In the name of God

PL_Compiler
Compiler Design Project

Instructor: Dr. Arash Shafiei

Students:

- **Mohammad Amin Saberi, Student ID: 993623026**
- **Abolfazl Shishegar, Student ID: 993623025**

University of Isfahan
Faculty of Computer Engineering

June 2024

June 2024

Table of Contents

Lexer (Phase 1)

Introduction

This program is a lexical analyzer designed for a simplified programming language called PL. It can read PL code, identify its components, and generate tokens.

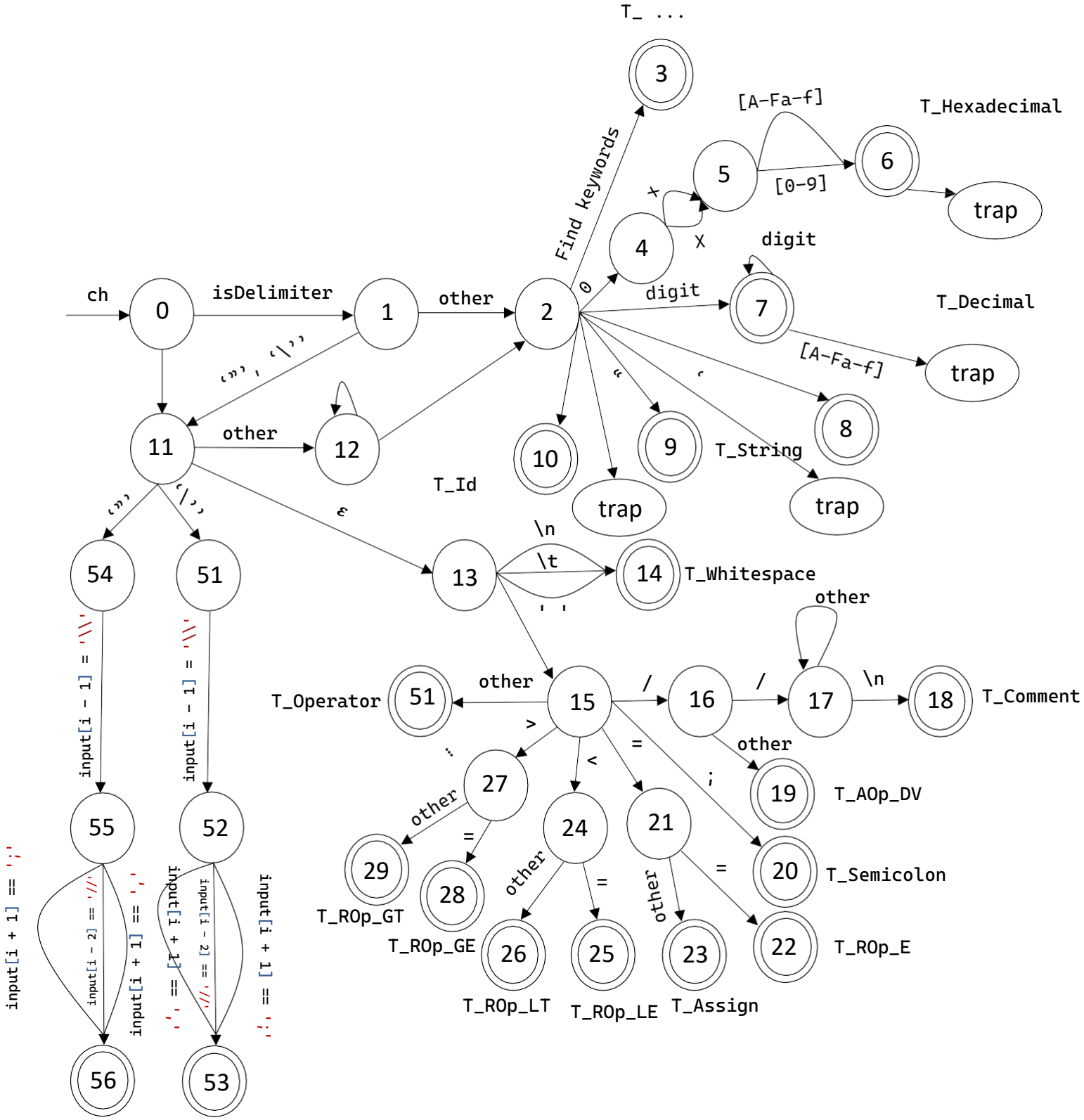
Functionality

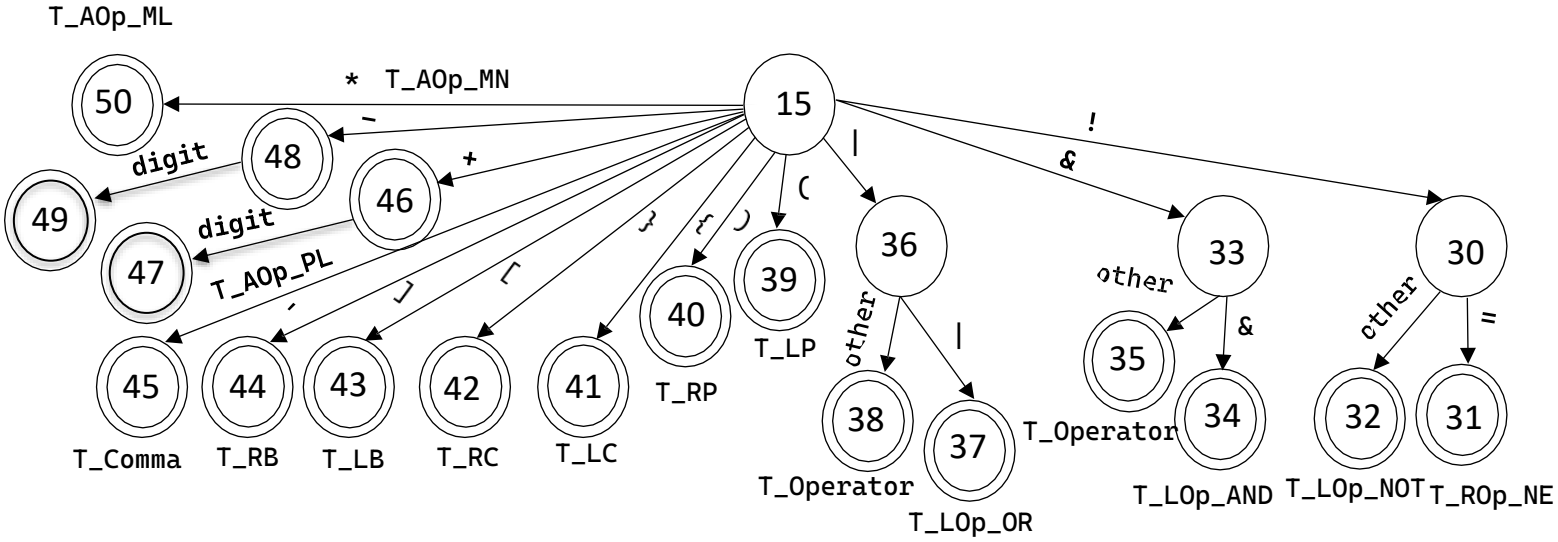
The program processes an input file with the .pl extension. It examines words, symbols, operators, and various structures in the code, identifies them as tokens, and prints information about each token.

Core Codes

1. **Declarations and Token Definitions**
    - Initially, tokens and keywords are declared.
    - The program requests the file path from the user.
2. **File Processing**
    - The program reads the content of the specified file into a string.
3. **Token Analysis**
    - The tokenize function iterates through each character of the file content, analyzes the text, identifies tokens, and prints their information.
4. **Token Identification**
