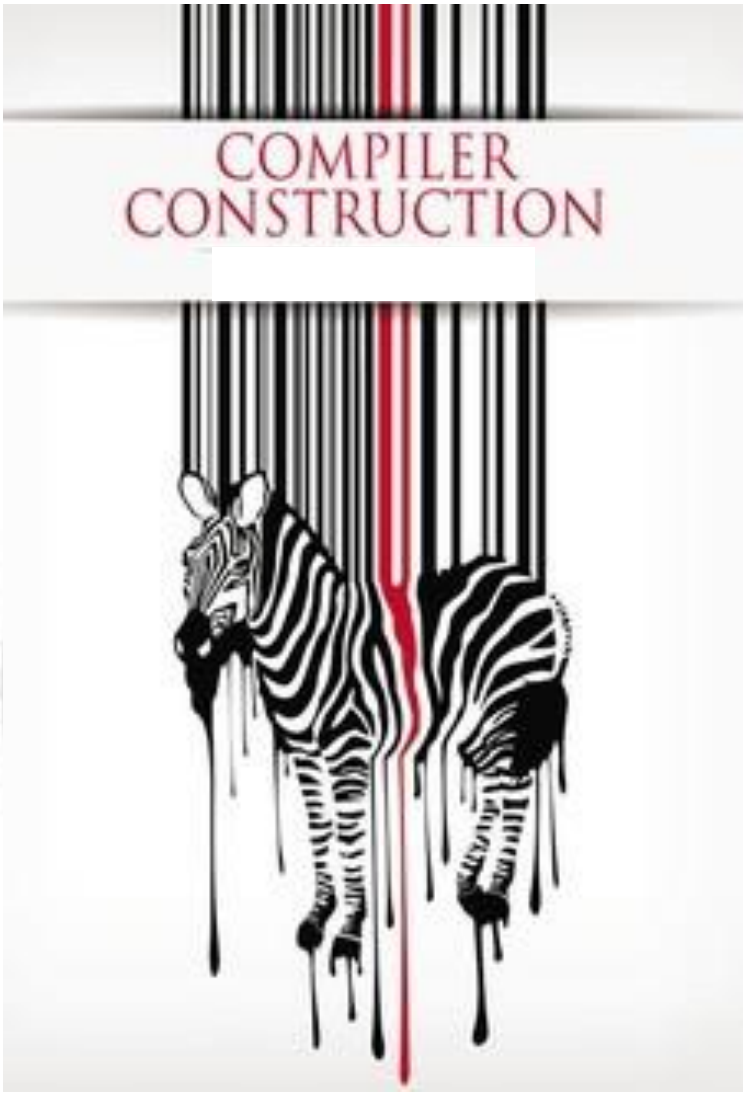
CS424L Compiler Construction Lab

Faculty of Computer Science and Engineering

Lab manual

COMPILER CONSTRUCTION



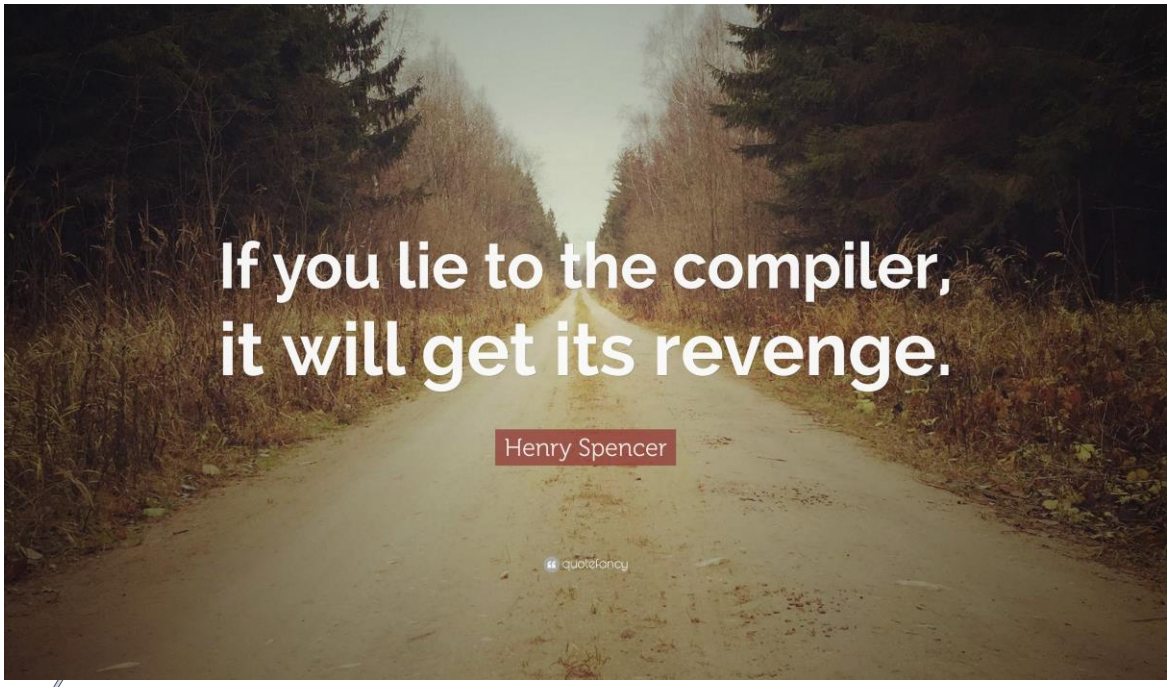
Faculty of Computer Science & Engineering (FCSE)

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology (GIKI)

PREPARED BY DR. HASHIM & MR. BADRE
PREPARED BY DR. HASHIM ALI & ENGR. BADRE MUNEER

CS424-L Compiler Construction Lab

Lab Outline



Dr. Hashim Ali, & Engr. Badre Munir
FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)
GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES & TECHNOLOGY (GIKI)

CS424L Compiler Construction Lab (1 CH)	CE

Pre-Requisite: CS424

Instructor: **FasihaImtiaz**
Office # G-36 FCSE, GIK Institute,
Email: *gcs2108@giki.edu.pk*
Office Hours: 11:00am ~ 12:00 pm

Lab Introduction

This course provides practical experience with compiler construction and instructs in the use of common tools for lexical analysis, parsing and translation.

Lab Contents

1. **Lexical Analysis (2 weeks)**
   Regular Expressions, NFA's, DFA's, Lexer Construction
2. **Syntax Analysis (4 weeks)**
   Top-down and Bottom-up Parsing, Grammar Classes: LL, LR, LALR.
3. **Syntax Directed Translation (3 weeks)**
   Syntax Directed Definitions and Translation Schemes, Semantic Analysis
4. **Intermediate Code Generation (3 weeks)**

Mapping of CLOs and PLOs

Sr. No	Course Learning Outcomes[+]	PLOs*	Taxonomy level (Psychomotor Domain)
CLO1	Build lexical analyzers using regular expressions	PLO 1	P3
CLO 2	Build parsers using top-down and bottom-up methods.	PLO 3	P3
CLO 3	Build translators for basic programming languages.	PLO 3	P4
	[+]Please add the prefix "Upon successful completion of this course, the student will be able to" [*]PLOs are for BS (CE) only		

CLO Assessment Mechanism

Assessment tools	CLO_1	CLO_2	CLO_3
Lab Tasks	X	X	X
Midterm Exam			
Final Exam			

Overall Grading Policy

Assessment Items	Percentage
Lab Evaluation	45%
Midterm Exam	20%
Final Exam	35%

Text and Reference Books

- Lab Manual
- Alfred Aho, Monica Lam, Ravi Sethi & Jeffrey Ullman; Compilers: Principles, Techniques, and Tools. 2nd Edition.
- Keith D. Cooper & Linda Torczon; Engineering a Compiler. 2nd Edition.

Administrative Instruction

- According to institute policy, 100% attendance is *mandatory* to appear in the final examination.
- Assignments must be submitted as per instructions mentioned in the assignments.
- For queries, kindly follow the office hours in order to avoid any inconvenience.

Computer Usage/Software tools

- Flex, Bison

Lecture Breakdown

1	Flex	7	Bison
2	Flex II	8	Bison with Ambiguous Grammars
3	Predictive Parsing	9	Translating Expressions
4	Symbol Tables	10	Logical And Relational Expressions
5	Syntax Trees and Intermediate Code	11	Type Checking and Type Coercion
6	Intermediate Code for Expressions	12	Open-ended lab

"Trying to outsmart a compiler defeats much of the purpose of using one"

Brian Kernighan

CS424-L Compilers Construction Lab

Lab Benchmark Report

Dr. Hashim Ali & Engr. Badre Munir
FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)
GHULAM ISHAQ KHAN INSTITUTEO F ENGINEERING SCIENCES AND TECHNOLOGY (GIKI)

The following were consulted to benchmark CS424-L:

Higher Education Commission (HEC) curriculum for Computer Science (Revised 2017)

In the curriculum defined by HEC, Compiler Construction is a Computer Science core course. However, there is no information about separate lab courses in the curriculum.
At FCSE, the contents of CS424L cover the entire hands-on aspects of the course contents as laid out in the curriculum manual (summarized in the table below).

Association of Computing Machinery (ACM) curriculum for CS (2013)

As per ACM Curriculum for CS (2013), Compiler Construction is categorized under the knowledge area of Programming Languages, as an elective course. There is no separate information for lab courses.
At FCSE, the contents of CS424L caters to the bulk of the body of knowledge coverage as laid out in the ACM curriculum (summarized in the table below).

