CA4003 ASSIGNMENT PROGRAM DESCPRITION

Name: Alex Randles

Student no: 15302766

The Javacc code is split into 4 main sections which are the options, user code, tokens and the grammar of the language.

Section 1: Options

The options include different settings the parser will use, the most important part of the options for this assignment is the Ignore case option shown below, this tells the parser to ignore the case of the language. Since the CAL Language is not case sensitive this prevents the parser from throwing up errors. Another option which came in useful when dealing with errors was the debug parser option which would help to show where the code is failing to parse correctly.

```
/**********OPTIONS**************/
options {
    JAVA_UNICODE_ESCAPE = true;
    IGNORE_CASE = true;

}
```

Section 2: User code

The user code is the part which initializes the parser and runs it against the file which is past as an argument or the standard input it receives. I have got this code from the notes.

I've modified it accordingly to be a CAL Parser and have removed the section which prints the tokens out as this is not required for this assignment. The CALParser is initialized with the name parser then the input feed to it will be parsed when the parser.program() function is called. This function is defined in the grammar section and starts the parsing process.

Once this function is called the input will be parsed and if its parsed correctly "CAL Parser: CAL program parsed successfully." will be printed to the screen. If the input is not parsed correctly a Parse Exception will be printed along with

"CAL Parser: Encountered errors during parse.". The parser is then stopped with the PARSER_END function.

SECTION 3: Tokens

The tokens consist of regular expressions that hopefully match the input stream of the parser. The token manager creates a Token object for each match of such a regular expression and returns it to the parser.

TOKEN_MGR_DECLS

The first part of the token section is the TOKEN_MGR_DECLS, there can be only one of these in the file. This is where we declare variable and methods that are accessible from all lexical actions. The variable commentNesting is placed in here to deal with nested comments within our input. This is the same way in which nested comments are handled in the notes. This variable is first given the value of 0.

SKIP

The next part of the token section consists of the skip, these are simply matched and ignored by the token manager.

The first skip section matches spaces, tabs and newlines and simply ignores them.

The next skip section matches a nested comment. The commentNesting variable declared in the TOKEN_MGR_DECLR is incremented when a * is next in the input stream. The lexical analyser state is then changed to the <IN_COMMENT>. In this new state the commentNesting variable is incremented each time a * is passed to it and when a */ is passed to it, the commentNesting variable is decremented. All other tokens between these are skipped by using **<~[]>** and once the commentNesting variable is 0, it switches back to the DEFAULT lexical state. This is the same way in which nested comments are dealt with in the notes.

The next skip section deals with non-nested comments. These are defined as beginning with // and is delimited by the end of line. **< "//" (~["\n"])* "\n" >** is used to deal with non-nested comments. This regular expression begins with // and then matches any number of characters until the newline character is reached.

TOKENS

The next part of the code following Skip is the Token. The tokens are several regular expressions that if matched will generate a token object which is then returned to the parser. The order in which tokens are is important as when two regular expressions that both match the longest match, the first one wins.

The first token section consists of the reserved words of the CAL language. These are defined in the language specification which was given to us. These are non-terminals used to match the reserved words.

The next token section are the keywords and punctuation of the language, these were all in the language specification and gave them names based on the symbol meaning.

The next token section was a bit tricky to figure out. It was the integers and identifiers of the language. The integers are represented by **< INTEGERS : "0"| ((<MINUS_SIGN>)? <NEGATIVE_DIGITS> (<DIGITS>)*)> .** This regular expression matches the string “0” or a string beginning with one or none “-“ followed by the digits (“1”-“9”) and zero or more (“0”-“9”). The identifier are represented by **< IDENTIFIER : <LETTER> (<LETTER> | <DIGITS> | "_")* >.** This regular expression matches a string beginning with a letter, following by zero or more letters, digits or underscores. The * is used to denoted zero or more. Identifiers cannot be reserved words, therefore I defined the reserved word tokens before the identifier tokens.

The final token section **< OTHER : ~[] >** is used to match anything that is not yet recognised.

GRAMMAR

The parser reads the grammar specification and converts it to a Java program that can recognize matches to the grammar. The grammar ensures the tokens passed to the parser are syntactically correct.

