DV1656 - Assignment 1

—

Lexical and Syntax Analysis

Suejb Memeti
Blekinge Institute of Technology

January 26, 2023

1 Introduction

In this assignment you are going to implement:

1. Lexical analysis (Scanner) for a given language specification (described in Section 2).
2. Syntax analysis (Parser) for a given EBNF grammar (described in Section 3)
3. Solve some theoretical exercises related to lexical and syntax analysis (described in Section 4).

Accordingly, this assignment is split into three parts, (1) Part 1: Lexical analysis, (2) Part 2: Syntax analysis and (3) Part 3: Theoretical exercises.

The goal of the first part of this assignment is to use the Flex tool to write a lexical specification that recognizes the language constructs required by the MiniJava grammar.

The goal of the second part of this assignment is to use the Bison tool to write a syntax specification for the given MiniJava grammar and generate the abstract syntax tree (AST).

The goal of the third part of this assignment is to demonstrate the knowledge and understanding of the theory behind the lexical and syntax analysis.

The examination of this project is done through demonstrations during the lab sessions. It is expected that you submit the source code in Canvas. The source code should be compressed in a zip/tar file.

Furthermore, you should also submit a report for the theoretical part. The report should be in .pdf format, and this file should be sent separately from the source code. A corresponding folder in Canvas will be published for submitting the report. The deadline for submitting the report is February 12, 2023.

The (soft) deadline for demonstrating this assignment is the last lab session for assignment 1, i.e., February 12, 2023.

The solution should be implemented using C or C++, and it should be compiled and executed correctly on a Unix-based operating system.

1.1 Laboratory groups

You are encouraged to work in groups of two students. Groups larger than two are not accepted. Groups of one student are accepted, but not encouraged. The project is designed for groups of two students.

Discussion and help between laboratory groups are encouraged. However, you should be careful to not cross the border to plagiarism, see section 1.3 below.

1.2 Lecture support

There are three lectures in total that are related to this assignment. One of the lectures focuses on Lexical analysis and two other lectures focus on Syntax Analysis.

In these lectures, we :

- Introduce the general concepts and theory of lexical and syntax analysis, including formal languages, regular expressions, context free grammars, top-down and bottom-up parsing.
- Provide examples related to lexical and syntax analysis.
- Explain common problems and solutions related to parsing.
- Provide a brief explanation of how flex and bison tools work. Additional information about these two tools can be found in the suggested literature book.

1.3 Plagiarism

All work that is not yours own should be properly referenced. If not, it will be considered as cheating and reported as such to the university disciplinary board.

2 Part 1: Lexical analysis

2.1 Problem Description

The goal of this assignment is to use the flex tool to write a lexical specification that recognizes the language constructs required by the MiniJava grammar. The MiniJava language specification was introduced in the book "Modern Compiler Implementation in Java" by Andrew Appel. In this course we have adopted the MiniJava language specification, and we have made slight changes to it. Listing 1 shows the MiniJava language specification.

Listing 1: The MiniJava language specification

```
            Goal ::= MainClass ( ClassDeclaration )* <EOF>
       MainClass ::= "public" "class" Identifier "{" "public" "static"
          "void" "main" "(" "String" "[" "]" Identifier ")" "{"
          Statement "}" "}"
 ClassDeclaration ::= "class" Identifier "{" ( VarDeclaration )* (
    MethodDeclaration )* "}"
   VarDeclaration ::= Type Identifier ";"
MethodDeclaration ::= "public" Type Identifier "(" ( Type Identifier ( "
   ," Type Identifier )* )? ")" "{" ( VarDeclaration | Statement )* "
   return" Expression ";" "}"
            Type ::= "int" "[" "]"
                   | "boolean"
                   | "int"
                   | Identifier
       Statement ::= "{" ( Statement )* "}"
                   | "if" "(" Expression ")" Statement ("else"
                     Statement)?
                   | "while" "(" Expression ")" Statement
                   | "System.out.println" "(" Expression ")" ";"
                   | Identifier "=" Expression ";"
                   | Identifier "[" Expression "]" "=" Expression ";"
      Expression ::= Expression ( "&&" | "||" | "<" | ">" | "==" | "+"
          | "-" | "*" | "/" ) Expression
                   | Expression "[" Expression "]"
                   | Expression "." "length"
                   | Expression "." Identifier "(" ( Expression ( ","
                     Expression )* )? ")"
                   | <INTEGER_LITERAL>
                   | "true"
                   | "false"
                   | Identifier
                   | "this"
                   | "new" "int" "[" Expression "]"
                   | "new" Identifier "(" ")"
                   | "!" Expression
                   | "(" Expression ")"
      Identifier ::= <IDENTIFIER>
```

