Sabancı University
Faculty of Engineering and Natural Sciences

CS305 Programming Languages

Homework 2

Due: March 6, 2020 - Friday @ 23:55

1 Introduction

In this homework you will write a context free grammar and implement a simple parser for BMO language for which you designed a scanner in the previous homework. Note that, there might be differences between the syntax of BMO language given in the previous homework and this one. Therefore, take the explanations on the syntax of BMO language given in this homework.

The language that will be generated by your grammar and other requirements of the homework are explained below.

2 Subset of the Language

The grammar you will write will generate the BMO language as described below. Here is an example program in this language to give you an idea how a BMO program looks like.

```
int vector digits, digits2 = [1,2,3,4,5,6,7,8,9,0];
int matrixM = 3;
int a = 5;
real realNum = 4.0E-12;
int resultV = a + matrixM;
int finalResult = a + resultV * 5;
int matrix result = [1,2;3,4;5,6;7,8;9,0];
int matrix matrix_1 = [1.2,2.2;3.2,4.1];
int matrix resultT = transpose(result);
digits = digits * matrixM;
int dotProduct = digits .* digits;
if(dotProduct <= matrixM || dotProduct >= result)
    resultT = resultT * dotProduct;
endif
resultT = b;
```

Note that the set of statements given above does not have to make sense. It may even have errors as a BMO program (e.g. an undeclared variable being used, multiplying vector with a matrix, or assigning a non-vector variable to a vector value). We are only using it for explanation purposes. Just focus on the syntactic features used in this example. Below is the detailed syntactic features of the BMO language.

1. A BMO program consists of a non-empty sequence of statements.
2. Each statement in our BMO can be a declaration, or an assignment statement or an `if` statement.
3. A declaration is given by the type, followed by a comma separated list of variable names, followed by an equality sign (i.e. the character =), followed by an expression, and followed by a semicolon.
4. The type can be one of the following: `int`, `int vector`, `int matrix`, `real`, `real vector`, `real matrix`.
5. An assignment statement given by a variable name, and an equality sign (i.e. the character =), and then an expression, finally followed by a semicolon. E.g.

   a = b+c;

6. An if statement starts with keyword `if`. After the keyword `if`, a boolean expression resides betweeen (and). This is followed by the body of the if statement, which is a non–empty sequence of statements. Finally, the keyword `endif` is given.
7. An expression can be a variable, or a number (integer or real), or a vector literal, or a matrix literal, or two expressions being applied to one of the operators $*$,+,−,/,.$*$ or a transpose expression.
8. A vector literal is given by a `[`, followed by a value row, followed by a `]`.
9. A value row is a comma separated list of values, where a value is either a number (integer or real), or a variable name.
10. A matrix literal is given by a `[`, followed by a semicolon separated list of value rows, followed by a `]`. There has to be at least two rows in a matrix literal.
11. A transpose expression is given by the keyword `transpose`, followed by a `(`, followed by an expression, followed by a `)`.
12. A boolean expression is either a simple comparison or two boolean expressions connected with a boolean operator, where we only have `&&` and `||` as the boolean operators.
13. A simple comparison must have exactly two variables separated with one of the comparison operators `>,<,>=,>=,!=,==`.

3 Terminal Symbols

Use the following terminal symbols in your grammar. You can assume that the scanner will return the corresponding token. Do not add any new tokens and do not change the name of the tokens.

