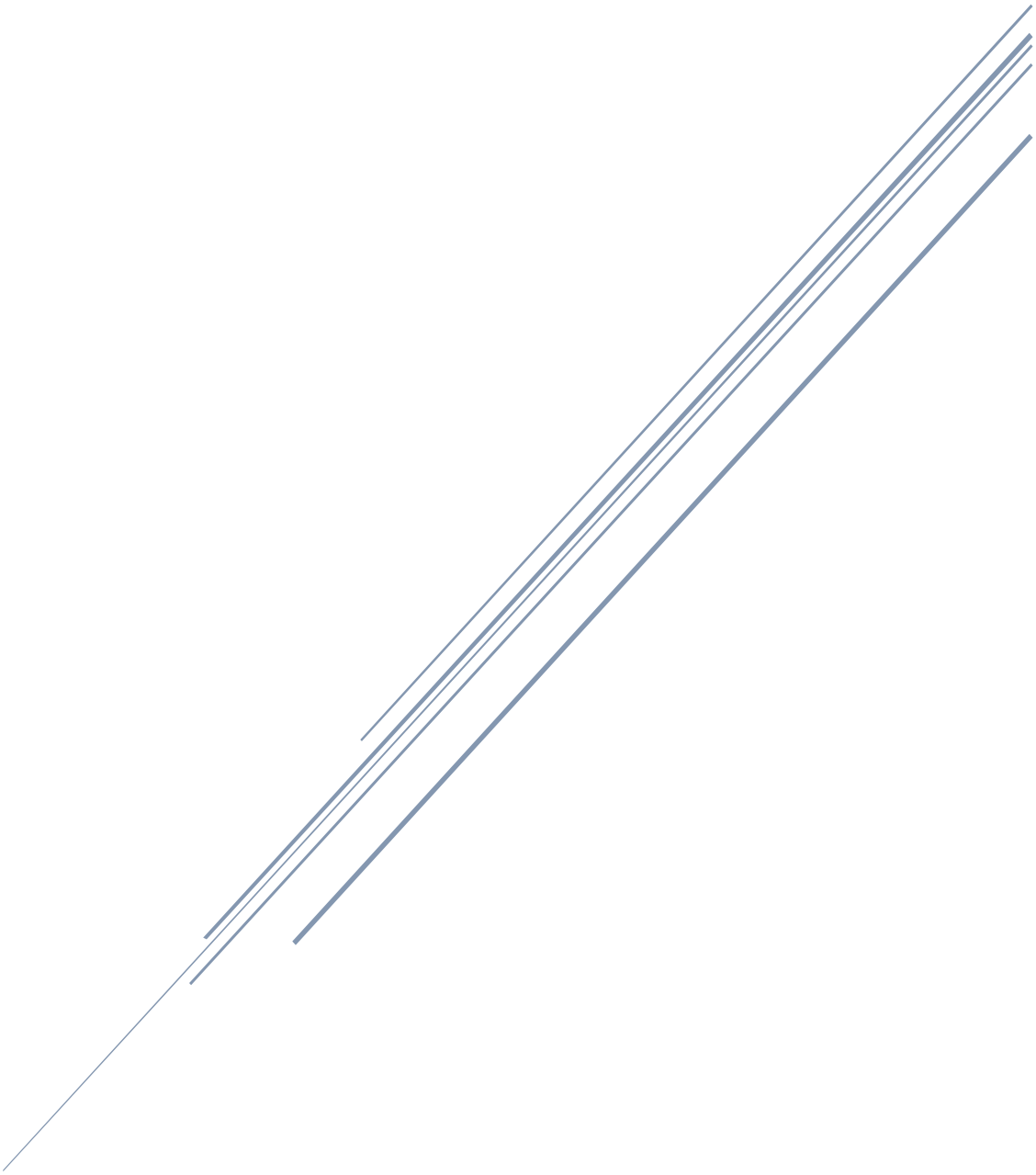
LANGUAGES AND COMPILERS

University of Abertay Dundee



Cullen Willis
1505633

Introduction

A compiler is a program that was made to process statements written in programming language and turns the code into machine language that the processor can understand and use. The process of converting programming language into machine language is known as compilation, hence the name compiler (Technopedia, unknown).

Behind every compiler there are four major steps: scanning, lexical analysis, syntactic analysis, and semantic analysis. These four steps help build the compiler and allows the processor the compute the code into instructors for the computer to do.

Methodology

This section will outline the necessary functions within the parser and semantic analyser to explain in detail what the compiler does and how it operates.

Parser

When implementing the parser, the provided text document: EBNF.txt was followed, along with some example programs from the practical tasks. The EBNF text document explains what functions you should create, and how each functions links with each other. It also provides the basic language structure which is the first step, writing some test data first will give the developer a good grasp on how to program the parser to detect what is wrong, and what is right. Overall the text document states that there should be 11 main functions that will allow the parser to identify the language.

Program

To start the parser the compiler will check from the necessary words which will call a function, this will ensure the program is properly syntaxed. The words it checks for are:

- PROGRAM
    - When calling PROGRAM, the next char char will be an identifier token to specify the name of the program.
- WITH
    - The WITH section of the language will declare variables, thus calling recVariableDeclares()
- IN
    - The IN section of the language will prompt statements, thus calling recStatementList() which will loop the statements until told to stop.
- END
    - Identifies the end of the program and calls the function to identify a end of file token.

Variable Declares

This function allows the compiler to read the language and declare the necessary variables, within this function there are 4 parts: loop, identifier token, mustBe(), recType() function call. Using these four parts the function will be able to identify any number of variable specified in the language.

