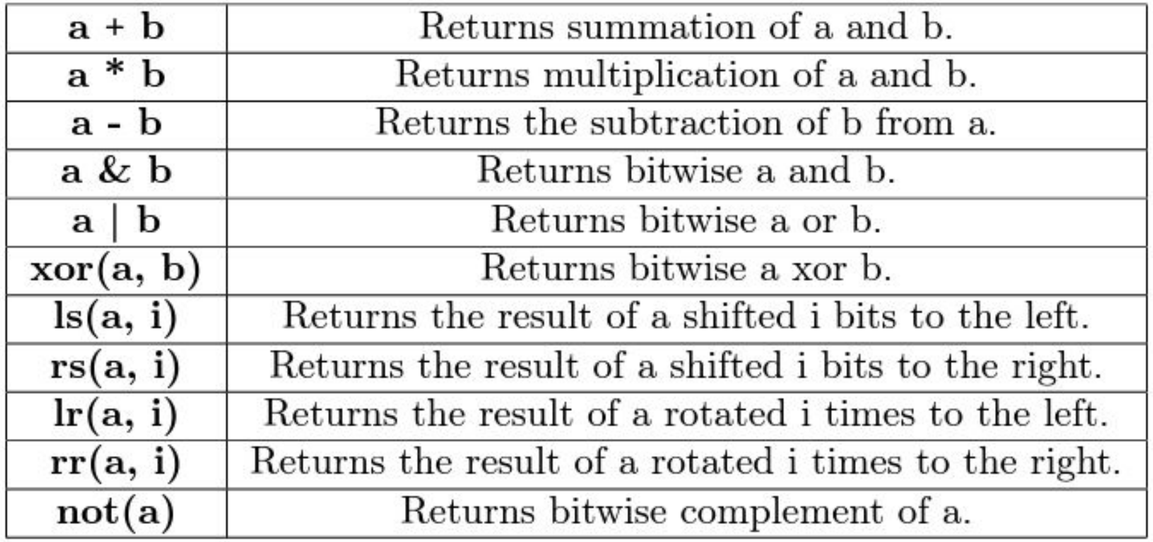
HW1:
Advanced Calculator Interpreter

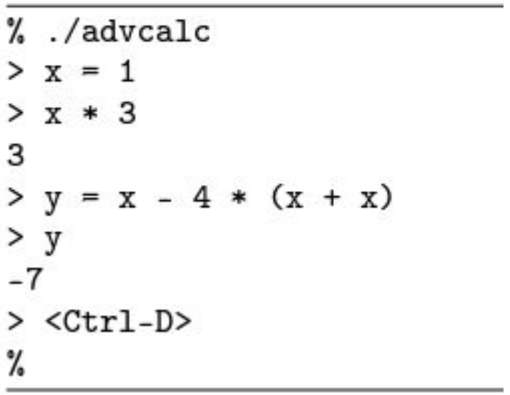
14.3.2023

Advanced Calculator



a + b	Returns summation of a and b.
a * b	Returns multiplication of a and b.
a - b	Returns the subtraction of b from a.
a & b	Returns bitwise a and b.
a | b	Returns bitwise a or b.
xor(a, b)	Returns bitwise a xor b.
ls(a, i)	Returns the result of a shifted i bits to the left.
rs(a, i)	Returns the result of a shifted i bits to the right.
lr(a, i)	Returns the result of a rotated i times to the left.
rr(a, i)	Returns the result of a rotated i times to the right.
not(a)	Returns bitwise complement of a.

Possible expressions



```
% ./advcalc
> x = 1
> x * 3
3
> y = x - 4 * (x + x)
> y
-7
> <Ctrl-D>
%
```

