EECS 483: Compiler Construction

Lecture 1:
Concrete and Abstract Syntax, Interpreters and Compilers

January 12, 2025

Announcements

- Office hours today after class, 3-4:30 in Max's office, Beyster 4628
- First homework to be released on Wednesday

Learning Objectives

1. Relation between **concrete** and **abstract** syntax
2. How we use a **parser generator** to generate a parser from a grammar.
3. How to represent programs as **abstract syntax trees**
4. How to implement basic interpreters, compilers and optimizers as recursive traversals of abstract syntax trees
5. Basics of x86 instructions and our calling convention

Extending the Snake Language

When we implement a compiler (to assembly) we need to address the following questions:

1. What is the syntax of the language we are compiling?
2. What is the semantics of the language we are compiling?
3. How can we implement that semantics in assembly code?
4. How can we generate that assembly code programmatically?

Snake v0.1: "Adder"

Today: add basic computation to adder, see how this affects the entire compiler pipeline.

Snake v0.1: "Adder"

Concrete Syntax	Answer
`42`	42
`add1(42)`	43
`sub1(42)`	41
`sub1(add1(add1(42)))`	43

Concrete Syntax: Grammar

<expr>:

| *NUMBER*

| `add1` `(` <expr> `)`

| `sub1` `(` <expr> `)`

Concrete Syntax: Grammar

‹expr›:

| *NUMBER*

| `add1` `(` ‹expr› `)`

| `sub1` `(` ‹expr› `)`

Concrete Syntax

The **concrete syntax** of a program is the textual representation of the program which is given as an input to the compiler.

The **compiler frontend**, specifically the **parser** performs two tasks:

1. **Recognition**: determine whether the input program is well-formed, reject it if it is not
2. **Parsing:** construct a more useful internal representation of the program: the **abstract syntax tree**

Concrete vs Abstract Syntax

The **concrete syntax** of a program is the textual representation of the program which is given as an input to the compiler.

- This includes many details which are not relevant to implementing an interpreter or a compiler such as: whitespace, comments, exact parenthesization, exact variable names etc.

The **abstract syntax tree (AST)** of a program is an abstraction of the concrete syntax that ignores some semantically irrelevant details of the concrete syntax.

Abstract Syntax abstracts Concrete Syntax

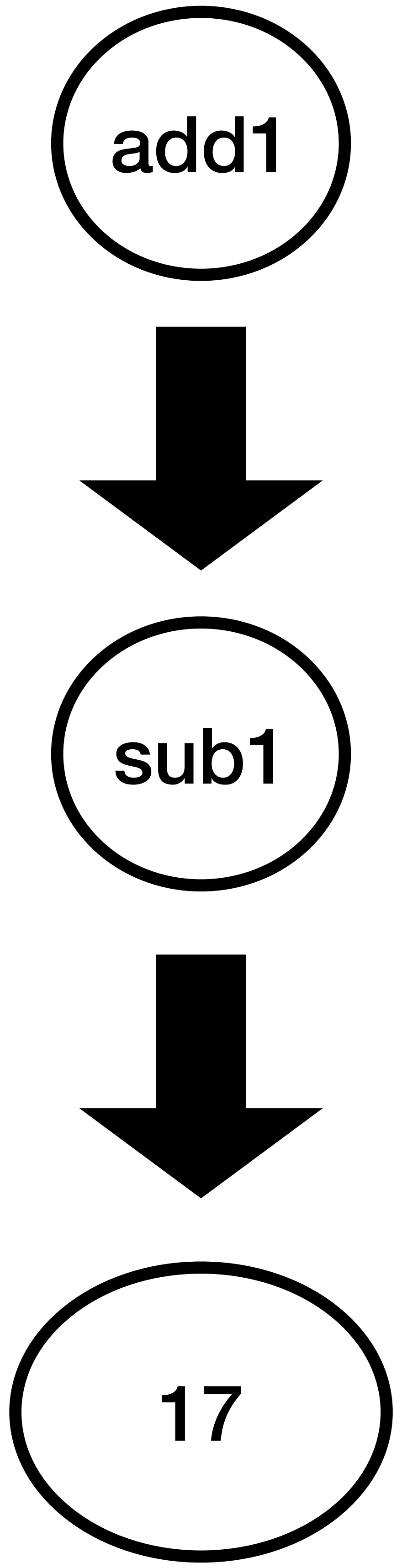
```
add1(sub1(17))
```

```
    add1
        (sub1
    (17       )
    )
```

```
add1(sub1(0x11))
```

