CS4201 Practical 1

160001826

5th October 2019

1 Introduction

This assignment requires the modification of the grammar for Oreo to be LL(1), and the implementation of a recursive descent parser to generate parse trees of valid Oreo, or produce meaningful error messages.

2 Usage

To lex and syntax analyse a source file of Oreo, and print the parse tree to stdout run:

```
cd src
python parser.py <SOURCE FILE PATH>
```

Note that the generated ASCII parse trees are often very wide, so if your terminal doesn't support scrolling sideways then you may prefer to redirect the output to a file and open it in a text editor.

```
cd src
python parser.py <SOURCE FILE PATH> > output
atom output
```

You can specify a custom grammar file with

```
cd src
python parser.py <SOURCE FILE PATH> <GRAMMAR DESCRIPTION FILE PATH>
```

3 Grammar

3.1 Approach

3.1.1 Initial Difficulties

The core of the practical was modifying the grammar to make it LL(1). My initial approach was to take the grammar as given and apply the algorithm for removing direct left recursion. However, the grammar produced had a lot of ambiguity remaining and I had particular difficulty resolving the indirect left recursion caused by the cycle between bool and expr and retaining LL(1) compliance.

3.1.2 Change of Strategy

Instead, I switched strategies, and implemented the bool and expr in a more granularised set of expansions which better captured the semantic value of the grammar, while still producing the same language. For example, the original grammar contains:

```
expr -> expr + expr
```

This is problematic as it leads to left recursion. My modified version contains

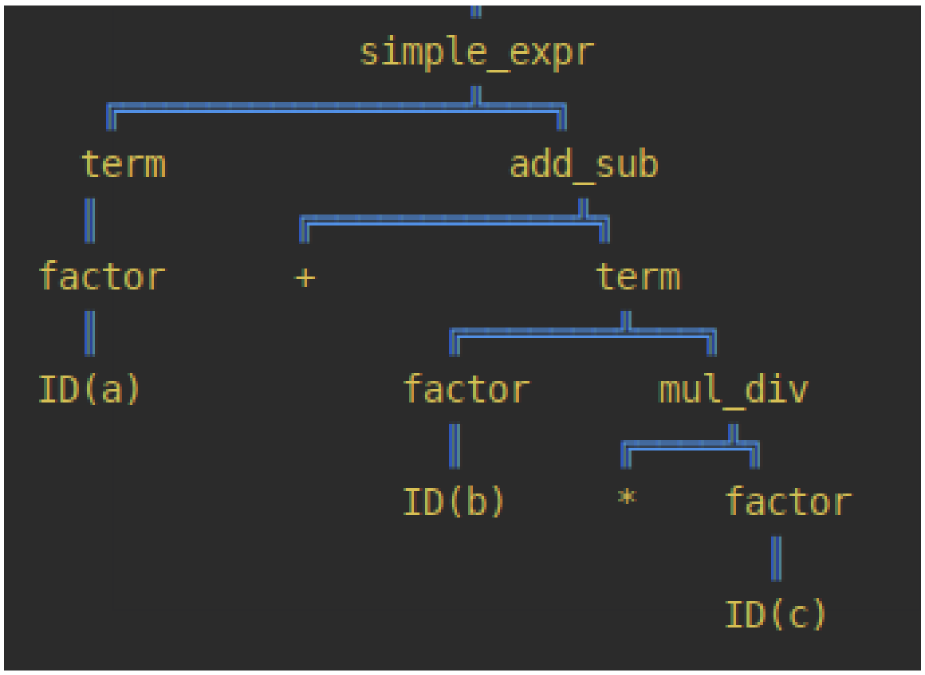
```
simple_expr -> term add_sub*
add_sub -> "+" term | "-" term
```

This more focused approach allowed designing around the LL(1) constraints (no left recursion, no ambiguity, one token of lookahead) from the ground up, rather than trying to refit the existing grammar to these constraints.