Note that the grammar is written in the Backus-Naur form (BNF), which is a notation used for context-free grammars. A brief summary of BNF:

- BNF uses a range of symbols and expressions that constitute the so-called production rules. For example, **Type ::= "int" "[" "]" | "boolean" | "int" | Identifier** means that a type could be either an array of integers, a boolean, an integer, or an identifier.
- The reserved keywords or symbol and operators of the language are enclosed with " " and outlined

in blue color. For example, the **"boolean", "this", "public", "!", "("**, ... are all constants, and they should be matched as such. These are called terminals, as they cannot be expanded further.

- The <**INTEGER_LITERAL**> and <**IDENTIFIER**> are also terminals, because they cannot expand further. The only difference between the keywords and the integer literals and identifiers is that the later require the use of regular expressions to match/recognize them. You should think what a valid integer is and what is not, or what is a valid identifier and what is not and write the regular expression accordingly.
- **The MainClass, ClassDeclaration, Identifier, Expression**, ... (indicated with yellow color) are called non-terminals mainly because they can be expanded. For instance, **VarDeclaration** could become either **int var; or int[] var; or boolean [var]** ...
- MiniJava supports single-line comments that start with //. Note that the syntax and the semantics of a MiniJava program are inherited from the Java language. This means, that if you are having troubles understanding what the grammar and the language constructs in the grammar mean, then think how such things work in Java.

2.1.1 Tasks to Complete

You are supposed to do the following:

- Get familiarized with the getting started example (See Section 6 in the Appendix).
- Write a lexer that recognizes the above grammar (Listing 1). You may build upon the getting started example to write your lexer that recognizes the MiniJava language specification.
- For each of the identified language constructs generate a token that will later be used by the parser.
- A set of valid java classes can be found in the *test_classes/valid* folder of the *getting_started* example. Additionally, the *test_classes/lecical_errors* folder contains a set of lexically incorrect classes, which may help you test whether your compiler can catch lexical errors. You are encouraged to extend these sets with additional examples to demonstrate that the scanner can recognize all the lexically valid java programs, and report errors for the ones that cannot be recognized.

2.1.2 Recommended approach

My recommendation is that you focus on the lexer first, i.e., try to write the lexer rules to identify all the language constructs of the MiniJava, and print the token name for each of them. For example, if the scanner recognizes a *while* I would print a token name *WHILE*, or if the scanner reads a *+*, I would print *ADDOP*, ...

The following shows an example of what the action for a rule in the lexer looks like:

"while" { *print("WHILE");* }

In the next step, we will generate the actual tokens that will be used by the parser (See Section 3).

Note that Flex spills out all the unidentified tokens, which means that if you see parts of the input printed out when you run the lexer, that means that those parts of the input are not recognized. A simple way to handle lexical errors is provided as part of the *getting_started* example.

3 Part 2: Syntax analysis

3.1 Problem Description

The goal of this assignment is to use the Bison tool to write a syntax specification for the given MiniJava grammar (Listing 1) and generate the abstract syntax tree (AST). Note that bison can automatically generate a parser for a given grammar, which means that the task in this assignment is not to implement any of the parsing algorithms introduced in the lectures, instead we simply write our grammar and let bison do that for us.

A short introduction to bison can be found in Canvas under the Modules section ("introduction_to_bison.pdf). An extended reference for the bison specification is provided in chapter 6 of the Flex and Bison book.

3.1.1 Tasks to Complete

You are supposed to do the following:

- Get familiarized with the getting started example (See Section 6 in the Appendix).
- Use Bison to write a syntax specification for the above grammar (Listing 1). You may build upon the getting started example to write your parser. Note that you need to convert the grammar to left-recursive form.
- Rewrite the grammar until all reduce-reduce errors are eliminated.
- Shift-reduce errors - try to resolve them, otherwise let Bison take care of them.
- Generate the abstract syntax tree (See Section 3.1.3).
- A set of valid java classes can be found in the *test_classes/valid* folder of the *getting_started* example. For each of the valid examples, check that the correct syntax tree is generated. Additionally, the *test_classes/syntax_errors* folder contains a set of syntactically incorrect classes, which may help you test whether your compiler can catch syntax errors. You are encouraged to extend these sets with additional examples to demonstrate that the parser can recognize all the syntactically valid java programs, and report errors for the ones that cannot be recognized.

