INFO-F403

Compiler for PASCALMAISPRESQUE - Part 1 Report

Mohammad SECUNDAR –> 504107
Dave Pikop Pokam –> 506979

INFO-F403 - Introduction to language theory and compiling
BA3 INFO
Université Libre de Bruxelles

Table des matières

1 Introduction

This project aims to build a compiler for PASCALMAISPRESQUE, a straightforward imperative language. The compiler development involves four key steps : lexical analysis (scanning), syntax analysis (parsing), semantic analysis, and synthesis. In the initial phase of the project, we implemented the lexical analyzer using the JFlex tool in Java. The fundamental concept involves taking a program written in the PASCALMAISPRESQUE language as input and systematically analyzing, identifying, and tokenizing each symbol based on the language's rules.

2 Lexical Analyzer Design

2.1 Definition

The lexical analyzer, often referred to as the scanner or lexer, is responsible for reading the source code and breaking it down into recognizable sequences called tokens. These tokens are sequences of characters in the source code that represent a singular entity. The primary objective of the lexical analyzer in the context of this project is to recognize and extract valid tokens from the PASCALMAISPRESQUE source code. This step is crucial as it prepares the source code for subsequent stages of the compilation process.

2.2 Regular expressions

Regular expressions play a fundamental role in the context of lexical analysis. In the lexical analysis phase of the compilation process, regular expressions are used to define patterns that describe the syntax of individual tokens in the programming language. These patterns are then employed by the lexical analyzer to recognize and tokenize the input source code.

2.3 Lexical Units

The lexical units signify the various token types that the lexer is capable of identifying and producing as output during the processing of the input source code. Each lexical unit aligns with a particular category of tokens that the lexer can detect while analyzing the source code. These tokens serve as foundational elements, laying the groundwork for subsequent phases of syntactic and semantic analysis in the subsequent stages of your compiler or interpreter.

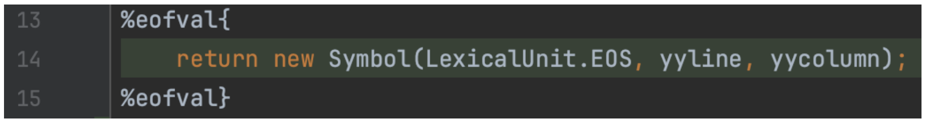
2.4 Symbol Table

The symbol table is a data structure used in the compilation process to associate variable names with information. This table is crucial to track variables and their attributes, ensuring proper code interpretation and error detection. In the context of our project, the Symbol class is vital in implementing the symbol table. It associates values with LexicalUnits, providing a mechanism to manage variables, their types, and other related attributes efficiently.

3 Implementation Details

Given a string in the input file is matched with one of our regular expressions, we display the corresponding token. Then, we print out token and its lexical unit correspondent. The variables identified during the file reading, are pushed in a TreeMap and they will be printed out at the end with their corresponding line numbers. This is done with the EOS token, we can identify that it is the end of the file.

EOS is defined in jflex source file as follow :



```
%eofval{
    return new Symbol(LexicalUnit.EOS, yyline, yycolumn);
%eofval}
```

3.1 JFlex Tool

JFlex is a powerful tool used for creating lexical analyzers in Java. It operates by reading a specification of the lexical analyzer to be generated in the form of regular expressions and then processes it to produce a Java class, which serves as the lexical analyzer. This analyzer can efficiently recognize and process the specified regular expressions. In the context of this project, JFlex plays a crucial role in recognizing lexical units for PASCALMAISPRESQUE.

We defined a **.flex** file with all the required specification for this language. This is the explanation of the file contain below.

Token Definitions

- Tokens are defined using regular expressions within the `<YYINITIAL>` state.
- Each regular expression corresponds to a specific lexical unit (e.g., keywords, operators, identifiers, numbers).
- Tokens are associated with specific actions, such as creating a `Symbol` object with relevant information (type, line number, column number).

States

- The lexer utilizes states (`<YYINITIAL>`, `<SHORTCOMMENTS>`, `<LONGCOMMENTS>`) to handle different scanning modes.

— For instance, it enters the `<SHORTCOMMENTS>` state when encountering `"**"` and ignores characters until the end of the line.
— The `<LONGCOMMENTS>` state is entered when encountering "”" and exited when encountering another "”".

