COMP 141: Course Project
Phase 2.2: Scanner for L_{imp}

Over the course of the six phases, we will construct an interpreter for a small imperative language, that we call L_{imp}.

For the current phase you will construct the parser module for L_{imp}, for which you have already developed a scanner in phase 1.2. A parser processes the stream tokens produced by the scanner according to the language grammar. The result will be an abstract syntax tree (AST) representing the parsed code.

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.

[1] tabs, spaces, new lines

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

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

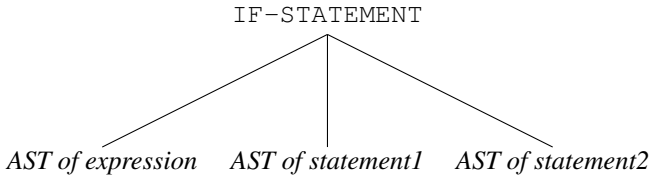
$$
\begin{aligned}
\textit{expression} &::= \textit{term}\ \{\ +\ \textit{term}\ \} \\
\textit{term} &::= \textit{factor}\ \{\ -\ \textit{factor}\ \} \\
\textit{factor} &::= \textit{piece}\ \{\ /\ \textit{piece}\ \} \\
\textit{piece} &::= \textit{element}\ \{\ \star\ \textit{element}\ \} \\
\textit{element}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

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:

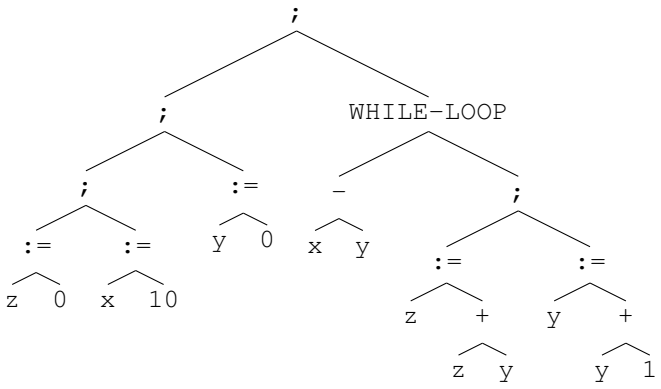
- The AST for `skip` statement consists of a single leaf node for the skip token.
- All of the nonterminals must also be dropped in the abstract syntax tree.

Example Consider the following program in this language:

```
z := 0;
x := 10;
y := 0;

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

The generated abstract syntax tree according to the defined grammar is as follows:



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 test driver) 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 parser.py test_input.txt test_output.txt
```

Your test driver program should be as follows:

Loop:

Read a line from input file.

Pass the input line to the scanner and receive the list of tokens.

Outside the loop, print the tokens in the list to the output file, in order, one token per line. Each line should report the token type and the token value.

Parse the list of tokens and generate the abstract syntax tree.

Print the abstract syntax tree to the output file.

Printing an abstract syntax tree You should do a preorder traverse of the generated AST and use proper indentation to print the abstract syntax tree in the output file[2]. For example, consider the AST given earlier for a piece of code. The printed version of this tree is:

[2]Note that the inorder traverse gives almost the text of the code.

```
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
```

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.

Example program I/O Assume that the input program consists of the following:

```
z := 0;
x := 10;
y := 0;

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

Then the output may be as follows:

```
Tokens:
IDENTIFIER z
SYMBOL :=
NUMBER 0
SYMBOL ;
IDENTIFIER x
```

```
SYMBOL :=
NUMBER 10
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
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
```

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.