I started the grammar by copying the language specified grammar given to us. Terminals are represented by tokens and non-terminals represented by function calls. After I had worked my way through the language specified grammar in the assignment specification, I ran the code and several parsing errors presented themselves.

The first errors which came to my attention were left-recursive errors. I noticed indirect left recursion in my expression() non-terminal. The reason for this expression() calls fragment() which then in turns calls expression(), thus creating an infinite loop of these non-terminals calling each other. I fixed this by adding an expression_recursion() non-terminal which prevented this from happening. I also removed expression() from fragment() to ensure left recursion would not happen

Code before left recursion fix:

```
void expression(): {}
{
    fragment() binary_arith_op() fragment()
|   <LBR> expression() <RBR>
|   <ID> <LBR> arg_list() ><RBR>
|   fragment()

}
```

Code after left recursion fix:

```
void expression()  : {}
{
  (fragment() expression_recursion())
| (<LBR> expression()  <RBR> expression_recursion())
}

void expression_recursion() : {}
{
  binary_arith_op() expression()
| {}
}
```

After I had fixed the left recursion in expression() I then had to fix the direct left recursion in the condition() non-terminal. I did this by creating a condition_recursion() non-terminal. The recursion problem arises when the line of code condition() (| | &) condition() is reached as condition will always be called.

Code before left recursion fix:

```
void condtion() : {}
{
   <NEGATION> condition()
|  condition()
|  expression() comp_op() expression()
|  condition() <OR> <OR> <AND> condition()

}
```

Code after left recursion fix:

```
void condition() : {}
{
  (<NEGATION> condition() condition_recursion())
| LOOKAHEAD(3) (<LBR> condition() <RBR> condition_recursion())
| (expression()  comp_op() expression()  condition_recursion())
}

void condition_recursion() : {}
{
  <OR> condition()
| <AND> condition()
| {}
}
```

After I fixed the left recursion errors, I then complied my code and was presented with choice conflict's.

The first choice conflict I fixed was with then nemp_parameter_list() non-terminal. The choice conflict arose with the **idenitifer:<type>| identifier:<type>,<nemp_parameter_list>.** I fixed this by adding a nemp_parameter_list_choice() non terminal shown below.

```
void nemp_parameter_list_choice()  : {}
{
  [<COMMA> nemp_parameter_list() ]
}
```

The next choice conflict I found was in the statement() non-terminal. The problem was the choice between identifier in line one over identifier in line 2. I added a new non-terminal called statement_choice(). This fixed the conflict error for the statement() non terminal.

Code before conflict error:

```
void statement()  : {}
{
  (<IDENTIFIER> <ASSIGNMENT> expression() <SEMI_COLON>)
| (<IDENTIFIER> <LBR> arg_list() <RBR> <SEMI_COLON>)
| (<BEGIN> statement_block()  <BEGIN>)
| (<IF> condition() <BEGIN> statement_block()  <END> <ELSE> <BEGIN> statement_block()  <END>)
| (<WHILE> condition() <BEGIN> statement_block()  <END>)
| (<SKP> <SEMI_COLON>)
}
```

Code after conflict error fixed:

```
void statement()  : {}
{
  (<IDENTIFIER> statement_choice())
| (<BEGIN> statement_block()  <BEGIN>)
| (<IF> condition() <BEGIN> statement_block()  <END> <ELSE> <BEGIN> statement_block()  <END>)
| (<WHILE> condition() <BEGIN> statement_block()  <END>)
| (<SKP> <SEMI_COLON>)
}

void statement_choice()  : {}
{
  <ASSIGNMENT> expression()  <SEMI_COLON>
| <LBR> arg_list() <RBR> <SEMI_COLON>
}
```

The last choice conflict was between the non-terminals condition() and expression(). I solved this by using a lookahead of 3, this means the parser looks ahead 3 tokens before deciding at a choice point.

The lookahead is shown below:

```
void condition() : {}
{
  (<NEGATION> condition() condition_recursion())
| LOOKAHEAD(3) (<LBR> condition() <RBR> condition_recursion())
| (expression()  comp_op() expression()  condition_recursion())
}
```