Department of Computer Science
Reykjavik University

Homework 1
Spring 2025

Programming Languages
Homework 1: Compiler and Interpreter

This programming assignment, which you should implement in **Python**, is based on material that has been covered in the first four weeks of the course, i.e. basic elements, abstract machines, grammars, and lexical analysis and parsing.

Description

The project is twofold:

1) Implement a compiler which compiles a program in a simple programming language `L` to stack-based intermediate code `S`. The intermediate code `S` runs in an abstract machine which is implemented in part 2. A string in the language `L` consists of a list of statements, where each statement is either an assignment statement or a print statement.

   Context-free grammar `G` *for* `L` *is*:

   Statements -> Statement **;** Statements | **end**
   Statement -> **id** **=** Expr | **print id**
   Expr- > Term | Term **+** Expr
   Term -> Factor | Factor ***** Term
   Factor -> **int** | **id** | **-** Factor **| (** Expr **)**

   Non-terminals are:
   Statements (start symbol), *Statement*, *Expr*, *Term*, and *Factor.*

   Tokens/terminals are:
   ; end id = print + - * int ( )

   The intermediate code `S` consists of the following commands:

PUSH op	// pushes the operand op onto the stack
ADD	// addition: pops the two top elements from the stack, adds their // values and pushes the result back onto the stack
MULT	// multiplication: pops the two top elements from the stack, // multiplies their values and pushes the result back onto the stack
UMINUS	// unary minus: pops the top element from stack, changes its sign // and pushes the result back onto the stack
ASSIGN	// assignment: pops the two top elements from the stack, assigns // the first element (a value) to the second element (a variable)
PRINT	// prints the value currently on top of the stack

2) Implement an abstract machine for `S`, i.e. an interpreter which makes it possible to run code written in `S`.

Implementation

You **must** use the following guidelines for the implementation (steps 1-3 correspond to the compiler, step 4 to the interpreter).

1. Implement the class **LToken**, which contains both a lexeme (string) and a token code (integer constant).

   The integer constants are the following:
   { ID, ASSIGN, SEMICOL, INT, PLUS, MINUS, MULT, LPAREN, RPAREN, PRINT, END, ERROR}

   See an explanation for ERROR in step 2.

   Implement these constants as class variables in the class LToken. For example, it should be possible to refer to the ID constant with `LToken.ID`.

2. Implement the class **LLexer,** a lexical analyzer. It should contain a method, *get_next_token()*, which scans the **standard input** (stdin), looking for patterns that match one of the tokens from 1). Use **sys.stdin.read(1)** to read the input, character by character.
   Note that the lexemes corresponding to the tokens PLUS, MINUS, MULT, LPAREN, RPAREN, ASSIGN, SEMICOL contain only a single letter. The patterns for the lexemes for the other tokens are:

   INT = [0-9]+
   ID = [A-Za-z]+
   END = end
   PRINT = print

   The lexical analyzer returns the token **ERROR** if some illegal lexeme is found.

   When you have implemented the LLexer class, you can test it with the program *test_lexer.py* which is given with the project description:

   ```
   python test_llexer.py < program.l
   ```
   (here program.l is a program in L)

3. Implement the class **LParser**, which is a syntax analyzer (parser). It should be implemented as a recursive-descent parser for the grammar *G* above. The output of

the parser is the stack-based intermediate code `S`, written to **standard output** (stdout). One empty line should be at the end of the intermediate code. You need to figure out where in the parser the individual intermediade code commands should be written out.

If the program being compiled is not valid according to the grammar, or if the lexical analyzer returns an ERROR token, the parser should print"Syntax error" (at the point where the error is found) and quit running.

The parser should have at least two instance variables, one of type **LLexer**, the other of type L**Token** (for the current token). The parser should contain the methods, *parse(),* for initiating the parse, and *next_token()* which gets the next token from the lexical analyzer and prints an error message if needed:

```
def parse(self):
    self.next_token()
    self.statements()
    print() # Make sure the intermediate code ends with
a newline

def next_token(self):
    self.curr_token = self.lexer.get_next_token()
    if self.curr_token.token_code == LToken.ERROR:
        self.error()
```

In addition, the parser of course contains various other functions corresponding to the grammar.

4. Implement the class **SInterpreter**. `sinterpreter.py` also contains a main program which does the following:

```
interpreter = SInterpreter()
interpreter.cycle()
```

cycle() performs the *fetch-decode-execute cycle*, i.e. reads the intermediate code *S* from **standard input** (stdin) line by line, decodes each command and calls operations in `SInterpreter` to run the corresponding command. The output is written to **standard output**.

If `SInterpreter` encounters an invalid operator, or if there are not sufficiently many arguments for an operator on the stack, then it should write out the error message: "Error for operator: nameOfOperator" (where nameOfOperator is the operator in question) and immediately quit.

Use a *stack* in `SInterpreter` to process the intermediate code (you can simply use a *list* í Python for a stack) and use a *dictionary* to store the values of variables. Default value for a variable is 0.

Running/Testing

The main program **lcompiler.py** which is given with the problem description does the following:

```
lexer = ELexer()
parser = EParser(lexer)
parser.parse()
```

The input into **lcompiler.py** is a program written in the language `L`. The output is the corresponding intermediate code `S` written to standard output. Let us, for example, assume that the following program, written in `L`, is in the file `program.l`

```
var = 3;
b = 4 * (7 + var);
cc = -b;
print cc;
end
```

Then the following command compiles the program:

```
python lcompiler.py < program.l
```

and the following intermediate code is written to standard output:

```
PUSH var
PUSH 3
ASSIGN
PUSH b
PUSH 4
PUSH 7
PUSH var
ADD
MULT
ASSIGN
PUSH cc
PUSH b
UMINUS
ASSIGN
PUSH cc
PRINT
```

<- Here is an empty line

The output from `lcompiler.py` can of course be redirected to a file and then interpreted:

```
python lcompiler.py < program.l > program.s
python sinterpreter.py < program.s
-40
```

Or:

```
python lcompiler.py < program.l | python sinterpreter.py
-40
```

Examples of illegal programs in L:

1) program.l:

```
var = 3 + ;
print var;
end
```

```
python lcompiler.py < program.l
PUSH var
PUSH 3
Syntax error
```

2) program.l:

```
var = 3 ! ;
print var;
end
```

```
python lcompiler.py < program.l
PUSH var
PUSH 3
Syntax error
```

What to return:

- The following four files (NOT as a single .zip file): ltoken.py, llexer.py, lparser.py, and sinterpreter.py.