3.1.2 Recommended approach

The recommended approach for this part is first to get familiar with how Bison works (Read the introduction_to_bison.pdf file) . Then open the *parser.yy* file and add all of the tokens that you have printed from the first part of the assignment. An example of defining tokens is provided in the getting started example (parser.yy).

We can define tokens as *%token <token_type> TOKENNAME.*

We can also define types for the production rules as *%type <type> productionRuleName.*

Change the actions in the lexer file, such that instead of printing the tokens you would return the corresponding tokens. E.g.,

"for" { *return yy::parser::make_FOR();* } or

*[0-9]** { *return yy::parser::make_INT(yytext);* }. Note that the regex for integers here is simplified. Also note that for some tokens we need their values, hence the *yytext* is sent as parameter to the make function.

Take the MiniJava EBNF grammar and convert it into a left-recursive form. Put the grammar in Bison. It is recommended that you use a step-by-step approach, where you select a small feature of MiniJava and add the grammar for that. Test it, and if it works, then you may start adding another feature.

3.1.3 Constructing the Abstract Syntax Tree

As part of the getting started example, you may find a class *Node.h*, which is used to store the AST. This is a very simple class, it has a variable named *id* (which is mainly used for pretty printing the tree), *lineno* (which keeps track of the line number in the provided source code; *lineno* is used in the later

phases of the compiler), *type* and *value* (which are used to store the information such as type of the node and the value of that node, for instance *type: Int; value: 3; or type: ID; value: varName*).

Note that the implementation of the node class is a single type of *Node*, which basically stores the type of the node in the type string. The getting started example uses a single type of node to construct the AST. An OOP developer may prefer to have a class hierarchy of node types, such that the *Node* would be the base class and then have specific types of nodes for each language construct by simply deriving the base class. For example, we could have a specific node type for arithmetic expression, assignment statements, if and while statements, etc. Then in Bison (in the rule action part) you would call the corresponding constructor.

The node class has a list of other *Nodes*, that is used to store the children of this node. Note that the entry point in our grammar creates the root node, and then we iteratively generate other *Nodes* that represent subtrees each corresponding to a specific language construct. For example, the tree for a simple expression *10 + (12 * 18)* should look like the figure below. In this case the root node is *Expression*, which has a child named *AddExpression*. The *AddExpression* has two children *Int* and *MultExpression*, and so on. This example depicts how we can construct a large AST by combining subtrees that correspond to subexpressions in our language.

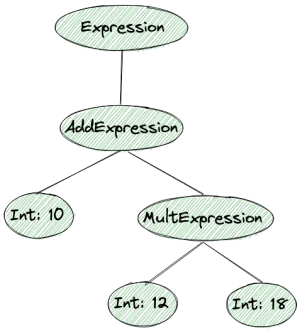

Figure 1: Example of an Abstract Syntax Tree for the expression: *10 + (12 * 18)*.

The node class has two pre-implemented methods named *print_tree()* and *generate_tree()*, both corresponding to printing the AST. In this case, *print_tree()* prints the tree to the console, and *generate_tree()* generates a dot file that could be compiled using *graphviz* tool to generate a visual representation of the AST (use *make tree* command to generate the tree). If you do not have *graphviz* tool, then you can install it using this command: *apt-get install graphviz*.

For each feature of the MiniJava check that the tree is properly constructed. In specific cases, you might need to rewrite the grammar if the tree is not generated correctly. For instance, the operator precedence should be encoded in the grammar to generate the correct AST.

Please note that some information might not be needed in the AST. For instance, in Figure 3.1.3 we have not created Nodes for the *(* and *)* symbols, because they are simply used to group expressions. We have grouped such expression by generating the subtree MultExpression, hence, such details are not needed. In the AST we only store important information that is needed for the later phases of the compiler.

A set of java classes that can be used for validating the correctness of the tree can be found in the *test_classes/tree_validation* folder of the *getting_started* example. For example, you may use the *operatorpriority.java* class to validate if the correct tree is generated for various arithmetic and logical operators; or you may use the *dangling_else.java* example to see if the correct tree is generated for if-else statements.

