Compiler Project: Introduction and Phase 1

Computer Science 371
Amherst College
Fall 2015

The project for this year's class is to write a compiler for Minijava, a subset of Java. You'll write your compiler in Java and will use a "compiler compiler" called SableCC. (You are not absolutely required to work in Java, but there will be challenges if you strike out on your own. In particular, the compiler-building tools that we discuss and the materials that I give you will all be based on Java.)

This document will discuss everything you need to know to get started on the project. Please note that the due date for the first phase of the project is **Friday, September 18**.

1 Programming Environment

You should work on a Unix system or a Mac for this course. You can use Eclipse or command line tools (such as emacs, javac, java).

One good option is to do your work on the department's Unix network. One way to access this network is to use the workstations in room 007. We are awaiting the arrival of some new workstations, which will make that room a fantastic resource. Until they arrive, you can use the existing machines in the following way:

1. On the initial login screen, click on the red button.
2. Choose "remote login".
3. Choose **vega.cs.amherst.edu.** That's our central server.
4. Login with your regular Amherst username and password.
5. When you're done, click on the word "Menu" at the bottom left.
6. Click on the Logout button, the second one from the bottom.

Our network shares the file system with remus.amherst.edu and romulus.amherst.edu, so you can work there, too. You can, of course, open an X connection from a laptop to any of these machines. I will set up an account and issue a lab key to each of you.

You are free to do your work on non-departmental systems. Any Unix-based system with Java 5 or higher should work. You can use a Mac if you download and install the Java developer code. I will distribute a tar file containing all of the files in each assignment.

If you use Eclipse, please make sure that you don't change any package names. Changing things makes it more difficult for me to examine and grade your work.

2 Working with a Partner

You can work with a partner on the project, and I encourage you to do so. Let me know if you are working with someone and I will set up a directory in which you can share files.

3 Copying the Project Directories

The distribution directory for files related to the project is `~lamcgeoch/cs371` . To copy the initial set of files, you should:

1. Create a working directory, for example, `cs371`. (If you have a partner, I'll do this step for you.)

2. cd into your working directory, and then issue the commands

```
cp -r ~lamcgeoch/cs371/hw1 .
cp -r ~lamcgeoch/cs371/sablecc-3.2amh .
```

   Note the period (preceded by a space) at the end of each of these lines. These commands do recursive copies of two of my directories into "dot", your current directory.

If you are working in a group directory, also issue the command

```
chmod -R a+w hw1 sablecc-3.2amh
```

If you are working on a machine not on our network, you'll need to find some other way to transfer the files.

4 A Tour of the Directories

Your working directory will initially contain two subdirectories, `sablecc-3.2amh` and `hw1`. The first directory contains all the code for the SableCC system, and you won't have to worry about the files in it. The `hw1` directory contains everything else that you'll need for the first part of the project.

(The code in `hw1` assumes that `hw1` and `sablecc-3.2amh` are both contained in a single parent directory. If this isn't true, you'll need to adjust certain files.)

Let's look inside `hw1`:

grammar: This file describes the grammar for Minijava. A copy of this file is attached. (The Minijava language is also described later in this document.) The grammar file is used by SableCC to generate a *lexer* and a *parser*. We'll talk (a lot!) about lexers and parsers in class.

The lexer locates the *tokens* that appear in the input file. Based on patterns describing the various tokens, it will seek the longest sequence of characters that matches some pattern. It will then return the corresponding token. If the sequence of input characters, the lexeme, matches more than one pattern, the token for the pattern that appears first in the grammar file will be the one that's returned.

The lexer can be in different states as it works. In our case, we'll use two states, *normal* and *comment*, depending on whether or not we are in the middle of processing a comment.

The Helpers section defines shortcuts that will let us refer to particular characters or strings. *letter* and *digit* are obvious. *all* refers to all Unicode characters. *tab* refers to the tab character, which is encoded by 9th Unicode character. *eol* gives the patterns that can mark the end of a line. *schar* gives that characters that can appear without special meaning in a string literal, at least the

way that I'd like to define it today. *sitem* elaborates on this, indicating that `\\`, `\n`, `\t`, and `\"` can all appear in strings. (Do you know what those codes mean?)

The Tokens section defines the patterns for the tokens. A vertical bar (`|`) means *or*, a star (`*`) means zero or more repetitions, and a plus (`+`) means one or more repetitions.

The Ignored Tokens section names the tokens that will be ignored when we get to the task of parsing.

You can ignore the Productions section for now. It affects the creation of the parser and describes how tokens work together to create legal Minijava programs.

minijava: This directory will contain all of the Java code for your compiler. Initially it contains just two files: `Main1.java` and `ErrorHandler/ErrorHandler1.java`. You will work on these files in the initial phase of the project, and I'll talk more about them in a moment.

tests1: This directory contains some sample Minijava programs. Both `Bad1.java` and `Bad2.java` contains lexical errors, in other words the lexer will become confused when it tries to identify tokens. When this happens, a `LexerException` will be thrown. The file `compile` is a script that you will use to compile Minijava programs. I'll talk more about this later.

