**COMP 412**
**Fall 2011**
**Programming Exercise Two:** *An LL(1) Parser Generator*

In this exercise, you will construct a program that generates the parsing table for a table-driven LL(1) parser. Your program will take, as input, a grammar in a notation adapted from Backus-Naur Form. *You can assume that the input grammar has no left recursion.*

A grammar is said to have the LL(1) property if and only if,

$\qquad$ *for all pairs of productions $A{\rightarrow}\beta$ and $A{\rightarrow}\delta$,* $\text{FIRST}^{+}(A{\rightarrow}\beta) \cap \text{FIRST}^{+}(A{\rightarrow}\delta) = \varnothing$,

where $\text{FIRST}^{+}(A{\rightarrow}\gamma)$ is the "augmented first set" defined on page 107 of *Engineering a Compiler, 2nd Edition* (hereafter, "the book").

For a grammar that has the LL(1) property, your program should produce, as output, a listing of the FIRST and FOLLOW sets for the grammar, and a version of the LL(1) parse table suitable for a human to read, with axes labeled appropriately. If the input grammar does not have the LL(1) property, your program should produce an error message that states which productions in the grammar produce the conflict.

Conceptually and structurally, the lab has three parts: a scanner, a parser, and a table generator. You will build a hand-coded scanner and a hand-coded recursive descent parser for Modified Backus-Naur Form (MBNF). Your parser will build a representation of the input grammar suitable for the processing done in the table generator. You must select suitable data structures to represent the productions and sets. The table generator will compute FIRST sets, FOLLOW sets, and the LL(1) parse table, as described in the online lecture notes and in Section 3.3.3 of the book.

## Modified Backus-Naur Form
Table 1 shows a grammar for the Modified Backus-Naur form that your parser generator must accept. Note that the grammar, as stated, relies on left recursion; before building your parser, you must transform the grammar for MBNF to make it suitable for a top-down, recursive descent parser. In Table 1, all terminal symbols are written in ALL CAPITAL LETTERS.
In MBNF, white space is allowed between words in the input and, in some cases, is required between them (e.g., between two consecutive SYMBOLs in a *SymbolList*). White space can consist of blanks, tab characters, and the end-of-line sequence.

## Scannner
You will build a hand-coded scanner for the words in our pseudo Backus-Naur form. Table 2 describes their microsyntax and assigns a "syntactic category" or "token type" to each word.

Your scanner will need to build a lookup table of the symbols (to enable efficient translation between names, productions, and sets such as FIRST, FOLLOW, and FIRST$^{+}$). See, for example, Section 5.7 and Appendix B in the book for advice on implementing hash tables and sets.

| 1 | *Grammar* | → | *ProductionList* |
|---|---|---|---|
| 2 | *ProductionList* | → | *ProductionSet* **SEMICOLON** |
| 3 | | \| | *ProductionList ProductionSet* **SEMICOLON** |
| 4 | *ProductionSet* | → | **SYMBOL DERIVES** *RightHandSide* |
| 5 | | \| | *ProductionSet* **ALSODERIVES** *RightHandSide* |
| 6 | *RightHandSide* | → | *SymbolList* |
| 7 | | \| | **EPSILON** |
| 8 | *SymbolList* | → | *SymbolList* **SYMBOL** |
| 9 | | \| | **SYMBOL** |

**Table 1: Definition of Modified Backus-Naur Form**

Your scanner should be robust and efficient. It should return the appropriate lexeme and token type. It should create a symbol table and enter the lexeme for each SYMBOL in the table.

## Parser

Once you have a working scanner, you will transform the MBNF grammar to make it suitable for recursive descent and build a hand-coded, recursive descent parser for it.

Your parser should build up a model, or an internal representation (IR), of the productions in the input grammar. The IR should be suitable for use in the computation of FIRST, FOLLOW, and FIRST$^+$ sets. Look over those algorithms to determine what kind of operations your program will perform on the productions and design an appropriate IR. i.

Your parser should be robust and efficient. It should produce useful error messages for ill-formed inputs. It should build the data structures needed by the table generator.

