CS 4386.001, Compiler Design, Fall 2025
Project Assignment #2

Due: 11:59pm, November 7, 2025
Submission: *.tar.xz (or *.zip) via elearning
Maximum points: 100
The submitted file should be named by the assignment, your net id, and your first-last name, e.g., project02-xxx123456-jon-bell.zip

I Introduction

In this part of the project, we will work on implementing our parser. We will scan a file of tokens and translate it into an Abstract Syntax Tree.

Project 2 is built on top of Project 1. Project 1 sample solution has been provided, but you are expected to work from your own solution.

Important Tip: Provided in the project assignment is the ScannerTest.java. ScannerTest.java is the driver code and should replace the LexerTest.java file from your project 1.

The tutorial will go through parsing the simpler grammar to help you get a good understanding of this project assignment.

To build the project, JDK 11 or above is needed.

II The Grammar to Implement

Project 2 has been divided into two phases to facilitate organizing the project structure and determining the implementation order. However, only a single final submission is required (i.e., the one you completed w.r.t. the Phase 2 grammar). Several test files are provided for both phases. Use these to run your implementation and verify the correctness of your parser. You are only required to submit the implementation of Phase 2 and results of your Phase 2 tests; Phase 1 tests are for testing purpose only. It is strongly advised to adopt the bottom-up approach, as demonstrated in the tutorial, for implementing expressions and statements prior to other non-terminals specified in Phase 2.

(Phase 1)

Program	→	Stmts
Stmts	→	Stmt Stmts | λ
Stmt	→	if (Expr) { Stmts } IfEnd | while (Expr) { Stmts } | Name = Expr ;
	|	read (Readlist) ; | print (Printlist) ; | printline (Printlinelist) ;
	|	id ( ) ; | id (Args) ; | return ; | return Expr ; | Name ++ ; | Name -- ;

	|	{ Stmts } Optionalsemi
IfEnd	→	else { Stmts } | λ
Name	→	id | id [Expr]
Args	→	Expr , Args | Expr
Readlist	→	Name , Readlist | Name
Printlist	→	Expr , Printlist | Expr
Printlinelist	→	Printlist | λ
Expr	→	Name | id ( ) | id (Args) | intlit | charlit | strlit | floatlit | true | false
	|	(Expr) | ~ Expr | - Expr | + Expr | (Type) Expr | Expr Binaryop Expr | (Expr ? Expr : Expr)
Binaryop	→	* | / | + | - | < | > | <= | >= | == | <> | || | &&

Order of Operations:

(), []	Left to Right
(prefix)+, (prefix)-,~,++,--	Right to Left
(type cast)	Left to Right
*,/	Left to Right
+,-	Left to Right
<,>,<=,>=	Left to Right
<>,==	Left to Right
&&	Left to Right
‖	Left to Right
?:	Right to Left

(Phase 2)

Program	→	class id { Memberdecls }
Memberdecls	→	Fielddecls Methoddecls
Fielddecls	→	Fielddecl Fielddecls | λ
Methoddecls	→	Methoddecl Methoddecls | λ
Fielddecl	→	Optionalfinal Type id Optionalexpr ;
	|	Type id [intlit] ;
Optionalfinal	→	final | λ
Optionalexpr	→	= Expr | λ

```
Methoddecl   →   Returntype id ( Argdecls ) { Fielddecls Stmts } Optionalsemi
Optionalsemi →   ; | λ
Returntype   →   Type | void
Type         →   int | char | bool | float
Argdecls     →   ArgdeclList | λ
ArgdeclList  →   Argdecl , ArgdeclList | Argdecl
Argdecl      →   Type id | Type id [ ]
Stmts        →   Stmt Stmts | λ
Stmt         →   if ( Expr ) Stmt IfEnd | while ( Expr ) Stmt | Name = Expr ;
             |   read ( Readlist ) ; | print ( Printlist ) ; | printline ( Printlinelist ) ;
             |   id ( ) ; | id ( Args ) ; | return ; | return Expr ; | Name ++ ; | Name -- ;
             |   { Fielddecls Stmts } Optionalsemi
IfEnd    →   else Stmt | λ
Name         →   id | id [ Expr ]
Args         →   Expr , Args | Expr
Readlist     →   Name , Readlist | Name
Printlist    →   Expr , Printlist | Expr
Printlinelist →  Printlist | λ
Expr         →   Name | id ( ) | id ( Args ) | intlit | charlit | strlit | floatlit | true | false
             |   ( Expr ) | ~ Expr | - Expr | + Expr | ( Type ) Expr |
             |   Expr Binaryop Expr | ( Expr ? Expr : Expr )
Binaryop     →   * | / | + | - | < | > | <= | >= | == | <> | \|\| | &&
```

III Goal and Submission

The goal is to be able to identify that a language is being parsed correctly and that the states are transitioning in an expected manner. We can do this by outputting a formatted version of our language based on the abstract syntax tree that we created. If the abstract syntax tree is correct, then the output should be as expected. To show that the tree has the correct structure, make sure to have the formatted output: use tabs to indent all new scopes (bodies of methods, while loops, and if statements), and wrap all "Expr" nodes with parenthesis. A file named example_input.txt is provided along with matching output example_output.txt; these files provide an example of the formatting we expect for your output files. Several test files are provided for both phases. Use these to run your implementation and verify the correctness of your parser. However, you are only required to submit the results of your **Phase 2 tests**, along with a tar or zip archive containing your **Phase 2 project solution**. Besides the provided tests, additional advanced tests will be conducted to evaluate Project 2.