JFlex PCAT Compiler

Jad Salhani

February 25, 2016

Abstract

This report provides the detailed specification of the PCATCompiler built with JFlex and CUP. This program provides a Syntax Analyser and Parser for the basic programming language PCAT, focusing on basic syntax analysis and token parsing. After reading this report the user will be able to successfully use the Compiler program to parse PCAT program files and output the list of tokens contained in the code, underlining any lexical errors that might be present.

Contents

Part I

Introduction

Chapter 1

Introduction

In accordance with the requirements of the Compiler Construction course (or CMPS274) at the American University of Beirut, this compiler was built using open source lexing libraries and open-source parsing libraries.
The tools used for this project are JFlex to generate the Lexical Analyser and CUP to generate the Parser. Although the tools used generate Lexers and Parsers for Java, this compiler has been adapted to the PCAT imperative programming language.
This program is written with the Java programming language, using Flex for the Lexer language specification file.

Chapter 2

Overview

2.1 Tools Used

These tools have been used together because they have been implemented in a way as they kept each other in mind. JFlex works best with CUP, as it contains integration of the CUP parser and cup flags in the language specification. The language it creates as well is build to be compatible to the CUP parser generator.

2.1.1 JFlex

JFlex is a lexical analyzer generator (also known as scanner generator) for Java, written in Java.

This lexical analyzer generator takes as input a specification file written in flex which holds a set of regular expressions and corresponding actions. It generates a program (a lexer) that reads input, matches the input against the regular expressions in the spec file, and runs the corresponding action if a regular expression matched. Lexers are the first front-end step in compilers, matching keywords, comments, operators, etc, and generating an input token stream for parsers.

JFlex is designed to work together with an LALR parser (mainly the parser generator CUP). It could also be used with LL parsers such as ANTLR

2.1.2 CUP

CUP stands for Construction of Useful Parsers and is an LALR parser generator for Java. It implements standard LALR(1) parser generation i.e. the productions contain only Right Hand Side (RHS) non-terminals. This parser generator can also handle function calls in the productions in case reductions need to be executed on the production before being processed. Parsers are the second step after the Lexical Analysers; they check the syntax for errors and because of the production implementation they can check for logical errors and type errors.

2.1.3 Advantages

There are a lot of Lexical Analyzer generators and Parser Scanner generator tools available as open-source libraries and as authored projects.
The JFlex lexical analyzer library was chosen due to its lightweight setup and easy integration with Java projects. In addition, JFlex provides seamless integration with the CUP parser generator by just adding a couple of flags to the "language.flex" language specification file.
JFlex's documentation is pretty straightforward; it describes the various symbols used within an example and provides suggestions as to implementing them further.
The key advantages of JFlex are the following points:

- JFlex has column and line counters.
- JFlex has debugging support.
- JFlex can generate a standalone-lexer, independent of any parser.

2.1.4 Disadvantages

One of the downsides of JFlex is their documentation; Despite being straightforward and to the point, it assumes the reader has basic knowledge of the Flex language specification, and thus before understanding the JFlex documentation, one must understand the Flex documentation.

Part II

Lexical Analysis

Chapter 3

Project Details

3.1 File Description

The required files for this project are split into two folders;

- The src/ folder which holds the main files, mainly the PCATLexer class and the SymbolTable
- The symbol/ folder which holds the interfaces responsible for providing the token string keys

3.1.1 Symbols

The symbols keys have been implemented as interfaces that are implemented by the generated PCATLexer to make the keys available

Delimiters.java

```
public interface Delimiters {
    public static final String COMMA = "TOK_COMMA";
    public static final String COLON = "TOK_COLON";
    public static final String SEMI_COLON = "TOK_SEMI";
    public static final String DOT = "TOK_DOT";
    public static final String LEFT_BRACKET = "TOK_LB";
    public static final String RIGHT_BRACKET = "TOK_RB";
    public static final String LEFT_BRACE = "TOK_LC";
    public static final String RIGHT_BRACE = "TOK_RC";
    public static final String LEFT_PAREN = "TOK_LP";
    public static final String RIGHT_PAREN = "TOK_RP";
}
```

This interface holds the string codes for the PCAT Delimiters.

