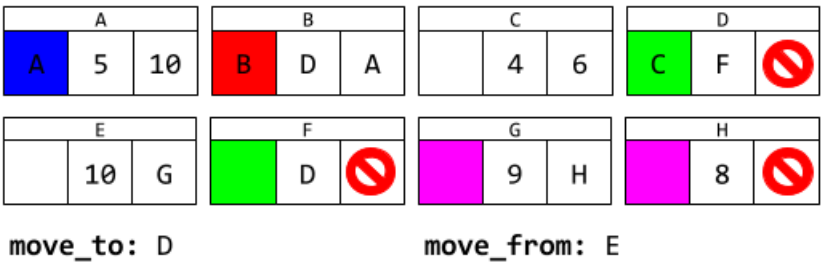
2. Since `move_from` $= B$ is marked, we set its metadata to `move_to` $= B$. We also need to increment `move_to` and `move_from`. This gives us the following diagram:



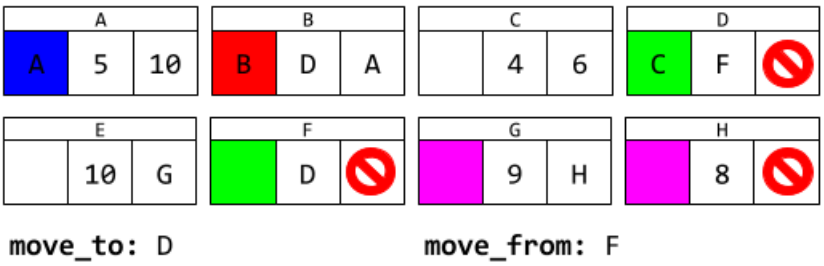
3. Since `move_from` $= C$ is **not** marked, we only increment `move_from`. This gives us the following diagram:



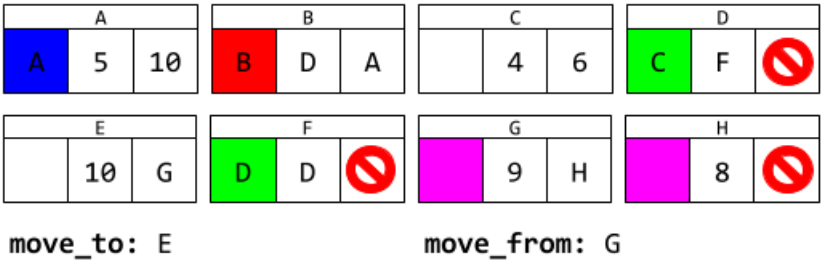
4. Since `move_from` $= D$ is marked, we set its metadata to `move_to` $= C$ (indicating that, after compacting, D should be moved to C's location). We also need to increment `move_to` and `move_from`. This gives us the following diagram:



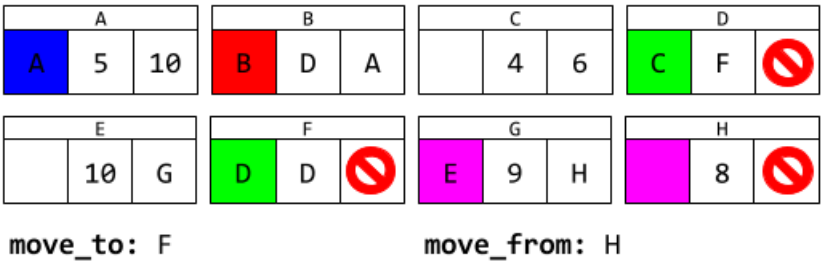
5. Since `move_from` $= E$ is not marked, we only increment `move_from`. This gives us the following diagram:



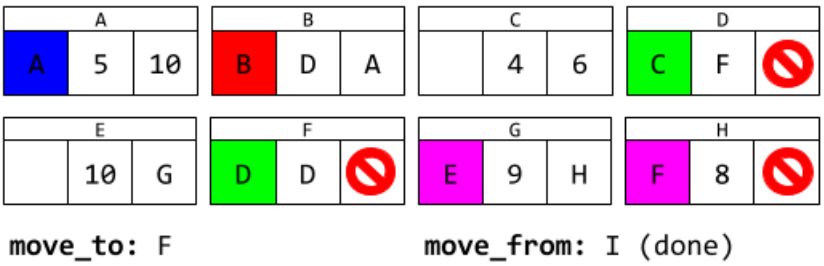
6. Since `move_from` $= F$ is marked, we set its metadata to `move_to` $= D$ (indicating that, after compacting, F should be moved to D's location). We also need to increment `move_to` and `move_from`. This gives us the following diagram:



7. Since `move_from` $= G$ is marked, we set its metadata to `move_to` $= E$. We also need to increment `move_to` and `move_from`. This gives us the following diagram:



8. Since `move_from` $= H$ is marked, we set its metadata to `move_to` $= F$. We also need to increment `move_to` and `move_from`. This gives us the following diagram:



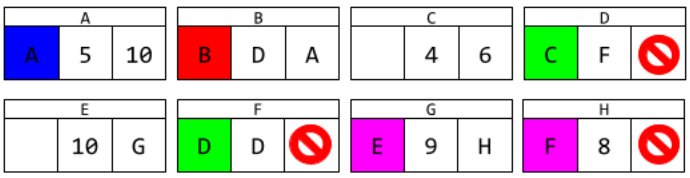
At this point, we're done. So, we know where each memory location should be moved to after compactness so that the heap is contiguous. *However*, we still need to update all the internal references within the heap!

