Lexer & Scanner

Course: Formal Languages & Finite Automata

Author: Daniela Vornic

Theory

Lexical analysis is a fundamental aspect of language processing, whether in the context of programming languages, domain-specific languages (DSLs), or any formal language. It involves the process of converting a sequence of characters into a sequence of **tokens**, which are the meaningful units of a language. These tokens typically represent identifiers, keywords, symbols, operators, and literals that conform to the syntax of the language being analyzed.

At the heart of lexical analysis lies the use of **regular expressions**, a powerful tool for pattern matching and text processing. Regular expressions define the patterns for tokens in a way that is both concise and expressive, allowing lexical analyzers, or lexers, to efficiently recognize and categorize the various components of the input text.

The role of a lexical analyzer, or lexer, is analogous to breaking down a sentence into words and punctuation in natural language processing. Just as understanding a sentence requires recognizing the words and their functions, understanding a program or command in a DSL involves recognizing the tokens and their roles within the syntax of the language. Lexers are usually the first phase in a compiler or interpreter pipeline.

Role of a lexer for DSLs

For this laboratory work, I chose to focus on implementing a lexer for an image processing DSL. The DSL is designed to define image transformations using a concise and expressive syntax. The lexer will be responsible for recognizing the tokens that make up the DSL, such as filter names and parameters. Given the specialized nature of a DSL, it's often beneficial to encapsulate the definitions of tokens and lexing rules within the same file or module for clarity and maintainability.

These are some examples of valid inputs for this DSL:

```
imp convert --img="path/to/image.jpg" --format="png"
```

```
imp --img="path/to/image.png" -> resize --w=200 --h=200 ->
convert --format="jpg"
```

Overview of PLY

For this implementation, I chose to use the **Python Lex-Yacc** (PLY) library [1], which provides a high-level interface for implementing lexers and parsers. It is is a powerful lexing and parsing tool inspired by the capabilities and functionality of traditional Unix tools Lex and Yacc. However, unlike its Unix counterparts, PLY is designed with Python's conventions and programming idioms in mind, making it an intuitive choice for Python developers working on compilers, interpreters, or domain-specific languages.

PLY consists of two main components: `lex.py` and `yacc.py`, which correspond to the lexer (lexical analyzer) and parser components, respectively.

- `lex.py`: This module is responsible for breaking down the raw input text into a stream of tokens based on a set of defined regular expression patterns. Each token represents a meaningful unit within the language, such as keywords, identifiers, literals, or operators.

- `yacc.py`: Building upon the token stream produced by the lexer, the parser module analyzes the tokens' structure to interpret the input's meaning. It does this by applying grammatical rules defined by the developer.

This is how the lexer component works in PLY:

1. When PLY's lexer is initialized (via `lex.lex()`), it introspects the user-defined token rules (functions with `t_` prefixes and `t_ignore`) and compiles their regular expression patterns into a form that Python's `re` module can execute efficiently.

2. PLY supports **state-based lexing**, allowing the lexer to change its behavior based on the current context (or state) of the input being scanned.

3. As the lexer scans the input text, it uses the compiled regular expressions to match substrings of the input. When a match is found, the corresponding token rule function is invoked. This function can modify the token, set its type, or perform other actions before the token is returned to the caller.

4. If the lexer encounters input that doesn't match any token rule, it calls the `t_error` function, if defined. This function can handle the error, skip the problematic input, or perform other error recovery actions.

5. The lexer generates tokens sequentially from the input text. Each token contains information about its type, value, and position in the input. These tokens are then consumed by the parser for syntactic analysis.

As such, PLY's lexer is a versatile tool for recognizing and classifying tokens based on regular expression patterns. In the context of an image processing DSL, for example, PLY can be used to create a lexer that recognizes commands like "crop" or "resize", flags such as "--width" or "--height", and arguments like file paths or numerical values. The parser can then interpret these tokens according to the DSL's syntax rules, allowing for the execution of complex image manipulation operations based on user commands.

Objectives

1. Understand what lexical analysis is.

2. Get familiar with the inner workings of a lexer/scanner/tokenizer.

3. Implement a sample lexer and show how it works.

Implementation description

The following steps were taken to implement the lexer (in the `lexer.py` file) for the image processing DSL with PLY:

Importing PLY's Lexer Module

```
import ply.lex as lex
```

Command and Flag Definitions

Here, `commands` and `flags` are tuples defining the valid commands and flags for the DSL. These will be used to validate the input text, ensuring that only recognized commands and flags are tokenized appropriately.

```
commands = ('crop', 'convert', 'rotate', 'resize', 'flipX', 'flipY', 'bw',
            'colorize', 'contrast', 'brightness', 'negative', 'blur',
            'sharpen', 'compress', 'ft', 'th')

flags = ('img', 'x', 'y', 'w', 'h', 'format', 'lvl', 'deg', 'help')
```

Token Declarations

This section declares the different types of tokens that the lexer can produce. Each token type represents a category of lexemes, such as commands, flags, numbers, file paths, etc.

```
tokens = (
    "START_COMMAND",
    "COMMAND",
    "FLAG",
    "NUMBER",
    "IMG_FORMAT",
    "IMAGE_PATH",
    "FOLDER_PATH",
    "EQUALS",
    "AND",
    "PIPELINE",
    "UNKNOWN_STRING",
)
```

Token Rules

The following sections define the rules for how each token type is recognized. In PLY, token rules are defined by functions whose names start with `t_` followed by the token type name. The docstring of each function contains a regular expression that describes the pattern of the lexeme that matches the token.