Curriculum Benchmarking for CS424-L Compiler Construction Lab				
Topic#	**Topic**	**FCSE, GIKI**	**HEC Curriculum for CS (Revised 2017)**	**ACM curriculum for CS (2013)**
1	Introduction to interpreter and compiler	✓	✓	✗
2	Compiler techniques and methodology	✗	✓	✗
3	Organization of compilers	✗	✓	✗
4	Lexical and syntax analysis	✓	✓	✓
5	Parsing techniques	✓	✓	✓
6	Types of parsers	✓	✓	✓
7	Top-down parsing	✓	✓	✓
8	Bottom-up parsing	✓	✓	✓
9	Type checking	✓	✓	✓
10	Semantic analyzer	✓	✓	✓
11	Object code generation and optimization	✓	✓	✓
12	Detection and recovery from errors	✓	✓	✗
13	Regular and context free languages including DFAs and NFAs	✓	✗	✓
14	Meggy Jr Simple runtime library	✗	✗	✓
15	AVR assembly code including the stack and heap memory model	✗	✗	✓
16	Visitor Design Pattern	✗	✗	✓
17	Data-flow analysis usage in register allocation	✗	✗	✓
18	Iterative compiler design and development	✗	✗	✓
19	Test-driven development and regression testing	✗	✗	✓
20	Revision control and pair programming	✗	✗	✓

CS424-L Compiler Construction Lab

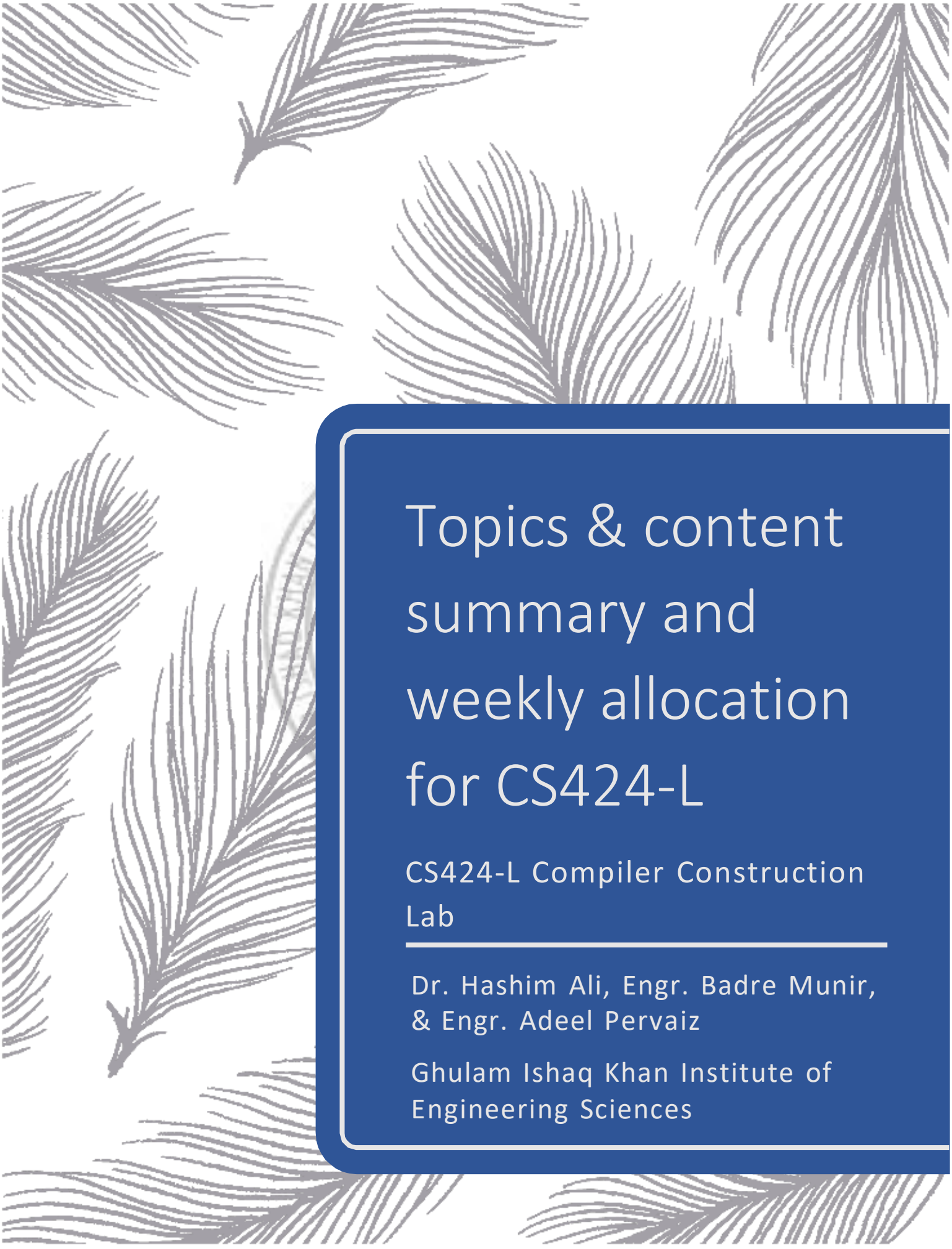
LAB EVALUATION RUBRICS

Dr. Hashim Ali, & Engr. Badre Munir
FACULTY OF COMPUTER SCIENCE AND ENGINEERING (FCSE)
GHULAM ISHAQ KHAN INSTITUTE OF ENGINEERING SCIENCES & TECHNOLOGY (GIKI)

Faculty of Computer Science & Engineering

CS424L – Compiler Construction Lab

Rubrics for Assessment of CS424-L Compiler Construction Lab				
Criteria	**Total Marks**	**Excellent (2)**	**Average (1)**	**Poor (0)**
Accuracy	2	The solution does exactly what has been asked in the task including for tricky cases and gives the accurate/expected results for all parts.	The solution does exactly as asked in the task but does not handle exceptions/tricky input.	The solution is poor, does not perform at all according to the task or gives absolutely wrong results even for standard input.
Clarity	2	The solution is clear, and the student is able to formulate and explain the solution coherently.	The solution is partially clear and the student can explain parts of the program	The student has not formulated the solution correctly or can not explain his/her solution at all.
Code	2	The code is coherently written and documented to the point that the reader can broadly understand the working of each major part.	The code is somewhat coherent, tabbing has not been used but has useful chunks written with broad documentation.	The code lacks readability, can not be explained without the coder and has little/no documentation.
Performance	2	The code is extremely fast and for standard medium sized input performs its processing quickly.	The code is somewhat fast, runs quickly for small inputs but may take long time for medium sized/large inputs.	The code is extremely slow and gets stuck even for small-sized inputs.
Completion Time	2	The final complete solution for all tasks was submitted before half of the class	The final complete solution for all tasks was submitted later than half of the class but before 80% of the size of class.	The final complete solution was submitted along with the last 20% of the class
Total	10	10	5	0



Topics & content summary and weekly allocation for CS424-L

CS424-L Compiler Construction Lab

Dr. Hashim Ali, Engr. Badre Munir, & Engr. Adeel Pervaiz

Ghulam Ishaq Khan Institute of Engineering Sciences

Faculty of Computer Science & Engineering

CS424L – Compiler Construction Lab

Week #	Topic	Contents	Page #
1	No lab	-	-
2	Interpreters and compilers – Lexical analyzer generator	Introducing Flex – the lexical analyzer generator.	01
3	Lexical and syntax analysis in compilers	More exercises and advanced topics on Flex.	08
4	Parsing in compilers	Recursive descent predictive parsing – combining lexical analyzer with a predictive parser.	11
5	Lexical analyzers and symbol tables in compilers	Support for declarations and nested scopes to lexical analyzers – Using symbol tables.	15
6	Lexical analyzers, syntax trees and object code generation in compilers	Extending a predictive parser to generate syntax trees and then traversing it to generate intermediate code.	23
7	Parser generator Bison	Introducing Bison – the parser generator.	35
8	Mid-term exams	-	-
9	Ambiguous and unambiguous grammar	Introduce ambiguous grammars, using Bison with ambiguous grammars and conversion from ambiguous to unambiguous grammar.	42
10	Types of parsers I - bottom-up parsing in compilers	Using a bottom-up parser to generate a syntax tree and traverse it to generate intermediate code.	46
11	Types of parsers II - top-down parsing in compilers	Using a top-down parser to generate a syntax tree and traverse it to generate intermediate code.	51
12	Handling logical expressions in compilers	Translating logical expressions using jumping code	65
13	Error detection and handling in compilers	Adding basic type checking and type coercion to compiler code.	71
14	Open ended lab	-	-
15	Lab finals	-	-

CS424-L Compiler Construction Lab

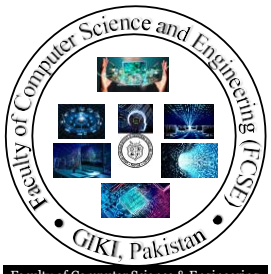
LAB I – LEXICAL ANALYSIS USING FLEX

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology



Faculty of Computer Science & Engineering (FCSE)


CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 1 — Lexical analyzer generation — Flex

Objective

The objective of this session is to learn what a lexical analyzer is, how it works and how to generate a lexical analyzer by using, for example, Flex.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Explain what a lexical analyzer generator is.
2. Use Flex to implement a lexical analyzer generator with specification of token classes.

Instructions

- Read through the handout sitting in front of a computer that has a Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Introduction to Flex

Flex is a lexical analyzer generator. It takes as input a specification of token classes in the form of regular expressions and some auxiliary code and outputs C source code that can be used as a lexical analyzer. There are other variants of Flex that generate lexical analyzers in other programming

languages such as Java. However, C++ can be mixed with Flex. The Flex manual is available at: http://www.delorie.com/gnu/docs/flex/flex.html. You should consult the manual whenever necessary. The input to flex is specified in a file with a '.l' extension. The format of the file is as follows.

```
definitions
%%
rules
%%
functions
```

Here's an example.

```
%option noyywrap
%%
.       ECHO;
\n      ECHO;
%%

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

Type this in a file example.l and run the command flex example.l on the command prompt. This will output a file called lex.yy.c which contains the generated C code for your lexical analyzer. Compile the code by running gcc lex.yy.c. The generated program is the lexical analyzer. Run the generated program.

By default the lexical analyzer reads from the standard input. For our program whatever you type in the standard input will be echoed back to screen. We can have the lexical analyzer read from a file. See the examples below for this.

The second section in the lex file is the most important. It specifies the kinds of patterns the lexical analyzer will match. This section is a list of rules. Each rule is a regular expression followed by some C code. The C code is executed whenever the corresponding regular expression matches some string.

There are two rules in the file above. The first rule has a regular expression '.' that matches any character except new line. ECHO is a C macro that outputs (prints to screen) whatever was matched. The second rule matches new lines and outputs the newline. Together these two rules simply print the input to the screen.

Common regular expressions are listed in the table below. Consult the flex manual (http://www.delorie.com/gnu/docs/flex/flex_7.html) for a full list.

Expression	Matches
.	any character except new line
ú	preceding expression 0 or more times
+	preceding expression 1 or more times
?	preceding expression 0 or 1
a|*b*	match either a or b
^	start of line
$	end of line
[]	character class

Some examples of these expressions.

Example	Matches
...	any three consecutive characters except new line
a.a	three characters beginning and ending with *a*.
aú	any number of a's
abc+	abc, abcc, abccc, ...
a(*bc*)*ú*	a, abc, abcbc, abcbcbc, ...
a(*bc*)?	a, abc
[*abc*]	a, b, c
^abc	abc at start of a line
[a-z]	a, b, c, ..., z
a|*b*	a or b

Following each regular expression is some C code that is run whenever the lexer matches a string by a regular expression. In the example above `ECHO` is a C macro that simply prints the matched string. In the last section you can write any C code you want including the main function.

Flex uses some names for its working. Some of these are listed in the table.

Name	Usage
`yylex()`	A function that runs the lexical analyzer. This keeps running until one of the rules specified executes the return statement
`char* yytext`	The matched lexeme is stored in yytext
`yyleng`	The length of the lexeme
`yylval`	Value associated with token
`FILE* yyin`	Specifies input file. Defaults to standard input
`ECHO`	Output matched string

2 Examples

2.1 Example 1

Count number of characters in on standard input. Here a variable `chars` has been defined. In the first section anything written inside `%{` and `%}` is copied to the output source file `lex.yy.c`. In this example the declaration is copied to the output file.