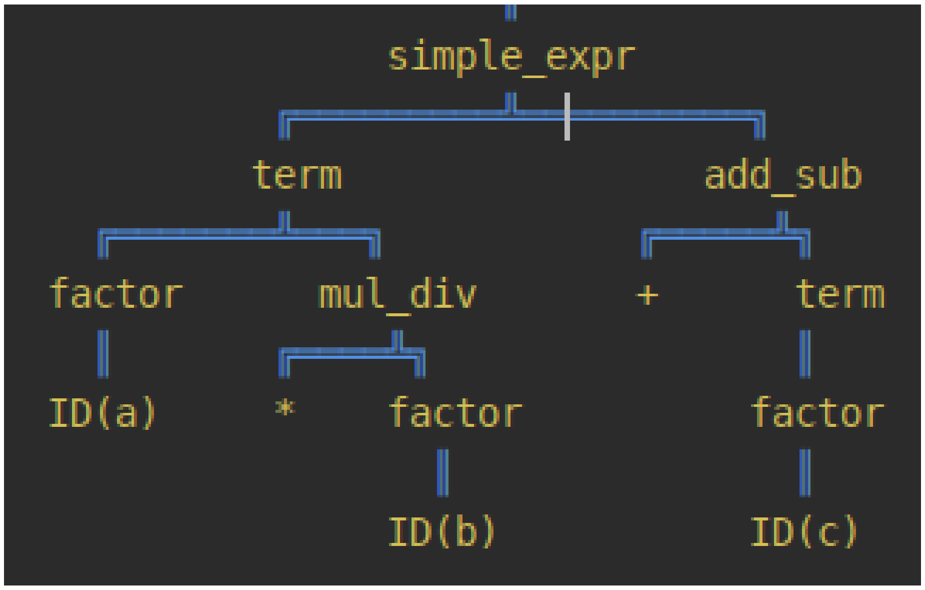
3.2 Terms and Factors for Order of Operations

The terms and factors approach, as discussed in lectures and in [1], was employed in the grammar, which captures directly in the parse tree the order of operations of addition, subtraction, multiplication and division, as well as the overriding of these rules by parentheses. My implementation draws on this terms and factors paradigm although the implementation details differ considerably - but the order of operations benefit still obtains. Note the differences in the parse tree generated for the following three expressions:

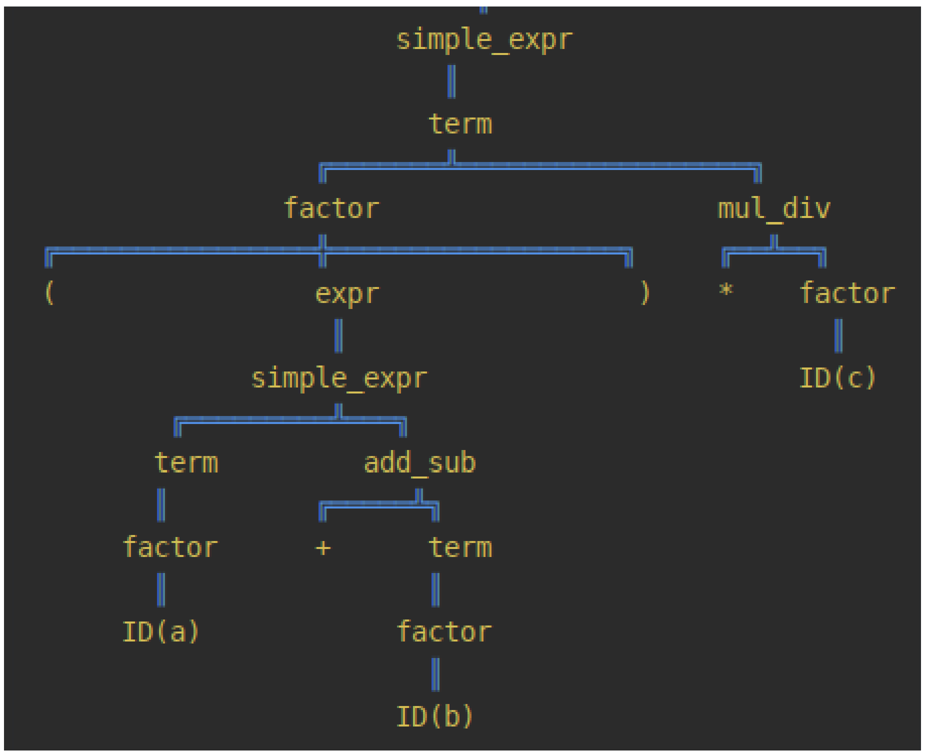
```
a + b * c;
```


simple_expr
term
add_sub
factor
+
term
ID(a)
factor
mul_div
ID(b)
*
factor
ID(c)

```
a * b + c;
```


simple_expr
term
add_sub
factor
mul_div
+
term
ID(a)
*
factor
factor
ID(b)
ID(c)

```
(a + b) * c;
```



