COMP 141: Course Project
Phase 3.2: Evaluator for L_{imp}

Over the course of the six phases, we will construct an interpreter for a small imperative language, called L_{imp}. In this last phase, you will construct the evaluator module for L_{imp}. The evaluator receives the AST of a program and returns the result of evaluation

Scanner Specification There are four types of tokens in our language, defined by the following regular expressions.

```
IDENIFIER = ([a-z] | [A-Z])([a-z] | [A-Z] | [0-9])*
   NUMBER = [0-9]+
   SYMBOL = \+  |  \-  |  \*  |  /  |  \(  |  \)  |  :=  |  ;
  KEYWORD = if  |  then  |  else  |  endif  |  while  |  do
          |  endwhile  |  skip
```

The following rules define how separation between tokens should be handled.

- White space[1] is not allowed in any token, so white space always terminates a token and separates it from the next token. Except for indicating token boundaries, white space is ignored.
- The principle of longest substring should always apply.
- Any character that does not fit the pattern for the token type currently being scanned immediately terminates the current token and begins the next token. The exception is white space, in which case the first rule applies.

Parser Specification Grammar of L_{imp} is defined as follows:

$$
\begin{aligned}
\textit{statement} &::= \textit{basestatement}\ \{\ \texttt{;}\ \textit{basestatement}\ \}\\
\textit{basestatement} &::= \textit{assignment} \mid \textit{ifstatement} \mid \textit{whilestatement} \mid \texttt{skip}\\
\textit{assignmet} &::= \texttt{IDENTIFIER}\ \texttt{:=}\ \textit{expression}\\
\textit{ifstatement} &::= \texttt{if}\ \textit{expression}\ \texttt{then}\ \textit{statement}\ \texttt{else}\ \textit{statement}\ \texttt{endif}\\
\textit{whilestatement} &::= \texttt{while}\ \textit{expression}\ \texttt{do}\ \textit{statement}\ \texttt{endwhile}
\end{aligned}
$$

$$
\begin{aligned}
\textit{expression} &::= \textit{term}\ \{\ +\ \textit{term}\ \}\\
\textit{term} &::= \textit{factor}\ \{\ -\ \textit{factor}\ \}\\
\textit{factor} &::= \textit{piece}\ \{\ /\ \textit{piece}\ \}\\
\textit{piece} &::= \textit{element}\ \{\ *\ \textit{element}\ \}\\
\textit{element} &::= \texttt{(}\ \textit{expression}\ \texttt{)} \mid \texttt{NUMBER} \mid \texttt{IDENTIFIER}
\end{aligned}
$$

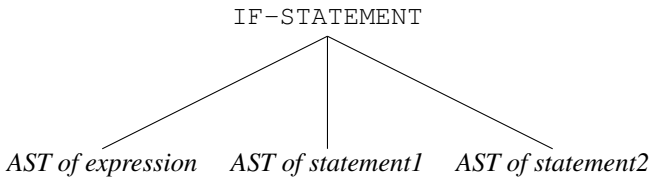
[1] tabs, spaces, new lines

Note that *statement* is the starting nonterminal.

Abstract syntax trees Consider the following details regarding the generated abstract syntax tree.

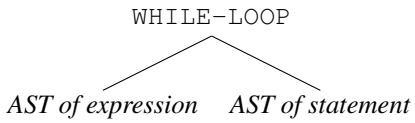
- All binary operations (numerical, sequencing (`;`) and assignment (`:=`)) must denote a tree with two subtrees, and the operator at the root.
- All parentheses must be dropped in the abstract syntax trees.
- If-statements must be represented by a tree with three subtrees, where the first subtree is corresponding to the expression, the second subtree is corresponding to the statement after `then` keyword, and the third subtree is corresponding to the statement after `else` keyword. The root of the tree must indicate that this is an if-statement. Therefore, the keywords `if`, `then`, `else`, and `endif` must be dropped in the generated abstract syntax tree.

