SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

COMPILER DESIGN LABORATORY

Course Code: CSE321

Semester: V

Lab Manual

2024

SHANMUGHA ARTS, SCIENCE, TECHNOLOGY AND RESEARCH ACADEMY
(SASTRA Deemed to be) University
Tirumalaisamudram, Thanjavur-613 401
School of Computing

Course Objective:

The Learners will be able to design and implement the following phases of compiler like scanning and parsing, ad-hoc syntax directed translation, code generation and code optimization for any formal language using LEX and YACC tools.

Course Learning Outcomes:

1. Demonstrate the scanner construction from using Lex
2. Develop parser using Lex & YACC
3. Apply context sensitive analysis for type Inferencing
4. Construct intermediate Code representation for a given source code
5. Identify appropriate techniques for code optimization
6. Explain about the code generation and register allocation components in the backend phase of a compiler

List of Experiments:

1. Develop a scanner using LEX for recognizing the tokens in a given C program.
2. Develop a program to find the FIRST and FOLLOW sets for a given Context Free Grammar
3. Extend the outcome of experiment 2 to implement a LL(1) parser in C or Java to decide whether the input string is valid or not
4. Implement a LR(1) bottom up parser in C or Java to decide whether the input string is valid or not (Context- Free Grammar, Action and GOTO tables are supplied as inputs)
5. Develop a parser for all branching statements of 'C' programming language using LEX& YACC
6. Develop a parser for all looping statements of 'C' programming language using LEX & YACC
7. Develop a parser for complex statements in 'C' programming language with procedure calls and array references using LEX & YACC
8. Use LEX and YACC to create two translators that would translate the given input (compound expression used in experiment 9) into three-address and postfix intermediate codes. The input and output of the translators should be a file
9. Write an optimizer pass in C or Java that does common-sub expression elimination on the three address intermediate code generated in the previous exercise.
10. Implement Local List Scheduling Algorithm.
11. Implement Register Allocation
12. Use LEX & YACC to write a back end that traverses the three address intermediate code and generates x86 code.

Exercise No. 1 IMPLEMENTATION OF LEXICAL ANALYZER (SCANNER) USING LEX

Develop a lexical analyser (scanner) using LEX for recognizing the various token types in a given input C program.

Objective:

Learner will be able to design a lexical analyser for recognizing any type of tokens of a programming language.

Prerequisite:

Structure of a LEX program, Lex Specification for Tokens.

Pre-lab Exercises:

Lex program to recognize a number and string.

Lex program to count the number of words, lines and characters in the given input.

Procedure:

- Define the rules for tokenizing the Input.
- Construct the regular expressions for each token.
- Translate the regular expressions in accordance with the LEX specification.
- Create a LEX source file with a "`.l`" extension. The LEX source file will contain the definitions, rules and auxiliary code sections.
- Compile the LEX source file. This will generate the C source file named "lex.yy.c".
- Then compile the generated C source file, the scanner.
- Finally, run the compiled scanner with an input file containing the code to be tokenized.

Sample Lex program to recognize an Integer " num.l"

```
%{
 #include <stdio.h>
 %}

%%
[0-9]+    { printf("Integer: %s\n", yytext); }
.         { printf("Unrecognized character: %s\n", yytext); }
%%

int main( )
{ yylex( ); return 0; }

int yywrap( ) { return 1; }
```

Sample Output

Compilation

```
$ flex num.l
$ gcc lex.yy.c
$ ./a.out
```

Sample Input:

45

Output:

Integer: 45

Sample Input / Output

Input	Output	
#include<stdio.h>	#include<stdio.h>	is a preprocessor directive
int main()	int	is a keyword
{	main()	is a function
printf("Hello World");	{	block begins
}	printf(	is a function
	"Hello World"	is a string
	)	symbol
	}	end of the block.

Additional Exercises:

1. Construct a lexical analyzer for Java Constructs
2. Construct a lexical analyzer for Python Constructs