```
%option noyywrap
%{
int chars=0;
%}

%%
.    {chars++;}
\n   {chars++;}
%%

int main(){
    yylex();
    printf("%d \n", chars);
}
```

2.2 Example 2

Same example as before but read from input file. File name is given on the command line.

```
%option noyywrap
%{
#include<stdio.h>
int chars=0;
%}

%%
.    {chars++;}
\n   {chars++;}
%%

int main(int argc, char* argv[]){
    if(argc==2)
        yyin = fopen(argv[1], "r");
    yylex();
    printf("%d \n", chars);
}
```

2.3 Example 3

Find all words that start with A. As given in the table above, `yytext` is a pointer to the matched lexeme and `yyleng` is its length.

```
%option noyywrap
%{
#include <stdio.h>
int found=0;
%}

%%
A[a-zA-Z]* {printf("found %s of length %d\n", yytext, yyleng); found
    ++;}
.    {}
\n   {}
%%
int main(int argc, char* argv[]){
     if(argc==2)
          yyin = fopen(argv[1], "r");
     yylex();
     printf("%d \n", found);
}
```

2.4 Example 4

You can also use some C++. Compile using `g++ lex.yy.c`.

```
%option noyywrap
%{
#include <iostream >
using namespace std;
int found=0;
%}
%%
A[a-zA-Z]* {cout<<"found: "<<yytext<<" length "<< yyleng<<'\n';found
    ++;}
.    {}
\n   {}
%%
int main(int argc, char* argv[]){
     if(argc==2)
          yyin = fopen(argv[1], "r");
     yylex();
     cout<<found<<'\n';
}
```

2.5 Example 5

When you call `yylex()` it runs (checking the rules) until something is returned. We had no return statements in the rules in the examples above. When we want to tokenize the input, we can match a string and return the token. In the example we call the function in a loop. The loop stops when the function returns `EOF`. This is implicit. We define the tokens as constants in the definitions section.

```
%option noyywrap
%{
#include<stdio.h>
#define LT 1
#define GT 2
#define OTHER 3
int found=0;
%}

%%
[ \t\n]+     /*ignore whitespace */
\<  {return LT;}
\>  {return GT;}
.  {return OTHER;}
%%

int main(int argc, char* argv[]){
    if(argc==2)
        yyin = fopen(argv[1], "r");
    int ret;
    while(ret=yylex()){
        switch(ret){
            case LT:
                printf("<LT>");
                break;
            case GT:
                printf("<GT>");
                break;
            case OTHER:
                printf("<OTHER>");
                break;
        }
    }
    return 0;
}
```

3 Exercises

1. Write a program that takes as input a C or Java source file and finds all alphanumeric names or identifiers (such as keywords, types, variables, function and class names etc) along with the line numbers where they were first found. You are not required to distinguish between the various types of labels and identifiers at this point.
2. Write a program that finds all numerical constants used in a given source file. Examples are 123, 123. 10, −1234.
3. Write a program that outputs a count of all characters, words and lines in a given text file.
4. Write a program that takes as input a C source file and generates a stream of tokens. For example `if (a<b)` should be converted into <IF><LPAREN><ID><LT><ID><RPAREN>. Handle as many tokens as possible and ignore whitespace and comments that begin with //.

CS424-L Compiler Construction Lab

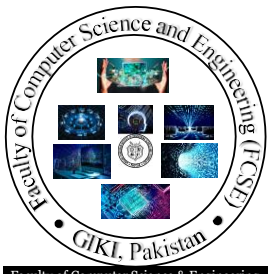
LAB II – LEXICAL ANALYSIS USING FLEX II

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology



Faculty of Computer Science & Engineering (FCSE)

CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 2 — Lexical and Syntax Analysis in Compilers — Advanced Topics

Objective

The objective of this session is to learn what a lexical analyzer is, how it works and how to generate a lexical analyzer by using, for example, Flex.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Explain what a lexical analyzer generator is.
2. Use Flex to implement a lexical analyzer generator with specification of token classes.

Instructions

- Read through the handout sitting in front of a computer that has a Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Flex - Advanced Topics

Look at the example and do the exercises below. Consult the handout for the last lab for details about flex.

When you call `yylex()` it runs until something is returned. We had no return statements in the examples in the last lab. When we want to tokenize the input, we can match a string and return

the token. In the example we call the function in a loop. The loop stops when the function returns EOF. This is implicit.

```
%option noyywrap
%{
#include<stdio.h>
#define LT 1
#define GT 2
#define OTHER 3
int found=0;
%}

%%
[ \t\n]+     /*ignore whitespace */
\<  {return LT;}
\>  {return GT;}
.  {return OTHER;}
%%

int main(int argc, char* argv[]){
    if(argc==2)
        yyin = fopen(argv[1], "r");
    int ret;
    while(ret=yylex()){
        switch(ret){
            case LT:
                printf("<LT>");
                break;
            case GT:
                printf("<GT>");
                break;
            case OTHER:
                printf("<OTHER>");
                break;
        }
    }
    return 0;
}
```

2 Exercises

1. Write a program that finds all numerical constants used in a given source file. Examples are 123, 123. 10, −1234.
2. Write a program that takes as input a C or Java source file and finds all identifiers including variables, function and class names etc., along with the line numbers where they were first found. Remember to remove comments and whitespace.
3. Write a program that takes as input a C source file and generates a stream of tokens. For example `if (a<b)` should be converted into <IF><LPAREN><ID><LT><ID><RPAREN>. Handle as many tokens as possible and ignore whitespace and comments that begin with //.

CS424-L Compiler Construction Lab

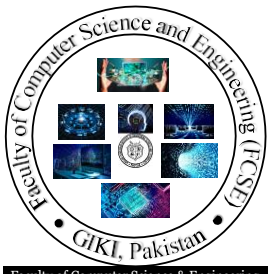
LAB III – PREDICTIVE PARSING

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology



Faculty of Computer Science & Engineering (FCSE)


CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 3 — Predictive Parsing

Objective

The objective of this session is to learn what a lexical analyzer is, how it works and how to generate a lexical analyzer by using, for example, Flex.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Explain what a lexical analyzer generator is.
2. Use Flex to implement a lexical analyzer generator with specification of token classes.

Instructions

- Read through the handout sitting in front of a computer that has a Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Recursive Descent Predictive Parsing

In this lab we will look at recursive descent predictive parsing. Recall that predictive parsing is a type of top-down parsing where the lookahead uniquely determines the production to use. Recursive descent parsers implement the parser as a set of recursive procedures.

Consider the following example from the course textbook.

stmt æ **expr ;**
| **if (expr)** *stmt*
| **for (** *optexpr*; *optexpr*; *optexpr***)** *stmt*
| **other**

optexpr æ **expr**
| ‘

The parser consists of the procedures *stmt*, *optexpr* for the non-terminals and *match* for matching terminals with the lookahead.

```
void stmt (){
    switch ( lookahead ){
    case expr :
        match(expr); match(';'); break;
    case if :
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    ...
    default :
        error("syntax error");
    }
}

void optexpr(){
    ...
}

void match(terminal t){
    /* When working with lex we normally define tokens to integers.
    Example: #define T_ID 5 in C defines T_ID as 5.
    So we can take the type terminal as int
    */

    /* nextTerminal below will be obtained by a call to yylex() */
    if (lookahead == terminal) lookahead = nextTerminal
    else error("syntax error");
}
```

In this lab you will combine a lexical analyzer created using flex with a predictive parser such as above. Write a lexical analyzer as before and from the parser call the function `yylex` whenever you need the next token. The predictive parser functions above can be written in the third section of the lex file.

The code for the example is included with the handout. Run `flex` on the lex file and compile with `g++`. Run the compiled parser on the sample input file. Try introducing syntax errors into the code and check the output of the parser.

The exercises requires you to write a parser for the expression grammar. From class recall that left-recursive grammars can cause predictive parsers to go into an infinite loop. Remember to rewrite such a grammar to remove left recursion.

Exercises

1. Write a parser for statements of the form *id* = *expr*;. Examples of valid input are `a=a+b*c;`, `a=3*b;` The grammar for this is the following.

$$
\begin{aligned}
stmt &\rightarrow \mathbf{id} = expr\ \mathbf{;} \\
expr &\rightarrow expr + term \\
&\mid term \\
term &\rightarrow term * factor \\
&\mid factor \\
factor &\rightarrow \mathbf{id} \\
&\mid \mathbf{num} \\
&\mid \mathbf{(}expr\mathbf{)}
\end{aligned}
$$

   Here **id** and **num** are the tokens for identifiers and numerals respectively. You need to extend the lexical analyzer to return these tokens. As it stands the grammar is left-recursive, so you need to rewrite the grammar to remove left recursion. As before the parser should output "syntax error" if the input is invalid.

2. Combine the expression parser from the previous exercise with the example. Replace the **expr** terminal with the grammar for *expr*.

CS424-L Compiler Construction Lab

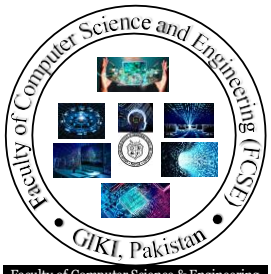
LAB IV – LEXICAL ANALYZERS & SYMBOL TABLES

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology



Faculty of Computer Science & Engineering (FCSE)

CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 4 — Lexical Analyzers and Symbol Tables in Compilers

Objective

The objective of this session is to learn to add support for declarations and nested scopes to the exercise from the previous lab.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Understand and explain the concept of symbol tables.
2. Add support for declarations and scopes to the parser for assignment statements.

Instructions

- Read through the handout sitting in front of a computer that has a C++ and Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Symbol Tables, Declarations and Nested Scopes

In this lab we add support for declarations and nested scopes to the exercise from the previous lab. Recall that scopes can be supported using symbol tables, where each block enclosed within braces gets its own symbol table. The symbol tables are chained together in a linked list. The last symbol table is currently active table. Whenever a new variable is declared, an entry is added

within the table. Whenever a variable is accessed, the linked list is traversed to find the first available declaration, which is then used. The table itself can be represented by a hashtable.

Given below is a rewrite of the symbol table example from the textbook in C++. The translation is done by implementing a syntax directed translation scheme (SDT) using a predictive parser. Input is of the form