`tINTTYPE`: The scanner returns this token when it sees "`int`" in the input.
`tINTVECTORTYPE`: The scanner returns this token when it sees "`int vector`" in the input.
`tINTMATRIXTYPE`: The scanner returns this token when it sees "`int matrix`" in the input.
`tREALTYPE`: The scanner returns this token when it sees "`real`" in the input.
`tREALVECTORTYPE`: The scanner returns this token when it sees "`real vector`" in the input.
`tREALMATRIXTYPE`: The scanner returns this token when it sees "`real matrix`" in the input.
`tTRANSPOSE`: The scanner returns this token when it sees "`transpose`" in the input.
`tIDENT`: The scanner returns this token when it sees an identifier in the input.
`tDOTPROD`: The scanner returns this token when it sees "`.*`" in the input.
`tIF`: The scanner returns this token when it sees "`if`" in the input.
`tENDIF`:The scanner returns this token when it sees "`endif`" in the input.
`tREALNUM`: The scanner returns this token when it sees a real number in the input.
`tINTNUM`: The scanner returns this token when it sees an integer number in the input.
`tAND`: The scanner returns this token when it sees "`&&`" in the input.
`tOR`: The scanner returns this token when it sees "`||`" in the input.
`tGT`: The scanner returns this token when it sees "`>`" in the input.
`tLT`: The scanner returns this token when it sees "`<`" in the input.
`tGTE`: The scanner returns this token when it sees "`>=`" in the input.
`tLTE`: The scanner returns this token when it sees "`<=`" in the input.
`tNE`: The scanner returns this token when it sees "`!=`" in the input.
`tEQ`: The scanner returns this token when it sees "`==`" in the input.

Besides these tokens, you if you need any more tokens in your grammar (e.g. `[`, `;`, etc.), directly use the lexeme of such tokens in your grammar.

4 Non-Terminal Symbols

Use the following non-terminal symbols in your grammar. This is the entire set of non-terminals that you will use in your grammar. Do not add a new non-terminal symbol, or do not change the name of a non-terminal symbol given here.

`<prog>`: This is the start symbol of the grammar.
`<stmtlst>`: Denotes a non–empty list of statements.
`<stmt>`: Denotes a single statement.
`<decl>`: Denotes a declaration.

`<type>`: Denotes the type in a declaration.
`<vars>`: Denotes the non–empty variable list in a declaration.
`<asgn>`: Denotes an assignment statement.
`<if>`: Denotes an if statement.
`<expr>`: Denotes an expression.
`<transpose>`: Denotes a transpose expression.
`<vectorLit>`: Denotes a vector literal.
`<matrixLit>`: Denotes a matrix literal.
`<row>`: Denotes a non–empty list of comma separated values.
`<rows>`: Denotes a non–empty list of semicolon separated rows. Note that only 1 row is possible, where a matrix literal requires at least 2 rows.
`<value>`: Denotes a value that can be either an integer number, or a real number, or a variable reference.
`<bool>`: Denotes a boolean expression.
`<comp>`: Denotes a simple comprasion.
`<relation>`: Denotes a binary relational operators (such as ==, etc.) that can be used in a simple comparison.

5 Output

Your parser must print out `OK` and produce a new line, if the input is grammatically correct. Otherwise, your parser must print out `ERROR` and produce a new line.

6 How to Submit

Submit your Bison file named as `username-hw2.y`, and flex file named as `username-hw2.flx` where `username` is your sucourse username and **do not zip your files.** We will compile your files by using the following commands:

```
flex username-hw2.flx
bison -d username-hw2.y
gcc -o username-hw2 lex.yy.c username-hw2.tab.c -lfl
```

So, make sure that these three commands are enough to produce the executable parser. If we assume that there is a text file named test17.BMO, we will try out your parser by using the following command line:

```
id-hw2 < test17.BMO
```

If the file `test17` includes a grammatically correct BMO program then your output should be `OK` otherwise, your output should be `ERROR`.

7 Notes

- **Important**: Name your files as you are told and **don't zip them.** [-10 points otherwise]
- **Important**: Make sure you include the right file in your scanner and make sure you can compile your parser using the commands given in the section 6. If we are not able to compile your code with those commands your **grade will be zero for this homework.**
- Make sure your output is exactly as it is supposed to be. (i.e. OK for grammatically correct and ERROR otherwise)
- No homework will be accepter if it is not submitted using SUCourse.
- You may get help from our TA or from your friends. However, **you must write your bison file by yourself**.
- Start working on the homework immediately.
- No late submission will be accepted.
- Since the grading will be done automatically on the flow.sabanciuniv.edu machine, we strongly encourage you to at least test your code on flow before submitting it.