CSE 231: Week 1 Adder

Ranjit Jhala *April 1, 2026*

What is *a Compiler?*

A function that maps an *input* string to an *output* string.

```
compiler :: String -> String
```

Typically, the *input* and *output* strings are *"programs"*

```
compiler :: SourceProgram -> TargetProgram
```

What is *a Compiler?*

For example, here are some well-known *compilers*

```
gcc, clang :: C          -> Binary          -- a.out
ghc        :: Haskell    -> Binary
javac      :: Java       -> JvmByteCode     -- .class
scalac     :: Scala      -> JvmByteCode
ocamlc     :: Ocaml      -> OcamlByteCode   -- .cmo
ocamlopt   :: Ocaml      -> Binary
gwt        :: Java       -> JavaScript      -- .js
v8         :: JavaScript -> Binary
nasm       :: X86        -> Binary
pdftex     :: LaTeX      -> PDF
pandoc     :: Markdown   -> PDF | Html | Doc
```

Key Requirements on output program:

1. Has the *same meaning* ("semantics") as input,
2. Is *executable* in relevant *context* (VM, processor, browser)

A Bit of History

Compilers were invented to avoid writing machine code by hand.

Richard Hamming — *The Art of Doing Science and Engineering*, p25:

> *In the beginning we programmed in absolute binary... Finally, a Symbolic Assembly Program was devised — after more years than you are apt to believe during which most programmers continued their heroic absolute binary programming. At the time [the assembler] first appeared I would guess about 1% of the older programmers were interested in it — using [assembly] was "sissy stuff", and a real programmer would not stoop to wasting machine capacity to do the assembly.*

John A.N. Lee, Dept of Computer Science, Virginia Polytechnical Institute:

> *One of von Neumann's students at Princeton recalled that graduate students were being used to hand assemble programs into binary for their early machine. This student took time out to build an assembler, but when von Neumann found out about it he was very angry, saying that it was a waste of a valuable scientific computing instrument to use it to do clerical work.*

What does a Compiler look like?

An input source program is converted to an executable binary in many stages:

- **Parsed** into an *Abstract Syntax Tree*
- **Checked** to make sure code is well-formed/typed
- **Simplified** to a convenient *Intermediate Representation*
- **Optimized** into (equivalent) but faster program
- **Generated** into assembly `x86`
- **Linked** against a run-time (usually written in C)

What is CSE 231?

- A *bridge* between two worlds
  - *High-level:* ML (**CSE 130/230**)
  - *Machine Code:* X86/ARM (**CSE 30**)

A sequel to both those classes.

- How to write **a compiler** for `Snek -> X86`
  1. Parsing
  2. Checking & Validation
  3. Simplification & Normalizing
  4. Optimization
  5. Code Generation
- But also, how to write **complex programs**
  1. Design
  2. Implement
  3. Test
  4. **Iterate**

How write a Compiler?

General recipe, applies to any large system — *gradually, one feature at a time!*

We will

- **Step 1** Start with a teeny tiny language,
- **Step 2** Build a full compiler for it,
- **Step 3** Add a few features,
- **Go to** Step 2.

(Yes, loops forever, but we will hit Ctrl-C in 10 weeks...)

Policies: Lectures

- We will not podcast lectures.
- We will have worksheets to be filled in and submitted in every lecture.
- We have a no-screens policy: students must keep their devices off during lectures.
- We require all exams be taken on the announced dates and times.

Policies: Grading

Your grade will be calculated from assignments, exams and worksheets.

- (**8–9**) **Assignments** [**30%**] are given periodically, typically at one or two week intervals. On each you'll get a score from 0–3 (Incomplete/No Pass, Low Pass, Pass, High Pass).
- (**2/3**) **Midterm Exams** [**50%**] There are three exams in the course, one in week 5 and one in week 9, given in the Friday discussion sections, and one in the finals week. Your top two exams will be counted.
- (**daily**) **Worksheets** [**20%**] Every lecture will come with a 1–2 page handout, that must be filled in and submitted at the end of the lecture. Credit is given for reasonable effort in engaging with the notes from the day on the handout. Turn in 75% of the worksheets to get full credit.

