DV1655 - Assignment 1

—

Lexical and Syntax Analysis

Suejb Memeti
Blekinge Institute of Technology

January 22, 2025

1 Introduction

In this assignment you will:

1. Perform Lexical Analysis (Scanner) based on a provided language specification (outlined in Section 2).
2. Conduct Syntax Analysis (Parser) following a given EBNF grammar (explained in Section 3).

Accordingly, this assignment consists of two parts (1) Lexical Analysis and (2) Syntax Analysis.

The primary objective of the first part is to use the Flex tool to create a lexical specification capable of recognizing the language constructs defined by the MiniJava grammar.

For the second part, your goal is to utilize the Bison tool to develop a syntax specification for the provided MiniJava grammar and generate the abstract syntax tree (AST).

Your project evaluation will involve in-lab demonstrations. You should submit the source code via Canvas, compressed in a zip/tar file.

The soft deadline for completing this assignment is the final lab session for assignment 1.

Ensure that your solution is implemented using either C or C++, and it should compile and run correctly on a Unix-based operating system.

1.1 Laboratory groups

Collaborative work in pairs of two students for this assignment is suggested. Please note that groups larger than two students are not permitted. While individual submissions are accepted, I strongly recommend working in pairs, as the project is designed for groups of two students.

Collaboration and discussions between laboratory groups are encouraged. However, it is essential to maintain the distinction between collaborative learning and plagiarism, as outlined in section 1.3 below.

1.2 Lecture support

There are three relevant lectures that complement this assignment. One lecture focuses on Lexical Analysis, while the other two concentrate on Syntax Analysis. During these lectures, the following is covered:

- Introduction to fundamental concepts and theories of lexical and syntax analysis, encompassing formal languages, regular expressions, context-free grammars, and top-down and bottom-up parsing.
- Presentation of practical examples related to lexical and syntax analysis.
- Explanation of common challenges and their solutions in parsing.
- Brief overview of how the flex and bison tools function. For more in-depth information about these tools, please consult the recommended literature.

1.3 Plagiarism

We emphasize the importance of academic integrity. Any work that is not your own must be appropriately referenced. Failure to do so will be considered plagiarism and reported to the university disciplinary board, in accordance with section 1.3.

2 Part 1: Lexical analysis

2.1 Problem Description

The primary objective of this assignment is to utilize the Flex tool to create a lexical specification capable of recognizing the language constructs required by the MiniJava grammar. The MiniJava language specification is derived from the book "Modern Compiler Implementation in Java" by Andrew Appel, with slight modifications tailored to this course.

2.1.1 MiniJava Grammar

The MiniJava language specification, which serves as the foundation for this assignment, is provided in Listing 1. It is expressed in the Backus-Naur form (BNF), a notation for context-free grammars.

Listing 1: The MiniJava language specification

```
            Goal ::= MainClass ( ClassDeclaration )* <EOF>
       MainClass ::= "public" "class" Identifier "{" "public" "static"
          "void" "main" "(" "String" "[" "]" Identifier ")" "{"
          Statement ( 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>
```

Key Concepts in BNF:

- **Production Rules**: BNF employs various symbols and expressions to define production rules. For example, `Type ::= "int" "[" "]" | "boolean" | "int" | Identifier` indicates that a type can be an array of integers, a boolean, an integer, or an identifier.

- **Terminals**: Reserved keywords, symbols, and operators of the language are enclosed in double quotes and highlighted in blue. These are constants and should be matched accordingly. Terminals cannot be further expanded.
- **Regular Expressions**: Terminals like <INTEGER_LITERAL> and <IDENTIFIER> require regular expressions to recognize valid patterns. You should define regular expressions that match valid integers and identifiers.
- **Non-Terminals**: Elements like `MainClass, ClassDeclaration, Identifier, Expression`, etc., are indicated in yellow. These are non-terminals because they can be expanded. For example, `VarDeclaration` can expand to various forms like `"int var;", "int[] var;"`, or `"boolean [var]"`.
- **Comments**: MiniJava supports single-line comments that start with //. The syntax and semantics of MiniJava closely follow those of the Java language.

