



CS4158 – Programing Language Technology

Project

Semester 2 AY 24/25

Name: Olan Healy

Student ID: 21318204

Table of Contents

Contents

Task 1: Lexical Analyser in C	2
Task 2: Lexical Analyser Using Lex	3
Task 3.....	4
(A) YACC-Based Arithmetic Expression Parser	4
(B) YACC For Variable Recognition.....	5
(C) YACC Arithmetic Calculator	6

(D) YACC BNF Rules To Generate Abstract Syntax Tree	7
Task 4: FIRST and FOLLOW	8
Task 5: LL(1) Parser	9
Task 6: Operator Precedence Parser	11
Task 7: Intermediate Code Generation for Expressions	13
Task 8: Loop Unrolling.....	14
Task 9.....	15

Task 1: Lexical Analyser in C

This task involved designing and implementing a basic lexical analyser in C for a simplified Java-like language. The program accepts code input line by line until a special delimiter (\$\$) is entered, after which it processes the input string to categorise tokens into five types: **keywords**, **identifiers**, **constants**, **operators**, and **symbols**. Keywords were identified using a helper function that checks for matches against a predefined list. The analyser also correctly skips over comments and redundant whitespace.

Each category was stored in its respective array, and the results were printed in a structured format showing all identified tokens.

The program was compiled using

```
gcc lexical_analyser.c -o lexical_analyser
```

```

C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 1>lexer.exe
Enter a line(s) of code (end input with '$$' on a new line):
int a,b,c;
//commnet to ignore
a=b+c;
$$

Constants:  []
Keyword:    ['int']
Identifier: ['a', 'b', 'c', 'a', 'b', 'c']
Operators:  ['=', '+']
Symbols:    [',', ',', ';', ';']

```

Figure 1 Output for Task 1

[GITHUB for TASK 1](#)

Task 2: Lexical Analyser Using Lex

In this task, a lexical analyser was implemented using the Lex tool to process Java-like input and categorise tokens such as keywords, constants, identifiers, operators, and comments. The Lex file defines a series of regular expressions that match specific lexical patterns, including:

- Reserved keywords (int, String, void, etc.)
- Constants (both integers and floating-point)
- Identifiers
- Arithmetic and special operators
- Line comments (//)
- Unknown symbols

A global variable commentCount was introduced to count and report the number of comments ignored during scanning. The scanner reads multiline input until the user enters \$\$ on a new line. The input is then passed to yy_scan_string() to initialise scanning. Each token type triggers a corresponding print statement, clearly identifying the category of each lexeme.

The program was compiled using:

```
flex analyser.1
```

```
gcc lex.yy.c -o analyser
```

```

Enter a program (end with $$ on a new line):
void main(){

    //hello world

    int a = 7,b=7.35;
}
$$
void Reserved Keyword
main Reserved Keyword
( Special Operator
) Special Operator
{ Special Operator
int Reserved Keyword
a Identifier
= Arithmetic Operator
7 Constant
, Special Operator
b Identifier
= Arithmetic Operator
7.35 Floating Number
; Special Operator
} Special Operator
1 Comments Ignored

```

Figure 2 Output for Task 2

[GITHUB for TASK 2](#)

Task 3

(A) YACC-Based Arithmetic Expression Parser

In this task, a parser for arithmetic expressions was implemented using Lex and YACC. The goal was to recognise valid arithmetic expressions involving +, -, *, /, numeric values, and parentheses.

The **Lex file** defines tokens for integers ([0-9]+), operators, and parentheses, while ignoring whitespace and tabs. Recognised numbers are converted to integers and returned as NUMBER tokens to the parser. Newlines are handled separately to trigger expression evaluation.