Course Outline

We will **write a compiler** for `Snek -> X86`

So we will write *many* compilers:

- Numbers and increment/decrement
- Local Variables
- Nested Binary Operations
- Booleans, Branches and Dynamic Types

- Functions
- Tuples and Structures
- Lambdas and closures
- Types and Inference
- Garbage Collection

What will you learn*?*

Core principles of compiler construction

- Managing Stacks & Heap
- Type Checking
- Intermediate forms
- Optimization

Several new languages

- `Rust` to write the compiler
- `C` to write the "run-time"
- `X86` compilation target

More importantly **how to write a large program**

- How to use types for design
- How to add new features / refactor
- How to test & validate

What do you need to know*?*

This class **depends very heavily** on CSE 130 / 230

- Datatypes (e.g. Lists, Trees, ADTs)
- Polymorphism
- Recursion

Also **depends on** CSE 30

- Experience with some `C` programming
- Experience with some assembly (`x86`)

Adder

A tiny language with 32-bit integers and three operations:
`add1`, `sub1`, and `negate`.

Goals for the first assignment

- See how **all** compiler infrastructure fits together *end to end*
- Go from source text to a running binary
- Build each piece from scratch

The Big Picture

"Compiling" an expression means generating assembly instructions that evaluate it and leave the answer in the `rax` register.

Result lands in the `rax` register — the x86-64 calling convention return register.

The pipeline:

```
source file (.snek) ---------> x86 binary (.run)
```

The Runtime (`runtime/start.rs`)

Rust program that calls our compiled code and prints the result

`unsafe` required by call into foreign code.

```
#[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}");
}
```

Key Points

- `#[link(name = "...")]`: expects precompiled `libour_code.a`
- `our_code_starts_here`: takes no args, returns a 64-bit int
- `unsafe` — needed by calls into foreign code

Assembly Basics

A hand-written assembly file that returns `31`

`ret` pops the return address from the stack and jumps to it — the caller sees whatever is in `rax`.

```
section .text
global our_code_starts_here
our_code_starts_here:
  mov rax, 31
  ret
```

- `mov rax, 31` — load `31` into the return register
- `ret` — return to caller (result is whatever is in `rax`)

Assembling and Linking

On Linux:

```
$ nasm -f elf64 test/31.s -o runtime/our_code.o
$ ar rcs runtime/libour_code.a runtime/our_code.o
$ rustc -L runtime/ runtime/start.rs -o test/31.run
$ ./test/31.run
31
```

On MacOS:

```
$ nasm -f macho64 test/31.s -o runtime/our_code.o
$ ar rcs runtime/libour_code.a runtime/our_code.o
$ rustc --target x86_64-apple-darwin -L runtime/
runtime/start.rs -o test/31.run
$ ./test/31.run
31
```

Steps

1. `nasm` **assembles** `.s` → `.o` (machine code + label info)
2. `ar` **archives** `.o` → `.a` (standard library format)
3. `rustc` **links** `.a` + `start.rs` → executable

Generating Assembly from Rust

Step 1 Get the source and destination file names from the command line arguments.

```
fn args() -> (String, String) {
  let args: Vec<String> = env::args().collect();
  (args[1].clone(), args[2].clone())
}
```

Step 2 Read the source file and returns it as a `String`.

```
fn read_source(in_name: String) -> String {
  let mut in_file = File::open(in_name).unwrap();
  let mut in_contents = String::new();
  in_file.read_to_string(&mut in_contents).unwrap();
  in_contents
}
```

Step 3 Convert Source `String` to Assembly `String`.

```
fn compile(program: String) -> String {
    let num = program.trim().parse::<i32>().unwrap();
    format!("mov rax, {}", num)
}
```

Step 4 Write the assembly string to the destination file.

```
fn write_asm(out_file: String, asm: String) ->
std::io::Result<()> {
    let asm_program = format!("
section .text
global our_code_starts_here
our_code_starts_here:
  {}
  ret
", asm);
    let mut out_file = File::create(out_file)?;
    out_file.write_all(asm.as_bytes())?;
    Ok(())
}
```

Bundle it all together

```
fn main() -> std::io::Result<()> {
  let (src_file, dst_file) = args();
  let src = read_source(src_file);
  let asm = compile(src);
  write_asm(dst_file, asm)
}
```

Makefile

Use `elf64` on Linux, `macho64` on Mac.