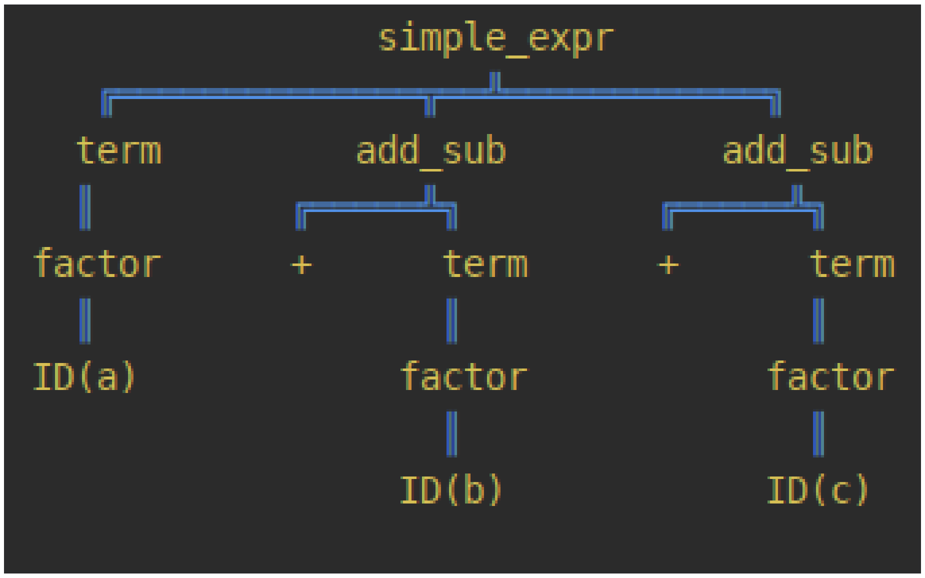
```
                           ╩
                      simple_expr
                         ║
                        term
                 ╔═══════╩══════════════════╗
               factor                     mul_div
╔════════════════╬══════════════════╗     ╔══╩══╗
(                expr                )    *    factor
                  ║                              ║
             simple_expr                       ID(c)
        ╔═════════╩═══╗
       term         add_sub
       ║         ╔═════╩╗
     factor      +      term
       ║                ║
     ID(a)            factor
                        ║
                      ID(b)
```

One place my implementation diverges from Wirth's notably is that I allow multiple addition/subtraction clauses through the use of a Kleene star:

```
simple_expr -> term add_sub*
```

This leads to tidy parse trees which represent the flat nature of multiple plain addition/subtraction operations in sequence. For example:

```
a + b + c
```



3.3 Addition of Functions

Function definitions and function calls were added to the grammar. To accurately recognise function definitions and calls, the addition of a new token was required `ID_PAREN` - an identifier immediately followed by an open parentheses, without whitespace in between. This allowed discrimination between function calls/definitions, and the use of parentheses in expressions, which was important for removing ambiguity, because an expression can legally contain an identifier, a function call, and a parenthesised sub-expression.

A `RETURN` token was also added, which forms part of the return statement:

```
return_statement -> "RETURN" optional_expr ";"
```

Return statements can appear within function bodies, but not outside. Similarly, function definitions can appear outside function bodies but not within (I decided that nested functions are not legal in Oreo).

```
statement -> v | pr | w | i | a | func_def | func_call
function_statement -> v | pr | w | i | a | return_statement | func_call
```

3.4 Addition of Comments

Adding comments required no change to the grammar, as these are recognised and discarded by the lexer. The regex is:

```
{-(?:[^-]|(?!))*-}
```

The full grammar can be found in Appendix A, and also in the `oreo.grammar` file.

4 Implementation

The implementation is in Python for clarity and conciseness, and because of its solid built-in regex support.

4.1 Lexer

The general approach of the lexer is to check for two kinds of matches: keywords, and special tokens.

