Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Guadalajara

Implementation of Computational Methods

Lexical Analyzer: User Manual and Code Description

Ana Camila Jiménez Mendoza - A01174422
Sebastián Cervera Maltos - A01068436
Alexei Delgado De Gante - A01637405

Group 601

11 mar 2024

User Manual: Expression Lexer

1. Installation

The provided code is a standalone Python script and doesn't require any specific installation. You'll need a Python interpreter (version 3 recommended) to run it.

2. File Preparation

The lexer expects a text file containing your expressions. Here are some guidelines for the file format:

- Each line can contain an expression.
- Supported characters include:
    - Letters (a-z, A-Z) for variable names.
    - Digits (0-9) for numbers.
    - Basic arithmetic operators: +, -, *, /
    - Assignment operator: =
    - Parentheses: (and)

Make sure there are no spaces around operators or between variable names and operators (e.g., x + y is valid, x +y is not).

3. Running the Lexer

Save the provided code as a Python file (e.g., lexical_analyzer.py).
Place your expression file (e.g., expressions.txt) in the same directory as the Python script.
Open a terminal or command prompt and navigate to the directory containing the script and file.
Run the script using the following command: python expression_lexer.py expressions.txt

Note: Replace "expressions.txt" with the actual name of your expression file.

4. Understanding the Output

The lexer will print each identified token along with its type on the console. Here's an explanation of the output format:

- Token: The actual characters that form the token (e.g., x, +, 12.5)
- Type: The category assigned to the token based on its content.
- The code defines types for variables, operators, parentheses, and integers/floats.

5. Adicional Specifications

- The code provided was made using Replit, here is the source code:https://replit.com/join/opfntubaeb-cam0117

Code:

Content on file “expressions.txt”:

```
Unset
b=7
a = 32.4 *(-8.6 - b)/    6.1
d = a * b
48x= 6745 - .80
casa
567863
 567x89=90-.90.98
890697635
```

Code on main.py file:

```python
def lexer(filepath):
 with open(filepath, 'r') as file:
  expressions = file.read()

 # Tipos de tokens
 TOKEN_TYPES = {
   '+': 'sum',
   '-': 'subtract',
   '*': 'multiply',
   '/': 'division',
   '=': 'assignment',
   '(': 'left parenthesis',
   ')': 'right parenthesis'
 }

 # Tabla de transiciones
 states = {
   0: {
     'letter': 8,
     'digit': 1,
     '=': 3,
     '+': 4,
     '-': 5,
     '*': 6,
     '/': 7,
     '(': 9,
```

```
    ')': 10,
    '.': 2
  },
  1: {
    'digit': 1,
    '.': 2
  },
  2: {
    'digit': 2
  },
  3: {},
  4: {},
  5: {},
  6: {},
  7: {},
  8: {
    'letter': 8
  },
  9: {},
  10: {}
}

# Initialize variables
current_state = 0
current_token = ''

# Handle tokens
def handle_token(token):
 if token:
  token_type = TOKEN_TYPES.get(
    token, 'variable'
    if token[0].isalpha() else 'float' if '.' in token else 'integer')
  print(f"{token}\t\t{token_type}")

# Lexer
for char in expressions:
 char_type = 'letter' if char.isalpha() else 'digit' if char.isdigit(
 ) else char

 if char_type in states[current_state]:
  current_state = states[current_state][char_type]
  current_token += char
 else:
  handle_token(current_token)
  current_token = ''
```

```
    current_state = 0
    if char_type in states[current_state]:
     current_state = states[current_state][char_type]
     current_token += char

  # Last token
  handle_token(current_token)


lexer("expressions.txt")
```

Code description:

The code defines a function called lexer that reads a file containing expressions and identifies the different parts (tokens) of those expressions.

TOKEN_TYPES: This dictionary defines the different types of tokens it can accept by our Lexical Analyzer and exactly defines what that token is, in case there is another token that isn't defined here, it will be identified as a variable.

The "states" dictionary is a finite state automaton (FSA) that helps identify tokens.

- Each key represents a state (0-10) in the FSA.
- Each value is a dictionary mapping character types ('letter', 'digit', etc.) to the next state the FSA should transition to when encountering that character type.

Forward in the code we initialize in state 0 and with an empty token.

Followed by the handle_token() that accepts a token as an argument, this function is called whenever a complete token is identified, it checks if the token is empty, if not it uses the TOKEN_TYPES to find the corresponding token.

- If the token is not found in the dictionary, it uses heuristics:
- If the first character is an alphabet character, it's considered a 'variable'.
- If a '.' is present, it's considered a 'float'.

Otherwise, it's considered an 'integer'. Finally, it prints the token and its type.

The Lexical Analyzer has a loop in which we iterate through the char in the strings contained in our txt file It determines the character type (char_type) based on its properties (alphabet, digit, etc.). It checks if the current state (current_state) in the states dictionary has a transition for the encountered char_type.

- If a valid transition exists, the code updates the current_state and adds the character to the current token (current_token).
- If there's no valid transition for the current character:

- It calls the handle_token function to process the completed token (current_token).
- It resets the current_state and current_token for the next token.
- It then checks again if there's a valid transition for the current character in the initial state (0).