CS327: Compilers - Lab Assignment #1

Introduction to Language Translation and Tokenization

Course: CS327 Compilers

Instructor: Shouvick Mondal

Semester: January-May 2026

Maximum Marks: 7

Deadline: January 20, 2026, 23:59:59 IST

Submission: Submit here (`A1_<YourRollNumber>.zip.pdf`)

Objective

The primary objective of this assignment is to:

1. Understand the concept of language translation and the compilation pipeline
2. Explore the difference between compilers and interpreters
3. Get hands-on experience with compilation stages (preprocessor, compiler, assembler, linker)
4. Implement a basic tokenizer from scratch to understand how compilers process strings
5. Set up and verify the development environment for the course

Prerequisites

Before starting this assignment, ensure you have:

1. **Knowledge Prerequisites:**
    a. Basic programming knowledge in C

    b. Understanding of command-line operations
    c. Familiarity with text editors
    d. String manipulation techniques
2. **Software Prerequisites:**
    a. Linux/Unix environment (or WSL on Windows)
    b. GCC compiler toolchain installed
    c. Text editor (vim, emacs, VS Code, etc.)
    d. (**very important**) Access to terminal/command line

Important Notes

- **This is an INDIVIDUAL assignment.** Work alone. Do not collaborate or share logic/code.
- **No library functions allowed** for string processing in Part 3 (tokenizer implementation). Only basic language constructs (loops, conditionals, arrays, basic I/O).
- Future assignments (A2, A3, A4) will be in groups of at most three students.

Assignment Description

Part 1: Understanding the Compilation Pipeline

Task 1.1: Create a simple C program and observe each stage of compilation.

Create a file `hello.c`:

```
#include <stdio.h>
#define MESSAGE "Hello from CS327"

int main() {
   printf("%s\n", MESSAGE);
   return 0;
}
```

Your tasks:

1. Run the following commands and save the outputs:

```
cpp hello.c > hello.i              # Preprocessor output
gcc -S hello.i                     # Compiler output (assembly)
as hello.s -o hello.o              # Assembler output (object file)
gcc hello.o -o hello               # Linker output (executable)
```

2. Also try:

```
gcc -v hello.c                     # Verbose compilation
gcc -save-temps hello.c            # Save intermediate files
```

3. Answer these questions in comments within a file `answers_part1.txt`:
    a. What changed in `hello.i` compared to `hello.c`? (List at least 3 observations)
    b. How many lines are in `hello.s`? What do the first 10 lines do?
    c. What is the size difference between `hello.o` and the final executable `hello`?

Deliverable:

- `hello.c` - Source file
- `hello.i`, `hello.s`, `hello.o`, `hello` (or `a.out`) - Intermediate files
- `answers_part1.txt` - Your observations (plain text, 50-100 words total)

Part 2: Compiler Optimization

Task 2.1: Demonstrate optimization in compiled code

Create a C program `optimize.c`:

```
#include <stdio.h>

int main() {
   int x = 0;
   x += 2;
   x += 2;
   --x;
   x += 5;
   ++x;
```

```
    printf("x = %d\n", x);
    return 0;
}
```

Your tasks:

1. Compile without optimization:

```
gcc -O0 -S optimize.c -o optimize_O0.s
```

2. Compile with optimization:

```
gcc -O2 -S optimize.c -o optimize_O2.s
```

3. Count the number of assembly instructions related to the arithmetic operations (x += 2, x += 2, --x, etc.) in both files.
4. Create a file `answers_part2.txt` with:
    a. Number of arithmetic instructions in `optimize_O0.s`: ___
    b. Number of arithmetic instructions in `optimize_O2.s`: ___
    c. What single value did the compiler optimize all operations to? (Hint: slide 24 shows `x += 18`)

Deliverable:

- `optimize.c` - Source code
- `optimize_O0.s` and `optimize_O2.s` - Assembly files
- `answers_part2.txt` - Your answers (plain text)

Part 3: Tokenizer Implementation

Task 3.1: Write a tokenizer in C

IMPORTANT RESTRICTIONS:

