Compilers DAT4005

Compiler Design PRAC1

Kyle Cuthbert – st20249341

Table of Contents

Overview

This design document has been created to provide the plan and the steps that have been taken to create and design a compiler for a language of the author's choice. The language of choice is c# and it has been written in the python.

The document will go through each phase of the compiler showing what was planned and how it has been implemented, starting from the source 'file' c# program code, right through the creation of the pseudo assembly code.

To build the compiler basic operations have been provided that should be handled by the compiler that is built.

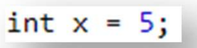
The basic operations that have been chosen in this language are listed below.

- Numerical variables
- Basic arithmetic (+, -, *, /)
- If, else if, and else.
- A loop
- Print text and numbers
- Comments

Syntax

Numerical variables - Syntax

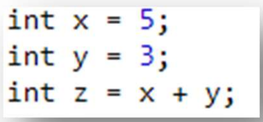
For number variables, the data type integer has been chosen to include in the compiler’s functionality.



```
int x = 5;
```

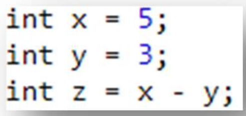
Basic arithmetic (+, -, *, /) - Syntax

Addition:



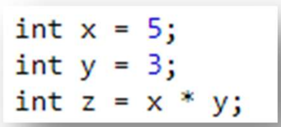
```
int x = 5;
int y = 3;
int z = x + y;
```

Subtraction:



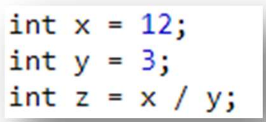
```
int x = 5;
int y = 3;
int z = x - y;
```

Multiplication:



```
int x = 5;
int y = 3;
int z = x * y;
```

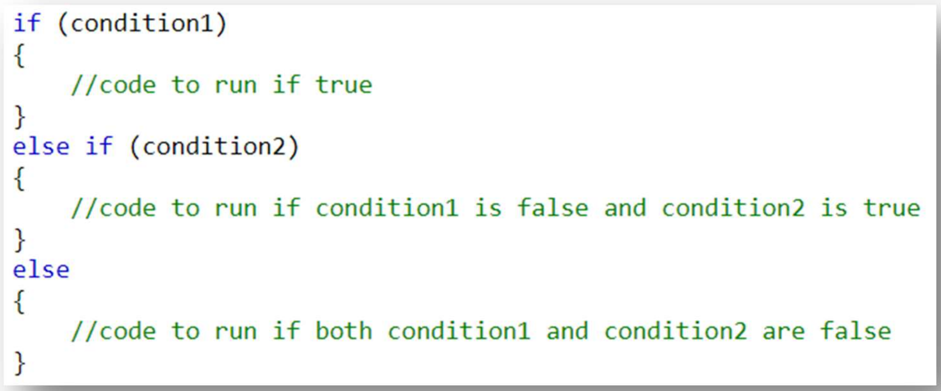
Division:



```
int x = 12;
int y = 3;
int z = x / y;
```

If, else if, and else - Syntax

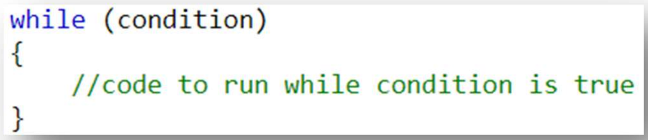
The syntax for the if, else if and else is as follows:



```
if (condition1)
{
    //code to run if true
}
else if (condition2)
{
    //code to run if condition1 is false and condition2 is true
}
else
{
    //code to run if both condition1 and condition2 are false
}
```

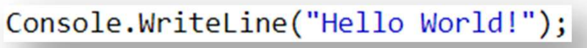
A loop - Syntax

The loop that has been chosen to include as part of the compiler is the ‘while loop’ and the syntax in C# is:



```
while (condition)
{
    //code to run while condition is true
}
```

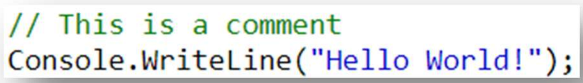
Print text and numbers - Syntax



```
Console.WriteLine("Hello World!");
```

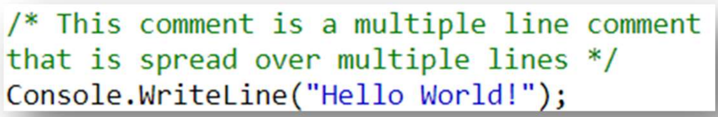
Comments - Syntax

One line comment:



```
// This is a comment
Console.WriteLine("Hello World!");
```

Multi-line comment:



