Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2019 Handout — Scanner-Parser Project Friday, Sep 7

PUBLIC: Monday, Sep 16, 11:59 pm

DUE: Wednesday, Sep 18, 11:59 pm

This project consists of two segments: lexical analysis (scanning) and syntactic analysis (parsing).

Preliminaries

Project Policy. Make sure to check the class policy regarding the collaboration, late days, building and running the tests, and third-party libraries:

- https://github.com/6035/fa19

Project Skeletons. In this section we will describe the infrastructure that is provided for the project.

- The Java skeleton is located at http://github.com/6035/java-skeleton.
- The Scala skeleton is located at https://github.com/6035/scala-skeleton.
- The Haskell skeleton is located at https://github.com/6035/haskell-skeleton.

You are encouraged to use this infrastructure, but you may also choose to ignore it and design your infrastructure from scratch. In such a case, you will still need to replicate all command line options and the functionality required for the scripts that build and execute the compiler.

Please get in touch with the course staff (`6.035-staff@mit.edu`) if you plan to use Haskell.

Provided Infrastructure

Java

For Java, a skeleton compiler infrastructure has been provided that includes a typical compiler directory organization, the ANTLR parser and scanner generator, and an Apache Ant build system. The Java package name for the sources provided is `decaf`.

The directory structure and the provided files are as follows:

```
.
|-- build.sh
|-- run.sh
|-- build.xml
`-- lib
    |-- antlr.jar
`-- src
    `-- edu
        `-- mit
            `-- compilers
                |-- Main.java
                `-- grammar
                    |-- scanner.g
                    |-- parser.g
                `-- tools
                    |-- CLI.java
    `-- decaf
        `-- Parallel
            |-- Analyze.java
```

The `lib` directory includes libraries that are needed during a *run* of your compiler; right now it contains the ANTLR tool. `build.xml` is the ant build file for your compiler. The *src* directory contains all your source code for the compiler. `parser.g` is a skeleton parser grammar and `scanner.g` is a skeleton scanner grammar. `Main.java` is the skeleton driver and `CLI.java` is the command-line interface library. Finally, `build.sh` and `run.sh` call `ant` and `java` in order to build and run your compiler, and are used by the TAs when grading your compiler.

Scala

The Scala skeleton infrastructure is similar to the Java infrastructure. There is the same top-level `lib` directory and `build.xml` file that you will use to build your code. Your source code also goes in `src`. The `parser` directory is a clone of the essential parts of the Java skeleton code, including the ANTLR `scanner.g` and `parser.g` files. The main differences are that the command-line interface library lives in `src/util/` and the equivalent of `Main.java` is `src/compile/Compiler.scala`. Also, we have included a sample unit test for convenience in the `unittests` directory, which may be run with the command: `ant test`.

Haskell

The Haskell skeleton code looks rather different:

```
.
|-- build.sh
|-- run.sh
|-- make-tarball.sh
|-- decafc.cabal
|-- Setup.hs
`-- alex
    |-- AlexTemplate-ghc-nopred
    |-- AlexWrapper-6.035
`-- src
    |-- CLI.hs
    |-- Configuration.hs
    |-- Main.hs
    |-- Parser.y
    |-- Scanner.x
    `-- Configuration
        |-- Types.hs
```

In the root directory, `build.sh` and `run.sh` serve the same roles. `make-tarball.sh` serves the same role as the `ant tar` target. `decafc.cabal` is the file that tells Cabal how to build the code, similar to ant's `build.xml`; however, you will probably want to use `build.sh` to invoke Cabal with some useful options rather than running it manually. The `alex` directory contains files for Alex, which will generate your scanner from the `Scanner.x` file. You do not need to modify any files in this directory. `CLI.hs`, `Configuration.hs`, and `Types.hs` are all components of the command-line parser supporting the same interface as the Java and Scala skeletons. They are invoked by `Main.hs`. You will need to update `Main.hs` and `Configuration.hs` for each project to reflect the stages of your compiler that you will execute; however, for this project you do not need to do that. The main files you will be editing are the skeleton scanner grammar `Scanner.x` and the skeleton parser gramamr `Parser.y`, used by Alex and Happy (our chosen parser generator for Haskell) to generate your scanner and parser.

