Project is about

- Module for lexical analysis .
- Module for syntax analysis .
- Module for semantic analysis .
- The parser will be based on an EBNF grammar .
- Code generator .

How to test

- Run the following script `npm install` then `npm start` .

Instructions

- Developing a Lexical Analyzer

we split the string by a single space, we map the produced substrings to their trimmed version and filter the empty strings.

```
const lex = str => str.split(' ').map(s => s.trim()).filter(s => s.length);
```

Invoking the lexer with an expression will produce an array of strings:

```
lex('mul 3 sub 2 sum 1 3 4')
```

`// ["mul", "3", "sub", "2", "sum", "1", "3", "4"]`

- Developing a Parser

EBNF is the grammar of our language:

`digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

`num = digit+`

`op = sum | sub | mul | div`

`expr = num | op expr+`

Parsing the following string :

`mul 3 sub 2 sum 1 3 4`

The parser will produce the following AST :

` mul`

` / \`

` 3 sub`

` / \`

` 2 sum`

` /|\`

` 1 3 4`

function called parse which accepts a single argument called tokens. Inside of it we define five more functions:

```
`const Op = Symbol('op');`

`const Num = Symbol('num');`

`const parse = tokens => {`

`let c = 0;`

`const peek = () => tokens[c];`

`const consume = () => tokens[c++];`

`const parseNum = () => ({ val: parseInt(consume()), type: Num });`

`const parseOp = () => {`

`const node = { val: consume(), type: Op, expr: [] };`

`while (peek()) node.expr.push(parseExpr());`

`return node;`

`};`

`const parseExpr = () => /\d/.test(peek()) ? parseNum() : parseOp();`

`return parseExpr();`

`};`
```

- peek - returns the element of tokens associated with the current value of the c local variable.

- consume - returns the element of tokens associated with the current value of the c local variable and increments c.

- parseNum - gets the current token (i.e. invokes peek()), parses it to a natural number and returns a new number token.

- parseOp - we'll explore in a little bit.

- parseExpr - checks if the current token matches the regular expression /\d/ (i.e. is a number) and invokes parseNum if the match was successful, otherwise returns parseOp.

- Developing the Transpiler

We have transpile function to handle transpiling strings :

```
`const transpile = ast => {`

` const opMap = { sum: '+', mul: '*', sub: '-', div: '/' };`

` const transpileNode = ast => ast.type === Num ? transpileNum(ast) :
transpileOp(ast);`

` const transpileNum = ast => ast.val;`

` const transpileOp = ast =>`(${ast.expr.map(transpileNode).join(' ' + opMap[ast.val]
+ ' ')})`;`

` return transpileNode(ast);`

`};`
```

- transpileNum - translates a number to a JavaScript number (simply by returning it).

- transpileOp - translates an operation to a JavaScript arithmetic operation. For each operation node, we want to transpile its sub-expressions first. We do that by invoking the transpileNode function.

- On the last line of transpile's body, we return the result of transpileNode applied to the root of the tree.

- Developing the semantic analyzer for English language

Word scores are normalized to a scale between -1 and 1 and all scores in a sentence are summed for the total score. The higher the score the more positive the sentiment.

Words are sourced from:

- English: the
[AFINN-165](http://www2.imm.dtu.dk/pubdb/views/publication_details.php?id=6010)

Semantic function detects positive or negative sentiment in text and scores them .