The **YACC file** defines grammar rules to parse arithmetic expressions using operator precedence. Binary operations (+, -, *, /) are supported with appropriate precedence using %left. Parentheses are handled recursively, and expressions are accepted once followed by a newline. Although semantic evaluation actions were not implemented in this version, the program

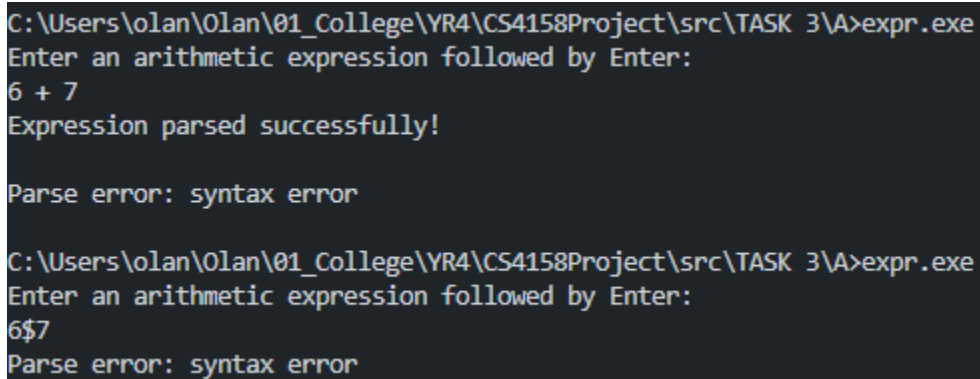
confirms if the input was syntactically correct with a message:
Expression parsed successfully!

The program was compiled using:

```
bison -d expr.y
```

```
flex expr.l
```

```
gcc expr.tab.c lex.yy.c -o expr_parser
```



```
C:\Users\olan\olan\01_College\YR4\CS4158Project\src\TASK 3\A>expr.exe
Enter an arithmetic expression followed by Enter:
6 + 7
Expression parsed successfully!

Parse error: syntax error

C:\Users\olan\olan\01_College\YR4\CS4158Project\src\TASK 3\A>expr.exe
Enter an arithmetic expression followed by Enter:
6$7
Parse error: syntax error
```

Figure 3 Task 3 (a) Output

[GITHUB for TASK 3\(A\)](#)

(B) YACC For Variable Recognition

This task involved writing a Lex and YACC program to recognise valid variable names that follow Java-like rules: they must start with a letter and can be followed by any combination of letters and digits. This ensures that identifiers like `abc`, `var123`, and `myVariable` are accepted, while invalid ones like `123abc` are rejected.

In the **Lex file**, a regular expression `[a-zA-Z][a-zA-Z0-9]*` matches valid identifiers and returns them as `IDENTIFIER` tokens, with their text stored in `yylval.s`. Whitespace is ignored, and newline characters are passed to the parser to trigger evaluation.

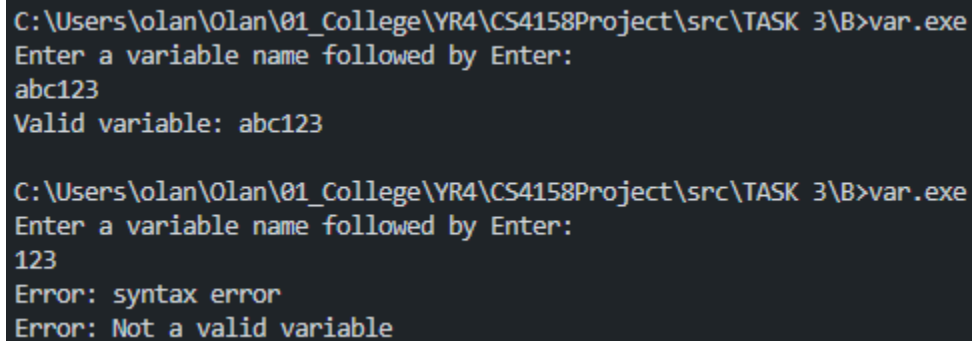
The **YACC file** defines the grammar to accept a valid identifier followed by a newline. If the token is valid, the program prints the recognised variable name. Otherwise, it catches the error and prints a custom message. The `%union` is used to associate the identifier string with the token and non-terminal variable.