12.4.4 Step 4: Forward Internal Addresses

Now that we've marked where everything should be moved to so we can maintain a contiguous heap structure, we still need to update all internal references so they point to the right places. For this, we'll use `fwd_internal`. The idea is that, for each block of memory, we want to consider each reference that the block has, if any. For each reference r:

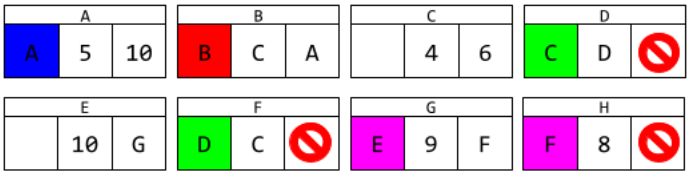
- Access the original memory block (before compacting) at r.
- Get its forwarding address and use that as the new reference.

At the moment, this is what our memory diagram looks like:



1. A has no internal references; there's no changes that need to be made.
2. One of B's references need to change.
   - Consider reference D. Looking up reference D, we see that its forwarding address is C. Therefore, we replace D with C.
   - Consider reference A. Looking up reference A, we see that its forwarding address is A. Therefore, we can keep this as A.
3. C is garbage, so we don't need to consider it.
4. For D, one of its references need to change.
   - Consider reference F. Looking up reference F, we see that its forwarding address is D. Therefore, we replace F with D here.
   - `nil` is kept the same.
5. E is garbage, so we don't need to consider it.
6. One of F's references need to change.
   - Consider reference D. Looking it up, we see that its forwarding address is C. So, we replace D with C.
   - `nil` is kept the same.
7. One of G's references need to change.
   - 9 is a number, so we leave it alone.
   - Consider reference H. Looking it up, we see that its forwarding address is F. So, we replace H with F.
8. H has no internal references, so no changes needed.

The result of changing the internal references is



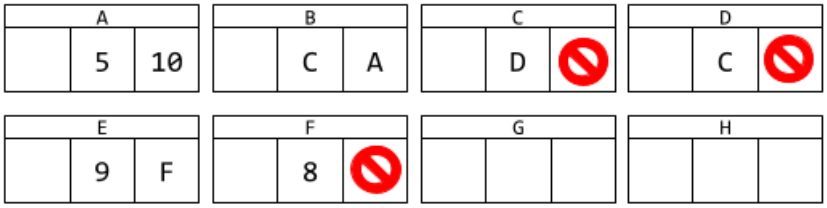
12.4.5 Step 5: Compacting the Heap

> Finally, we want to compact the heap. Compacting will be *iteration* (like `fwd_headers`), **not** traversal (like `mark`). In particular, iteration involves iterating over each live data and copying it to its forwarding pointer.

In the compacting step, we want to

- copy each live data to the forwarding pointer, and
- un-mark each data.

This gives us



13 Optimization

In this section, we'll talk more about optimization.

Definition 13.1: Optimization

An **optimization** (for a compiler) is a version that produces programs that evaluate the same answer as the prior version, but are "better" on some cost metric.

13.1 Examples of Cost Metrics

Some examples of cost metrics that we might want to improve on include

- **Time:** How long does it take for the program to run?
- **Space:** How much process memory does the program use?
- **Binary Size:** The size of the compiled binary, and the number of instructions.
- **Executed Instructions:** The overall number of instructions executed (compared to the total number of instructions). This is also similar to the number of jumps in the resulting assembly.

Remarks:

- The first two metrics – time and space – are generally the most important ones.
- We might also care about properties like compile time, extensibility, debuggability, and platform independence, although these are harder to measure.

13.2 High-Level Optimization Suggestions

Some suggestions for optimizations include

- **Register Allocation:** Storing values in registers rather than in memory, since access to registers are generally faster than access to memory.
- **Dead Code Elimination:** Remove code that the compiler can prove will never run.

  (Example.) Consider the following code:

  ```
  if false 3 4
  ```

  Here, we know for sure the `3` will never execute.

  This also includes things like removing unused variables from compilation.

- **Constant Folding:** Evaluate "what you can solve" in the compiler.

  (Example.) Consider the following code:

  ```
  (+ (* 2 3) input)
  ```

  Here, we know that `(* 2 3)` should be `6`, so this is basically equivalent to

  ```
  (+ 6 input)
  ```

- **Common Subexpression Elimination:** Eliminate repeated code in favor of a single instance of it.

- **Memory Packing:** Eliminate unused memory due to alignment (e.g., struct alignment).
- **Loop Unrolling:** We can unroll a loop if we know that it has a constant bound. In other words, essentially hardcode all iterations.

(Example.) Consider the following code:

```
(let (x 0) (loop (if (< x 3) (set! x (add1 x)) (break x))))
```

This is equivalent to

```
(let (x 0) (block
    (set! x (add1 x))
    (set! x (add1 x))
    (set! x (add1 x))
))
```

- **Type-Directed Compilation:** We can remove some code that involves type checking if we know for sure that we're working with the correct types.

(Example.) Consider the code

```
(+ 1 (* input 2))
```

While type checking is necessary for `(* input 2)`, it's probably not necessary for the plus expression since we can assume that both sides are numbers.

- **Peephole Optimization:** We can remove redundant move operations in the resulting assembly.
- **Inlining Functions:** We can inline function calls, especially if we have a small one.

(Example.) Consider the following function and resulting code:

```
(fun (f x)
    <body>)
(f 10)
```

This could be functionally equivalent to

```
(let (x 10) <body>)
```

Note that we might need to consider things like recursion or other function calls in the function body, since that might prevent us for optimizing.

- **Instruction Selection:** We can also possibly exploit the structure of our code.

(Example.) Consider the following code:

```
(if (< x 10) ... ...)
```

Generally, our compiler would put either `true` or `false` into `rax`, and then evaluate `rax` when deciding where to jump. However, in this particular code, we can probably just conditionally jump on the spot.

