ECOLE POLYTECHNIQUE DE BRUXELLES

2023-2024

INFOF403 - Introduction to Language Theory and Compilation

PascalMaisPresque Lexer

Ferreira Brenno

Mutkowski Philippe

Gilles GEERAERTS

2023

Table des matières

1 Introduction

The first part of this report will talk about the implementation of a lexical analyzer for the PascalMaisPresque language.

It serves as the first crucial phase of the compilation process, responsible for turning the written code into a stream of meaningful symbols, or tokens. These tokens form the basis for subsequent analysis stages, including parsing, semantic analysis, and code generation. The lexical analyzer acts as the gatekeeper, ensuring that the source code adheres to the language's grammar, while also extracting essential information for further processing.

2 Token Identification and States

Tokens are like the building blocks of the code. The notion of token is implemented in the Symbol class (see section further). Token identification is done by matching the text from an input file to one of the regular expressions defined below. In the case of PascalMaisPresque, tokens are separated in a few categories. Here are all their expression and the corresponding lexical unit in the case of this project (see section further).

1. Reserved keywords : these are the words with special meaning that define specific actions. They can't be defined as variable names.
   - "begin" - LexicalUnit.BEG
   - "end" - LexicalUnit.END
   - "if" - LexicalUnit.IF
   - "then" - LexicalUnit.THEN
   - "else" - LexicalUnit.ELSE
   - "and" - LexicalUnit.AND
   - "or" - LexicalUnit.OR
   - "while" - LexicalUnit.WHILE
   - "do" - LexicalUnit.DO
   - "print" - LexicalUnit.PRINT
   - "read" - LexicalUnit.READ
2. Structure symbols : they are used to separate or group code elements.
   - "(" - LexicalUnit.LPAREN
   - ")" - LexicalUnit.RPAREN
   - "{" - LexicalUnit.LBRACK
   - "}" - LexicalUnit.RBRACK

3. Operation symbols : it's the symbols used for math and assigning data.
   - " :=" - LexicalUnit.ASSIGN
   - "-" - LexicalUnit.MINUS
   - "+" - LexicalUnit.PLUS
   - "*" - LexicalUnit.TIMES
   - "/" - LexicalUnit.DIVIDE
4. Relation symbols : used for comparison, here "equality" and "lower than".
   - "=" - LexicalUnit.EQUAL
   - "<" - LexicalUnit.SMALLER
5. Literals : these are constant values used for calculations and operations, they are usually assigned to variables. The regular expressions are defined below.
   - {VarName} - LexicalUnit.VARNAME
   - {Number} - LexicalUnit.NUMBER

2.1 Regular Expressions

This is the definition of the more complex regular expressions not defined above :

1. AlphaNum :

$$[a-zA-Z0-9]$$

   This regular expression accepts every string of character that contains lowercase letter, uppercase letters and numbers.

2. VarName :

$$[a-z]\{AlphaNum\}*$$

   The regular expression for name of variables is defined as starting by a lowercase letter and then, a string of alphanumeric characters. Thus, it starts by defining the lowercase letters and then by saying that it will be zero or several repetitions of alphanumeric characters.

3. Number :

$$"-"?[0-9]+$$

   The number is defined as a string of digits. That's why we defined numbers as being a string of numerical characters that could be negative by the eventual presence of a '-' symbol. The

symbol + at the end implies that it must have one or more occurrences of those characters in order to be considered as number.

4. Comments :

   Short comments :

   "**"

   Long comment :

   "’’"

   When the comment expressions are matched, the lexer switches states, depending if it's a long comment or a short comment.

5. EndOfLine :

   ["\r""\n""\r\n"]

   In order to identify the end of a short comment, we need to match the end of the line characters. Theses characters are $\backslash r$ the carriage return character and $\backslash n$ is the newline character.

2.2 States

In the lexer, there are 3 exclusive states. The "YYINITIAL" initial state is the main state when the code is running, it matches every expression like cited above. The state switches when the expression for a comment is matched.
The "LONG_COMMENT" state begins when the "’’" regex is matched (two ’ characters). In this state, everything is ignored except for the "’’" regex. When this regex is found, the lexer switches back to its initial state.
The "SHORT_COMMENT" ignores everything except for the EndOfLine operator, where it switches back to the initial state.

3 Design

3.1 Symbol.class

This is the class that implements the notion of token. Each symbol object is constructed with its lexical unit, a line and column number and its value. The class has methods like "toString()" used to convert the token to a string representation.

3.2 LexicalUnit class

The lexical unit class only contains an enumeration of the different lexical units used in PascalMaisPresque. It's used to create the correct Symbol object for each token.

3.3 Lexical Analyzer

The lexer (short for lexical analysis program, or lexical analyzer) is build in java 1.18.
The lexer is generated using JFlex. It's a tool that simplifies greatly the process of writing the actual lexer. It uses the principle of DFAs (deterministic finite automata). The technical details of writing JFLEX code will not be explained in depth in this report, but the working principle of JFlex is that it takes as input a specification with a set of regular expressions and corresponding actions. The main action when a regex is matched is to return the corresponding symbol (a Symbol object) or to switch states (in case of comments).

3.4 Main class

The main class is a important piece of code responsible for utilizing the Lexer to scan input files provided as command-line arguments. It can handle multiple input files, and each file is processed

by the lexer.

At the outset, the main class initializes all the necessary variables for its execution and supplies the input file to the lexer through its constructor. A map is created to associate variable names with their first occurrence line numbers. Subsequently, the standard output is directed to an output file, and the scanning of the input file commences.

The input file is scanned token by token, using the `yylex()` function from the lexer, which retrieves the next token. The code continues looping through tokens until it encounters the End of Stream (EOS). During this process, the code prints the symbol and its corresponding lexical unit. Whenever a variable is encountered, it is added to the variable map.

Finally, the code prints the variable table and closes the output file.

4 Test examples

4.1 All lexical units prompt

This validation test aims to show that the lexer can correctly recognize every defined lexical unit. The code itself is not very useful, but it is a thorough demonstration of all the lexical units we have defined. The output file that is produced attests to the successful identification of each and every lexical unit.

4.2 Comments prompt

This prompt contains a mix of short and long comments. The variable checkpoint serves as a demonstration that all comments are effectively ignored, ensuring that the code runs without any issues.

4.3 Fibonacci prompt

This is a sample code snippet written in PascalMaisPresque language, showcasing the presence of comments, assignments, and several keywords.

4.4 Euclid prompt

This is the example prompt provided in the project statement. It was utilized to confirm that our lexer functions in accordance with the statement's requirements.

Articles

[GSB] GEERAERTS, Gilles, Mathieu SASSOLAS et Léonard BRICE (s. d.). « PASCALMAISPRESQUE Introduction to language theory and compiling Project – Part ». en. In : ().

[KRD] KLEIN, Gerwin, Steve ROWE et Regis DECAMPS (s. d.). « JFlex User's Manual ». en. In : ().