The program was compiled using:

```
bison -d var.y
```

```
flex var.l
```

```
gcc var.tab.c lex.yy.c -o var_parser
```

A screenshot of a terminal window showing the execution of a program named var.exe. The terminal has a dark background with light-colored text. The first run shows a prompt 'Enter a variable name followed by Enter:', the user enters 'abc123', and the program outputs 'Valid variable: abc123'. The second run shows the same prompt, the user enters '123', and the program outputs two lines of error messages: 'Error: syntax error' and 'Error: Not a valid variable'.

```
C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 3\B>var.exe
Enter a variable name followed by Enter:
abc123
Valid variable: abc123

C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 3\B>var.exe
Enter a variable name followed by Enter:
123
Error: syntax error
Error: Not a valid variable
```

Figure 4 Task 3 (b) Output

GITHUB for TASK 3(B)

(C) YACC Arithmetic Calculator

In this task, a fully functional arithmetic calculator was developed using Lex and YACC, capable of evaluating expressions involving +, -, *, /, parentheses, and multi-digit integers.

The **Lex file** tokenises whitespace, newlines, operators, parentheses, and numeric constants. When a number is encountered, it is converted into an integer using `atoi()` and stored in `yylval.s`, to be used by the YACC parser.

The **YACC file** implements grammar rules to evaluate expressions following the correct operator precedence, defined using `%left`. The `expr` non-terminal performs the actual arithmetic during parsing by evaluating the result at each level using C-style operators (e.g., `$$ = $1 + $3`). Multiple expressions can be entered sequentially, with each result printed immediately.

The program was compiled using:

```
bison -d calc.y
```

```
flex calc.l
```

```
gcc calc.tab.c lex.yy.c -o calculator
```

```

C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 3\C>calculator.exe
Enter expression:
5*(4+1)
Result = 25

C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 3\C>calculator.exe
Enter expression:
5$4
Error: syntax error

C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 3\C>

```

Figure 5 Task 3 (C) Output

GITHUB for TASK 3(C)

(D) YACC BNF Rules To Generate Abstract Syntax Tree

This task extended expression parsing by generating an Abstract Syntax Tree (AST) from a mini-program written in a C-style language and then producing three-address code using the AST structure.

The implementation was split across multiple files:

- **ast.h / ast.c** defines the `ast_node` structure and helper functions to create and manage AST nodes (e.g., operators, identifiers, assignments).
- **YACC (parser.y)** defines grammar rules for declarations, assignment statements, and arithmetic expressions. Each rule returns a subtree of the AST. The complete program is represented as a program node with declaration and statement subtrees.
- **Lex (scanner.l)** tokenises keywords, identifiers, numbers, and operators. Tokens are mapped to `yylval.s` for AST construction.

Once parsing is complete and the AST is built, the code generator traverses the AST to print three-address instructions. Each arithmetic operation or assignment generates one instruction.

The program was compiled using:

```
bison -d parser.y -o parser.tab.c
```

```
flex -o scanner.c scanner.l
```

```
gcc -c ast.c
```

```
gcc parser.tab.c scanner.c main.c ast.o -o ast_parser
```

```

C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 3\D>parser.exe < test.txt
-----
Pos Operator Arg1 Arg2 Result
-----
0 * b c t0
1 + a t0 t1
2 = t1 c
3 / y c t2
4 = t2 x
-----

```

Figure 6 Task 3 (D) Output

[GITHUB for TASK 3\(D\)](#)

Task 4: FIRST and FOLLOW

This task involved implementing a program in C to simulate the computation of FIRST and FOLLOW sets for a context-free grammar, a fundamental step in parser construction.

The program accepts grammar productions from standard input in the format $A \Rightarrow \dots$, with alternatives separated by $|$. Each production rule is parsed and stored internally, with both terminals and nonterminals automatically detected and listed.

The program then recursively computes:

- **FIRST sets**, which determine the set of terminals that can begin strings derived from a nonterminal.
- **FOLLOW sets**, which determine which terminals can appear immediately to the right of a nonterminal in sentential forms.

Recursive helper functions (computeFirst and computeFollow) were designed to process dependencies across rules, including handling nullable sequences (epsilon). The implementation supports basic epsilon propagation and conflict resolution, and avoids duplicate entries using setInsert.

