Interpreter Test Rapport

Student: Faizal Supriadi

Studentnummer: 1735666

Email: Faizal.faizalrachmansupriadi@student.hu.nl

Github: https://github.com/FaizalSupriadi/CodePlanner

Inhoud

1. Inleiding

Voor het vak Advanced Technical Programming (ATP) is de opdracht gegeven om een interpreter te maken waarbij men gebruik kan maken van een bestaand of een eigen geschreven taal. Zo heb ik gekozen om een eigen taal te schrijven genaamd “CodePlanner”, waarbij de benaming “.traffic” wordt gebruikt als extensie. De thema van de taal gaat over het gebruik van transport.

Zo zijn er drie grote files geschreven in Python die de interpreter systeem compleet maakt: myLexer, myParser en myInterpreter. Om de betrouwbaarheid van de interpreter te waarborgen zijn unit tests geschreven en is dit rapport geschreven.

2. Methoden

Er is gekozen dat de scripts minstens getest worden met unit tests die de meest belangrijke onderwerpen bevat:

- myLexer.py – Correcte gecreëerde Tokens
- myParser.py – Correcte gecreëerde AST
- myInterpreter.py – Correcte gecreëerde Interpretatie en Symbol Table

Voor de lexer, parser en interpreter zal ook een test gemaakt worden die de system test heet. De system test wordt gebruikt om te verfiëren dat elke fase correct werkt. In de system test wordt de script genaamd “test_ system.traffic” gebruikt. De functie van de script is het printen van rekensommen. In de context van de lexer wordt er getest of de tokens correct zijn, in de parser wordt de AST getest en in de interpreter worden de symbol table en interpretatie op juiste waarden getest.

2.1. Lexer

Wat belangrijk is in myLexer.py is dat alle tokens correct gecreëerd worden. Een token is een klasse die een type en een waarde bevat. Er wordt daarom gekeken of deze variabelen correct zijn. Om dit te gaan testen wordt de make_tokens() functie van de lexer gebruikt. Om deze functie te gebruiken moet een CodePlanner script gebruikt worden en gegeven aan de functie. Voor de lexer is een TestLexer klasse gemaakt.

2.2. Parser

Het belangrijkste bij myParser.py is dat de tokens die worden meegegeven aan de parser, de juiste Abstract Syntax Tree (AST) volgen. Hiervoor zijn meerdere node typen gemaakt die corresponderen aan de verschillende beschikbare functies in de taal. Om de betrouwbaarheid van de parser te valideren wordt daarom een unit test gemaakt die een verwachte waarde met het gerealiseerde waarde vergelijkt. Voor de parser is een TestParser klasse gemaakt.

2.3. Interpreter

Wat belangrijk is bij interpreter.py is dat de AST die meegegeven wordt aan de interpreter, correct geïnterpreteerd wordt. Hierbij moet het geïnterpreteerde resultaat en de symbol table correct zijn. De symbol table is een array met functies uit de codeplanner taal. Om de betrouwbaarheid van de interpreter te valideren wordt daarom een unit test gemaakt. Voor de interpreter is een TestInterpreter klasse gemaakt.

