THE OPEN UNIVERSITY OF SRI LANKA
FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
BACHELOR OF SOFTWARE ENGINEERING HONOURS

EEX6363 – Compiler Construction

TMA — #1 Deadline on 10 June 2025 2024/2025

Question 1

Download the attachments 'LexAndYacc.pdf' and 'LexAndYaccCode.zip' from the Moodle page (or refer at `http://epaperpress.com/lexandyacc/index.html`) and study the files in the attachments to answer the followings:

(a) Summarize the procedures that the author has followed for the third type of calculator.
(b) Consider the following program segment:

```
p = 15; q = 3;
while (p > q) {
if p < q then
    p = p - q;
else
    q = p + q;
}
print q;
```

Apply the steps you summarized in (a) to obtain the *lex* and *yacc* files for the above program segment and attach the both files with output results each.

Question 2

Your task is to design and implement a lexical anlayzer for a programming language whose specifications are given below. The scanner identifies and outputs tokens (valid words and punctuation) in the source program. Its output is a token that can thereafter be used by the syntax analyzer to verify that the program is syntactically valid. When called, the lexical analyzer should extract the next token from the source program. The syntax of the language will be specified in TMA #2.

Punctuation		**Operators**		**Reserved words**		
(	;	+	<	else	integer	self
)	,	−	>	float	isa	construct
{	.	*	<=	func	private	then
}	:	/	>=	if	public	local
[	=>	:=	or	implement	read	void
]		==	and	class	return	while
		<>	not	attribute		write

Lexical elements

id	→	letter alphanum*
letter	→	**a . . . z** | **A . . . Z**
alphanum	→	letter | digit | _
digit	→	**0 . . . 9**
integer	→	nonzero digit* | **0**
nonzero	→	**1 . . . 9**
float	→	integer fraction [**e** [+ | –] integer]
fraction	→	**.** digit* nonzero | **.0**

Comments:

- Inline comments start with // and end with the end of the line they appear in.
- Block comments start with /* and end with */ and may span over multiple lines.

Answer to the following tasks:

(a) Identify the lexical specification, expressed as regular expressions, that you used in the design of the lexical analyzer.
(**Note:** You should **completely describe** the data structure of your design and any changes that you may have applied to the original lexical specifications should be also included.)

(b) Describe the **complete** operation of your lexical analyzer using finite state diagram.

(c) Write Lex (and Yacc) files to construct a lexical analyzer that recognizes the above-stated tokens. It should be a function that returns a token data structure containing the information about the next token identified in the source program file. The token data structure should contains: (1) the token type; (2) the lexeme; and (3) the location, of the token in the source code.
(**Note:** The parser that Yacc generates should be able to call the lexical analyzer generated by Lex and perform parsing of the code generated by the given grammar. Your Lex file should be able to remove any white space from the code. All correspondences should be stored in a symbol table and printed out as output.)

(d) Identify **all the possible** lexical errors that the lexical analyzer might encounter and the possible error recovery techniques you could use. Choose one of them and discuss why you have chosen it.

(e) Include many test cases that test a wide variety of valid and invalid cases.
(**Note:** The appropriate test cases will be used in later TMAs and Design Project.)

— End —

THE OPEN UNIVERSITY OF SRI LANKA
FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
BACHELOR OF SOFTWARE ENGINEERING HONOURS

EEX6363 – Compiler Construction

TMA — #2 Deadline on 17 August 2025 2024/2025

This assignment is about the design and implementation of a syntax analyzer for the language specified by the grammar below. The syntax analyzer should use as input the token stream produced by the lexical analyzer (you implemented in TMA #1), and prove whether or not the token stream is a valid program according to the grammar. While doing so, it should locate, report, and recover from eventual syntax errors.

Answer to the following tasks:

Use Lex/Yacc with a compatible programming language (prefer C/C++) for implementations.

(a) Transform the grammar into LL(1) by following operations:
Remove all the EBNF notations and replace them by right-recursive list-generating productions. Analyze and list all the ambiguities and left recursions. Modify the grammar so that the left recursions and ambiguities are removed without modifying the language. Include the set of productions that can be parsed using the top-down predictive parsing method, i.e. an LL(1) grammar. (**Note:** you should <u>not</u> modify the given language.)

(b) Derive the FIRST and FOLLOW sets for your transformed grammar and list them.

(c) Implement a predictive parser (recursive descent or table-driven) for your modified set of grammar rules and the parser should write to a file the derivation that derives the source program from the starting symbol. (**Note:** you should need to use your lexical analyzer developed in TMA – #1.)

(d) Check the syntax correctness of the program and give a overview of the **<u>overall</u>** structure of your solution, as well as a description of the role of **<u>each</u>** component of your implementation.

(e) Write a set of source files that enable to test the parser for all syntax structures involved.

The reference grammar is given in a variant of extended BNF as below. The meta-notation used:

- Non-terminals are represented in normal font.
- Terminals (lexical elements, or tokens) are represented in **bold font.**
- Double curly brackets without bold {{ ... }} represents repetition (i.e., zero or more occurrence) of the sentential form enclosed in the brackets.
- Double square brackets without bold [[...]] represents an optionality (i.e., either zero or one occurrence) of the sentential form enclosed in the brackets.
- The symbol ϵ means the empty string in the grammar.
- The symbol | is denoted the alternative productions in the defining grammar.
- **intLit** and **floatLit** follow specification for integer and float, respectively, given in TMA #1.

