Ministry of Education, Culture and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work *No. 6*

Formal Languages & Finite Automata

Elaborated by:
Gorcea Alexandrina, FAF-223
Verified by: Dumitru Crețu, univ. assist.

Chișinău, 2024

Theoretical considerations:

Parsing is a crucial step in the compilation process, where the sequence of tokens generated by the lexer is analyzed to determine if it conforms to the syntax rules specified by the grammar of the language. This process involves constructing a parse tree or abstract syntax tree (AST) that represents the hierarchical structure of the input program.

The parser typically employs techniques such as top-down or bottom-up parsing algorithms to systematically analyze the tokens and derive the syntactic structure. During parsing, the parser matches the sequence of tokens against the production rules defined in the grammar. If the sequence matches a rule, the parser constructs a subtree corresponding to that rule. If there's a mismatch or ambiguity, the parser may raise an error or attempt to resolve the ambiguity based on predefined rules or heuristics.

The AST, derived from the parse tree, provides a more concise and abstract representation of the input program. In the AST, each node corresponds to a construct in the language, such as expressions, statements, or declarations. The leaf nodes of the AST represent tokens generated by the lexer, while internal nodes represent syntactic structures derived during parsing.

One of the key characteristics of an AST is its ability to abstract away irrelevant details of the input program, such as whitespace, comments, and punctuation. This abstraction simplifies subsequent stages of the compilation process, such as semantic analysis and code generation, by focusing only on the essential syntactic elements of the program.

Lexical analysis, performed by the lexer, precedes parsing and involves categorizing the input characters into tokens based on predefined rules specified by regular expressions. Each token typically consists of a token type and its corresponding value or lexeme. Regular expressions are used to define patterns that match specific lexical elements, such as identifiers, keywords, operators, and literals, within the input string. These patterns serve as the basis for tokenizing the input string into meaningful units for further processing by the parser.

In summary, parsing, abstract syntax trees, lexical analysis, and regular expressions are integral components of the compilation process, each playing a distinct role in transforming source code into executable programs. By systematically analyzing and abstracting the syntactic and lexical elements of the input program, these processes lay the foundation for subsequent stages of compilation, such as semantic analysis, optimization, and code generation.

Objectives:

1. Get familiar with parsing, what it is and how it can be programmed [1].
2. Get familiar with the concept of AST [2].
3. In addition to what has been done in the 3rd lab work do the following:
    i. In case you didn't have a type that denotes the possible types of tokens you need to:
        a. Have a type TokenType (like an enum) that can be used in the lexical analysis to categorize the tokens.
        b. Please use regular expressions to identify the type of the token.
    ii. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
    iii. Implement a simple parser program that could extract the syntactic information from the input text.

Implementation description:

In the following code, the lexer method adeptly utilizes regular expressions to categorize elements of the program, while the AST implementation provides a structured representation of the code's hierarchical structure. The manual construction of the AST demonstrates a thorough understanding of how nodes and relationships are established based on the program's syntax, while the utilization of Python's 'ast' module offers a more automated approach for generating ASTs, enhancing scalability for complex programs. The parser class effectively parses simple Python statements, employing logical methods for moving through tokens, matching expected patterns, and handling errors gracefully. Overall, the implementation showcases a robust integration of lexer, AST generation, and parsing, facilitating comprehensive analysis and understanding of the input code.

The main file orchestrates the testing and integration of the implemented components, demonstrating their functionality with diverse inputs. The inclusion of testing scenarios with both valid and invalid statements underscores the reliability and robustness of the lexer and parser.

To the lexer method from laboratory nr. 3, I have created another lexer method that uses regular expressions to categorize the elements of the program. First, we create a list that contains the token types and the regular expressions used to identify the type of the element in the program. Next, we create a pattern by joining the regular expressions for all token types and compile the pattern into a regular expression object. With this object, we iterate over matches in the input string and add the tokens to the list.

```
token_types = [
    ('NUMBER', r'\d+(\.\d*)?'),
    ('STRING', r'\".*?\"|\'.*?\''),
    ('KEYWORD', r'(if|else|while|for|def|return|in)(?!\w)'),
    ('IDENTIFIER', r'[a-zA-Z_]\w*'),
    ('OPERATOR', r'[+\-*%/]'),
    ('COMPARISON', r'==|!=|<=|>=|<|>'),
    ('ASSIGNMENT', r'='),
    ('SEPARATOR', r'\(|\)|\[|\]|\{|\}|,|;|:'),
    ('NEWLINE', r'\n'),
    ('SKIP', r'[ \t]+'),
    ('COMMENT', r'#.*'),
    ('UNKNOWN', r'.'),
]

token_pattern = '|'.join('(?P<%s>%s)' % spec for spec in token_types)
regex = re.compile(token_pattern)

for match in regex.finditer(self.program):
    token_type = match.lastgroup
    token_value = match.group(token_type)
    if token_type == 'UNKNOWN':
        tokens.append((token_type, ord(token_value)))
    elif token_type != 'SKIP' and token_type != 'NEWLINE':
        tokens.append((token_type, token_value))

return tokens
```