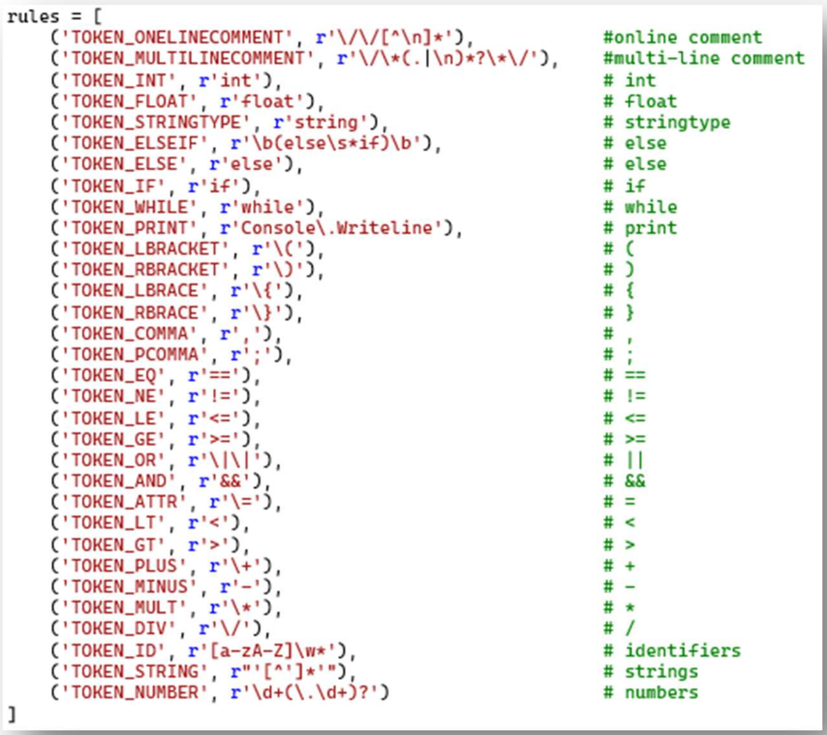
```
/* This comment is a multiple line comment
that is spread over multiple lines */
Console.WriteLine("Hello World!");
```

Lexical Analyser (Tokenizer)

A lexical analyser or the tokenizer as it is otherwise known is the first stage of a compiler and has the purpose of taking source code and extracting the tokens from the values included.

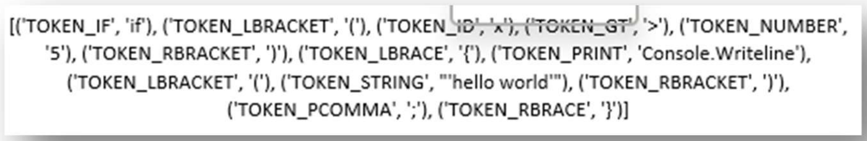
To identify the tokens string literals, operators, symbols, words, and regular expressions can be used. Techniques such as building NFAs and DFAS can be used to build regular expressions that can be used.

For the compiler that is being designed for this assessment, the lexical analyser will scan the input code and try to identify each token by taking the following rules to identify the tokens:



```
rules = [
    ('TOKEN_ONELINECOMMENT', r'\/\/[^\n]*'),                #online comment
    ('TOKEN_MULTILINECOMMENT', r'\/\*(.|\n)*?\*\/'),        #multi-line comment
    ('TOKEN_INT', r'int'),                                  # int
    ('TOKEN_FLOAT', r'float'),                              # float
    ('TOKEN_STRINGTYPE', r'string'),                        # stringtype
    ('TOKEN_ELSEIF', r'\b(else\s*if)\b'),                   # else
    ('TOKEN_ELSE', r'else'),                                # else
    ('TOKEN_IF', r'if'),                                    # if
    ('TOKEN_WHILE', r'while'),                              # while
    ('TOKEN_PRINT', r'Console\.Writeline'),                 # print
    ('TOKEN_LBRACKET', r'\('),                              # (
    ('TOKEN_RBRACKET', r'\)'),                              # )
    ('TOKEN_LBRACE', r'\{'),                                # {
    ('TOKEN_RBRACE', r'\}'),                                # }
    ('TOKEN_COMMA', r','),                                  # ,
    ('TOKEN_PCOMMA', r';'),                                 # ;
    ('TOKEN_EQ', r'=='),                                    # ==
    ('TOKEN_NE', r'!='),                                    # !=
    ('TOKEN_LE', r'<='),                                    # <=
    ('TOKEN_GE', r'>='),                                    # >=
    ('TOKEN_OR', r'\|\|'),                                  # ||
    ('TOKEN_AND', r'&&'),                                   # &&
    ('TOKEN_ATTR', r'\='),                                  # =
    ('TOKEN_LT', r'<'),                                     # <
    ('TOKEN_GT', r'>'),                                     # >
    ('TOKEN_PLUS', r'\+'),                                  # +
    ('TOKEN_MINUS', r'-'),                                  # -
    ('TOKEN_MULT', r'\*'),                                  # *
    ('TOKEN_DIV', r'\/'),                                   # /
    ('TOKEN_ID', r'[a-zA-Z]\w*'),                           # identifiers
    ('TOKEN_STRING', r"'[^']*'"),                           # strings
    ('TOKEN_NUMBER', r'\d+(\.\d+)?')                        # numbers
]
```

These tokens have been based on the syntax that was given in the previous section. Once the input code has been scanned an example of the returned tokens would be as per the below, these represent the token identified along with the value that was picked up by the rule above:



```
[('TOKEN_IF', 'if'), ('TOKEN_LBRACKET', '('), ('TOKEN_ID', 'x'), ('TOKEN_GT', '>'), ('TOKEN_NUMBER',
'5'), ('TOKEN_RBRACKET', ')'), ('TOKEN_LBRACE', '{'), ('TOKEN_PRINT', 'Console.Writeline'),
('TOKEN_LBRACKET', '('), ('TOKEN_STRING', "'hello world'"), ('TOKEN_RBRACKET', ')'),
('TOKEN_PCOMMA', ';'), ('TOKEN_RBRACE', '}')]
```

The input string for this example would be: **if(x>5){Console.Writeline('hello world');}**

To handle comment statements the lexical analyser has a method(strip_comment_tokens) to strip these from the tokens and can then pass a stripped_tokens to the parser.