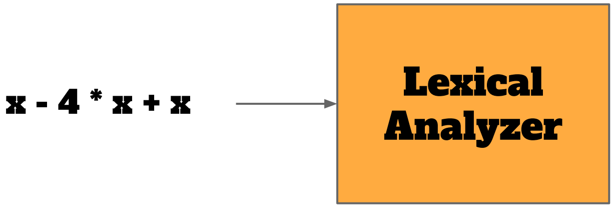
Example

Implementation

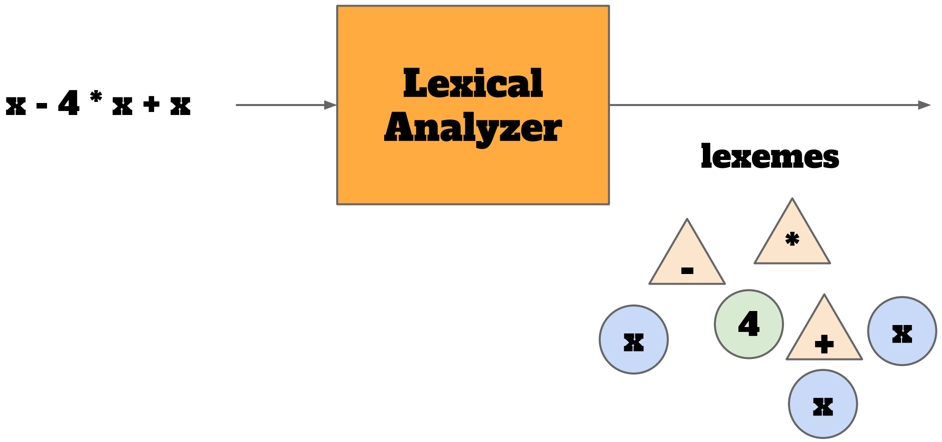
```
x - 4 * x + x
```

Implementation

A **lexeme** is a sequence of characters that form a token.