The program was compiled using:

```
gcc first_follow.c -o first_follow
```



```

C:\Users\olan\olan\01_College\YR4\CS4158Project\src\TASK 4>first_follow.exe
Enter grammar productions (one per line). End input with an empty line:
E=TR
R=+TR|#
T=FY
Y=*FY|#
F=(E)|i

Transitions Read: (8)
E=TR
R=+TR
R=#
T=FY
Y=*FY
Y=#
F=(E)
F=i

Non-Terminals Encountered: (5)
E R T Y F
Terminals Encountered: (5)
+ * ( ) i
First(E) = { (, i }
First(R) = { +, # }
First(T) = { (, i }
First(Y) = { #, * }
First(F) = { (, i }
Follow(E) = { ), $ }
Follow(R) = { ), $ }
Follow(T) = { +, ), $ }
Follow(Y) = { +, ), $ }
Follow(F) = { +, *, ), $ }

```

Figure 7 Output For Task 4

[GITHUB for TASK 4](#)

Task 5: LL(1) Parser

This task focused on building an LL(1) predictive parser capable of analysing context-free grammars. The implementation allows the user to input a set of grammar rules, after which the program computes the FIRST and FOLLOW sets for all non-terminals. These sets are then used to construct the predictive parsing table, which forms the basis of the LL(1) parsing algorithm.

Each production rule is parsed and tokenised, supporting alternatives via the pipe symbol (`|`). Once the table is built, the user can input a string to be parsed. The parser uses a stack-based

simulation to process the string according to table entries, printing each step and action taken (Shift, Matched, or Error), as well as the final outcome.

The program highlights whether the grammar is suitable for LL(1) parsing by detecting any table conflicts. This makes it a useful educational tool for understanding how predictive parsers work and how grammars influence parsing behaviour.

The program was compiled using:

```
gcc ll(1)_parser.c -o ll(1)_parser
```

```

C:\Users\olan\olan\01_College\YR4\CS4158Project\src\TASK 5>"ll(1)_parser.exe"
Enter number of production rules: 2
Enter production 1: S->aA
Enter production 2: A->a|b

Grammar
Starting symbol is: S
Non-Terminal symbols: S A
Terminal symbols: a b $
Production rules:
S->aA
A->a
A->b

FIRST( S ): { a }
FIRST( A ): { a, b }

FOLLOW( S ): { $ }
FOLLOW( A ): { $ }

Predictive Parsing Table:

      a      b      $
-----
S      |      S->aA      _      _
A      |      A->a      A->b      _

Enter string for parsing: ab

Parsing sequence and actions
STACK      INPUT      ACTION
-----
$S          ab$      Shift: S->aA
$Aa         ab$      Matched: a
$A          b$      Shift: A->b
$b          b$      Matched: b
$           $
Parsed successfully.

```

Figure 8 Output for Task 5

[GITHUB for TASK 5](#)

Task 6: Operator Precedence Parser

This task involved implementing an Operator Precedence Parser in C that can evaluate expressions based on a user-defined precedence table between terminal symbols. The parser uses shift-reduce logic to determine whether a given input string can be derived from the grammar.

The user first inputs:

- The number of terminal symbols (e.g. +, *, i, \$)
- The terminal symbols themselves
- The operator precedence relations between each pair (<, >, =, or custom)

These are stored in a matrix (opTable) indexed by the position of symbols in the terminal array. The input string is then parsed using a stack. At each step, the program:

- **Shifts** if the relation is < or =
- **Reduces** if the relation is >
- Accepts the string if both stack and input hold \$ or if an A relation is encountered

The stack and input pointer are printed at every stage, clearly showing parser actions.

This parser can detect conflicts or invalid configurations when an unrecognised precedence is found, ensuring clarity in grammar design.