Build System

Please review the Ant build file, `build.xml`, that is provided. For more information on Ant, please visit:

```
http://ant.apache.org/
```

To build the system, execute `ant` from the root directory of the system.

The build file includes tasks for running ANTLR on your scanner and parser grammars, compiling the sources, packaging the compiler into a jar file, packaging up your project for submission, and cleaning the infrastructure. The default rule is for complete compilation and jar creation. The build file creates a few directories during compilation: `classes`, `autogen`, and `dist`. The `classes` directory contains all `.class` files created during compilation. The `autogen` directory contains all the generated sources for the compiler (from scanner and parser grammars). The `dist` directory contains the jar archive file for the compiler and any other runtime libraries that are necessary for running the compiler. Running `ant clean` will delete these three directories. You should not add these directories to your repository.

The Scala ant buildfile contains the same targets, and generates the same directories, although in slightly different places.

The Cabal buildfile can be run with `cabal build`, `cabal configure`, and `cabal clean`; you may need to run `cabal configure` after running `cabal clean` if you are not using the `build.sh` script to build the compiler. It places all of its build files in the `dist` folder.

Getting Started

First, you should make sure you've fully configured git and Github. That can be done by following Github's instructions at this link: https://help.github.com/articles/set-up-git

After that, a good place to start would be to initialize your personal repository and then add in the skeleton code and Decaf test cases for the scanner and parser. For Java, this can be accomplished with the following commands on Athena (substitute the appropriate locker / repository names for Scala or Haskell as necessary, and make sure to read the Athena handout if using Scala):

```
# Initial setup:
add -f java git
mkdir -p ~/Private/6.035
cd ~/Private/6.035
mkdir <github username>6035
cd <github username>6035
git init

# Acquire skeleton code
git remote add skeleton https://github.com/6035/java-skeleton.git
git pull skeleton master
```

You need to send your Github username to the course staff for permission to access the `tests` repository. You will receive an automated email from Github that invites you to join the 6035 team. You need to accept this invitation.

```
# Acquire Decaf tests
git remote add tests https://github.com/6035/tests-fa18.git
git pull tests master --allow-unrelated-histories

# Note: If you add as a submodule instead, you'll need to update scripts
# to appreciate the extra folder created for the submodule.

# Fix any conflict in the readme or .gitignore (or don't bother), then
git commit -a

# build:
ant
```

If all of those steps succeeded, you can then run the skeleton code:

```
# run scanner:
./run.sh --target=scan tests/scanner/input/char1

# run parser:
./run.sh --target=parse tests/parser/legal/legal-01
```

You can specify `--debug` for debugging information.

After we have configured the personal repositories for submission, you can then upload your repository:

```
git remote add origin https://github.com/6035/<github username>6035.git
git push -u origin master
```

Scanner

Your scanner must be able to identify tokens of the Decaf language, the simple imperative language we will be compiling in 6.035. The language is described in the Decaf Language Handout. Your scanner should note illegal characters, missing quotation marks, and other lexical errors with reasonable error messages. The scanner should find as many lexical errors as possible, and should be able to continue scanning after errors are found. The scanner should also filter out comments and whitespace not in string and character literals.

You will not be writing the scanner from scratch. Instead, you will generate the scanner using ANTLR. This program reads in an input specification of regular expression-like syntax (grammar) and creates a Java program to scan the specified language. More information on ANTLR (including the manual and examples) can be on the web at:

```
http://antlr2.org/doc/index.html
http://www.antlr.org/wiki/display/CS652/CS652+Home
```

To get you started, we have provided a template in

```
src/edu/mit/compilers/grammar/scanner.g
```

If you chose to work from this skeleton, you must complete the existing ANTLR rules and add new ones for your scanner.

