University of Maryland University College

CMSC 430 - Theory of Language Translation
Programming Project 2

The second project involves writing the syntactic analyzer for the compiler that was begun in the previous project. The grammar of the language is the following:

```
program:
  {function}

function:
  FUNCTION IDENTIFIER [parameters] RETURNS type ; body

parameters:
  parameter {, parameter}

parameter:
  IDENTIFIER : type

type:
  INTEGER | REAL | BOOLEAN

body:
  {variable} BEGIN statement END ;

variable:
  IDENTIFIER : type IS statement

statement:
  expression ; |
  IF expression THEN statement ELSE statement ENDIF ;

expression:
  IDENTIFIER |
  IDENTIFIER (expression {, expression}) |
  INT_LITERAL | REAL_LITERAL | BOOLEAN_LITERAL |
  NOT expression |
  expression operator expression |
  (expression)

operator: ADDOP | MULOP | RELOP | AND | OR
```

In the above grammar, the red symbols are nonterminals, the blue symbols are terminals and the black punctuation are EBNF metasymbols. The braces denote repetition 0 or more times and the brackets denote optional.

The grammar must be rewritten to eliminate the EBNF brace and bracket metasymbols and to incorporate the significance of parentheses, operator precedence and left associativity for all operators. Among arithmetic operators the multiplying operators have higher precedence than the adding operators. All relational operators have the same precedence. Among the binary logical operators, `and` has higher precedence than `or`. Of the categories of operators, the unary logical operator has highest precedence, the arithmetic operators have next highest precedence, followed by the relational operators and finally the binary logical operators. The directives to specify precedence and associativity, such as `%prec` and `%left`, may not be used

The syntactic analyzer should be created using bison. It should detect and recover from syntax errors to the extent possible. The semicolon should be used as the primary synchronization token. The compiler should produce a listing of the program with error messages included after the line in which they occur. A count of the number of lexical and syntactic errors and the number of total errors should follow the compilation listing.

Your parser should, however, be able to correctly parse any syntactically correct program without any problem.

You will lose points from the design portion of your grade if your bison input produces any shift/reduce or reduce/reduce errors.

The 70 points that you will receive for the functionality portion of your grade on this project will be based two criteria shown below:

Parses all syntactically correct programs	40 points
Detects and recovers from errors in the function header	5 points
Detects and recovers from errors in variable declarations	5 points
Detects and recovers from errors in conditional expressions	5 points
Detects and recovers from errors in arithmetic expressions	5 points
Detects and recovers from errors in the function body	5 points
Detects and recovers from multiple errors	5 points

Test data will be provide to test each of the above cases.

The next two phases of the project do not require that you implement error checking.