Derick Dao - 17726966

ProgrammingLanguages

Assignment 1 - Report

Contents

[Introduction 2](#_Toc463340811)

[Lexer 2](#_Toc463340812)

[Parser 2](#_Toc463340813)

[Assumptions 3](#_Toc463340814)

[Makefile 4](#_Toc463340815)

[Testing 4](#_Toc463340816)

[Conclusion 5](#_Toc463340817)

# Introduction

This is a report that contains the description of how I built the parser and syntax checker for my assignment this year. It will contain information about the design decisions I made to design the program and necessary explanations for various parts of the code that might seem complex or that might be misunderstood as something else.

# Lexer

The Lexer (check.l) that I have created consists of all the tokens that are used in the EBNF graph. I have chosen to split up “FOR EACH”, “END DO”, “END WHILE”, “END FOR” and “END IF” so that the parser won’t need to pick up extra whitespaces and not have to skip over words to throw errors due to finding a single word token that isn’t defined. Rather, I split those words up so that the parser is able to pick up all the words without having to skip to the next word to know that both words are related to each other; it allows for efficiency for the parsing and for the simplicity of designing the program itself.

My Lexer (check.l) has two print statements TOKF and TOKV which both take in the token that was scanned in. The difference between the two functions is that one prints out the full word being passed through (TOKV) where the other prints a standard word that has been identified for the pre-defined tokens (TOKF) such as ARRAY, BEGIN or DECLARATION. Numbers consist of numbers from 0-9 inclusive where any of those numbers can appear more than once. A number is to be of 1 or more digits. An identifier is a lowercase alphabetical sequence where an alphabet can appear more than once. An identifier must be one letter or more. I have implemented a case where the Lexer skips over empty spaces and/or otherwise, if it doesn’t detect any of the previously defined tokens, will print an error message for an unexpected character and the program will exit.

Each of the functions TOKF and TOKV in my Lexer file returns a token with a “T\_” prefix which will dictate the various tokens picked up by the parser.

# Parser

My parser (check.y) imitates the EBNF diagram that was given in the appendix of the assignment specifications. I have a pre-defined function called NONTERMINAL which prints out what the parser finds and the symbol it picks up. All the tokens that are picked up are all prefixed with “T\_” to indicate the various identifiers that the parser can pick up.

The body of the parser consists of the translation of the EBNF graphs provided along with the assignment with some few adjustments for the efficiency of programming. I have included a few more declarations so that looping is able to be done in the program. I have included the various extra declarations below:

* O\_const\_declaration
  + Allows for optional Constant declarations done in the declaration\_unit
* O\_var\_declaration
  + Allows for optional Variable declarations done in the declaration\_unit
* O\_type\_declaration
  + Allows for optional Type declarations done in the declaration\_unit
* O\_procedure\_interface
  + Allows for optional Procedure Interface done in the declaration\_unit
* O\_function\_interface
  + Allows for optional Function Interfaces done in the declaration\_unit
* O\_formal\_parameters
  + Allows for optional parameters to be included in procedure\_interface and function\_interface
* O\_parameters
  + Enables for multiple identifiers separated my semicolons to be evident in parameters
* Comma\_const\_declarations
  + Creates the body of the Constant declarations and allows for loops of const\_var\_declarations separated by a comma
* Const\_var\_declarations
  + Identifies a variable declaration to be an identifier with and equals sign and a number.
  + This is used to loop with comma\_const\_declarations for multiple constant declarations.
* Comma\_variable\_declaration
  + Creates the body for Variable declarations and allows to loop var\_declaration separated by a comma
* Var\_declaration
  + Declaration of a variable with an identifier, a colon and another identifier. This is used with comma\_variable declaration for multiple variable declarations.
* Comma\_ident
  + Used for looping an identifier separated by a comma to be declared for an enumerated\_type
* Compound\_sequence
  + Body of a compound statement which consists of one or more statements separated by a semicolon.
* A\_s\_term
  + Used to add and subtract one or more terms.
* M\_d\_idnum
  + Used to multiply and divide one or more terms.

The parser (check.y) has a main which reads in a command line argument and opens up the file provided on the command line for reading. After reading the file, the lexer will analyse the contents of the file and will parse through tokens from the file to the parser which outputs on the screen whether a token is acceptable for syntax checking. It will throw a yyerror() if the parser comes by an unexpected error and prints out a message.

# Assumptions

I assumed the keywords provided will be uppercase letters which makes things convenient when dealing with keywords and it will not overlap with lowercased identifiers. I also assumed that the ‘:=’ and the ‘..’ (assign and range) symbol will not contain spaces which I found a need to define them as a token themselves.