```
{ int x; x; {float x; x; } x;}
```

Running the translator will tell you which usage of `x` corresponds to which declaration. Multiple declarations of the same variable in the same scope lead to an error. Any variable must be declared before it can be used. This implements the following syntax directed translation scheme from the book.

program	→	*block*	{ *top* = **null**; }
block	→	'{'	{ *saved* = *top*; *top* = **new** *Env*(*top*); print("{ "); }
		decls stmts '}'	{ *top* = *saved*; print("} "); }
decls	→	*decls decl*	
	|	ϵ	
decl	→	**type id** ;	{ *s* = **new** *Symbol*; *s.type* = **type**.*lexeme* *top.put*(**id**.*lexeme*, *s*); }
stmts	→	*stmts stmt*	
	|	ϵ	
stmt	→	*block*	
	|	*factor* ;	{ print("; "); }

factor	→	**id**	{ *s* = *top.get*(**id**.*lexeme*);
			print(**id**.*lexeme*);
			print(":"); }
			print(*s.type*);

The symbol table definitions are in the file `symtab.h`

```
#include <iostream>
#include <string>
#include <map>

using namespace std;

//symbol table entry
struct sym_ent {
    string type;
```

```
};

class sym_tab {
    public:
    //The symbol table is a C++ map (hash table)
    map<string, sym_ent*> table;
    //Tables are chained in a linked list
    sym_tab* prev;

    sym_tab(sym_tab* p){
        prev = p;
    }

    sym_tab* get_prev(){
        return prev;
    }

    bool put(string s, sym_ent* sym){
        if (table.count(s) == 0){
            table[s] = sym;
            return true;
        }
        return false;
    }


    sym_ent* get(string s){
        for(sym_tab* i = this; i!=NULL; i=i->prev){
            if (i->table.count(s) > 0)
                return i->table[s];
        }
        return NULL;
    }

    //delete table entries in destructor
    ~sym_tab(){
        for(map<string, sym_ent*>::iterator x=table.begin();
            x!=table.end(); x++){
            delete (*x).second;
        }
    }
};
```

The code for the translator is as follows.

```
%option noyywrap
%{
```

```
#include <cstdlib >
#include <cstdio >
#include <string >
#include <iostream >

#include "symtab. h"

using namespace std;


struct sym_ent;
class sym_tab;

void program ();
void match(int);
void error();

void block ();
void stmts ();
void stmt ();
void decls ();
void decl ();
void factor();

sym_tab* top=NULL;

int lookahead;
int line =1;


//tokens
const int T_TYPE=350;
const int T_ID=351;
%}

letter [A-Za-z]
digit [0-9]
id     {letter}({letter}|{digit})*

%%
[ \t]+     ;/*ignore whitespace */
\n                    { line++; }
int|float|bool|char { return T_TYPE; }
{id}                  { return T_ID; }
.                     { return *yytext;}
```

```
%%

int main(int argc, char* argv[])
{
    if (argc == 2)
        yyin = fopen(argv[1], "r");


    //read first token
    lookahead = yylex();
    program();

    //check for trailing input
    if (lookahead)
        error();
}

void match(int terminal)
{
    if (lookahead == terminal )
        lookahead = yylex();
    else
        error();
}

void error(){
    cerr<<"syntax error on line "<<line<<"\n";
    exit(1);
}

void program(){
        top = NULL;
    block();
}

void block(){
    match('{');

        //save old symbol table. Create new table
        sym_tab * saved = top ;
        top = new sym_tab(top);

        cout<<"{\n";

    decls();
```

```
    stmts ();
    match ('}');

        //restore previous scope
        delete top ;
        top = saved;
        cout <<"}\n";
}

void decls (){
    if (lookahead == T_TYPE ){
        decl (); decls ();
    }
}

void  decl (){
    if (lookahead  ==  T_TYPE){
            //store type lexeme in C++ string
            string  type(yytext);
        match(T_TYPE);

            //store id lexeme
            string  id(yytext);
        match(T_ID);

        match(';');

            //store symbol in symbol table
            sym_ent  *s  =  new  sym_ent;
            s->type  =  type;

            if (!top->put(id,  s)){
                cerr <<"Error:  Multiple  declarations  of  "<<id;
            }
    }else {
        error ();
    }
}

void stmts (){
    if (lookahead == '{' || lookahead  ==  T_ID ){
        stmt (); stmts ();
    }
}
```

```
void  stmt (){
     switch (lookahead){
     case '{':
          block ();
          break ;

     case  T_ID:
          factor (); match (';');
               cout<<";\n";
          break ;

     default :
          error ();

     }
}


void  factor (){
     if (lookahead == T_ID ){
          string id(yytext);
          match(T_ID);

               sym_ent* s = top->get(id);
               if (!s)
                    cerr<<"Undeclared variable "<<id<<'\n';
               else
                    cout<<id<<":"<<s->type;
     }else
          error ();
}
```

2 Exercises

1. Add support for declarations and scopes to the parser for assignment statements from the previous lab. Assignment statements are of the form *id* = *expr*;. Each variable must be declared before it can be used. To do this write a grammar. Convert the grammar into a syntax directed translation scheme. Then implement the SDT inside a predictive parser.

   For example, a valid input can be of the form:

```
{
    int x;
    int y;
    x = 2 + 3*4;
    {
        int x;
        x = 5+10;
    }
    y = 4 + x;
}
```

   Another example:

```
{
    int x;
    int y;
    bool y;
    x = 2 + 3*4;
    y = 4 + x;
}
```

   This should show a multiple declarations error, since `y` has been declared twice.

CS424-L Compiler Construction Lab

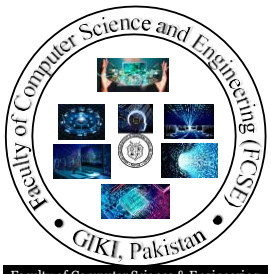
LAB V – SYNTAX TREES & INTERMEDIATE CODE GENERATION

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology



Faculty of Computer Science & Engineering (FCSE)

CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 5 — Syntax Trees and Intermediate Code Generation in Compilers

Objective

The objective of this session is to learn how to extend a predictive parser to generate a syntax tree and then traverse the syntax tree to generate intermediate code.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Extend a predictive parser to generate a syntax tree.
2. Traverse the syntax tree to generate immediate code.

Instructions

- Read through the handout sitting in front of a computer that has a C++ and Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Predictive Parser and Syntax Tree

In this lab we extend a predictive parser to generate a syntax tree. Then we traverse the syntax tree to generate intermediate code.

Recall that a syntax tree (or abstract syntax tree) is a representation of a source program where the internal nodes of the tree are program constructs. We will construct syntax trees for the following simple language.

program æ block
block æ { stmts }
stmts æ stmt stmts | ‘
stmt æ expr ;
stmt æ if (expr) stmt
stmt æ block
expr æ rel = expr | rel
rel æ factor
factor æ id | num

Since we will be using a predictive parser we need to first left factor the grammar and then check if the grammar is LL(1). Left factoring produces the following.

program æ block
block æ { stmts }
stmts æ stmt stmts | ‘
stmt æ expr ;
stmt æ if (expr) stmt
stmt æ block
expr æ rel r’
r’ æ = expr | ‘
rel æ factor
factor æ id | num

The only thing we need to check here is if the non-terminal *r’* can be used to uniquely identify one of its two productions. For this we look at the FIRST and FOLLOW sets. The FIRST set for the first of its two productions (i.e., FIRST(= *expr*)) is ***{=}***. The FOLLOW set of *r’* is ***{), ;}***. Since the two sets are disjoint the grammar is LL(1) and we can use a predicive parser.

To generate the syntax tree we use the following syntax directed translation scheme.

```
program æ block    {return block.n}
    block æ { stmts }    {block.n=stmts.n}
   stmts æ stmt stmts     {stmts.n = new seq(stmt.n, stmts.n)}
   stmts æ ‘    {stmts.n = null}
    stmt æ expr ;     {stmt.n = new eval(expr.n)}
    stmt æ if expr stmt     {stmt.n = new if(expr.n, stmt.n)}
    stmt æ block     {stmt.n = block.n}
    expr æ rel r     {if r'.n == null expr.n = rel.n else expr.n = assign(rel.n, r'.n)}
      r æ = expr     {r'.n = expr.n}
      r æ ‘    {r'.n = null}
     rel æ factor     {rel.n = factor.n}
  factor æ id     {factor.n = new id(id.lexeme)}
  factor æ num     {factor.n = new num(num.value)}
```

Each syntax tree node has a `gen()` function that generates the code for that node. Recall from class or look at the example code below to see how that works. Once we call the function at the top of the syntax tree, we get the code for the whole program as output.

2 Syntax Tree and Intermediate Code Generation

The example below implements the above translation scheme in C++. There are two files. The first `ast.h` contains the syntax tree code. The other is a lex file `predictive.l`. Run the example on the example program below.

```
{
a=1;
if (a){
   b=2;
   if (b)  {a=3;  z=10;}
}
h=10;
}
```

This should produce the following output.

