Arizona State University
Computer Science and Engineering
CSE 340 Fall 2012
Project 2

Due by 11:59 pm October 9, 2012

Abstract

The ultimate goal of this project is to develop a program that constructs a predictive recursive-descent parser automatically from a grammar description. The input to your program will be a description of a context-free grammar G. To keep it simple, the output will be the FIRST and FOLLOW sets of the non-terminals of the grammar.

1 Grammar Description

The grammar description is provided in a text file. The grammar specification has multiple sections separated by the # symbol. The grammar specification is terminated with ##. If there are any symbols after the ##, they are ignored. All grammar symbols as well as the # symbol are space-separated.

Grammar Sections The first section contains all the non-terminals of the grammar. The first non-terminal in the first section is the start symbol of the grammar. The second section lists all the terminals of the grammar and subsequent sections contain grammar rules. We explain the format of each section and then give an example.

Terminal and Non-terminal Symbols The name of a symbol (terminal or non-terminal) is given by the regular expression:

$$\texttt{letter}\ (\texttt{letter} + \texttt{digit})*$$

Grammar Rules A grammar rule starts with a non-terminal symbol (left-hand side of the rule) followed by `->`, then followed by a sequence of zero or more terminals and non terminals which represent the right-hand side of the rule. Each grammar symbol is either the name of a non-terminal or the name of a token (terminal). The tokens are listed in the second section of the input. All grammar symbols are case sensitive.

If the sequence of terminals and non-terminals following the left-hand side of a rule is empty, this represents a rule of the form $A \to \epsilon$. Here is an example:

```
decl idList1 idList #
COLON ID COMMA #
decl -> idList COLON ID #
idList -> ID idList1 #
idList1 -> #
idlist1 -> COMMA ID idList1 ##
```

The first 2 lines specify the set of non-terminals and terminals in order:

$$\begin{aligned} \text{NonTerminals} &= \{\ \text{decl},\ \text{idList1},\ \text{idList}\ \} \\ \text{Terminals} &= \{\ \text{COLON},\ \text{ID},\ \text{COMMA}\ \} \end{aligned}$$

And here is the grammar:

$$\begin{aligned} \text{decl} &\to \text{idList}\ \ \text{COLON}\ \ \text{ID} \\ \text{idList} &\to \text{ID}\ \ \text{idList1} \\ \text{idList1} &\to \epsilon \\ \text{idlist1} &\to \text{COMMA}\ \ \text{ID}\ \ \text{idList1} \end{aligned}$$

2 Requirements

Your program should output the FIRST and FOLLOW sets of each *non-terminal symbol.* The FIRST sets are listed first then the FOLLOW sets are listed. The FIRST sets and the FOLLOW sets are listed for non-terminals in the order in which non-terminals appear in the first section of the grammar specification. For the grammar above, the output looks as follows:

```
FIRST(decl) = { ID }
FIRST(idList1) = { #, COMMA }
FIRST(idList) = { ID }
FOLLOW(decl) = { $ }
FOLLOW(idList1) = { COLON }
FOLLOW(idList) = { COLON }
```

Notice that in the example, the FIRST set of `idList1` appears before the FIRST set of `idList` because `idList1` appears before `idList` in the first section of the grammar specification. In the output, # is used to represent ϵ and \$ is used to represent `eof`. Note that # and \$ cannot both appear in the same set. If # or \$ are in a set they must be listed first followed by other symbols in the set. All symbols other than # or \$ should be sorted according to dictionary order. C provides functions for comparing strings.

It is very important that the output is produced according to the specification above. If your program calculates the FIRST and FOLLOW sets correctly but produce them in the wrong order, no partial credit will be assigned.
If there is an error in the input, the output should simply be of the form `ERROR CODE` followed by a number. Here is the list of errors and their codes:

- ⋆ Input not according to format (syntax error): `ERROR CODE 0`
- ⋆ Non-terminal symbol listed in the first section but does not have any rules: `ERROR CODE 1`
- ⋆ Symbol appears in a rule but is not listed in the first nor the second section: `ERROR CODE 2`
- ⋆ Terminal symbol appearing on the left-hand side of a rule: `ERROR CODE 3`

If there is an error in the input, the FIRST and FOLLOW sets should not be calculated. The input might have multiple errors. If input is not according to format, you output should only include one line: ERROR CODE 0. If the input is according to format and there is more than one type of error (error codes 1 to 3), you output should have one line per error code. The lines should be ordered according to the numeric value of the code. For example, if the output has both ERROR CODE 1 and ERROR CODE 3, then ERROR CODE 1 should be listed before ERROR CODE 3.

3 Implementation Suggestions

It is very important that you plan you implementation before you start coding. Read the specifications a couple of time to make sure you understand what is required, then come up with a design for your solution. At a high-level, your program will do the following:

1. Read set of non-terminals and store them in an array.
2. Read set of terminals and store them in an array.
3. Read grammar rules and store them in appropriate data structures.

4. Check for errors while doing the previous steps and in case an error is found, add the error to a list of errors. r

5. If at the end of reading the input, errors have been found, print all errors in the list of errors (in the correct order) and terminate execution

6. Calculate FIRST sets for non-terminals and output FIRST sets in appropriate order.

7. Calculate FOLLOW sets for non-terminals and output FOLLOW sets in appropriate order.

Implementing the rules for calculation of FIRST and FOLLOW sets requires doing a lot of set operations like intersection and union. So you need a data structure to store sets and functions for set operations. A simple and efficient way of implementing a set data structure is to use fixed-length binary arrays. For example let's say we want to represent a FIRST set of some symbol. A FIRST set can contain any one of the terminals of the grammar and ϵ. So we form a reference set that includes all terminals and ϵ:

$$\text{U} = \{\ \epsilon,\ \text{COLON},\ \text{ID},\ \text{COMMA}\ \}$$

Now if we want to represent `FIRST(idList)`, we can use the following binary array of length 4:

$$a = \{0, 0, 1, 0\}$$

This binary array specifies which elements of the reference set are present in a subset of that reference set. In this example, the only element present in `FIRST(idList)` is `ID`. This data structure is very easy to use and powerful, for example if you want to implement set union operation, you can simply calculate the logical OR of the elements of the operands and if you want the intersection of two sets, you can simply use AND. Here is an example:

$$\begin{aligned} a &= \{0, 1, 0, 1\} \\ b &= \{0, 0, 1, 1\} \\ a \cup b &= \{0, 1, 1, 1\} \end{aligned}$$

But be careful when using this technique, the operands should be subsets of the same reference set, otherwise the result is not meaningful. You will need a reference set for this project that contains all terminals plus ϵ and `eof` symbol. Note that the order of elements in the reference set is important and should not change during the execution.

4 Grading

1. Correctly calculating FIRST sets:

    (a) FIRST sets for grammars with no ϵ: 35 points
    (b) FIRST sets for grammars with ϵ: 15 points

2. Correctly calculating FOLLOW sets:
    (a) FOLLOW sets for grammars with no ϵ: 15 points
    (b) FOLLOW sets for grammars with ϵ: 15 points

3. Detecting errors in the input: 20 points (5 per `ERROR CODE`)

The distribution of points is not necessarily proportional to the difficulty.

5 Submission

1. You should submit all your code on blackboard by 11:59:59 pm on due date.
2. You should submit one .zip file that includes all your source files and provide instructions on how to compile and execute your code.
3. Make sure your submission is correctly uploaded.
4. Make sure you follow the naming conventions for your files.