Part 1 - Recursive Descent Parser (80 points)

We've seen VM Code and how that can be translated to Assembly and Machine code, but these languages are represented as basic sequences of instructions -- how do we handle the nested and varied structures of high-level programming languages?

Using your preferred programming language (Python, C++ or Java) implement the CompilerParser as described below. This practical assignment follows a similar approach to the Nand2Tetris Compilation Engine.

- Template files are provided for each of these programming languages.
  - Download the Python version HERE .
  - Download the Java version HERE .
  - Download the C++ version HERE .
- You will need to complete the methods provided in the CompilerParser class.
- The provided `ParseTree` & `Token` classes should not be modified.
- Only submit files for 1 programming language.

Getting Started

1. Start by reviewing chapter 10 of the textbook.
2. Each of the methods listed below needs to apply the corresponding set of grammar rules to the series of tokens given.
   For each set of these grammar rules:
   - A new parse tree is created.
   - The tokens are processed 1-by-1.
   - Tokens matching the grammar rule are added to a ParseTree for that rule.
   - If the rules are broken (i.e. the sequence of tokens does not match the rules), a ParseException should be thrown/raised.
   - Otherwise the ParseTree data structure is returned.
   - Some of the sets grammar rules require other sets of grammar rules.
     For example, the whileStatement rule requires the rules for expression and statements.
     These rule sets should be applied recursively.
3. A ParseTree data structure is returned

Tokens

Each token has a type and corresponding value.

Tokens can have the following types and possible values:

Token Type	Value

`keyword`	`'class'|'constructor'|'function'|'method'|'field'|'static'|`
	`'var'|'int'|'char'|'boolean'|'void'|'true'|'false'|'null'|'this'|`
	`'let'|'do'|'if'|'else'|'while'|'return'|'skip'`

`symbol`	`'{'|'}'|'('|')'|'['|']'|'.'|','|';'|'+'|'-'|'*'|'/'|'&'|'|'|'<'|'>'|'='|'~'`
`integerConstant`	A decimal integer in the range 0..32767
`stringConstant`	`'"'` A sequence of characters not including double quote or newline `'"'`
`identifier`	A sequence of letters, digits, and underscore (`'_'`), not starting with a digit.

We can read the type of the token with the `Token.getType()` method, and its value with `Token.getValue()`

Parse Trees

Each node in the ParseTree has a type, a value, and a list of children (parse trees nested inside this tree).

When creating a ParseTree, we set the type and value in the constructor. We can then add parse trees via the `ParseTree.addChild(ParseTree)` method. If needed, we can read the type of the ParseTree with the `ParseTree.getType()` method, and its value with `ParseTree.getValue()`.

To review the structure of a ParseTree object, it can be printed; this will output a human readable representation.

ParseTrees can have the following types which correspond with a set of grammar rules:

Parse Tree Type	Grammar Rule
`class`	$\texttt{'class'}\ className\ \texttt{'\{'}\ classVarDec^*\ subroutineDec^*\ \texttt{'\}'}$
`classVarDec`	$(\texttt{'static'}\|\texttt{'field'})\ type\ varName\ (\texttt{','}\ varName)^*\ \texttt{';'}$
`subroutine`	$(\texttt{'constructor'}\|\texttt{'function'}\|\texttt{'method'})(\texttt{'void'}\|type)\ subroutineName$ $\texttt{'('}\ parameterList\ \texttt{')'}\ subroutineBody$
`parameterList`	$((type\ varName)\ (\texttt{','}\ type\ varName)^*)?$
`subroutineBody`	$\texttt{'\{'}\ varDec^*\ statements\ \texttt{'\}'}$
`varDec`	$\texttt{'var'}\ type\ varName\ (\texttt{','}\ varName)^*\ \texttt{';'}$
`statements`	$statement^*$ where *statement* matches the following rule: $letStatement \mid ifStatement \mid whileStatement \mid doStatement \mid returnStatement$
`letStatement`	$\texttt{'let'}\ varName(\texttt{'['}\ expression\ \texttt{']'})?\ \texttt{'='}\ expression\ \texttt{';'}$
`ifStatement`	$\texttt{'if'}\ \texttt{'('}\ expression\ \texttt{')'}\ \texttt{'\{'}\ statements\ \texttt{'\}'}\ (\texttt{'else'}\ \texttt{'\{'}\ statements\ \texttt{'\}'})?$
`whileStatement`	$\texttt{'while'}\ \texttt{'('}\ expression\ \texttt{')'}\ \texttt{'\{'}\ statements\ \texttt{'\}'}$
`doStatement`	$\texttt{'do'}\ expression\ \texttt{';'}$
`returnStatement`	$\texttt{'return'}\ (expression)?\ \texttt{';'}$
`expression`	$\texttt{'skip'}\|(term\ (op\ \ term)^*)$ Note the addition of the **skip** keyword
`term`	$integerConstant \mid stringConstant \mid keywordConstant \mid varName \mid$ $varName\texttt{'['}expression\texttt{']'} \mid \texttt{'('}expression\texttt{')'} \mid (unaryOp\ term) \mid subroutineCall$
`expressionList`	$(expression(\texttt{','}\ expression)^*)?$

Which match the methods we're implementing.