4 Part 3: Theoretical exercises

4.1 Lexing

- Write regular expressions that can recognize integer and float numbers.
  Note that valid integers are all possible single-digits (0-9), multi-digits that do not start with 0, and any negative number that do not start with 0. For example, 10, 0, -12, 1, -7, etc. Anything else is considered an invalid integer number. For example, 04, -0, -015, etc.
  Examples of valid float numbers include, 10.0, 3.14, 0.22, -0.22, -20.38, 0.002, etc. Examples of invalid float numbers include, .5, 23., 01.10, -0.0, etc.
- Construct a Finite Automaton from the given regular expressions.
- Merge the two automatons using the empty transitions, which would form a new nondeterministic finite automaton. Use the Power Construction Set algorithm to convert the NFA into a DFA. (Show the conversion steps also.)

4.2 Parsing

For the given grammar:

1. $M \rightarrow VSr$
2. $V \rightarrow VTIs \mid \epsilon$
3. $S \rightarrow SIeEs \mid \epsilon$
4. $T \rightarrow i \mid b \mid I$
5. $I \rightarrow n$
6. $E \rightarrow x \mid I$

where M is the starting symbol, $\{M, V, S, T, I, E\}$ are non-terminals, and $\{r, s, e, i, b, n, x\}$ are terminals. This grammar accepts following strings: $\{r$, *nnsr, nensr, insbnsr, nnsnnsnensr, insbnsinsnnsinsr, bnsbnsnnsr, nnsinsnnsr, nnsinsnexsr, nnsbnsinsnensr, insnnsr, nensnensr, insnexsnexsr, ...*$\}$
For the above grammar, do the following:

- Compute NULLABLE, FIRST, and FOLLOWS functions, and construct the parse table. It is expected that you annotate which rule was used a terminal (similarly to how we did it in the lecture).
- Prove that this grammar is not LL(1) grammar.
- Rewrite the grammar (using the techniques explained in the Syntax Analysis lecture) to make it LL(1)construct the parse table again (show the steps to compute NULLABLE, FIRST, and FOLLOWS). Note that LL(1) requires the elimination of ambiguity, left-recursion, and nondeterminism from the grammar.
- Using the following string: *"insbnsinsnnsinsr"* to simulate a stack-based execution of an LL(1) parser using the newly constructed parse table.

5 Examination

During the demonstration:

- Your compiler should be easily compiled using the Makefile.
- You should be able to explain briefly how you have dealt with specific issues. For example, "How did you handle operator precedence?". Show me the code that does ...
- Have a set of valid test classes and invalid test classes ready. During the demonstration, you should use those test classes to demonstrate that the compiler accepts the valid ones and rejects or shows error messages for invalid ones.
- You are expected to show the AST visually (i.e. use the make tree command) for each of the test classes used.
- The report related to the theoretical exercises will be evaluated offline.

6 Getting started example

I have put together a very simple lexer and parser, that can recognize very simple arithmetic expressions. The lexer recognizes the integers and the operators and generates the corresponding tokens. The parser then reads the sequence of tokens and checks the syntax according to the defined grammar using bison. The getting started package contains the following files:

- The lexer.flex file - which contains the language specification
- The parser.yy file - which contains the syntax specification. Note that the bison file right now prints the rule number for each of the production rules. It will help you to understand how a bottom-up parser works. Feel free to remove them at any time.
- The main.c - which contains the entry point of our compiler (I refer to the project as compiler, even though, at this point we have only are working at the first two phases of the compiler)
- The *Makefile* - which contains the necessary commands to build the compiler
- The *test.txt* and *test_invalid.txt* - which contain simple test examples.

Assuming that you have installed the flex tool:

- You may use the *make* command to build the lexer and the parser
- You may use *make clean* command to clean the build files
- Execute the *./compiler file_name* from the command line to run the lexer and the parser with the input from the file named *file_name*. If no file name is provided as argument, then the compiler will read from stdin.
  Note that when you use the compiler, it will generate a tree.dot file, which is a graphviz visualization of the abstract syntax tree.
- Execute the *make tree* command to generate a *.pdf* file of the abstract syntax tree. Requires that the graphviz tool is installed in your system.