

|  |
| --- |
| **CMP409 – Languages and Compilers Report**  Aaron Stones  BSc Computing with Honours, 2020  Word Count: 11,216 |

|  |
| --- |
| School of Design and Informatics  Abertay University |

Contents

[Summary 3](#_Toc40808262)

[Parsing 3](#_Toc40808263)

[Semantic Analysis 3](#_Toc40808264)

[Confidence 3](#_Toc40808265)

[Testing 4](#_Toc40808266)

[Explanation 5](#_Toc40808267)

[References 6](#_Toc40808268)

# Summary

To begin with, an extensive search of the current solutions to the project proposed by Doctor Sampson was done. This consisted of a google search where a GitHub repository was uncovered that contained (what is assumed) a previous student’s work into the matter.

On further analysis of the code they had produced there were flaws and bottlenecks within the code as well as unnecessary features that could be taken out. The structure however, made sense to develop the project to match this as it followed an effective structure in terms of splitting up the source files into three distinct groups;

1. token handler, parser, semantic analysis and scanner
2. a sub directory containing the error class and defines how errors are managed and displayed to the user
3. code generation where orbitasm is utilised.

On upon communication with Doctor Sampson, it was made abundantly clear that any code that was a direct copy and not a new idea would not gain any marks. Any code that has been utilised from the original project has been marked accordingly.

## Parsing

There are generally two types of parser a top-down parser and a bottom-up parser, for this particular project a top-down recursive descent parser has been selected. With the wide array of grammar that the PAL language supports, it is key that the parser selected can cope with this factor. Therefore, a recursive descent parser is ideal to complete this project.

The parser has been split into sections; the first section is utilities for managing the different functionalities that has been set out in the user specified code. The second is used to manage the flow of the program and ensure there is a methodical operation to the project. The third provides extra functionality for the second section to ensure code repetition is kept to a minimum.

## Semantic Analysis

This section of the compiler has been setup with two main goals in mind, firstly to check the validity of a statement made within the user’s program and secondly to report any issues from the analyser to the error subdirectory to be logged and displayed post compilation.

## Testing

With the code that was found online, there was an extensive testing kit provided by the previous class, equipped with testing files with; valid compilations, syntactically incorrect programs and semantically incorrect programs. These have been utilised, however the task set out specified to create your own testing suite. With this in mind a further testing suite has been developed to provide double the coverage and reduce the chances of errors being missed during the development and testing phases.

# Testing

|  |  |  |  |
| --- | --- | --- | --- |
| Test ID | Purpose | Test Data | Result |
| Test Suite located online | | | |
| T001 | To test the syntactical analysing capabilities of the PAL compiler | located in the /testFiles/syntax directory | Results came out as expected and errors that the project developer can see have been identified and reported |
| T002 | To test the semantic analysis capabilities of the PAL compiler | located in the /testFiles/semantic directory | Results came out not as the developer expected, however there were a couple that were missed in assigning expressions |
| T003 | To test the semantic analysis capabilities of the PAL compiler | located in the /testFiles/valid directory | All tests passed and no errors were reported as expected |
| Test suite developed by the project developer | | | |
| T004 | To test the syntactical analysing capabilities of the PAL compiler | located in the /testFiles/Mysyntax directory | Results came out as expected and errors that the project developer can see have been identified and reported |
| T005 | To test the semantic analysis capabilities of the PAL compiler | located in the /testFiles/Mysemantic directory | Results came out not as the developer expected, however there were a couple that were missed in assigning expressions |
| T006 | To test the semantic analysis capabilities of the PAL compiler | located in the /testFiles/Myvalid directory | All tests passed and no errors were reported as expected |

As previously mentioned, the use of two testing suites has allowed for greater analysis of the compiler code as to when it is operational and functioning optimally and when it is not. There have been a few errors slipping through the cracks in the semantic analysis that could easily be ironed out in future versions of the project. These errors stem from being mis represented as the wrong type of error. An experienced programmer would be able to understand there is a mistake and be able to work around it, however inexperienced programmers would find this a difficult challenge to overcome.

# Explanation

As previously alluded to in the main directory of the source files is where the classes; parseCheck, scanFile and semanticCheck are located. This explanation will look at these classes predominantly, but where needed further classes will be brought into the discussion to give a better understanding or more context to the explanation.