4.1.1 Keyword Matching

Keywords come in two kinds: keywords that require a boundary afterwards (eg `program`, `or`) and those that don't (eg `;`, `<=`). Keyword matching simply involves checking if any keyword matches at the start of the remaining input stream.

4.1.2 Special Token Matching

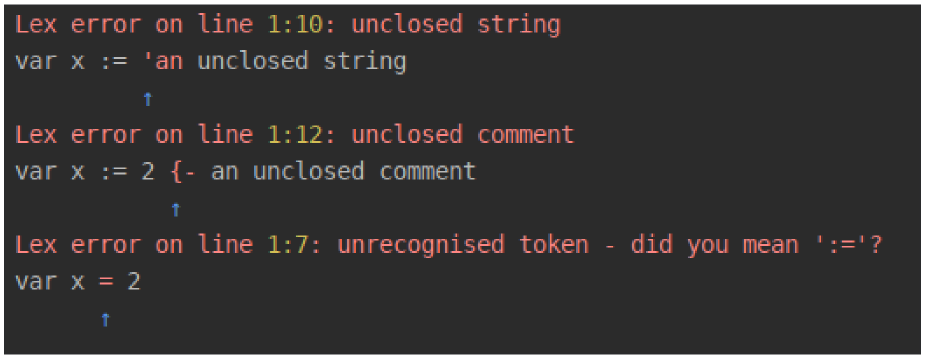
Special token matching is more involved - each special token has a name, regex, and optionally a group priority. If the regex matches, then an instance of the special token is created, and assigned its name.

The attribute is, by default, the entire matched string, but in some cases this is not desirable. For example, the entire matched string of `"hello"` is `"hello"`, but we actually just want `hello` to be stored as the token's attribute (so the token will be `name: STRING, attribute: hello`. To achieve this, the group_priority can be specified - this is a list of named capture groups, and the token's attribute is set to the first group in the list which matched a non-empty string.

This allows us to successfully extract the relevant content of strings, and if more complex tokens were added to Oreo in the future, this powerful approach would typically be able to handle them seamlessly.

4.1.3 Lex Errors

A focus was making lex errors as informative as possible. Three common lex error cases are identified by heuristics, which in almost all cases will immediately alert the user to the precise location and nature of their error, and clarify how it should be fixed.



```
Lex error on line 1:10: unclosed string
var x := 'an unclosed string
         ↑
Lex error on line 1:12: unclosed comment
var x := 2 {- an unclosed comment
           ↑
Lex error on line 1:7: unrecognised token - did you mean ':='?
var x = 2
      ↑
```

In cases where no match is found, and the next character in the input stream is a string opening quote, the first error in the screenshot above is given.

In cases where no match is found, and the next two characters in the input stream are the opening of a comment, the second error is given.

In other cases, fuzzy text matching techniques are used. Whichever keyword has the highest fuzzy match ratio (above a minimum of 50% to rule out improbable matches) is suggested as a replacement, as in the third error.

See Appendix B for a full list of tokens in Oreo.

4.2 Syntax Analyser

4.2.1 Ontology and General Implementation

The ontology of the syntax analyser consists in Terminals, NonTerminals and Expansions. Expansions are stored in a Dict mapping from the NonTerminal on the left hand side of the expansion, to an Expansion object. Expansions store the right hand side: either a list of Terminals and NonTerminals, or None in the case of an expansion to the empty string (ϵ).

The recursive descent parser is implemented in `ParseTreeNode.parse_tokens()`. The array of tokens generated by the lexer is compared to the expected tokens generated by expanding NonTerminals into subtrees with Terminals at the leaves.

The syntax analyser is agnostic to the actual grammar - the syntax analyser simply takes in a list of expansions which form the basis of its analysis. This felt a much more natural solution than directly implementing the Oreo grammar in Python. One concrete benefit is that the record of what the grammar is, and its implementation, are one and the same (the oreo.grammar file), so as modifications are made to one or the other, the two never become unfaithful representations of each other. This allowed me to empirically test ideas for how to modify the grammar rapidly with a low chance of mis-encoding my idea for the grammar when translating it into Python.