ANTLR generated scanners throw exceptions when they encounter an error. Each exception includes a text of a potential error message, and the provided skeleton driver prints them out. You are free to use use the messages provided by ANTLR, if you have verified that they make sense and are specific enough. Error messages must be printed to standard error.

ANTLR is invoked by the *scanner* and *parser* tasks of the Ant build file. The generated files, `DecafLexer.java`, etc, are placed in the `autogen` directory by the task. Note that ANTLR merely generates a Java source file for a scanner class; it does not compile or even syntactically check its output. Thus, typos or syntactic errors in the scanner grammar file will be propagated to the output.

An ANTLR generated scanner produces a string of tokens as its output. Each token has the following fields:

`type`	the integer type of the token
`text`	the text of the token
`line`	the line in which the token appears
`col`	the column in which the token appears

Every distinguishable terminal in your Decaf grammar will have an automatically generated unique integer associated with it so that the parser can differentiate them. These values are created from your scanner grammar and are stored in the `*TokenTypes.java` file created by ANTLR.

Parser

Your parser must be able to correctly parse programs that conform to the grammar of the Decaf language. Any program that does not conform to the language grammar must be flagged with at least one error message. As with the scanner, all error messages must be printed to standard error.

As mentioned, we will be using the ANTLR LL(k) parser generator, same tool as for the Scanner. You will need to transform the reference grammar in the Decaf Language Handout into a grammar expressed in the ANTLR grammar syntax. Be careful with checking your spelling, as ANTLR does match rule uses to declarations.

Your parser does not have to, and should not, recognize context-sensitive errors *e.g.,* using an integer identifier where an array identifier is expected. Such errors will be detected by the static semantic checker. *Do not* try to detect context-sensitive errors in your parser. However, you might need to create syntactic actions in your parser to check for some context-free errors.

You might want to look at Section 3 in the "Tiger" book, or Sections 4.3 and 4.8 in the Dragon book for tips on getting rid of shift/reduce and reduce/reduce conflicts from your grammar. You can tell ANTLR to print out the parse states (useful for resolving conflicts) by adding `trace="yes"` to the ANTLR target (please see the build file).

Scanner and Parser Outputs

Scanner output format

When `-t scan` is specified, the output of your compiler should be a scanned listing of the program with one row for each token in the input. Each line will contain the following information: the line number (starting at 1) on which the token appears, the type of the token (if applicable), and the token's text. Please print only the following strings as token types (as applicable): `CHARLITERAL`, `INTLITERAL`, `BOOLEANLITERAL`, `STRINGLITERAL` and `IDENTIFIER`.

For `STRINGLITERAL` and `CHARLITERAL`, the text of the token should be the text, as appears in the original program, including the quotes and any escaped characters.

Each error message should be printed on its own line, *before* the erroneous token, if any. Such messages should include the file name, line and column number on which the erroneous token appears.

Here is an example table corresponding to `print("Hello, World!");`:

```
1 IDENTIFIER print
1 (
1 STRINGLITERAL "Hello, World!"
1 )
1 ;
```

You are given both a set of test files on which to test your scanner and the expected output for these files (see next Section). **The output of your scanner should match the provided output exactly on all files without errors** (e.g. successful exit status of the `diff` command). This helps us to automate the grading process as much as possible.

Parser output format

When `-t parse` is specified, any syntactically incorrect program should be flagged with at least one error message, and the program should exit with a non-zero value (see `System.exit()`). Multiple error messages may be printed for programs with multiple syntax errors that are amenable to error recovery. Given a syntactically valid program, your parser should produce no output, and exit with the value zero (success). The exact format for parse error messages is not stipulated.

Provided Test Cases and Expected Output

The provided test cases for scanning and parsing can be found in the tests repository, under the `parser` and `scanner` directories. The expected output for each scanner test can be found in: `scanner/output`. We will also release hidden tests for the parser and scanner (see the next section). The directory structure for the hidden tests is the same as described above.