Automate the compile-assemble-link steps

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

test/%.run: test/%.s runtime/start.rs
  nasm -f macho64 test/$*.s -o runtime/our_code.o
  ar rcs runtime/libour_code.a runtime/our_code.o
  rustc --target x86_64-apple-darwin -L runtime/
runtime/start.rs -o test/$*.run
```

Then just run

```
$ make test/37.run
```

The Adder Language

Three pieces define any language we compile

1. **Concrete syntax** — the text the programmer writes
2. **Abstract syntax** — a Rust datatype our compiler uses
3. **Semantics** — the behavior (what values programs produce)

Concrete Syntax

```
<expr> :=
  | <number>
  | (add1 <expr>)
  | (sub1 <expr>)
  | (negate <expr>)
```

Uses **s-expression** style — parenthesized prefix notation

- Easy to parse (many libraries available)
- No ambiguity, no precedence rules
- Same style as Scheme, Lisp, WebAssembly

Abstract Syntax

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

`Box<T>` is a heap pointer — always a fixed 64-bit size, enabling recursive types.

Why `Box<Expr>` and not just `Expr`?

- `Expr` is recursive — Rust needs to know the size at compile time
- `Box<T>` is a heap pointer — always a fixed size (64-bit address)
- Same role as a pointer in a C struct

Semantics

Adder programs always evaluate to a single `i32`

Expression	Result
`Num(n)`	`n`
`Add1(e)`	`eval(e) + 1`
`Sub1(e)`	`eval(e) - 1`
`Negate(e)`	`eval(e) * (-1)`

Examples

- `(add1 (sub1 5))` → `5`
- `(negate (add1 3))` → `-4`

Interpreter (Warm-up)

```
fn eval(e: &Expr) -> i32 {
  match e {
    Expr::Num(n)     => *n,
    Expr::Add1(e1)   => eval(e1) + 1,
    Expr::Sub1(e1)   => eval(e1) - 1,
    Expr::Negate(e1) => -eval(e1),
  }
}
```

Test with `cargo test`

```
#[cfg(test)]
mod tests {
    #[test]
    fn test1() {
        let expr1 = Expr::Num(10);
        assert_eq!(eval(&expr1), 10);
    }
}
```

Parsing: S-Expressions

Add the `sexp` crate to `Cargo.toml`

```
[dependencies]
sexp = "1.1.4"
```

Parse a string into an `Expr`

```
use sexp::*;
use sexp::Atom::*;

fn parse(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(e))),
        [Sexp::Atom(S(op)), e] if op == "sub1" =>
          Expr::Sub1(Box::new(parse(e))),
        [Sexp::Atom(S(op)), e] if op == "negate" =>
          Expr::Negate(Box::new(parse(e))),
        _ => panic!("parse error"),
     }
    },
    _ => panic!("parse error"),
  }
}
```

- `Sexp::Atom(I(n))` — matches an integer literal
- `Sexp::List(vec)` — matches a parenthesized list

- Pattern guards (`if op == "add1"`) select which operation

Code Generation

```
fn compile(e: &Expr) -> String {
  match e {
    Expr::Num(n)     => format!("mov rax, {}", *n),
    Expr::Add1(e1)   => compile(e1) + "\nadd rax, 1",
    Expr::Sub1(e1)   => compile(e1) + "\nsub rax, 1",
    Expr::Negate(e1) => compile(e1) + "\nneg rax",
  }
}
```

Each recursive call compiles the subexpression, leaving the result in `rax`. Then one instruction transforms it.

Key idea

- Recursively compile subexpression — result lands in `rax`
- Then emit one instruction to transform `rax`

Example: `(sub1 (sub1 (add1 73)))`

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

Result: `72`

End-to-End Test

```
$ cat test/add.snek
(sub1 (sub1 (add1 73)))

$ make test/add.run
...

$ ./test/add.run
72
```

The Four Key Pieces

1. **Assembler** (`nasm`), **runtime** (`start.rs`): build executables
2. **Abstract syntax** (`enum Expr`): represent programs as data
3. **Parser** (`parse_expr`): `String<Src>` → `Expr`
4. **Code generator** (`compile`): `Expr` → `String<Asm>`

These same four pieces appear in every compiler.