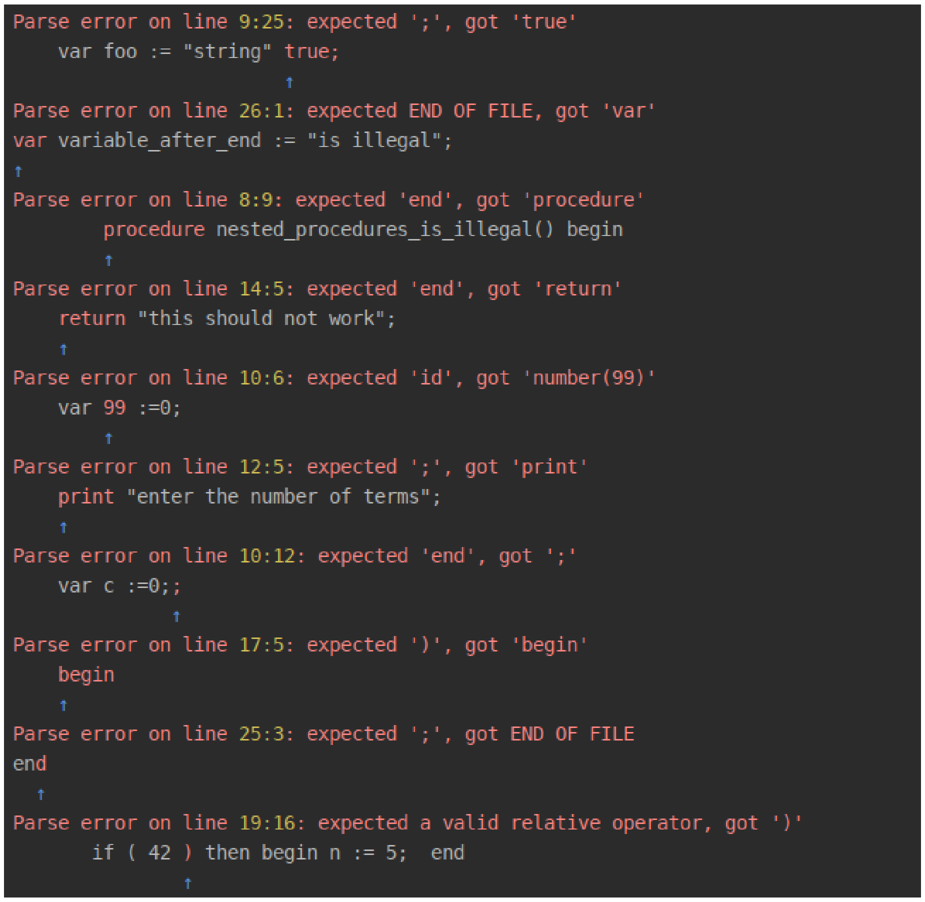
4.2.2 Kleene Stars

Kleene stars are implemented in the grammar parser and syntax analyser. If a NonTerminal has a Kleene star, when it is the next node to be expanded, if there are no valid expansions (the next token does not match any of the possible expansions' first tokens), then no parse error is thrown, and the node is quietly destroyed to keep the parse tree tidy. If a node with a Kleene starred NonTerminal is successfully expanded, a duplicate sibling is added to the tree, which also has a Kleene star, allowing optionally an arbitrary additional number of the same NonTerminal to be expanded.

The advantage of having Kleene stars is that the generated parse trees are flat as opposed to having a sloping form, which aids clarity and better communicates the semantic relationship between the sibling nodes.