- t_START_COMMAND: Matches the initial command imp (image processing abbreviation), which likely signifies the start of a command in the DSL.

```
t_START_COMMAND = r'imp'
```

- t_NUMBER: Recognizes any sequence of digits as a number.

```
t_NUMBER = r'\d+'
```

- t_EQUALS, t_AND, and t_PIPELINE: Match the =, &&, and -> symbols, respectively. These might be used for assignment, command chaining, and pipeline operations within the DSL.

```
t_EQUALS = r'='
t_AND = r'&&'
t_PIPELINE = r'->'
```

- t_IMG_FORMAT: Matches image format specifications enclosed in quotes, e.g., "jpg" or "png".

```
def t_IMG_FORMAT(t):
    r'"(png|jpg|jpeg|gif|bmp|tiff|webp)"'
    t.value = t.value[1:-1]
    return t
```

- t_IMAGE_PATH and t_FOLDER_PATH: Recognize file and folder paths enclosed in quotes. They account for both Windows-style (with drive letters and backslashes) and Unix-style (with forward slashes) paths.

```
def t_IMAGE_PATH(t):
    r'"(?:[a-zA-Z]:\\|/)?(?:[^"/\\]+[\\/])*[^"/\\]*\.
(png|jpg|jpeg|gif|bmp|tiff|webp)"'
    t.value = t.value.strip('"')
    return t


def t_FOLDER_PATH(t):
    r'"(?:[a-zA-Z]:\\|/)?(?:[^"/\\]+[\\/])+[^"/\\]*/?"'
    t.value = t.value.strip('"')
    return t
```

- t_FLAG: Matches command flags, which start with --. The function checks if the flag is in the list of known flags and prints a message if it encounters an unknown flag.

```
def t_FLAG(t):
    r'--([a-zA-Z]+)'
    t.value = t.value[2:]

    if t.value not in flags:
        print(f"Unknown flag '{t.value}' at position {t.lexpos}")
        t.lexer.skip(1)
        return

    return t
```

- `t_COMMAND`: Matches any word-like string as a command. If the string is imp, it's classified as a START_COMMAND; otherwise, it checks against known commands and reports unknown commands.

```
def t_COMMAND(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'

    if t.value == "imp":
        t.type = "START_COMMAND"
        return t

    if t.value not in commands:
        print(f"Unknown command '{t.value}' at position {t.lexpos}")
        t.lexer.skip(1)
        return

    return t
```

- `t_UNKNOWN_STRING`: Catches any string enclosed in quotes that doesn't match other more specific token rules, treating it as an unknown string.

```
def t_UNKNOWN_STRING(t):
    r'"[^"]+"'
    t.value = t.value.strip('"')
    return t
```

- `t_ignore`: Skips over whitespace characters, such as spaces and tabs, which are not significant in the DSL's syntax.

```
t_ignore = ' \t\n'
```

Error Handling

This function is called by PLY when it encounters an illegal character—any character or sequence of characters that doesn't match any of the defined token rules. It prints a message indicating the error and skips the offending character.

```python
def t_error(t):
    print(f"Illegal character '{t.value[0]}' at position {t.lexpos}")
    t.lexer.skip(1)
```

Generating the Lexer

The lexer is generated by calling the `lex.lex()` function, which processes the token rules and returns a lexer object that can be used to tokenize input text.

```
lexer = lex.lex()
```

Conclusions / Results

In conclusion, the implementation of a lexer for an image processing domain-specific language demonstrates a practical application of lexical analysis principles in the context of a specialized field. The lexer designed for this lab effectively translates a series of characters into a sequence of tokens, which include commands, flags, numerical values, image formats, and paths. By leveraging PLY the lexer is equipped with the flexibility to define complex token patterns and handle various input scenarios gracefully. Overall, this lab not only reinforces the theoretical concepts of formal languages and finite automata but also provides hands-on experience in implementing a lexer for a practical application.

The source code is available in the `lexer.py` file in this folder, while the detailed results are documented in the Jupyter Notebook titled `lab3.ipynb`. This interactive notebook includes code snippets and execution outputs that demonstrate the implementation of the lexer and its behavior when tokenizing input text.

Bibliography

[1] PLY (Python Lex-Yacc)