BCSE307P – Compiler Design Lab (L41 + L42)

Experiment 1 – Phases of a Compiler

Divyanshu Patel (23BAI1214)

Introduction

A compiler is a software system that translates high-level source code into machine-level instructions. Its design is organized into a sequence of well-defined phases, each responsible for ensuring correctness, structure, meaning, and efficiency. These phases work together to transform human-readable code into executable programs.

1. Lexical Analysis (Scanner Phase)

Lexical analysis reads the input program character by character and groups them into **tokens**, which are the smallest meaningful units like keywords, identifiers, operators, constants, and punctuation symbols. It also removes comments and whitespace.

Responsibilities

- Token generation
- Removal of whitespace/comments
- Error detection for illegal characters

Examples

Example 1 – Variables and Literals

C Code:

```
int x = 10;
```

Tokens: int, x, =, 10, ;

Example 2 – Keywords and Operators

```
float y = x + 2.5;
```

Tokens: float, y, =, x, +, 2.5, ;

Example 3 – Strings and Identifiers

```
printf("Hello");
```

Tokens: printf, (, "Hello",), ;

2. Syntax Analysis (Parsing Phase)

The parser takes tokens and checks whether they follow the **grammar** of the language. It builds a **Parse Tree** or **Syntax Tree**, representing the program's structure.

Responsibilities

- Grammar checking
- Construction of parse tree / AST
- Syntax error reporting

Parse Tree Example

Expression: `a + b * c`

```
       (+)
      /   \
    a      (*)
          /   \
         b     c
```

Examples

Example 1 – Valid Statement

```
int x = 10;
```

Grammar accepted.

Example 2 – Invalid Syntax

```
int = x 10;
```

Error: unexpected token '='.

Example 3 – Missing Bracket

```
if (x > 0)
    printf("ok");
```

Valid.

```
if (x > 0
    printf("ok");
```

Error: missing ')'.

3. Semantic Analysis

Semantic analysis verifies whether the program has meaningful and type-correct operations.

Responsibilities

- Type checking
- Scope resolution
- Declaration checking
- Function argument validation

Examples

Example 1 – Type Mismatch

```
int x = "hello";
```

Error: string cannot be assigned to int.

Example 2 – Undeclared Variable

```
x = 5;
```

Error: x is not declared.

Example 3 – Parameter Type Check

```
void f(int a) {}
f("hello");
```

Error: expected int, got string.

4. Intermediate Code Generation (ICG)

The compiler converts the syntax tree into **intermediate representation (IR)** such as TAC (Three Address Code). This representation is machine-independent.

Responsibilities

- Convert expressions to TAC
- Introduce temporary variables
- Preserve semantics

Examples

Example 1 – Arithmetic Expression

C Code:

```
a = b + c * d;
```

IR:

```
t1 = c * d
a = b + t1
```

Example 2 – Conditional

```
if (a > b) c = a;
```

IR:

```
if a > b goto L1
goto L2
L1: c = a
L2:
```

Example 3 – Function Call

```
x = sum(a, b);
```

IR:

```
push a
push b
call sum
x = return_value
```

5. Code Optimization

Optimization improves intermediate code by removing redundancy and improving speed.

Responsibilities

- Remove common subexpressions
- Constant folding
- Dead code elimination
- Loop optimization

Examples

Example 1 – Common Subexpression Elimination

Before:

```
t1 = a + b
t2 = a + b
```

After:

```
t1 = a + b
```

Example 2 – Constant Folding

Before:

```
t1 = 5 * 10
```

After:

```
t1 = 50
```

Example 3 – Dead Code Elimination

Before:

```
x = 10
x = 20
```

After:

```
x = 20
```

6. Target Code Generation

Generates assembly or machine code for the target architecture.

Responsibilities

- Register allocation
- Instruction selection
- Memory layout

Examples

Example 1 – Simple Assignment

C Code:

```
x = y + 1;
```

Assembly:

```
LOAD y, R1
ADD 1, R1
STORE R1, x
```

Example 2 – Multiplication

```
z = a * b;

LOAD a, R1
LOAD b, R2
MUL R1, R2, R3
STORE R3, z
```

Example 3 – Conditional Jump

```
if (x > y) z = 1;

LOAD x, R1
LOAD y, R2
CMP R1, R2
JLE L1
MOV 1, z
L1:
```

Example: Demonstrating All Compiler Phases

C Program

```
int main() {
    int a = 5;
    int b = 10;
    int c = a + b;
    return c;
}
```

1. Lexical Analysis

Tokens:
int, main, (,), {, int, a, =, 5, ;, int, b, =, 10, ;, int, c, =, a, +, b, ;, return, c, ;, }

2. Syntax Analysis

- Valid function definition
- Valid declarations and expressions
- Parse tree created

3. Semantic Analysis

- a, b, c declared before use
- int types compatible
- return type correct

4. Intermediate Code Generation

```
t1 = 5
t2 = 10
t3 = t1 + t2
return t3
```

5. Code Optimization

```
t3 = 15
return t3
```

6. Target Code Generation

```
LOAD 5, R1
LOAD 10, R2
ADD R1, R2, R3
MOV R3, return_register
```