4.2.3 Syntax Analyser Errors

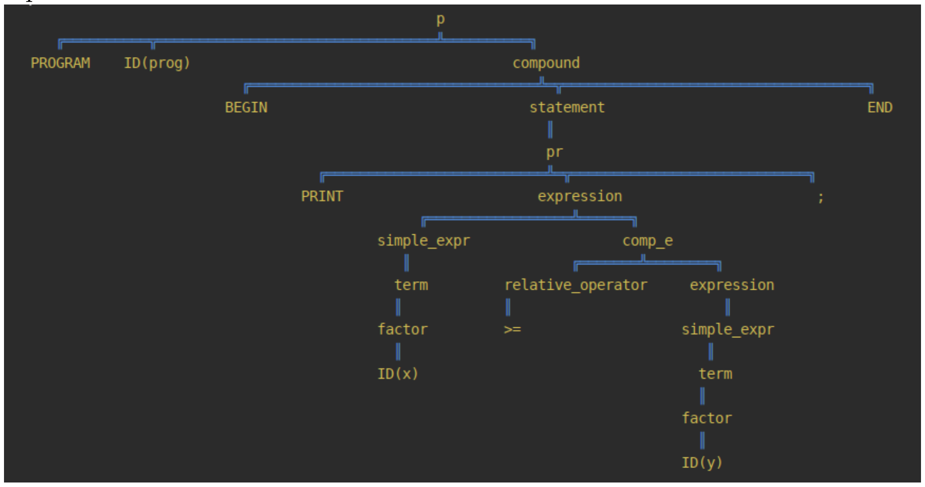
Descriptive and useful parse errors were a core focus. The ParseError Exception provides messages that describe the line and column number of the unexpected token, as well as printing the full line for context. The erroneous token is highlighted in red, and a little blue arrow points to it. Finally, the exception describes what kind of token was expected, and what was received instead. The exception is generic, and works for any combination of expected tokens/non-terminals and received tokens, so a full list of possible error messages - instead I have included a representative sample, generated from intentionally syntactically incorrect files in the data directory, below:



```
Parse error on line 9:25: expected ';', got 'true'
    var foo := "string" true;
                        ↑
Parse error on line 26:1: expected END OF FILE, got 'var'
var variable_after_end := "is illegal";
↑
Parse error on line 8:9: expected 'end', got 'procedure'
        procedure nested_procedures_is_illegal() begin
        ↑
Parse error on line 14:5: expected 'end', got 'return'
    return "this should not work";
    ↑
Parse error on line 10:6: expected 'id', got 'number(99)'
    var 99 :=0;
        ↑
Parse error on line 12:5: expected ';', got 'print'
    print "enter the number of terms";
    ↑
Parse error on line 10:12: expected 'end', got ';'
    var c :=0;;
              ↑
Parse error on line 17:5: expected ')', got 'begin'
    begin
    ↑
Parse error on line 25:3: expected ';', got END OF FILE
end
  ↑
Parse error on line 19:16: expected a valid relative operator, got ')'
       if ( 42 ) then begin n := 5;  end
               ↑
```

4.2.4 Parse Tree Printouts

I was pleased with my solution to pretty-printing the parse trees. See the example below:



4.3 Grammar Parser

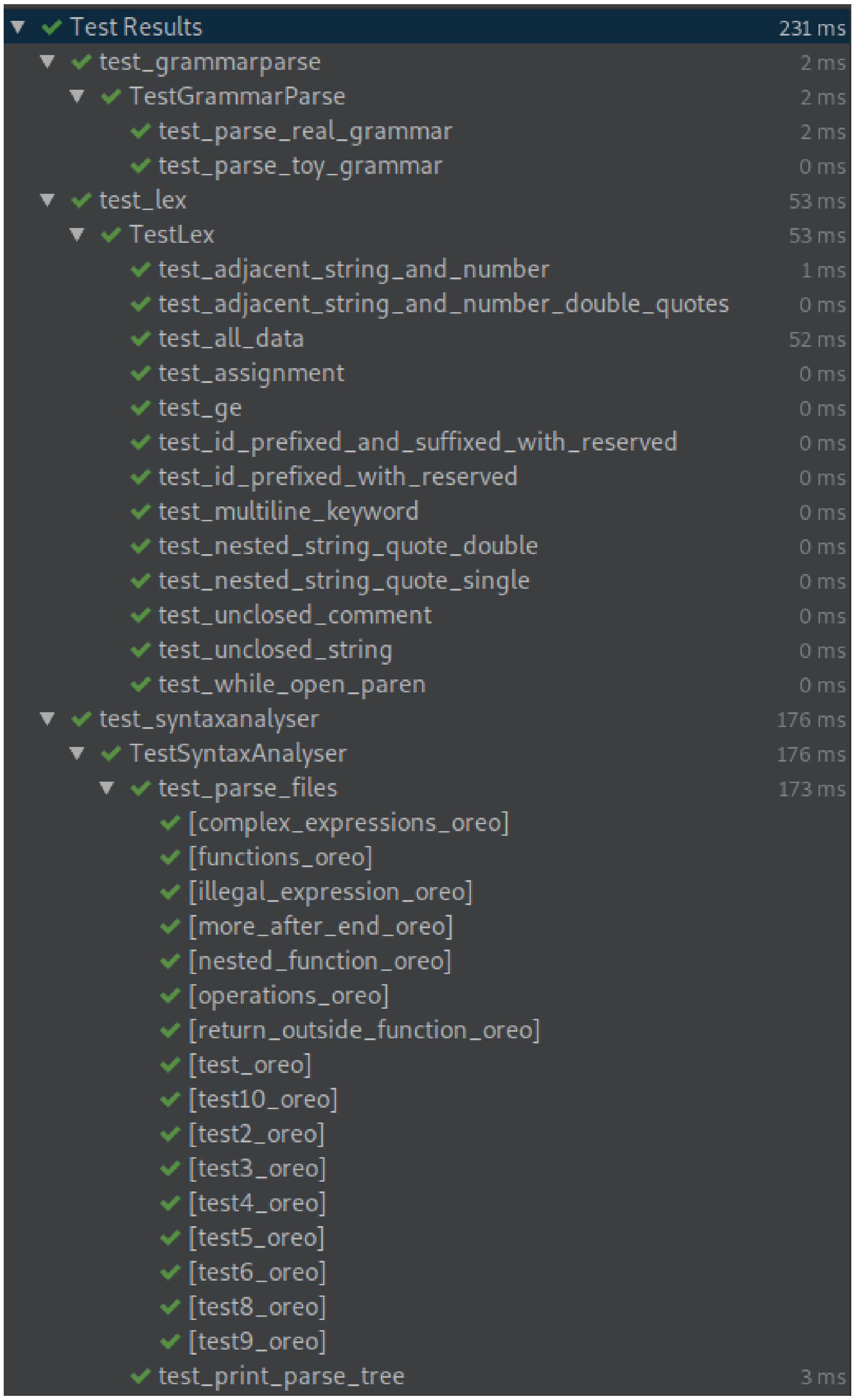
Parsing grammars is performed in grammarparse.py, and should be largely self describing. The dictionary of Expansions, as described above as an input to the syntax analyser, is generated by parsing a file of productions, of the form:

`nonterminal_a -> "TERMINAL_A" nonterminal_b` | `"TERMINAL_C"` | ϵ

5 Testing

Comprehensive unit tests can be found in the tests folder. Most crucially, the `test_syntaxanalyser.py test_parse_files` performs a subtest for each input files in the data directory. The input files are the provided test files, as well as some of my own with tricky cases and function syntax. For inputs where an Oreo syntax error is detected, a dictionary holds the precise line number where a ParseError must be thrown for the subtest to pass. For other inputs, no ParseErrors must be encountered for the test to pass. Visual inspection was used to ensure that the parse trees were correctly generated, as the complex trees are very cumbersome to verify by unit test.

All tests were passing:



```
▼ ✓ Test Results                                          231 ms
  ▼ ✓ test_grammarparse                                     2 ms
    ▼ ✓ TestGrammarParse                                    2 ms
        ✓ test_parse_real_grammar                           2 ms
        ✓ test_parse_toy_grammar                            0 ms
  ▼ ✓ test_lex                                             53 ms
    ▼ ✓ TestLex                                            53 ms
        ✓ test_adjacent_string_and_number                   1 ms
        ✓ test_adjacent_string_and_number_double_quotes     0 ms
        ✓ test_all_data                                    52 ms
        ✓ test_assignment                                   0 ms
        ✓ test_ge                                           0 ms
        ✓ test_id_prefixed_and_suffixed_with_reserved       0 ms
        ✓ test_id_prefixed_with_reserved                    0 ms
        ✓ test_multiline_keyword                            0 ms
        ✓ test_nested_string_quote_double                   0 ms
        ✓ test_nested_string_quote_single                   0 ms
        ✓ test_unclosed_comment                             0 ms
        ✓ test_unclosed_string                              0 ms
        ✓ test_while_open_paren                             0 ms
  ▼ ✓ test_syntaxanalyser                                 176 ms
    ▼ ✓ TestSyntaxAnalyser                                176 ms
      ▼ ✓ test_parse_files                                173 ms
          ✓ [complex_expressions_oreo]
          ✓ [functions_oreo]
          ✓ [illegal_expression_oreo]
          ✓ [more_after_end_oreo]
          ✓ [nested_function_oreo]
          ✓ [operations_oreo]
          ✓ [return_outside_function_oreo]
          ✓ [test_oreo]
          ✓ [test10_oreo]
          ✓ [test2_oreo]
          ✓ [test3_oreo]
          ✓ [test4_oreo]
          ✓ [test5_oreo]
          ✓ [test6_oreo]
          ✓ [test8_oreo]
          ✓ [test9_oreo]
        ✓ test_print_parse_tree                             3 ms
```