Handling Comments

— Comments are ignored by entering the `<SHORTCOMMENTS>` or `<LONGCOMMENTS>` states, preventing them from being treated as regular code.

Error Handling

— An error is thrown when unmatched symbols are encountered, aiding in identifying and reporting syntax errors in the source code.

Special Tokens

— Special tokens like `LexicalUnit.EOS` (End of Stream) are handled for proper termination of the lexer.

Whitespace Handling

— Whitespace characters and end-of-line characters are ignored in the `<YYINITIAL>` state.

Variable Tracking

— The `addVariableIfNecessary` method is employed to track variables by adding them to a `TreeMap` if they are of type `LexicalUnit.VARNAME`.
— The `printVariables` method facilitates the printing of variables along with their corresponding line numbers.

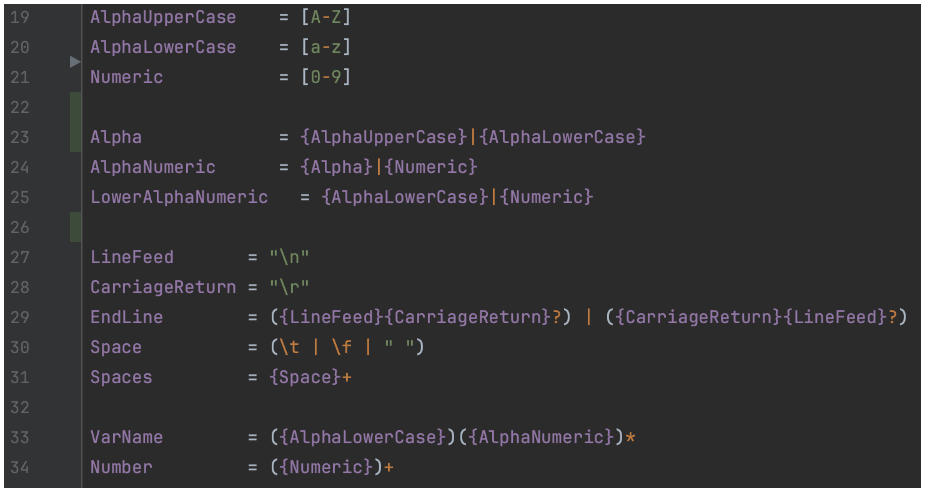
Exception Handling

— Exceptions, such as `FileNotFoundException` and `IOException`, are caught, and appropriate error messages are displayed.

3.2 Regular Expressions

3.2.1 Extended regular Expressions

We declared some basic regular expressions(Macros) which will be used later to create other complex regular expressions.
— **AlphaUpperCase** : Matches any single uppercase alphabetical character from A to Z.
— **AlphaLowerCase** : Matches any single lowercase alphabetical character from a to z.
— **Numeric** : Matches any single digit from 0 to 9.



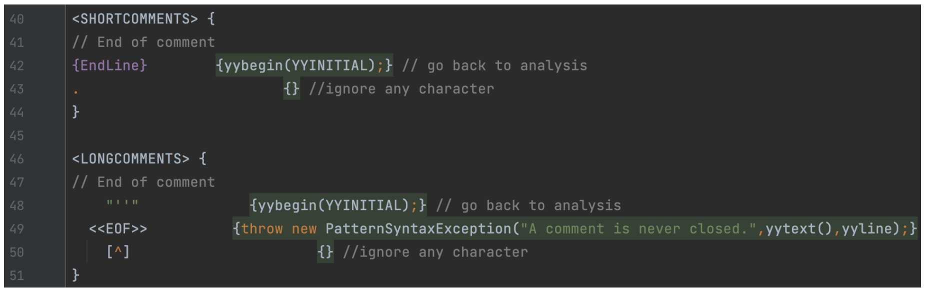
```
AlphaUpperCase    = [A-Z]
AlphaLowerCase    = [a-z]
Numeric           = [0-9]

Alpha             = {AlphaUpperCase}|{AlphaLowerCase}
AlphaNumeric      = {Alpha}|{Numeric}
LowerAlphaNumeric   = {AlphaLowerCase}|{Numeric}

LineFeed       = "\n"
CarriageReturn = "\r"
EndLine        = ({LineFeed}{CarriageReturn}?) | ({CarriageReturn}{LineFeed}?)
Space          = (\t | \f | " ")
Spaces         = {Space}+

VarName        = ({AlphaLowerCase})({AlphaNumeric})*
Number         = ({Numeric})+
```

