MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Mihalachi Mihai FAF-221

Report

Laboratory work no.6

of Formal Languages & Finite Automata

Chișinău – 2024

Theory

Parsing is a fundamental process in computer science and linguistics, serving as the bridge between raw textual data and structured representations that computers can understand and manipulate. At its core, parsing involves analyzing the syntactic structure of a piece of text according to a formal grammar. This process is particularly crucial in programming languages, where compilers or interpreters must transform human-readable code into machine-executable instructions.

Parsing typically operates in several stages. First, the input text is tokenized, breaking it down into fundamental units such as keywords, identifiers, operators, and literals. Next, these tokens are analyzed according to the grammar rules of the language to determine their syntactic structure. This phase often involves constructing a parse tree or, more commonly, an abstract syntax tree (AST), which represents the hierarchical structure of the parsed code, abstracting away details that are not relevant to its meaning or execution. Each node in the tree corresponds to a syntactic construct, such as a statement or an expression, and the edges represent the relationships between these constructs. By traversing this tree, compilers and interpreters can perform various tasks, such as type checking, optimization, and code generation. Overall, parsing plays a vital role in enabling computers to understand and process human-generated textual input, forming the foundation for many aspects of language processing, including programming language implementation, natural language processing, and more.

Objectives:

1. Get familiar with parsing, what it is and how it can be programmed;
2. Get familiar with the concept of AST;
3. In addition to what has been done in the 3rd lab work do the following:
    a. 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.
    b. Implement the necessary data structures for an AST that could be used for the text you have processed in the 3rd lab work.
    c. Implement a simple parser program that could extract the syntactic information from the input text.

Implementation description:

1. To implement a parser I created a class called Parser that I use to parse the given input text. It has several methods, like *parse(), expr(), factor(), advance(),* etc. Every single method will be explained. Besides this class I also have a Lexer class used to determine the lexer, error classes used to print errors in case some are found during parsing, and nodes classes that represent nodes in the abstract syntax tree (AST). Also there is a ParseResult class which prints the result of parsing operation. It has 3 methods: *register(),* which handles the result of a parsing operation. It checks if the input has an error, and if it has, it adds an error attribute to it, otherwise, it returns the node found in the input. Next one is *success().* This method is used to signal a successful parsing result. It takes a node representing the parsed AST node and sets the node attribute of the current result object to this node. The other method is *failure(),* which is used to signal a parsing failure due to an error. It takes an error representing the error encountered during parsing and sets the error attribute of the current result object to this error. Here is the ParseResult class code:

```
class ParseResult:
    def __init__(self):
        self.error = None
        self.node = None

    9 usages
    def register(self, res):
        if isinstance(res, ParseResult):
            if res.error:
                self.error = res.error
            return res.node

        return res

    4 usages
    def success(self, node):
        self.node = node
        return self

    3 usages
    def failure(self, error):
        self.error = error
        return self
```

Figure 1: ParseResult class in Python

2. Now let's talk about the Parser class and its methods. It starts with the *parse()* method, which initiates the parsing process by calling the *expr()* method, which parses an expression. It then checks if parsing was successful and if the end of the input (EOF token) was reached. If parsing is successful and the end of the input is reached, it returns the parsed AST (abstract syntax tree) node. Otherwise, it returns an error indicating that an unexpected token was encountered. Also, to advance the token index we use a method called *advance()*. Here is the code snippet for these methods:

```
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
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

2 usages
def expr(self):
    return self.bin_op(self.term,  ops: (TT_PLUS, TT_MINUS))
```

Figure 2: Basic methods for parsing in Python

3. The next three methods we use to parse factors, terms and binary operations. For the *factor()* method, I retrieve the token and check its type, then do the following:

    a. If the token is a unary operator (+ or -), it advances the token index, recursively parses the factor, and returns a UnaryOpNode with the unary operator token and the parsed factor node.

    b. If the token is an integer or float literal, it advances the token index and returns a NumberNode with the parsed token.

    c. If the token is a left parenthesis ((), it advances the token index, recursively parses an expression, and checks if the next token is a right parenthesis ()). If it is, it advances the token index again and returns the parsed expression. Otherwise, it returns an error indicating a missing right parenthesis.

    d. If none of the above conditions are met, it returns an error indicating an invalid syntax.

    Here is the code snippet for this method:

```
def factor(self):
    res = ParseResult()
    tok = self.current_tok
    if tok.type in (TT_PLUS, TT_MINUS):
        res.register(self.advance())
        factor = res.register(self.factor())
        if res.error:
            return res
        return res.success(UnaryOpNode(tok, factor))
    elif tok.type in (TT_INT, TT_FLOAT):
        res.register(self.advance())
        return res.success(NumberNode(tok))
    elif tok.type == TT_LPAREN:
        res.register(self.advance())
        expr = res.register(self.expr())
        if res.error:
            return res
        if self.current_tok.type == TT_RPAREN:
            res.register(self.advance())
            return res.success(expr)
        else:
            return res.failure(InvalidSyntaxError(
                self.current_tok.pos_start, self.current_tok.pos_end,
                details: "Expected ')'"
            ))
    return res.failure(InvalidSyntaxError(
        tok.pos_start, tok.pos_end,
        details: "Expected int or float"
    ))
```

Figure 3: factor method in Python

4. Since term is a sequence of factors separated by multiplication or division operators, I uses the *bin_op()* method to parse the term, passing the *factor()* method as the parsing function and a tuple containing the multiplication and division token types as the allowed operators. *bin_op()* method parses a binary operation. It takes two parameters: func, which is a parsing function for the operands, and ops, which is a tuple containing the allowed operator token types. It first parses the left operand using the provided parsing function (func). Then, it enters a loop where it checks if the current token is one of the allowed operators. If it is, it advances the token index, parses the right operand using the same parsing function, and creates a BinOpNode with the left and right operands and the operator token. The loop continues until no more allowed operators are found. Finally, it returns the parsed AST node representing the binary operation. Here are the code snippets for these two functions:

```
def term(self):
    return self.bin_op(self.factor, ops: (TT_MUL, TT_DIV))

2 usages
def bin_op(self, func, ops):
    res = ParseResult()
    left = res.register(func())
    if res.error:
        return res

    while self.current_tok.type in ops:
        op_tok = self.current_tok
        res.register(self.advance())
        right = res.register(func())
        if res.error:
            return res
        left = BinOpNode(left, op_tok, right)

    return res.success(left)
```

Figure 4: term and bin_op methods in Python

Conclusion

In conclusion, the laboratory work focused on creating a custom parser has been a valuable learning experience, offering insights into the intricate process of transforming raw textual data into structured representations. Throughout the project, we delved into fundamental concepts of parsing, including lexical analysis, syntactic analysis, and abstract syntax tree construction.

By implementing our own parser, we gained a deeper understanding of how programming languages and other formal languages are interpreted and executed by computers. We explored the role of grammars in defining language syntax, the importance of tokenization in breaking down input text into meaningful units, and the process of constructing abstract syntax trees to represent parsed expressions.

Moreover, the laboratory work provided hands-on experience in writing code to handle various parsing tasks, such as recognizing different types of tokens, handling operator precedence and associativity, and detecting and reporting syntax errors. Through iterative development and testing, we honed our problem-solving skills and gained proficiency in debugging and error handling.

Overall, the laboratory work not only equipped us with practical parsing skills but also fostered a deeper appreciation for the complexity and elegance of language processing. It served as a foundation for further exploration into topics such as compiler design, natural language processing, and advanced parsing techniques, laying the groundwork for future academic and professional endeavors in the field of computer science.