Looking at the EBNF graph as explained before, I found that separating the words such as FOR EACH and END WHILE was a more efficient way to read in the files to the parser so that the program doesn’t need to be doing anything more than read in one token at a time and not having to read the next token to see if both relate to each other.

# Makefile

The Makefile consists of 4 command line prompts to create the program PL2016\_check. The Parser had the flags -v, -d and -o. ‘-d’ allowed for the generation of y.tab.h, ‘-o’ allowed for an output file of what the parser has parsed through and ‘-v’ allowed for the description and report on conflicts generated in the EBNF grammar.

The clean method got rid of all ‘.c’, ‘.h’, ‘.o’, ‘.output’, and the PL2016\_check executable for recompilation.

# Testing

When testing the test files (.txt) supplied for testing the program, there were a few errors within the test files that restricted the program to pick up the actual mistake that was implemented in them. The Declaration-\*.txt files, all TYPE declarations didn’t end with a semicolon which stopped my program from detecting the error in Declaration-Invalid3.txt which was the uppercase letter for the function declaration. Although it didn’t affect the first two test files, it stopped the third test case from detecting the actual error in the file.

The following errors were found in each of the following test cases:

Declaration-Invalid1.txt

* CONST was declared after a VAR declaration which does not comply with the supplied EBNF graphs. Also evident was the semicolon was non-existent after the TYPE declaration which also doesn’t follow the EBNF diagrams

Declaration-Invalid2.txt

* Next to the identifier ‘darp’ should be a semicolon instead of a colon which was picked up by my program and again, the non-evident semicolon next to the TYPE declaration.

Declaration-Invalid3.txt

* The TYPE declaration didn’t have a semicolon to end the statement which stopped my program from finding the actual error with was the uppercase ‘M’ in the function declaration. I inserted a semicolon next to the TYPE declaration and ran my program and it picked up the uppercase letter in the function declaration.

Declaration-Valid1.txt

* The TYPE declaration did not have a semicolon which make a valid test case invalid. I inserted a semicolon next to the TYPE declaration and ran it with the program. The program threw no faults or error messages and parsed the file correctly.

When testing the Implementation-\*.txt files, some of the test files had a DO loop without a WHILE clause which caused the program to fault and throw an error which was not intended.

The following errors were found in each of the following test cases:

Implementation-Invalid4.txt

* The error was one of the BEGIN … END statements not ending with an ‘END’ but this error was not picked up due to the DO statement that was nested in not having a WHILE clause which did not comply with the EBNF graph supplied for the assignment. I added in a WHILE clause for testing purposes and my program picked up that one of the BEGIN and END statements did not have an ‘END’.

Implementation-Invalid5.txt

* The error evident was the ending semicolon for the IMPLEMENTATION unit, where it is supposed to be a full stop instead. Again, this was stopped due to the DO loop which was nested in not consisting of a WHILE loop. This didn’t reflect the EBNF graph given for the assignment. I added in a WHILE clause for the DO loop and my program picked up the semicolon error at the end instead of a full stop.

Implementation-Invalid6.txt

* The error in this test file is the END FOR ending clause for a WHILE construct. This error was not picked up by my program due to the DO Loop. I added in a WHILE clause for the test file and tested with my program and my program picked up the wrong END FOR closing statement for a WHILE loop, which was supposed to be END WHILE instead.

Implementation-Invalid7.txt

* The error is the semicolon next to the CALL statement and this did not follow the EBNF graph supplied. The DO loop without having a WHILE clause stopped this error from being picked up. I added in a WHILE clause and tested with my program. My program picked up the semicolon that was evident for the CALL statement.

Implementation-Invalid8.txt

* The error is having CONST term = 2; on a separate line where it is supposed to be linked with the line above it. This is an error because after having all the specifications of a block, the implementation is after which is a compound statement (I.e. consists of BEGIN and END) but the structure starts with CONST which throws a fault when scanning through. This error was picked up by my program

Implementation-Valid2.txt

* This test file was supposed to be valid but because the DO loop had no WHILE clause, the test file became invalid. I added in the WHILE clause for the DO loop and the program parsed through the file successfully.

Implementation-Valid3.txt

* This test file was tested and it was correctly parsed through.

The files supplied were semi-inefficient due the underlying faults that it had but I had inserted various alternatives into those test files so that each could pick up its errors.

# Conclusion

In conclusion, I hope this report outlines the various differences I have made in relation to the supplied EBNF graph and hopefully it shows the design intentions that I have made for the program to work the way it does. This assignment was quite interesting in that it allowed for a new exploration of a platform that I haven’t ever experienced.