Jim Lambers

CS143 Compilers

Summer Quarter 2000-01

Programming Project 2

This assignment must be submitted electronically by Monday, July 23.

1 Overview

In this project, you will create an LR parser for the Mocha language. This parser will use the lexical analyzer from Project 1 to obtain a sequence of tokens from a given input string and ensure that the tokens form a syntactically valid Mocha program.

2 Functional Specification

To create this parser, you will use the parser generating tool `bison`, for which a handout is provided to help you with using this tool. You will also be provided an updated version of the Mocha specification, which now includes the syntax of Mocha, written in Extended Backus-Naur Form (EBNF). It will be necessary to convert this EBNF to a context-free grammar, which will then serve as the input to `bison`.

The parser is to behave as follows: whenever it performs a reduction using a production $A \to \alpha$, you must print a line to standard output including the name of the non-terminal A, an arrow `->`, and the name of each grammar symbol in α from left to right, separated by a single space. The names of the symbols are given in the EBNF.

The context-free grammar you create from the EBNF will generate conflicts in the LR parsing table. To resolve these conflicts, you must specify the proper rules for precedence and associativity for the operators of Mocha, using the directives provided by `bison` for this purpose.

If a syntax error occurs, the function `yyerror` will be called by the parser generated by `bison`. This function has already been implemented for you. After calling this function, the parser will exit, rather than attempt to recover from the error.

3 Implementation

Your first task is to convert the given syntactic rules for Mocha from EBNF to a context-free grammar. Most of the conversion process will be straightforward, but there are a few rules that MUST be followed EXACTLY:

- In the EBNF, where you see the notation $[A]$, where A is a non-terminal, you must create a new non-terminal A' with the same name as A, with the suffix *Opt*. A' has two productions, $A' \rightarrow A$, and $A' \rightarrow \epsilon$. The interpretation of A' is "an optional A". Wherever $[A]$ appears in a rule, you must replace it with A'.

- In the EBNF, where you see the notation $[a]$, where a is a terminal, you must create a new non-terminal A' with the same name of the token corresponding to a, removing the $T\_$ prefix, capitalizing the name, and adding the suffix *Opt* For example, `T_static` becomes *StaticOpt*. A' has two productions, $A' \rightarrow a$, and $A' \rightarrow \epsilon$. The interpretation of A' is "an optional a". Wherever $[a]$ appears in a rule, you must replace it with A'.

- Where you see the notation $\{A\}$, where A is a non-terminal, you must create a non-terminal A' with the same name as A, with the suffix s to pluralize this name. This non-terminal has two productions, $A' \rightarrow A'A$, and $A' \rightarrow \epsilon$. A' is used to denote "a sequence of A's". Wherever $\{A\}$ appears in a rule, you must replace it with A'.

- Where you see a rule in the EBNF $A \rightarrow \beta\{b\beta\}$, where β is a terminal or non-terminal and b is a terminal, you must replace this rule with two productions: $A \rightarrow \beta$, and $A \rightarrow Ab\beta$.

- Where you see a rule in the EBNF $A \rightarrow \alpha[\beta]$, where both α and β are strings of grammar symbols and β is not a single non-terminal (otherwise the first rule above would apply), you must add a new non-terminal A' with the same name as A, with the suffix *Rest*. You must also add the productions $A' \rightarrow \beta$ and $A' \rightarrow \epsilon$. The non-terminal A' is used to denote "the rest of A".

Just as in Project 1, you will be provided a set of starter files for this project. You will only be modifying one file, `parser.y`. This is the input file for `bison`, in which you will input the Mocha grammar. In this file, you will also specify rules for precedence and associativity of operators or other terminal symbols as necessary to eliminate any conflicts in the parsing table.

It is suggested that you consult the handout on `bison` for more information on how to do this.

4 Testing

Once you are ready to test your parser, simply type the command `make` at the prompt (assuming your project files are in your current working directory) to compile your code. If there are no compiler or linker errors, the executable, named `pp2`, will be written to the current directory.

We have provided several sample inputs to help you test your scanner. These may be found in

```
/usr/class/cs143/assignments/pp2/samples
```

With each sample input, there is a corresponding output file (with a `.out` extension). This is the actual output generated by our solution, so you should ensure that your output matches ours. We strongly recommend that you create other input files for testing, as we will test your code on a larger collection of inputs.

To run the scanner with any of the samples, or any input files that you may create, simply use this command from the directory containing the executable `pp1`:

```
./pp2 < filename >& outputfile
```

where *filename* is the name of the input file you are using and *outputfile* is the name of an output file of your choosing. When testing against the sample input files, you should then use the `diff` command to compare your output against the sample output file. Please make sure that you format your output in the same was as the solution, in order to facilitate grading.

5 Submission

This assignment must be submitted electronically by the given due date. You can find detailed information about the electronic submission process and our policy on late assignments on the course web site.

6 Grading

This assignment is worth 100 points, and counts for 10% of your overall grade in this course. We will grade this assignment by running your program against all of our input files, and comparing the output against the output generated by our solution using `diff`, and points will be deducted based on the differences that are detected.

NOTE: programs that do not compile, or cause segmentation faults when run, will only receive limited partial credit. We will not try to fix your code in these situations. Please do your best to make sure that your code works before you submit it.

7 Portability

While you have a choice as to where you may work on this assignment, you do not have a choice as to where it will be graded. Your program will be run on the AIR Solaris workstations, and must be submitted from there. Before you submit, please test on these workstations to ensure that your code will work just as well for us, where and when we grade it, as it did for you, wherever you wrote it.