They can also have the same types as listed above for Tokens (and Tokens can be added as children to ParseTrees via typecasting)

You may have noticed that some grammar elements shown above and in the Jack Grammar are missing from this list. These rules are listed below. They should be used as part of the rules above, but are not themselves ParseTree types:

`returnStatement`	$\texttt{'return'}\ (expression)?\ \texttt{';'}$
`expression`	$\texttt{'skip'} \mid (term\ (op\ \ term)^*)$ Note the addition of the **skip** keyword
`term`	$integerConstant \mid stringConstant \mid keywordConstant \mid varName \mid$ $varName\texttt{'['}expression\texttt{']'} \mid \texttt{'('}expression\texttt{')'} \mid (unaryOp\ term) \mid subroutineCall$
`expressionList`	$(expression(\texttt{','}\ expression)^*)?$

Which match the methods we're implementing.

They can also have the same types as listed above for Tokens (and Tokens can be added as children to ParseTrees via typecasting)

You may have noticed that some grammar elements shown above and in the Jack Grammar are missing from this list. These rules are listed below. They should be used as part of the rules above, but are not themselves ParseTree types:

Grammar Element	Grammar Rule
`className`	$identifier$
`varName`	$identifier$
`subroutineName`	$identifier$
`type`	$\texttt{'int'}\mid\texttt{'char'}\mid\texttt{'boolean'}\mid className$
`op`	$\texttt{'+'}\mid\texttt{'-'}\mid\texttt{'*'}\mid\texttt{'/'}\mid\texttt{'\&'}\mid\texttt{'\|'}\mid\texttt{'<'}\mid\texttt{'>'}\mid\texttt{'='}$
`unaryOp`	$\texttt{'-'}\mid\texttt{'\~{}'}$
`keywordConstant`	$\texttt{'true'}\mid\texttt{'false'}\mid\texttt{'null'}\mid\texttt{'this'}$
`subroutineCall`	$subroutineName\texttt{'('}expressionList\texttt{')'} \mid (className \mid varName)\texttt{'.'}subroutineName$

Suggested Approach

A suggested approach is outlined in section 10.1.4 of the Text book.

This involves writing a `process(token)` method which:

- Checks if the next token in the list of tokens matches an expected token
  - If the token matches, add it to the ParseTree
  - If the token does not match, throw/raise a ParseError
- Advances to the next token (if needed)
  - This can be done by removing/popping the token from the list

Task 1.1 - Program Structure (40 points)

Complete the program structure related methods:

- `compileProgram`

Jack Code	Tokens	Returned ParseTree Structure
`class Main {` `}`	keyword class identifier Main symbol { symbol }	◦ class ▪ keyword class ▪ identifier Main ▪ symbol { ▪ symbol }
`static int a ;`	keyword static keyword int identifier a symbol ;	ParseError (the program doesn't begin with a class)

- `compileClass`

Example Jack Code	Tokens	Returned ParseTree Structure

`class Main {`
`  static int a ;`
`}`
	keyword class
	identifier Main
	symbol {
	keyword static
	keyword int
	identifier a
	symbol ;
	symbol }
		◦ class
		▪ keyword class
		▪ identifier Main
		▪ symbol {
		▪ classVarDec
		▪ ...
		see classVarDec below
		▪ symbol }

- `compileClassVarDec`

Example Jack Code	Tokens	Returned ParseTree Structure
`static int a ;`	keyword static keyword int identifier a symbol ;	◦ classVarDec ▪ keyword static ▪ keyword int ▪ identifier a ▪ symbol ;

`compileSubroutineBody`

Example Jack Code	Tokens	Returned ParseTree Structure

`{`
`  var int a ;`
`  let a = 1 ;`
`}`
	symbol {
	keyword var
	keyword int
	identifier a
	symbol ;
	keyword let
	identifier a
	symbol =
	integerConstant 1
	symbol ;
	}
		◦ subroutineBody
		▪ symbol {
		▪ varDec
		▪ ...
		(see varDec below)
		▪ statements
		▪ ...
		(see statements below)
		▪ symbol }

`compileVarDec`

Example Jack Code	Tokens	Returned ParseTree Structure
`var int a ;`	keyword var keyword int identifier a symbol ;	◦ varDec ▪ keyword var ▪ keyword int ▪ identifier a ▪ symbol ;