13.3 Optimization: Register Allocation

Let's consider the following code:

```
(let (n (+ 5 9))
    (let (m (+ 2 3))
        (let (x (+ n 1))
            (let (y (+ m 2))
                (+ x y)))))
```

The corresponding assembly[21], along with the corresponding code from the above, is shown below.

```
sub rsp, 40
  mov rax, 10
  mov [rsp + 0], rax    ; LHS of (+ 5 9)
  mov rax, 18
  add rax, [rsp + 0]

  mov [rsp + 0], rax    ; Variable n in (let (n ...))

  mov rax, 4
  mov [rsp + 8], rax    ; LHS of (+ 2 3)
  mov rax, 6
  add rax, [rsp + 8]

  mov [rsp + 8], rax    ; Variable m

  mov rax, [rsp + 0]    ; Variable n lookup
  mov [rsp + 16], rax   ; LHS of (+ n 1)
  mov rax, 2
  add rax, [rsp + 16]

  mov [rsp + 16], rax   ; Variable x

  mov rax, [rsp + 8]    ; Variable m lookup
  mov [rsp + 24], rax   ; LHS of (+ m 2)
  mov rax, 4
  add rax, [rsp + 24]

  mov [rsp + 24], rax   ; Variable y

  mov rax, [rsp + 16]   ; Variable x lookup
  mov [rsp + 32], rax
  mov rax, [rsp + 24]   ; Variable y lookup
  add rax, [rsp + 32]
add rsp, 40
```

One thing to notice immediately is that we reused some memory locations. One example is `[rsp + 8]`, which is where we stored both a temporary for addition and a value associated with a variable. We can generalize how many memory locations we ultimately *will* use by using the `depth` function. In particular, if `depth(expr)` $\leq$ Available Registers, then we can avoid memory entirely.

There are two questions we should now consider.

1. (`x86_64`.) What registers should we use?

[21] With tag checks removed to make the assembly more concise.

We can use the registers `rbx`, `r12`, `r13`, `r14`, which are callee-saved registers. Note that we aren't using `r15` because this register is specifically the heap pointer.

2. (Design.) How should we implement this?

We can create a `Loc` *enum* that holds either a register or a stack location (offset). Then, our environment can be represented by `HashMap<String, Loc>`.

Suppose we have a list of registers that we can use. We can create a `get_loc` function which takes a stack index and returns the new location to be used; this might look something like

```
let regs = [...];
get_loc(si):
    if si < regs.size():
        return regs[si];
    else:
        return Stack(si - regs.len());
```

Then, we can use this location to update the environment, like

```
...
| ELet(x, val, body) => {
    env.update(x, get_loc(si));
}
```

Note that, while this is an *improvement* to how our program is compiled, this can still be made a *lot better*. Some other implementation notes to consider include:

- We need to add code to save and restore registers in function definitions.
- We need to compute stack size based on `depth - available registers`.

Some improvements we could make to what we have so far include

- Registers for outer bindings and stack for inner bindings.
- Frequency matters.
- Precompute registers and locations for all variables and temporaries across functions.
- Are we using the minimal number of locations? (e.g., is the depth minimal?)

Remark: The register allocation algorithm we're talking about, which uses an idea similar to `depth`, is similar to the *Sethi-Ullman algorithm*.

13.3.1 High Level Steps

At a high level, we aim to answer the following questions:

- The first step is to find the minimal number of locations needed to store all the working variables in an expression.
- What pairs of variables must be stored (or must be "live") at the same time?

13.3.2 The Minimal Number of Locations

Consider the following program:

```
(let (b 4)
    (let (x 10)
        (let (i (if input
                    (let (z 11) (+ z b))
                    (let (y 9) (+ y 1))))
            (let (a (+ i 5))
                (+ a x)))))
```

To answer the second question, note that

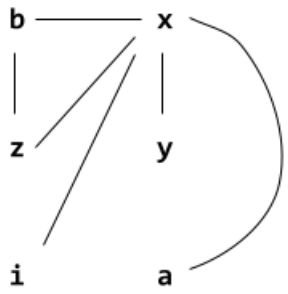
- `i` and `x` need storage at the same time.
- `a` and `b` do not need storage at the same time[22].

How many memory locations are needed? We'll look at the program from the *end* to the beginning.

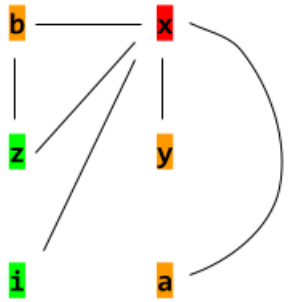
- We first begin by looking at what variables are in use at the end. In this case, `a` and `x` are in use. The set of all variables in use is
$$\{a, x\}.$$
- We're going to go back "up" the program. When we get to a `let`-bindings, we're going to remove it from the set of variables that are in use right now. In the next level, we're *using* `i` and `x`, but we aren't using `a` here since `a` is being created. The set of all variables in use is
$$\{i, x\}.$$
- The `if`-expression is more interesting. We need to consider both branches of the `if`-expression. Note that, in this step, `i` is being created, so we don't have access to `i` yet.
  - Looking at the end of the "else" branch, at the body of the `let` binding, notice how `y` is being used. `x` is still around. The set of all variables in use is
$$\{y, x\}.$$
  - Looking at the end of the "then" branch, at the body of the `let` binding, notice how `z` and `b`[23] are in use. As usual, `x` is still around. The set of all variables in use is
$$\{z, b, x\}.$$
- At the `let`-binding for `i` (*not* in the body), we no longer have `z` or `y`, and `i` is being initialized here (so we aren't using `i` here). Thus, this gives us the variables in use
$$\{x, b\}.$$
- Moving "up" the program to the `let`-binding for `x`, we now only have the variables in use $\{x\}$.
- Finally, moving "up" the program to the `let`-binding for `b`, we have the variables in use $\emptyset$.

