Building a Compiler Step by Step

Mahmoud Hany Sharaby	231017889
Omar Ahmad Mohamad	221007593
Ahmed Osama Elsayed	221011400

INTRODUCTION

The development of a compiler is a complex yet fascinating endeavor that bridges the gap between human-readable code and machine-executable instructions. This report outlines the design and implementation of a compiler tailored to process a while loop construct, leveraging an LL parser for syntax analysis. The project adheres to the foundational phases of compiler construction, ensuring rigorous validation and optimization at each stage.

The primary objective is to transform input code containing variable declarations, expressions, and a while loop into optimized intermediate code. The compiler follows a structured workflow:

1. Context-Free Grammar (CFG) Design: A precise grammar is defined to encapsulate the syntax of the while loop, variable declarations, and arithmetic/logical expressions.

2. Lexical Analysis: This phase tokenizes the input by stripping whitespace, discarding comments, and identifying valid tokens (e.g., keywords, identifiers, operators).

3. Syntax Analysis (LL Parser): As a top-down parsing method, the LL parser validates the token stream against the CFG and generates an Abstract Syntax Tree (AST) in JSON format, reflecting the hierarchical structure of the while loop and associated statements.

4. Semantic Analysis: This phase enforces language rules by verifying variable declaration scopes, ensuring type compatibility in expressions, detecting duplicate variables, and constructing a symbol table.

5. Optimized Intermediate Code Generation: Three-Address Code (TAC) is produced to represent the intermediate instructions, followed by optimization techniques (e.g., constant folding, dead code elimination) to enhance efficiency.

The report elaborates on the challenges and solutions in each phase, emphasizing the LL parser's role in efficiently parsing the while loop's recursive structure. By integrating these components, the compiler achieves a balance between correctness, readability, and performance, demonstrating the practical application of theoretical compiler design principles.

==

Context-Free Grammar

==

```
<S>            → Main(){ <Statements> }
<Statements>   → <Statement> <Statements> | ε
<Statement>    → <Declaration> | <WhileStmt> | <ReturnStmt> | <BreakStmt>
                 | <Expr> SEMICOLON
```

==

Lexical Productions

==

```
<NAME>         → <LETTER> <NAME_TAIL> | UNDERSCORE <NAME_TAIL>
<NAME_TAIL>    → <LETTER> <NAME_TAIL> | <DIGIT> <NAME_TAIL>
                 | UNDERSCORE <NAME_TAIL> | ε

<DIGIT>        → ZERO | <NON_ZERO_DIGIT> <DIGIT_TAIL>
<DIGIT_TAIL>   → <DIGIT> <DIGIT_TAIL> | ε

<REAL>         → <NON_ZERO_DIGIT> <DIGIT_TAIL> DOT <DIGIT> <DIGIT_TAIL>
                 | ZERO DOT <DIGIT> <DIGIT_TAIL>
```

==

Character Classes

==

```
<LETTER>          → A | B | C | D | E | F | G | H | I | J | K | L | M
                    | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
                    | a | b | c | d | e | f | g | h | i | j | k | l | m
                    | n | o | p | q | r | s | t | u | v | w | x | y | z
<NON_ZERO_DIGIT> → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<ZERO>          → 0
<DOT>          → .
<UNDERSCORE>     → _
```

==

Declarations

==

```
<Declaration>     → TYPE NAME SEMICOLON
                    | TYPE NAME ASSIGN_OP <Expr> SEMICOLON
```