The program was compiled using:

```
gcc op_parser.c -o op_parser
```

```
C:\Users\olan\olan\01_College\YR4\CS4158Project\src\TASK 6>op_precedence_parser.exe
Enter the number of terminals: 4
Enter the terminal symbols (e.g. + * i $): +*i$

Enter the operator precedence table values:
Enter relation for + vs +: >
Enter relation for + vs *: <
Enter relation for + vs i: <
Enter relation for + vs $: >
Enter relation for * vs +: >
Enter relation for * vs *: >
Enter relation for * vs i: <
Enter relation for * vs $: >
Enter relation for i vs +: >
Enter relation for i vs *: >
Enter relation for i vs i: =
Enter relation for i vs $: >
Enter relation for $ vs +: <
Enter relation for $ vs *: <
Enter relation for $ vs i: <
Enter relation for $ vs $: A
```

Enter the input string: i+i*i\$

STACK	INPUT STRING	ACTION
\$	i+i*i\$	Shift i
\$i	+i*i\$	Reduce
\$	+i*i\$	Shift +
\$+	i*i\$	Shift i
\$+i	*i\$	Reduce
\$+	*i\$	Shift *
\$+*	i\$	Shift i
\$+*i	\$	Reduce
\$+*	\$	Reduce
\$+	\$	Reduce
\$	\$	String Accepted

Figure 9 Output For Task 6

[GITHUB for TASK 6](#)

Task 7: Intermediate Code Generation for Expressions

In this task, a C program was developed to generate three-address intermediate code from assignment statements involving arithmetic expressions. The approach uses infix-to-postfix conversion followed by stack-based code generation.

The program expects an input line in the form $x = a + b * c$ where the left-hand side (x) is the target variable, and the right-hand side is an infix expression. It performs the following steps:

1. **Infix to Postfix Conversion:** Using operator precedence (* and / have higher precedence than + and -), the expression is converted into postfix notation using a stack-based algorithm.
2. **Three-Address Code Generation:** The postfix expression is evaluated using another stack. For each operator, the top two operands are popped, and a new temporary variable (T0, T1, ...) is created to store the result. A corresponding three-address instruction is printed (e.g., $T0 = b * c$, followed by $x = a + T0$).

Each operation is clearly separated, enabling easy integration with later compiler phases such as optimisation or code emission.

The program was compiled using:

```
gcc intermediate_codegen.c -o intermediate_codegen
```

```
C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 7>intermediate_code_generation.exe
Enter an expression:
x=a+b*c-d/e+f
Intermediate code:
T0 = b * c
T1 = a + T0
T2 = d / e
T3 = T1 - T2
T4 = T3 + f
x = T4
```

Figure 10 Output For Task 7

[GITHUB for TASK 7](#)

Task 8: Loop Unrolling

This task involved implementing and comparing a standard loop and a loop unrolling technique to optimise the summation of elements in a large array. Loop unrolling is a low-level compiler optimisation technique aimed at reducing loop control overhead and increasing performance by manually replicating the loop body multiple times.

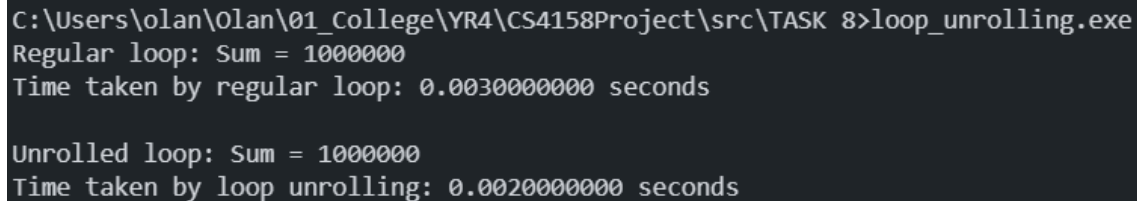
A static array of one million integers was initialised with all values set to 1. Two summation functions were defined:

- **sum_regular:** Performs a typical element-by-element summation using a standard for loop.
- **sum_unrolled:** Implements loop unrolling by processing four elements per iteration, reducing the total number of iterations and loop control instructions. Any leftover elements are processed separately.

