EECS 483: Compiler Construction

Lecture 3:
Complex Expressions, Evaluation Order, Basic Blocks

January 21, 2025

Announcements

- Yuchen will be holding office hours 3-4:30pm on Thursday the 23rd in Beyster Atrium in place of Max.
- Assignment 1 is due next Friday, the 31st.

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: Finish Adder by adding binary arithmetic operations

Snake v0.1: "Adder"

‹prog›: `def` `main` `(` *IDENTIFIER* `)` `:` ‹expr›

‹expr›:

| *NUMBER*
| *ADD1* `(` ‹expr› `)`
| *SUB1* `(` ‹expr› `)`
| *IDENTIFIER*
| *LET* *IDENTIFIER* *EQ* ‹expr› *IN* ‹expr›
| ‹expr› `+` ‹expr›
| ‹expr› `-` ‹expr›
| ‹expr› `*` ‹expr›
| `(` ‹expr› `)`

Abstract Syntax

```
enum Prim {
    Add1,
    Sub1,
    Add,
    Sub,
    Mul,
}

enum Expression {
  ...
  Prim(Prim, Vec<Expression>),
}
```

no constructor for parentheses

Precedence

Parser uses precedence rules (PEMDAS) to produce an AST

```
(2 - 3) + 4 * 5

(2 - 3) + (4 * 5)
```

both parse into the same AST:

```
Prim(Add,
  [Prim(Sub, [Number(2), Number(3)]),
   Prim(Mul, [Number(4), Number(5)])])
```

Semantics

In an expression e1 op e2, do we evaluate e1 and then e2 or vice-versa?

Does it make a difference in Adder?

Does it make a difference in realistic extensions of Adder?

```
print(6) * print(7)
```

Compiling Binary Operations

Why is compiling binary operations more complex than unary?

Compiling Binary Operations

Why is compiling binary operations more complex than unary?

Recall: current strategy is to store intermediate results in rax

```
((4 - 3) - 2) * 5
```

```
mov rax, 4
sub rax, 3
sub rax, 2
mul rax, 5
```

Compiling Binary Operations

```
(2 - 3) + (4 * 5)
```

```
mov rax, 2
sub rax, 3
?????
```

compound expressions have **implicit** intermediate results

solution: translate to a form where these intermediate results are explicit, and operations are only ever applied to **immediate** expressions (constants/variables)

```
let first = 2 - 3 in
let second = 4 * 5 in
first + second
```

Intermediate Representation

We add a new pass lowering our AST into an **intermediate representation**.

An **intermediate representation** is a language used internally in the compiler.

Typically, humans don't write programs in the intermediate representation directly, only generated by compiler passes.

Intermediate representation should be "closer" to the target language (x86) than the source program ASTs. I.e., the translation from intermediate representation to x86 should be relatively simple.

Static Single Assignment v1: Basic Blocks

The intermediate representation we use in this course is called **Static Single Assignment** (**SSA**).

For Adder, we only need a fragment of SSA: we will compile the source to a single **basic block**.

Static Single Assignment v1: Basic Blocks

Live code: AST for SSA

Static Single Assignment v1: Basic Blocks

Summary:

1. An SSA program consists of an entry point, a parameter and a block

2. A block is a sequence of primitive operations performed on immediately available values (variables or numbers) ending in a return statement.

3. Variables in SSA are **immutable**, just like our source language.

4. All bound variables in SSA should be globally unique.

Static Single Assignment v1: Basic Blocks

SSA programs aren't written by humans so they don't need a "concrete syntax"

but to make debugging easier, we will print SSA programs in the style shown below:

```
entry(x):
  y = add 2 x
  z = sub 18 3
  w = mul y z
  ret w
```

Static Single Assignment v1: Basic Blocks

Now we've reduced the compilation to two tasks:

1. "Lowering" our AST into an SSA program

2. Producing x86 assembly from an SSA program

SSA to x86

Since SSA is essentially a simplified version of Adder, we can apply the same techniques for generating assembly code from SSA. The only extension is that we need to handle binary primitives.

SSA to x86

```
entry(x):
  y = add 2 x
  z = sub 18 3
  w = mul y z
  ret w
```

```
;; entry(x):
mov [rsp - 8], rdi
;; y = add 2 x
mov rax, 2
mov r10, [rsp - 8]
add rax, r10
mov [rsp - 16], rax
;; z = sub 18 3
mov rax, 18
mov r10, 3
sub rax, r10
mov [rsp - 24], rax
;; w = mul y z
mov rax, [rsp - 16]
mov r10, [rsp - 24]
imul rax, r10
mov [rsp - 32], rax
;; ret w
mov rax, [rsp - 32]
ret
```

Adder to SSA

Live Code

Adder to SSA

Summary:

Translate Adder to SSA using **continuation-passing style:** expression lowering function is parameterized by a **continuation** consisting of

1. the name of the destination variable for the result.

2. a block of code to run **after** the compiled code places the result in the destination.

Need to generate **unique** names in this process to make sure that the generated variable names are all distinct and distinct from the original program variables