predictive-parser

Predictive parser for simple expressions

File structure

```
pred_parser/
| definition/
| | grammar.py       # classes defined for parser and translator (abstract syntax)
| | token.py         # classes defined for scanner and parser (token table)
| function/
| | parser.py        # transforms a token list to an abstract syntax tree
| | scanner.py       # transforms strings to a token list
| | translator.py    # transforms an abstract syntax tree to preordered
intermediate code string
| run.py             # runs the parser
temp/
| output.txt         # result of parser
test/
| input.txt          # test input
| output.txt         # correct output
| run.py             # compare temp/output.txt with test/output.txt
README.md            # document
test.bat             # testing script for windows
test.sh              # testing script for shellscript
```

Token

```
#CLASS
Id          := "[a-zA-Z]"
Num         := "[0-9]+"
Oper        := "+|-|*|/"
Plus        := "+"
Minus       := "-"
Times       := "*"
Over        := "/"
Error       := scanner error
```

Grammar

```
#CLASS        #BNF
Goal          <goal>
Expr          <expr>
Term          <term>
Factor        <factor>
```

```
Empty         EOF
Add           <term>+<expr>
Sub           <term>-<expr>
Mul           <factor>*<term>
Div           <factor>/<term>
Number        NUMBER
Indentifier   ID
```

Scanner

transforms a string to a token list

```
class Scanner(object):
  def __init__(self):
    pass

  def _scan_without_whitespace(self, word: str) -> list:
    # input: a word(string) which has no white space
    # output: scanned list of tokens
    # Recursively scan a word which has no white space
    pass

  def scan(self, code: str) -> list:
    # input: string(code)
    # output: scanned list of string(code)
    # Split all words in string in terms of white spaces
    pass
```

Parser

transforms a token list to an abstract syntax tree

```
class Parser(object):
  def __init__(self, token_list: list):
    # Store index of token list
    # Initialize token index
    # Create a stack
    pass

  def _next_token(self):
    # move token index
    pass

  def parse(self) -> Goal:
    # call expr()
    # return parse tree
    pass

  def expr(self):
```

```
    # call term()
    # call expr_prime()
    # if the <expr> was matched,
    # pop 3 elements in stack,
    # create a subtree
    # push it in stack
    pass

  def expr_prime(self):
    # if token is Plus -> push Add()
    # else if token is Minus -> push Sub()
    pass

  def term(self):
    # call factor()
    # call term_prime()
    # if the <term> was matched,
    # pop 3 elements in stack,
    # create a subtree
    # push it in stack
    pass

  def term_prime(self):
    # if token is Times -> push Mul()
    # else if token is Over -> push Div()
    pass

  def factor(self):
    # if token is Num -> push Number()
    # else if token is Id -> push Identifier()
    pass
```

Translator

transforms an abstract syntax tree to preordered intermediate code string

```
class Translator(object):
  def __init__(self, ast: Goal):
    # Store AST
    pass

  def _translate(self, tree: Goal) -> str:
    # Recursively translate each subtrees
    pass

  def translate(self) -> str:
    # call _translate()
    pass
```

Examples

input.txt

```
x-2*y
a + 35 - b
10+*5
```

output.txt

```
-x*2y
+a-35b
incorrect syntax
```

Test

Command line(bash)

```
python3 pred_parser/run.py test/input.txt temp/output.txt
python3 test/run.py test/output.txt temp/output.txt
```

Windows Script

```
.\test.bat
```

Shell Script

```
sh ./test.sh
```

Result

- "COMPILE SUCCESS" -> temp/output.txt == test/output.txt
- "COMPILE FAILURE" -> temp/output.txt != test/output.txt