Assignment 7

- Due Friday by 23:59
- Points 80
- Submitting an external tool

Assessment Overview

Weighting:	80 Points (8% of course grade)
Due date:	Friday, 31 Oct 11:59pm
Task description:	Develop a Parser to convert high-level programming language into a parse tree. Doing so should help you to: • Practice applying grammar rules • Understand how complex and nested code structures can be broken down to their component parts. • Understand the basics of Recursive Descent Parsing. Please post your questions on Piazza or ask during your workshop.
Academic Integrity Checklist	Do ✅ Discuss/compare high level approaches ✅ Discuss/compare program output/errors ✅ Regularly submit your work as you progress Be careful ❓ Using online resources to find the solutions rather than understanding them yourself won't help you learn. Do NOT ❌ Submit code not solely authored by you. ❌ Use a public GitHub repository (use a private one instead). ❌ Post/share complete VM/Assembly/Machine code in Piazza/Discord or elsewhere on the Internet etc. ❌ Give/show your code to others

Your Task

Your task for this practical assignment is to write a parser to convert high-level language programs into a parse tree that can be later converted to VM Code.

1. Complete the Parser as described and as outlined below.
   - Submit your work regularly to Gradescope as you progress.
   - Additional resources and help will be available during your workshop sessions.
2. Test your code.

We're know that things are tight at the end of semester, so we've kept this assignment short (and hopefully simple).

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 (https://myuni.adelaide.edu.au/courses/101158/files/18016951?wrap=1) (https://myuni.adelaide.edu.au/courses/101158/files/18016951/download?download_frd=1)** .
  - Download the Java version **HERE (https://myuni.adelaide.edu.au/courses/101158/files/18016950?wrap=1) (https://myuni.adelaide.edu.au/courses/101158/files/18016950/download?download_frd=1)** .
  - Download the C++ version **HERE (https://myuni.adelaide.edu.au/courses/101158/files/18016949?wrap=1) (https://myuni.adelaide.edu.au/courses/101158/files/18016949/download?download_frd=1)** .
- 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:

<table>
<tr><th>Token Type</th><th>Value</th></tr>
<tr><td><code>keyword</code></td><td>'class'|'constructor'|'function'|'method'|'field'|'static'|
'var'|'int'|'char'|'boolean'|'void'|'true'|'false'|'null'|'t
'let'|'do'|'if'|'else'|'while'|'return'|'skip'</td></tr>
<tr><td><code>symbol</code></td><td>'{'|'}'|'('|')'|'['|']'|'.'|','|';'|'+'|'-'|'*'|'/'|'&'|'|'|'<'</td></tr>
<tr><td><code>integerConstant</code></td><td>A decimal integer in the range 0..32767</td></tr>
<tr><td><code>stringConstant</code></td><td>'"' A sequence of characters not including double quote or newline '"'</td></tr>
<tr><td><code>identifier</code></td><td>A sequence of letters, digits, and underscore ('_'), not starting with a digit.</td></tr>
</table>

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

You can assume that all tokens have been correctly tokenized (i.e. you will not have to check for and handle bad tokens)

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`	$(\texttt{'static'} \mid \texttt{'field'})\ type\ varName\ (\texttt{','}\ varName)^*\ \texttt{';'}$
`subroutine`	$(\texttt{'constructor'} \mid \texttt{'function'} \mid \texttt{'method'})(\texttt{'void'} \mid type)\ subroutineN$ $\texttt{'('}\ parameterList\ \texttt{')'}\ subroutineBody$
`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: $letStatement \mid ifStatement \mid whileStatement \mid doStatement \mid retur$
`letStatement`	$\texttt{'let'}\ varName(\texttt{'['}\ expression\ \texttt{']'})?\ \texttt{'='}\ expression\ \texttt{';'}$
`ifStatement`	$\texttt{'if'}\ \texttt{'('}\ expression\ \texttt{')'}\ \texttt{'\{'}\ statements\ \texttt{'\}'}\ (\texttt{'else'}\ \texttt{'\{'}\ statemer$
`whileStatement`	$\texttt{'while'}\ \texttt{'('}\ expression\ \texttt{')'}\ \texttt{'\{'}\ statements\ \texttt{'\}'}$
`doStatement`	$\texttt{'do'}\ expression\ \texttt{';'}$