- `compileSubroutine`

Example Jack Code	Tokens	Returned ParseTree Structure

`function void myFunc ( int a ) {`
`  var int a ;`
`  let a = 1 ;`
`}`
	keyword function
	keyword void
	identifier myFunc
	symbol (
	keyword int
	identifier a
	symbol)
	symbol {
	keyword var
	keyword int
	identifier a
	symbol ;
	keyword let
	identifier a
	symbol =
	integerConstant 1
	symbol ;
	}
		◦ subroutine
		▪ keyword function
		▪ keyword void
		▪ identifier myFunc
		▪ symbol (
		▪ parameterList
		▪ ...
		(see parameterList below)
		▪ symbol)
		▪ subroutineBody
		▪ ...
		see subroutineBody below

- `compileParameterList`

Example Jack Code	Tokens	Returned ParseTree Structure
`int a, char b`	keyword int identifier a symbol , keyword char identifier b	◦ parameterList ▪ keyword int ▪ identifier a ▪ symbol , ▪ keyword char ▪ identifier b

Task 1.2 - Statements (40 points)

Complete the statement related methods:

- `compileStatements`

Example Jack Code	Tokens	Returned ParseTree Structure

`let a = skip ;`
`do skip ;`
`return ;`
	keyword let
	identifier a
	symbol =
	keyword skip
	symbol ;
	keyword do
	keyword skip
	symbol ;
	keyword return
	symbol ;
		◦ statements
		▪ letStatement
		▪ ...
		(see letStatement below)
		▪ doStatement
		▪ ...
		(see doStatement below)
		▪ returnStatement
		▪ ...
		(see doStatement below)

- `compileLet`

Example Jack Code	Tokens	Returned ParseTree Structure
`let a = skip ;`	keyword let identifier a symbol = keyword skip symbol ;	◦ letStatement ▪ keyword let ▪ identifier a ▪ symbol = ▪ expression ▪ ... see expression below ▪ symbol ;

- `compileIf`

Example Jack Code	Tokens	Returned ParseTree Structure

`if ( skip ) {`
`} else {`
`}`
	keyword if
	symbol (
	keyword skip
	symbol)
	symbol {
	symbol }
	keyword else
	symbol {
	symbol }
		◦ ifStatement
		▪ keyword if
		▪ symbol (
		▪ expression
		▪ ...
		see expression below
		▪ symbol)
		▪ symbol {
		▪ statements
		▪ ...
		▪ symbol }
		▪ keyword else
		▪ symbol {
		▪ statements
		▪ ...
		▪ symbol }

- `compileWhile`

Example Jack Code	Tokens	Returned ParseTree Structure
`while ( skip ) {` `}`	keyword while symbol (keyword skip symbol) symbol { symbol }	◦ whileStatement ▪ keyword while ▪ symbol (▪ expression ▪ ... see expression below ▪ symbol) ▪ symbol { ▪ statements ▪ ... ▪ symbol }

- `compileDo`

Example Jack Code	Tokens	Returned ParseTree Structure
`do skip ;`	keyword do keyword skip symbol ;	◦ doStatement ▪ keyword do ▪ expression ▪ ... see expression below ▪ symbol ;

- `compileReturn`

Example Jack Code	Tokens	Returned ParseTree Structure
`return skip ;`	keyword return keyword skip symbol ;	◦ returnStatement ▪ keyword return ▪ expression ▪ ... see expression below ▪ symbol ;

For some of the above methods, you will also need to partially implement the compileExpression method below. At this stage, implement the compileExpression to match the grammar rule **`'skip'`**.

Task 1.3 - Expressions (Optional - up to 20 BONUS points)

Complete the expression related methods:

This section is optional and is worth Bonus Points

- `compileExpression`

Example Jack Code	Tokens	Returned ParseTree Structure
`skip`	keyword skip	◦ expression ▪ keyword skip
`1 + ( a - b )`	integerConstant 1 symbol + symbol (identifier a symbol - identifier b symbol)	◦ expression ▪ term ▪ ... see term below ▪ symbol + ▪ term ▪ ... see term below

- `compileTerm`

Example Jack Code	Tokens	Returned ParseTree Structure
`1`	integerConstant 1	◦ term ▪ integerConstant 1
`( a - b )`	symbol (identifier a symbol - identifier b symbol)	◦ term ▪ symbol (▪ expression ▪ term ▪ identifier a ▪ symbol - ▪ term ▪ identifier b ▪ symbol)

- `compileExpressionList`

Example Jack Code	Tokens	Returned ParseTree Structure
`1 , a - b`	integerConstant 1 symbol , identifier a symbol - identifier b	◦ expressionList ▪ expression ▪ ... see expression above ▪ symbol , ▪ expression ▪ ... see expression above