جامعة الإسكندرية
ALEXANDRIA
UNIVERSITY

*

Abdelrahman Mohamed Abdelmonem - 6865
Nour Hesham Shaheen - 7150
Seif El Din Mahmoud - 6773
Yahia Walid - 7137
Ahmed Wael - 6704

*

Programming Languages Translation

Final Project:
Recursive-descent Compiler for Mini-language

*

Dr. Ahmed El Nahhas

Introduction

This report presents the design and implementation of a recursive-descent compiler for translating assignment statements in a mini-language into assembly language for a stack machine. The compiler utilizes zero-address instructions and uses an evaluation stack for carrying out computations.

Compiler Architecture:

The compiler consists of a lexical analyzer, parser, and code generator. The lexical analyzer receives a string of characters representing the assignment statement as input and produces a sequence of tokens. The parser then uses a recursive descent algorithm to analyze the tokens and generate code instructions. The code generator takes the generated code instructions and produces a string of assembly language instructions.

Code Snippets + Comments:

- We start by initializing the class “Compiler” and its variables. The following function is the constructor function that initializes the instance variables of the `Compiler` class. It sets `self.tokens` to an empty list, `self.current_token` to `None`, `self.index` to `0`, and `self.output` to an empty list.

```
import re

class Compiler:
    def __init__(self):
        self.tokens = []
        self.current_token = None
        self.index = 0
        self.output = []
```

- The following function takes an input string and tokenizes it into a list of tokens using a regular expression. It sets `self.tokens` to the list of tokens and `self.current_token` to the first token in the list.

```
# takes an input string and splits it into a list of tokens using a
regular expression
    def tokenize(self, input_string):
        self.tokens = re.findall(r'\w+|[^\s\w]', input_string)
        self.current_token = self.tokens[self.index]
```

- This function takes an expected token as an argument and moves through the list of tokens one at a time. If the current token matches the expected token, it advances to the next token in the list. If there are no more tokens, it sets the current token to `None`. If the current token does not match the expected token, it raises an exception.

```
    def consume(self, expected_token):
        if self.current_token == expected_token:
            self.index += 1
            if self.index < len(self.tokens):
                self.current_token = self.tokens[self.index]
            else:
                self.current_token = None
        else:
            raise Exception(f'Error: Expected {expected_token}, got
{self.current_token}')
```

- This function handles individual factors in an expression, which can be either numbers or variables. If the current token is a number, it adds a `LIT` instruction to the output list with the value of the number, consumes the token, and returns. If the current token is a variable name, it adds a `LIT` instruction with the name of the variable, consumes the token, and checks if the next token is a left square bracket `([)` indicating an index into an array. If it is, it compiles the expression inside the brackets using the `expr()` function and adds an `ADD` instruction to the output. Then it adds a `LOAD` instruction to the output and returns. If there is no left square bracket, it simply adds a `LOAD` instruction to the output and returns. If the current token is neither a number nor a variable name, it raises an exception.

```
    def factor(self):
        if re.match(r'\d+', self.current_token):
            value = int(self.current_token)
            self.output.append(f'LIT {value}')
            self.consume(self.current_token)
        elif re.match(r'\w+', self.current_token):
            value = self.current_token
            self.output.append(f'LIT {value}')
            self.consume(self.current_token)
            if self.current_token == '[':
                self.consume('[')
                self.expr()
                self.consume(']')
                self.output.append('ADD')
                self.output.append('LOAD')
            else:
                self.output.append('LOAD')
        else:
            raise Exception(f'Error: Invalid token
{self.current_token}')
```

- This function handles multiplication and division in an expression. It calls `factor()` to parse the first factor in the term, and then repeatedly checks if the current token is a * or / operator. If it is, it consumes the operator, calls `factor()` again to parse the next factor in the term, and adds a `MUL` or `DIV` instruction to the output.

```
    #handles multiplication and division
    def term(self):
        self.factor()
        while self.current_token in ['*', '/']:
            op = self.current_token
            if op == '*':
                self.consume('*')
                self.factor()
                self.output.append('MUL')
            elif op == '/':
                self.consume('/')
                self.factor()
                self.output.append('DIV')
```