2.1.2 Tasks to Complete

You are supposed to do the following:

- **Familiarization with Getting Started Example**: Start by becoming familiar with the "getting started" example outlined in Section 5 in the Appendix. This will provide you with a foundational understanding of how to work with the Flex tool.
- **Lexer Development**: Your main task is to develop a lexer that can recognize the MiniJava language constructs defined in the grammar (Listing 1). You can use the "getting started" example as a basis and build upon it to create your lexer.
- **Token Generation**: For each identified language construct within the MiniJava grammar, generate appropriate tokens. These tokens will be later used by the parser to understand the structure of MiniJava programs.
- **Testing and Validation**: Utilize the provided sets of valid Java classes in the *test_classes/valid* folder of the "getting_started" example. Additionally, there is a collection of lexically incorrect classes in the *test_classes/lexical_errors* folder. These samples can assist you in testing your lexer's ability to identify lexical errors. It is highly recommended to expand these sets by adding additional examples to ensure that your lexer can recognize all lexically valid Java programs and correctly report errors for those that are not recognized.

2.1.3 Recommended approach

- **Lexer Rules for Language Constructs**: Begin by defining lexer rules that can identify all the language constructs specified in the MiniJava grammar. For each recognized construct, print the corresponding token name. For instance, when the scanner encounters the keyword "while," you should print the token name `WHILE`. Similarly, if it encounters a symbol like "+," you can print `ADDOP`. You can use the defined `USE_LEX_ONLY` macro to control the behavior of lex actions.

E.g. `"while" if(USE_LEX_ONLY) {print("WHILE");} else{return YY::PARSER::make_WHILE(yytext);}`

- **Token Generation**: After successfully identifying language constructs, proceed to generate actual tokens that will be used by the parser (as outlined in Section 3). Create tokens for each recognized construct and assign appropriate token names, such as `IDENTIFIER, INTEGER`, etc.
- **Handling Lexical Errors**: Flex will automatically report any unrecognized tokens, which can help you identify lexical errors. You can utilize the provided methods in the `getting_started` example to handle these errors effectively.

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 (as shown in Listing 1) and generate the abstract syntax tree (AST). It's important to note that Bison can automatically generate a parser for a given grammar. This means that the primary task in this assignment is not to implement parsing algorithms, as covered in the lectures, but to define our grammar, and Bison will handle the parsing process.

For additional guidance on Bison, you can refer to a short introduction available in Canvas under the Modules section ("introduction_to_bison.pdf"). For a more detailed reference on Bison specification, you can consult chapter 6 of the Flex and Bison book.

3.1.1 Tasks to Complete

You are supposed to do the following:

- **Getting Familiarized**: Start by becoming familiar with the "getting started" example, as outlined in Section 5 in the Appendix. This will provide you with the foundational understanding needed for working with Bison.
- **Syntax Specification**: Utilize Bison to create a syntax specification for the provided grammar (Listing 1). You can use the "getting started" example as a basis for writing your parser. Ensure that you convert the grammar into a left-recursive form.
- **Eliminate Reduce-Reduce Errors**: Rewrite the grammar as needed until all reduce-reduce errors are eliminated. This process involves refining the grammar to ensure that the parser can uniquely identify and parse language constructs without ambiguity.
- **Handling Shift-Reduce Errors**: Address shift-reduce errors wherever possible. If any shift-reduce errors remain unresolved, allow Bison to handle them as part of its parsing process.
- **Abstract Syntax Tree (AST) Generation**: Implement the generation of an Abstract Syntax Tree (AST), as detailed in Section 3.1.3. The AST will capture the hierarchical structure of parsed MiniJava code.
- **Testing and Validation**: Utilize the provided collection of valid Java classes in the *test_classes/valid* folder within the "getting_started" example. For each valid example, verify that the parser generates the correct syntax tree. Additionally, there is a set of syntactically incorrect classes in the *test_classes/syntax_errors* folder, which can assist you in testing your parser's ability to detect syntax errors. It is highly recommended to expand these sets by adding additional examples to demonstrate that the parser can recognize all syntactically valid Java programs and correctly report errors for those that are not recognized.

3.1.2 Recommended approach

- **Familiarize with Bison**: Start by gaining a good understanding of how Bison works by reading the `"introduction_to_bison.pdf"` file. This will provide you with a foundation for working effectively with Bison.
- **Token Definition**: Open the `"parser.yy"` file and define all the tokens that you previously printed in the lexer part of the assignment. You can define tokens using the format `%token <token_type> TOKENNAME`.
- **Production Rule Types**: Define types for the production rules using the format `%type <type> productionRuleName`. This step helps specify the expected types for different parts of your grammar.
- **Lexer Actions**: In the lexer file, modify the actions so that instead of printing the tokens, you return the corresponding tokens. For example:

  - For a keyword like `"for"`, use `return yy::parser::make_FOR();`.
  - For integers, use `return yy::parser::make_INT(yytext);`.