=======================================

Full Expression Hierarchy

=======================================

```
<Expr>            → <AssignmentExpr> | <LogicExpr>
<AssignmentExpr>  → NAME ASSIGN_OP <Expr>

<LogicExpr>       → <EqualityExpr> <LogicTail>
<LogicTail>       → LOGIC_OP <EqualityExpr> <LogicTail> | ε

<EqualityExpr>    → <RelationalExpr> <EqualityTail>
<EqualityTail>    → EQUALITY_OP <RelationalExpr> <EqualityTail> | ε

<RelationalExpr>  → <AdditiveExpr> <RelationalTail>
<RelationalTail>  → RELATIONAL_OP <AdditiveExpr> <RelationalTail> | ε

<AdditiveExpr>    → <MultiplicativeExpr> <AdditiveTail>
<AdditiveTail>    → (PLUS | MINUS) <MultiplicativeExpr> <AdditiveTail> | ε

<MultiplicativeExpr> → <PrimaryExpr> <MultiplicativeTail>
<MultiplicativeTail> → (MULTIPLY | DIVIDE | PERCENT) <PrimaryExpr>
<MultiplicativeTail> | ε
```

==

Atomic Elements

==

<PrimaryExpr> → <Literal> | NAME | LPAREN <Expr> RPAREN | <ArrayAccess>
| <FunctionCall>

==

Terminals (Code-Exact Matches)

==

WHILE → while

RETURN → return

BREAK → break

CONTINUE → continue

TYPE → int | float | bool | double | char | string

ASSIGN_OP → = | += | -= | *= | /= | %=

LOGIC_OP → && | || | !

EQUALITY_OP → == | !=

RELATIONAL_OP → < | > | <= | >=

PLUS → +

MINUS → -

MULTIPLY → *

DIVIDE → /

PERCENT → %

LPAREN → (

RPAREN	$\rightarrow$)
LBRACE	$\rightarrow$ {
RBRACE	$\rightarrow$ }
LBRACKET	$\rightarrow$ [
RBRACKET	$\rightarrow$]
SEMICOLON	$\rightarrow$;
COMMA	$\rightarrow$,

The Tokenizer

a key part of the compiler that reads input code and breaks it down into smaller, meaningful pieces called tokens. Think of tokens like the words in a sentence—each has a specific role (e.g., keywords like "while," numbers like "123," or symbols like "+"). Below is a simple breakdown of how the Tokenizer works.

What Does the Tokenizer Do?

1. Reads the Input File:

- The Tokenizer starts by opening the input file (e.g., a code file containing a `while` loop). If the file doesn't exist or has errors, it shows a helpful message.

2. Processes Line by Line:

- It reads the code line by line and scans each character to identify tokens.

- Spaces, tabs, and newlines are used to separate tokens. For example, in `x = 5;`, the tokens are `x`, `=`, `5`, and `;`.

3. Handles Special Characters and Symbols:

- The code checks for symbols like `+`, `-`, `=`, `(` or `{`. Some symbols might form longer tokens (e.g., `==` is treated as one token instead of two `=`).

- For example, if it sees `++`, it groups them into one token instead of treating them as two separate `+` signs.

4. Skips Comments:

- Comments (text ignored by the compiler) are skipped:
  - Single-line comments starting with `//` are ignored.
- Multi-line comments between `/` and `/` are also skipped.

5. Classifies Tokens:

- After splitting the code into tokens, the Tokenizer labels each token based on its type:
  - Keywords: Words like `while`, `int`, or `return`.
  - Numbers: Digits like `42` or `3.14`.
  - Operators: Symbols like `+`, `==`, or `&&`.
  - Punctuation: Brackets `{ }`, parentheses `( )`, or semicolons `;`.
  - Variables: Names like `count` or `result`.

Key Features of the Code

- Error Handling: If the file path is incorrect, it prints clear instructions to check the path, permissions, or file existence.

- Flexibility: It handles complex cases, like numbers with decimal points (e.g., `5.5`) or variable names with underscores (e.g., `total_score`).

- Efficiency: It processes each character only once, making it fast even for large files.