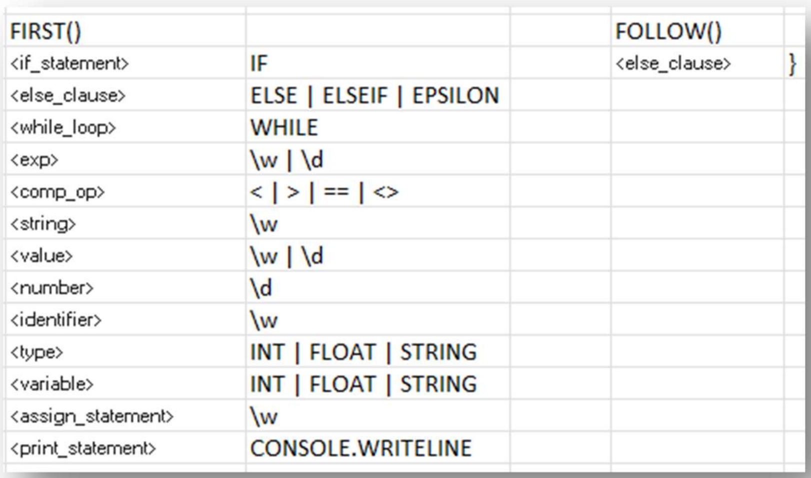
Parsing (Syntax Analyser)

The next stage of the compiler that is being designed will be the Syntactical Analyser which is also known as the Parser.

The purpose of this stage is to take the tokens that have been identified by the lexical analyser and ensure that the tokens, that represent the input string, adhere to the rules(syntax) that have been defined. If there are any sequences that do not abide by the accepted rules, then an error will be reported to the developer to inform them of a parsing error.

There are many different parsers that can be implemented but the parser that is being employed within this design is known as an LL(1) parser. This type of parser reads the tokens from left to right with left-most derivation and one lookahead symbol. The LL(1) parser can employ a parse table that can serve as the engine of the parser and allow the parser to get the next production rule which if identified as non-terminal will expand the rule out to the rules that make up the non-terminal rule. Meanwhile, terminals will be consumed by the parser if expected.

To assist with building the rules out of non-terminal and terminal production rules, the LL(1) parser employs a technique called the first and follow. Here the non-terminal rules will be listed under the 'first' and all the possible terminal options for the 'first' will be listed. In grammar the only function that we have that will have a follow will be the else clause as it can either be else, else if, or none. Please see the below which shows the first and follow for this design:

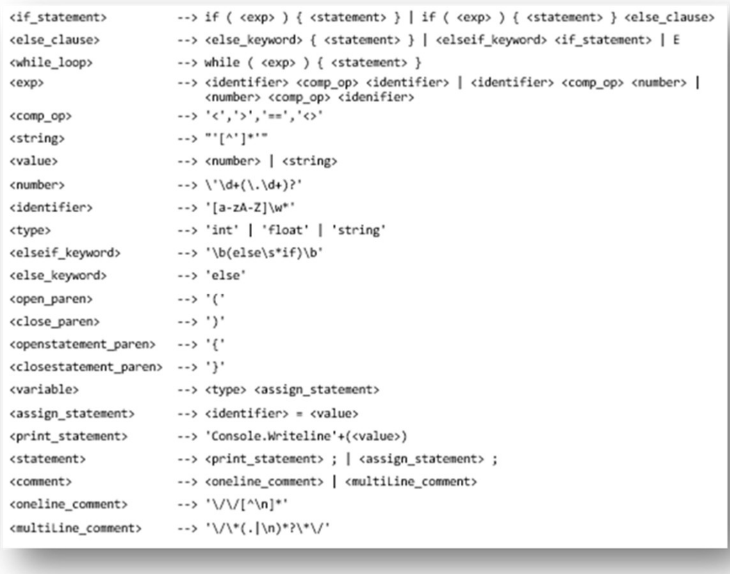


FIRST()			FOLLOW()	
<if_statement>	IF		<else_clause>	}
<else_clause>	ELSE | ELSEIF | EPSILON			
<while_loop>	WHILE			
<exp>	\w | \d			
<comp_op>	< | > | == | <>			
<string>	\w			
<value>	\w | \d			
<number>	\d			
<identifier>	\w			
<type>	INT | FLOAT | STRING			
<variable>	INT | FLOAT | STRING			
<assign_statement>	\w			
<print_statement>	CONSOLE.WRITELINE			

Previously, in the syntax section, the syntax for the c# functionality that is being targeted were laid out.

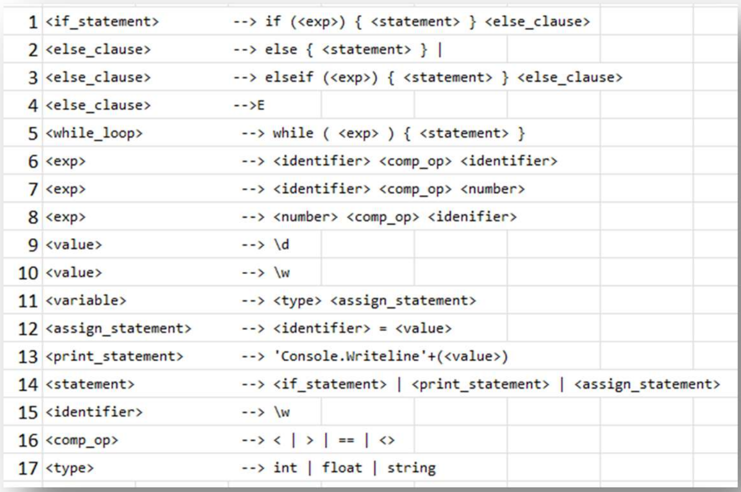
By using context-free grammar, the structure of the syntax can be easily designed so that the developer can identify valid sequences of the tokens that are accepted. Please note that some of these rules were re-named in the code during development.

