Parsing and building an AST

Parfene Daniel, FAF-222

April 29, 2024

1 Introduction

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.

2 Implementation Overview

The algorithm is implemented using numerous other parts which include the Lexer, Parser, Nodes classes etc. Most important parts of implementing an AST are:

1. Building the Lexer
2. Building the Parser
3. Defining Nodes
4. Building the Tree

3 Algorithm Details

3.1 Building the Lexer

Lexical tokenization is conversion of a text into (semantically or syntactically) meaningful lexical tokens belonging to categories defined by a "lexer" program. In case of a natural language, those categories include nouns, verbs, adjectives, punctuations etc. In case of a programming language, the categories include identifiers, operators, grouping symbols and data types.

```

class Lexer:

### PART OF CODE WHICH DEFINES THE TYPE OF TOKEN
    def make_token(self):
        if self.line.next() in "()|_^":
            return self.make_punctuator()

        if self.line.next() in "~+-*/%":
            return self.make_operator()

        if self.line.next() in "0123456789.":
            return self.make_number()
```

3.2 Building the Parser

Parsing, syntax analysis, or syntactic analysis is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar.

```

class Parser:

### PARSING THE EXPRESSION USING BASIC METHODS

    def next(self):
        return self.tokens[self.i]

    def take(self):
        token = self.next()
        self.i += 1
        return token

    def expecting_has(self, *strings):
        if self.next().has(*strings):
            return self.take()

        raise ParserError(self.next(), f"Expecting has {strings}")

    def expecting_of(self, *kinds):
        if self.next().of(*kinds):
            return self.take()

        raise ParserError(self.next(), f"Expecting of {kinds}")
```

3.3 Defining Nodes

A node is a basic unit of a data structure, such as a linked list or tree data structure. Nodes contain data and also may link to other nodes. Links between nodes are often implemented by pointers. In graph theory, the image provides a simplified view of a network, where each of the numbers represents a different node.

```

class PrimaryNode(Node, ABC):
    def __init__(self, token):
        self.token = token

    def nodes(self):
        return [self.token]

    def tree_repr(self, prefix = " " * 4):
        return f"{type(self).__name__}        {self.token}"


class BinaryNode(Node, ABC):
    def __init__(self, left, op, right):
        self.left = left
        self.op = op
        self.right = right

    def nodes(self):
        return [self.left, self.op, self.right]
```

3.4 Building the Tree

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.

```

### METHOD WHICH BUILD A BINARY TREE FIRST

    def construct_binary(cls, parser, make, part, ops):
        node = part.construct(parser)

        while parser.next().has(*ops):
            op = parser.take()
            right = part.construct(parser)
            node = make(node, op, right)

        return node

### METHOD WHICH CREATES NODES BASED ON TOKENS AND BUILDS THE TREE

    def make_tree(self, tokens):
        self.tokens, self.i = tokens, 0
        node = Expression.construct(self)
```

```
19
20          if self.next().has("EOF"):
21              return node
22
23  ### TREE REPRESENTATION
24  #
25  #
26          for i, node in enumerate(nodes):
27              at_last = (i == len(nodes) - 1)
28              symbol = "          " if at_last else "          "
29              prefix_symbol = "" if at_last else "   "
30
31              node_string = node.tree_repr(f"{prefix}{prefix_symbol
    }{' ' * 4}")
32              string += f"\n{prefix}{symbol} {node_string}"
33
34  #
35  #
```

4 Code Usage

Abstract Syntax Trees (ASTs) are used in various fields within computer science and software engineering. Some common applications include:

1. Compiler Construction: Represent code structure for compilation.
2. Code Analysis: Detect errors, format code.
3. Code Transformation: Refactor, translate code.
4. IDE Features: Provide syntax highlighting, code completion.
5. Language Tooling: Develop interpreters, transpilers.
6. Program Understanding: Aid in code comprehension.

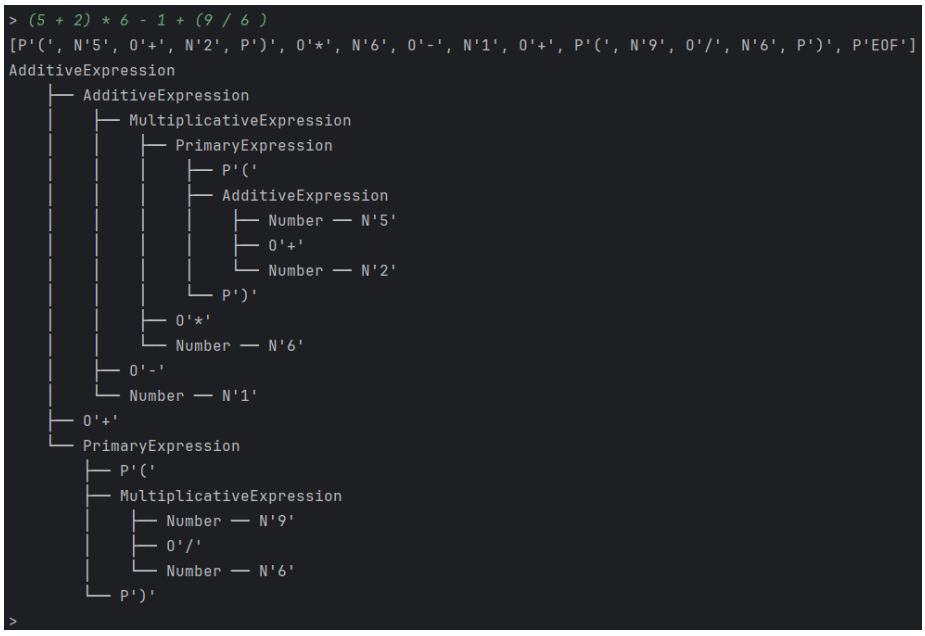

Figure 1: AST based on mathematical expression

5 Conclusion

The construction of Abstract Syntax Trees (ASTs) in this laboratory work has provided valuable insight into the internal representation of source code. By implementing algorithms to parse and construct ASTs, we have gained a deeper understanding of how programming languages structure their syntax and semantics.

Through this process, we have learned not only about the syntax rules of the language but also about the hierarchical relationships between different elements of the code. This knowledge is foundational for various applications in software development, including compiler construction, code analysis, and program transformation.

Moreover, building ASTs has reinforced key programming concepts such as recursion, tree traversal, and data abstraction. These concepts are not only essential for working with ASTs but also have broader applicability in computer science and software engineering.

In conclusion, the laboratory work on building ASTs has been an enriching experience, providing practical exposure to fundamental concepts in programming languages and compiler construction. The understanding gained from this exercise will undoubtedly serve as a solid foundation for future exploration in software development and related fields.