To implement the AST data structure effectively, a robust foundation is laid with the creation of the "ASTNode" class. This class serves as the fundamental building block, encapsulating the essential elements of the abstract syntax tree: token type, value, and child nodes. By structuring the ASTNode class in this manner, it facilitates the representation of diverse syntactic constructs within the AST, ensuring flexibility and scalability. Each ASTNode instance encapsulates a specific syntactic element, such as identifiers, literals, expressions, or statements, allowing for precise representation and traversal of the AST.

```python
class ASTNode:
    def __init__(self, token_type, value=None):
        self.token_type = token_type
        self.value = value
        self.children = []

    def add_child(self, child):
        self.children.append(child)
        return child

    def __str__(self):
        return f'{self.token_type}: {self.value}'
```

For the below Python function from lab. 3, I can create the AST in 2 different ways, first is to manually create an AST, and, second, to use the "ast" module in Python.

```python
def sum_list(numbers):
    total = 0
    for num in numbers:
        total = total + num
    return total
```

The manual construction of the AST involves systematically traversing the tokenized program structure, adding nodes to represent each syntactic element. Beginning with the root node representing the program, subsequent nodes reflect the hierarchical organization of statements, expressions, and control flow constructs. Each node encapsulates a token type and value, accurately representing the corresponding syntactic element. By following the program's structure and adding child nodes for nested constructs, such as functions, loops, and assignments, the resulting AST provides a concise and structured representation of the code's syntax. This method offers a clear and intuitive approach to understanding the program's structure within the AST framework, facilitating further analysis and interpretation.

```
    root = ASTNode('PROGRAM')
    current_node = root
    current_node = current_node.add_child(ASTNode(tokens[0][0], tokens[0][1]))
    current_node.add_child(ASTNode(tokens[1][0], tokens[1][1]))
    current_node.add_child(ASTNode(tokens[3][0], tokens[3][1]))
    current_node = current_node.add_child(ASTNode('body', None))
    parent = current_node
    current_node = current_node.add_child(ASTNode(tokens[7][0], tokens[7][1]))
    current_node.add_child(ASTNode(tokens[6][0], tokens[6][1]))
    current_node.add_child(ASTNode(tokens[8][0], tokens[8][1]))
    current_node = parent
    current_node = current_node.add_child(ASTNode(tokens[9][0], tokens[9][1]))
    current_node.add_child(ASTNode(tokens[10][0], tokens[10][1]))
    current_node.add_child(ASTNode(tokens[11][0], tokens[11][1]))
    current_node.add_child(ASTNode(tokens[12][0], tokens[12][1]))
    current_node = current_node.add_child(ASTNode(tokens[15][0], tokens[15][1]))
    current_node.add_child(ASTNode(tokens[14][0], tokens[14][1]))
    current_node = current_node.add_child(ASTNode(tokens[17][0], tokens[17][1]))
    current_node.add_child(ASTNode(tokens[16][0], tokens[16][1]))
    current_node.add_child(ASTNode(tokens[18][0], tokens[18][1]))
    current_node = parent
    current_node = current_node.add_child(ASTNode(tokens[19][0], tokens[19][1]))
    current_node.add_child(ASTNode(tokens[20][0], tokens[20][1]))

    return root
```

In the second method, the AST is generated using Python's built-in 'ast' module, which parses the source code into an abstract syntax tree directly. The 'parse' function of the 'ast' module takes the source code as input and produces a structured AST representing the program's syntax. Once parsed, the AST can be further analyzed or manipulated programmatically. The 'ast.dump' function is then used to print the AST in a human-readable format for debugging or analysis purposes. This printed representation provides a detailed view of the AST's structure, including nodes, their attributes, and relationships.

```
def create_ast_method_2(source_code):
    tree = ast.parse(source_code)
    print(ast.dump(tree))
```

Given the complexity of parsing the entire Python programming language, a focused approach is taken to develop a parser tailored for handling simple Python statements. The parser class is instantiated with essential attributes, including the list of tokens to parse, the current token being processed, and the current index indicating the position within the token list. Initializing these attributes sets the stage for parsing operations. Additionally, the 'next_token' method is invoked to advance the parser to the first token in the list, preparing it for subsequent parsing actions. This structured setup streamlines the parsing process, enabling efficient handling of Python statements while avoiding the extensive effort required for a comprehensive language parser.

```
self.tokens = tokens
self.current_token = None
self.index = -1
self.next_token()
```