Example: How It Works for a `while` Loop

Imagine the input code has:

```
while (x < 10) {
  x = x + 1;
}
```

The Tokenizer would break this into:

- `while` (keyword)

- `(` (punctuation)

- `x` (variable)

- `<` (operator)

- `10` (number)

- `)` (punctuation)

- `{` (punctuation)

- `x` (variable)

- `=` (operator)

- `x` (variable)

- `+` (operator)

- `1` (number)

- `;` (punctuation)

- `}` (punctuation)

Why This Matters

The Tokenizer is the first step in the compiler. By splitting code into tokens, it prepares the input for the next phases:

1. Syntax Analysis: Checks if tokens follow grammar rules (e.g., `while` loops are structured correctly).

2. Semantic Analysis: Ensures variables are declared and used properly.

3. Code Generation: Converts tokens into machine-readable instructions.

Without a Tokenizer, the compiler couldn't understand the code's structure. This code ensures accuracy and simplicity, laying a strong foundation for the rest of the compiler.

THE LL PARSER CODE

This report explains the LL Parser, a part of the compiler that checks if the code follows grammar rules and builds a structured "tree" (called an Abstract Syntax Tree or AST) to represent the code. Think of the AST like a family tree for code—it shows how parts of the code (like loops or variables) relate to each other. Below is a simple breakdown of how the LL Parser works.

What Does the LL Parser Do?

1. Reads Tokens:

- The parser takes tokens (small code pieces like `while`, `x`, or `=`) generated by the Tokenizer.

- It checks if these tokens are arranged correctly. For example, a `while` loop must have parentheses `()` and curly braces `{}`.

2. Builds the Syntax Tree (AST):

- The parser organizes tokens into a tree structure. For example, a `while` loop becomes a parent node with child nodes for its condition and body.

- The tree is saved in JSON format (a readable way to represent data).

3. Handles Errors:

- If tokens are in the wrong order (e.g., missing `;` or unbalanced `{`), the parser flags errors.

Key Features of the Code

1. Parsing a `while` Loop

Imagine the input code is:

```c
while (x < 10) {
   x = x + 1;
}
```

The parser does the following:

- Step 1: Detects the `while` keyword and starts building a `WHILE` node in the tree.
- Step 2: Checks for `(` and processes the condition (`x < 10`).
- Step 3: Checks for `{`, then parses the loop body (`x = x + 1;`).
- Step 4: Closes the loop with `}`.

The resulting AST looks like this (simplified):

```json
{
  "type": "WHILE",
  "children": [
    { "type": "CONDITION", "value": "x < 10" },
    { "type": "BODY", "value": "x = x + 1" }
  ]
}
```

2. Handling Other Statements

The parser also handles:

- `return`: Marks where a function returns a value.

- `break`/`continue`: Used in loops to stop or skip iterations.

- Expressions: Math operations like `x = 5 + 3`.

3. Grammar Rules (CFG)

The parser follows rules to organize tokens. For example:

- A `while` loop must start with `while (condition) { ... }`.

- Statements end with `;`.

- Variables must be declared before use.

Example: How the Parser Works

Let's break down the code line `x = x + 1;`:

1. The parser sees `x` (variable), `=` (operator), and `x + 1` (expression).

2. It creates an `Expr` node with children:

   - `NAME: x`

   - `ARTH_OP: +`

   - `DIGIT: 1`

3. The tree ensures the equation follows rules (e.g., no missing `;`).

Why This Matters

The LL Parser is the second step in the compiler. It ensures the code's structure is correct before moving to:

1. Semantic Analysis: Checks if variables are used properly (e.g., no undeclared variables).

2. Code Generation: Converts the AST into machine-readable instructions.

Without the parser, the compiler couldn't catch errors like missing braces or invalid loops. This code acts like a "grammar teacher" for the compiler, ensuring the code makes sense.

Section: Annotated AST and Symbol Table

This section explains the Abstract Syntax Tree (AST) and Symbol Table, two critical components of the compiler that help organize and validate code.

Abstract Syntax Tree (AST)

The AST is a tree-like structure that represents the code's logical structure. Think of it as a family tree for your code:

- Nodes: Each part of the code (like a `while` loop or a math operation) becomes a node.

- Hierarchy: Parent nodes (e.g., a `WHILE` loop) connect to child nodes (e.g., the loop's condition and body).