The below represents the syntax of valid grammar or syntax that will be accepted by the parser.

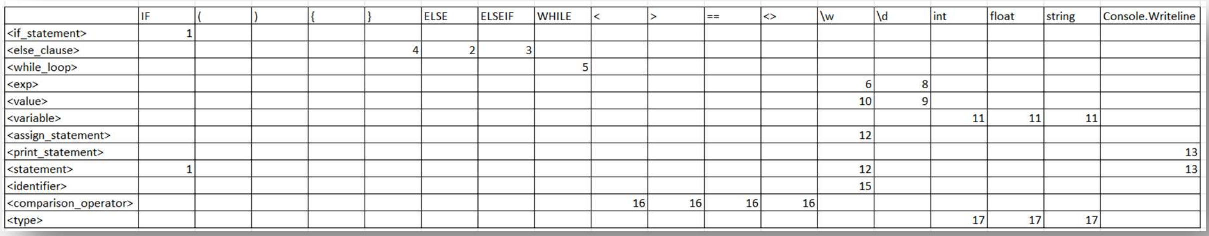


```
<if_statement>          --> if ( <exp> ) { <statement> } | if ( <exp> ) { <statement> } <else_clause>
<else_clause>           --> <else_keyword> { <statement> } | <elseif_keyword> <if_statement> | E
<while_loop>            --> while ( <exp> ) { <statement> }
<exp>                   --> <identifier> <comp_op> <identifier> | <identifier> <comp_op> <number> |
                            <number> <comp_op> <idenifier>
<comp_op>               --> '<','>','==','<>'
<string>                --> "'[^']*'"
<value>                 --> <number> | <string>
<number>                --> \'\d+(\.\d+)?'
<identifier>            --> '[a-zA-Z]\w*'
<type>                  --> 'int' | 'float' | 'string'
<elseif_keyword>        --> '\b(else\s*if)\b'
<else_keyword>          --> 'else'
<open_paren>            --> '('
<close_paren>           --> ')'
<openstatement_paren>   --> '{'
<closestatement_paren>  --> '}'
<variable>              --> <type> <assign_statement>
<assign_statement>      --> <identifier> = <value>
<print_statement>       --> 'Console.Writeline'+(<value>)
<statement>             --> <print_statement> ; | <assign_statement> ;
<comment>               --> <oneline_comment> | <multiLine_comment>
<oneline_comment>       --> '\/\/[^\n]*'
<multiLine_comment>     --> '\/\*(.|\n)*?\*\/'
```

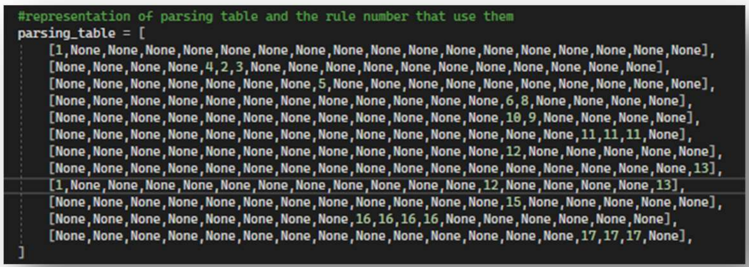
The production rules for this grammar can be seen below in both tabular format and within the parser. Also included is the tabular parse table which become a driving force and point of reference when trying to add different functions for the parser to accept during development:



1	<if_statement>	--> if (<exp>) { <statement> } <else_clause>
2	<else_clause>	--> else { <statement> } |
3	<else_clause>	--> elseif (<exp>) { <statement> } <else_clause>
4	<else_clause>	-->E
5	<while_loop>	--> while (<exp>) { <statement> }
6	<exp>	--> <identifier> <comp_op> <identifier>
7	<exp>	--> <identifier> <comp_op> <number>
8	<exp>	--> <number> <comp_op> <idenifier>
9	<value>	--> \d
10	<value>	--> \w
11	<variable>	--> <type> <assign_statement>
12	<assign_statement>	--> <identifier> = <value>
13	<print_statement>	--> 'Console.Writeline'+(<value>)
14	<statement>	--> <if_statement> | <print_statement> | <assign_statement>
15	<identifier>	--> \w
16	<comp_op>	--> < | > | == | <>
17	<type>	--> int | float | string

