Parser & Building an Abstract Syntax Tree

Course: Formal Languages & Finite Automata

Author: Daniela Vornic

Theory

Parsing is a fundamental process in computer science and language processing where a sequence of characters (such as the text of a program) is analyzed to determine its grammatical structure with respect to a given formal grammar [1]. Parsing is crucial for a variety of applications including compilers, interpreters, and data processing tools.

Parsing involves two primary stages: lexical analysis and syntactic analysis. **Lexical analysis** (performed by a **lexer**) breaks down the input into a series of tokens, which are sequences of characters with a collective meaning. **Syntactic analysis** (performed by a **parser**) then organizes these tokens into a hierarchical structure that reflects the grammar of the language.

Implementing a parser can be done manually, where a developer writes code to handle each expected pattern in the input. However, it's more common to use a parser generator like **PLY (Python Lex-Yacc)** [2], which allows for defining grammar rules declaratively. PLY takes these rules and generates Python code that can parse texts according to the defined grammar. This greatly simplifies the development of parsers, ensuring accuracy and efficiency.

An Abstract Syntax Tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language [3]. Each node in the tree denotes a construction occurring in the source code. The abstract nature of the syntax tree is due to its omission of certain syntactic details which are not relevant for the analysis or transformation of the source code.

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:
    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

Grammar modifcations

I had to make some changes to improve the robustness of the grammar and the parser, such as:

- Remove the `&&` (and) operator as it seemed redundant for the image processing Domain-Specific Language (DSL).
- Rename the `--img` flag to `--target` to better reflect its purpose, given that it can specify both single images and directories.

The lexer was updated accordingly to reflect these changes. Furthermore, it already included types for tokens as well as regular expressions, given that PLY was used.

AST Definitions

In the context of this image processing DSL, the AST represents the parsed commands as tree-like structures. Below are the class definitions that form the backbone of our AST for the DSL.

The `ASTNode` class is the foundational building block for all other nodes within the AST. It is designed to be a superclass from which all other specific node types inherit. It encapsulates the most basic property shared by all nodes – a name.

```
class ASTNode:
    def __init__(self, name):
        self.name = name
```

The `StartCommand` node acts as the root of the AST for any given valid command. It contains references to the initial command (`imp`), the target flag specifying the image or folder to act upon, and a sequence of subsequent commands.

```
class StartCommand(ASTNode):
    def __init__(self, command, target_flag, command_sequence):
        super().__init__(command)
        self.target_flag = target_flag
        self.command_sequence = command_sequence
```

The `TargetFlag` node represents a flag within our DSL that specifies the target image files of an operation. The node holds a flag identifier and a value indicating the path to the target.

```
class TargetFlag(ASTNode):
    def __init__(self, flag, path):
        super().__init__(flag)
        self.value = path
```

The `Command` node defines a specific operation to be performed, such as resizing an image. It stores the command name, an optional list of flags that modify the command, and optionally, a reference to the next command in sequence if the current command is part of a pipeline.

```
class Command(ASTNode):
    def __init__(self, command, flags=None, next_command=None):
        super().__init__(command)
        self.flags = flags or []
        self.next_command = next_command
```

The `Flag` node is used to represent a single flag associated with a command, encapsulating both the flag's name and its value. This structure allows the AST to precisely represent the settings and parameters that each command in the DSL may require.

```
class Flag(ASTNode):
    def __init__(self, flag, value):
        super().__init__(flag)
        self.value = value
```

Parser

For this task, I continued using PLY, a Python implementation of `lex` and `yacc` parsing tools. Under the hood, PLY works by compiling the parsing-related method calls into a parsing table and then using this table to parse the input text. PLY employs an LALR (Look-Ahead Left to Right) parsing algorithm, which is efficient and well-suited for most parsing needs. Upon encountering a sequence of tokens that match a particular grammar rule, PLY executes the associated Python code. This usually involves constructing nodes of an AST (althoooough the result can be a simple tuple or any other data structure), which provides a hierarchical structure of the parsed code.

In PLY, parser functions are defined using a specific convention where each function's docstring contains a grammar rule and the function body defines what to do when that rule is matched. The naming convention `p_` for function names is a requirement by PLY to identify parser functions. Grammar rules are specified in the docstrings of functions. Each rule begins with the name of the rule followed by a colon and then the pattern that it matches. The `'''command : COMMAND'''` rule, for example, specifies that whenever the parser recognizes a token of type `COMMAND`, it should execute the corresponding function. Below there are the parser functions that were implemented for the image processing DSL.