This information is telling us what variables need to be stored at the same time. Something we can do with this information is turn this into a **graph** where there's an edge between two variables *if* they're in use at the same time.

[22] Notice how we only use `b` once: in the `if`-expression. After that, we don't use `b` again.
[23] Even though `b` is defined at the top, this is the first time we're seeing `b` in use.



This is a graph where if there are two variables that had to be live at the same time, then there is an edge. How do we make it so we can have a set of locations where each variable can be assigned to a register that's different from all the things it conflicts with? This problem is known as **graph coloring**. The idea is that we want to find k colors assigning $1 \dots k$ to each node such that k is minimal and no edge has the same index for both nodes. A coloring for this graph is



We only need 3 colors! In terms of what our compiler would output, we would end up with the environment

```
{x: 1, b: 2, y: 2, a: 2, z: 3, i: 3}
```

In other words, `x` gets abstract location 1, `b` gets abstract location 2, and so on. Note that this makes a few assumptions:

- All intermediates are carefully named and used (no useless temporaries).
- Assuming all temporaries are explicit, this could replace `depth(e)`. Note that this means *simple constants!*
- All variables are distinctly named (although we can rename all non-distinct names if needed).

13.3.3 Algorithm

The algorithm for this process is as follows:

- Visit last, or innermost, expression first. THis means recurse, then working with result.
- Track set of variables we have seen used, then remove from set at the let-bindings.

So, going back to the example code, we have the following set of active variables.

```
(let (b 4)                                  ; {}
    (let (x 10)                             ; {b}
        (let (i (if input                   ; {b, x}
                    (let (z 11) (+ z b))    ; {z, b, x}
```

```
            (let (y 9) (+ y 1))))   ; {y, x}
    (let (a (+ i 5))                ; {i, x}
        (+ a x)))))                 ; {a, x}
```

For each pair of active variables that appear at the same time, we draw an edge between them in the graph.

13.3.4 Restrictions

One of the main restrictions of the algorithm for register allocation is simply temporary values: what do we do with register allocation for temporary values that don't have any names?

13.4 Intermediate Representation

Let's consider the following program:

```
(+ (- 5 input) (* input (if (> 0 input) 1 -1)))
```

Below is a transformed version of the program where every nested expression that would have introduced a temporary is now a `let`-bound variable.

```
(let (tmp 1 (- t input))
    (let (tmp 2 (> 0 input))
        (let (tmp3 (if tmp2 1 -1))
            (let (tmp4 (* input tmp3))
                (+ tmp1 tmp4)
            )
        )
    )
)
```

This makes the order of operations very explicit. Notice that it's very clear that we're doing left-to-right operation. This process also makes code generation for operations a lot easier, since everything is already stored on the stack or in a register.

(Example.) Perform the transformation on the function

```
(fun (sumsquares x y)
    (+ (* x x) (* y y)))
```

As mentioned, we want to break our complex expression into much simpler types. We can accomplish this by defining any computations into `let`-bindings. This gives us

```
(fun (sumsquares x y)
    (let (val (* x x))
        (let (val2 (* y y))
            (+ val val2)
        )
    )
)
```

This transformation is fairly common, and there is a fairly standard algorithm for this transformation that takes every non-trivial or non-atomic (basically, everything that's not a literal value) and puts them in a `let`-binding.

13.4.1 Different Grammar Forms

There are two types of grammars we want to consider.

- **A-Normal Form:** This is essentially our grammar as is. This cares about scope, binding, order or evaluations, and so on.

```
<expr> := <number> | <id> | true | false | nil
    | (+ <expr> <expr>)
    | (- <expr> <expr>)
    | (if <expr> <expr> <expr>)
    | (break <expr>)
    | ...
    | (let (<id> <expr>) <expr>)
    | ...
```

- **ANF-Restricted Grammar:** This grammar is effectively the transformed grammar; that is, given our A-Normal Form, we can transform it into a more restricted version. We'll denote this as `AExpr`.

```
<val> := <number> | <id> | true | false | nil
<expr> := (+ <val> <val>)
    | (pair <val> <val>)
    | (if <val> <block> <block>)
    | (break <val>)
<block> := (let (<id> <expr>) <block>)
    | (loop <block>)
    | (break <block>+)
    | ...
    | <expr>
    | <val>
```

  This is restricted in the sense that the grammar is broken up into three different groups (productions). You have expressions (`<expr>`) that form blocks (`<block>`). Blocks have expressions in them that can perform calculations; so, we can think of `<expr>` as something that performs a calculation of some type (e.g., binary operations, creating a new pair, and so on). All the arguments to these expressions that do calculations must be primitive values or identifiers.

Remark: Loops are an interesting case to think about here.

13.4.2 Going from Normal to Restricted

How do we create an algorithm that transforms code written under one grammar to code written in the restricted grammar? Note that this is a very standard algorithm, so we'll mainly gain some intuition. One choice to make is whether we want to introduce a new `enum` for ANF expressions. For example, is it worth it to introduce a bunch of new `enum`s like shown below?

```
enum Val {
    VNum,
    VId(String),
    ...
}

enum AExpr {
    APlus(...),
    APair(...),
    ...
```

```
}

enum Block {
    BLet(...),
    ...
}
```

In any case, we can write a few functions to facilitate the conversion process.

- `anf_to_val(e: &Expr) -> Val`: Converts an A-Normal Expression Form to a literal value under the ANF-Restricted Expression Form.
- `anf_to_expr(e: &Expr) -> AExpr`: Converts an A-Normal Expression Form to a computation expression under the ANF-Restricted Expression Form.
- `anf_to_block(e: &Expr) -> Block`: Converts an A-Normal Expression Form to a block expression under the ANF-Restricted Expression Form.