`returnStatement`	$\texttt{'return'}\ (expression)?\ \texttt{';'}$
`expression`	$\texttt{'skip'} \mid (term\ (op\ \ term)^{*})$ Note the addition of the **skip** keyword
`term`	$integerConstant \mid stringConstant \mid keywordConstant \mid varName \mid varName\texttt{'['}expression\texttt{']'} \mid \texttt{'('}expression\texttt{')'} \mid (unaryOp\ term) \mid$
`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`	$\texttt{'+'} \mid \texttt{'-'} \mid \texttt{'*'} \mid \texttt{'/'} \mid \texttt{'\&'} \mid \texttt{'} \mid \texttt{'} \mid \texttt{'<'} \mid \texttt{'>'} \mid \texttt{'='}$
`unaryOp`	$\texttt{'-'} \mid \texttt{'\textasciitilde'}$
`keywordConstant`	$\texttt{'true'} \mid \texttt{'false'} \mid \texttt{'null'} \mid \texttt{'this'}$
`subroutineCall`	$subroutineName\texttt{'('}expressionList\texttt{')'} \mid (className \mid varName)\texttt{'}$

Suggested Approach

To help make this process easier, a suggested approach is to implement the 4 additional method signatures provided as discussed in lectures.

- The `next()` method
  - Advanced to the next token in the list of tokens.
- The `current()` method
  - Returns the current token in the list of tokens.
- The `have(type,value)` method
  - Checks if the current token in the list of tokens matches the given token type and/or value.
  - Returns true or false depending on if the current token matches.
- The `mustbe(type,value)` method
  - Checks if the current token in the list of tokens matches the given token type and/or value.
  - If the current token matches it is returned.
    - Advance to the next token before returning
  - If the token does not match, throw/raise a ParseException.

Guidance on implementing these will be added to the Week 11 workshop.

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	◦ class ▪ keyword class

}	symbol { keyword static keyword int identifier a symbol ; symbol }	▪ 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 ;

- `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

- `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 ;

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`

<table>
<tr><th>Example Jack Code</th><th>Tokens</th><th>Returned ParseTree Structure</th></tr>
<tr><td><code>1</code></td><td>integerConstant 1</td><td>term
integerConstant 1</td></tr>
<tr><td><code>(a - b)</code></td><td>symbol (
identifier a
symbol -
identifier b
symbol)</td><td>term
symbol (
expression
term
identifier a
symbol -
term
identifier b
symbol)</td></tr>
</table>

- `compileExpressionList`

<table>
<tr><th>Example Jack Code</th><th>Tokens</th><th>Returned ParseTree Structure</th></tr>
<tr><td><code>1 , a - b</code></td><td>integerConstant 1
symbol ,
identifier a
symbol -
identifier b</td><td>expressionList
expression
...
see expression above
symbol ,
expression
...
see expression above</td></tr>
</table>

Examples

See above

You're done!

Submit your work to Gradescope using the button below.

- You may submit via file upload or GitHub.
  - If using GitHub, ensure your repository is private.
- Your files should either be:
  - In the root of your submission (i.e. no subdirectory)
    ~ or ~

  - In a directory named `prac7`

Be sure to submit all files with each submission.