The root command of our DSL scripts is captured by the `start_command` parsing rule. When PLY recognizes a start command followed by a target flag and a command sequence, it constructs a `StartCommand` node.

```
def p_start_command(p):
    '''start_command : START_COMMAND target_flag command_sequence'''
    p[0] = StartCommand(p[1], p[2], p[3])
```

The `target_flag` rules create nodes that specify the an image or a directory, creating either an `ImageFlag` or a `FolderFlag` node depending on the input.

```
def p_img_flag(p):
    '''target_flag : FLAG EQUALS IMAGE_PATH'''
    p[0] = TargetFlag(p[1], p[3])

def p_folder_flag(p):
    '''target_flag : FLAG EQUALS FOLDER_PATH'''
    p[0] = TargetFlag(p[1], p[3])
```

The `command_sequence` rule identifies individual commands or chains of commands linked by a pipeline operator. The `pipelined_command` rule explicitly handles the case where a command is followed by a `PIPELINE` operator.

```
def p_command_sequence(p):
    '''command_sequence : command
                        | pipelined_command'''
    p[0] = p[1]

def p_pipelined_command(p):
    '''pipelined_command : command PIPELINE command_sequence'''
    p[0] = p[1]
    p[0].next_command = p[3]
```

To ensure that a pipeline is always followed by a command sequence, I defined a specific error rule, `pipelined_command`, which prints a tailored error message if this condition is not met.

```
def p_pipeline_error(p):
    '''pipelined_command : command PIPELINE error'''
    print(f"PIPELINE operator must always be followed by a command
sequence, but '{p[3].value}' found")
    p[0] = p[1]  # Continue with the valid command part for possible
recovery
```

The `command` rules construct `Command` nodes. `Commands` may stand alone or be accompanied by a sequence of flags (options). The `flag_sequence` rule assembles a list of `Flag` nodes associated with a command.

```
def p_command(p):
    '''command : COMMAND'''
    p[0] = Command(p[1])

def p_command_with_flags(p):
    '''command : COMMAND flag_sequence'''
    p[0] = Command(p[1], p[2])
```

The `value` rule parses numbers, image formats, or image paths, which are the basic value types in this DSL.

```
def p_value(p):
    '''value : NUMBER
             | IMG_FORMAT
             | IMAGE_PATH'''
    p[0] = p[1]
```

The `p_error` function is called by PLY when a syntax error is detected. It prints an error message, which could be adapted to handle errors more gracefully, such as by attempting to recover or by providing suggestions to the user.

```
def p_error(p):
    if p is not None:
        print(f"Syntax error at token '{p.value}'")
    else:
        print("Syntax error at EOF")
```

Finally, we create the parser instance using `yacc.yacc()`. This parser is then ready to be used to parse input strings and construct the corresponding ASTs.

Visualization

Visualization is a powerful tool for understanding and debugging ASTs. As such, I created the `render_ast_diagram` function in `utils.py`. This function recursively traverses the AST and creates nodes and edges in a Graphviz `Digraph` object to represent the tree structure.

When `render_ast_diagram` is invoked, it first checks if the provided `node` is an instance of `ASTNode`. If so, a label is generated that combines the type of the node and its name. This is what will be displayed on the graph:

```
label = f'{node.__class__.__name__}: <B>{node.name}</B>'
```

This label is then used to create a Graphviz `node`. If the `node` has an attribute `value`, which typically represents the value of a flag, this information is appended to the label.

Using `id(node)` ensures each node is given a unique identifier in the graph, allowing Graphviz to distinguish between different instances of nodes that may share the same name:

```
node_id = str(id(node))
graph.node(node_id, label=f'<{label}>')
```

If the current `node` is not the root node (`parent` is not `None`), an edge is drawn from the parent to the current node, thereby constructing the hierarchical relationships between nodes.

The function then recursively processes each child of the current `node`. It looks for known attributes that could store children, such as `target_flag`, `command_sequence`, `commands`, `flags`, and `next_command`:

```
for attr in ['target_flag', 'command_sequence', 'commands', 'flags',
'next_command']:
    child = getattr(node, attr, None)
    # ... (Recursively call render_ast_diagram for each child)
```

For nodes that represent leaf elements, such as strings or numbers, these are directly added to the graph without children.

In the event of an error token (`p` is `None`), the function emits a generic syntax error message.

Conclusions / Results

Through the implementation of a parser using PLY and the definition of a structured AST, I've' gained insight into how grammatical rules of a language can be transformed into a programmatically manipulable format. The construction of the AST and its subsequent visualization using Graphviz allowed to see the hierarchical nature of language constructs, facilitating a deeper comprehension of the parsing process and error handling within the chosen DSL.

The source code is available in the `/src` folder of this directory, with the detailed results documented in the Jupyter Notebook titled `main.ipynb`. This interactive notebook includes code snippets and execution outputs that demonstrate the implementation of the AST and parser.

Bibliography

[1] PLY (Python Lex-Yacc)

[2] Parsing Wiki

[3] Abstract Syntax Tree Wiki