

Massachusetts Institute of Technology  
Department of Electrical Engineering and Computer Science

6.035, Fall 2007

Handout 6 — Scanner-Parser Project

Thursday, September 6

---

**DUE: Monday, September 17**

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

## Preliminaries

In this section we will describe the infrastructure that is provided for the project. You are encouraged to use it, but you may also choose to ignore it and design your infrastructure from scratch. The skeleton is located in `/mit/6.035/provided/skeleton`.

## Provided Infrastructure

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:

```
.
|-- bin
|-- build.xml
|-- lib
|  |
|  |-- antlr.jar
'-- src
    |
    |-- decaf
    |  |
    |  |-- Lexer.g
    |  |-- Parser.g
    |  |-- Main.java
    |-- java6035
    |
    |-- tools
    |  |
    |  |-- CLI
    |  |  |
    |  |  |-- CLI.java
```

The `lib` directory includes libraries that are needed to *run* your compiler; right now it contains the ANTLR tool. The `bin` directory contains the tools needed to *compile* your compiler. However, since ANTLR uses the same binary as a tool and as a library, and we already included it in the `lib` directory, `bin` director is empty. `build.xml` is the ant build file for the infrastructure. The `src` directory contains all your source code for the compiler. `src/decaf/Parser.g` is a skeleton parser grammar and `src/decaf/Lexer.g` is a skeleton scanner grammar. `src/decaf/Main.java` is the skeleton driver and `src/java6035/tools/CLI/CLI.java` is the command-line interface library.

To access and build the provided infrastructure, you must add the following lockers to your file system.

```
add 6.035
add java_v1.5.0_06
add gnu
add sipb
add eclipse-sdk
```

The eclipse-sdk locker is optional and it adds the eclipse IDE. No changes to your `$CLASSPATH` variable are required, but due to an interaction between `sipb` and `java_v1.5.0_06`, we recommend that you set an environment variable `JAVA_HOME` to `/afs/athena.mit.edu/software/java_v1.5.0_06/`; all the classes and jar files required for compilation are included by the Ant build file (see next Section).

## Ant 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, and cleaning the infrastructure. The default rule is for complete compilation and jar creation. The build file creates three directories during compilation: `classes`, `java`, and `dist`. The `classes` directory contains all `.class` files created during compilation. The `java` 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. A clean will delete these three directories. If you are using CVS, you should not add these directories to your repository.

## Getting Started

To start the project we recommend that each group:

1. Create a CVS repository in your group locker.
2. Copy the skeleton infrastructure to your group locker.
3. Add all the skeleton files to your CVS repo under a new module using `cvs import`.
4. Have each group member checkout the new module, either locally on Athena (`cvs checkout`), remotely (using remote CVS), or in Eclipse.
5. Begin working.

## 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 Handout 5. 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.cs.usfca.edu/~parrt/course/652/lectures/antlr.html>

To get you started, we have provided a template in

```
/mit/6.035/provided/skeleton/src/decaf/Lexer.g
```

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

*Note: The newest version of ANTLR is 3.0.1, but for this class we will be using version 2.7.7. These are very different versions of the tool. When looking up information about ANTLR, please be careful to look at correct versions.*

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 the messages provided by ANTLR, if you have verified that they make sense and are specific enough.

ANTLR is invoked by the *antlr* task of the Ant build file. The generated files, `DecafLexer.java`, etc, are placed in the `java/decaf` 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:

<code>type</code>	the integer type of the token
<code>text</code>	the text of the token
<code>line</code>	the line in which the token appears
<code>col</code>	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 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 Handout 5 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).

## What to Hand In

Follow the directions given in Handout 3 when writing up the hard copy for your project. Include a full description of how you resolved ambiguities in the Decaf grammar provided.

For the electronic portion of the hand-in, provide a gzipped tar file named `leNN-parser.tar.gz` in your group locker, where NN is your group number. This file should contain all relevant source code. Additionally, you should provide a Java archive, produced with the `jar` tool, named `leNN-parser.jar` in the same directory. Unpacking the tar file and running `ant` should produce the same Java archive. (All future hand-ins will be roughly identical, except with different names for the files.)

A sample driver file has been provided:

```
/mit/6.035/provided/skeleton/src/decaf/Main.java
```

This includes calls for the scanner and the parser. Assuming `$COMPILER_HOME` points to the root directory of your infrastructure, the compiler can be called from the command-line by issuing:

```
java -jar $COMPILER_HOME/dist/Compiler.jar <options> <infile>
```

The driver will parse the command line by using CLI (see handout 4) and run your parser or standalone scanner depending on what is requested. For this project you will probably not have to edit the driver. Your compiler must handle both `scan` and `parse` as arguments to the `-target` flag (assume `parse` by default). The source for CLI is located at:

```
/mit/6.035/provided/skeleton/src/java6035/tools/CLI/CLI.java
```

## Scanner format

When `-target 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 TA requests that the output of your scanner matches the provided output exactly on all files without errors** (successful exit status of the `diff` command). The TA would like to automate the grading process as much as possible.

## Parser format

When `-target 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:

```
/mit/6.035/provided/scanner
/mit/6.035/provided/parser
```

The expected output for each scanner test can be found in:

```
/mit/6.035/provided/scanner/output
```

We will test your scanner/parser on these and a set of hidden tests. You will receive points depending on how many of these tests are passed successfully. In order to receive full credit, we also expect you to complete the written portion of the project as described in Handout 3.

## A 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. 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+-2147483648
```

This corresponds to a parse tree of:

```
  +
 / \
x   -
   |
  INT(2147483648)
```

We cannot confirm in the scanner that the integer literal -2147483648 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(-2147483648)
   |
  INT(2147483648)      ----->
```

Of course, if the integer token was clearly out of range (e.g. 9999999999) 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.