The first completion deadline is for a working scanner and parser. You must turn in code that the laboratory assistant can run, along with three example MBNF grammars on which you tested the code. Your parser should print out enough diagnostic information that the laboratory assistants can easily see that the code is working.

| Terminal Symbol | Regular Expression | Token Type |
|---|:---:|:---:|
| SEMICOLON | ; | 0 |
| DERIVES | : | 1 |
| ALSODERIVES | \| | 2 |
| EPSILON | EPSILON \| epsilon \| Epsilon | 3 |
| SYMBOL | [a-z \| A-Z \| 0-9]$^+$ | 4 |
| EOF | (*not applicable*) | 5 |

**Table 2: Microsyntax of Words in the MBNF Grammar**

## Table Generator

After you have a working scanner and parser, you will implement an LL(1) table generator. It should compute FIRST sets, FOLLOW sets, and FIRST$^+$ sets. It should use these sets to generate an LL(1) parse table. The algorithms for these constructions are found in Chapter 3 of the book and in the lecture notes on parsing.

### *Command-line Flags*

To simplify debugging and grading, your program will need command line options to print out the table of productions, the FIRST, FOLLOW, and FIRST$^+$ sets, and the LL(1) table. You must implement at least the following command-line flags.

> -d      print declarations, as specified below, to stdout.
>
> -p      print the productions as recognized by the parser in a human readable form
>
> -f      print the FIRST sets for each grammar symbol, in a human readable form
>
> -g      print the FOLLOW sets for each nonterminal, in the same form
>
> -h      print the FIRST$^+$ sets for each production, in the same form
>
> -t      print the LL(1) table, with each column and row labeled. Use production numbers from the production table (-p above) for the non-error entries.

Of course, nonsensical command line flags should produce a listing of the valid command line flags, as should the flag -?.

### *Declarations*

Your parser generator should also produce, with the –d flag, declarations for use in a skeleton parser. We will provide the skeleton parser, written in C. A separate document will describe the declarations that you must produce and the process for compiling them into the skeleton parser and testing it. (Emitting the C declarations should be the last step in debugging your project.)

### *Numbering*

To allow for a clean interface between your parser generator and the skeleton parser, you must use a consistent scheme for numbering grammar symbols and productions.

1. Productions should be numbered from 0 to $n$, in the order of presentation in the input.

2. Terminal symbols should be numbered, beginning with zero, in the order in which they are encountered in the input file. For a grammar with $k$ terminals, the numbers would run from 0 to $k$-1.

3. The special symbol EOF should have a number one greater than the highest-numbered terminal symbol. With $k$ terminals, EOF receives the number $k$.

4. Nonterminals are numbered sequentially, beginning immediately above EOF. Nonterminals are assigned numbers in the order in which they are encountered in the input file.

Table 3 shows how these rules would number the symbols in our grammar for Modified Backus-Naur Form. Productions would be numbered as in Table 1.

| Terminal Symbols | Number | Nonterminal Symbols | Number |
|---|---|---|---|
| SEMICOLON | 0 | *Grammar* | 6 |
| DERIVES | 1 | *ProductionList* | 7 |
| ALSODERIVES | 2 | *ProductionSet* | 8 |
| EPSILON | 3 | *RightHandSide* | 9 |
| SYMBOL | 4 | *SymbolList* | 10 |
| EOF | 5 | | |

**Table 3: Numbering Symbols in the MBNF Grammar**

## Schedule

This programming exercise has three due dates: one for the scanner and parser, another for the table generator, and a third for your lab report.

⇒ The Scanner and Parser are due on **Monday, October 24, 2009** at 11:59PM.

⇒ The Table Generator is due on **Friday, November 4, 2009** at 11:59PM. (This date should provide plenty of time, given that you finish the scanner and parser on time.

⇒ The lab report is due on **Sunday, November 6, 2009** at 11:59PM.

The labbies will post a notice on the class web site describing how to submit the various parts of your lab.

*Late Policy:* The due dates on this lab are immovable, because of the need to start lab 3 before the end of the semester. Work turned in by 11:59PM on the due date will receive full credit. *Late work will be graded, but will lose 5% credit per day past the deadline,* computed as a simple additive penalty. That is, one day late is 95% credit; two days late is 90% credit, and so on.

*Lab Report:* your lab report should discuss your experiences in writing the scanner, parser, and table generator. It should document your symbol table structure, your transformed grammar, and your set representation(s). You should include a section on how you tested each of the three components and why you believe that they function correctly. Include an example of your program's behavior on an invalid input. Be sure to discuss both the asymptotic and expected complexity of the various pieces of your program.

**EXTRA CREDIT:** For 10% extra credit, implement an option in your table generator that removes left recursion using the algorithm presented in the book. Extra credit on this lab will be added to your final grade. (For example, 10% extra credit on this lab would produce 10% of the points allotted to the lab, added to your final grade.