```
a=1
ifFalse a goto .L1
a=2
ifFalse b goto .L0
a=3
z=10
```

```
.L0:
.L1:
h=10
```

The syntax tree code from `ast.h` is listed below

```
#include<string>
#include<iostream>
#include<cstdlib>
#include<sstream>
using namespace std;

namespace ast {
int t_idx = 0;
string temp(){
    stringstream ss;
    ss<<".t"<<t_idx++;
    return ss.str();

}

int l_idx = 0;
string label(){
    stringstream ss;
    ss<<".L"<<l_idx++;
    return ss.str();

}

class node{
};

class stmt:public node{
    public:
    virtual void gen(){
    }

};

class expr: public node{
    public:
        virtual expr * lvalue (){
            cerr<<"Error: invalid lvalue\n";
            exit(1);
        }
        virtual expr* rvalue(){
            return this;
```

```
        }

        virtual string str(){
            return "Not implemented";
        }
};

class if_stmt: public stmt{
    public:
        expr * left ;
        stmt *right;
        string l;


        if_stmt(expr* e, stmt* s):left(e),right(s){
            l = label();
        }

        void gen(){
            expr* e=left->rvalue();
            cout<<" ifFalse "<<e->str()<<" goto "<<l<<"\n";
            right->gen();
            //cout<<l<<":\n";
            cout<<l<<": ";
        }
};

class eval: public stmt{
    public:
        expr* left;

        eval(expr* e):left(e){}

        void gen(){
            left->rvalue();
        }
};

class seq: public stmt{
    public:
        stmt * left ;
        stmt* right;

        seq(stmt* l, stmt* r):left(l),right(r) {}
```

```
        void gen(){
            left->gen();
            if(right)
                right->gen();
        }
};


class op: public expr{
    public:
    enum op_type{OP_PLUS, OP_MINUS, OP_TIMES, OP_DIV};

    op_type type;
    expr *left, *right;
};


class assign:public expr{
    public:
        expr *left, *right;

        assign(expr* l, expr* r):left(l),right(r) {}

        expr* rvalue(){
            expr* l = left->lvalue();

            expr* r = right->rvalue();

            cout<<l->str()<<"="<<r->str()<<"\n";
            return r;

        }
};

class id:public expr{
    public:
        string lexeme;

        id(string name):lexeme(name) {}

        expr* lvalue(){
            return this;
        }

        string str(){
```

```
            return  lexeme;
        }
};


class num:public expr{
    public:
        int  value;

        num(int n):value(n)  {}

        string  str(){
            stringstream  t;
            t<< value ;
            return t.str();
        }
};
}
```

The code for the translator is as follows.

```
%option noyywrap

%{
#include<iostream>
#include<cstdlib>

#include "ast.h"
using namespace std;
//define token labels

const int  T_IF =310;
const int T_OTHER= 320;
const int T_ID = 350;
const int T_NUM= 360;

//define lookahead as a global variable
int lookahead;
int line = 1;

void match(int);
void error();

ast :: stmt * stmts ();
ast :: stmt * stmt ();
ast::stmt* program();
ast::stmt* block();
```

```
ast :: expr * expr ();
ast::expr* factor();
ast::expr* rel();
ast::expr*  rp();
%}

alpha [a- zA - Z_]
alphanum [0-9a-zA-Z_]
digit [0-9]
%%
[ \t]+ ;
\n   { line++;}
if   { return T_IF;}
{alpha}{alphanum}*   { return T_ID;}
{digit}+     {return T_NUM; }
.    { return *yytext;}
%%



int main(int argc, char* argv[])
{
     if(argc == 2)
          yyin = fopen(argv[1], "r");

     lookahead = yylex();
     ast::stmt* s=program();
     //check trailing input
     if(lookahead)
        error();

     //generate code. s points to the ast root
     s->gen();
}

ast::stmt* program(){
     return block();
}

ast::stmt*  block(){
     if(lookahead == '{'){
          ast::stmt* s;
          match('{'); s=stmts(); match('}');
          return s;
     }
```

```
    else
        error();
}


ast::stmt *stmts(){
    switch(lookahead){
    case T_ID:
    case T_IF:
            ast::stmt* l;
            ast::stmt* r;
        l = stmt(); r= stmts();
            return new ast::seq(l, r);

        break;
    case '{':
            ast::stmt *s;
        s = block();
            return s;
        break;
    default:
        return NULL;
    }
}

ast::stmt* stmt(){
    ast::expr* e;
    ast::stmt *s;

    switch(lookahead){
    case T_ID:
        e=expr(); match(';');
            return new ast::eval(e);
        break;

    case T_IF:
        match(T_IF); match('('); e=expr(); match(')'); s=stmt();
            return new ast::if_stmt(e, s);
        break;

    case '{':
        s=block();
            return s;
    default:
        error();
```

```
    }
}

//expr has been left factored
//expr -> rel rp
//rp -> = expr | epsilon
ast::expr* expr(){
    ast::expr *left, *right;
    switch(lookahead){
    case T_ID:
    case T_NUM:
        left = rel(); right = rp();

            //check if two or one children
            if(right==NULL){
                return left;
            }else{
                return new ast::assign(left, right);
            }
        break;

    default:
        error();
    }
}

// rp -> = expr | epsilon
// FOLLOW(rp) is ), ;
ast :: expr * rp (){
    switch(lookahead){
    case '=':
            ast :: expr * e;
        match('='); e=expr();
            return e;
        break;
    case ')':
    case ';':
        return NULL;
        break;

    default:
        error();

    }
```

```
}

ast::expr* rel(){
    return factor();
}

ast::expr* factor(){
    int value;
    string name(yytext);

    switch(lookahead){
    case T_ID:
            //store lexeme in string
        match(T_ID);
            //return new tree node
            return new ast::id(name);
        break;

    case T_NUM:
            //store lexeme's integer value
            value = atoi(yytext);
        match(T_NUM);
            return new ast::num(value);
        break;

    default:
        error();
    }
}

void match(int terminal)
{
    if(lookahead == terminal){
        lookahead = yylex();
    }else
        error();
}

void error(){
    cerr<<"syntax error on line "<<line<<"\n";
    exit(1);
}
```

3 Exercises

The example above only has if statements and assignments of the form `id=id` or `id=num`. Add support for while and do-while loops to the example. To do this study the `if_stmt` class in the example code. Your program should be able to translate the following code.

```
{
     a=1;
     if (a) {
          b=a;
          while (b) {
               a=2;
               b=a;
          }
          do{
               c=b;
          } while (b);
    }
    d=a;
}
```

CS424-L Compiler Construction Lab

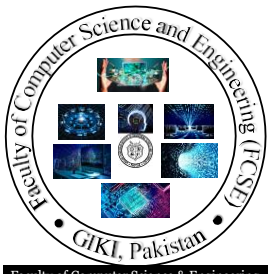
LAB VI – PARSING USING BISON

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology



Faculty of Computer Science & Engineering (FCSE)

CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 6 — Parsing Through Bison in Compilers

Objective

The objective of this session is to learn the concept of parsing a grammar and how to use Bison to perform parsing.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Explain the concept of parsing.
2. Use Bison to generate a parser for a specific grammar.

Instructions

- Read through the handout sitting in front of a computer that has a C++ and Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Introduction to Bison

Bison is a parser generator. It takes as input a grammar and some code and produces a parser for that grammar. The generated parser is a shift-reduce parser.

The input to Bison is specified in a file with a '.y' extension. The format of the file is as follows.

```
declarations
%%
grammar rules
%%
functions
```

Bison requires a lexical analyzer defined as a function `yylex`. This function can be created by hand or by using Flex. It also defines a function `yyparse` which is invoked to run the parser. A default function `yyerror` is called when there is an error. This function can be redefined.

We will be using Flex for lexical analysis, but there are a few things that we will do differently. One of these is that tokens will now be defined in the bison file rather than in the flex specification which will only use them.

2 Examples

2.1 Example 1: Parsing an Expression

Here's an example.

```
%{
#include <cstdio >
#include <cstdlib >
#include <iostream >
using namespace std;
//declare lexical analyzer. This will be generated using lex.
int yylex(void);
//input file pointer
extern FILE* yyin;
//This function is called on error
void yyerror(const char *);
%}

/* tokens will be declared in the .y file now */
%token NUM

%%
exprlist: exprlist expr ';'
            | /* empty */
            ;

expr: term
      | expr '+' term
        ;

term: NUM;
%%
void yyerror(const char *error)
```

```
{
    std::cerr<<error<<'\n';
}

int main(int argc, char *argv[]){
    if(argc==2)
        yyin = fopen(argv[1], "r");
    //run parser
    yyparse();
    return 0;
}
```

Put this in a file `example.y` and run the command `bison -d example.y` on the command prompt. This will output two files called `example.tab.c`, containing the parser, and `example.tab.h` containing the token definitions. The Flex file is as follows.

```
%option noyywrap
%{
#include<cstdlib>
using namespace std;

//since we are using bison,
//token declarations are now in the
//following header
#include "example.tab.h"
%}

%%
[ \t]+   ;
[0-9]+   {return NUM;}
.        {return *yytext;}
%%
```

Put this in a file `example.l` and run the command `flex example.l` on the command prompt. Now compile the parser by running `g++ example.tab.c lex.yy.c`. Run the generated program and give some expressions in a file or on standard input followed by a semicolon. If you pass an incorrect expression it should give an error.

The first part of the bison file declares a token or terminal NUM. This is used in the grammar rules and by the lexical analyzer. The second section specifies the grammar rules. There are three non-terminals. Productions for the same non-terminal are separated by the pipe symbol. For simplicity, for single character terminals the character itself is used.

This parser simply parses without doing any translation. We can also add semantic actions to productions which are code fragments to be executed whenever the right hand side of a production is reduced to the left hand side.

2.2 Example 2: Evaluating an Expression

The following modifies the first example to compute the value of the expression.

```
%{
#include <cstdio >
#include <cstdlib >
#include <iostream >
using namespace std;
//declare lexical analyzer. This will be generated using lex.
int yylex(void);
//input file pointer
extern FILE* yyin;
//This function is called on error
void yyerror(const char *);
%}

/* tokens will be declared in the .y file now */
%token NUM

%%
exprlist: exprlist expr ';'     {printf("%d\n", $2);}
        |                       /* empty */
        ;

expr: term              {$$ = $1; }
    | expr '+' term { $$ = $1 + $3; }
    ;

term: NUM     {$$ = $1;}
    ;

%%
void yyerror(const char *error)
{
    std::cerr <<error <<'\n';
}

int main(int argc, char *argv[]) {
    if (argc==2)
        yyin = fopen(argv[1], "r");
    //run parser
    yyparse ();
    return 0;
}
```

The flex file is modified as follows.