The execution time for each method was measured using `clock()` from the C standard library. Output showed the total sum and the time taken by each approach, allowing for a performance comparison.

Program was compiled using:

```
gcc loop_unrolling.c -o loop_unrolling
```



```
C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 8>loop_unrolling.exe
Regular loop: Sum = 1000000
Time taken by regular loop: 0.0030000000 seconds

Unrolled loop: Sum = 1000000
Time taken by loop unrolling: 0.0020000000 seconds
```

Figure 11 Output For Task 8

[GITHUB for TASK 8](#)

Task 9

In this task, a C program was developed to perform constant propagation, a key optimisation technique used in compiler design. The goal is to replace variables known to have constant values with their actual values in subsequent statements, improving efficiency and enabling further optimisations like constant folding.

The program accepts a series of three-address code statements formatted as:

<operator> <operand1> <operand2> <destination>

Statements are read from standard input until an empty line is encountered.

It maintains a constant pool (an array of variable-value pairs) for assignments like `= 5 x`, which means `x` is assigned the constant 5. For any subsequent operations involving `x`, the program checks if its value is in the constant pool and substitutes it accordingly. Only non-assignment operations are printed in the optimised output, with constant values propagated into their operand positions.

This approach demonstrates how compile-time knowledge of constant values can simplify the code and reduce unnecessary computations at runtime.

The program was compiled using:

```
gcc constant_propagation.c -o constant_propagation
```

```
C:\Users\olan\Olan\01_College\YR4\CS4158Project\src\TASK 9>constant_propagation.exe
Enter 3-address code statements (end input with an empty line):
= 3 - a
+ a b t1
+ a c t2
+ t1 t2 t3

Optimised Code (after Constant Propagation):
+ 3 b t1
+ 3 c t2
+ t1 t2 t3
```

Figure 12 Output For Task 9

[GITHUB for TASK 9](#)

References

- *Writing Lex programs (Scanner using Lex/Flex)* – [Flex \(Fast Lexical Analyzer Generator\) | GeeksforGeeks](#)
Helped implement the Lex scanner for Task 2 to classify tokens like operators, identifiers, and numbers.
- *Lex and YACC: Variable name validation* – [GeeksforGeeks](#)
Used for Task 3B to check and validate identifiers using regular expressions in Lex and semantic checks in YACC.
- *Arithmetic Calculator using Lex and YACC* – [How to Compile & Run LEX and YACC Programs on Windows 8 10 and 11 by Dr. Mahesh Huddar](#)
Aided in constructing the fully evaluated calculator in Task 3C with semantic rules and operator precedence.
- *Generating Abstract Syntax Trees and Three-Address Code* – [Unit-3.pdf](#)
Guided the implementation of AST creation and three-address code generation for Task 3D.
- *FIRST and FOLLOW Sets Explained* – [FIRST\(\) and FOLLOW\(\) Functions](#), [First and Follow Sets Explained](#)
Clarified the logic for computing FIRST and FOLLOW sets in Task 4.
- *LL(1) Parsing Table Construction and Simulation* – [GeeksforGeeks](#)
Used to guide predictive parsing table construction and stack-based parsing implementation in Task 5.

- *Operator Precedence Parser in C* – [Operator Precedence Parsing Program in C - C Program to Implement Operator Precedence Parsing](#)
Helped implement the operator precedence logic and table simulation in Task 6.
- *Infix to Postfix and Three-Address Code Generation* – [GeeksforGeeks](#) Supported expression transformation and code generation strategy in Task 7.
- *Constant Propagation in Compiler Optimisation* – [GeeksforGeeks](#)
Provided the theory and implementation approach for Task 9's constant propagation pass.
- *Lecture notes CS4158: [Module Overview - CS4158 - PROGRAMMING LANGUAGE TECHNOLOGY 2024/5 SEM2](#)*
- *Notes [Reference Material - OneDrive](#)*
- *Youtube Tutorial [\(94\) Tutorial: programming with lex/yacc - YouTube](#)*
- *GitHub: [olanhealy/CS4158-Project](#)*