How the AST is Built

The `LLParser` code generates the AST by:

1. Parsing Tokens: It reads tokens (like `while`, `x`, `<`, `10`) and organizes them into nodes.

2. Recursive Rules: Functions like `parseStatement()` and `parseExpr()` define how nodes connect.

Example AST for a `while` Loop

For the code:

```
while (x < 10) {
  x = x + 1;
```

The AST in JSON format would look like this:

```
{
  "type": "WHILE",
  "children": [
    {
      "type": "CONDITION",
      "children": [
        {"type": "NAME", "value": "x"},
        {"type": "REAL_OP", "value": "<"},
        {"type": "DIGIT", "value": "10"}
      ]
    },
    {
      "type": "BODY",
      "children": [
        {
          "type": "ASSIGNMENT",
          "children": [
            {"type": "NAME", "value": "x"},
            {"type": "ARTH_OP", "value": "+"},
            {"type": "DIGIT", "value": "1"}
          ]
        }
      ]
    }
  ]
```

2. Symbol Table

The Symbol Table is like a phonebook for variables. It tracks:

- Variable Names: E.g., `x`, `count`.

- Details: Type (e.g., `int`), scope (where the variable is used), and memory location.

How the Symbol Table Works

Though not fully shown in the code, the semantic analyzer would:

1. Check Declarations: Ensure variables like `x` are declared before use.

2. Detect Errors: Flag issues like duplicate variables or type mismatches (e.g., `x = "hello"` when `x` is a number).

Example Symbol Table

Variable	Type	Scope
x	int	`main` loop

Why These Components Matter

- AST: Makes the code's structure visible and easier to process for later phases (e.g., generating TAC).

- Symbol Table: Ensures variables are used correctly, preventing bugs like typos or type errors.

Connecting to the Code

- The `ASTNode` struct (in `LLParser.h`) defines how nodes store their type, value, and children.

- The `traverse()` function (in `LLParser.cpp`) navigates the AST to generate TAC.

- A real-world compiler would link the Symbol Table to the semantic analyzer to enforce rules like "no undeclared variables."

The AST and Symbol Table are the compiler's "brain." They ensure code is both structured correctly and logically sound. Without them, compilers couldn't catch errors or generate efficient machine code.

THE THREE-ADDRESS CODE (TAC) GENERATOR

This report explains the Three-Address Code (TAC) Generator, a key part of the compiler that converts structured code (like loops and math operations) into simple, step-by-step instructions that a computer can easily execute. The TAC Generator takes the Abstract Syntax Tree (AST)a tree-like representation of the code—and breaks it down into basic operations.

What Does the TAC Generator Do?

The TAC Generator works in two main steps:

1. Converts Expressions into Simple Instructions

- It takes complex expressions (like `x = y + 5  z`) and splits them into smaller steps using temporary variables (`t1`, `t2`, etc.).

- For example:

```
t1 = 5  z

t2 = y + t1

x = t2
```

2. Handles Assignments and Statements

- If the code has a simple assignment (like `x = 10`), it directly generates:

```
x = 10
```

- For `while` loops and other control structures, it generates labeled jumps (not shown in this code, but typically part of TAC).

How the Code Works

1. Generating Temporary Variables

- The code creates temporary variables (`t1`, `t2`, etc.) to store intermediate results.

- Example:

```
string newTemp() {

    return "t" + std::to_string(++tempVarCounter); // Returns "t1", "t2", etc.

}
```

2. Processing Expressions

- The `generateExpr` function breaks down math operations (`+`, `-`, ``, `/`) into smaller steps.

- Example:

  - Input AST for `x = y + 5  z`:

```
{
  "type": "Expr",
  "children": [
    {"type": "NAME", "value": "y"},
    {"type": "ARTH_OP", "value": "+"},
    {"type": "Expr", "children": [...]}
  ]
}
```

  - Generated TAC:

```
t1 = 5  z
t2 = y + t1
x = t2
```

3. Handling Assignments

- The `traverse` function checks for assignment statements (`x = ...`) and generates TAC accordingly.

