

**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică**

**Ciumacenco Pavel FAF-232**

**Report**

*Laboratory work nr.6*

***Course: Formal Languages & Finite Automata***

Checked by:

**Cretu Dumitru,** *university prof.*

FCIM, UTM

**Chișinău – 2025**

**Theory:**

### 1 Parsing: Definition and Implementation

Parsing is a fundamental process in computer science and software engineering that involves analyzing a sequence of symbols, either in natural language, computer languages, or data structures, to determine its grammatical structure according to a given formal grammar. In the context of programming languages, parsing is a crucial step in the compilation or interpretation process, as it converts linear code into a structured format that a computer can process and evaluate.

A parser is a program or a component of a compiler that reads input text (commonly referred to as the source code) and converts it into a data structure that reflects the hierarchical structure of the language constructs. This data structure is often referred to as a parse tree or an Abstract Syntax Tree (AST).

**Types of Parsers:**Parsers are generally classified based on the direction in which they read the input and how they construct the parse tree. Common types include:

* **Top-Down Parsers:** These parsers start from the highest level of the parse tree (the start symbol) and work their way down to the leaves (the input symbols). Examples include Recursive Descent Parsers and LL parsers.
* **Bottom-Up Parsers:** These start from the input symbols and work upwards towards the start symbol. LR parsers are a well-known example.

**Parser Implementation:**Programming a parser typically involves defining the grammar of the language in a formal notation such as Backus-Naur Form (BNF) or Extended BNF (EBNF), and then implementing a parser that can recognize this grammar. Parser generators such as ANTLR, Bison, and YACC can automate this process by generating parser code from grammar definitions.

In modern programming, parsers can be implemented in various languages such as Python, Java, or C++, often by utilizing built-in libraries or external modules specifically designed for parsing tasks.

### 2 Abstract Syntax Tree (AST)

An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node in the tree denotes a construct occurring in the source code. Unlike a parse tree, an AST omits unnecessary syntactic details such as parentheses and semicolons and focuses on the logical structure of the code.

**Purpose of ASTs:**ASTs serve as an intermediate representation of the source code, which is easier to analyze and manipulate than the raw text or a parse tree. They are widely used in:

* **Compilers:** For syntax analysis, semantic analysis, optimization, and code generation.
* **Interpreters:** To evaluate expressions or execute statements.
* **Code Analysis Tools:** For static code analysis, refactoring, or code transformation.

**Structure of an AST:**In an AST, each node represents a language construct. For instance:

* A binary operation like a + b would have a node for the + operator, with a and b as child nodes.
* A function call would have a node for the call, with the function name and arguments as children.

**Generating ASTs:**ASTs can be generated as a part of the parsing process. Some parser generators or parsing libraries automatically construct an AST while parsing the source code. In languages like Python, the built-in ast module provides functionality to parse Python source code into its AST form, allowing developers to traverse and manipulate the structure programmatically.

**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 Description:**

## 1️ Imports and Dependencies

import sys

from enum import Enum

from typing import List, Union, Optional

**Explanation:** This section imports standard Python libraries. sys is used for system-specific parameters and functions (like writing to stderr). Enum allows the creation of enumerated constants, and typing provides type hints for better code clarity and error checking.

## 2️ TokenType Enum Definition

class TokenType(Enum):

TOKEN\_EOF = 0

TOKEN\_NUMBER = 1

...

TOKEN\_DIVIDE = 18

**Explanation:** This Enum defines various token types the lexer can identify — like numbers, operators, brackets, and keywords. Each token type is assigned a unique integer for identification during parsing.

## 3️ NodeType Enum Definition

class NodeType(Enum):

PROGRAM = 0

BINARY\_EXPR = 1

...

ASSIGNMENT = 6

**Explanation:** Similarly, this Enum categorizes the kinds of nodes that can appear in the abstract syntax tree (AST). Each kind corresponds to a language construct like a binary expression or a while loop.

## 4️ Token Class

class Token:

def \_\_init\_\_(self, type: TokenType, value: str):

self.type = type

self.value = value

**Explanation:** This class represents individual tokens produced by the lexer. Each token has a type (from TokenType) and an associated string value.

## 5️ Lexer Class

**Overview:**The Lexer turns raw source code text into a stream of Token objects.

**Important methods:**

* \_\_init\_\_ — Initializes the lexer with input text and sets the current character.
* advance / retreat — Moves the current character index forward or backward.
* skip\_whitespace — Skips over whitespace and newlines.
* collect\_id — Gathers identifier names (like x or while).
* get\_next\_token — Core lexer function that identifies the next token, handling numbers, operators, and keywords.