Operators.java

```
public interface Operators{
    public static final String NOT="NOT";
    public static final String OR="OR";
    public static final String AND="AND";
    public static final String EQUAL="=";
    public static final String NOT_EQUAL="<>";
    public static final String LESS_THAN="<";
    public static final String LESS_THAN_EQUAL="<=";
    public static final String GREATER_THAN=">";
    public static final String GREATER_THAN_EQUAL=">=";
    public static final String ASSIGN=":=";
    public static final String ADD="+";
    public static final String SUBSTRACT="-";
    public static final String TIMES="*";
    public static final String DIVIDE="/";
    public static final String IDIV="DIV";
    public static final String MOD="MOD";
}
```

This interface holds the string code for the PCAT Operators. It is used concurrently with the OperatorHandler class which matches the token from the Operators interface with a key from the token-value map generated by the handler.

ReservedWords.java

```
public interface ReservedWords {
    public static final String ARRAY = "TOK_ARRAY";
    public static final String BEGIN = "TOK_BEGIN";
    public static final String BY = "TOK_BY";
    public static final String DIV = "TOK_DIV";
    public static final String DO = "TOK_DO";
    public static final String ELSE = "TOK_ELSE";
    public static final String ELSEIF = "TOK_ELSEIF";
    public static final String END = "TOK_END";
    public static final String EXIT = "TOK_EXIT";
    public static final String FOR = "TOK_FOR";
    public static final String IF = "TOK_IF";
    public static final String IS = "TOK_IS";
    public static final String LOOP = "TOK_LOOP";
    public static final String NOT = "TOK_NOT";
    public static final String OF = "TOK_OF";
    public static final String PROCEDURE = "TOK_PROCEDURE";
    public static final String PROGRAM = "TOK_PROGRAM";
    public static final String READ = "TOK_READ";
    public static final String RECORD = "TOK_RECORD";
    public static final String RETURN = "TOK_RETURN";
    public static final String THEN = "TOK_THEN";
    public static final String TO = "TOK_TO";
    public static final String TYPE = "TOK_TYPE";
    public static final String VAR = "TOK_VAR";
    public static final String WHILE = "TOK_WHILE";
    public static final String WRITE = "TOK_WRITE";
}
```

This interface holds the string code for the PCAT reserved words

TokenLiteral.java

```
public interface TokenLiteral {
    public static final String STRING_LITERAL = "TOK_STRLIT";
    public static final String INTEGER_LITERAL = "TOK_INTLIT";
    public static final String REAL_LITERAL = "TOK_REALLIT";
    public static final String IDENTIFIER_LIT = "TOK_IDENTIFIER";
}
```

This interface holds the string codes which identify type literals, such as integer, string and real.

3.1.2 Handlers

SymbolManager.java (too large to be included)

SymbolManager is, as it names describes, the class that creates the symbols based on the token and prints the correct output containing the line number and column number. It has three constructors, to be used depending on the type of lexeme it is parsing; for example for operators, the operatorHandler instance should be passed to return the correct key.

OperatorHandler.java (too large to be included)

The OperatorHandler class, implementing the Operators interface, provides an Operator map which holds the token name and token value for each operator (required by the project specification). It contains an Operator class as well to be able to create operator instances and well-type the created tokens.

PCATLexer.java (too large to be included)

This class is the lexical analyser generated by the JFlex library based on the flex language specification defined in **pcat.flex**.