    - During token analysis, keywords, numbers, strings, symbols, and operators are identified and categorized based on their types. Each token is printed accordingly.

State Diagrams




T_AOp_ML
50
* T_AOp_MN
15
!
|
&
48
-
digit
49
46
+
digit
47
T_AOp_PL
45
44
43
42
41
40
39
36
33
30
other
other
other
35
&
38
37
34
32
31
T_Comma
T_RB
T_LB
T_RC
T_LC
T_RP
T_LP
T_Operator
T_LOp_OR
T_Operator
T_LOp_AND
T_LOp_NOT
T_ROp_NE

Regular Expressions

1. **Hexadecimal Numbers**
    - Regex: 0[xX][0-9a-fA-F]+
        - Must start with 0x or 0X, followed by one or more hexadecimal characters.
2. **Decimal Numbers**
    - Regex: -?[0-9]+
        - May start with a negative sign, followed by one or more decimal digits.
3. **Characters**
    - Regex: '[0-9A-Fa-f operators delimiters]*'
        - Identifies characters enclosed in single quotes.
4. **Strings**
    - Regex: "[0-9A-Fa-f operators delimiters]*"
        - Similar to characters but enclosed in double quotes.
5. **Identifiers (IDs)**
    - Regex: _?[a-fA-F]*[0-9a-fA-F]*
        - Identifies variable names starting with an optional underscore, followed by alphanumeric characters.
6. **Operators**
    - Any recognized operator (e.g., ==, <=, ||, &&) is identified and categorized.
7. **Comments**
    - Regex: (//)+[0-9A-Fa-f]*other*\n
        - Identifies comments starting with // and ending at a newline.