```
%option noyywrap
%{
#include "example2.tab.h"
```

```
%}

%%
[ \t]+      ;/* ignore whitespace */
[0-9]+        {
                yylval = atoi(yytext);
                return NUM;
              }
.             {return *yytext;}
%%
```

Now the productions have been augmented by adding code fragments. These code fragments are executed when the body of the production is reduced. Each grammar symbol in a production has an attribute: $1 is the attribute of the first symbol in the body, $2 for the second and so on. $$ is the attribute corresponding to the non-terminal on the left of the production. For terminals the attribute corresponds to the variable `yylval`. In other words, whatever is put in `yylval` can be accessed by the attribute in the bison specification. Here `yylval` is set by the lexical analyzer to the integer value of the number. The default type of `yylval` is `int`.

2.3 Example 3: Evaluating an Expression with Double Values

The following modifies the above to handle floating point numbers.

```
%{
#include <cstdio>
#include <iostream>
int yylex (void);
extern FILE* yyin;
void yyerror(const char*);
%}

/* type of yylval is now union with a single float inside */
/* %union redefines the type of the attributes */
%union {
    double value;
}

%token <value> NUM
/* this redefines the attribute type for the non-terminals */
%type <value> term expr

%%
exprlist: exprlist expr ';'     {printf("%f\n", $2);}
        |                       /* empty */
        ;

expr: term              {$$ = $1; }
```

```
    | expr '+' term {$$ = $1 + $3; }
    ;

term: NUM     {$$ = $1;}
    ;
%%
void yyerror(const char* error)
{
    std::cerr<<error<<'\n';
}

int main(int argc, char *argv[]){
    if (argc==2)
        yyin = fopen(argv[1], "r");
    yyparse();
    return 0;
}
```

The %union declaration changes the type of yylval from its default of int to a union with a single member value with type double. The next two lines declare that the attributes for DIGIT, term and expr be taken from the value member of yylval.

The flex file is modified as follows.

```
%option noyywrap
%{
#include "example3.tab.h"
%}

%%
[ \t]+     ;/*ignore whitespace */
[0-9]+(\.[0-9]+)?   {
            /* yylval is a union */
            yylval.value = atof(yytext);
             return NUM;
            }
.           {return *yytext;}
%%
```

3 Exercises

1. Change the last example above to include subtraction, division, multiplication and parentheses in expressions. Handle floating point numbers.

CS424-L Compiler Construction Lab

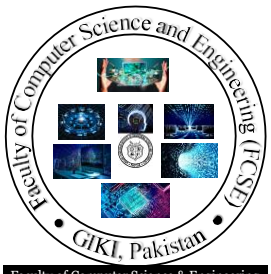
LAB VII – PARSING USING BISON WITH AMBIGUOUS GRAMMAR

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology



Faculty of Computer Science & Engineering (FCSE)


CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 7 — Parsing Through Bison with Ambiguous Grammars

Objective

The objective of this session is to learn parsing ambiguous grammars using Bison.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Explain and identify ambiguous grammars.
2. Use Bison to parse ambiguous grammars.

Instructions

- Read through the handout sitting in front of a computer that has a C++ and Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Bison with Ambiguous Grammars

Bison can also be made to work with some ambiguous grammars. Recall that one way of dealing with ambiguity is to rewrite the grammar. The other is to use some ad-hoc rules to help remove ambiguity.

2 Examples

2.1 Example 1: An Ambiguous Grammar

Run the following example.

```
%{
#include <stdio.h>
int yylex ( void );
extern FILE* yyin;
%}

%token NUM

%%
exprlist: exprlist expr '\n'
        | /* empty */
        ;

expr: expr '*' expr
    | expr '+' expr
    | NUM
    ;

%%
int main(int argc, char *argv[]){
    if (argc==2)
        yyin = fopen(argv[1], "r");
    yyparse();
    return 0;
}
```

The Flex file is as follows.

```
%option noyywrap
%{
#include <stdio.h>
#include "example.tab.h"
%}

%%
[ \t]+     ;/*ignore whitespace */
[0-9]+       {
              yylval = atoi(yytext);
              return NUM;
             }
[+\n]        {return *yytext;}
.            {return *yytext;}
```

```
%%
```

If you run Bison, it should say that it encountered shift-reduce conflicts, since the grammar is ambiguous. You can see the states of the automaton (item sets) and by running `bison -d -v example.y`. This will generate a file that shows the states of the automaton and tells you where the shift-reduce conflicts were found.

There default actions for shift-reduce and reduce-reduce conflicts in bison. For a shift-reduce conflict, it is to shift. For a reduce-reduce conflict the action is to reduce using the production that comes first.

2.2 Example 2: Making the Grammar Unambiguous

We can make the grammar unambiguous by declaring explicitly the precedence of the operators along with their associativity. The following also computes the value of the expression.

```
%{
#include <stdio.h>
int yylex ( void );
extern FILE* yyin;
%}

%token NUM
%left '+' '-'
%left '*' '/'

%%
exprlist: exprlist expr '\n' {printf("result: %d\n", $2);}
        |                    /* empty */
        ;

expr: expr '+' expr {$$ = $1 + $3;}
    | expr '*' expr {$$ = $1 * $3;}
    | NUM           {$$ = $1;}
    ;

%%
int main(int argc, char *argv[]) {
    if (argc==2)
        yyin = fopen(argv[1], "r");
    yyparse();
    return 0;
}
```

The line `%left '+' '-'` says that plus and minus are left associative and have the same precedence. The next line `%left '*' '/'` says that multiplication and division have the same precedence but higher than that of plus and minus. In general the operators appear in increasing order of precedence.

3 Exercises

1. Modify the second example to evaluate expressions involving addition, subtraction, multiplication, division and exponentiation where the numbers might be integers or floating point numbers. You can use the `pow` function in `math.h` for exponentation.

CS424-L Compiler Construction Lab

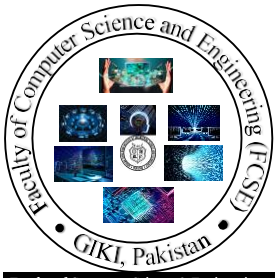
LAB VIII – SYNTAX TREES & INTERMEDIATE CODE GENERATION

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology



Faculty of Computer Science & Engineering (FCSE)


CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 8 — Syntax Trees and Intermediate Code Generation

Objective

The objective of this session is to learn how to traverse the syntax tree to generate intermediate code using a bottom-up parser.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Explain what a bottom-up parser is.
2. Traverse the syntax tree to generate immediate code using a bottom-up parser.

Instructions

- Read through the handout sitting in front of a computer that has a C++ and Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Syntax Trees and Intermediate Code Generation

In lab 5 we used a predictive parser to generate a syntax tree and traversed the tree to generate intermediate code. In this lab we use a bottom-up parser to do the same. Consult the lab handout for lab 5 for details.

Recall that a syntax tree (or abstract syntax tree) is a representation of a source program where the internal nodes of the tree are program constructs. We will construct syntax trees for the following simple language.

program æ *stmts*
stmts æ *stmt stmts* | ‘
stmt æ **id** = *expr* ;
stmt æ *if* (*expr*)*stmt*
expr æ *expr* + *expr*
expr æ *expr* ú *expr*
expr æ **num**
expr æ **id**

We will write a syntax directed translation scheme in Bison to generate a syntax tree. The syntax tree code is almost the same as from lab 5. The difference is that now we also have expressions.

Given an input of the following form

```
if (a+b+c)
    a=b+1;
c=a+1;
```

our goal is to produce the following.

```
.t1=a+b
.t0=.t1+c
ifFalse .t0 goto .L0
.t2=b+1
a=.t2
.L0:
.t3=a+1
c=.t3
```

The code for the parser is as follows. Consult the files `ast.cpp` and `ast.h` for the syntax tree code.

```
%{
#include <cstdio>
#include <iostream>
#include <string>
#include "ast.h"

using namespace std;

int yylex(void);
extern FILE* yyin;
```

```
void yyerror ( const char *);
string Temp();

int var = 0;
%}

/* This sets the type of yylval. */

%union {
    int    value  ;
    std::string* id;
    ast :: expr * pe;
    ast::stmt* ps;
};

%token T_IF
/* The types of tokens    and non-terminals are chosen from the yylval
   union. */
%token <id> T_ID
%token <value> NUM
%type <pe> expr
%type <ps> A
%type <ps> S
%type <ps> Seq

%left '+' '-'
%left '*' '/'

%%
P : Seq {
        //walk syntax tree to generate code
        $1->gen();
    }

Seq : Seq S { $$ = new ast::seq($1, $2); }
  | %empty {$$ = NULL; }

S : A { $$=$1;}
  | T_IF '(' expr ')' S {
                          $$ = new ast::if_stmt($3, $5);
                      }

;

A : T_ID '=' expr ';' {
```

```
                        ast :: id  * t  =  new  ast :: id (* $1 );
                        ast::expr* p = new ast::assign(t, $3);
                        $$ = new ast::eval(p);
                        }
    ;

expr: expr '+' expr {

                        $$ = new ast::op("+", $1, $3);
                        }
    | expr '*' expr {
                        $$ = new ast::op("*", $1, $3);

                        }
    | NUM               {
                        $$ = new ast::num($1);
                        }

    |  T_ID              {
                        $$ = new ast::id(*$1);
                        }