  In general, the AST for `if` *expression* `then` *statement1* `else` *statement2* `endif` is as follows:



- While-statements must be represented by a tree with two subtrees, where the first subtree is correspoding to the expression, and the second subtree is corresponding to the statement within the while loop. The root of the tree must indicate that this is a while-statement. Therefore, the keywords `while`, `do`, and `endwhile` must be dropped from the abstract syntax tree.

  In general, the AST for `while` *expression* `do` *statement* `endwhile` is as follows:



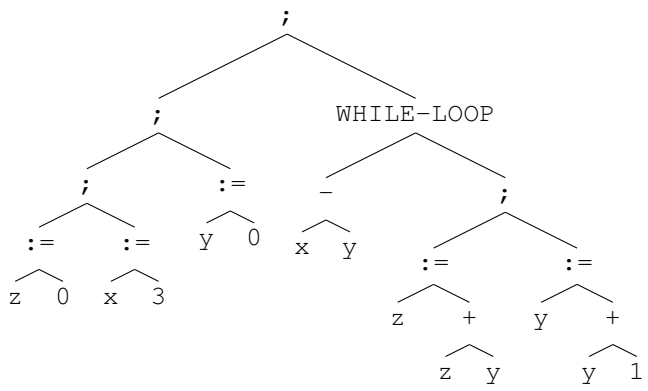
- The AST for `skip` statement consists of a single leaf node for the skip token.
- All of the nonterminals must also be dropped in the abstract syntax tree.

Example Consider the following program in L_{imp}. Let's call this program prog:

```
z := 0;
x := 3;
y := 0;

while x-y do
        z := z + y;
        y := y + 1
endwhile
```

The AST returned by parser would be as follows, which is the input top the evaluator.



Evaluator Specification Evaluator consists of three major data structures:

1. a tree that represents the partially evaluated program, t
2. a mapping that represents the memory (i.e., it maps identifiers to values), m
3. a stack to evaluate expressions, s.

The AST resulted from the parsing phase must be passed to the evaluator, initially. Evaluator iteratively searches the AST to find the **leftmost subtree associated with a base statement**, and evaluates it. Each step of evaluation may modifies the AST. Along with the modification of the tree, the memory is updated (through assignments). Memory can be implemented by dictionaries, where the dictionary keys are the identifiers and the dictionary values are the mapped values.

Consider the following for the implementation of the evaluator:

- Each subtree representing an expression must be evaluated to return a value. That value must be substituted with the whole subtree that corresponds to the expression. Stack s is used to evaluate expressions (phase 3.1).
- Identifiers retain the value when they appear in an expression. Therefore, upon evaluation of an expression, using stack s, the associated value of the identifier must be pushed into s.
- Identifiers must be held in the memory.
- Memories are maps or dictionaries from identifiers to values.
- Identifiers are added to the memory when they appear for the first time on the LHS of an assignment.
- The mapped value for an identifier (retained in memory) is set by an assignment.
- There is not any scoping in this language, i.e., all identifiers are global (i.e., they are accessible from any point in the program).
- Values evaluate as expected, e.g., `56` evaluates to number 56, and `26` evaluates to number 26.
- Identifiers get their value from look-up in the memory. If an identifier is not already added to the memory but used in an expression, evaluator must raise an exception.
- Operators are applied to the values, returned by their subtrees.
- Operators evaluate as expected[2], i.e.,
  - + denotes numerical addition.

[2]There is one exceptions, however. Subtracting larger number from a smaller number results in 0, rather than a negative number, e.g., 3 - 6 should be evaluated to 0. This is due to the fact that the language is not supporting negative numbers.

  - `*` denotes numerical multiplication.
  - `-` denotes numerical subtraction. There are no negative numbers in this language. Refer to lexical structure for clarification.
  - `/` denotes division on nonnegative integers, e.g., `3/2` evaluates to `1`. In the case of division by zero, the evaluator must stop and raise an exception.

