src/main.rs

```
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) =>
        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 expr = parse_expr(&parse(&in_contents).unwrap());
  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(())
}
```

abstract syntax "AST"

gen asm that puts the answer for e into RAX

if open errors, return that err from main

if parse errors, panic, or return val if no err

Makefile

```
test/%.s: test/%.snek src/main.rs
    cargo run -- $< test/$*.s

test/%.run: test/%.s runtime/start.rs
    nasm -f elf64 test/$*.s -o runtime/our_code.o
    ar rcs runtime/libour_code.a runtime/our_code.o
    rustc -L runtime/ runtime/start.rs -o test/$*.run
```

runtime/start.rs

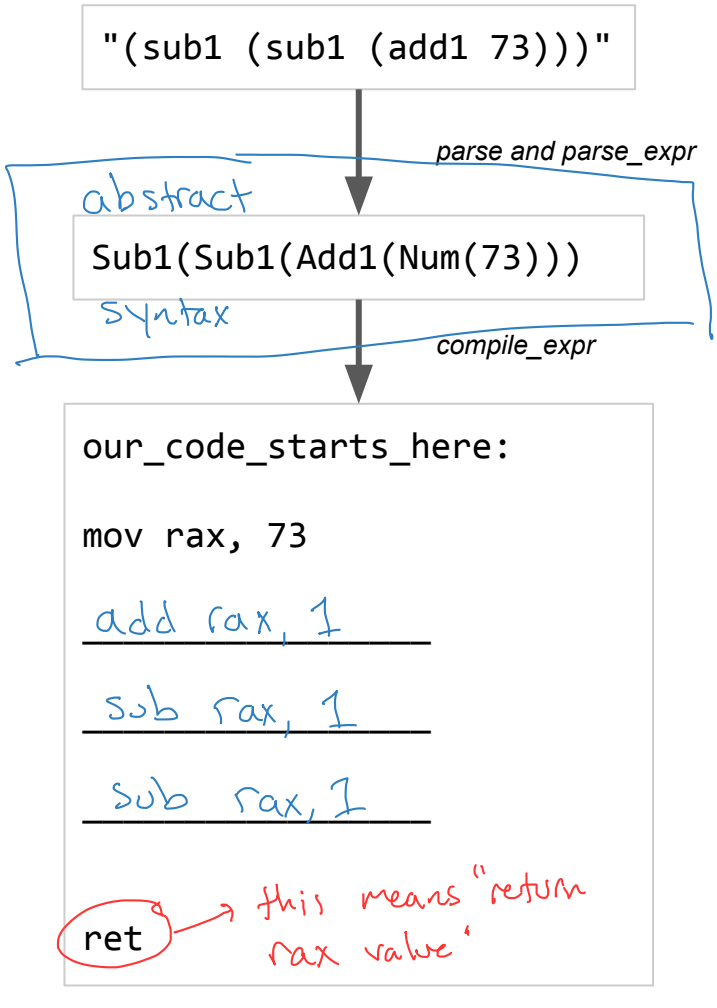
```
#[link(name = "our_code")]
extern "C" {
  fn our_code_starts_here() -> i64;
}

fn main() {
  let i : i64 = unsafe { our_code_starts_here()  };
  println!("{i}");
}
```

test/add.snek

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

Concrete syntax

```
$ make test/add.run
```



https://github.com/ucsd-cse131-fa25/2025-09-26-lecture

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

```
interface Expr { }
class Num { int n; }
class Add1 { Expr e; }
class Sub1 { Expr e; }
```

```
typedef struct Expr Expr;

struct Expr {
  enum { Num, Add1, Sub1 } tag;
  union {
    struct Num { int n; } Num;
    struct Add1 { Expr* e; } Add1;
    struct Sub1 { Expr* e; } Sub1;
  }
}
```

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

error[E0072]: recursive type `Expr` has infinite size

```
typedef struct Expr Expr;

struct Expr {
  enum { Num, Add1, Sub1 } tag;
  union {
    struct Num { int n; } Num;
    struct Add1 { Expr e; } Add1;
    struct Sub1 { Expr e; } Sub1;
  }
}
```

thing.c:7:24: error: field has incomplete type 'Expr' (aka 'struct Expr')
struct Add1 { Expr e; } Add1;

From crate sexp, see Cargo.toml

```
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 2)"`

`List(vec![Atom(S("sub1")),` _____

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

`List(vec![Atom(S("sub1")),`

_____

_____

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