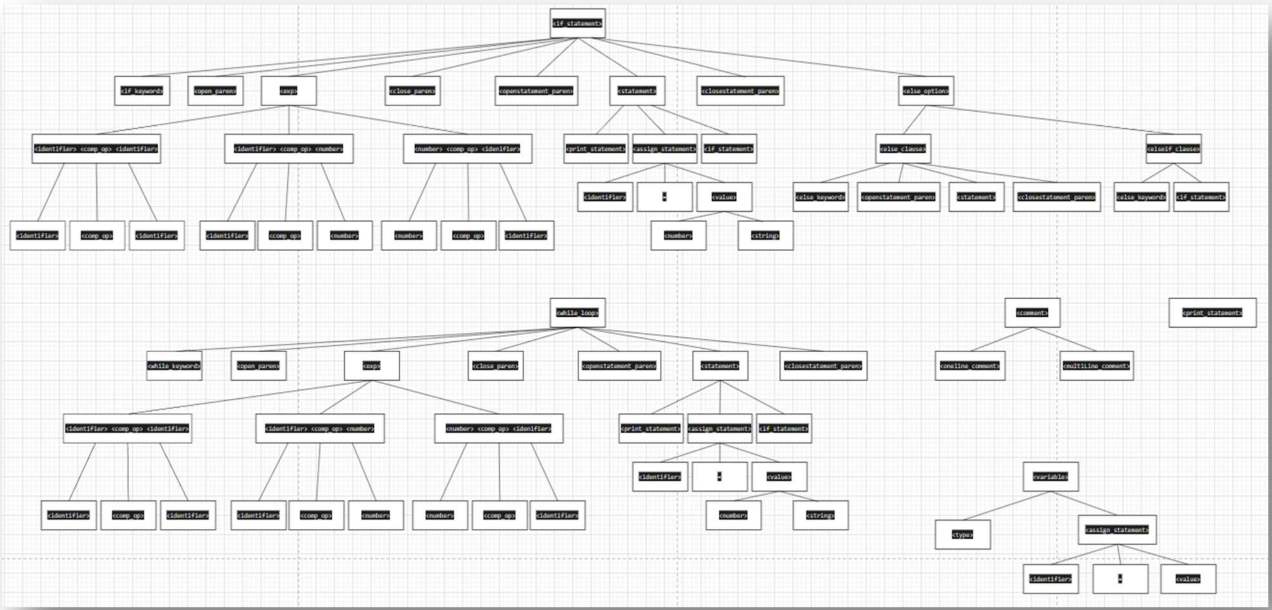


	IF	(	)	{	}	ELSE	ELSEIF	WHILE	<	>	==	<>	\w	\d	int	float	string	Console.Writeline
<if_statement>	1																	
<else_clause>					4	2	3											
<while_loop>								5										
<exp>													6	8				
<value>													10	9				
<variable>															11	11	11	
<assign_statement>													12					
<print_statement>																		13
<statement>	1												12					13
<identifier>													15					
<comparison_operator>									16	16	16	16						
<type>															17	17	17	



```python
#representation of parsing table and the rule number that use them
parsing_table = [
    [1,None,None,None,None,None,None,None,None,None,None,None,None,None,None,None,None,None],
    [None,None,None,None,4,2,3,None,None,None,None,None,None,None,None,None,None,None],
    [None,None,None,None,None,None,None,5,None,None,None,None,None,None,None,None,None,None],
    [None,None,None,None,None,None,None,None,None,None,None,None,6,8,None,None,None,None],
    [None,None,None,None,None,None,None,None,None,None,None,None,10,9,None,None,None,None],
    [None,None,None,None,None,None,None,None,None,None,None,None,None,None,11,11,11,None],
    [None,None,None,None,None,None,None,None,None,None,None,None,12,None,None,None,None,None],
    [None,None,None,None,None,None,None,None,None,None,None,None,None,None,None,None,None,13],
    [1,None,None,None,None,None,None,None,None,None,None,None,12,None,None,None,None,13],
    [None,None,None,None,None,None,None,None,None,None,None,None,15,None,None,None,None,None],
    [None,None,None,None,None,None,None,None,16,16,16,16,None,None,None,None,None,None],
    [None,None,None,None,None,None,None,None,None,None,None,None,None,None,17,17,17,None],
]
```

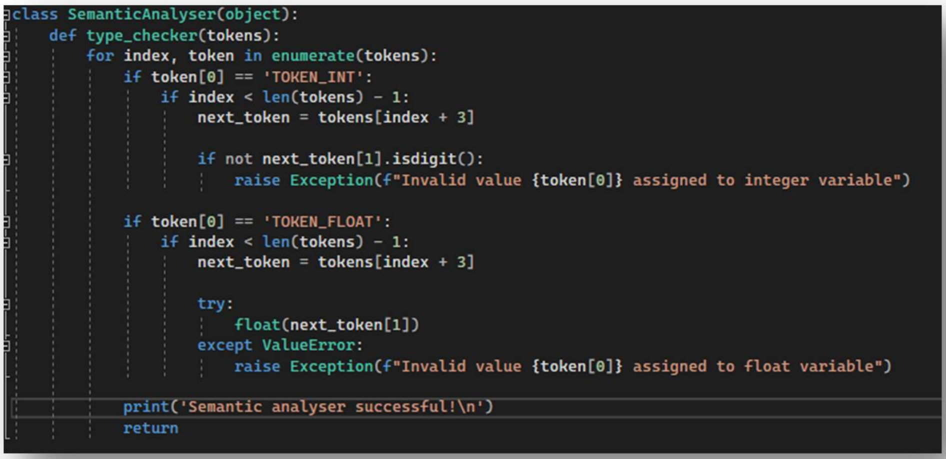
To prepare a visual representation of the context-free grammar, a parse tree (also known as an abstract syntax tree) was built which shows the above grammar that can be accepted. This shows a top-down left-most derivation. This is used to analyse the structure of the given input string so that it can work from the top breaking down the non-terminals into small parts by using the production rules.