Grammar:

prog	→	{{ classOrImplOrFunc }}
classOrImplOrFunc	→	structDecl | implDef | funcDef
classDecl	→	**class id** [[**isa id** {{ **,** **id** }}]] **{** {{ visibility memberDecl }} **} ;**
implDef	→	**implement id {** {{funcDef}} **}**
funcDef	→	funcHead funcBody
visibility	→	**public** | **private**
memberDecl	→	funcDecl | attributeDecl
funcDecl	→	funcHead **;**
funcHead	→	**func id (** fParams **) =>** returnType | constructor **(** fParams **)**
funcBody	→	**{** {{varDeclOrStmt}} **}**
localVarDeclOrStmt	→	localVarDecl | statement
attributeDecl	→	**attribute** varDecl
localVarDecl	→	**local** varDecl
varDecl	→	**id :** type {{arraySize}} **;**
statement	→	assignStat **;** | **if (** relExpr **) then** statBlock **else** statBlock **;** | **while (** relExpr **)** statBlock **;** | **read (** variable **) ;** | **write (** expr **) ;** | **return (** expr **) ;** | functionCall **;**
assignStat	→	variable assignOp expr
statBlock	→	**{** {{statement}} **}** | statement | ϵ
expr	→	arithExpr | relExpr
relExpr	→	arithExpr relOp arithExpr
arithExpr	→	arithExpr addOp term | term
sign	→	**+** | **–**
term	→	term multOp factor | factor
factor	→	variable | functionCall | **intLit** | **floatLit** | **(** arithExpr **)** | **not** factor | sign factor
variable	→	{{idnest}} **id** {{indice}}
functionCall	→	{{idnest}} **id (** aParams **)**
idnest	→	idOrSelf {{indice}}**.** | idOrSelf **(** aParams **) .**
idOrSelf	→	**id** | **self**
indice	→	**[** arithExpr **]**
arraySize	→	**[intLit]** | **[ ]**
type	→	**integer** | **float** | **id**
returnType	→	type | **void**
fParams	→	**id :** type {{arraySize}} {{fParamsTail}} | ϵ
aParams	→	expr {{aParamsTail}} | ϵ
fParamsTail	→	**,** **id :** type {{arraySize}}
aParamsTail	→	**,** expr
assignOp	→	**:=**
relOp	→	**==** | **<>** | **<** | **>** | **<=** | **>=**
addOp	→	**+** | **–** | **or**
multOp	→	***** | **/** | **and**

THE OPEN UNIVERSITY OF SRI LANKA
FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
BACHELOR OF SOFTWARE ENGINEERING HONOURS

EEX6363 – Compiler Construction

TMA — #3 Deadline on 25 October 2025 2024/2025

This assignment is about the design and implementation of a semantic analyzer for the described language. First, complement the parser developed in TMA #2 with syntax-directed translation to generate an abstract syntax tree (AST) data structure. The implementation of the semantic analysis phase implies that you traverse the generated AST, and trigger semantic actions related to two inter-related sub-phases that should be triggered separately by at least two separate traversals of the AST:

- generation of symbol tables;
- semantic checking and type-checking semantic actions.

The semantic analyzer should use as input the AST generated by the parser. The semantic analyzer should locate, report, and recover from eventual semantic errors. The operation of the semantic analyzer should produce a symbol table data structure that is to be used by the further processing stages, including scoping and binding checks, type checks, and eventually code generation. It should also write to a file a text representation of the symbol table, as well as potential semantic error reporting in another separate error reporting file.

Answer to the following tasks:

Your task is to implement a semantic analyzer for the language described in TMA – #2, by performing the following sub-tasks:

(a) Implement the properties (i.e., the data structures and functions) for the symbol tables which can be used to support the type checking semantic actions.

(b) Write the complete list of the semantic rules used in the above implementation.

(c) Use syntax-directed translation to implement the abstract syntax tree (AST) that triggers semantic actions which occurs in the source code and **describe**:

  [i] For **each phase**, mapping of semantic actions to AST nodes, along with a description of the effect/role of each semantic action.

  [ii] The **overall** structure of the solution and the roles of the individual components used in the applied solution.

(d) Report all the existing semantic errors and warnings present in the entire program implemented in (c) and **include**:

  [i] The location in the program source file where error was located.

  [ii] The errors in synchronized order, even the errors are found in different phases.

— End —

THE OPEN UNIVERSITY OF SRI LANKA
FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
BACHELOR OF SOFTWARE ENGINEERING HONOURS

EEX6363 – Compiler Construction

Design Project Deadline on 15 November 2025 2024/2025

In this assessment, you will implement a code generation phase and finalize the compiler (which were partially implemented in each TMA), hence it is the completion of the design project. In addition, you will verify your implemented compiler by using the rules predefined in each TMA.

Answer to the following tasks:

The following sub-tasks need to be completed in order to succeed in the completion of the design project.

(a) **Describe** the:

[i] Register allocation/deallocation scheme used in the implementation.

[ii] Memory usage scheme used in the implementation (stack-based, tag-based), including for temporary variables, function calls (free functions or member functions, data members, and calculation of variables' allocated memory).

[iii] Purpose of each phase involved in the implemented code generation. For each phase, mapping of semantic actions to AST nodes, along with a description of the effect/role of each semantic action.

(b) Implement the code generation scheme that generates code for your chosen machine architecture and include the test results.

(c) Implement and test **the compiler** with your all predefined rules defined in each TMA.

Compilers are straightforward to test, since they are stateless at a high level. You can use manual testing or automatic testing.

- Manual testing – is consistency in tests, inputs and results;
  — You can use *test* files.
- Automatic testing – is important for your validating compiler;
  — Given an input, they produce an output (e.g., source -> tokens)

— Test cases can be made easily from the specifications in each TMA;
— Your tests should be easy and fast to run.

The viva dates will be scheduled after the deadline and it is a compulsory.

— End —