- Example:

  - Input AST for `x = y + 10`:

~~~
```json
{
  "type": "Statement",
  "children": [
~~~

```
      {"type": "NAME", "value": "x"},
      {"type": "REAL_OP", "value": "="},
      {"type": "Expr", "children": [...]}
    ]
  }
```

- Generated TAC:

```
  t1 = y + 10
  x = t1
```

4. Saving TAC to a File

- The final TAC instructions are written to a file (`tac.txt`).
- Example:

```
 void exportToFile(const string& filename = "tac.txt") {
     ofstream file(filename);
     for (const auto& line : code) {
         file << line << "\n"; // Writes each TAC line
     }
     file.close();
 }
```

Why This Matters?

1. Simplifies Complex Code: Breaks down complicated expressions into easy-to-execute steps.

2. Prepares for Machine Code: The generated TAC can later be converted into actual machine instructions.

3. Enables Optimizations: Since TAC is simple, compilers can optimize it (e.g., removing redundant calculations).

Example: Full TAC Generation

Input Code

x = y + 5 z;

Generated TAC

t1 = 5 z

t2 = y + t1

x = t2

The TAC Generator acts like a "translator" between human-friendly code and computer-friendly instructions. By converting the AST into three-address code, it ensures that the compiler can efficiently process and optimize the program before generating final machine code.

CONCLUSION

This series of reports explains how a simple compiler works by breaking down the process into three main phases: tokenization, parsing, and code generation. Each phase plays a critical role in transforming human-readable code into instructions a computer can execute. Here's a summary of how they all fit together:

1. Tokenizer: Breaking Code into Pieces

- The Tokenizer acts like a "word detector." It reads code line by line and splits it into meaningful units called tokens (e.g., keywords like `while`, numbers like `10`, and symbols like `+`).

- Why it matters: Without tokens, the compiler couldn't understand the basic building blocks of the code.

2. LL Parser: Checking Structure and Building a Tree

- The LL Parser acts like a "grammar checker." It uses tokens to verify that the code follows syntax rules (e.g., `while` loops have parentheses `()` and braces `{}`).

- It builds an Abstract Syntax Tree (AST), a visual map of the code's structure.

- Why it matters: The parser ensures the code is logically organized and catches errors like missing semicolons or mismatched brackets.

3. TAC Generator: Creating Simple Instructions

- The Three-Address Code (TAC) Generator acts like a "translator." It converts the AST into step-by-step instructions using temporary variables (e.g., `t1 = x + 5`).

- Why it matters: TAC simplifies complex code into basic operations, making it easier to optimize and convert into machine code later.

Why This Process Matters

1. Error Detection: Each phase catches specific errors early (e.g., invalid tokens, syntax mistakes).

2. Readability: The AST and TAC provide clear, structured views of the code.

3. Optimization: Simplified TAC allows the compiler to improve performance (e.g., removing redundant calculations).

4. Flexibility: This design works for many programming languages and use cases, from small loops to large programs.

Real-World Example

For a `while` loop like:

```c
while (x < 10) {
    x = x + 1;
}
```

The compiler:

1. Tokenizes it into `while`, `(`, `x`, `<`, `10`, `)`, `{`, `x`, `=`, `x`, `+`, `1`, `;`, `}`.

2. Parses it into an AST to validate the structure.

3. Generates TAC:

```
t1 = x < 10
if_not t1 goto END
x = x + 1
goto START
```

Final Thoughts

This project demonstrates how compilers bridge the gap between human ideas and machine execution. By understanding these phases, we gain insight into the tools that power programming languages and software development. Whether you’re writing a simple loop or a complex application, compilers work behind the scenes to turn your code into action.

What’s Next?

Future work could add machine code generation (converting TAC to binary). For now, this foundation highlights the elegance and practicality of compiler design.