

compiler lab for btech s7 cse tutorial

Published: September 27, 2026 | Index ID: #-

Compiler Lab for B.Tech S7 CSE Tutorial

In the world of computer science, a compiler is the bridge that translates human readable source code into machine executable instructions. For students in the 7th semester of B.Tech Computer Science Engineering (CSE), a hands on compiler lab is an indispensable part of the curriculum. This tutorial walks you through every step of setting up, executing, and mastering the compiler lab, ensuring you gain both theoretical knowledge and practical experience.

Why the Compiler Lab Matters in B.Tech S7 CSE

By the time students reach the 7th semester, they have already covered data structures, algorithms, operating systems, and computer architecture. A compiler lab complements these topics by:

- **Reinforcing Theory:** It demonstrates how lexical analysis, parsing, semantic analysis, and code generation work in real time.
- **Building Practical Skills:** Students learn to debug complex compiler components, a skill highly valued by employers.
- **Encouraging Research:** The lab encourages exploration of optimization techniques, which can lead to research projects or internships.
- **Facilitating Collaboration:** Working in teams on compiler projects mirrors industry practices, enhancing teamwork and communication.

Getting Started: Lab Objectives and Learning Outcomes

The primary objectives of the compiler lab for B.Tech S7 CSE are:

1. Understand the architecture of a compiler.
2. Implement a simple lexical analyzer using tools like Flex or ANTLR.
3. Build a parser that constructs an abstract syntax tree (AST).
4. Perform semantic analysis, including type checking and symbol table management.
5. Generate intermediate code (Three Address Code or SSA form).
6. Translate intermediate code into target machine code or assembly.
7. Apply optimization passes such as constant folding, dead code elimination, and loop invariant code motion.
8. Debug and test the compiler using a suite of test programs.

Upon successful completion, students will be able to:

- Design a modular compiler for a subset of a programming language.
- Utilize compiler construction tools effectively.
- Analyze and improve compiler performance.
- Document their design and implementation choices clearly.

Prerequisites: Knowledge and Tools

Before diving into the lab, students should be comfortable with:

- Programming in C/C++ or Java.
- Basic concepts of operating systems and computer architecture.
- Understanding of data structures such as stacks, queues, trees, and hash tables.
- Experience with command line interfaces and version control systems like Git.

Essential tools for the lab include:

- Flex (Fast Lexical Analyzer) or ANTLR for lexical analysis.
- Bison or ANTLR for parsing.
- LLVM or GCC as a backend for code generation.
- GDB or LLDB for debugging.
- Make or CMake for build automation.
- Git for version control.
- Integrated Development Environment (IDE) such as Visual Studio Code or CLion.

Lab Environment Setup

Installing the Required Software

Below is a step by step guide for setting up a typical Linux environment (Ubuntu 22.04). Windows users can adapt these commands using WSL or Cygwin.

1. Update Package Lists:
`sudo apt update`
`sudo apt upgrade -y`
2. Install Flex and Bison:
`sudo apt install flex bison`
3. Install LLVM:
`sudo apt install llvm clang`
4. Install Git:
`sudo apt install git`
5. Install CMake (optional but recommended):
`sudo apt install cmake`
6. Install Visual Studio Code (or your preferred IDE):
`sudo snap install --classic code`

Creating a Repository

Organize your lab work in a Git repository for version control and collaboration.

```
mkdir compiler_lab
cd compiler_lab
git init
```

Add a .gitignore file to exclude build artifacts:

```
# Compiled object files
*.o

# Executables
*.exe
*.out
```

```
# Build directories
build/
```

Designing the Compiler Architecture

A compiler can be decomposed into five major phases:

1. Lexical Analysis – Converts source code into tokens.
2. Syntax Analysis – Parses tokens into a parse tree or AST.
3. Semantic Analysis – Performs type checking and builds symbol tables.
4. Intermediate Code Generation – Transforms the AST into intermediate representation (IR).
5. Code Generation & Optimization – Generates target machine code and applies optimizations.

For the B.Tech S7 CSE lab, we focus on a subset of a simple language (let's call it **MiniLang**) featuring arithmetic expressions, variable declarations, and control flow constructs (if, while).

Defining the Language Grammar

Below is a concise EBNF representation of MiniLang:

```
program    ::= { declaration | statement } .
declaration ::= "int" IDENTIFIER [ "=" expression ] ";" .
statement  ::= assignment | if_statement | while_statement | compound .
assignment ::= IDENTIFIER "=" expression ";" .
if_statement ::= "if" "(" expression ")" statement [ "else" statement ] .
while_statement ::= "while" "(" expression ")" statement .
compound   ::= "{" { statement } "}" .
expression ::= term { ("+" | "-") term } .
term        ::= factor { ("*" | "/" ) factor } .
factor      ::= NUMBER | IDENTIFIER | "(" expression ")" .
```

Choosing the Right Tools

For this lab, we'll use **Flex** for lexical analysis and **Bison** for parsing. The generated C code will be compiled using GCC. We'll also integrate LLVM for code generation, but you can start with a simple assembly backend if preferred.