13.4.3 ANF to Value

Let's suppose we want to implement

```
fn anf_to_val(e: &Expr) -> Val {
    match e {
        ...
    }
}
```

- For an expression `e` like `Number(n)`, we can trivially return `VNum(n)`.
- For an expression `e` like `Plus(e1, e2)`, this becomes more complicated. This will involve a nested plus expression, and we should return an identifier that we can use later. Note that `e1` and `e2` may be complicated expressions, but we want them to be `Val`s as well. So, we'll probably need to do some recursive calls to ideally break `e1` and `e2` down into `let`-bindings.

  ```
  Plus(e1, e2) => {
      let (v1, b1) = anf_to_val(e1);
      let (v2, b2) = anf_to_val(e2);
      // Assume new_label() is a function that returns a new identifier.
      let new_name = new_label();
      // This isn't Rust syntax, but basically we want all the bindings from
      // b1, all the bindings from b2, and our new binding in this vector.
      (VId(new_name), vec![...b1, ...b2, (new_name, APlus(v1, v2))])
  }
  ```

  Notice how we're returning a tuple. The first element is the identifier that will store the result of the evaluation of this expression. However, we also need to return list of `let`-bindings we need to eventually stick in front of *this* identifier in order for it to work (otherwise, the identifier won't be bound). So, we should modify the function return type:

  ```
  fn anf_to_val(e: &Expr) -> (Val, Vec<(String, AExpr)>) { ... }
  ```

13.5 Intermediate Representation

13.5.1 Rust AST

Given what we've just discussed, the Rust representation of A-Normal Form and ANF-Restricted Form might look something like

```
pub enum AVal {
    Num(i32),
    True,
    False,
    Id(String),
}

pub enum AExpr {
    Plus(Box<AVal>, Box<AVal>),
    Eq(Box<AVal>, Box<AVal>),
    Lt(Box<AVal>, Box<AVal>),
    Print(Box<AVal>),
    Set(String, Box<AVal>),
    Call1(String, Box<AVal>),
    Call2(String, Box<AVal>, Box<AVal>),
    Pair(Box<AVal>, Box<AVal>),
    Fst(Box<AVal>),
    Snd(Box<AVal>),
    Break(Box<AVal>),
    Loop(Box<ABlock>),
    If(Box<AVal>, Box<ABlock>, Box<ABlock>),
    Val(Box<AVal>),
}

pub enum ABlock {
    Let(String, Box<AExpr>, Box<ABlock>),
    Block(Vec<ABlock>),
    Op(Box<AExpr>),
}
```

13.5.2 A Problem With Loops

Consider the following function:

```
(fun (range low high)
    (let (n high)
        (let (lst nil)
            (loop
                (if (= n low) (break lst)
                    (block
                        (set! n (+ n -1))
                        (set! lst (pair n lst)))
                )
            )
        )
    )
)
```

Converting the above function, which is written in A-Normal Form, to ANF-Restricted Form, yields:

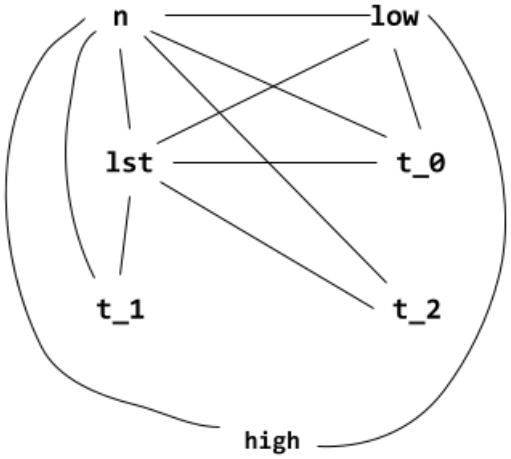
```
(fun (range low high)
    (let (n high)
        (let (lst nil)
            (loop
                (let (%t_0 (= n low))
                    (if %t_0 (break lst)
```

```
                        (block
                            (let (%t_1 (+ n -1)) (set! n %t_1))
                            (let (%t_2 (pair n lst)) (set! lst %t_2)))
                    )
                )
            )
        )
    )
)
```

Note that the `%` identifiers are just the convention we're using for temporaries. Let's now consider all interfering variables, which we can do by considering all variables inside-out.

```
(fun (range low high)
    (let (n high)           ; low, high
        (let (lst nil)                  ; n, low
            (loop
                (let (%t_0 (= n low))           ; lst, n, low, t_0
                    (if %t_0 (break lst)        ; n, lst, t_0
                        (block
                            (let (%t_1 (+ n -1)) (set! n %t_1))         ; t_1, n, lst
                            (let (%t_2 (pair n lst)) (set! lst %t_2)))  ; t_2, lst, n
                    )
                )
            )
        )
    )
)
```

Notice that we end up with the set of variables $\{\texttt{n}, \texttt{low}, \texttt{lst}, \texttt{t\_0}, \texttt{t\_1}, \texttt{t\_2}, \texttt{high}\}$. A graph representing this would look like



(Exercise.) How would the graph change if we swapped the order of the `set!` lines?

Suggestion: Because the last use of `lst` would be in the first `set!` instead of the last line, there wouldn't be an edge between `lst` and `t_1`.

But, one thing to keep in mind is that the `set!` is in a loop. So, `lst` needs to be in scope for the remainder of the block statement. Otherwise, a temporary could be assigned to the same register that was being used for `lst`.

Remarks:

- Remember that our algorithm just went from the end to beginning, but now we need to consider what identifiers have been defined outside of the loop.
- The issue with loops is that you have implicit backedges from the last expression in the loop to the beginning of the loop. We don't have names associated with these loops, which makes things difficult since we have names in the first place so we know where everything is.