3. Resultaten

3.1. Lexer

3.1.1. Token Test

De token test is een unit test. In de token test wordt de token generatie van de lexer getest. In de test zal een tekstfile geopend worden waarin alle tekens en woorden staan die de codeplanner taal ondersteund. De script is echter niet een script die uitvoerbaar is voor de parser, het is dus bedoeld om echt alleen de token generatie te testen. Hierbij wordt gedacht aan bijvoorbeeld de plus teken, de return statements, etc. Om te verifiëren dat de juiste tokens en waarden aangemaakt worden, zijn er lijsten gemaakt die de verwachte typen en waarden bevatten. Met de assertEqual functie is het mogelijk om de aangemaakte tokens te vergelijken met de verwachte tokens om daardoor de functionaliteit ervan te testen.

```
#Test to check if the lexer creates all tokens and if they have their correct values.
def test_all_token_creation(self):

    # Create tokens from function
    with open("tests/test_lex.traffic", "r") as f:
        text = f.read()
    tokens, _ = lex.make_tokens([], lex.Position(0, 0, 0, "lex_test", text), text)

    # All token types
    types = [tt.TT_PLUS, tt.TT_MINUS, tt.TT_MUL, tt.TT_DIV, tt.TT_PLUS, tt.TT_MINUS, tt.TT_MUL, tt.TT_DIV, tt.TT_LT, tt.TT_GT, tt.TT_IDENTIFIER, tt.TT_EQ,
        tt.TT_KEYWORD, tt.TT_IDENTIFIER, tt.TT_IDENTIFIER, tt.TT_KEYWORD, tt.TT_KEYWORD, tt.TT_KEYWORD, tt.TT_KEYWORD, tt.TT_KEYWORD, tt.TT_KEYWORD,
        tt.TT_KEYWORD, tt.TT_KEYWORD, tt.TT_KEYWORD, tt.TT_KEYWORD, tt.TT_NEWLINE, tt.TT_IDENTIFIER, tt.TT_LPAREN, tt.TT_RPAREN, tt.TT_LSQUARE,
        tt.TT_RSQUARE, tt.TT_LCURLY, tt.TT_RCURLY, tt.TT_COMMA, tt.TT_COLON]

    # All token type values
    values = [None, None, None, None, None, None, None, None, None, None,
        "equals", None, "NOT", "RED", "GREEN", "DESTINATION", "TRAFFIC",
        "BYPASS", "FLEE", "GPS", "REFUEL", "ROUTE", "VEHICLE", "DRIVING",
        "SPEEDING", None, "Identity", None, None, None, None, None, None,
        None, None,]

    for i in range(len(tokens)):
        self.assertEqual(tokens[i].type, types[i])
        self.assertEqual(tokens[i].value, values[i])
```

Figuur 1 Token Test Code

3.1.2. Script Test

De script test is een unit test. In de script test wordt een functioneel codeplanner script getest. Door deze test is het dus mogelijk om de token generatie van een functioneel script te waarborgen. Hierbij wordt de comments script gebruikt waarin een functie aangemaakt wordt met verschillende comments die genegeerd moeten worden. Hierbij worden de verwachte tokens in een lijst gezet en wordt het geverifieerd met de assertEqual functie.

```
#Test to check functionality with "comments.traffic" script
def test_run_lexer_on_script(self):

    # Create tokens from function
    with open("examples/comments.traffic", "r") as f:
        text = f.read()
    tokens, _ = lex.make_tokens([], lex.Position(0, 0, 0, "comments", text), text)

    # All script token types
    types = [tt.TT_NEWLINE, tt.TT_KEYWORD, tt.TT_INT, tt.TT_EE, tt.TT_INT, tt.TT_KEYWORD,
        tt.TT_NEWLINE, tt.TT_NEWLINE, tt.TT_KEYWORD, tt.TT_LPAREN, tt.TT_INT, tt.TT_RPAREN]

    # All script token type values
    values = [None, "TRAFFIC", 2, None, 2, "GPS", None, None, "PRINT", None, 100, None]

    for i in range(len(tokens)):
        self.assertEqual(tokens[i].type, types[i])
        self.assertEqual(tokens[i].value, values[i])
```

Figuur 2 Script Test Code

Met het runnen van de token en script tests is het succesvol in 0.002 seconden uitgevoerd.

```
[Running] python -u "c:\Users\Anwar\Desktop\ProjectCodePlan\CodePlanner\test_lexer.py"
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```

Figuur 3 Unit Test Resultaat van Token en Script test

3.2. Parser

3.2.1. AST Test

De AST test is een unit test. In de AST test wordt getest of de tokens die aan de parser gegeven worden correct door de AST gaan. Om te beginnen zal een script geopend en gegeven worden aan de lexer. De lexer genereerd dan tokens en worden dan gegeven aan de parser. Om de data te valideren is gekozen om de repr() functie te gebruiken. Door deze functie kan de AST die uit de parser komt, zichzelf weergeven in de vorm van een string, waardoor het met de assertEqual functie vergeleken kan worden met de verwachte uitkomst die ook string is.

