Indian Institute of Technology, Gandhinagar

H** Programming Language

Authors:
Pulkit Gautam 21110169
Manav Parmar 21110120
Shubh Agarwal 21110205
Aman Singh 21110020

Lexer Documentation

Compilers

CS 327

Under the guidance of

Prof. Abhishek Bichhawat

February 4, 2024

Lexer Token Classes and Regular Expressions

1. Token Classes

The Lexer categorizes the source code into various token classes. Here is a list of token classes identified by the lexer:

- `quotation`: Represents double quotes used in string literals.
- `constant`: Represents numeric constants, both integers and decimals. Also includes string eg. "Hello World".
- `keyword`: Represents programming language keywords such as `var`, `if`, `else`, etc.
- `operator`: Represents operators used in expressions, e.g., `+=`, `=`, `*`.
- `comparator`: Represents comparison operators like `==`, `!=`, `<`.
- `punctuation`: Represents punctuation marks, including commas and colons.
- `endOfStatement`: Represents the end of a statement, marked by a semicolon (`;`).
- `parenthesis`: Represents various types of parentheses: `()`, {}, `[]`.
- `identifier`: Represents identifiers, such as variable names and function names.
- `error`: Represents unrecognized tokens, indicating potential syntax errors.

2. Examples

Below is a table providing examples of tokens falling under each token class:

Token Class	Example Tokens
quotation	""
constant	123, 3.14, 0.5, "Hello"
keyword	var, const, if, elif, else, while, do, for, func, return, try, catch, finally, throw, print, true, false, tuple, list, arr, ExceptionType, null, break, continue, this, typeof
operator	+=, -=, ++, --, +, -, *, /, %, =
comparator	==, !=, <=, >=, <, >, !
punctuation	,, :, .
endOfStatement	;
parenthesis	(,), {, }, [,]
identifier	variable_name, func_name, myVar
error	@#%, $invalid$, &

3. Regular Expressions Used for Tokenization

The lexer utilizes several regular expressions to match and identify different token classes in the source code. Here is an overview of the regular expressions used:

Removing Single-Line Comments and Compressing Whitespaces

```
source_code = re.sub(r'//.*', '', self.source_code)
source_code = re.sub(r'\s+', ' ', source_code)
```

Creating Regular Expressions for Operators and Comparators

```
operators_regex = '|'.join(map(re.escape, operators + comparators))
continuous_operators_regex = rf'(?:{operators_regex})+'
```

Tokenizing Source Code

```
tokens = re.findall(
    r'"(?:\\.|[^"])*"|\b\d+\.\d+\b|\b\d+\b|\b(?:' + '|'.join(map(re.escape, keywords)) +
    continuous_operators_regex + r'|\b[a-zA-Z_]\w*\b|[^\s\w]',
    source_code)
```

Tokenizing String Literals

```
if re.match(r'^"[^"]*"$', token):
    # Tokenize string literals: "example"
    self.tokens.append(('quotation', '\"'))
    self.tokens.append(('constant', token))
    self.tokens.append(('quotation', '\"'))
```

Handling Special Cases

```
elif token == '"':
    # Tokenize standalone double quotes
    self.tokens.append(('quotation', '\"'))
else:
    # Tokenize unrecognized tokens as errors
    self.tokens.append(('error', token))
```

Summary and Conclusion

In conclusion, the Lexer presented here is designed to tokenize source code written in a simplified programming language. It employs a series of regular expressions to categorize various language elements into distinct token classes. The key token classes include keywords, operators, comparators, punctuation, identifiers, literals, and error tokens.

The Lexer successfully removes single-line comments and compresses consecutive whitespaces, enhancing the source code preprocessing. It creates regular expressions for operators and comparators, enabling the identification of continuous sequences of these elements. The tokenization process involves matching different token types such as double-quoted strings, numeric constants, keywords, operators, and identifiers.

Special attention is given to string literals, standalone double quotes, and handling unrecognized tokens as errors. The resulting token classes provide a comprehensive representation of the language elements present in the source code.

This document serves as a comprehensive guide to the Lexer's implementation, offering insights into the regular expressions and logic used for tokenization. It is intended to aid developers and students in understanding the inner workings of a basic lexer and how it dissects source code into meaningful tokens.

The lexer's modular structure allows for easy adaptation and extension to handle additional language features. As a fundamental component of a compiler or interpreter, the lexer lays the groundwork for subsequent phases of language processing. Further improvements and optimizations can be explored based on specific language requirements and design considerations.

Overall, the lexer demonstrates an essential step in the compilation or interpretation process, paving the way for subsequent phases to analyze and execute the parsed code. Its role in breaking down source code into manageable units contributes to the overall efficiency and accuracy of the language processing pipeline.