Program Description

Introduction

The purpose of this assignment is to use the ANTLR parser generator to write a lexer and a parser for a programming language named CCAL.

The grammar written for the language defines how the lexer and parser work. It is defined in a file named **ccal.g4**. The grammar contains definitions for fragments, tokens and parser rules which are necessary for ANTLR to automatically generate a lexer and a parser.

The Lexer

The lexer is responsible for converting a stream of characters from the input file into a sequence of valid tokens in the language and prints an error message if a sequence of characters does not match any defined token or matches multiple token rules.

I wrote the token rules based on the keywords written in bold in the CCAL language specification. For example, the language contains keywords such as **return**, **var** and single character tokens such as **{** and **(**. Another resource that was useful was the ANTLR grammar specification for the Java programming language (link) which defines how tokens in the Java language are defined. Some of the token rules are defined directly in terms of string literals or regular expressions and others are composed of fragments which recognize part of a token. The format for each token definition is the token name on the left hand side of the rule and the expression used to recognize the token on the right hand side.

The following example code shows two token rules for the tokens NUMBER and IDENTIFIER:

```
fragment Letters: [a-zA-Z];
fragment LettersNumbersAndUnderScores: [a-zA-Z0-9_];

NUMBER: [0] | [1-9][0-9]*;
IDENTIFIER: Letters LettersNumbersAndUnderScores*;
```

The NUMBER token rule uses a regular expression to recognize valid integer numbers from the input stream and the IDENTIFIER rule uses two fragments, each of which are defined using regular expressions, to recognize function or variable names.
ANTLR uses these lexer rules to automatically generate a lexer which we can then use to recognize tokens in our language.

The Parser

Once the program has been converted into a sequence of tokens, the parser will use parsing rules to group sequences of tokens into valid CCAL sentences or throw an error if a sequence of tokens does not match any parsing rule.

I used the parsing rules defined in the CCAL language specification (ccal.pdf) to write the parsing rules in ANTLR and also drew inspiration from the ANTLR grammar for the Java programming language which defines the parsing rules for the Java programming language.

Since the syntax of ANTLR is different to the mathematical language used to define the grammar rules in the CCAL specification, I had to write some of the parsing rules in a form different to the grammar specification while still preserving the meaning and behaviour of each rule.

For example, declaration lists, ***decl_list,*** are defined as shown below in the grammar file:

$$decl\_list = (\langle decl \rangle ; \langle decl\_list \rangle \mid \epsilon)$$

As the epsilon symbol does not exist in ANTLR, I had to define this rule in a different way. I found that defining this rule and many others using regular expressions was the best approach.

Although the rule in the specification says that *decl_list* should match one or more declarations, in my grammar I found that matching zero or more declarations was necessary for correctly parsing empty functions which only contained return statements. I also made the same change for the *statement* rule.

A *decl_list* which matches zero or more declarations, *decl,* can be defined in ANTLR using the following regular expression:

```
decl_list: decl*;
```

Another significant problem was that one of the parsing rules in the grammar specification contained indirect left recursion which cannot be handled by ANTLR. Left recursion exists between the following two rules but only indirectly when the two rules interact:

```
expression:
    item binary_arith_op item
    | '(' expression ')'
    | IDENTIFIER '(' arg_list ')'
    | item
    ;

item: IDENTIFIER | '-' identifier_or_number | NUMBER | TRUE | FALSE |
expression;
```

The problem can be traced to the first line of the expression rule which can be expanded left or right. If it is expanded to the left as the comment below shows, indirect left recursion will occur:

```
/*
expression: item binary_arith_op item
expression: (expression) binary_arith_op item
expression: (item binary_arith_op item) binary_arith_op item
expression: (expression binary_arith_op item) binary_arith_op item
*/
```

The solution to the problem is to rewrite the grammar using direct left recursion, which ANTLR can handle, or right recursion. I choose to make the expression rule right-recursive by rewriting it in the following way:

```

expression:
    item binary_arith_op right_item
    | '(' expression ')'
    | IDENTIFIER '(' arg_list ')'
    | item
    ;
```

```
item: IDENTIFIER | '-' identifier_or_number | NUMBER | TRUE | FALSE;

right_item: item | expression;
```

Using these updated parsing rules, the first clause of the expression rule can only be expanded with right recursion which eliminates the possibility of indirect left recursion.

The Java Program

Running **antlr ccal.g4** automatically generates the lexer and the parser from the grammar definition. However, we still need a way of reading in CCAL source files as input for the lexer and parser. We can do this by writing a Java program for this purpose named **ccal.java**. The program takes a .ccl file as a command line argument and passes the input characters to a lexer which generates tokens. The parser then uses these tokens to generate a parse tree which ANTLR can display visually if the -gui flag is set.