Additional Resources

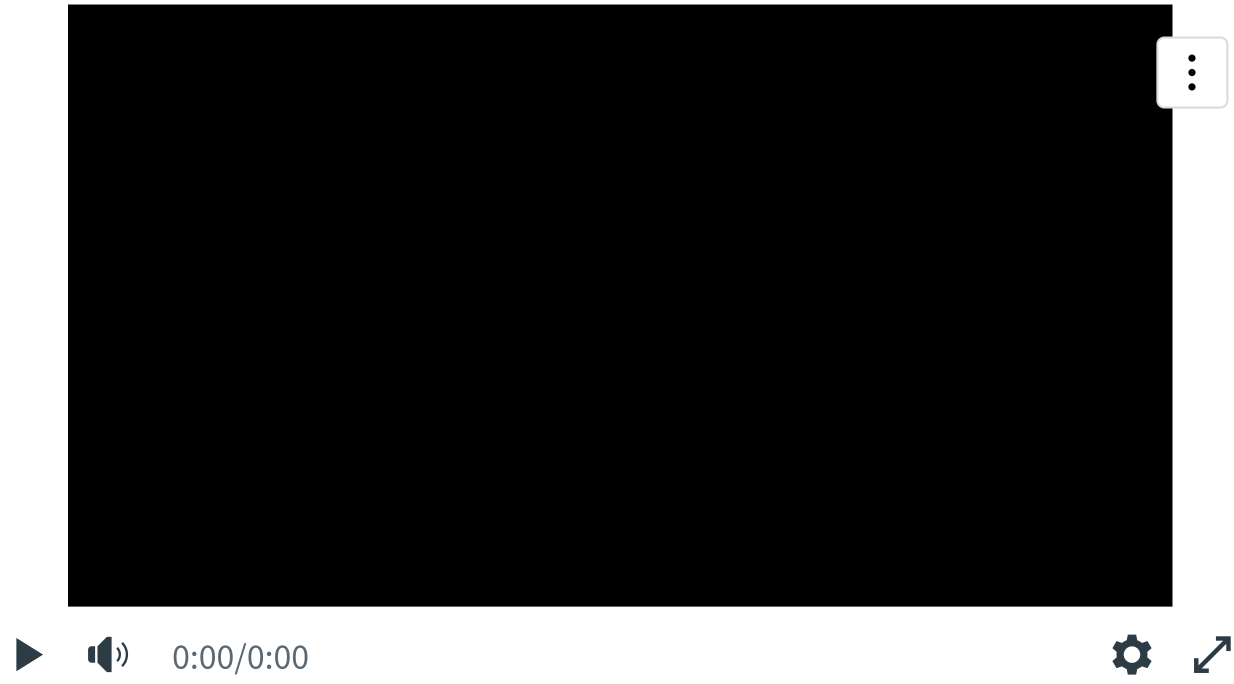
The following resources may help you complete this assignment:

- **[Chapter 10 of the Text Book](https://myuni.adelaide.edu.au/courses/101158/external_tools/1284)** **(https://myuni.adelaide.edu.au/courses/101158/external_tools/1284)** for Compiler Implementation
  - Section 10.1.4 includes basics of a suggested approach.
- Week 11 & 12 Workshops
- **[Guide to Testing and Writing Test Cases](https://myuni.adelaide.edu.au/courses/101158/pages/guide-to-testing-and-writing-test-cases)** **(https://myuni.adelaide.edu.au/courses/101158/pages/guide-to-testing-and-writing-test-cases)**
- **[Figure 10.5 on page 201 of the Text Book](https://myuni.adelaide.edu.au/courses/101158/external_tools/1284)** **(https://myuni.adelaide.edu.au/courses/101158/external_tools/1284)** for specification of the Jack Grammar.
- Further resources will be added over the coming days.

Some videos that provide guidance on how to approach Assignment 7 (Recursive Decent Parser for Jack) are provided below:

INTRODUCTION - Interpreting Assignment 7 & Relevant Course Content

TOKENIZER - Implementation of Token Helper Functions in the 'Suggested Approach' Section



PARSER - Example Implementations of compileProgram() & compileClass() Parse Methods from Task 1.1

TESTING & NOTES - Comments on Development & Testing as well as Cross-Referencing Course Content

This tool needs to be loaded in a new browser window

Load Assignment 7 in a new window