  Make sure to adapt the actions to return the appropriate tokens and values where necessary.

- **Grammar Conversion**: Take the MiniJava EBNF grammar and convert it into a left-recursive form. Then, incorporate this grammar into Bison. It's advisable to follow a step-by-step approach. Begin by adding the grammar for a small feature of MiniJava, test it to ensure it works correctly, and gradually expand by adding more features.

3.1.3 Constructing the Abstract Syntax Tree (AST)

- **Node Class**: In the provided `"Node.h"` class, you will find a simple structure for representing nodes in the AST. It includes attributes such as `"id", "lineno", "type",` and `"value"`. These attributes are used to store information about each node in the tree, including its type, value, and position in the source code.

- **Hierarchy of Node Types (Optional)**: While the provided implementation uses a single type of node (`"Node"`) to construct the AST, you may choose to create a hierarchy of node types. In this case, `"Node"` would serve as the base class, and you could derive specific node types for various language constructs. For instance, you could have distinct node types for arithmetic expressions, assignment statements, if and while statements, and so on. This approach allows for more object-oriented structuring of the AST.

- **Child Nodes**: Each node can have a list of child nodes, representing its children in the AST. These child nodes correspond to the subtrees that capture specific language constructs within the MiniJava code.

- **Tree Construction**: Constructing the AST involves creating nodes for each language construct and assembling them into a tree structure. The root node typically corresponds to the top-level construct in your MiniJava code, and you iteratively generate child nodes for subexpressions and substatements as you traverse the grammar.

- **Printing and Visualization**: The `"Node"` class provides pre-implemented methods for printing and visualizing the AST. The `"print_tree()"` method prints the tree to the console, while the `"generate_tree()"` method generates a dot file that can be compiled using the `"graphviz"` tool to create a visual representation of the AST. You can use the `"make tree"` command to generate the tree visualizations. If you don't have `"graphviz"` installed, you can install it using `"apt-get install graphviz"`.

- **Validation**: Test the correctness of your AST construction for various features of MiniJava. You may need to refine your grammar to ensure the correct generation of the AST. Consider operator precedence in the grammar to ensure that the AST reflects the correct order of operations. Selective Information: Be mindful of the information you store in the AST. Only store essential data needed for later phases of the compiler. For instance, symbols like parentheses `"("` and `")"` may not need to be represented explicitly in the AST if they are solely used for grouping expressions.

- **Validation Classes**: The `"getting_started"` example includes a set of Java classes in the `"test_classes/tree_validation"` folder that you can use to validate the correctness of your generated AST. For instance, you can use the `"operatorpriority.java"` class to check if the correct tree is generated for various arithmetic and logical operators. The `"dangling_else.java"` example can help validate if the correct tree is generated for `if-else` statements.

4 Examination

- **Makefile Compilation**: Ensure that your compiler can be easily compiled using the provided `Makefile`. This should streamline the compilation process for your demonstration.

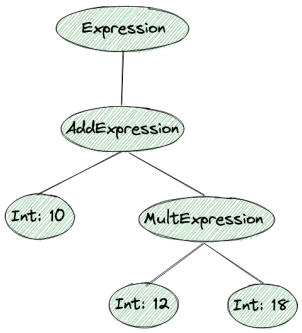

Figure 1: Example of an Abstract Syntax Tree for the expression: *10 + (12 * 18)*.

- **Explanation of Specific Issues**: Be prepared to explain how you handled specific issues in your compiler. For example, you may need to describe your approach to dealing with operator precedence. During the demonstration, you should showcase the relevant code or logic you implemented to address such issues.
- **Test Classes**: Have a set of both valid and invalid test classes readily available. During the demonstration, use these test classes to showcase that your compiler correctly accepts valid inputs and provides error messages or rejection for invalid ones.
- **Visual AST Presentation**: Utilize the `"make tree"` command to visually display the Abstract Syntax Tree (AST) for each of the test classes you use during the demonstration. This visual representation helps validate that your AST generation is functioning correctly.
- **Test script**: Ensure that the test script is functioning correctly on your computer. It will be utilized during the demonstrations to assess the overall functionality of the code.