13.5.3 A Rewrite of the Grammar

How do we rewrite our grammar to account for loops?

```
pub enum Val {
    Num(i32),
    True,
    False,
    Id(String),
}


pub enum Expr {
    Plus(Box<Val>, Box<Val>),
    Eq(Box<Val>, Box<Val>),
    Lt(Box<Val>, Box<Val>),
    Print(Box<Val>),
    Call1(String, Box<Val>),
    Call2(String, Box<Val>, Box<Val>),
    Pair(Box<Val>, Box<Val>),
    Fst(Box<Val>),
    Snd(Box<Val>),
    Val(Box<Val>),
}


pub enum Step {
    Label(String),
    If(Box<Val>, String, String)
    Goto(String),
    Do(Expr),
    Set(String, Expr)
}


pub struct Block {
    pub steps: Vec<Step>,
}
```

Some things to notice here:

- Rather than expressing our program as loops with breaks, we'll express them as labels and gotos.
- In the `if`-condition, the first `String` is the label to jump to if the condition is true; the last `String` is the label to jump to if the condition is false.
- So, whereas ANF is about scope and temporary variables, this is about control flow. Essentially, we're iteratively getting closer to assembly.

Here, we introduce the concept of **intermediate representation**. With the above new representation, we have the following intermediate representation:

```
range(low,high) {
    n <- high
    lst <- nil
   loop_0:
    %t_0 <- n == low
    if %t_0 thn_3 els_4
   thn_3:
    rax <- lst
    goto end_1
    goto ifend_2
   els_4:
    %t_1 <- pair(n, lst)
    lst <- %t_1
    %t_2 <- n + -1
    n <- %t_2
    rax <- n
    goto ifend_2
   ifend_2:
    goto loop_0
   end_1:
    return rax
}
```

Note that this is the compiled output of the `range` function, not in x86_64. Some things to note:

- There's no nesting of expressions anymore. There's no notion of labels being inside other labels.
- Lots of IRs have support for functions (including things like function scope, blocks within functions, etc.), but leave the actual calling convention to the language.
- The pipeline we would have is that surface syntax turns into ANF, and then ANF turns into IR. It's possible to do everything in one pass, but it's easier to do it in two passes.
- In this example of `range`, `rax` is now our designated answer variable, and we expect all lines before the `return rax` line to store the answer into `rax` (similar to what our compiler does right now).

With that said, running through the IR representation from last to start, the variables in use are:

```
range(low,high) {           ; {}
    n <- high               ; n
    lst <- nil              ; n, lst
   loop_0:                  ; n, lst
    %t_0 <- n == low        ; t_0, n, lst
    if %t_0 thn_3 els_4     ; t_0, n, lst
   thn_3:                   ; lst
    rax <- lst              ; rax, lst
    goto end_1              ; rax
```

```
    goto ifend_2            ; <empty for now>
   els_4:                   ; n, lst, t_1
    %t_1 <- pair(n, lst)    ; n, lst, t_1
    lst <- %t_1             ; lst, n, t_1
    %t_2 <- n + -1          ; t_2, n
    n <- %t_2               ; t_2, n
    rax <- n                ; n, rax
    goto ifend_2            ; <empty for now>
   ifend_2:                 ; <empty for now>
    goto loop_0             ; <empty for now>
   end_1:                   ; rax
    return rax              ; rax
}
```

Any time you reach a `goto`, any variables that are used at the jump target are copied over to the `goto`. In any case, we should not use this information to construct the graph needed to figure out how many registers we need to allocate. In particular, at these three instructions,

```
%t_2 <- n + -1          ; t_2, n
n <- %t_2               ; t_2, n
rax <- n                ; n, rax
```

there's no edge between `lst` and `t_2`. This means that we could potentially store `lst` and `t_2` into the same register, losing a possible important value. So, we need to run this same algorithm over and over until none of the sets change (i.e., when saturation occurs). Running through the algorithm again gives us:

```
range(low,high) {           ; {}                ;
    n <- high               ; n                 ;
    lst <- nil              ; n, lst            ;
   loop_0:                  ; n, lst            ;
    %t_0 <- n == low        ; t_0, n, lst       ;
    if %t_0 thn_3 els_4     ; t_0, n, lst       ;
   thn_3:                   ; lst               ;
    rax <- lst              ; rax, lst          ;
    goto end_1              ; rax               ;
    goto ifend_2            ; <empty for now>   ; n, lst
   els_4:                   ; n, lst, t_1       ;
    %t_1 <- pair(n, lst)    ; n, lst, t_1       ;
    lst <- %t_1             ; lst, n, t_1       ;
    %t_2 <- n + -1          ; t_2, n            ; lst
    n <- %t_2               ; t_2, n            ; lst
    rax <- n                ; n, rax            ; lst
    goto ifend_2            ; <empty for now>   ; n, lst
   ifend_2:                 ; <empty for now>   ; n, lst
    goto loop_0             ; <empty for now>   ; n, lst
   end_1:                   ; rax               ;
    return rax              ; rax               ;
}
```

Note that the last column are the variables *added* to the variables mentioned in the second columns. Anyways, after running this algorithm again, we reach saturation – we don't find any additional variables that need to be added. This gives us a complete graph that we can use to determine what registers can be allocated.

13.6 Flow Analysis

Let's consider the following code,

```
(let (curr lst)
    (let (total 0)
        (loop
            (if (= lst nil) (break total)
                (block
                (set! total (+ total (fst lst)))
                (set! lst (snd lst)))))))
```

How can we use flow analysis to reduce the amount of tag checking generated in the final assembly?

13.6.1 The `check` Instruction

An idea we want to do is to make tag checks explicit with `check` steps: which checks can we remove? One new step we can introduce in the intermediate representation is

```
check <some bool expr>
```

The semantics are simple: if the check is true, then everything continues as normal. Otherwise, an error is thrown.