Project Overview

1. **Includes and Declarations**
    - The program starts by importing standard libraries and defining essential variables and functions.
2. **File Input and Reading**
    - A file path is taken as input from the user, and the file's content is read for processing.
3. **Tokenization**
    - The tokenize function iterates through the file content, identifies tokens, and processes them accordingly.
4. **Token Types**
    - Keywords, symbols, operators, numbers, and strings are identified.

Tokenize Function

1. **Variable Definitions and Loop Initialization**
    - Essential variables such as the current token string, flags for string or character detection, and token type are initialized.
2. **Main Loop**
    - The function iterates through the file content, processing each character.
3. **Character Prioritization**
    - If the character is a token separator, the current token is printed and reset.
4. **Token Type Identification**
    - If within a string or character, the program processes and identifies it as such.
5. **Loop Termination**
    - The function ends when all characters in the file have been processed and all tokens are printed.

Parser (Phase 2)

Introduction

The parser is a program (parser.cpp) that transforms the source code into a structured representation called an Abstract Syntax Tree (AST). This tree represents the syntactic structure of the code and is used for semantic analysis or code generation.

Core Components

1. **Header Files**
    - Includes standard libraries and the Lexer header file.
2. **Enum NodeType**
    - Defines various node types that can appear in the AST.
3. **Struct Node**
    - Represents a node in the AST, containing its type, value, and children.
4. **Class ASTPrinter**
    - Contains functions to print the AST in a readable and indented format.

Grammar Rules

Grammar defines the syntax of the programming language and specifies the structure of the code. The following rules are provided:

1. **Program**
    - Program → { statement };
2. **Statement**
    - Includes declarations, assignments, conditionals, loops, print statements, and comments.
3. **Expressions**
    - Logical, relational, additive, and multiplicative expressions.

Parsing Tables

1. **First Sets**
    - Examples:
        - FIRST(Program) = { ... }
        - FIRST(statement) = { type, identifier, if, for, ... }
2. **Follow Sets**
    - Examples:
        - FOLLOW(Program) = { $ }
        - FOLLOW(statement) = { type, identifier, if, ... }
3. **LL(1) Parsing Table**
    - Due to the size of the parsing table, it is stored in a separate file named LLdecompositionTable(1) attached within this directory.
4. **Error Recovery Tables**
    - The grammar rules can help identify common errors. For instance:
        - **Assignment Error:** If no = follows an identifier, the error message "Missing '=' in assignment" is displayed.
        - **Block Error:** If a block is not properly closed with }, the error message "Unclosed block" is displayed.
    - Like the parsing table, the error recovery table is included in the separate file.

Function Details

1. **Enum NodeType**
    - Defines the possible types of nodes that can appear in the AST:

```
enum NodeType {
    NODE_PROGRAM,
    NODE_STATEMENT,
    NODE_DECLARATION,
    NODE_ASSIGNMENT,
    NODE_IF_STATEMENT,
    NODE_FOR_LOOP,
    NODE_FUNCTION_DEFINITION,
    NODE_BLOCK,
    NODE_EXPRESSION,
    NODE_LITERAL,
    NODE_IDENTIFIER,
    NODE_PRINT_STATEMENT,
    NODE_UNARY_OP,
    NODE_BINARY_OP,
    NODE_COMMENT,
    NODE_JUMP_STATEMENT
};
```

2. **Struct Node**
    - Represents a single node in the AST, including its type, value, and children:

```
struct Node {
  NodeType type;
  std::string value;
  std::vector<Node> children;
};
```

3. **Class ASTPrinter**
    - A utility class to print the AST nodes with proper indentation:

```
class ASTPrinter {
public:
    void print(const Node &node, int indent = 0) const {
        printIndent(indent);
        std::cout << nodeTypeToString(node.type) << ": " << node.value << "\n";
        for (const Node &child : node.children) {
            print(child, indent + 2);
        }
    }

private:
    void printIndent(int indent) const {
        for (int i = 0; i < indent; ++i) {
            std::cout << ' ';
        }
    }
```

```
    std::string nodeTypeToString(NodeType type) const {
        switch (type) {
            case NODE_PROGRAM: return "Program";
            case NODE_DECLARATION: return "Declaration";
            case NODE_STATEMENT: return "Statement";
            case NODE_EXPRESSION: return "Expression";
            case NODE_IDENTIFIER: return "Identifier";
            case NODE_LITERAL: return "Literal";
            case NODE_PRINT_STATEMENT: return "PrintStatement";
            case NODE_IF_STATEMENT: return "IfStatement";
            case NODE_FOR_LOOP: return "ForLoop";
            case NODE_BLOCK: return "Block";
            case NODE_ASSIGNMENT: return "Assignment";
            case NODE_BINARY_OP: return "BinaryOp";
            case NODE_UNARY_OP: return "UnaryOp";
            case NODE_COMMENT: return "Comment";
            case NODE_JUMP_STATEMENT: return "Jump";
            default: return "Unknown";
        }
    }
};
```