    ;
%%

//Creates a new temporary variable such as ".L5"
string Temp(){
    stringstream t;
    t<<".L"<<var;
    var++;
    return t.str();
}

void yyerror (const char *error)
{
  std::cout << error << '\n';
}


int main(int argc, char *argv[]) {
    if (argc==2)
        yyin = fopen(argv[1], "r");
    yyparse();
    return 0;
}
```

2 Exercises

1. The example above has expressions with addition and multiplication. Add support for division and subtraction and subexpressions. You should be able to translate expressions of the form a*(b+2)+5/d.

2. Add support for while and do-while loops to the example.

CS424-L Compiler Construction Lab

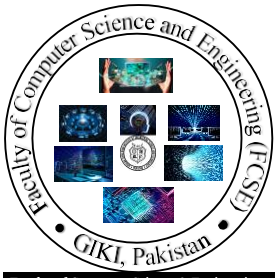
LAB IX – INTERMEDIATE CODE FOR EXPRESSIONS

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology



Faculty of Computer Science & Engineering (FCSE)

CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 9 — Intermediate Code for Expressions

Objective

The objective of this session is to learn how to traverse the syntax tree to generate intermediate code using top-down approach.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Differentiate between top-down and bottom-up parsing.
2. Generate intermediate code for expressions using a top-down parser.

Instructions

- Read through the handout sitting in front of a computer that has a C++ and Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Intermediate Code for Expressions

In the previous lab we generated code for expressions using a bottom-up parser. This time we use a top-down parser to do the same.

The new grammar has the operators + and − corresponding to the token op and looks like the following. Compare with the previous lab.

$$
\begin{aligned}
&program \text{ æ } block\\
&block \text{ æ } \{\ stmts\ \}\\
&stmts \text{ æ } stmt\ stmts \mid \text{‘}\\
&stmt \text{ æ } expr\ ;\\
&stmt \text{ æ } if\ (expr\ )stmt\\
&stmt \text{ æ } block\\
&expr \text{ æ } rel = \ expr \mid rel\\
&rel \text{ æ } add\\
&add \text{ æ } add\ op\ factor\\
&add \text{ æ } factor\\
&factor \text{ æ } id \mid num
\end{aligned}
$$

Left factoring and removing left recursion produces the following.

$$
\begin{aligned}
&program \text{ æ } block\\
&block \text{ æ } \{\ stmts\ \}\\
&stmts \text{ æ } stmt\ stmts \mid \text{‘}\\
&stmt \text{ æ } expr\ ;\\
&stmt \text{ æ } if\ (expr)\ stmt\\
&stmt \text{ æ } block\\
&expr \text{ æ } rel\ r'\\
&r' \text{ æ } = \ expr \mid \text{‘}\\
&rel \text{ æ } add\\
&add \text{ æ } factor\ add'\\
&add' \text{ æ } op\ factor\ add' \mid \text{‘}\\
&factor \text{ æ } id \mid num
\end{aligned}
$$

To generate the syntax tree we use the following syntax directed translation scheme.

```
program æ block    {return block.n}
    block æ { stmts }    {block.n=stmts.n}
   stmts æ stmt stmts    {stmts.n = new seq(stmt.n, stmts.n)}
   stmts æ ‘    {stmts.n = null}
    stmt æ expr ;    {stmt.n = new eval(expr.n)}
    stmt æ if (expr) stmt    {stmt.n = new if(expr.n, stmt.n)}
    stmt æ block    {stmt.n = block.n}
    expr æ rel r'    {if r'.n == null expr.n = rel.n else expr.n = assign(rel.n, r'.n)}
      r' æ = expr    {r'.n = expr.n}
      r' æ ‘    {r'.n = null}
     rel æ add    {rel.n = add.n}
     add æ factor add'
        {if add'.n=null add.n=factor.n; else add'.n.left = factor.n; add.n=add'.n}
    add' æ op factor add'2
        {add'.n = new Op(op); if add'2.n=null add'.n.right= factor.n);
         else add'2.n.left = factor.n; add'.n.right=factor.n}
    add' æ ‘    add'.n =null
  factor æ id    {factor.n = new id(id.lexeme)}
  factor æ num    {factor.n = new num(num.value)}
```

Given the following input.

```
{
     a=b+c+d-e+400;
     if (a+b-c){
          c=d;
          b=10;

     }
}
```

This should produce the following output. The variables that start with a t are compiler generated temporary variables.

```
t.0=e +400
t.1=d- t.0
t.2=c+t.1
t.3=b+t.2
a=t.3
t.4=b-c
t.5=a+t.4
```

```
ifFalse t.5 goto .L0
c=d
b=10
.L0:
```

The code for the C++ implementation is listed below.

```
#include<string>
#include<iostream>
#include<cstdlib>
#include<sstream>
using namespace std;

namespace ast{
int t_idx = 0;
string temp(){
    stringstream ss;
    ss<<"t."<<t_idx++;
    return ss.str();

}

int l_idx = 0;
string label(){
    stringstream ss;
    ss<<".L"<<l_idx++;
    return ss.str();

}

class node{
};

class stmt:public node{
    public:
    virtual void gen(){
    }

};

class expr : public node {
    public:
        virtual expr * lvalue (){
            cerr<<"Error: invalid lvalue\n";
            exit(1);
        }
        virtual expr* rvalue(){
```

```
            return this;
        }

        virtual string str(){
            return "Not implemented";
        }
};


class id:public expr{
    public:
        string lexeme;

        id(string name):lexeme(name) {}

        expr* lvalue(){
            return this;
        }

        string str(){
            return lexeme;
        }
};


class num:public expr{
    public:
        int value;

        num(int n):value(n) {}

        string str(){
            stringstream t;
            t<< value ;
            return t.str();
        }
};

class if_stmt : public stmt {
    public:
        expr * left ;
        stmt *right;
        string l;
```

```
        if_stmt(expr* e, stmt* s):left(e),right(s){
            l = label();
        }

        void gen(){
            expr* e=left->rvalue();
            cout<<"ifFalse "<<e->str()<<" goto "<<l<<"\n";
            right->gen();
            cout<<l<<":\n";
        }
};

class eval : public stmt {
    public:
        expr* left;

        eval(expr* e):left(e){}

        void gen(){
            left->rvalue();
        }
};

class seq : public stmt {
    public:
        stmt *left ;
        stmt* right;

        seq(stmt* l, stmt* r):left(l),right(r) {}

        void gen(){
            left->gen();
            if(right)
                right->gen();
        }
};


class op: public expr{
    public:
    enum op_type{OP_PLUS, OP_MINUS, OP_TIMES, OP_DIV};
    string _op;

    op_type type;
    expr *left, *right;
```

```
    op(expr* l,   expr* r,  string  op):left(l),right(r),_op(op){}


    //generates code thatcomputes the value of an expression.
    //returns an id node containing the result
    expr* rvalue(){
        expr *l = left->rvalue();
        expr* r = right->rvalue();

        id* tid = new id(temp());
        cout<<tid->str()<<"="<<l->str()<<_op<<r->str()<<"\n";
        return tid;
    }
};

class assign:public expr{
    public:
        expr *left, *right;

        assign(expr* l,   expr* r):left(l),right(r)  {}

        expr* rvalue(){
            expr *l = left->lvalue();

            expr* r = right->rvalue();

            cout<<l->str()<<"="<<r->str()<<"\n";
            return r;

        }
};


}
```

The code for the translator is as follows.

```
%option noyywrap

%{
#include<iostream>
#include<cstdlib>

#include "ast.h"
using namespace std;
//define token labels
```

```
const  int   T_IF =310;
const int T_OTHER = 320;
const  int T_ID = 350;
const  int T_NUM = 360;
const int T_RELOP = 370;
const  int T_OP  =  380;

//define lookahead as a global variable
int  lookahead;
int  line  =  1;

void  match(int);
void  error();

ast :: stmt * stmts ();
ast :: stmt * stmt ();
ast::stmt* program();
ast :: stmt * block ();
ast :: expr * expr ();
ast :: expr *  rp ();
ast::expr* factor();
ast::expr* rel();
ast::expr* add();
ast::op* addp();
%}

alpha [a- zA - Z_]
alphanum [0-9a-zA-Z_]
digit [0-9]
%%
[ \t]+ ;
\n   { line++;}
\<|\<=|\>=|==|\>  {return  T_RELOP;}
\+|\-  {return  T_OP;}
if   {  return  T_IF;}
{alpha}{alphanum}*   {  return  T_ID;}
{digit}+     {return  T_NUM;  }
.    {  return  *yytext;}
%%



int  main(int  argc,  char*  argv[])
{
     if(argc  ==  2)
```

```
        yyin = fopen(argv[1], "r");

    lookahead = yylex();
    ast::stmt* s=program();
    //check trailing input
    if (lookahead)
        error();

    //generate code. s points to the root
    s->gen();
}

ast::stmt* program(){
    return block();
}

ast::stmt* block(){
    if (lookahead == '{'){
        ast::stmt* s;
        match('{'); s=stmts(); match('}');
        return s;
    }
    else
        error();
}


ast :: stmt * stmts (){
    switch (lookahead){
    case T_ID:
    case T_IF:
            ast::stmt* l;
            ast::stmt* r;
        l = stmt(); r= stmts();
            return new ast::seq(l, r);

        break;
    case '{':
            ast::stmt *s;
        s = block();
            return s;
        break;
    default:
        return NULL;
    }
```

```
}

ast::stmt* stmt(){
    ast::expr* e;
    ast::stmt *s;

    switch(lookahead){
    case T_ID:
        e=expr(); match(';');
            return new ast::eval(e);
        break;

    case T_IF:
        match(T_IF); match('('); e=expr(); match(')'); s=stmt();
            return new ast::if_stmt(e, s);
        break;

    case '{':
        s=block();
            return s;
    default:
        error();

    }
}

//expr has been left factored
//expr -> rel rp
//rp -> = expr | epsilon
ast::expr* expr(){
    ast::expr *left, *right;
    switch(lookahead){
    case T_ID:
    case T_NUM:
        left = rel(); right = rp();

            //check if two or one children
            if(right==NULL){
                return left;
            }else{
                return new ast::assign(left, right);
            }
        break;

    default:
```

```
            error();
        }
}

// rp -> = expr | epsilon
// FOLLOW(rp) is ), ;
ast :: expr * rp (){
    switch(lookahead){
    case '=':
            ast :: expr * e;
        match('='); e=expr();
            return e;
        break;
    case ')':
    case ';':
        return NULL;
        break;

    default:
        error();

    }
}

ast::expr* rel(){
    return add();
}

ast :: expr *  add (){
    ast::expr* operand;
    ast::op* rest;

    operand = factor();

    //addp returns an operator node
    rest= addp();
    if(rest !=NULL){
        rest->left = operand;
        return rest;
    }else
        return operand;
}