> (Example.) `check sametag(curr, nil)` checks to see if `curr` has the same tag as `nil`. Something we've incorporated into our compiler is the `isnum(x)` and `isbool(x)` checks, which checks to see if the expression x is a number or boolean, respectively.

With this said, the corresponding intermediate representation of the above code is

```
sum(lst) {
    start0:     curr <- lst
    start1:     total <- 0
    loop_0:     check sametag(curr, nil)
    loop_1:     %t_0 <- curr == nil
    loop_2:     if %t_0 thn_0 els_0
    thn_0:      rax <- total
    thn_1:      goto end_0
    thn_2:      goto ifend_0
    els_0:      check isnonnilpair(curr)
    els_1:      %t_1 <- fst curr
    els_2:      check isnum(total)
    els_3:      check isnum(%t_1)
    els_4:      %t_2 <- total + %t_1
    els_5:      total <- %t_2
    els_6:      check isnonnilpair(curr)
    els_7:      %t_3 <- snd curr
    els_8:      curr <- %t_3
    els_9:      rax <- curr
    els_10:     goto ifend_0
    ifend_0:    goto loop_0
    end_0:      return rax
}
```

We can use flow analysis to analyze how data flows through a program. We can use this information to identify variables that hold values at different points in the program, and how these values change over time. For our purposes, we wish to use flow analysis to reduce the amount of unnecessary tag checking. Using the `check` instruction that was mentioned, we can do just this.

13.6.2 A Flow Analysis Walkthrough

The flow analysis we'll do starts from the beginning and goes to the end (this is known as *forward analysis*). The information we'll keep track of are the *potential* tags. Let's analyze each line of the intermediate representation. For each line executed, we consider what possible tag value each variable can represent. The potential tags are **N**umbers, **B**ooleans, **Nil**, and **P**airs. At any point in the program, each variable can hold a set of these possible types. Let $A = \{N, B, \text{Nil}, P\}$ be the set of all types.

IR	`lst`	`curr`	`total`	t_0	t_1	t_2	t_3	`rax`
`start0: curr <- lst`	A	$\to A$						
`start1: total <- 0`	A	A	$\to N$					
`loop_0: check sametag(curr, nil)`	A	$\to \{\text{Nil}, P\}$	N					
`loop_1: %t_0 <- curr == nil`	A	$\{\text{Nil}, P\}$	N	$\to B$				
`loop_2: if %t_0 thn_0 els_0`	A	$\{\text{Nil}, P\}$	N	B				
`thn_0: rax <- total`	A	{Nil, P}	N	B				N
`thn_1: goto end_0`	A	{Nil, P}	N	B				N
`thn_2: goto ifend_0`[100]								
`els_0: check isnonnilpair(curr)`[101]	A	$\{\text{Nil}, P\} \to P$	N	B				
`els_1: %t_1 <- fst curr`	A	P	N	B	$\to A$			
`els_2: check isnum(total)`	A	P	$N \to N$	B	$\to A$			
`els_3: check isnum(%t_1)`	A	P	N	B	$A \to N$			
`els_4: %t_2 <- total + %t_1`	A	P	N	B	N	$\to N$		
`els_5: total <- %t_2`	A	P	$N \to N$	B	N	N		
`els_6: check isnonnilpair(curr)`	A	$P \to P$	N	B	N	N		
`els_7: %t_3 <- snd curr`	A	P	N	B	N	N	$\to A$	
`els_8: curr <- %t_3`	A	$P \to A$	N	B	N	N	A	
`els_9: rax <- curr`	A	A	N	B	N	N	A	$\to A$
`els_10: goto ifend_0`	A	A	N	B	N	N	A	A
`ifend_0: goto loop_0`	A	A	N	B	N	N	A	A
`end_0: return rax`								

Remarks:

- At (100), we have dead code. So, nothing needs to be filled out.
- At (101), we can copy the tag information we have from the `goto` instruction which jumps to this line. In this case, we copied this information from the line `loop_2`.
- In general, we can copy the information from the `goto` to the target label. This is especially important when we have a `goto` that goes to a label that's *before* where the `goto` occurred.

At `goto loop_0`, we now perform a backwards jump back to the label `loop_0` and perform additional forward analysis with the information we found prior to the `goto`. These tags are denoted by red.

IR	`lst`	`curr`	`total`	t_0	t_1	t_2	t_3	`rax`
`start0: curr <- lst`	A	$\to A$						
`start1: total <- 0`	A	A	$\to N$					
`loop_0: check sametag(curr, nil)`	A	$A \to \{\text{Nil}, P\}$	N	$\to B$	$\to N$	$\to N$	$\to A$	$\to A$
`loop_1: %t_0 <- curr == nil`	A	$\{\text{Nil}, P\}$	N	$\to B$	N	N	A	A
`loop_2: if %t_0 thn_0 els_0`	A	$\{\text{Nil}, P\}$	N	B	N	N	A	A
`thn_0: rax <- total`	A	{Nil, P}	N	B	N	N	A	$A \to N$
`thn_1: goto end_0`	A	{Nil, P}	N	B	N	N	A	N
`thn_2: goto ifend_0`								
`els_0: check isnonnilpair(curr)`[102]	A	$\{\text{Nil}, P\} \to P$	N	B	N	N	A	A
`els_1: %t_1 <- fst curr`	A	P	N	B	$\to A$			
`els_2: check isnum(total)`	A	P	$N \to N$	B	$\to A$			
`els_3: check isnum(%t_1)`	A	P	N	B	$A \to N$			
`els_4: %t_2 <- total + %t_1`	A	P	N	B	N	$\to N$		
`els_5: total <- %t_2`	A	P	$N \to N$	B	N	N		
`els_6: check isnonnilpair(curr)`	A	$P \to P$	N	B	N	N		
`els_7: %t_3 <- snd curr`	A	P	N	B	N	N	$\to A$	
`els_8: curr <- %t_3`	A	$P \to A$	N	B	N	N	A	
`els_9: rax <- curr`	A	A	N	B	N	N	A	$\to A$
`els_10: goto ifend_0`	A	A	N	B	N	N	A	A
`ifend_0: goto loop_0`	A	A	N	B	N	N	A	A
`end_0: return rax`								

Remark:

- At (102), note that we're not directly copying N from `thn_1` to `els_0`. Rather, we're copying the tag information from the `goto` instruction that jumps to this line.