Identifier List

The purpose of this function is to store a list of tokens which will be declared at the end of Variable Declares, storing the tokens inside a list will ensure each token gets declared inside the semantic analyser, thus preventing errors.

Type

The recType function is quite simple, the purpose of this function is to identify whether the type is an integer or a real number, once its identified the type the function will return the appropriate LanguageType (either integer or real) and declare the token within the semantic analyser.

Statement

This function identifies the statement inside the language, once identified the correct statement it will call another function. Inside the statement there are five possibilities:

- Identifier token
    - This will call the assignment statement, which will assign a new value to a variable
- UNTIL
    - This is a loop condition which will keep iterating through until it is told to stop
- IF condition
    - A basic if condition, compares two variables
- INPUT & OUTPUT
    - This function will input from the console and output to the console
- (
    - This will call recFactor() to determine + or –

Assignment

The purpose of the assignment function is to assign a new value to a variable, for instance INT = 5. During the function it will identify any tokens and check the current ID, it will also call recExpression(). Finally, the function will check the assignment with the semantics, this will ensure the variable can assign the value.

UNTIL

This is a simple function which will recognise the basic foundations of a UNTIL statement which are: UNTIL, REPEAT, and ENDLOOP. Once it's identified the loop it will call recBooleanExpression() which will identify the current expression being handled. Finally, it will recall recStatement() to get any additional statements throughout the UNTIL loop

Conditional

The condition function is similar compared to the UNTIL statement as it will identify the necessary words: IF, THEN, ELSE, and ENDIF. Once it's identified the words it will call recBooleanExpression() and recStatement() like the UNTIL function. If the word ELSE is specified in the language then it will call recStatementList() which will simply loop the conditions until it's finished.

Boolean Expression

As stated above the Boolean expression function will compare the expressions and identify the correct operators, for instance 'INT > 5'. Throughout the function it will gather the expressions and check the Boolean with the semantics which is described in detail during the Semantic – Boolean Checks chapter.

Input / Output

The input / output function will identify the words INPUT and OPUT and call the necessary function once identified. If INPUT is recognised it will call the identifier list which will gather the necessary tokens inside the INPUT statement. When OUTPUT is found the function will check the expressions with recExpression() to ensure the expression exists.

Expression

When calling the expression function it will check for + and -, whilst checking for these words it will gather the current term by calling recTerm() to check the expression with the semantics which is detailed in the Semantic – Expression Checks chapter.

Term

Similar to the expression sub-chapter the recTerm() function will check for * and /, whilst during these checks it will gather the recFactor (+ or -) and check the expression with the semantics, which is further explained in the semantics sub-section called Expression Checks.

Factor

As specified above the recFactor() function will check for + and -, once found it will check for an '(' in the language, if found it will call recExpression() to gather the type which it will return.

Value

The recValue() function is a simple function that checks if the language is an identifier token, integer token, or an real token. Once identified it will return the type of the token. If no token was found it will return an undefined token.

Semantics

The main purpose of the semantic analyser during the implementation of the PAL compiler is to check and declare the token ID's. During the process of the semantic analyser the functions will store semantic errors - if any errors do occur the compiler will print the error to console.

Declare Function

Before the semantic analyser declares the token, it will check if the token is valid and check if the token is defined in the scope of the compiler. If the token is not found or invalid it will return a semantic error, otherwise the semantic analyser will add the token and the current type in integer format to the scope called symbols (symbol table).

Check Function

When checking the token, the semantic analyser will first check if the token is defined in the current scope of the compiler, if the token is not defined it will show an error and return undefined. Otherwise, the check function will return an int generated in a new function called checkType.

CheckType Function

The checkType function returns an int into the check function once called. The purpose of this function is to check the token against each of the four types of tokens: Undefine, Identifier, Integer, Real. Once checking the current token against each type, if the type is confirmed it will return which type of token it should be, otherwise, it will return an error.

To further develop the semantic analyser the use of the 'C parser' from the GGC github repository came in good use. In their parser file they specified functions that deal with Boolean checks, expression checks, and assignment checks.

Boolean Checks

The Boolean check is a simple function which checks to ensure the left type and the right type are defined, if they aren't defined they'll be replaced by the corresponding type. The last check ensures the left type doesn't equals the right type, otherwise it'll report a semantic error.

Expression Checks

The expression check does the same techniques as the Boolean check. The function will ensure each type is defined, if not it will replace with the corresponding type – and report a semantic error if necessary.

Assignment Checks

The assignment check was written to ensure the current token, left type and the right type are undefined and the variables match. If this function was not implemented a program will be able to assign a INTEGER variable with a REAL variable, which we don't want.

Testing

To test the compiler there were three test cases, two test cases that were intentionally correct, and one that was wrong.

Correct Testcases

```
PROGRAM testcase
        WITH
                number AS INTEGER
        IN
                number = 0
                IF number = 0 THEN OUTPUT 1 ENDIF
END
```

```
PROGRAM testcase
        WITH
                number AS INTEGER
            tree AS INTEGER
        IN
                number = 1
                tree = 2
                IF tree > number THEN OUTPUT 1 ENDIF
END
```

Incorrect testcase

```
PROGRAM testcase
        WITH
                number AS INTEGER
            tree AS REAL
        IN
                number = 1
                tree = 2
                IF (tree * 2) > number THEN OUTPUT 1 ENDIF
END
```

During the test of the incorrect testcase, the intentional error was in the IF statement. The error is when the REAL variable (tree) is compared to and INTEGER variable (number). In theory, if this was written in any language this would appear as an error.

The purpose of this testcase it to test the semantic analysis, more specifically the expression check assignment. When the testcase is ran an error occurred, this shows that the semantic analysis is working as expected.