```
class TestParser(unittest.TestCase):

    #Test to check if the parser creates the correct AST and if they have their correct values.
    def test_correct_ast(self):
        with open("tests/parser_test.traffic", "r") as f:
            text = f.read()
        tokens, _ = lex.make_tokens([], lex.Position(0, 0, 0, "parser_test", text), text)
        ast, _ = parser.parse(tokens)
        str_ast = repr(ast)
        self.assertEqual(str_ast, "(LISTNODE: ELEMNODES: [(PRINTNODE, (BINOPNODE: NUMBERNODE: INT:1, PLUS, NUMBERNODE: INT:1))])")

if __name__ == '__main__':
    unittest.main()
```

Figuur 4 AST Test Code

Met het runnen van de AST is het succesvol in 0.001 seconden uitgevoerd.

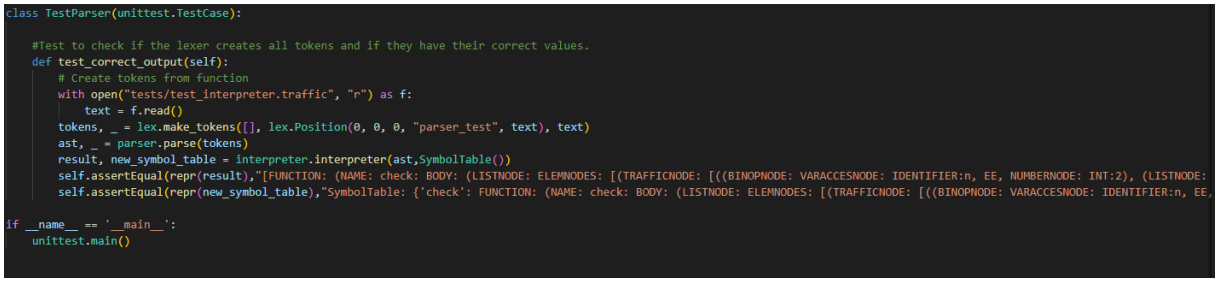
```
[Running] python -u "c:\Users\Anwar\Desktop\ProjectCodePlan\CodePlanner\test_parser.py"
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Figuur 5 Unit Test Resultaat van AST

3.3. Interpreter

3.3.1. Symbol & Output Test

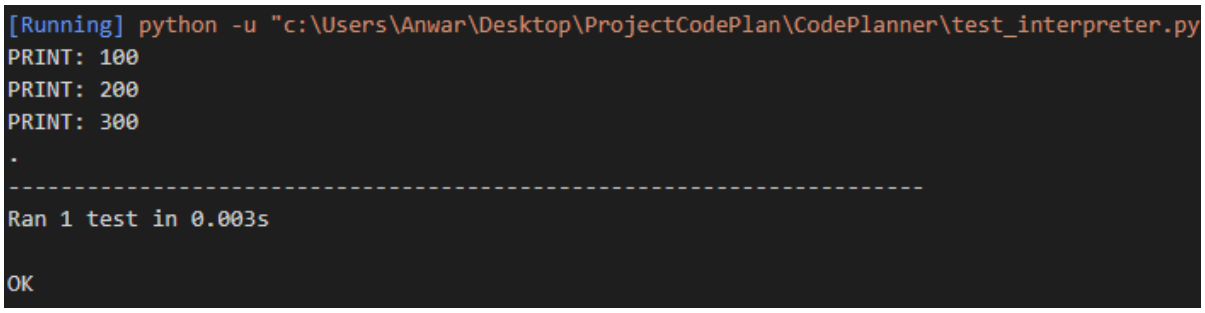
De Symbol & Output test is een unit test. In de symbol en output test zal getest worden of de AST goed geïnterpreteerd wordt door de interpreter. De interpreter krijgt de AST en een lege symbol table klasse. Uiteindelijk geeft de interpreter zijn interpretatie en een symbol table. Om deze te vergelijken met de verwachte resultaten wordt de repr() functie gebruikt om het als een leesbare string te krijgen. Hierdoor kan het met de assertEqual getest worden.



```
class TestParser(unittest.TestCase):

    #Test to check if the lexer creates all tokens and if they have their correct values.
    def test_correct_output(self):
        # Create tokens from function
        with open("tests/test_interpreter.traffic", "r") as f:
            text = f.read()
        tokens, _ = lex.make_tokens([], lex.Position(0, 0, 0, "parser_test", text), text)
        ast, _ = parser.parse(tokens)
        result, new_symbol_table = interpreter.interpreter(ast,SymbolTable())
        self.assertEqual(repr(result),"[FUNCTION: (NAME: check: BODY: (LISTNODE: ELEMNODES: [(TRAFFICNODE: [((BINOPNODE: VARACCESNODE: IDENTIFIER:n, EE, NUMBERNODE: INT:2), (LISTNODE:
        self.assertEqual(repr(new_symbol_table),"SymbolTable: {'check': FUNCTION: (NAME: check: BODY: (LISTNODE: ELEMNODES: [(TRAFFICNODE: [((BINOPNODE: VARACCESNODE: IDENTIFIER:n, EE,

if __name__ == '__main__':
    unittest.main()
```