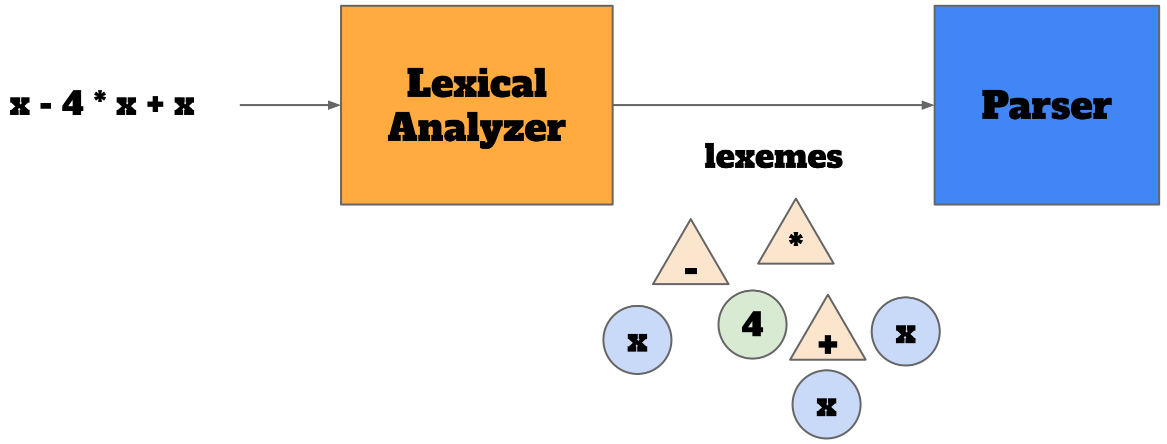


Implementation

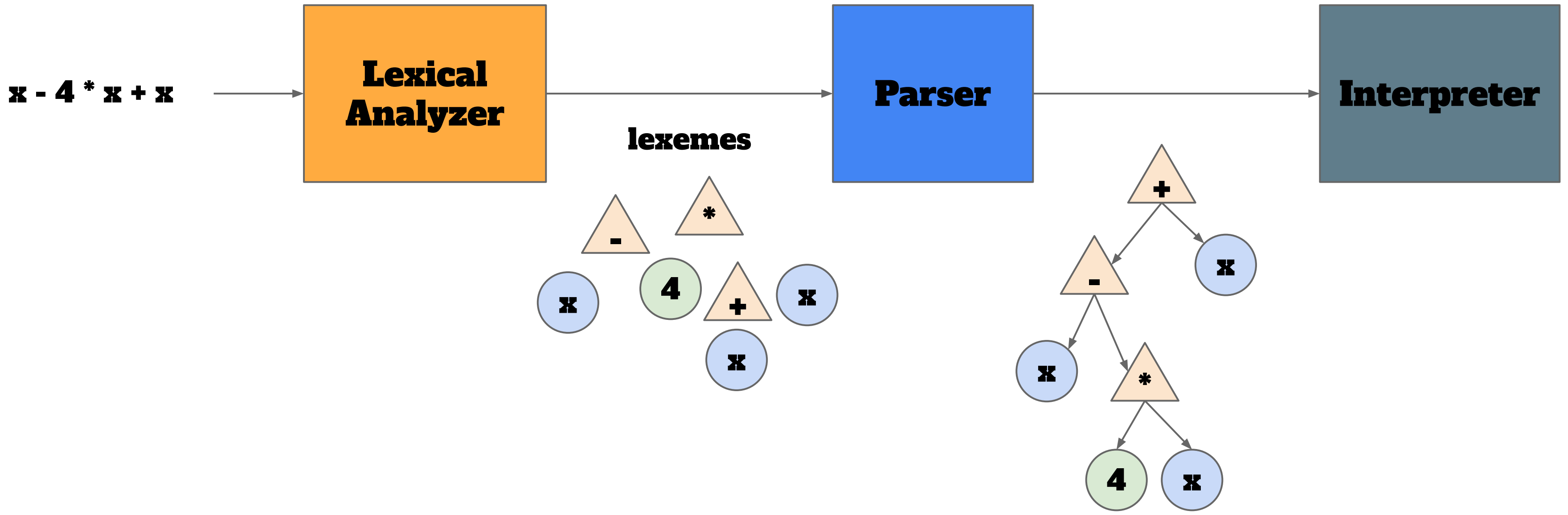


Implementation

the process of recognizing a phrase in the stream of tokens is called **parsing**

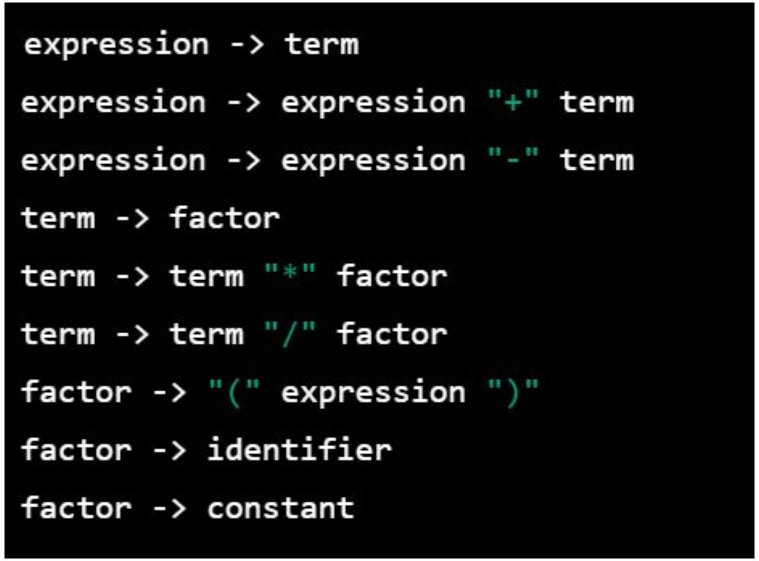


Implementation

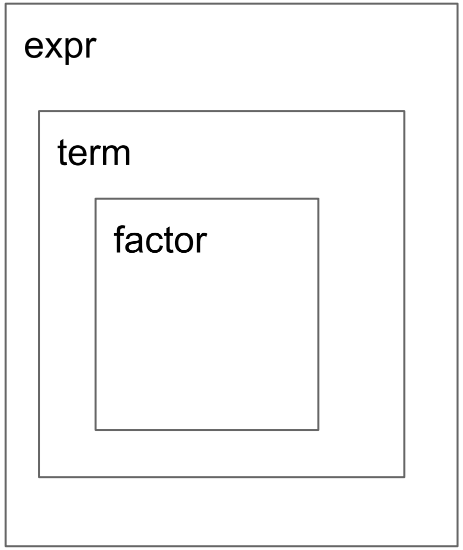


Parsing/Syntax Analysis

- A grammar specifies the syntax of a language in a concise manner.
- A grammar consists of a sequence of *rules*