//addp -> op factor addp | epsilon
ast::op* addp(){
```

```
    ast::expr* operand;
    ast::op* next;
    if(lookahead == T_OP){
        //store operator in a string
        string sop = string(yytext);

        match ( T_OP );
        operand = factor();
        next= addp();

        ast::op* t_op = new ast::op(NULL, NULL, sop);
        if(next != NULL){
            next->left = operand;
            t_op->right = next;
        }else{
            t_op->right = operand;
        }
        return t_op;
    }
    return NULL;
}

ast::expr* factor(){
    int value;
    string name(yytext);

    switch(lookahead){
    case T_ID:
            //store lexeme in string
        match(T_ID);
            //return new tree node
            return new ast::id(name);
        break;

    case T_NUM:
            //store lexeme's integer value
            value = atoi(yytext);
        match(T_NUM);
            return new ast::num(value);
        break;

    default:
        error();
    }
}
```

```
void  match(int  terminal)
{
     if(lookahead==terminal){
          lookahead  =  yylex();
     }else
          error();
}

void error(){
     cerr<<"syntax  error  on line  "<<line<<"\n";
     exit(1);
}
```

Exercises

1. The example above only has the addition and subraction operators. Add support for multiplication and division (with the correct precedence) to the above example.

CS424-L Compiler Construction Lab

LAB X – TRANSLATING LOGICAL & RELATIONAL EXPRESSIONS

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Faculty of Computer Science & Engineering (FCSE)

CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 10 — Translating Logical and Relational Expressions

Objective

The objective of this session is to learn how to translate a logical and relational expression using jumping code.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Explain what logical expressions are and the concept of jumping code.
2. Translate logical expressions using jumping code.

Instructions

- Read through the handout sitting in front of a computer that has a C++ and Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Translating Logical and Relational Expressions

In this lab we translate logical expressions using jumping code. Logical expressions involve logical operators such as ||, &&, !. Recall from class that these do not appear in the translation but are instead handled using jumps. For example the statement `a || b` can be translated using

```
if a
goto L1
if b
goto L1
goto L2

#true jumps go here
L1: ...

#false jumps go here
L2: ...
```

The following example code generates the code for just the statements involving logical AND and relational operators. So given input code of the form

```
if (a+b<b+c&&c<d)
    a=1;
a=2;
```

it produces the following output.

```
.t0=a+b
.t1=b+c
if .t0 < .t1 goto .L12
goto .L7
.L12: if c < d goto .L8
goto .L7
.L8:a=1
.L7:a=2
```

This code is generated by traversing an abstract syntax tree (as before). There are special nodes for logical and relational operators. See the file ast.cpp for details on how this is handled. The AST code currently only handles the AND operator. The OR operator is an exercise for this lab.

The code for the parser is given below.

```
%{
#include <cstdio>
#include <iostream>
#include <string>
#include "ast.h"

using namespace std;

int yylex(void);
extern FILE* yyin;
void yyerror(const char*);
string Temp();

int var = 0;
```

```
%}

%union {
    int       value      ;
    std :: string *   id;
    std :: string * relop ;
    ast  ::  expr  *   pe;
    ast ::boolean_expr* be;
    ast::stmt* ps;
};

%token  T_IF
%token  T_AND
%token  T_OR
%token  <relop>  T_RELOP
%token  <id>  T_ID
%token  <value>  NUM

%type  <pe>  expr
%type  <ps>  A
%type  <ps>  S
%type  <be>  B
%type  <ps>  Seq

%left  '+'  '-'
%left  '*'  '/'
%left  T_AND
%left  T_OR


%%
P : Seq  {
        //walk syntax tree to generate code
        $1->gen();
    }

Seq  :  Seq  S  { $$  =  new  ast::seq($1,  $2); }
  |   {$$  =  NULL; }

S  :  A  { $$=$1;}
  |  T_IF  '('  B  ')'  S  {
                          $$  =  new  ast::if_stmt($3,  $5);
                      }

;
```

```
A : T_ID '=' expr ';' {
                        ast :: id * t = new ast :: id (* $1 );
                        ast::expr* p = new ast::assign(t, $3);
                        $$ = new ast::eval(p);
                        }
    ;

expr: expr '+' expr {

                        $$ = new ast::op("+", $1, $3);
                        }
    | expr '*' expr {
                        $$ = new ast::op("*", $1, $3);

                        }
    | NUM               {
                        $$ = new ast::num($1);
                        }

    |  T_ID              {
                        $$ = new ast::id(*$1);
                        }

    ;

B: B T_AND B { $$ = new ast::logical_expr($1, $3, "&&"); }


    | expr T_RELOP expr {$$ = new ast::relational_expr($1, $3, *$2);
        }
    ;

%%

//Creates a new temporary variable such as ".L5"
string Temp(){
    stringstream t;
    t<<".L"<<var;
    var++;
    return t.str();
}

void yyerror (const char *error)
{
```

```
    std::cout << error << '\n';
}


int main(int argc, char *argv[]) {
    if (argc==2)
        yyin = fopen(argv[1], "r");
    yyparse();
    return 0;
}
```

2 Exercises

1. Complete given example for the complete set of logical operators. For this you will need to add code to the `ast.cpp` file and the parser file.
2. Add support for translating `if/else` statements. Make sure to check that the jumps work when the statement is deeply nested.

CS424-L Compiler Construction Lab

LAB XI – TYPE CHECKING & TYPE COERCION

Dr. Hashim Ali, Engr. Badre & Mr. Adeel

Faculty of Computer Science & Engineering

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology

Faculty of Computer Science & Engineering (FCSE)

CS424 - Compiler Construction Lab (using Flex and Bison)

Lab 11 — Type Checking and Type Coercion

Objective

The objective of this session is to learn adding basic type checking and type coercion to a compiler.

Learning Outcomes

After the successful completion of this lab, the student will be able to:

1. Explain the concepts of basic type checking and type coercion.
2. Implement type checking and type coercion in compilers.

Instructions

- Read through the handout sitting in front of a computer that has a C++ and Flex software.
- Practice each new command by completing the examples and exercise.
- Turn-in the answers for all the exercise problems as your lab report.
- When answering the problems, indicate the commands you entered and output displayed.
- It may be easier to open a word document, and after each command you may copy and paste the commands and outputs to the word document.

1 Type Checking and Type Coercion

In this lab we add basic type checking and type coercion to our code. Type checking involves assigning types to nodes in the syntax tree and checking whether the assigned types satisfy the rules of the language. Type coercion is implicit conversion between some types to satisfy the rules.

In our case we have just two types `int` and `float`. An `int` can be coerced into a float but the reverse operation is invalid in our language. To force a conversion into float we assume that we

have a function `float` that can convert an integer representation to a floating point representation. To apply type coercion to an expression `a+b`, we find the larger of the two types (say float) of `a` and `b`, and add an instruction to convert the type of the smaller one (say int).

The following shows an example.

```
float a;
int b;
int c;

b = 2;
c = 3;
a = b+c;

if (a+b<c && b+2<3)
    b=3;
```

This produces the following output.

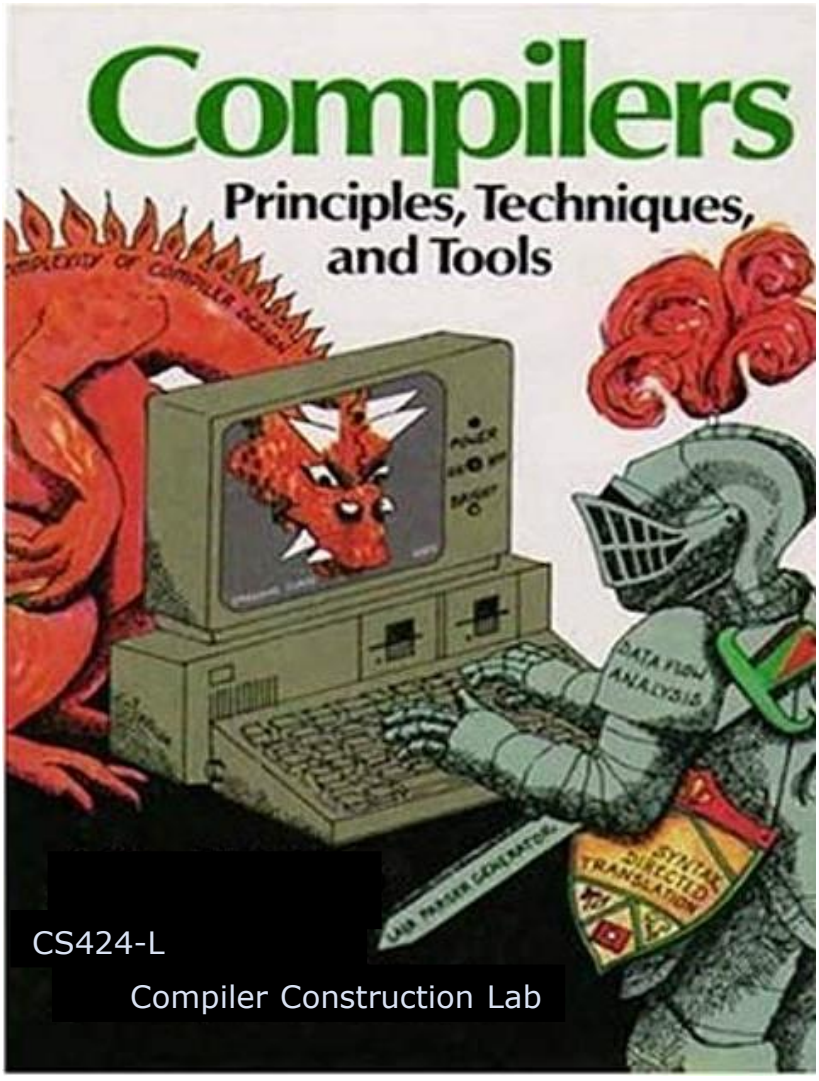
```
b=2
c=3
.t0=b+c
.t1=.t0
a= float (.t1)
.t3= float (b)
.t3=a+.t3
if .t3 < c goto .L19
goto .L16
.L19:.t4=b+2
if .t4 < 3 goto .L17
goto .L16
.L17:b=3
.L16:
```

2 Exercises

1. Add a new type bool to the language and two constants `true` and `false`. The bool type can be implicitly converted into `int` or `float`. Extend the `if` statement so that it also accepts statement of the form `if(a) ...` with a single argument. The condition in the `if` statement must have type bool so implement coercion for the case when it is not.

Ghulam Ishaq Khan Institute of Engineering Sciences & Technology (GIKI)

Faculty of Computer Science & Engineering (FCSE)



CS424L – Compiler Construction Lab Manual

PREPARED BY DR. HASHIM ALI & ENGR. BADRE MUNEER