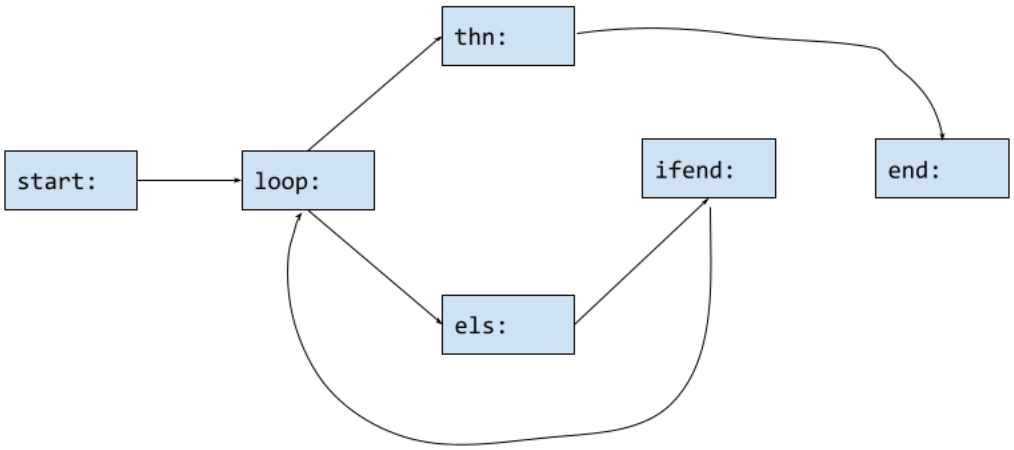
Let's consider the lines `els_2: check isnum(total)` and `els_6: check isnonnilpair(curr)`. Based on the forward analysis, these two lines of code are useless. Likewise, `els_0: check isnonnilpair(curr)` could be *optimized* (not removed) to check if `curr` is `nil`.

13.6.3 In Summary

In summary, the idea behind flow analysis is that there's really two steps:

1. Do the analysis and gather information
2. Rescan the program with that information and change the program to remove/optimize any code as needed.

The corresponding **control flow graph** looks like



At the start, we have a bunch of instructions. This eventually leads to a loop. In the `thn` branch, we go straight to the end since we have the dead code. In the `els` branch, we eventually get to the `ifend` statement where we end up going to the loop.

13.6.4 Another Flow Analysis Walkthrough

Consider the following program:

```
(fun (same_at vec1 vec2 i)
    (= (index vec1 i) (index vec2 i)))
```

We'll perform another flow analysis[24], again by starting from the beginning and going to the end. The information we'll keep track of are the *potential* tags. Our tags are now slightly more refined; in particular, we now have the set of tags,

$$\text{tag} := Z|\text{Pos}|\text{Neg}|B|V|\text{Nil}$$

Here, `Z` means the number zero, `Pos` means positive number, `Neg` means negative number. We also have `B` for boolean, `V` for vector, and `Nil` for nil. Once again, we let A represent the set of all possible tags. We also introduce $N = \{Z, \text{Pos}, \text{Neg}\}$ for numbers[25].

IR	`vec1`	`vec2`	`i`	t_0	t_1	`rax`
`start0: check isnonnilvec(vec1)`	$\to V$					
`start1: check isnum(i)`	V		$\to N$			
`start2: check i >= 0`	V		$N \to \{Z, \text{Pos}\}$			
`start3: check i < len(vec1)`	V		$\{Z, \text{Pos}\} \to \{Z, \text{Pos}\}$			
`start4: %t_0 <- vec1[i]`	V		$\{Z, \text{Pos}\}$	$\to A$		
`start5: check isnonnilvec(vec2)`	V	$\to V$	$\{Z, \text{Pos}\}$	A		
`start6: check isnum(i)`	V	V	$\{Z, \text{Pos}\}$	A		
`start7: check i >= 0`	.	.	.	.		
`start8: check i < len(vec2)`	.	.	.	.		
`start9: %t_1 <- vec2[i]`	.	.	.	.		
`start10:check sametag(%t_0, %t_1)`						
`start10:rax <- %t_0 == %t_1`						
`return rax`						

Note that, at `start7`, at our second pass, we can either reduce this to just checking `true`, or just deleting the check altogether.

13.6.5 Abstract Domains

We're now incorporating forward analysis with some numeric range analysis. We can call these types of analysis abstract domains. We have seen three different abstract domains for analysis:

- `Set<String>` for liveliness analysis (for register allocation).
- `Dict<String, Set<Tag>>` for forward data analysis.

There were two different types of tags:

- `Z | P | N`: information about numbers (positive, negative, etc.)
- `N | B | Nil | P`: other relevant tag information.

How do we unify these domains? We can consider ideas like:

- Frequency for variable use.
- Frequency of branches.
- Ranges of numbers.
- Booleans as true or false.
- Lengths of vectors (if we have constants, e.g., setting the tag to be the length of the vector).

[24] Note that this IR representation was generated by hand.
[25] Note that there's no overlaps in this set; we either have negative numbers, positive numbers, and zero.