1 Introduction to Compilers (Continued)

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 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`[1]:

- 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.

[1] 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
```