3.1.3 Core Files

Main.java

```
public class Main {

    public static void main(String argv[]) {
        if (argv.length == 0) {
            System.out.println("Usage : java PCATLexer [ --encoding <name> ] <
        }
        else {
            int firstFilePos = 0;
            String encodingName = "UTF-8";
            if (argv[0].equals("--encoding")) {
                firstFilePos = 2;
                encodingName = argv[1];
                try {
                    java.nio.charset.Charset.forName(encodingName); // Side-e
                } catch (Exception e) {
                    System.out.println("Invalid encoding '" + encodingName + "
                    return;
                }
            }
            for (int i = firstFilePos; i < argv.length; i++) {
                PCATLexer scanner = null;
                try {
                    java.io.FileInputStream stream = new java.io.FileInputStre
                    java.io.Reader reader = new java.io.InputStreamReader(stre
                    scanner = new PCATLexer(reader);
                    while ( !scanner.getEOF() ) scanner.yylex();
                }
                catch (java.io.FileNotFoundException e) {
                    System.out.println("File not found : \""+argv[i]+"\"");
                }
                catch (java.io.IOException e) {
                    System.out.println("IO error scanning file \""+argv[i]+"\
                    System.out.println(e);
                }
                catch (Exception e) {
                    System.out.println("Unexpected exception:");
                    e.printStackTrace();
                }
            }
        }
```

```
    }
}
```

The main class that runs the compiler. It handles file encoding in case the user wants to change from UTF-8 and accepts paths-to-files as arguments to run the lexical analyser on The main functions throws the appropriate functions in case the file path is not provided, or the encoding is invalid or an error occurs.

3.2 Errors Handled

The four major error types specified by the project requirements are handled by this compiler:

1. Dangling comment error; This error is breaking i.e. it stops execution
2. Unrecognized symbol error; This error is printed but does not break execution
3. Unterminated string error; This error is printed but does not break execution
4. Invalid identifier name error; This error is printed but does not break execution

3.3 Requirements

3.3.1 Completed

1. Implemented **PCAT** language specification using flex
2. Implemented interfaces to hold the token values
3. Implemented SymbolTable using a data class with the proper functions to add, get and print elements
4. Handled required errors

3.3.2 Missing

- Symbol table is not pre-loaded with the keywords before parsing the provided pcat file.

Part III

Personal Comments

3.4 Program Specification

3.4.1 Pre-Requisites

The makefile assumes that the Java executable is in your "PATH" and accessible through the Command Line. In addition, it assumes JFlex is present on the system and has an alias pointing to the JFlex executable binary as well. If the above is not present, the code below can give pointers on how to implement them:

Add Java to the PATH

Add the below line to your shell configuration file

```
export PATH="\$PATH":/path/to/your/jdk/bin
```

Add the alias to the shell

Run this line in your terminal

```
ln -s /path/to/jflex /usr/bin/jflex
```

3.4.2 How-To-Run

The Java files which create the Compiler should be compiler using the *Makefile* provided in the tarball.

```
JFLAGS = -cp .
JC = javac
JAVA = java

CLASSES =    \
        Delimiters.java \
        Operators.java \
        OperatorHandler.java \
        ReservedWords.java \
        TokenLiteral.java \
        SymbolManager.java \
        SymbolTable.java \
        PCATLexer.java \
        Main.java

all: build

build:
        jflex pcat.flex
```

```
        $(JC) $(JFLAGS) $(CLASSES)

default: all

clean:
        $(RM) *.class PCATLexer.java PCATLexer.java~

test:
        $(JAVA) Main testfiles/test-scanner-01.pcat
        $(JAVA) Main testfiles/test-scanner-02.pcat
        $(JAVA) Main testfiles/test-scanner-03.pcat
        $(JAVA) Main testfiles/test-scanner-04.pcat
        $(JAVA) Main testfiles/test-scanner-05.pcat
```

- Run **make** in your terminal to create the lexer and compile the java files
- Run **make test** to run the Main class on 5 test files provided in the tarball
- Run **make clean** to remove the generated lexer and class files after you're done

Part IV

Conclusion