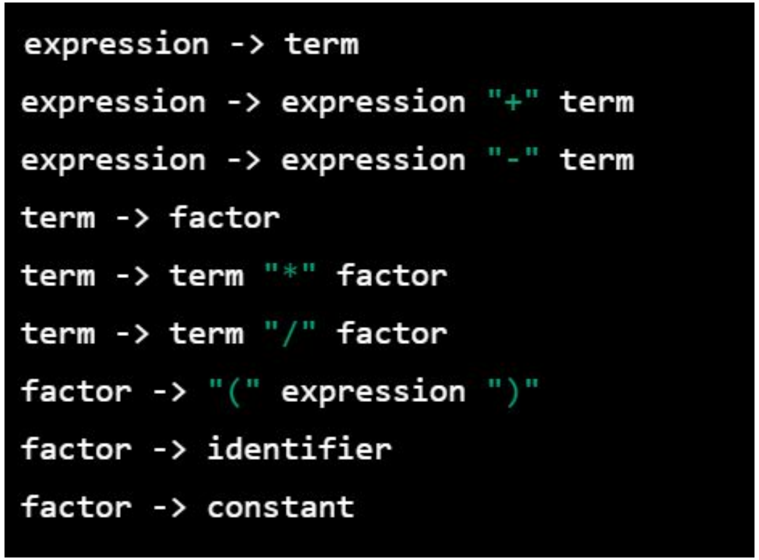
```
expression -> term
expression -> expression "+" term
expression -> expression "-" term
term -> factor
term -> term "*" factor
term -> term "/" factor
factor -> "(" expression ")"
factor -> identifier
factor -> constant
```



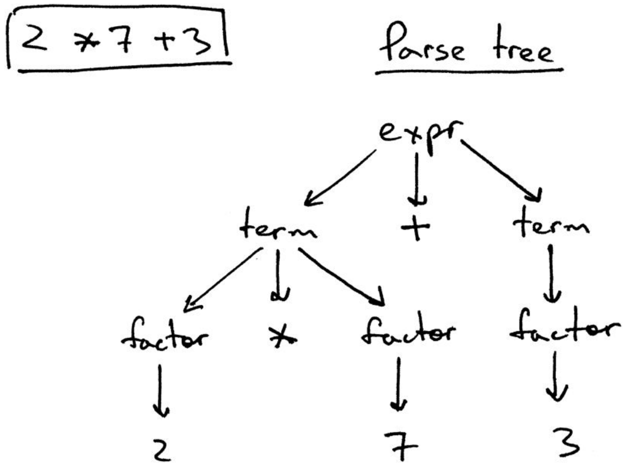
Precedence

Parse Trees

A *parse-tree* is a tree that represents the syntactic structure of a language construct according to grammar definition.



```
expression -> term
expression -> expression "+" term
expression -> expression "-" term
term -> factor
term -> term "*" factor
term -> term "/" factor
factor -> "(" expression ")"
factor -> identifier
factor -> constant
```



Grammar

```
function parse_expression():
    term = parse_term()
    while current_token.type is "+" or current_token.type is "-":
        operator = current_token
        advance()
        next_term = parse_term()
        term = new_binary_expression_node(operator, term, next_term)
    return term
```

```
expression -> term
expression -> expression "+" term
expression -> expression "-" term
```

Grammar

```
function parse_term():
    factor = parse_factor()
    while current_token.type is "*" or current_token.type is "/":
        operator = current_token
        advance()
        next_factor = parse_factor()
        factor = new_binary_expression_node(operator, factor, next_factor)
    return factor
```

```
term -> factor
term -> term "*" factor
term -> term "/" factor
```

Grammar

```
function parse_factor():
    if current_token.type is "identifier":
        node = new_identifier_node(current_token)
        advance()
        return node
    elif current_token.type is "constant":
        node = new_constant_node(current_token)
        advance()
        return node
    elif current_token.type is "(":
        advance()
        expression = parse_expression()
        if current_token.type is not ")":
            error("Expected ')' but found " + current_token.value)
        advance()
        return expression
    else:
        error("Unexpected token: " + current_token.value)
```

```
factor -> "(" expression ")"
factor -> identifier
factor -> constant
```