Semantic Analyser

The semantic analyser is the stage of the compiler that will analyse and validate various behaviour of the source code. This can include type checking, scope checking and functional flow. For example, for type checking the semantic analyser could check those integers only accept whole numbers, or for functional flow, it could check for infinite loops or un-reachable code.

For this compiler, the type checking for integer and float types has been included to ensure that only values that match the respective type are assigned. The below is the type_checker function that loops through the tokens that have been captured by the lexical analyser, looking for an integer token or a float token. If either of those tokens are found it will do a check to ensure that the token value is that of the respective type.

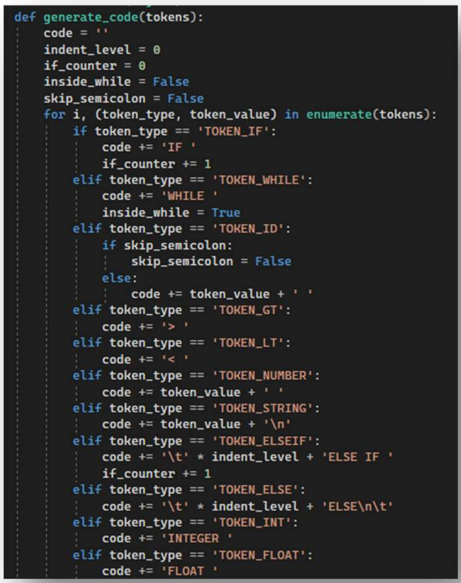


```
class SemanticAnalyser(object):
    def type_checker(tokens):
        for index, token in enumerate(tokens):
            if token[0] == 'TOKEN_INT':
                if index < len(tokens) - 1:
                    next_token = tokens[index + 3]

                    if not next_token[1].isdigit():
                        raise Exception(f"Invalid value {token[0]} assigned to integer variable")

            if token[0] == 'TOKEN_FLOAT':
                if index < len(tokens) - 1:
                    next_token = tokens[index + 3]

                    try:
                        float(next_token[1])
                    except ValueError:
                        raise Exception(f"Invalid value {token[0]} assigned to float variable")

            print('Semantic analyser successful!\n')
            return
```

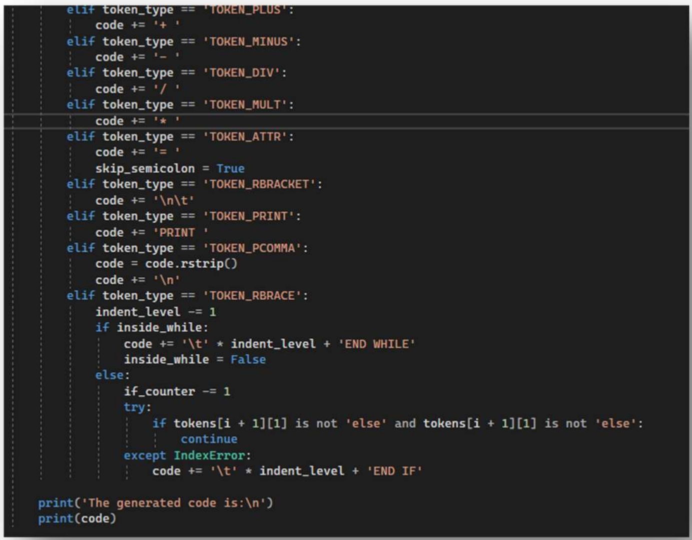
Code Generation

This phase is where the modern coding language will be translated into low-level language such as assembly code or machine code. By this point, the tokens that are once again passed in will have been parsed and are there for syntactically correct and any semantic analysis will have already been performed.

For this compiler, the conversion will be into a form of pseudocode that will represent the low-level language. The code generator was in most cases easy to add additional conversion by adding a token to the else if and then including what behaviour or format should following that token.



```python
def generate_code(tokens):
    code = ''
    indent_level = 0
    if_counter = 0
    inside_while = False
    skip_semicolon = False
    for i, (token_type, token_value) in enumerate(tokens):
        if token_type == 'TOKEN_IF':
            code += 'IF '
            if_counter += 1
        elif token_type == 'TOKEN_WHILE':
            code += 'WHILE '
            inside_while = True
        elif token_type == 'TOKEN_ID':
            if skip_semicolon:
                skip_semicolon = False
            else:
                code += token_value + ' '
        elif token_type == 'TOKEN_GT':
            code += '> '
        elif token_type == 'TOKEN_LT':
            code += '< '
        elif token_type == 'TOKEN_NUMBER':
            code += token_value + ' '
        elif token_type == 'TOKEN_STRING':
            code += token_value + '\n'
        elif token_type == 'TOKEN_ELSEIF':
            code += '\t' * indent_level + 'ELSE IF '
            if_counter += 1
        elif token_type == 'TOKEN_ELSE':
            code += '\t' * indent_level + 'ELSE\n\t'
        elif token_type == 'TOKEN_INT':
            code += 'INTEGER '
        elif token_type == 'TOKEN_FLOAT':
            code += 'FLOAT '
```