5 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:

- **lexer.flex**: This file contains the language specification for the lexer. It recognizes integers and operators, generating corresponding tokens.
- `parser.yy`: The syntax specification is defined in this file. Note that the bison file currently prints the rule number for each production rule, aiding in understanding the operation of a bottom-up parser. You can remove these rule numbers as needed.
- `main.c`: This file serves as the entry point for our compiler project, even though it currently covers only the first two phases of the compiler.
- `Makefile`: This contains essential commands for building the compiler.
- `test.txt`, `test_lexical_errors` and `test_syntax_errors`: These files contain simple test examples for both valid and invalid input.
- `testScript.py`: A Python script that you can use to test the correctness of your solution. See Section 6 for more details.

To work with this example:

- **Install Flex Tool**: Ensure that you have installed the Flex tool.
- **Building the Compiler**:
  - Use the `make` command to build the lexer and parser.
  - Use `make clean` to clean up the build files.
- **Running the Compiler**:
  - Execute `./compiler file_name` from the command line to run the lexer and parser with input from the file named `file_name`. If no file name is provided as an argument, the compiler will read from `stdin`.
  - Note that when you use the compiler, it will generate a tree.dot file, which is a visualization of the abstract syntax tree.
- **Generating Abstract Syntax Tree (AST)**:
  - Execute `make tree` to generate a .pdf file of the abstract syntax tree. Ensure that the Graphviz tool is installed on your system for this feature.

6 Compiler Test Script

The Python script is designed to automate the testing of a custom compiler. It runs the compiler against a set of test files (categorized by test types such as lexical errors, syntax errors, etc.) specified via command-line arguments. After running the tests, the script enters an interactive mode where you can query specific test results or view a summary.

6.1 Features

- **Run Tests**: Automatically runs compiler tests in specified categories (e.g., lexical, syntax, semantic).
- **Interactive Mode**: After running tests, the script enters an interactive mode for detailed examination.
- **Detailed Examination**: Allows viewing detailed information about individual test files, including expected errors, unexpected errors, and raw compiler output.
- **Summary View**: Provides a summary of all test results, showing which tests passed or failed.

6.2 Running the Script

- **Start the Script**:
  - Run the script from the command line with one or more test types as arguments.
  - Valid Arguments:
    - `-lexical`: Runs tests in the `test_files/lexical_errors` directory. These tests check for lexical issues, such as invalid tokens or unrecognized characters.
    - `-syntax`: Runs tests in the `test_files/syntax_errors` directory. These tests validate the compiler?s ability to detect syntactic errors, such as missing brackets or improper nesting.
    - `-valid`: Runs tests in the `test_files/valid` directory. These tests ensure that valid files are parsed and compiled correctly without errors.
    - `-semantic`: Runs tests in the `test_files/semantic_errors` directory. These tests verify semantic correctness, such as type mismatches or undeclared variables, which are part of Assignment 2.
    - `-interpreter`: Runs tests in the `test_files/assignment3_valid` directory. These tests are designed for valid inputs related to interpreter functionality in Assignment 3.
  - Example: `python testScript.py -lexical -syntax -valid` This command runs all tests in the `lexical_errors`, `syntax_errors`, and `valid` directories.
- **Interactive Commands**:
  - `help`: Displays help information about available commands.
  - `<id>`: Displays the detailed test result for the file with the specified ID.
  - `<id> <output_type>`: Displays specific details for a file. The `output_type` can be `raw` (the raw output from the compiler), `expected` (the list of expected errors based on the file annotations, i.e., //@error ... comments), or `unexpected` (the errors that the compiler has reported but were not annotated in the source file).
  - `summary`: Shows a summary of all test results.
  - `exit`: Exits the script.

6.3 Command Examples

- `python testScript.py -lexical`: Runs only lexical error tests.
- `python testScript.py -valid -semantic`: Runs tests for valid files and semantic errors.
- `1 raw`: Shows the raw compiler output for the file with ID 1 in interactive mode.
- `2 expected`: Shows the expected errors for the file with ID 2 in interactive mode.
- `summary`: Displays a summary of all test results in interactive mode.

6.4 Script Output

- The script outputs summarized results for each test file after running all tests.
- In interactive mode, it provides detailed information based on user commands.
- Outputs are color-coded for clarity: green for success and red for failure or errors.

6.5 Notes

- Ensure the test files are organized in the appropriate directories (`test_files/lexical_errors`, `test_files/syntax_errors`, etc.) before running the script.
- The script assumes the compiler executable is named `./compiler` and is located in the same directory as the script.
- The script uses ANSI color codes for terminal output, which might not display correctly on all systems or terminals.