- This function handles addition and subtraction in an expression. It first checks if the current token is a unary - operator. If it is, it consumes the operator and checks if the next token is a number, in which case it negates the number and adds a `LIT` instruction with the negated value to the output. If the next token is not a number, it calls `term()` to parse the first term in the expression, and then adds a `NEG` instruction to the output. If the current token is not a unary - operator, it calls `term()` to parse the first term in the expression. It then repeatedly checks if the current token is a + or - operator, and if it is, it consumes the operator, calls `term()` again to parse the next term, and adds an `ADD` or `SUB` instruction to the output.

```
# handles addition and subtraction
    def expr(self):
        if self.current_token == '-':
            op = '-'
            self.consume('-')
            if re.match(r'\d+', self.current_token):
                value = int(self.current_token)
                value *= -1
                value = str(value)
                value.replace('-', '')
                value = '-' + value
                value = int(value)
                value = str(value)
```

```
              value.replace('--', '')

              self.output.append(f'LIT {value}')
              self.consume(self.current_token)
          else:
              self.term()
              self.output.append('NEG')
      else:
          self.term()
      while self.current_token in ['+', '-']:
          op = self.current_token
          if op == '+':
              self.consume('+')
              self.term()
              self.output.append('ADD')
          elif op == '-':
              self.consume('-')
              self.term()
              self.output.append('SUB')
```

- This function parses and compiles a variable assignment. It first checks that the current token is a valid variable name, and adds a `LIT` instruction with the name of the variable to the output. It then checks if the next token is a left square bracket (`[`), indicating an index into an array.

```
def assign(self):
      if re.match(r'\w+', self.current_token):
          value = self.current_token
          self.consume(self.current_token)
          if self.current_token == '[':
              self.consume('[')
              self.expr()
              self.consume(']')
              self.output.append('ADD')
          else:
                self.output.append(f'LIT {value}')
            self.consume('=')
            self.expr()
            self.output.append('STORE')
        else:
            raise Exception(f'Error: Invalid token
{self.current_token}')
```

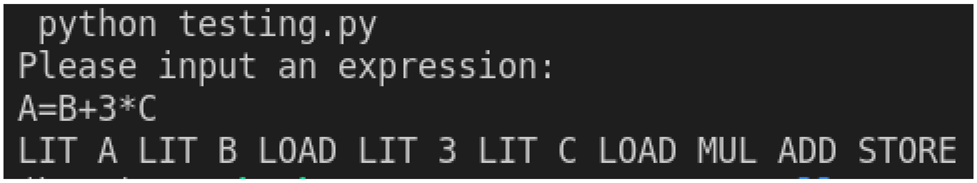
```
    # takes an input string, tokenizes it, and repeatedly calls
assign() to compile each assignment in the input string.
    # The resulting output is a list of instructions, which are
concatenated into a single string and printed.
    def compile(self, input_string):
        self.tokenize(input_string)
        while self.current_token is not None:
            self.assign()
```

```
def main():
    compiler = Compiler()
    input_string = input()
    compiler.compile(input_string)
    output_string = " ".join(compiler.output)
    print(output_string)

if __name__ == '__main__':
    main()
```

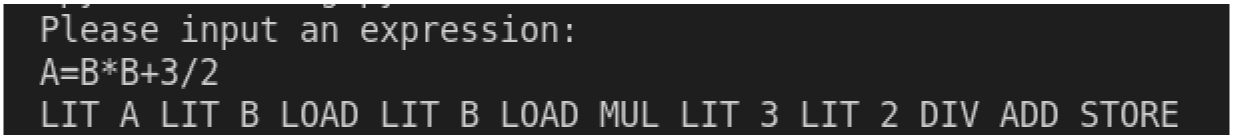
Examples of input data and generated output:

A = B + 3 * C



```
 python testing.py
Please input an expression:
A=B+3*C
LIT A LIT B LOAD LIT 3 LIT C LOAD MUL ADD STORE
```

A = B*B+3/2



```
Please input an expression:
A=B*B+3/2
LIT A LIT B LOAD LIT B LOAD MUL LIT 3 LIT 2 DIV ADD STORE
```

Code:

```
import re

class Compiler:
    def __init__(self):
        self.tokens = []
        self.current_token = None
        self.index = 0
        self.output = []

    # takes an input string and splits it into a list of tokens using
a regular expression
    def tokenize(self, input_string):
        self.tokens = re.findall(r'\w+|[^\s\w]', input_string)
        self.current_token = self.tokens[self.index]

    # it moves through the list of tokens, one at a time. It takes an
expected token as an argument
    #if the current token matches, it advances to the next token in
the list.
    # If there are no more tokens, it sets the current token to None.
    # If the current token does not match the expected token, it
raises an exception.
    def consume(self, expected_token):
        if self.current_token == expected_token:
            self.index += 1
            if self.index < len(self.tokens):
                self.current_token = self.tokens[self.index]
            else:
                self.current_token = None
        else:
            raise Exception(f'Error: Expected {expected_token}, got
{self.current_token}')

    #handles individual factors (numbers or variables)
    def factor(self):
        if re.match(r'\d+', self.current_token):
            value = int(self.current_token)
            self.output.append(f'LIT {value}')
            self.consume(self.current_token)
        elif re.match(r'\w+', self.current_token):
            value = self.current_token
            self.output.append(f'LIT {value}')
            self.consume(self.current_token)
            if self.current_token == '[':
                self.consume('[')
                self.expr()
                self.consume(']')
                self.output.append('ADD')
                self.output.append('LOAD')
            else:
                self.output.append('LOAD')
        else:
            raise Exception(f'Error: Invalid token
{self.current_token}')

    #handles multiplication and division
    def term(self):
        self.factor()
        while self.current_token in ['*', '/']:
            op = self.current_token
            if op == '*':
                self.consume('*')
                self.factor()
                self.output.append('MUL')
            elif op == '/':
                self.consume('/')
                self.factor()
                self.output.append('DIV')
    # handles addition and subtraction
    def expr(self):
        if self.current_token == '-':
            op = '-'
            self.consume('-')
            if re.match(r'\d+', self.current_token):
                value = int(self.current_token)
                value *= -1
                value = str(value)
                value.replace('-', '')
                value = '-' + value
                value = int(value)
                value = str(value)
                value.replace('--', '')

                self.output.append(f'LIT {value}')
                self.consume(self.current_token)
            else:
                self.term()
                self.output.append('NEG')
        else:
            self.term()
        while self.current_token in ['+', '-']:
            op = self.current_token
            if op == '+':
                self.consume('+')
                self.term()
                self.output.append('ADD')
            elif op == '-':
                self.consume('-')
                self.term()
                self.output.append('SUB')

    # used to parse and compile variable assignments.
    def assign(self):
        if re.match(r'\w+', self.current_token):
            value = self.current_token
            self.consume(self.current_token)
            if self.current_token == '[':
                self.consume('[')
                self.expr()
                self.consume(']')
                self.output.append('ADD')
            else:
                self.output.append(f'LIT {value}')
            self.consume('=')
            self.expr()
            self.output.append('STORE')
        else:
            raise Exception(f'Error: Invalid token 
{self.current_token}')

    # takes an input string, tokenizes it, and repeatedly calls 
assign() to compile each assignment in the input string.
    # The resulting output is a list of instructions, which are 
concatenated into a single string and printed.
    def compile(self, input_string):
        self.tokenize(input_string)
        while self.current_token is not None:
            self.assign()

def main():
    compiler = Compiler()
    input_string = input("Please input an expression:\n")
    compiler.compile(input_string)
    output_string = " ".join(compiler.output)
    print(output_string)

if __name__ == '__main__':
    main()
```