6 Conclusion

I am pleased with my results - I found modifying the grammar difficult but a rewarding and interesting challenge. Using Python felt very natural, particularly for things like the grammar parser, which ended up being very concise. I am hopeful that this will provide a good foundation for the second practical.

7 References

[1] Wirth, Niklaus. *Compiler construction. Vol. 1.* Reading: Addison-Wesley, 1996.

8 Appendices

8.1 Appendix A: Oreo Grammar

The following is verbatim the contents of oreo.grammar.

```
p -> "PROGRAM" "ID" compound
compound -> "BEGIN" statement statement* "END"
statement -> v | pr | w | i | a | function_definition | function_call
v -> "VAR" "ID" var_assign ";"
var_assign -> ":=" expression |
pr -> "PRINT" expression ";" | "PRINTLN" expression ";" | "GET" "ID" ";"
w -> "WHILE" "(" bool ")" compound ";"
i -> "IF" "(" bool ")" "THEN" compound optional_else ";"
optional_else -> "ELSE" compound |
a -> "ID" ":=" expression ";"
function_call -> "ID_PAREN" parameters ")" ";"
function_definition -> "PROCEDURE" "ID_PAREN" func_def_args ")" function_compound
func_def_args -> "VAR" "ID" later_func_def_arg |
later_func_def_arg -> "," "VAR" "ID" later_func_def_arg |
function_compound -> "BEGIN" function_statement function_statement* "END"
function_statement -> v | pr | w | i | a | return_statement | function_call
return_statement -> "RETURN" optional_expr ";"
optional_expr -> expression |
relative_operator -> "<" | ">" | "==" | ">=" | "<="
expression -> simple_expr comp_e and_or_b
comp_e -> relative_operator expression comp_e |
simple_expr -> term add_sub*
add_sub -> "+" term | "-" term
term -> factor mul_div*
mul_div -> "*" factor | "/" factor
factor -> "NUMBER" | "STRING" | "ID" | "ID_PAREN" parameters ")" | "TRUE" and_or_b
        | "FALSE" and_or_b | "(" expression ")" | "NOT" bool
parameters -> expression later_parameters |
later_parameters -> "," expression later_parameters |
bool -> "TRUE" and_or_b | "FALSE" and_or_b | "NOT" bool
        | simple_expr relative_operator expression
and_or_b -> "AND" bool | "OR" bool |
```

8.2 Appendix B: Tokens in Oreo

8.2.1 Special Tokens

- `NUMBER`: a sequence of digits.

  ```
  \d+
  ```

- `ID`: a valid identifier.

  ```
  [a-zA-Z]\w*
  ```

- `ID_PAREN`: a valid identifier followed by an open parentheses character.

  ```
  [a-zA-Z]\w*\(
  ```

- `STRING`: a quotes sequence of characters, optionally containing escaped quotes.

  ```
  (?:'(?P<single_quote>[^']*)')|(?:"(?P<double_quote>[^"]*)")
  ```

- `COMMENT`: a sequence of characters surrounded by {`-` and `-`}.

  ```
  {-(?:[^-]|(?!))*-}
  ```

8.2.2 Keywords

`PROGRAM, BEGIN, END, VAR, PRINT, PRINTLN, GET, WHILE, IF, THEN, ELSE, OR, AND, NOT, TRUE, FALSE, PROCEDURE, RETURN, ;, :=, +, -, *, /, (, ), <=, >=, ==, <, >, ","`