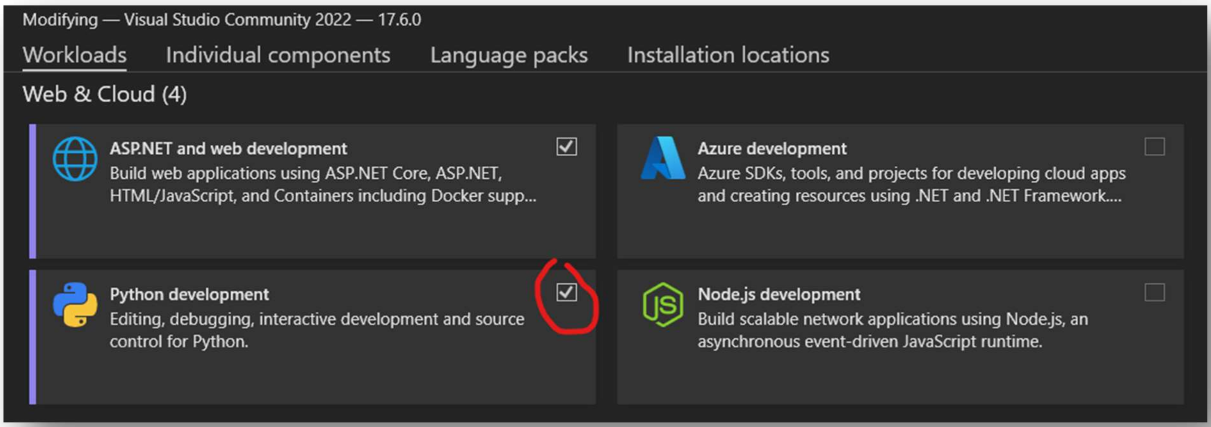
```python
        elif token_type == 'TOKEN_PLUS':
            code += '+ '
        elif token_type == 'TOKEN_MINUS':
            code += '- '
        elif token_type == 'TOKEN_DIV':
            code += '/ '
        elif token_type == 'TOKEN_MULT':
            code += '* '
        elif token_type == 'TOKEN_ATTR':
            code += '= '
            skip_semicolon = True
        elif token_type == 'TOKEN_RBRACKET':
            code += '\n\t'
        elif token_type == 'TOKEN_PRINT':
            code += 'PRINT '
        elif token_type == 'TOKEN_PCOMMA':
            code = code.rstrip()
            code += '\n'
        elif token_type == 'TOKEN_RBRACE':
            indent_level -= 1
            if inside_while:
                code += '\t' * indent_level + 'END WHILE'
                inside_while = False
            else:
                if_counter -= 1
                try:
                    if tokens[i + 1][1] is not 'else' and tokens[i + 1][1] is not 'else':
                        continue
                except IndexError:
                    code += '\t' * indent_level + 'END IF'

    print('The generated code is:\n')
    print(code)
```

Compiler Implementation

To implement the compiler, the developer used Visual Studio 17.6 and Python 3.11.

Firstly, python had to be downloaded from https://www.python.org/downloads/windows/.

Secondly, using the Visual Studio Installer program that comes with Visual Studio, the python development pack was installed to Visual Studio:



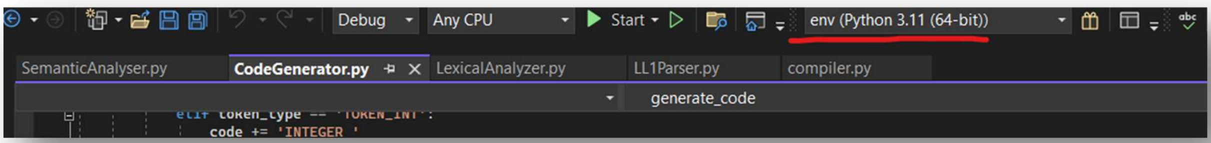
Modifying — Visual Studio Community 2022 — 17.6.0

Workloads | Individual components | Language packs | Installation locations

Web & Cloud (4)

- [✓] ASP.NET and web development
  Build web applications using ASP.NET Core, ASP.NET, HTML/JavaScript, and Containers including Docker supp...
- [ ] Azure development
  Azure SDKs, tools, and projects for developing cloud apps and creating resources using .NET and .NET Framework....
- [✓] Python development
  Editing, debugging, interactive development and source control for Python.
- [ ] Node.js development
  Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

From within the solution in Visual Studio an environment can then be run to allow execution(using start or f5, with or without breakpoints) of the code which will be the version of python that was downloaded:



Debug | Any CPU | Start | env (Python 3.11 (64-bit))

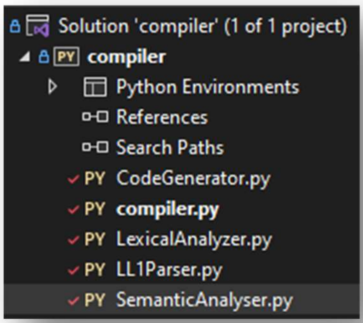
SemanticAnalyser.py | CodeGenerator.py | LexicalAnalyzer.py | LL1Parser.py | compiler.py

generate_code

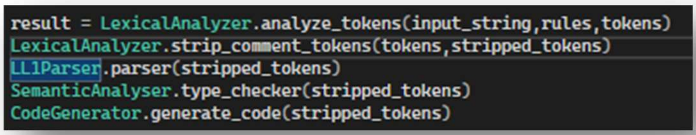
```
elif token_type == 'TOKEN_INT':
    code += 'INTEGER '
```

Please note that the compiler has been built using a 'match' function that is only available in versions of python above 3.10.

