```
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(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")
 }
}

fn compile_expr(e : &Expr) -> String {
 match e {
   Expr::Num(n) => format!("mov rax, {}", *n),
   Expr::Add1(subexpr) =>
A    compile_expr(subexpr) + "\nadd rax, 1",
   Expr::Sub1(subexpr) =>
B   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);        C
 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),
}
```

```
Vec<T> {
  int cap;
  int size;
  T* contents
}
```

fixed size indirection

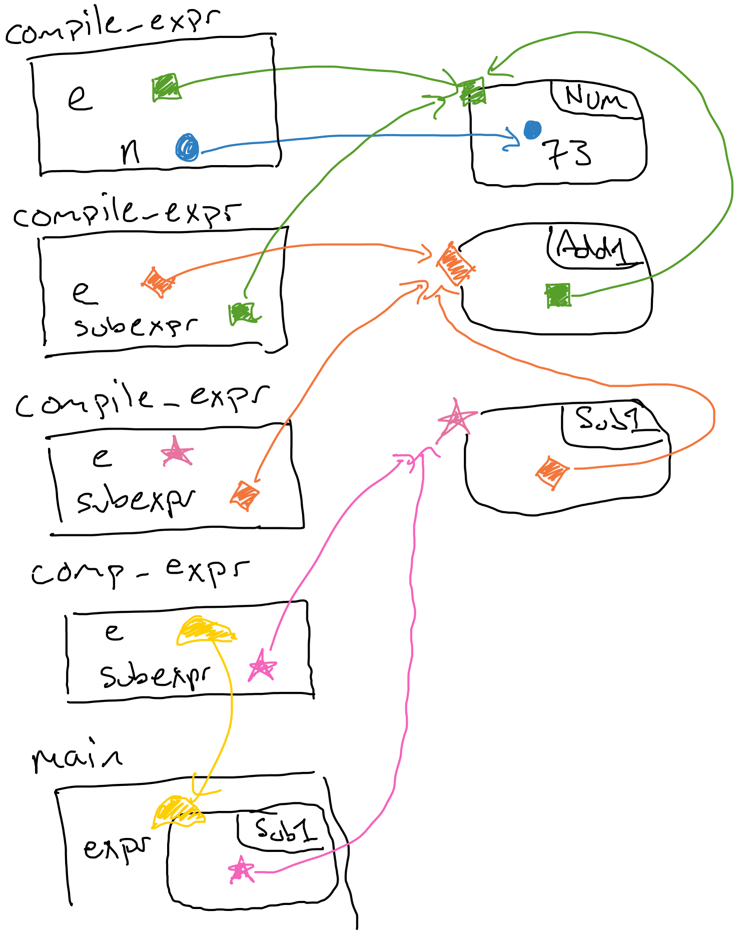
Why is Vec<Box<Sexp>> or Box<Vec<Sexp>> not used above?

Vec introduces fixed-size ind.

"(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’s add (+ <expr> <expr>) to the compiler!

```
(*
expr := <number>
      | (add1 <expr>)
      | (+ <expr> <expr>)
*)
enum Expr {
  Num(i32),
  Add1(Box<Expr>),
  Plus(Box<Expr>,
       Box<Expr>)
}
```

```
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::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"
```

```
let e1_instrs = compile_expr(e1, si);
let e2_instrs = compile_expr(e2, si + 1);
let stack_offset = si * 4;
format!("
    {e1_instrs}
    mov [rsp - {stack_offset}], rax
    {e2_instrs}
    add rax, [rsp - {stack_offset}]
")
```

(+ (+ 100 50) 2)

```
mov rax, 100
mov rbx, rax
mov rax, 50
add rax, rbx
mov rbx, rax
mov rax, 2
add rax, rbx
```

(+ 500 (+ 10 3))

```
mov rax, 500
mov rbx, rax
mov rax, 10
mov rbx, rax
mov rax, 3
add rax, rbx   ; 13
add rax, rbx
```

↳ hoping this would be 500!

```
(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