Program Description

I will go through each of the main components and briefly describe how I implemented them.

Main Method

The main method of my CCAL Parser is essentially identical to the main method of the SLP Parser found in the JavaCC notes. It begins by initialising the parser to read from the correct place; it will read from standard input or it will read in a file using the path specified as an argument to the program.

Once it has initialised the parser, it will attempt to begin parsing the input file using the parsers Prog() method which is defined in the grammar section. If the file parsers correctly it will print a success message, otherwise it will print the ParseException error.

Lexical Analyser

Nested Comments

Nested comments are handled the same way as in the JavaCC notes. A commentNesting variable is incremented when `\*` is encountered and the lexical state is altered. While in this new state, the same variable is incremented each time `\*` is encountered and is decremented when `*/` is encountered. All other tokens are skipped. Each time commentNesting is decremented, we check to see if it is equal to 0 and if it is we switch back to the default lexical state.

Single-line Comments

The token for single-line comments is defined by `< "//" (~["\n"])* "\n" >` which will match `//` followed by anything other than a newline character up until a newline character is encountered.

Syntax Analyser

The grammar was initially defined by mechanically working through the grammar specification with terminals being represented by tokens and non-terminals being represented by method calls.

Left-Recursion

Once I finished the process of mechanically writing out the grammar, I tried to compile the `.jj` file and I was presented with two left-recursion errors. The first of which was an issue with my expression and fragment non-terminals.

My code initially looked like this:

```
void Expr() : {}
{
    (Frag() BinOp() Frag())
  | (<LPAREN> Expr() <RPAREN>)
  | (<ID> <LPAREN> ArgList() <RPAREN>)
  | Frag()
}

void Frag() : {}
{
    <ID>
  | (<MINUS> <ID>)
  | <NUMBER>
  | <TRUE>
  | <FALSE>
  | Expr()
}
```

This is a left-recursive grammar because `Expr()... -> Frag()... -> Expr()...`. To fix this I had to remove `Expr()` from the left hand side of the production rule in `Frag()`. My workaround resulted in the following code:

```
void Expr() : {}
{
    (Frag() Term())
  | (<LPAREN> Expr() <RPAREN> Term())
}

void Term() : {}
{
    BinOp() Expr()
  | {}
}

void Frag() : {}
{
    (<ID> [<LPAREN> ArgList() <RPAREN>])
  | (<MINUS> <ID>)
  | <NUMBER>
  | <TRUE>
  | <FALSE>
}
```

The grammar states that fragment and expression can be used interchangably. I removed `Expr()` from

fragment and appended an optional `BinOp() Expr()` to each production in `Expr()` using my newly created `Term()` non-terminal. I took a similar approach to resolving the left-recursion in my conditional non-terminal.

Choice Conflicts

Once I resolved the two left-recursion errors, my parser compiled with a number of choice conflict warnings. Most of these were similar and were relatively straightforward to resolve.

```
void Stm() : {}
{
    (<ID> <ASSIGN> Expr() <SEMIC>)
  | (<ID> <LPAREN> ArgList() <RPAREN> <SEMIC>)
  | ...
}
```

I fixed the above choice conflict involving the `<ID>` in `Stm()` by introducing a new non-terminal `StmPrime()`.

```
void Stm() : {}
{
    (<ID> StmPrime())
  | ...
}

void StmPrime() : {}
{
    (<ASSIGN> Expr() <SEMIC>)
  | (<LPAREN> ArgList() <RPAREN> <SEMIC>)
}
```

I was unable to remove one of the choice conflicts as it involved production rules from two non-terminals (`Cond()` and `Expr()`). I used a LOOKAHEAD of three to resolve this conflict. This is the only instance of LOOKAHEAD I used.