Figuur 6 Symbol en Output Test Code

Met het runnen van de Symbol & Output test is het succesvol in 0.003 seconden uitgevoerd. Er is hierbij ook te zien dat de juiste waarden worden geprint door de interpreter.

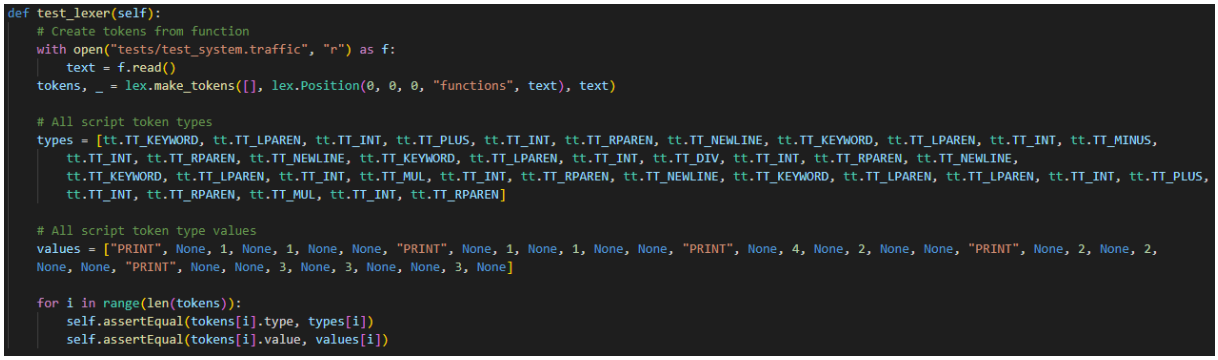


```
[Running] python -u "c:\Users\Anwar\Desktop\ProjectCodePlan\CodePlanner\test_interpreter.py
PRINT: 100
PRINT: 200
PRINT: 300
.
----------------------------------------------------------------------
Ran 1 test in 0.003s

OK
```

