# CSE 304: Programming Assignment #2

## Parser (Syntax Analysis) Module

**Due:** Fri., Oct 2, 2009

Read the following description carefully before starting on the assignment. Also, go through the brief overview of the project path that describes the steps involved. A list of what materials are available and what you are expected to submit are also given.

The deliverables for this homework are a little different from the usual. Please make sure you go through the Section 5 of this handout and understand what is required, what is optional and encouraged, and why.

## 1 Overview

In this assignment, you will build the parser module for the `Decaf` compiler. You will use `Bison`, a scanner-generator tool, for this purpose. The syntax of `Decaf` is defined in the accompanying `Decaf` manual (Sections 2 through 4) are relevant for this assignment).

`Decaf`'s syntax is given in Extended Backus-Naur Form (EBNF). Specifically, grammar rules in EBNF are of the form

$$L \ ::= \ r_1 \mid r_2 \mid \ \ldots r_n$$

where $L$ is a non-terminal symbol (left hand side of a production) and each $r_i$ is a (possibly empty) right hand side of a production. In the standard notation of grammars described in class, the right hand side of a production is a sequence of terminal and non-terminal symbols. In EBNF, the right hand side may be written using a regular expression-like notation as well. For instance, the following EBNF rule represents a string of one or more `a`'s separated by `b`'s (e.g. `ababa`):

$$S \ ::= \ \texttt{a} \ (\texttt{b} \ \texttt{a})*$$

The single rule above can be represented by the set of productions in standard notation below:

$$\begin{array}{lcl} S & \longrightarrow & \texttt{a}\, T \\ T & \longrightarrow & \epsilon \mid \texttt{b}\ \texttt{a}\, T \end{array}$$

Your task in this assignment is to come up with a Bison specification for `Decaf` syntax specified in EBNF. First of all, note that Bison specifications follow the standard grammar notation. Hence you need to expand the EBNF notation to the standard notation. Secondly, Bison generates an LALR(1) parser, which means that it can handle only a subset (but a large subset) of context free grammar specifications. Hence you will need to make sure that the expanded grammar can be handled by Bison.

In this assignment, you need to prepare Bison specification so as to determine whether or not a `Decaf` program is syntactically valid. The parser must be constructed such that it outputs the reduction sequence used to parse the input file. Specifically, whenever a production $A \rightarrow \alpha$ is used in a reduction, you will print the name of the non-terminal on the left-hand side of the production used in reduction, i.e., in this example, $A$. *Only those productions that correspond to the non-terminals mentioned in the grammar description should be printed. If you have introduced intermediate or auxiliary non-terminals to expand the grammar, these SHOULD NOT be printed.*

## 2 Integrating with Lexical Analyzer

The parser you build will be integrated with a lexical analyzer that I have supplied (see lex.yy.c in the accompanying source archive). You can compile and link this with your parser using the supplied Makefile.

The tokens emitted by the lexical analyzer are:

- Tokens for reserved words:

```
TOK_BOOLEAN TOK_BREAK TOK_CLASS TOK_CONTINUE TOK_DO TOK_ELSE
TOK_EXTENDS TOK_FALSE TOK_FLOAT TOK_FOR TOK_IF TOK_INT
TOK_NEW  TOK_NULL TOK_PRIVATE TOK_PUBLIC TOK_RETURN TOK_STATIC
TOK_SUPER TOK_THIS TOK_TRUE TOK_VOID TOK_WHILE
```

- Tokens for constants:

| | |
|---|---|
| `TOK_INT` | `Integer constant` |
| `TOK_FLOAT` | `Floating point constant` |
| `TOK_STRING` | `String constant` |