Project Evaluation

This project will not be officially graded. However, each student will still be required to write and evaluate his or her own scanner and parser. There are several reasons for this shape of the project:

- Writing (and debugging!) a front-end for a non-trivial language is a useful skill. Think about being able to quickly prototype a domain specific language in a few hours once you master the parsing techniques and tools.
- The following projects in this class will require a front-end for the Decaf language. Typically, this front-end will be constructed by combining the parsers written by the individual students.
- Your parser serves as an advertisement to the potential group mates. It gives a good indicator for how much your group can expect from you in the follow-up projects.

To catalyze the group formation process, we will make all students' repositories visible to all other students in the class a few days before the due date (see the schedule in the document header). This will also be an opportunity for the students to assess the similarities and differences in their approaches and how to combine the scanners/parsers together.

However, copying code between the projects is strictly forbidden. While the students are allowed to inspect and discuss other students' solutions, copying code will be considered cheating. We will be strict in enforcing this policy!

Evaluation

Once the repositories become visible to all students, we will release the set of additional, hidden tests that will evaluate different aspects of the scanner/parser. We will also provide the scripts that automate the part of the testing process.

The students will be able to continue improving their code and fixing errors until the due date. We encourage students to test other students' code and provide feedback on the passed and failed tests.

Deliverable

As the outcome of this project, the students will form groups for the remaining projects. Each group will write a short report and send it to the TA.

The report should list the members of the new groups.

The report should provide the following information for each group member:

- **Language:** In which language was the scanner/parser implemented.
- **Public Test Score:** Report on the percentage of passed/failed public tests on the project's due day. In addition, provide a brief description (several sentences) on what was the most challenging part of the implementation.
- **Hidden Test Score:** Report on the percentage of passed/failed hidden tests on the project's due day. In addition, provide a brief description (several sentences) on what was the most challenging part of the implementation.
- **Comparison With Previous Results:** Report how much the percentage of passed tests (both public and hidden) has improved since the testing before obtaining the hidden tests.

In addition, the group should write a one paragraph plan on how they intend to integrate their scanners/parsers in the next project (and give a rationale for their decision).

Appendix: Why we defer integer range checking until the next project

When considering the problem of checking the legality of the input program, there is no fundamental separation between the responsibilities of the scanner, the parser and the semantic checker. Often, the compiler designer has the choice of checking a certain constraint in a particular phase, or even of dividing the checking across multiple phases.q However, for pragmatic reasons, we have had to divide the complete scan/parse/check unit into two parts simply to fit the course schedule better.

As a result, we will have to mandate certain details about your implementations that are not necessarily issues of correctness. For example, one cannot completely check whether integer literals are within range without constructing a parse tree.

Consider the input:

```
x+-9223372036854775808
```

This corresponds to a parse tree of:

```
  +
 / \
x   -
    |
INT(9223372036854775808)
```

We cannot confirm in the scanner that the integer literal -9223372036854775808 is within range, since it is not a single token. Nor can we do this within the parser, since at this stage we are not constructing an abstract syntax tree. Only in the semantic checking phase, when we have an AST, are we able to perform this check, since it requires the unary minus operator to modify its argument if it is an integer literal, as follows:

```
  +                        +
 / \                      / \
x   -          ---->     x   INT(-9223372036854775808)
    |
INT(9223372036854775808)
```

Of course, if the integer token was clearly out of range (e.g. 999999999999999999) the scanner could have rejected it, but this check is not required since the semantic phase will need to perform it later anyway.

Therefore, rather than do some checking earlier and some later, we have decided that ALL integer range checking must be deferred until the semantic phase. So, your scanner/parser must not try to interpret the strings of decimal or hex digits in an integer token; the token must simply retain the string until the semantic phase.

When printing out the token table from your scanner, do not print the value of an `INTLITERAL` token in decimal. Print it exactly as it appears in the source program, whether decimal or hex.