Next, we create a method to move to the next token in the list given by the Lexer.

```
self.index += 1
if self.index < len(self.tokens):
    self.current_token = self.tokens[self.index]
else:
    self.current_token = None
```

Then, we create a method to check if the current token matches the expected token. If it does, we move to the next token, otherwise we raise a syntax error. This error handling ensures that parsing proceeds only when the expected token type is encountered, maintaining the syntactic integrity of the input code.

```
if self.current_token and self.current_token[0] == expected_token:
    self.next_token()
else:
    raise SyntaxError(f"Expected {expected_token}, but found {self.current_token[0]}")
```

The next method that we have to create is the one that parses an expression consisting of terms and operators. the parser begins by parsing a term, which typically represents a basic unit of an expression, such as a number or an identifier.

```
self.parse_term()
while self.current_token and self.current_token[0] == 'OPERATOR':
    self.match('OPERATOR')
    self.parse_term()
```

This method is responsible for parsing a term, which represents a basic unit of an expression, such as an identifier or a number. The method first checks if there is a current token available and if its type is either 'IDENTIFIER' or 'NUMBER', indicating the presence of a valid term. If the current token matches one of these expected types, the parser consumes the token using the 'match' method, ensuring syntactic correctness.

```
if self.current_token and self.current_token[0] in ['IDENTIFIER', 'NUMBER']:
    self.match(self.current_token[0])
else:
    raise SyntaxError(f"Expected IDENTIFIER or NUMBER, but found {self.current_token[0]}")
```

However, if the current token does not match the expected 'IDENTIFIER' or 'NUMBER' types, it indicates a syntax error. In such cases, the method raises a 'SyntaxError' exception, indicating that the input expression does not adhere to the expected syntax. The exception message

provides additional context by specifying the actual token type encountered, helping developers identify and rectify syntax errors in their code effectively. This error handling mechanism ensures robustness in parsing, enabling the parser to enforce strict adherence to the syntax rules of the language.

```
self.match('IDENTIFIER')
self.match('ASSIGNMENT')
self.parse_expression()
```

We also write a method to parse the assignment statement. Thus, it will expect an identifier, an assignment operator, and the expression on the right-hand side of the assignment.

Then, we write the general method to parse the statement.

```
self.parse_assignment()
if self.current_token:
    raise SyntaxError(f"Unexpected token: {self.current_token[0]}")
```

Finally, we write the last method, which will output weather the program is written correctly or not.

```
try:
    self.parse_statement()
    print("Parsing successful. The statement is correctly written.")
except SyntaxError as e:
    print("Parsing failed. The statement is incorrectly written.")
    print(e)
```

Bellow is the main file which instantiates the classes and tests the methods.

```
def build_graph(node, graph):
    graph.node(str(id(node)), str(node))
    for child in node.children:
        graph.edge(str(id(node)), str(id(child)))
        build_graph(child, graph)


python_program = '''
def sum_list(numbers):
    total = 0
    for num in numbers:
        total = total + num
    return total
'''
lexer_11 = lexer.Lexer(python_program)
tokens = lexer_11.get_tokens_with_regex()
for token in tokens:
    print(token)
```

```
ast = AST.AST.create_ast(tokens)
graph = Digraph()
build_graph(ast, graph)
graph.render('ast_graph', format='png', view=True)
AST.AST.create_ast_method_2(python_program)


python_program = '''
total = total + num
'''
lexer_11 = lexer.Lexer(python_program)
tokens = lexer_11.get_tokens_with_regex()
for token in tokens:
    print(token)

parser = parser_program.Parser(tokens)
parser.parse()
```

The main file orchestrates tokenization, AST generation, visualization, and parsing operations for language processing. It imports 'lexer', 'graphviz', 'AST', and 'parser_program' modules. Python code snippets undergo tokenization using the lexer module, with tokens printed for inspection. ASTs are generated via manual construction and 'ast' module parsing, then visualized using 'graphviz'. A second code snippet undergoes similar tokenization and parsing, with results printed to verify correctness. This comprehensive testing suite showcases the functionality and integration of language processing components.

