INFO-F403 : Project - part 1

Genial Imperative Language for Learning and the Enlightenment of Students - Lexer

Devroye Louis (000523920)
Akli-Kodjo-Mensah Gloria (000616562)

October 22, 2024

Contents

1 Regular Expressions - REGEX

For this first part of the project, we used *Extended Regular Expressions* to represent the needed lexical units:

$$\begin{aligned} \texttt{ProgName} &= \texttt{[A-Z][A-Za-z\_]*} \\ \texttt{VarName} &= \texttt{[a-z][A-Za-z0-9]*} \\ \texttt{Number} &= \texttt{[0-9]+} \\ \texttt{WhiteSpace} &= \texttt{(" " | "\t" | "\r" | "\n")+} \end{aligned}$$

These regular expressions need to be listed in a certain order since the first expression that matches will dictate the token recognized by the lexical analyzer.

1.1 ProgName

Identifies a program name, which is a string of letters and _ that must start with a capital letter. We ensure that the first character is restricted to uppercase letters with `[A-Z]` and, if any, followed by combination of letters or underscores with `[A-Za-z_]*` (this notation makes sure that program names with underscore character preceded and followed by letters are correctly handled).

1.2 VarName

Identifies a variable, which is a string of digits and letter that must start with a lowercase letter. We define this requirement as starting with `[a-z]`, ensuring that the first character is always a lowercase letter, then the variable name can contain any combination of letters and digits, represented by `[A-Za-z0-9]*`.

1.3 Number

This regular expression is designed to capture integral numerical constants using `[0-9]+`, indicating that a valid number consists of one or more digits.

1.4 WhiteSpace

To focus our analyze to on meaningful tokens of the GILLES language, we define thid regex to include spaces, tabs, carriage returns, and newlines, as specified by `(" " | "\t" | "\r" | "\n")+`. This ensures that any sequence of whitespace characters is treated as a single unit and ignored by the lexer.

2 Presentation of Our Work

In this section, we describe the key aspects of our implementation for the lexical analyzer. The goal was to design a scanner capable of recognizing the language's lexical units and managing a symbol table. We followed the instructions provided and utilized the JFlex tool for generating the lexical analyzer.

2.1 Lexical Analyzer

We designed the lexical analyzer in `LexicalAnalyzer.flex`. This file defines the rules for recognizing various tokens in the GILLES language, including program names (`ProgName`), variable names (`VarName`), numbers, and keywords such as `LET`, `IF`, and operators like `+`, `-`, `*`, and `/`. We used regular expressions to identify these lexical units and ensured proper tokenization by listing rules in the correct order.
We structured the file with three main sections: configuration (sets up the JFlex options that control the behavior of the lexical analyzer), lexical rules (define the different lexical states), and token definitions (contains the regular expressions for the different tokens and each token has an associated action, returning a Symbol object). At the beginning of the file, we define several important JFlex options. These options determine the behavior of the lexical analyzer and specify the structure of the tokens being generated. Three of them are particularly needed for our project:

1. `%type Symbol`: We specify that the tokens produced by the analyzer will be of type Symbol, which is a class that stores information about the lexical units (lexical type, value, line, and column).

2. `%function nextToken`: This defines the name of the function that retrieves the next token from the input stream. The nextToken function is invoked by the parser during the analysis phase (which will be useful to correctly format our output).

3. `%eofval`: The eofval block specifies the return value when the end of the file is reached. In this case, it returns a symbol indicating the end of stream (EOS).

We define several lexical states to handle different parts of the GILLES language syntax, particularly for handling comments:

1. `YYINITIAL`: This is the default state where the recognition of the regular token is handled (such as keywords, variable names, and numbers).

2. `SHORTCOMMENTS` and `LONGCOMMENTS`: These states are used to recognize and ignore comments in the source code. Once a comment is detected, the scanner enters one of these states and ignores any content until the comment is closed.

2.2 Handling Comments

Comments in GILLES can either be short (starting with a `$` symbol and ending at the line's termination) or long (enclosed between `!!` symbols). In both cases, the content within the comments is ignored by the lexer:

1. `Short Comments`: We designed the lexer to return to the YYINITIAL state when a newline is encountered, effectively skipping the comment.

2. `Long Comments`: Long comments can span multiple lines, and the lexer remains in the LONGCOMMENTS state until another !! is found. If the end of the file is reached without closing the comment, an exception is thrown to signal an error in the source file.

2.3 Error Handling

Unrecognized characters in the source code are handled by throwing a `PatternSyntaxException`, indicating that the input contains invalid tokens. There are 2 cases, the first one is when an unidentified character appears, the Lexer then throws a `PatternSyntaxException` indicating that an unexpected character has been read. The second case occurs when the end of the file is reached while inside of a long comment. In this scenario, the lexical analyzer expects the comment to be properly closed with `!!`. If the closing delimiter is missing and the file ends, a `PatternSyntaxException` is thrown, signaling that the comment was not closed properly.

2.4 Resulting output

In `main.java`, the program read a source file provided as a command-line argument (thanks to `FileReader`) and use the `LexicalAnalyzer` to tokenize the input. The tokens are processed using `lexer.nextToken()`, and each recognized token is printed. The loop continues until the end-of-file token (`LexicalUnit.EOS`) is reached.

For variables, the program maintains a *symbol table* using a `TreeMap` (to directly store the variables in a lexicographical order). When a new variable is encountered, it is added to the symbol table if it hasn't been seen before. At the end, the program prints each variable and the line number where it first appeared.

3 Test files

Several example files have been added, in addition to the provided `euclid.gls` file:

1. `AdditionalTokens.gls`: Verify that the lexer can handle redundant tokens such as "LET" and "END".

2. `InvalidNumber.gls`: Make sure that invalid number are detected by the lexer.

3. `InvalidProgName.gls` (resp. `InvalidVarName.gls`): Different files testing different types of invalid program (resp. variable) names to ensure that only valid names are recognized.

4. `LongComments.gls` (resp. `ShortComments.gls`): Ensure that well-formed long (resp. short) comments are handled correctly.

5. `UnclosedLongComments.gls`: Check the detection of unclosed long comments.

6. `ValidProgName.gls` (resp. `ValidVarName.gls`): Guaranty that well-formed `ProgName` (resp. `VarName`) are handled correctly.

7. `RandomWhitespaces.gls`: Verify that white spaces are properly ignored.

4 Nested Comments

The main difficulty of nested comments arises from detecting and properly balancing the opening and closing comment symbols. In our current implementation, GILLES does not support nested comments. To handle nested comments, the lexical analyzer would need to maintain a counter that tracks the depth of nesting, incrementing for each opening symbol and decrementing for each closing symbol, ensuring all comments are properly matched. If the counter reaches zero, the nested comment is fully balanced otherwise this would result in errors.