- **NO library functions** for string processing allowed
- **C users:** Cannot use `strtok`, `strstr`, `strchr`, `strpbrk`, or any `<string.h>` functions except `strlen`

- You can only use: loops, conditionals, character arrays/strings, basic indexing, basic I/O

Requirements:

Your program `tokenizer.c` should:

1. Read input from standard input (stdin) line by line until EOF
2. For each line, identify and print tokens in the format: `<TOKEN_TYPE, lexeme>`
3. Recognize the following token types:
    a. **KEYWORD**: `int`, `float`, `char`, `if`, `else`, `while`, `return`
    b. **IDENTIFIER**: starts with letter or underscore, followed by letters/digits/underscores
    c. **NUMBER**: sequence of digits (integers only)
    d. **OPERATOR**: +, -, *, /, =
    e. **PUNCTUATION**: (,), {, }, ;, ,
    f. **UNKNOWN**: anything else that doesn't match above categories
4. Ignore whitespace (spaces, tabs, newlines)
5. Process tokens in the order they appear

Example:

Input:

```
int x = 10;
if (x > 5) {
   return x + 20;
}
```

Expected Output:

```
<KEYWORD, int>
<IDENTIFIER, x>
<OPERATOR, =>
<NUMBER, 10>
<PUNCTUATION, ;>
<KEYWORD, if>
<PUNCTUATION, (>
<IDENTIFIER, x>
<UNKNOWN, >>
```

```
<NUMBER, 5>
<PUNCTUATION, )>
<PUNCTUATION, {>
<KEYWORD, return>
<IDENTIFIER, x>
<OPERATOR, +>
<NUMBER, 20>
<PUNCTUATION, ;>
<PUNCTUATION, }>
```

Implementation Tips:

- Read input character by character
- Build tokens by accumulating characters
- When you encounter whitespace or punctuation, that signals end of current token
- Check if accumulated string matches any keyword
- Use helper functions like `isLetter()`, `isDigit()` that you implement yourself

Deliverable:

- `tokenizer.c` - Your tokenizer implementation
- Must compile/run without errors
- Must handle all test cases correctly

Test Cases

Your tokenizer will be evaluated against both public and private test cases.

Public Test Cases (Available to you)

Test Case 1: `public_test1.txt`

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

Expected Output: `public_output1.txt`

```
<KEYWORD, int>
<IDENTIFIER, main>
<PUNCTUATION, (>
<PUNCTUATION, )>
<PUNCTUATION, {>
<KEYWORD, return>
<NUMBER, 0>
<PUNCTUATION, ;>
<PUNCTUATION, }>
```

Test Case 2: public_test2.txt

```
int x = 10 + 20;
float y = 30;
```

Expected Output: public_output2.txt

```
<KEYWORD, int>
<IDENTIFIER, x>
<OPERATOR, =>
<NUMBER, 10>
<OPERATOR, +>
<NUMBER, 20>
<PUNCTUATION, ;>
<KEYWORD, float>
<IDENTIFIER, y>
<OPERATOR, =>
<NUMBER, 30>
<PUNCTUATION, ;>
```

Test Case 3: public_test3.txt

```
if (x > 10) {
   y = x * 2;
}
```

Expected Output: public_output3.txt

```
<KEYWORD, if>
<PUNCTUATION, (>
<IDENTIFIER, x>
<UNKNOWN, >>
<NUMBER, 10>
<PUNCTUATION, )>
<PUNCTUATION, {>
<IDENTIFIER, y>
<OPERATOR, =>
<IDENTIFIER, x>
<OPERATOR, *>
<NUMBER, 2>
<PUNCTUATION, ;>
<PUNCTUATION, }>
```

Private Test Cases (Hidden - for evaluation)

Your tokenizer will also be tested against hidden test cases that include:

- Edge cases (empty lines, multiple spaces, tabs)
- Longer identifiers and numbers
- Mixed valid and unknown tokens
- Various combinations of keywords and operators
- Complex nested expressions