## Input Parsing

Parsing of the file is carried out at all points during the compilation of the code provided by the user. From the entry file used to get the entry file used to get correct input from the user and until the compilation is finished. Once the compiler has decided that the user has entered a correct file, invoke() is called by the compiler, where the scanner is instructed to collect the first token from the source code. From here the compiler’s parser uses its utilities section to assure the first token present is ‘PROGRAM’ as this is what must always come first in a PAL program.

The compiler does this by – calling mustBe() which checks if the current token matches the token provided by the program, if not it sends the compiler into rec mode and consumes tokens until it gets to the correct one.

if(scanner\_.currentToken()->is(type)) {

scanner\_.nextToken();

}

else {

syntaxError(type);

}

This process is repeated with the next token needing to be an identifier and then next must be ‘WITH’, until we come to the first variable that is to be declared. The next step is to gather an identifier name from the user’s code, there is a possibility to initialise multiple amounts of variables in PAL just like C++ or C etc. so this has to be taken into account. For this a vector is setup to contain all of the variable names a program will store (the remaining code for this is conducted by the semantic analyser). However, only real and integers can utilise the multiple declaration functionality so this needs to be checked. If any other variable type has been used then a syntax error has been conducted.

if(have("INTEGER")) {mustBe("INTEGER");type = PALType::Integer;}

else if(have("REAL")) {mustBe("REAL"); type = PALType::Real;}

else {

syntaxError("type name");

if(have(Token::Identifier)) {

scanner\_.nextToken();

}

}

The parser needs to then decide whether what the program does next, this can either be a conditional statement, an INPUT/OUTPUT statement or another identifier (in which case the previous code is executed). If the compiler detects none of these a syntax error is logged. We are then in the loop of these two code blocks until there is a syntax error or there is an END token specified by the user, in which the compiler must assure that the correct token was EOF and if this is true then no errors have been occurred and the compiler outputs nothing.

Errors overall are returned to the entry file through the error vector – runs synonymously throughout the compiler. Where if the compiler has found errors they are sorted into ascending order and if not false is returned to the entry file.

sort(err.begin(), err.end(), [](const rec<reportErr>& a, const rec<reportErr>& b) {

return \*a < \*b;

});

return err.size() == 0;

## Input Scanning

The starting code to this was provided by Dr Sampson in C#, however, C++ is a very powerful and similar language that is extremely fast within the command line so I have decided to use this.

Since the scanner is called by the parser throughout the program I will reference where it is called and what the scanner will do.

The only function that is called by the parser is getNToken, this function is called when the program begins and when we are assuring the tokens the program is expecting is matching the tokens received by the scanner from the program file. The compiler lexes the next token provided by the source file and returns it to the parser for comparison. The tokens that can be received are compared using a case statement and then are either passed to the errorToken vector or are used to set the state of the compiler. An example of this is -

case compState::inte:

switch(intCheck(Character())){

case 1:

state = compState::real;

break;

case 2:

state = compState::inte;

break;

case 3:

errorToken = std::make\_shared<lexToke>(lexToke::inte, buffer.str(), Line, Column);

break;

}

break;

The other states the scanner can be in is – whitespace, inte, real, punc, ident, eof or inv.

The scanner utilises the getCharacter function also that is used to read the data in from the source files.

## Semantic Analysis

As previously stated, semantics has either one of two purposes, to log the errors or to detect the errors from a user. There are many errors that have been identify during the course of this module. The main ones that are most pertinent to semantics are – semantic error, assignment error, expression error, variable error, redeclaration error and Boolean error.

Each function sets the message of an error if one has been detected and adds it to the error vector for showing to the user at the end of the program’s compilation. An example of this code is –

bool exists = varExists(x); //true or false if variable already exists

if (exists == false){

vars[x->getContains()] = {def, x}; //define variable

}

else { //varibale already defined

std::string message = "Already defined variable - " + x->getContains();

logVar(x, declVar(x), message);

}

bool PALSemantics::variableExists(RC<Token> var) {

return variables\_.find(var->value()) != variables\_.end();

}

These are how most of the errors are defined and stored within the error vector.

# References

<https://github.com/amyinorbit/palc>

<https://www.geeksforgeeks.org/recursive-descent-parser/>

<https://en.wikipedia.org/wiki/Recursive_descent_parser>

<https://www.booleanworld.com/building-recursive-descent-parsers-definitive-guide/>

<https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm>

Allan Milne's C# implementation PALScanner.cs.