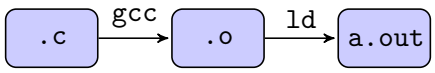
ECE251: Programming Languages & Translators Fall 2010

Lecture 1 — September 14, 2010

Patrick Lam *version 1*

Compiler courses traditionally teach you how to build a compiler for a general-purpose programming language.



This is difficult and useless to most of you.

This course will be different. We will instead talk about using traditional compiler techniques to build domain-specific languages.

Examples are everywhere. Here are some:

About me. My core expertise is in compilers and static analysis. I'm excited to be teaching (and re-developing) this course; although it is the last offering of ECE251, it will form the core of the new ECE351 course. Come and see me anytime my door is open, and especially during office hours.

Highlights from the course outline

I won't reread the course outline, but I will point out a few nonstandard details.

Labs. This word has two meanings in the context of this course. (1) During selected lab hours, you may see TAs in RCH108; see the course website for who's where when each week. (2) You hand in two marked "labs" (also known as assignments) during the term. (L1 is a data processing language and L2 is an SQL parser). They will help you develop expertise for the course project. *Don't hand in things that don't compile!*

Tutorials. Not mandatory. Held when appropriate. Next week: XML reading library. Week after that: Java parsing libraries. *Q: Who would attend Friday evening tutorials?*

Lateness. Grace days like PDEng. Handing in something will never lower your mark.

Lecture notes. I'll make typed lecture notes with blanks available the night before lecture on the course web site. On a best-effort basis I'll try to prepare them a week in advance, but life is hard sometimes. I recommend that you print the lecture notes beforehand and fill them in during lecture.

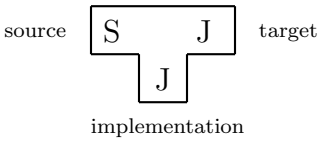
Puzzle

How do you build the first compiler?

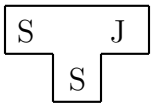
Usually, you'd like to build a compiler for (general-purpose) language X in language X itself. How can you pull that off?

(Aside: Say all of our computing infrastructure got destroyed, but we still know everything we know today. How long would it take to get back to where we are? What are the obstacles?)

Example. Say you want to compile Scala into Java. We'll use the following notation.

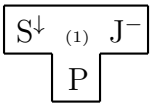


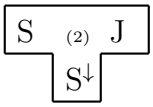
This is unfortunate; we'd rather write a Scala compiler in Scala. So, what we actually want is:



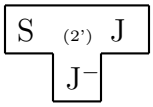
But we can't run that yet, since we can't run Scala code without a Scala compiler!

Let's say that $S^{\downarrow}$ is a simple Scala subset, J^{-} is inefficient Java, and P is our second-favourite, extremely portable programming language. We can then implement two compilers:

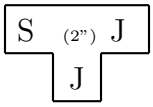




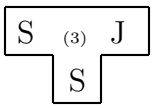
By executing (1) with input (2), we can get:



which is a Scala compiler producing Java, expressed in inefficient Java. To get a more efficient compiler, just run (2') on (2) to get:



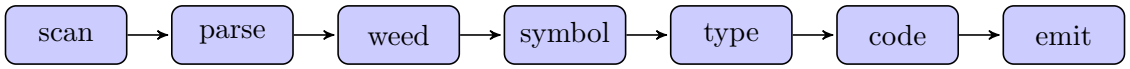
Compiler (2") is still written in Java (we wouldn't want to hack on it), but it's efficient and we can now compile (2) with it. We can also upgrade (2) to use all of Scala:



which is what we originally wanted; also, now we can compile (3) with (2") and with itself.

Parts of a compiler

Next, we'll explain the parts of a compiler that we'll see in this course. See Section 1.6 of the textbook for more information.

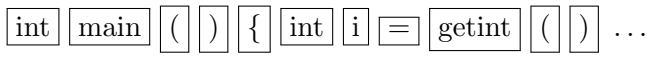


(I've omitted "optimize" between code and emit. Compilers can extensively optimize code to make it run faster.)

Scanning. In this phase, we convert the input into tokens with a *lexer*. Here is some input:

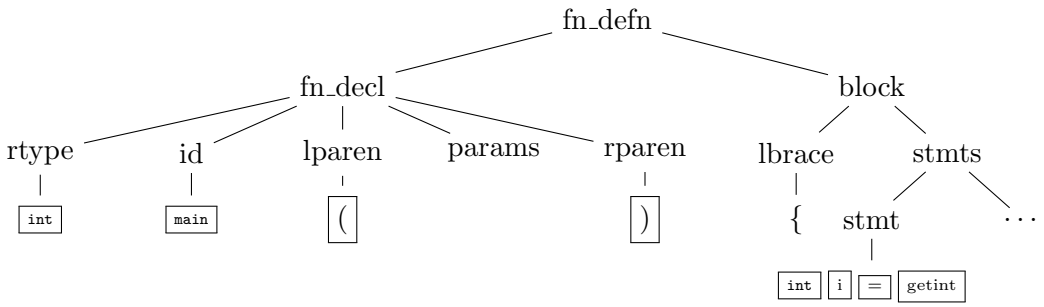
```
int main() {
  int i = getint(), j = getint();
  // etc...
```

which would lex to:



We got rid of whitespace and newlines, collected line and column numbers, and combined letters into tokens. This can be done with a Mealy machine, but we'll call the machinery a *nondeterministic finite automata* because we're in computer science land here.

Parsing. Next, we convert the stream of tokens into a *parse tree*, which conforms to the language's context-free grammar. (We'll talk about that later.)



Weeding. The parse tree is hard to handle, so we convert it to an Abstract Syntax Tree.

```
interface FunctionDefn {
  public Type getRType();
  public String getName();
  public List<Param> getParams();
  public Body getBody();
}
```

We'll probably ensure that AST implementations are easily walkable and visitable.

Symbol table. During compilation, we need to have information about names and types for e.g. function names (`main`, etc) and variable names (`i`, `j`). The symbol table is the data structure for storing this information, and it interacts with scopes.

Type checking. We ensure that operations are type-consistant, e.g. in non-JavaScript languages, you're not trying to add strings to integers, or calling methods that don't exist.

Code generation. We're now ready to generate (naive) code from the type checked AST, using the symbol table. This code won't run very fast, but it will run.

Optimization. Beyond the scope of this class, but it's possible to do a lot of work to make programs run fast, even faster than hand-written assembly for modern architectures. (Want to know more? Take ECE459, "Programming for Performance"!)

Emit. Write the optimized code to disk.