Introduction - Part 2

COMP 520: Compiler Design (4 credits)
Professor Laurie Hendren
hendren@cs.mcgill.ca
TRF 10:30-11:30, MC 103
RECAP and Some Ideas from the Readings!

Please note that these topics are discussed using the blackboard, and you may want to take notes on this discussion.

- Why study compilers?
- Who has a compiler with them today?
- General-Purpose vs. Domain-Specific Languages
- Interpreters vs. compilers
- Compilers that generate assembly code or machine code (pure, augmented, virtual machine)
- Ahead-of-time versus JIT compilers
The top 10 list of reasons why we use C for compilers:

10) it’s tradition;
9) it’s (truly) portable;
8) it’s efficient;
7) it has many different uses;
6) ANSI C will never change;
5) you must learn C at some point;
4) it teaches discipline (the hard way);
3) methodology is language independent;
2) we have flex and bison; and
1) you can say that you have implemented a large project in C.
The top 10 list of reasons why we use Java for compilers:

10) you already know Java from previous courses;
9) run-time errors like null-pointer exceptions are easy to locate;
8) it is relatively strongly typed, so many errors are caught at compile time;
7) you can use the large Java library (hash maps, sets, lists, . . .);
6) Java bytecode is portable and can be executed without recompilation;
5) you don’t mind slow compilers;
4) it allows you to use object-orientation;
3) methodology is language independent;
2) we have sablecc, developed at McGill; and
1) you can say that you have implemented a large project in Java.
Bootstrapping as illustrated in text, "Crafting a Compiler"

Figure 1.2: Bootstrapping a compiler that generates VM instructions. The shaded portion is a portable compiler for $L$ that can run on any architecture supporting the VM.
How to bootstrap a compiler (SCALA example):

- we are given a source language (L in the reading), say SCALA; and
- a target language (M in the reading), say Java.

We need the following:

```
     S   J
source  J
      J
implementation
     S   J
target
```

Of course, actually we like SCALA much better than Java and would therefore rather implement SCALA in itself:

```
     S   J
source  S
      S
implementation
     S   J
target
```
Define the following:

- $S\downarrow$ is a simple subset of SCALA;
- $J^-$ is inefficient Java code, and
- $P$ is our favourite programming language, here “Pizza”.

We can easily implement:

![Diagram 1: $S\downarrow_1 J^- P$]

and in parallel, using $S\downarrow$, we can implement:

![Diagram 2: $S_2 J S\downarrow$]

using basically our favourite language.
Combining the two compilers, we get:

which is an inefficient SCALA compiler (based on generated Java code) generating efficient Java code.

A final combination gives us what we want, an efficient SCALA compiler, written in SCALA, running on the Java platform.