- Token for identifier:

```
TOK_ID
```

- Punctuation symbols:

| | |
|---|---|
| `TOK_COMMA` | `,` |
| `TOK_DOT` | `.` |
| `TOK_SEMICOLON` | `;` |
| `TOK_OPEN_SQ_BRACKET` | `[` |
| `TOK_CLOSE_SQ_BRACKET` | `]` |
| `TOK_OPEN_PAREN` | `(` |
| `TOK_CLOSE_PAREN` | `)` |
| `TOK_OPEN_BRACE` | `{` |
| `TOK_CLOSE_BRACE` | `}` |
| `TOK_PLUS` | `+` |
| `TOK_MINUS` | `-` |
| `TOK_MULTIPLY` | `*` |
| `TOK_DIVIDE` | `/` |
| `TOK_PLUS_PLUS` | `++` |
| `TOK_MINUS_MINUS` | `--` |
| `TOK_EQUAL` | `=` |
| `TOK_AND` | `&&` |
| `TOK_OR` | `\|\|` |
| `TOK_NOT` | `!` |
| `TOK_GREATER` | `>` |
| `TOK_LESSER` | `<` |
| `TOK_EQUAL_EQUAL` | `==` |
| `TOK_NOT_EQUAL` | `!=` |
| `TOK_GREATER_OR_EQUAL` | `>=` |
| `TOK_LESSER_OR_EQUAL` | `<=` |

## 3 Project Path

You will be writing the specifications for the parser in a file. Conventionally, grammar specifications are stored in files with a `.yy` suffix. ("yy" tells Bison to generate C++ code).

Let the specifications be in a file `decaf.yy`. Use the parser-generator `bison` to create the analyzer from the specifications. The generated analyzer should be a `C++` program, stored in a file called `decaf.tab.cc`.

Compile `decaf.tab.cc` to get `decaf.tab.o`, link it with `driver.o` and `lex.yy.o` to produce an executable, called `demo`.

Use the `-d` option of `bison` to automatically generate the necessary parser-lexer interface file (`decaf.tab.hh`).

Most of the above steps are automated in the Makefile provided to you.

You should get a complaint about one shift-reduce conflict from Bison, which arises because of the if-then Vs if-then-else conflict. If you get any more, then there must be something wrong with the way you have translated the grammar provided to you. Go back and check it — most probably, you missed some precedence or associativity information. If this does not work, use `-v` option of `bison` to get a "verbose" file `decaf.tab.cc.output`. This file contains the productions, states and transitions used by the parser. This file will also tell you where the conflicts are, i.e., which states and which productions. (Tip: there are several conflict messages at the top of the file. You should ignore them, as they do not provide much useful information. The conflicting transitions are enclosed in brackets, so you can get to them by searching for for open brackets (i.e., '[')). From this information, you should be able to make out what the offending grammar rules are and how they should be modified. If, after several attempts, you are unable to resolve the problem, contact the Instructor or the TA.

**IMPORTANT:** *To avoid running into difficulties with conflicts in the grammar*, you should add the grammar rules gradually — as few at a time as possible. Then run `bison` to ensure that you do not run into any conflicts. If you do run into conflicts this way, then the problem is most likely to be in the rules that you just added.

Start with the top-level structure of the language, going through class definitions, field definitions and method definitions.

## 4 Available Material

The following material is available in the accompanying `.tgz` archive:

- The driver source file `driver.cc`
- My lexical analyzer file `lex.yy.c`.
- A sample Makefile that automates the building of your executable `demo`. Modify it as you deem fit, but I do not anticipate that you will need make any modifications to this file.
- Sample input files `in*` and the respective output files `out*` in the `tests` subdirectory.
- A sample Bison specification file `decaf.yy`, with all of the token declarations and only one grammar rule. As given, it can be used to build `demo`. This can process only the empty program correctly. It lacks the grammar rules need to process any of the test input files. You will need to modify this sample `decaf.yy` file to gradually add the remaining grammar rules, or modify the existing rules in this file so as to conform to the `Decaf` grammar.
- A reference parser (`ref_decaf.tab.cc`), which if used instead of `decaf.tab.cc` that you would generate, would produce a reference program to test your parser (and compare outputs).

## 5 Deliverables

The required deliverable will be a single file: `decaf.yy`. I should be able to create `demo` as outlined above with *your* `decaf.yy` file. You will upload this file on Blackboard using the assignment form (**HW2, Part b**).

In order to make sure you don't get stuck in this homework in the last minute, I encourage you to submit assignment form **HW2, Part a** as well. This is an optional submission. In **Part a**, you will describe how you plan to finish HW2: the more detailed the plan, the better. This optional submission is due on Sunday, Sep 27, by midnight. The idea behind this extra submission is to make sure you flesh out some details of HW2 before the due date of **Part b**. If I see any serious issues in your submissions to **Part a**, I will go over these in class on Tuesday, Sep 29.