— **Alpha** : Matches any single alphabetical character, regardless of case.
— **AlphaNumeric** : Matches any single alphanumeric character, i.e., any letter or digit.
— **LowerAlphaNumeric** : Matches any single lowercase alphabetical character or digit.
— **VarName** : Matches a variable name that begins with a lowercase letter, followed by zero or more alphanumeric characters.
— **Number** : identifies a numerical constant, and is made up of a string of digits only

— **LineFeed** : Defined as the line feed (newline) character, represented by `"\n"`. This character is used to mark the end of a line of text and the beginning of a new one.
— **CarriageReturn** : Defined as the carriage return character, represented by `"\r"`. This character is used to return the cursor to the beginning of the line without advancing to the next line.
— **EndLine** : A combination of `LineFeed` and `CarriageReturn`, designed to match different types of line endings used across various operating systems. It matches either a line feed followed optionally by a carriage return, or a carriage return followed optionally by a line feed. This ensures compatibility with Unix/Linux (which typically uses just a line feed), Windows (which uses a carriage return followed by a line feed), and older Macintosh systems (which use just a carriage return).
— **Space** : Defined to match any single whitespace character that is not a line feed or carriage return. It includes the tab character (`\t`), the form feed character (`\f`), and the space character (`" "`).
— **Spaces** : Builds upon the `Space` definition and matches one or more consecutive whitespace characters, as defined by `Space`.

3.2.2 Rules for handling comments

The <SHORTCOMMENTS> and <LONGCOMMENTS> are two states of the lexer which define rules for handling comments.



```
<SHORTCOMMENTS> {
// End of comment
{EndLine}          {yybegin(YYINITIAL);} // go back to analysis
.                           {} //ignore any character
}

<LONGCOMMENTS> {
// End of comment
    "'''"               {yybegin(YYINITIAL);} // go back to analysis
  <<EOF>>             {throw new PatternSyntaxException("A comment is never closed.",yytext(),yyline);}
    [^]                         {} //ignore any character
}
```

3.2.3 Rules for identifying tokens

The <YYINITIAL> state contains rules for identifying various tokens in the language, such as keywords (begin, end, if, then, etc.), operators (:=, +, -, etc.), and other constructs like numbers, variable names, etc.

3.2.4 Others rules

— The {...} in the rules represent actions to be taken when a certain pattern is matched. For example, when the lexer identifies the token `begin`, it returns a `Symbol` object with the type `LexicalUnit.BEG`.
— The {} in some rules signifies ignoring characters that match the pattern.
— The code handles spaces and end-of-line characters appropriately.
— The last rule, [ˆ], throws an exception if an unmatched symbol is found.

3.3 Main class

The **Main class** reads a PASCALMAISPRESQUE source file, creates a Lexical analyzer, and iterates through the tokens until the end of the stream is reached.

3.4 Environment

For this project, the environment chosen for implementation is JAVA 1.8. Two essential files have been provided to aid in the process : LexicalUnit.java and Symbol.java.

4 Tests

We performed multiple tests to verify the analyser compliance. The test files are in the test directory. The tests are based on Artithmetic Operators, Basic key words and operators, Boolean operators and comparaisons, comments, error scenarios, Loops and 3 other general files to test different cases.

5 Bonus : Handling Nested Comments

Dealing with nested comments can be challenging since it is difficult to distinguish the start and end of a comment when using the same delimiters. For instance, a single comment delimiter such as "

Example :

"This is a nested comment " Part two of the comment "
"Another nested comment " Part two of this comment " okay "

One potential, albeit suboptimal, solution involves the parser attempting every possible nested comment combination until the correct structure is identified. This approach, however, would exponentially increase the compiler's complexity. An alternative method could involve the compiler making educated guesses regarding which portions of the text contain nested comments. Nevertheless, this solution's feasibility is uncertain as the compiler must operate deterministically.