15-411: Compiler Design — Spring 2023

Recitation 0: Course Introduction — 20 January

Welcome to 15-411 Recitation!

In recitation, we will both review what was discussed in lecture and also give you practical tips, tricks, and advice that will help you when implementing your compilers. Please participate and ask questions! We're here to help you succeed.

Course Infrastructure

We'll be giving each of you access to two GitHub repositories:

- `dist`: Starter code, test cases, and tools (read-only)
- <Your team name>: Where you implement your compiler

To submit your compiler, use Gradescope's GitHub integration to upload the branch of your team repo that you wish to be graded. Gradescope will run the corresponding test suites and give you a score. We have also provided Docker to allow you to run the autograder yourself in an environment identical to that on Gradescope. More detailed instructions will be on the Lab 1 writeup.

Choosing a Language

- OCaml: If you don't have any particular desire to use another language, you should use OCaml. The majority of students choose this language each semester. It's a functional language that is very similar to SML but has many more standard libraries that make compiler implementation easier.
- Rust: It has native support for functional features while being a systems language that you can really optimize. However, Rust may not be as easy to pick up as OCaml, so we recommend using it only if you have some prior experience.
- Other: If you choose another language, make sure that you are very familiar with it and that it has the libraries and features you need to properly implement your compiler.

Collaboration

Here are a few tips for getting started collaborating on Lab 1 with your partner:

- You absolutely need to set up at least two meetings per week with your partner. Expect each of the meetings to last at least two hours. Use this time to discuss the overall architecture of your implementation and divide up the work.
- In your first meeting, you should spend a lot of time talking about your collaboration styles and making plans of attack.
- Use GitHub pull requests to read and review your partner's code. One partner makes a pull request, and the other reads it, makes comments, and eventually merges it into master.
- It might be tempting to divide the work between frontend and backend, but this is usually a bad idea because (1) the backend will require more work than the frontend and (2) it will be harder for both partners to gain knowledge of the entire system.
- Since Lab 1 is significantly easier than the later labs, it is **highly recommended** that you spend time on your register allocator. You'll save time later on and be able to focus on other parts of the compiler. Lab 1 checkpoint is intended to help you with this.

x86 Assembly - Resources

The backend of your compiler will demand a lot of work with x86 assembly. There are a lot of resources out there to help you with assembly, so we encourage you to search around! We also recommend these sites:

- `https://www.felixcloutier.com/x86/`: Very good instruction reference
- `https://godbolt.org/`: gcc compiler explorer that lets you input C code, and shows you the assembly produced by gcc. Very useful for debugging the assembly produced by your own compiler!

Inductive Definitions and Inference Rules

As you'll soon see, it's convenient to define the rules and structures of a programming language inductively. For example, you could say that

> "If a is an expression and b is an expression, then $\texttt{plus}(a, b)$ is also an expression."
> "Integer contants are expressions."

and use these rules (and others) to build up an entire mathematical expression.

To make it easier to express more complex inductive systems, such as an entire programming language, we express inference rules using the following form:

$$\frac{J_1\ J_2\ \ldots\ J_n}{J}$$

We call $J_1 \ldots J_n$ and J *judgments*. $J_1 \ldots J_n$ are the *premises* of the rule and J is the *conclusion*. (Sometimes you may see more than one conclusion; this is shorthand for two rules with the same premises). In this form, we could express our example rules from above as

$$\frac{a\ \textsf{exp} \quad b\ \textsf{exp}}{\texttt{plus}(a,b)\ \textsf{exp}}\ A_1 \qquad \frac{n \in \mathbb{Z}}{n\ \textsf{exp}}\ A_2$$

Checkpoint 0

Given `zero` to denote the number 0 and $\texttt{succ}(n)$ to denote the successor of n, write two rules that inductively define the judgment "n nat" to describe the natural numbers.

Now, write two more rules that inductively define the judgment "$\textsf{sum}(a, b) = c$" to mean that the sum of the natural numbers a and b is equal to c.

Typing Judgements

Arguably the most important use case of judgements and inference rules in compilers is for checking whether a programs is well-typed. The *context* (or *type environment*) Γ assigns types to variables. This is often defined explicitly with

$$\Gamma ::= \cdot \mid \Gamma, x{:}\tau$$

where we write $\Gamma(x) = \tau$ when $x{:}\tau$ in Γ. The typing judgment for expressions

$$\Gamma \vdash e : \tau$$

verifies that the expression e is well-typed with type τ, assuming the variables are typed as prescribed by Γ. Most of the rules are straightforward; we show a couple.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \mathsf{bool}}{\Gamma \vdash e_1 \texttt{ \&\& } e_2 : \mathsf{bool}}$$

Tracing Through the Backend

In this recitation, we're going to discuss an example of the processing done by the compiler backend. Since you won't have to touch the frontend for Lab 1 (unless you opt to not use any starter code), we'll leave it for a future week. Here's the code and AST we'll use for the example:

```
int main() {
  int x = 42;
  int z;
  if (x % 2 == 0) {
     x++;
     z = 1;
  } else {
     z = -1;
  }
  int y = 1;
  while (y <= x - 1) {
     y = x + y;
  }
  return z * y;
}
```

```
declare(x, seq(
  assign(x, const(42)),
  declare(z, seq(
     if(compare(mod(x, const(2)), const
        (0), EQ), seq(
       incr(x),
       assign(z, const(1))
     ),
       assign(z, neg(const(1)))
     ),
     declare(y, seq(
       assign(y, const(1)), seq(
         while(
           compare(y, minus(x, 1),
              LEQ),
           assign(y, add(x, y))
         ),
         return(times(z,y))
       )
     ))
  ))
))
```

Code Generation

As discussed in lecture, we use the "maximal munch" algorithm to generate abstract 3-address assembly from IR. For each line in the IR, we recursively pattern-match as deep as possible into each sub-expression and generate lines of assembly at each step. The top-down formulation of the rules is shown as follows:

e	codegen-expr(d, e)
c	$d \leftarrow c$
x	$d \leftarrow x$
$e_1 \oplus e_2$	codegen-expr(t_1, e_1), codegen-expr(t_2, e_2), $d \leftarrow t_1 \oplus t_2$

s	codegen-stmt(s)
$x = e$	codegen-expr(x, e)
return e	codegen-expr(`%rax`$, e)$, ret

A bottom-up approach can also work:

e	codegen-expr(e)	up
c	$\cdot$	c
x	$\cdot$	x
$e_1 \oplus e_2$	$t \leftarrow$ codegen-expr$(e_1) \oplus$ codegen-expr(e_2)	t

s	codegen-stmt(s)
$x = e$	$x \leftarrow$ codegen-expr(e)
return e	`%rax` $\leftarrow$ codegen-expr(e), ret

In the former, the destination temporary is supplied as an argument to codegen-expr. In the latter, it is returned by codegen-expr.

Checkpoint 1

Translate the above AST into abstract 3-address assembly using the bottom-up approach. You do not need to turn it into SSA form here. Answers may differ depending on the exact rules you're using in your head to generate instructions.