Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Formal Languages and Finite Automata

Laboratory work 3:
Lexer and Scanner

Elaborated:

st. gr. FAF-221 Reabciuc Daria-Brianna

Verified:

asist. univ. Cretu Dumitru
prof. univ. Cojuhari Irina

Chisinau 2024

Objectives

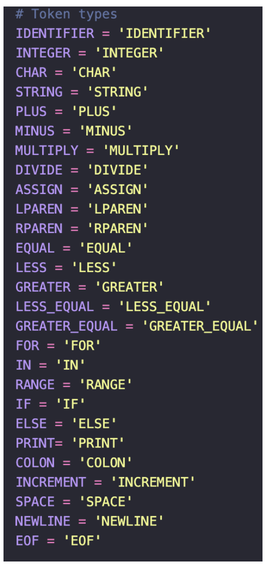
The term lexer comes from lexical analysis which, in turn, represents the process of extracting lexical tokens from a string of characters. There are several alternative names for the mechanism called lexer, for example tokenizer or scanner. The lexical analysis is one of the first stages used in a compiler/interpreter when dealing with programming, markup or other types of languages. The tokens are identified based on some rules of the language and the products that the lexer gives are called lexemes. So basically the lexer is a stream of lexemes. Now in case it is not clear what's the difference between lexemes and tokens, there is a big one. The lexeme is just the byproduct of splitting based on delimiters, for example spaces, but the tokens give names or categories to each lexeme. So the tokens don't retain necessarily the actual value of the lexeme, but rather the type of it and maybe some metadata.

1. Understand what lexical analysis [1] is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

IMPLEMENTATION

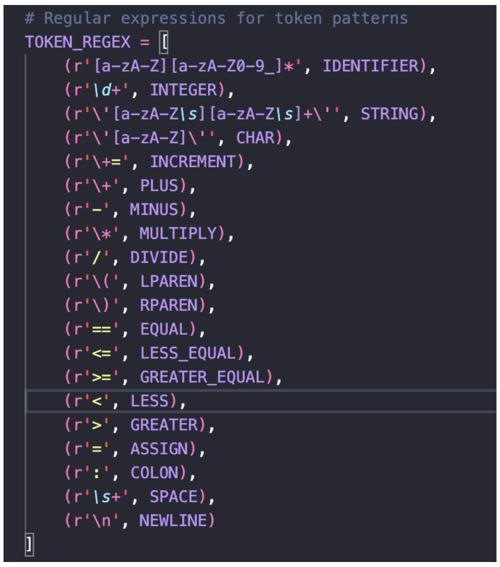
In this laboratory work I work in python because it is the most familiar and easy to work with:

Declare some types of tokens, which will be:



```python
# Token types
IDENTIFIER = 'IDENTIFIER'
INTEGER = 'INTEGER'
CHAR = 'CHAR'
STRING = 'STRING'
PLUS = 'PLUS'
MINUS = 'MINUS'
MULTIPLY = 'MULTIPLY'
DIVIDE = 'DIVIDE'
ASSIGN = 'ASSIGN'
LPAREN = 'LPAREN'
RPAREN = 'RPAREN'
EQUAL = 'EQUAL'
LESS = 'LESS'
GREATER = 'GREATER'
LESS_EQUAL = 'LESS_EQUAL'
GREATER_EQUAL = 'GREATER_EQUAL'
FOR = 'FOR'
IN = 'IN'
RANGE = 'RANGE'
IF = 'IF'
ELSE = 'ELSE'
PRINT= 'PRINT'
COLON = 'COLON'
INCREMENT = 'INCREMENT'
SPACE = 'SPACE'
NEWLINE = 'NEWLINE'
EOF = 'EOF'
```

Specifically an array named TOKEN_REGEX, which is used for lexical analysis or tokenization in a parser or a lexer (a component of a compiler or interpreter). Each element of the array is a tuple, where the first item is a regular expression pattern (regex), and the second item is an associated token type. This setup is typical in programming languages parsing or interpreters where you need to break down the input string into a series of tokens that the parser can understand.



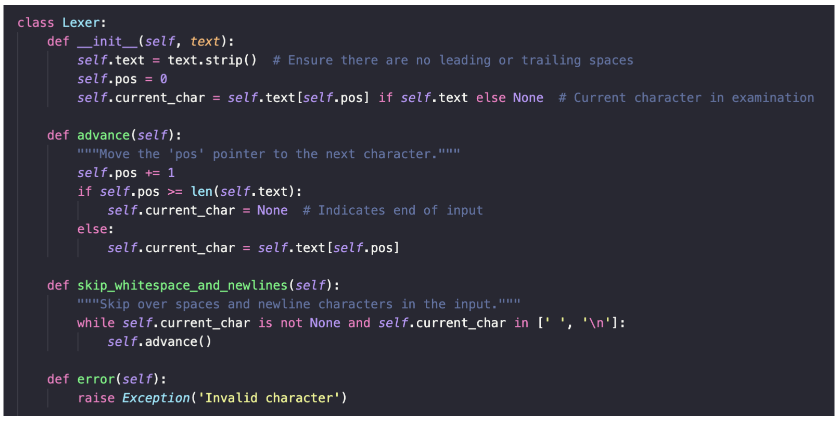
```python
# Regular expressions for token patterns
TOKEN_REGEX = [
    (r'[a-zA-Z][a-zA-Z0-9_]*', IDENTIFIER),
    (r'\d+', INTEGER),
    (r'\'[a-zA-Z\s][a-zA-Z\s]+\'', STRING),
    (r'\'[a-zA-Z]\'', CHAR),
    (r'\+=', INCREMENT),
    (r'\+', PLUS),
    (r'-', MINUS),
    (r'\*', MULTIPLY),
    (r'/', DIVIDE),
    (r'\(', LPAREN),
    (r'\)', RPAREN),
    (r'==', EQUAL),
    (r'<=', LESS_EQUAL),
    (r'>=', GREATER_EQUAL),
    (r'<', LESS),
    (r'>', GREATER),
    (r'=', ASSIGN),
    (r':', COLON),
    (r'\s+', SPACE),
    (r'\n', NEWLINE)
]
```

