MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

Laboratory work 6:
Formal Languages and Finite Automata

Elaborated:
st. gr. FAF-223
Vieru Mihai

Verified:
asist. univ.
Cretu Dumitru

Chișinău, 2024

Content

Parser

To understand the parser, I will explain only a part of my program, which is literal parsing.

This parser builds the AST using recursive descent and a lookahead to determine the parsing method.

Parser:

```
from Tokenizer import *


class Parser:
    def parse(self, string):
        self._string = string
        self._tokenizer = Tokenizer(string)

        self._lookahead = self._tokenizer.getNextToken()

        return self.Program()
```

Next, we break apart the strings by declaring the non-terminals using Python methods. The ”Program” method/non-terminal is the main non-terminal from which the parsing starts. It consists of a statement list.

Program:

```
def Program(self):
        return {
            "type": "Program",
            "body": self.StatementList()
        }
```

The statement list is a list of all the statements made inside the program. A statement is simply a string terminated by ”;”. We access all statements by iterating through all tokens until there are none left.

AdvanceToken:

```
# StatementList : Statement | StatementList Statement ;
    def StatementList(self):
        statementList = []
```

```
        while self._lookahead != None:
            if self._lookahead["type"] == "}":
                break
            statementList.append(self.Statement())

        return statementList

    # Statement : ExpressionStatement ;
    def Statement(self):
        return self.ExpressionStatement()

    # ExpressionStatement : Expression ';' ;
    def ExpressionStatement(self):
        expression = self.Expression()
        self._eat(";")
        return {
            "type": "ExpressionStatement",
            "expression": expression
        }
```

As you can see, we use the _eat method to determine that the next token is indeed a ";", so we can determine the end of an expression. What _eat does, is advance the lookahead by one token, but does this safely by checking that our current token is not null and is the expected token.

ExpressionParsing:

```
def _eat(self, tokenType):
    token = self._lookahead

    if token == None:
        raise SyntaxError(f"Unexpected end of input, expected {tokenType}.")

    if token["type"] != tokenType:
```

```
            print(self._string[self._tokenizer._coursor],
             ↪  self._tokenizer._coursor, self._lookahead)
            val = token["value"]
            raise SyntaxError(f"Unexpected token {val}, expected {tokenType}.")

        self._lookahead = self._tokenizer.getNextToken()

        return token
```

Now, let's discuss how we parse an expression. For our current architecture, we only have literals, but typically, we would have to declare what our language is made of, e.g., IfStatement, ForLoopStatement, Block, etc. However, in this case, it consists only of literals, which can be of two types of non-terminals: NumericLiteral and StringLiteral. We pick which one to parse based on the type of the lookahead. Parsing the literal is simply advancing to the next lookahead by checking that the current one is indeed of the correct type, using the `eat` method.

Program:

```
    # Expression : Literal ;
        def Expression(self):
            return self.Literal()

        # Literal : NumericLiteral | StringLiteral ;
        def Literal(self):
            match self._lookahead["type"]:
                case "STRING": return self.StringLiteral()
                case "NUMBER": return self.NumericLiteral()

        # NumericLiteral : NUMBER ;
        def NumericLiteral(self):
            token = self._eat("NUMBER")
            return {
                "type": "NumericLiteral",
                "value": int(token["value"])
```

```
        }

    # StringLiteral : STRING ;
    def StringLiteral(self):
        token = self._eat("STRING")
        return {
            "type": "StringLiteral",
            "value": token["value"]
        }
```

In the end, all of these dictionaries pop back up on the stack, and the final result looks something like this.

For the input:

10;

”Hello Dumitru”;

AST:

```
{
  "type": "Program",
  "body": [
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "NumericLiteral",
        "value": 10
      }
    },
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "StringLiteral",
        "value": ""Hello Dumitru""
      }
```

```
        }
      ]
    }
```

Conclusion:

Through this lab, I implemented a parser for a small programming language, using recursive descent parsing and lookahead techniques. By dissecting the process of parsing literals, such as numeric and string literals, I gained insight into the fundamental workings of a parser. This exercise allowed me to understand how to construct abstract syntax trees (ASTs) from parsed tokens, providing a structural representation of the program.

Moreover, by implementing methods to handle non-terminals and advancing tokens, I learned practical techniques for building parsers in programming languages like Python. These included error handling mechanisms like checking for null tokens and ensuring expected tokens are consumed. Additionally, I grasped the importance of clear code organization and documentation, evident in the use of Python methods to declare non-terminals and comments to explain parsing logic.

Overall, this lab enhanced my understanding of parsing techniques and their application in language processing tasks. I gained valuable experience in implementing parsers, which can be extended to handle more complex grammars and language features in future projects.