Results:

Input:

```
    def sum_list(numbers):
        total = 0
        for num in numbers:
            total = total + num
        return total
```

Output (get_tokens_with_regex()):

```
('KEYWORD', 'def')
('IDENTIFIER', 'sum_list')
('SEPARATOR', '(')
('IDENTIFIER', 'numbers')
('SEPARATOR', ')')
('SEPARATOR', ':')
('IDENTIFIER', 'total')
('ASSIGNMENT', '=')
('NUMBER', '0')
('KEYWORD', 'for')
('IDENTIFIER', 'num')
('KEYWORD', 'in')
('IDENTIFIER', 'numbers')
('SEPARATOR', ':')
('IDENTIFIER', 'total')
('ASSIGNMENT', '=')
('IDENTIFIER', 'total')
('OPERATOR', '+')
('IDENTIFIER', 'num')
('KEYWORD', 'return')
('IDENTIFIER', 'total')
```

Example 1. Input:

```
    total = total + num
```

Output:

```
Parsing successful. The statement is correctly written.
```

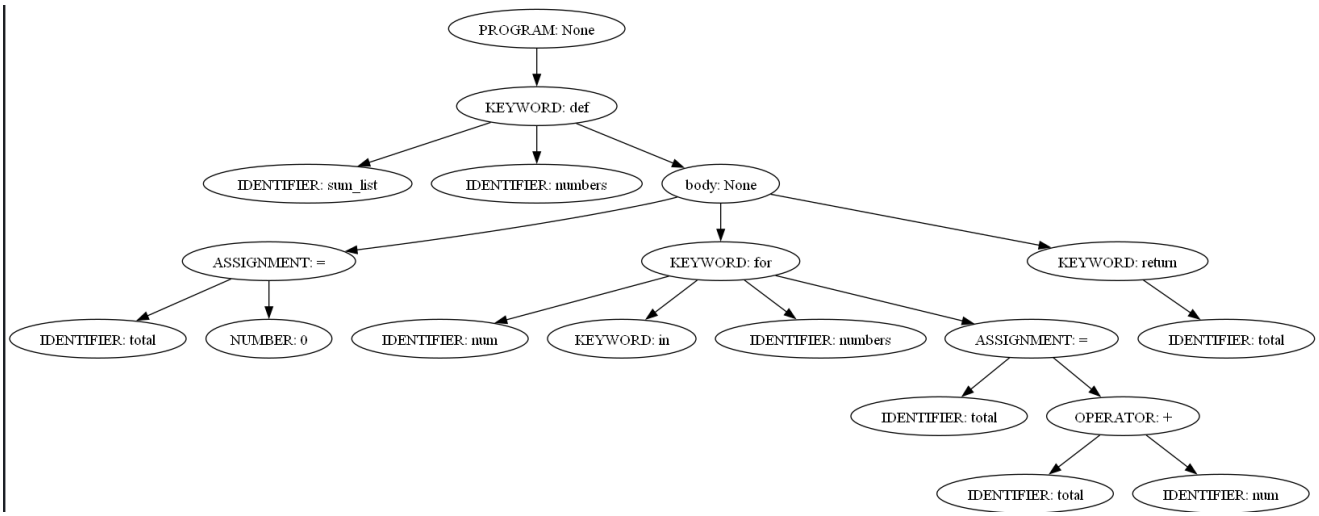
Example 2. Input:

```
total = + + num
```

Output:

```
Parsing failed. The statement is incorrectly written.
Expected IDENTIFIER or NUMBER, but found OPERATOR
```

AST created with the first method



AST created with the second method

```
Module(
    body=[
        FunctionDef(name='sum_list', args=arguments(posonlyargs=[],
args=[arg(arg='numbers')], kwonlyargs=[], kw_defaults=[], defaults=[]),
            body=[
                    Assign(targets=[Name(id='total', ctx=Store())],
value=Constant(value=0)),
                    For(target=Name(id='num', ctx=Store()), iter=Name(id='numbers',
ctx=Load()),
                        body=[Assign(targets=[Name(id='total', ctx=Store())],
value=BinOp(left=Name(id='total', ctx=Load()), op=Add(), right=Name(id='num',
ctx=Load())))], orelse=[]),
                    Return(value=Name(id='total', ctx=Load()))
                ], decorator_list=[])
        ], type_ignores=[]
    )
```

Conclusion:

In conclusion, the Abstract Syntax Tree (AST) emerges as a valuable tool for analyzing the structural elements of a programming language directly. It provides a clear representation of code syntax, aiding in understanding and manipulating program structures efficiently.

Furthermore, leveraging regular expressions in the lexer method significantly reduces its complexity and length, streamlining the tokenization process. This optimization enhances readability and maintainability of the lexer codebase. Parsing stands out as a pivotal aspect of programming languages, enabling code interpretation and execution. It plays a crucial role in transforming source code into a structured representation, facilitating subsequent analysis and execution.

Lastly, while both ASTs and parse trees serve to represent the syntax of a program, the key distinction lies in their node structure. Unlike parse trees, ASTs typically do not contain non-terminal nodes, resulting in a more streamlined representation of program syntax. Understanding these differences enhances comprehension of language processing techniques and aids in the development of efficient parsing algorithms.