Ensure your tokenizer handles:

- Multiple consecutive spaces/tabs
- Empty lines
- Very long identifiers (e.g., `variable_name_123`)
- Large numbers (e.g., 999999)
- All punctuation marks correctly

Compilation and Execution Instructions

```
# Compile
gcc tokenizer.c -o tokenizer
```

```
# Run with input file
./tokenizer < public_test1.txt

# Or run interactively
./tokenizer
(type input and press Ctrl+D when done)
```

For Parts 1 and 2:

```
# Basic compilation
gcc filename.c -o output

# View preprocessor output
cpp filename.c > filename.i

# Generate assembly
gcc -S filename.c

# Save all intermediate files
gcc -save-temps filename.c

# Verbose compilation
gcc -v filename.c

# Different optimization levels
gcc -O0 filename.c -o output_O0
gcc -O2 filename.c -o output_O2
```

Submission Guidelines

Individual Submission (No Groups):

- This is an **INDIVIDUAL** assignment
- Submit your work independently

What to Submit:

Create a ZIP file named `A1_<YourRollNumber>.zip` (e.g., `A1_21110001.zip`) containing:

```
A1_21110001/
├── Part1/
│   ├── hello.c
│   ├── hello.i
│   ├── hello.s
│   ├── hello.o
│   ├── hello (or a.out)
│   └── answers_part1.txt
├── Part2/
│   ├── optimize.c
│   ├── optimize_O0.s
│   ├── optimize_O2.s
│   └── answers_part2.txt
├── Part3/
│   ├── tokenizer.c
│   ├── public_test1.txt
│   ├── public_output1.txt
│   ├── public_test2.txt
│   ├── public_output2.txt
│   ├── public_test3.txt
│   └── public_output3.txt
└── README.txt
```

README.txt should contain:

```
Name: Your Full Name
Roll Number: Your Roll Number
Email: your.email@iitgn.ac.in
Programming Language Used (Part 3): C
Compilation Instructions:
(Write how to compile and run your tokenizer)

Known Issues/Limitations:
(If any)
```

Submission Link:

Upload to **Submit here** (`A1_<YourRollNumber>.zip.pdf`)

Evaluation Process:

1. Your tokenizer will be compiled/executed using the commands in your README
2. It will be tested against all public test cases (output must match exactly)
3. It will be tested against 5-7 private test cases
4. Partial credit for partially correct output
5. No credit if code doesn't compile/run

Important Notes

1. **Academic Integrity:** Follow IIT Gandhinagar Student Honour Code strictly (https://iitgn.ac.in/students/honourcode). Any form of plagiarism will result in **zero marks** and disciplinary action.
2. **No Extensions:** Deadline is firm at 23:59:59 IST on January 20, 2026.
3. **Testing:** Test your tokenizer thoroughly with various inputs including edge cases.
4. **No Library Functions:** For Part 3, you cannot use built-in string processing functions. Implement everything from scratch using basic constructs.
5. **Output Format:** Your tokenizer output must match the expected format exactly (including spaces, commas, angle brackets).
6. **Resources:**
    a. Course GitHub: https://github.com/SET-IITGN/CS-327-Compilers
    b. Lecture slides

Learning Outcomes

After completing this assignment, you will:

- Understand the complete compilation pipeline from source to executable
- Know the stages: preprocessing, compilation, assembly, linking
- Appreciate how compiler optimizations work
- Have hands-on experience implementing a tokenizer (the first phase of a compiler)
- Understand how compilers process strings and recognize different types of tokens

- Be well-prepared for Assignment #2 which will build upon this foundation

Reference

This assignment is based on concepts from Lecture 1 (January 5, 2026):

- **Slides 23-26:** Compiler vs Interpreter, Compilation Pipeline
- **Slide 20:** Compilers work with strings (tokens, words, sentences)
- **Slide 24:** Compiler optimization example

Good luck with your first compiler assignment!

Remember: This individual assignment builds the foundation for understanding how compilers process source code. Future assignments will be group-based where you'll implement more complex compiler phases.