Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Course: Formal Languages & Finite Automata

Laboratory work 6: Parser, AST Build.

Elaborated:

st. gr. FAF-221 Lupan Lucian

Verified:

asist. univ. Dumitru Crețu

Chişinău – 2024

Theory

What is a parser?

In computer technology, a parser is a program that's usually part of a compiler. It receives input in the form of sequential source program instructions, interactive online commands, markup tags or some other defined interface.

Parsers break the input they get into parts such as the nouns (objects), verbs (methods), and their attributes or options. These are then managed by other programming, such as other components in a compiler. A parser may also check to ensure that all the necessary input has been provided.

How does parsing work?

A parser is a program that is part of the compiler, and parsing is part of the compiling process. Parsing happens during the analysis stage of compilation.

In parsing, code is taken from the preprocessor, broken into smaller pieces and analyzed so other software can understand it. The parser does this by building a data structure out of the pieces of input.

More specifically, a person writes code in a human-readable language like C++ or Java and saves it as a series of text files. The parser takes those text files as input and breaks them down so they can be translated on the target platform.

The parser consists of three components, each of which handles a different stage of the parsing process. The three stages are:

1. Stage 1: Lexical analysis
2. Stage 2: Syntactic analysis
3. Stage 3: Semantic analysis

What is an AST?

An abstract syntax tree (AST) is a data structure used in computer science to represent the structure of a program or code snippet. It is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text. It is sometimes called just a syntax tree.

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:
    1. In case you didn't have a type that denotes the possible types of tokens you need to:
        1. Have a type ***TokenType*** (like an enum) that can be used in the lexical analysis to categorize the tokens.
        2. Please use regular expressions to identify the type of the token.
    2. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
    3. Implement a simple parser program that could extract the syntactic information from the input text.

Implementation

For this laboratory work, I continued what I did for my 3rd lab. Since I had already implemented the parser and the token types, I only had to work on an AST class that would generate an abstract syntax tree for the parsed expression.

To start off, it's best we take a look at the grammar, and its rules. Below, we can see the grammar which our lexer and parser is built on. We have expressions, terms and factors, which all denote different operations. Our grammar allows the usage of parentheses to group arithmetic expressions together.

```
expr: term ((PLUS|MINUS) term)*
term: factor ((MULT|DIV) factor)*
factor: INT|FLOAT
      : (PLUS|MINUS) factor
      : LPAREN expr RPAREN
```

After we defined our grammar, we need to take a look at the possible token types. Apart from the standard types from the grammar, we also have an EOF token, which is appended after the program is over.

```
TT_INT = 'INT'
TT_FLOAT = 'FLOAT'
TT_PLUS = 'PLUS'
TT_MINUS = 'MINUS'
TT_MULT = 'MULT'
TT_DIV = 'DIV'
TT_LPAREN = 'LPAREN'
TT_RPAREN = 'RPAREN'
TT_EOF = 'EOF'


9 usages  Lucian-Lu
class Token:
    Lucian-Lu
    def __init__(self, type_, value=None, pos_start=None, pos_end=None):
        self.type = type_
        self.value = value
```

Parsing is done by checking a character one by one. We also make use of expressions, terms and factors, as defined in the grammar above. Starting off, we define an advance function which increments the current token and checks if it's valid, until we reach the EOF token. If it isn't, we show an error the user.

```
class Parser:
    Lucian-Lu
    def __init__(self, tokens):
        self.tokens = tokens
        self.tok_idx = -1
        self.advance()

    8 usages (2 dynamic)  Lucian-Lu
    def advance(self):
        self.tok_idx += 1
        if self.tok_idx < len(self.tokens):
            self.current_tok = self.tokens[self.tok_idx]
        return self.current_tok

    1 usage  Lucian-Lu
    def parse(self):
        res = self.expr()
        if not res.error and self.current_tok.type != TT_EOF:
            return res.failure(InvalidSyntaxError(
                self.current_tok.pos_start, self.current_tok.pos_end,
                details: "Expected '+', '-', '*', or '/'"
            ))
        return res
```

After we've parsed the entire expression, we create another class that's dedicated to generating the abstract syntax tree. By passing it the parsed expression, we start off by splitting the expression using regex, using a pattern, splitting every token:

```
def split_expression(self):
    left_par_count = 0
    edges = []
    root = ''
    counters = {'PLUS': 0, 'MINUS': 0, 'MULT': 0, 'DIV': 0}
    pattern = r'(?:INT:\d+|FLOAT:\d+\.\d+|PLUS|MINUS|MULT|DIV|\(|\))'
    results = re.findall(pattern, self.expression)
```

After we split the expression, we start searching for the root, and adding the edges to the array. This process is rather straightforward, making use of our grammar's rules to parse the expression. The only interesting thing is that we need to attribute each of the operators an index, i.e., PLUS_2, in order to properly connect the edges and ensure that there's no problems while generating the graph. This is done via the use of a dictionary which keeps track of all the possible operation nodes, and incrementing them by 1 whenever they're used.

```
for i, result in enumerate(results):
    if result == '(':
        left_par_count += 1
    elif result == ')':
        left_par_count -= 1
    elif left_par_count == 1:
        if result in ['PLUS', 'MINUS', 'MULT', 'DIV']:
            root = f'{result}_{counters[result]}'
            counters[result] += 1
```