Phase 1: Lexical Analysis

Writing the Flex Specification

Create a file named `lexer.l` with the following content:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "parser.tab.h"
%}
```

```
digit [0-9]
```

```

letter [a-zA-Z]
id      {letter}({letter})({digit})*
num     {digit}+

%%

"int"    { return INT; }
"if"     { return IF; }
"else"   { return ELSE; }
"while"  { return WHILE; }
"{"      { return LBRACE; }
"}"      { return RBRACE; }
"("      { return LPAREN; }
")"      { return RPAREN; }
";"      { return SEMICOLON; }
"="      { return ASSIGN; }
"+"      { return PLUS; }
"-"      { return MINUS; }
"*"      { return MULT; }
"/"      { return DIV; }
{id}     { yylval.str = strdup(yytext); return ID; }
{num}    { yylval.num = atoi(yytext); return NUM; }
[ \t\r\n]+ { /* skip whitespace */ }
.        { fprintf(stderr, "Unknown character: %s\n", yytext); exit(1); }

%%

```

Key points:

- The %{ %} section includes C headers and the parser header.
- Each regex pattern returns a token defined in parser.tab.h.
- Identifiers and numbers are stored in the yylval union.

Compiling the Lexer

Run the following command to generate lex.yy.c:

```
flex lexer.l
```

Phase 2: Syntax Analysis

Writing the Bison Grammar

Create a file named parser.y:

```

%{
#include <stdio.h>;
#include <stdlib.h>;
#include <string.h>;

```

```
#include "ast.h"
```

```
void yyerror(const char *s);  
int yylex(void);  
%}
```

```
%union {  
    int num;  
    char* str;  
    struct ASTNode* node;  
}
```

```
%token NUM
```

```
%token ID
```

```
%token INT IF ELSE WHILE LBRACE RBRACE LPAREN RPAREN SEMICOLON ASSIGN PLUS MINUS MULT  
DIV
```

```
%type program declaration statement assignment if_statement while_statement compound expression term factor
```

```
%%
```

```
program:
```

```
    /* empty */ { $$ = create_program_node(NULL); }  
    | program declaration { $$ = add_declaration($1, $2); }  
    | program statement { $$ = add_statement($1, $3); }  
    ;
```

```
declaration:
```

```
    INT ID ASSIGN expression SEMICOLON  
    { $$ = create_declaration_node($2, $4); }  
    | INT ID SEMICOLON  
    { $$ = create_declaration_node($2, NULL); }  
    ;
```

```
statement:
```

```
    assignment  
    | if_statement  
    | while_statement  
    | compound  
    ;
```

```
assignment:
```

```
    ID ASSIGN expression SEMICOLON  
    { $$ = create_assignment_node($1, $3); }
```

;

if_statement:

IF LPAREN expression RPAREN statement ELSE statement

{ \$\$ = create_if_node(\$3, \$5, \$7); }

| IF LPAREN expression RPAREN statement

{ \$\$ = create_if_node(\$3, \$5, NULL); }

;

while_statement:

WHILE LPAREN expression RPAREN statement

{ \$\$ = create_while_node(\$3, \$5); }

;

compound:

LBRACE program RBRACE

{ \$\$ = create_compound_node(\$2); }

;

expression:

term

| expression PLUS term

{ \$\$ = create_binop_node(PLUS, \$1, \$3); }

| expression MINUS term

{ \$\$ = create_binop_node(MINUS, \$1, \$3); }

;

term:

factor

| term MULT factor

{ \$\$ = create_binop_node(MULT, \$1, \$3); }

| term DIV factor

{ \$\$ = create_binop_node(DIV, \$1, \$3); }

;

factor:

NUM

{ \$\$ = create_num_node(\$1); }

| ID

{ \$\$ = create_id_node(\$1); }

| LPAREN expression RPAREN

{ \$\$ = \$2; }

;

%%

Baca Juga (Artikel Terkait)

- [collins cobuild advanced learner s dictionary](http://basicgravity.com/collins-cobuild-advanced-learner-s-dictionary.pdf)

URL: <http://basicgravity.com/collins-cobuild-advanced-learner-s-dictionary.pdf>

- [collins cobuild advanced dictionary cd rom](http://basicgravity.com/collins-cobuild-advanced-dictionary-cd-rom.pdf)

URL: <http://basicgravity.com/collins-cobuild-advanced-dictionary-cd-rom.pdf>

- [collins cobuild advanced dictionary](http://basicgravity.com/collins-cobuild-advanced-dictionary.pdf)

URL: <http://basicgravity.com/collins-cobuild-advanced-dictionary.pdf>

- [collins business grammar and practice intermediate](http://basicgravity.com/collins-business-grammar-and-practice-intermediate.pdf)

URL: <http://basicgravity.com/collins-business-grammar-and-practice-intermediate.pdf>

Tags: [#compiler](#) [#btech](#) [#tutorial](#)