- Statements are evaluated as follows:
  - Assignment: Evaluate RHS expression (using stack) and store result in memory entry for the LHS identifier. Assignment adds an entry to the memory if there does not exist any memory entry for the LHS identifier. Finally, remove the subtree that corresponds to the assignment.
  - If statement: Evaluate the expression first (using the stack).
    * If the expression evaluates to a positive number then replace the whole subtree associated with the if-statement with the subtree that corresponds to the statement after `then`.
    * If the expression evaluates to 0 then replace the whole subtree associated with the if-statement with the subtree that corresponds to the statement after `else`.
  - While loop: Evaluate the guard expression (using the stack).
    * If it evaluates to a positive number then substitute the whole while loop subtree with a sequencing subtree, where the left child corresponds to the statement in the body of the while loop, and the right child of sequencing operator corresponds to the original while loop.
    * If it evaluates to 0 then remove the whole subtree that corresponds to the while loop.
  - Skip: Remove the subtree that corresponds to `skip`. (There is nothing to do for this statement).
  - Sequencing: First evaluate the statement on the LHS of sequencing operator. Next replace the whole tree that is associated with sequencing with the subtree associated with the statement on the RHS of the sequencing operator.

Example evaluation Refer to the document that discusses a step-by-step evaluation for prog.

Language Selection

- You can build your interpreter in any language that you like.
- For more obscure languages, you will be responsible for providing instructions detailing how the program should be compiled or interpreted. Check with the instructor for the level of detail necessary for a particular language.

Input and Output Requirements Your full program (scanner module, parser module and evaluator) must read from an input file and write an output file. These files should be passed to the program as an argument in the command line. For example, if your parser is a python program in parser.py, we will test it using a command like:

```
python interpreter.py test_input.txt test_output.txt
```

The interpreter's output should be sent to the output file. The output should include:

1. The list of tokens produced by the scanner.
2. The abstract syntax tree produced by the parser.
3. The state of the memory at the end of the interpreted program. ***There is no need to output the state of memory in each intermediary step. Final state of the memory suffices.***

Example output of interpreter The output for prog

```
Tokens:
IDENTIFIER z
SYMBOL :=
NUMBER 0
SYMBOL ;
IDENTIFIER x
SYMBOL :=
NUMBER 3
SYMBOL ;
IDENTIFIER y
SYMBOL :=
NUMBER 0
SYMBOL ;
KEYWORD while
IDENTIFIER x
SYMBOL -
IDENTIFIER y
KEYWORD do
IDENTIFIER z
SYMBOL :=
IDENTIFIER z
SYMBOL +
IDENTIFIER y
SYMBOL ;
IDENTIFIER y
SYMBOL :=
IDENTIFIER y
SYMBOL +
NUMBER 1
SYMBOL ;
KEYWORD endwhile


AST:
SYMBOL ;
        SYMBOL ;
                SYMBOL ;
                        SYMBOL :=
                                IDENTIFIER z
                                NUMBER 0
                        SYMBOL :=
                                IDENTIFIER x
                                NUMBER 10
                SYMBOL :=
                        IDENTIFIER y
                        NUMBER 0
        WHILE-LOOP
                SYMBOL -
                        IDENTIFIER x
                        IDENTIFIER y
                SYMBOL ;
                        SYMBOL :=
```

```
                IDENTIFIER z
                SYMBOL +
                        IDENTIFIER z
                        IDENTIFIER y
        SYMBOL :=
                IDENTIFIER y
                SYMBOL +
                        IDENTIFIER y
                        NUMBER 1
```

```
Output:
z = 3
x = 3
y = 3
```

Error Handling Requirements If your scanner encounters an error while processing an input line, it should abort in a graceful manner. An error message and the input line that caused the error should be printed in the output file. After the error is reported, the program should shut down.

If the parser encounters an error, it should abort in a graceful manner. An error message and the token that caused the error should be printed in the output file. After the error is reported, the program should shut down.

If the evaluator encounters an error, e.g., when it wants to evaluate `2 / 0`, it should report a simple error message and shut down. There is no need to be explicit about the reason for error.

Submission Format Requirements Your submission should include the source code for your scanner module, parser module and test driver, along with a text file containing instructions for building and running your program. Every source code file should have your name and the phase number in a comment at the top of the file. The instruction file must at least identify the compiler that you used for testing your program.

Submit an archive file (.zip) containing the code and instruction file.