__init__: Initializes the lexer with given text, stripping white spaces from ends, and setting initial reading position and character.

advance: Moves the reading position one character forward and updates the current character or sets it to None if the end is reached, effectively iterating through the text.

skip_whitespace_and_newlines: Skips over spaces and newlines in the input, advancing the lexer's position past irrelevant characters to focus on significant ones.

error: Raises an exception for any unrecognized characters, signaling an error in parsing due to unexpected input.



```python
class Lexer:
    def __init__(self, text):
        self.text = text.strip()  # Ensure there are no leading or trailing spaces
        self.pos = 0
        self.current_char = self.text[self.pos] if self.text else None  # Current character in examination

    def advance(self):
        """Move the 'pos' pointer to the next character."""
        self.pos += 1
        if self.pos >= len(self.text):
            self.current_char = None  # Indicates end of input
        else:
            self.current_char = self.text[self.pos]

    def skip_whitespace_and_newlines(self):
        """Skip over spaces and newline characters in the input."""
        while self.current_char is not None and self.current_char in [' ', '\n']:
            self.advance()

    def error(self):
        raise Exception('Invalid character')
```

The *get_next_token method* in the lexer iterates through the input text to extract tokens:

Skips whitespaces and newlines to focus on relevant characters.

Matches the text against predefined patterns (from TOKEN_REGEX). If a pattern matches:

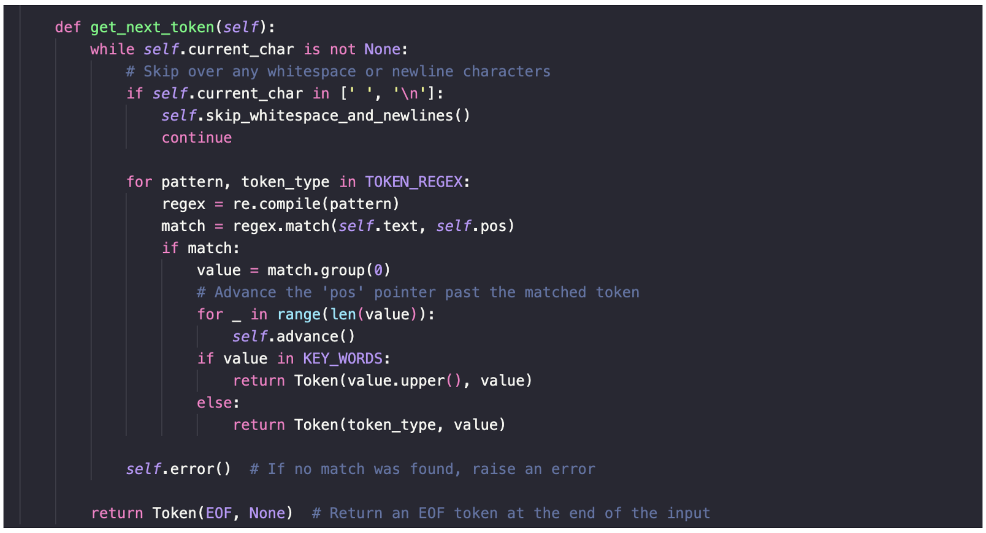
- The text corresponding to the match is captured.

- The lexer's position is advanced past this text.

- A new token is created and returned; if the text is a keyword, it's converted to uppercase.

If no pattern matches, an error is raised for invalid characters.

Once the end of the text is reached, an end-of-file (EOF) token is returned, indicating no more tokens are left to parse.



```python
def get_next_token(self):
    while self.current_char is not None:
        # Skip over any whitespace or newline characters
        if self.current_char in [' ', '\n']:
            self.skip_whitespace_and_newlines()
            continue

        for pattern, token_type in TOKEN_REGEX:
            regex = re.compile(pattern)
            match = regex.match(self.text, self.pos)
            if match:
                value = match.group(0)
                # Advance the 'pos' pointer past the matched token
                for _ in range(len(value)):
                    self.advance()
                if value in KEY_WORDS:
                    return Token(value.upper(), value)
                else:
                    return Token(token_type, value)

        self.error()  # If no match was found, raise an error

    return Token(EOF, None)  # Return an EOF token at the end of the input
```

Conclusion

The entire lexer and scanner code serves as a fundamental component of a programming language processor or data interpreter. This code is responsible for transforming raw text input into a structured sequence of tokens, which represent the smallest elements with meaning within the language's syntax, such as identifiers, keywords, and symbols.

The lexer starts by initializing with the input text, then systematically reads through it, character by character, identifying and skipping over whitespace and newlines to focus on significant text segments. Using a set of predefined regular expression patterns, it matches segments of the text to token types, such as operators, numbers, or identifiers. It handles special strings, recognizing them as keywords or specific symbols, and generates corresponding tokens.

When the lexer encounters characters or sequences that don't match any known patterns, it raises an error, indicating unrecognized or invalid syntax. The process continues until all input is consumed, ending with the generation of an end-of-file (EOF) token to signal completion.

Overall, this lexer and scanner framework is crucial for the initial phase of parsing and interpreting code, turning unstructured input into a stream of tokens that can be further analyzed or compiled by a parser according to the rules of the language's grammar. This structured approach enables the effective processing, understanding, and execution of code or data formats.

https://github.com/dariabrianna/DSL_personal_laboratories/tree/main/Lexer_Scanner