ASTPrinter Class Functions

- **print Function:** Prints a node and its children with proper indentation.
- **printIndent Function:** Creates the necessary spaces for indentation.
- **nodeTypeToString Function:** Converts a node type to its string representation.

Parsing Functions

Each parsing function processes a specific grammar rule and generates a corresponding AST node. Examples include:

1. **Program Parsing (parseProgram)**
    - Processes the entire program and generates a root node of type NODE_PROGRAM.

```
Node parseProgram() {
    Node programNode;
    programNode.type = NODE_PROGRAM;

    while (lexer.hasNext()) {
        if (isDeclaration()) {
            programNode.children.push_back(parseDeclaration());
        } else {
            programNode.children.push_back(parseStatement());
        }
    }
    return programNode;
}
```

2. **Statement Parsing (parseStatement)**
    - Identifies the type of statement and invokes the appropriate parsing function.
3. **Declaration Parsing (parseDeclaration)**
    - Processes variable or function declarations.
4. **Assignment Parsing (parseAssignment)**
    - Handles assignments and validates the syntax.
5. **If Statement Parsing (parseIfStatement)**
    - Parses conditional statements.
6. **For Loop Parsing (parseForLoop)**

- Processes for loop structures.

7. **Print Statement Parsing (parsePrintStatement)**
    - Parses print statements.
8. **Block Parsing (parseBlock)**
    - Processes blocks of code enclosed in {}.
9. **Expression Parsing (parseExpression)**
    - Handles arithmetic or logical expressions.

Semantic (Phase 3)

Introduction

After syntactic analysis (parsing), the next stage in compiling is semantic analysis. This phase ensures that the code adheres to the semantic rules of the programming language, such as correct variable declarations, proper type usage, and function validity.

SymbolTable Class

Manages symbols (variables, functions, etc.) used throughout the program.

Class Variables:

1. table: A vector of unordered_map objects, each representing symbols in a specific scope.
2. currentScope: Tracks the current scope level.

Class Functions:

1. **Constructor (SymbolTable)**
    - Initializes the first scope.
2. **enterScope Function**
    - Creates a new scope.
3. **exitScope Function**
    - Exits the current scope.
4. **insert Function**
    - Adds a new symbol to the current scope.
5. **lookup Function**
    - Searches for a symbol in all scopes, starting from the current scope.

SemanticAnalyzer Class

This class analyzes the AST generated by the parser to ensure semantic correctness.

Class Variables:

1. symbolTable: An instance of the SymbolTable class.
2. Errors: A vector of strings to store semantic errors.

Class Functions:

1. **analyze Function**
    - Starts semantic analysis by verifying the presence of a main function and recursively checks all AST nodes.
2. **checkMainFunction Function**
    - Ensures the program includes a main function.
3. **checkNode Function**
    - Recursively examines each node based on its type.
4. **checkDeclaration Function**
    - Validates variable declarations.
5. **checkAssignment Function**
    - Verifies variable assignments.
6. **checkIfStatement Function**
    - Ensures if conditions are logical and valid.
7. **reportError Function**
    - Logs semantic errors and halts execution if necessary.

TypeChecker Class

This class ensures type compatibility for variables, functions, and operations within the program.

Class Variables:

1. symbolTable: Tracks variable types.
2. functionTable: Tracks function return types and parameters.

Class Functions:

1. **check Function**
    - Initiates type checking for all nodes.
2. **checkDeclaration Function**
    - Validates variable types during declarations.
3. **checkAssignment Function**
    - Ensures the compatibility of variable types during assignments.
4. **checkFunctionCall Function**
    - Verifies function calls against their definitions.
5. **checkBinaryOp Function**
    - Checks compatibility of binary operations (e.g., addition).
6. **getLiteralType Function**
    - Determines the type of a literal value (e.g., int, bool).
7. **inferType Function**
    - Infers the type of an expression or operation.