Makefile1: This is a special script that you'll use to compile your compiler. Let's discuss this now.

5 Getting Ready

Move into your `hw1` directory and issue two more commands:

```
ln -s Makefile1 Makefile
mv minijava/ErrorHandler/ErrorHandler1.java minijava/ErrorHandler/ErrorHandler.java
```

The effect of these commands is to 1) let `Makefile1` be known by a second name, and 2) to rename the error-handler class.

6 Compiling Your Compiler

To compile your compiler, make sure that you are in your `hw1` directory and then type

```
make
```

The Makefile, a copy of which is attached, runs SableCC on the grammar file. This creates code (in directory `minijava`) for the lexer and the parser. It then compiles everything in `minijava`.

If you modify any part of your compiler, simply run `make` again to rebuild it. SableCC will run again only if the grammar has changed. (The ability to do conditional rebuilding is one of the key elements of the `make` system.)

If you type

```
make distclean
```

all of the class files and all of the files generated by SableCC will be deleted. This is an appropriate thing to do before trying to distribute your program to others. If you type

```
make backupclean
```

all of the emacs backup files will be deleted. (I've made this separate from `distclean` because I sometimes run `distclean` just to tidy up my directories. On the other hand, I'd like to save backup files until I'm really sure that I don't need them.)

If you ever decide to modify `Makefile`, be very, very sure that there's a newline (carriage return) character at the end. Otherwise the last line is ignored!

7 Files Created by SableCC

When you run SableCC, four directories are created within the `minijava` directory: `parser`, `analysis`, `node`, and `lexer`. We'll ignore the first two during the first phase of the project. Two classes within `node` are important at this point:

- `Token`: an object of this type (or of some subclass) is returned when a token is found. The supported instance methods include:
  - *getText()*: returns a String, the lexeme for the token.
  - *getLine()*: returns the line number for the token, with 1 being the first line.
  - *getPos()*: returns the position of the first character of the token in the line, with 1 being the first position. A tab character counts as a single character, i.e. it occupies a single position.
- `EOF`: this is a subclass of `Token`. An object of this type is returned when the end of an input file is reached.

Two classes within `lexer` are important, `Lexer` and `LexerException`. A `Lexer` is an object representing a lexer. The constructor requires one argument, a `PushbackReader`. A `PushbackReader` is a kind of `Reader` object that supports the ability to "unread" characters. It is documented in the `java.io` package.

A `Lexer` object supports a method called *next()*, which returns the next Token obtained from the input file. Two kinds of exceptions can be thrown, `IOException` (if something goes wrong in the act of reading) and `LexerException` (if an actual lexical error, for example the presence of an illegal character, occurs in the input file).

Running the *getMessage()* method on a `LexerException` object will return a message similar to the following:

```
[11,16] Unknown token: &
```

In this example, 11 is the line number and 16 is the position within the line.

8 Main1.java

You now have all the pieces needed to understand the file `Main1.java`, a copy of which is attached. Note that the file begins with the line `package minijava;`. All classes that you create must be declared to be in `minijava` or in a subpackage, with the precise choice of a package depending on the directory that contains the class.

Note that the main method constructs an `ErrorHandler` object and then uses it if a `LexerException` occurs.

9 Testing Your Compiler

Your compiler doesn't do much so far. It will simply print each token found in a file, even if the token is something that will ultimately be ignored, such as white space or a comment. To test it, cd into the test directory, and then issue a command like:

```
./compile Ex1.java
```

This runs the compilation script on the given file.

10 Your First Task: Writing a Good ErrorHandler

If a `LexerException` occurs, the method `errorHandler.getLongMessage(message)` is called, where `message` is similar to the message shown above. You should modify `ErrorHandler` so the message would be more verbose, something like:

```
Error during parsing: [1,7] Unknown token: &

The error was detected at line 1, column 7.

Here is line 1.  The carat mark (^) indicates where the error was detected.

 if (i & j) {
       ^
```

Your code will need to extract the line and column number from the message. Be sure that it works even if there are tab characters in the input file.

To debug, you might try fabricating a short error message string for each token you read. For example,

```
[11,16] Found token: id
```

Then try calling `getLongMessage` with that string.

11 Submitting Your Work

You should do an electronic submission of your entire hw1 directory. To do this, first create a tar file for your entire `hw1` directory. Then submit it with a browser by going to `http://www.cs.amherst.edu/submit`, selecting this course, going into Homework 1, and uploading your tar file. If you are working with a partner, only do one submission.