

```

use std::fs::File;
use std::env;
use std::io::prelude::*;
use sexp::*;
use sexp::Atom::*;

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

fn parse_expr(s : &Sexp) -> Expr {
    match s {
        Sexp::Atom(l(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")
    }
}

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"
    }
}

fn main() -> std::io::Result<> {
    let args: Vec<String> = env::args().collect();

    let in_name = &args[1];
    let out_name = &args[2];

    let mut in_file = File::open(in_name)?;
    let mut in_contents = String::new();
    in_file.read_to_string(&mut in_contents)?;

    let sExpr = parse(&in_contents).unwrap()
    let expr = parse_expr(&sExpr);
    let result = compile_expr(&expr);
    let asm_program = format!(
section .text
global our_code_starts_here
our_code_starts_here:
{}
ret
", result);

    let mut out_file = File::create(out_name)?;
    out_file.write_all(asm_program.as_bytes())?;

    Ok(())
}

```

src/main.rs

```

pub enum Sexp {
    Atom(Atom),
    List(Vec<Sexp>),
}
pub enum Atom {
    S(String),
    I(i64),
    F(f64),
}

```

Why is Vec<Box<Sexp>> or Box<Vec<Sexp>> not used above?

"(sub1 (sub1 (add1 73)))"

Assume we run main with a file containing the contents above.

What does the stack & heap look like when format!("mov rax, {}", *n) evaluates?


```
(let (x (let (y 10) (add1 y)))  
  (sub1 x))
```

- A. 9
- B. 10
- C. 11
- D. 12
- E. Error

```
(let (x (let (x 10) (add1 x)))  
  (sub1 x))
```

- A. 9
- B. 10
- C. 11
- D. 12
- E. Error

```
(let (x (let (x 10) (add1 x)))  
  (sub1 x))
```

- A. 9
- B. 10
- C. 11
- D. 12
- E. Error