The abstract syntax tree is generated using the networkx library, which is composed of several functions, used for studying graphs and networks. By getting the root and edges using the aformentioned split_expression() function, we initialize a DiGraph and add the root and the edges to it. After that, we call a hierarchy_pos function (credits to Joel on StackOverflow), which organizes the graph in the shape of a tree, given a root node:

```
def generate_ast(self):
    root, edges = self.split_expression()
    G = nx.DiGraph()
    G.add_node(root)
    for edge in edges:
        G.add_edge(edge[0], edge[1])

    pos = self.hierarchy_pos(G, root)
    nx.draw(G, pos=pos, with_labels=True)
    plt.show()
```

Results

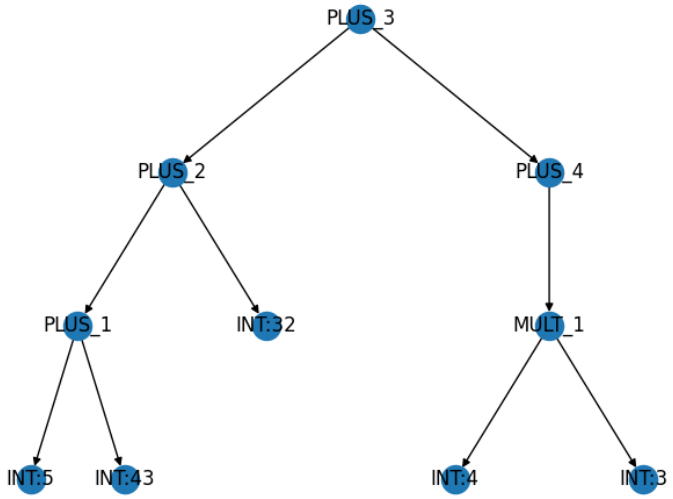
Input:

```
def run(fn, text):
    lexer = Lexer(fn, text)
    tokens, error = lexer.make_tokens()

    if error:
        return None, error.as_string()

    parser = Parser(tokens)
    ast = parser.parse()
    print(f'Parser result = {ast.node}')
    ast2 = AST(ast.node)
    ast2.generate_ast()

    return ast.node, ast.error


lexer = run( fn: "test",  text: "5 + 43 + 32 + ((4 * 3) + 3)")
```

Output (Screenshot + Text):

```
Parser result = (((INT:5, PLUS, INT:43), PLUS, INT:32), PLUS, ((INT:4, MULT, INT:3), PLUS, INT:3))
Split expression = ['(', '(', '(', 'INT:5', 'PLUS', 'INT:43', ')', 'PLUS', 'INT:32', ')', 'PLUS', '(', '(', 'INT:4', 'MULT', 'INT:3', ')', 'PLUS', 'INT:3', ')', ')']
Root = PLUS_3
Edges = [['PLUS_1', 'INT:5'], ['PLUS_2', 'PLUS_1'], ['PLUS_1', 'INT:43'], ['PLUS_3', 'PLUS_2'], ['PLUS_2', 'INT:32'], ['MULT_1', 'INT:4'], ['PLUS_4', 'MULT_1'], ['MULT_1', 'INT:3'], ['PLUS_3', 'PLUS_4']]
```

Parser result = (((INT:5, PLUS, INT:43), PLUS, INT:32), PLUS, ((INT:4, MULT, INT:3), PLUS, INT:3))
Split expression = ['(', '(', '(', 'INT:5', 'PLUS', 'INT:43', ')', 'PLUS', 'INT:32', ')', 'PLUS', '(', '(', 'INT:4', 'MULT', 'INT:3', ')', 'PLUS', 'INT:3', ')', ')']
Root = PLUS_3
Edges = [['PLUS_1', 'INT:5'], ['PLUS_2', 'PLUS_1'], ['PLUS_1', 'INT:43'], ['PLUS_3', 'PLUS_2'], ['PLUS_2', 'INT:32'], ['MULT_1', 'INT:4'], ['PLUS_4', 'MULT_1'], ['MULT_1', 'INT:3'], ['PLUS_3', 'PLUS_4']]



Conclusions

In this laboratory work, I expanded on the previously developed solution, created for the 3rd lab. Since I had already created the token types for the lexer and the parser, along with the grammar, I only had to create an abstract syntax tree function which would split the parsed expression and make use of the tokens in order to create nodes and edges.

In order to draw the graph, I had to make use of 2 Python libraries – matplotlib.pyplot and networkx. Using the latter, I've created an instance of a class, where I could freely define nodes and edges, as well as declaring the graph type, after which I've used the pyplot to show the graph as an output when running the program.

I didn't really have any troubles with this laboratory work, except for very small issue which took a bit of time to solve – I forgot to pass the @staticmethod attribute to the hierarchy_pos function (which displayed the graph as a tree), and as a result, I've had unexpected errors.

Overall, however, this laboratory work was much easier than the previous one. Since this is the last laboratory work, as feedback to the lab courses as a whole – they were great. I've had a lot of fun working on the laboratory works (with the exception of the first laboratory work, which had an NFA for my variant, before we've learnt about the algorithm for transforming an NFA to a DFA)

References:

1. Can one get hierarchical graphs from networkx with python 3? [online]. [accessed 29.04.2024]. Available: https://stackoverflow.com/questions/29586520/can-one-get-hierarchical-graphs-from-networkx-with-python-3/29597209#29597209
2. What is a Parser? Definition, Types and Examples [online]. [accessed 29.04.2024]. Available: https://www.techtarget.com/searchapparchitecture/definition/parser
3. Abstract Syntax Tree [online]. [accessed 29.04.2024]. Available: https://en.wikipedia.org/wiki/Abstract_syntax_tree