all are **parsed**

into the same AST



Much of Concrete Syntactic Details are Irrelevant

Applicative

```
add1(sub1(17))
```

Lisp

```
(add1 (sub1 17))
```

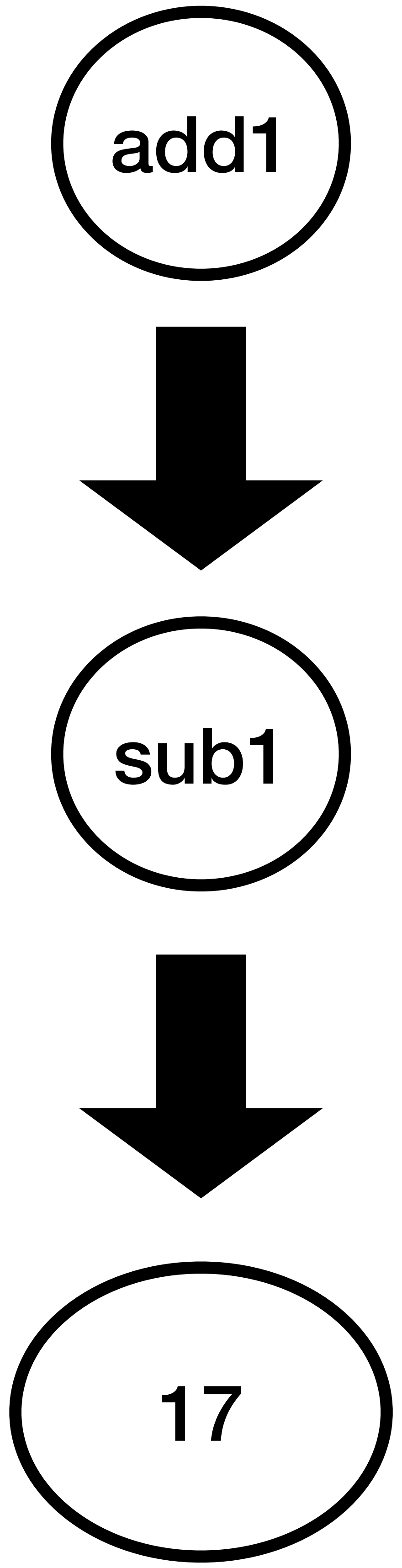
Method-call

```
17.sub1.add1
```

Assembly-like

```
17; sub1; add1
```

all can be represented by the same AST



Live Code

x86 Basics

x86 "Abstract Machine" abstracts from low-level hardware details into a reasonable machine for us to think about.

x86 Registers

16 general-purpose 64-bit registers

- rax, rcx, rdx, rbx, rdi, rsi, rsp, rbp, r8-r15

Each holds a 64-bit value, so 128 bytes of extremely fast memory.

Mostly indistinguishable to different instructions. rsp the "stack pointer" is the main exception.

There are also ways to operate on only a portion of the bits:

- eax (32 bits), ax (16 bits), al (low 8 bits), ah (high 8 bits) of rax

- for simplicity, we'll work with the full 64-bit versions unless required by certain instructions

x86 Instructions: mov

mov dest, src

Copies the value of src into the memory location dest.

Can be used for loads, stores, constants, complex address calculations.

For now:

```
mov rdx, 13
mov rax, rdi
```

Full details are very complex: https://github.com/xoreaxeaxeax/movfuscator

x86 Instructions: add

```
add dest, src
```

Semantics: Adds the value of src to dest and stores the result in dest

"dest += src"

(side-effect: updates the RFLAGS register)

```
mov rdx, 13
mov rax, rdi
```

Caveat: constants can only be 32-bit values!

Reference: https://www.felixcloutier.com/x86/add

x86 Instructions: sub

```
sub dest, src
```

Semantics: Subtracts the value of src from dest and stores the result in dest

"dest -= src"

(side-effect: updates the RFLAGS register)

```
sub rdx, 13
sub rax, rdi
```

Caveat: constants can only be 32-bit values!

Reference: https://www.felixcloutier.com/x86/sub

x86 Instructions: ret

```
ret
```

Semantics: pops a return address off of the stack (as determined by rsp) and jumps to it

Simplification: if you are implementing a function and rsp is unchanged from when you were called, this will return control to the caller.

More details when we come back to function calls.

Reference: https://www.felixcloutier.com/x86/ret

Extending the Snake Language

When we implement a compiler (to assembly) we need to address the following questions:

1. What is the syntax of the language we are compiling?

2. What is the semantics of the language we are compiling?

3. How can we implement that semantics in assembly code?

4. How can we generate that assembly code programmatically?

Snake v0.1: "Adder"

Extend adder to take an input

Grammar

⟨prog⟩: `def` `main` `(` `x` `)` `:` ⟨expr⟩

⟨expr⟩:

| *NUMBER*

| `x`

| `add1` `(` ⟨expr⟩ `)`

| `sub1` `(` ⟨expr⟩ `)`

Live Code

System V AMD64 Calling Convention (So Far)

Return value is stored in rax

First argument is stored in rdi

rsp points to the return address, so that ret returns if rsp is unchanged.

Summary

- Language extension: add1, sub1, input
- Concrete syntax, grammar, Recognizers
- Abstract syntax, Parsers and Parser generators
- Programming with abstract syntax trees: enum, pattern matching, recursive-descent
- x86 basics: registers, mov, add, sub, ret
- Basic optimization: compile-time vs runtime computation

For Next Time

- Try out today's live code, write tests, experiment with generating code.
- Work through the chapters 5,6,8 and 9 of the Rust book: https://rust-book.cs.brown.edu/