**Explanation:** The Lexer reads the input text one character at a time and uses different helper functions to classify characters into meaningful tokens for the parser.

## 6️ AST Node Classes

**Base Classes:**

class Expr:

def \_\_init\_\_(self, kind: NodeType):

self.kind = kind

* Expr represents any kind of expression node (like arithmetic or comparison operations).

class Stmt:

def \_\_init\_\_(self, kind: NodeType):

self.kind = kind

* Stmt is the base class for any statement node (like a while loop or assignment).

**Subclasses:**

* BinaryExpr, Identifier, NumericLiteral, ConditionalExpr, AssignmentExpr — represent specific expression types like arithmetic, variables, numbers, conditionals, and assignments.
* WhileStmt — represents a while loop statement.
* Program — acts as the root of the AST, holding a list of statements.

**Explanation:** These classes collectively define the structure of the AST by modeling each type of expression and statement. Each has its own data members and string representation for easier debugging.

## 7️ Parser Class

**Overview:**The Parser reads tokens from the lexer and builds the AST.

**Important methods:**

* \_\_init\_\_ — Initializes the parser with a lexer and retrieves the first token.
* advance / eat — Move through tokens and ensure expected token types.
* produce\_ast — Main loop to build the program's AST.
* parse\_stmt — Decides what kind of statement to parse next (like a while loop or assignment).
* parse\_while — Parses while loops.
* parse\_expr — Parses arithmetic or conditional expressions.
* parse\_additive\_expr, parse\_multiplicative\_expr, parse\_primary\_expr — Handle operator precedence by breaking expressions into layers (like multiplication before addition).
* parse\_conditional\_expr — Parses conditions inside while statements.

**Explanation:** The parser reads tokens one at a time and builds a structured AST by calling specialized functions for different language constructs. It uses recursion and operator precedence to correctly parse nested and complex expressions.

## 8️ main() Function

def main():

test\_program = """

while (x > 5) {

y = y + 1;

}

"""

lexer = Lexer(test\_program)

parser = Parser(lexer)

program = parser.produce\_ast()

print("=== Generated AST ===")

print(program)

**Explanation:** This is the test harness that defines a small program string, initializes the lexer and parser, generates the AST, and prints it to the console.

## 9️ Entry Point Check

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Explanation:** This line ensures that the main() function runs only when the script is executed directly, not when imported as a module.

**4.Conclusions.Screenshots.Results.**

In this report, I explored the fundamental concepts of **parsing** and **abstract syntax trees (ASTs)**, and how they can be implemented programmatically. Parsing is a crucial step in the process of interpreting or compiling programming languages, where source code is analyzed and converted into a structured format that a computer can understand. The abstract syntax tree (AST) is one of the most common ways to represent this structure, providing a tree-like hierarchy of nodes that correspond to different language constructs such as expressions, statements, and operators.

To better understand these concepts, I implemented a simple parser and lexer in Python. The lexer was responsible for reading the input source code and breaking it down into a sequence of meaningful tokens, such as numbers, identifiers, operators, and punctuation. The parser then processed these tokens according to a set of rules to build an AST, reflecting the logical structure of the original code.

The code supported basic programming features like arithmetic expressions, assignments, while loops, and conditional comparisons. Through this practical exercise, I gained hands-on experience with how a parser handles operator precedence, constructs nested expressions, and validates syntax using recursive descent parsing techniques.

Overall, this project helped solidify my understanding of how source code can be transformed into a structured and navigable format using lexers and parsers, and how an AST serves as a bridge between high-level code and lower-level execution. It demonstrated the importance of designing clean, modular components for language processing and gave me a deeper appreciation for the internal workings of interpreters and compilers.

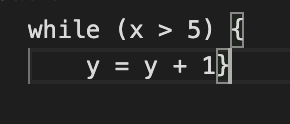


Figure 2: Input image

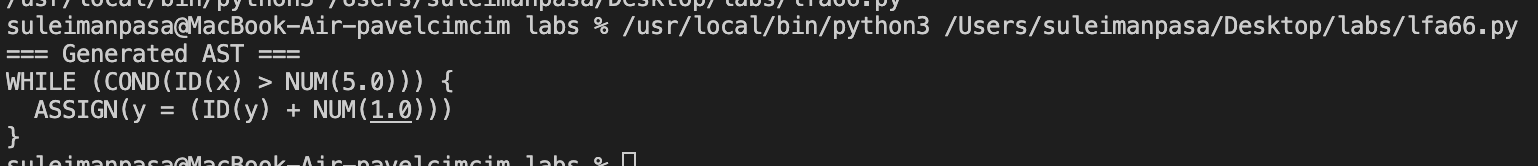


Figure 3: Results from console image