Figuur 7 Unit Test Resultaat van Symbol & Output en Math Test

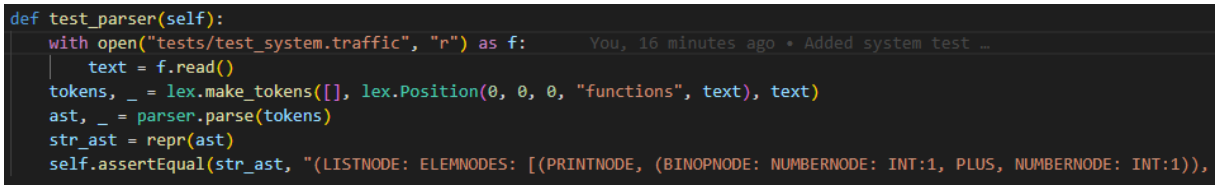
3.4. System Test

De system test waarborgt dat de stappen van tokens > AST > interpretatie goed verlopen. Zo is een TestSystem klasse gemaakt waarin de lexer, parser en interpreter getest worden. De lexer, parser en interpreter worden dan getest op dezelfde manieren in hun individuele testen. Zo wordt de lexer getest met hetzelfde idee als de script test, de parser wordt getest met het idee van de AST test en de interpreter wordt dan getest met het idee van de Symbol & Output test. De verwachte resultaten zoals de token types zijn dan veranderd naar de juiste context.



```
def test_lexer(self):
    # Create tokens from function
    with open("tests/test_system.traffic", "r") as f:
        text = f.read()
    tokens, _ = lex.make_tokens([], lex.Position(0, 0, 0, "functions", text), text)

    # All script token types
    types = [tt.TT_KEYWORD, tt.TT_LPAREN, tt.TT_INT, tt.TT_PLUS, tt.TT_INT, tt.TT_RPAREN, tt.TT_NEWLINE, tt.TT_KEYWORD, tt.TT_LPAREN, tt.TT_INT, tt.TT_MINUS,
        tt.TT_INT, tt.TT_RPAREN, tt.TT_NEWLINE, tt.TT_KEYWORD, tt.TT_LPAREN, tt.TT_INT, tt.TT_DIV, tt.TT_INT, tt.TT_RPAREN, tt.TT_NEWLINE,
        tt.TT_KEYWORD, tt.TT_LPAREN, tt.TT_INT, tt.TT_MUL, tt.TT_INT, tt.TT_RPAREN, tt.TT_NEWLINE, tt.TT_KEYWORD, tt.TT_LPAREN, tt.TT_LPAREN, tt.TT_INT, tt.TT_PLUS,
        tt.TT_INT, tt.TT_RPAREN, tt.TT_MUL, tt.TT_INT, tt.TT_RPAREN]

    # All script token type values
    values = ["PRINT", None, 1, None, 1, None, None, "PRINT", None, 1, None, 1, None, None, "PRINT", None, 4, None, 2, None, None, "PRINT", None, 2, None, 2,
    None, None, "PRINT", None, None, 3, None, 3, None, None, 3, None]

    for i in range(len(tokens)):
        self.assertEqual(tokens[i].type, types[i])
        self.assertEqual(tokens[i].value, values[i])
```

Figuur 8 System Test van Lexer



```
def test_parser(self):
    with open("tests/test_system.traffic", "r") as f:      You, 16 minutes ago • Added system test …
        text = f.read()
    tokens, _ = lex.make_tokens([], lex.Position(0, 0, 0, "functions", text), text)
    ast, _ = parser.parse(tokens)
    str_ast = repr(ast)
    self.assertEqual(str_ast, "(LISTNODE: ELEMNODES: [(PRINTNODE, (BINOPNODE: NUMBERNODE: INT:1, PLUS, NUMBERNODE: INT:1)),
```

Figuur 9 System Test van Parser

```
def test_interpreter(self):
    with open("tests/test_system.traffic", "r") as f:
        text = f.read()
    tokens, _ = lex.make_tokens([], lex.Position(0, 0, 0, "functions", text), text)
    ast, _ = parser.parse(tokens)
    result, new_symbol_table = interpreter.interpreter(ast,SymbolTable())
    self.assertEqual(repr(result),"[PRINTED, PRINTED, PRINTED, PRINTED, PRINTED]")
    self.assertEqual(repr(new_symbol_table),"SymbolTable: {}")
```

Figuur 10 System Test van Interpreter

De system test is succesvol uitgevoerd met een tijd van 0.002 seconden.

```
[Running] python -u "c:\Users\Anwar\Desktop\ProjectCodePlan\CodePlanner\test_system.py"
PRINT: 2
PRINT:. 0
PRINT: 2.0
PRINT: 4
PRINT: 18
..
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK

[Done] exited with code=0 in 0.729 seconds
```

Figuur 11 System Test Resultaat

4. Conclusie

De lexer, parser en interpreter worden getest op de belangrijkste delen en het systeem wordt in zijn geheel getest.

Met de testen van de lexer is het mogelijk om de verwachte resultaten van token generatie te valideren. Met de testen van de parser weet men dat de AST juist wordt gevolgd en gemaakt. Met de testen van de interpreter wordt de kwaliteit van de verwachte symbol table en interpretatie gewaarborgd. Uiteindelijk met de system test worden de lexer, parser en interpreter getest op hun individuele kwaliteiten en worden deze ook gewaarborgd. Tijdens runtime werden deze testen allemaal succesvol voltooid in een tijd van 0.003 seconden of minder.

Al in al is het mogelijk om met deze test set de kwaliteit van de “CodePlanner” interpreter te waarborgen.