The program has the following structure of classes whereby 'compiler' is the main method that calls the stages in sequence as per the below:

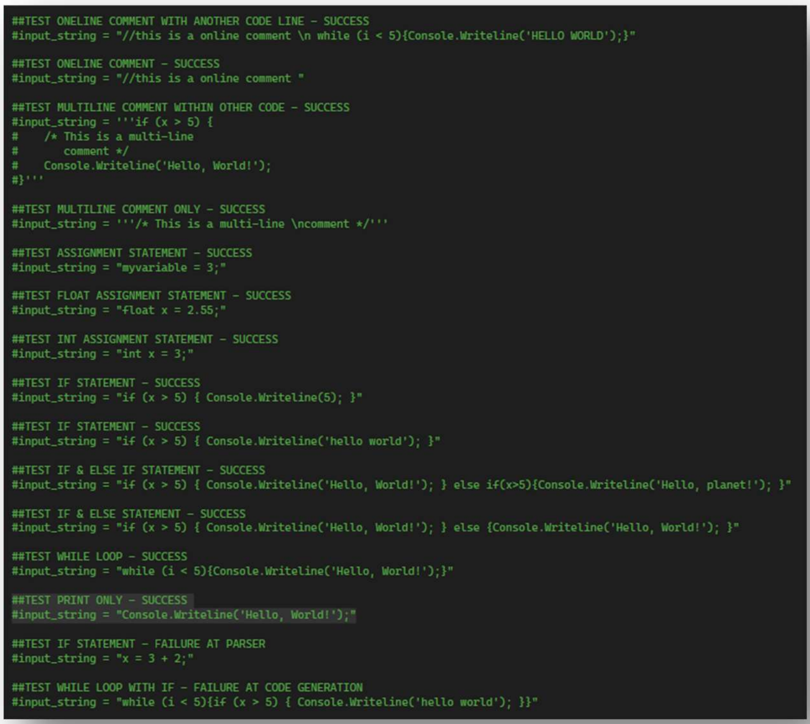


```
Solution 'compiler' (1 of 1 project)
  compiler
    Python Environments
    References
    Search Paths
    CodeGenerator.py
    compiler.py
    LexicalAnalyzer.py
    LL1Parser.py
    SemanticAnalyser.py
```



```
result = LexicalAnalyzer.analyze_tokens(input_string,rules,tokens)
LexicalAnalyzer.strip_comment_tokens(tokens,stripped_tokens)
LL1Parser.parser(stripped_tokens)
SemanticAnalyser.type_checker(stripped_tokens)
CodeGenerator.generate_code(stripped_tokens)
```

Within the compiler.py file there are test inputs that have been commented out to assist with testing. To use one of these test cases uncomment out one of the input_string = lines:



```
##TEST ONELINE COMMENT WITH ANOTHER CODE LINE - SUCCESS
#input_string = "//this is a online comment \n while (i < 5){Console.Writeline('HELLO WORLD');}"

##TEST ONELINE COMMENT - SUCCESS
#input_string = "//this is a online comment "

##TEST MULTILINE COMMENT WITHIN OTHER CODE - SUCCESS
#input_string = '''if (x > 5) {
#    /* This is a multi-line
#       comment */
#    Console.Writeline('Hello, World!');
#}'''

##TEST MULTILINE COMMENT ONLY - SUCCESS
#input_string = '''/* This is a multi-line \ncomment */'''

##TEST ASSIGNMENT STATEMENT - SUCCESS
#input_string = "myvariable = 3;"

##TEST FLOAT ASSIGNMENT STATEMENT - SUCCESS
#input_string = "float x = 2.55;"

##TEST INT ASSIGNMENT STATEMENT - SUCCESS
#input_string = "int x = 3;"

##TEST IF STATEMENT - SUCCESS
#input_string = "if (x > 5) { Console.Writeline(5); }"

##TEST IF STATEMENT - SUCCESS
#input_string = "if (x > 5) { Console.Writeline('hello world'); }"

##TEST IF & ELSE IF STATEMENT - SUCCESS
#input_string = "if (x > 5) { Console.Writeline('Hello, World!'); } else if(x>5){Console.Writeline('Hello, planet!'); }"

##TEST IF & ELSE STATEMENT - SUCCESS
#input_string = "if (x > 5) { Console.Writeline('Hello, World!'); } else {Console.Writeline('Hello, World!'); }"

##TEST WHILE LOOP - SUCCESS
#input_string = "while (i < 5){Console.Writeline('Hello, World!');}"

##TEST PRINT ONLY - SUCCESS
#input_string = "Console.Writeline('Hello, World!');"

##TEST IF STATEMENT - FAILURE AT PARSER
#input_string = "x = 3 + 2;"

##TEST WHILE LOOP WITH IF - FAILURE AT CODE GENERATION
#input_string = "while (i < 5){if (x > 5) { Console.Writeline('hello world'); }}"
```

To accompany the source code a brief demo video called DemoVideo.mkv has been included which can be found in the st12345678-DAT4005-PRAC1 folder alongside the program and the design word document. All artifacts in this document have also been included.

Optimization

For this compiler design the developer was conscious of ensuring that loops were kept to a minimum and there weren't too many nested loops to aid with the time taken to run the code through to code generation.

Within the LL(1) parser there is a loop to get the indices of the parse table and with more research this could possibly be removed to optimize the code. In addition, the code could be somewhat re-factored to improve readability and maintainability so that additional functions and syntax could be quickly added.

Getting a fully working example of a compiler for the given statements was very useful to build problem-solving skills, testing and debugging skills as well as gaining an understanding of key data structures and features of the Python language. Working through the design techniques of the LL(1) parser made building it easier to understand what was required so that time and efficiency of the overall computation code be clear before development.