Lebanese University
Faculty of Science
Section II

Automate lab Report– Info403 Course, M1 Computer Science

Prepared by : Georges SAYDE(59759)

Issa AL ALALI (62153)

Hassan MOUALLEM (62152)

Presented to : Dr. Joseph COSTANTINE

Table of Contents

1. Data Structures

Automaton Structures

```
struct State {     int id;     bool isFinal;
vector<pair<char, int>> transitions; };

struct Automaton {     int id;
vector<State> states;     int
initialState; };
```

These structures implement finite state automata:

- `State`: Represents a single state in the automaton
- `id`: Unique identifier for the state
- `isFinal`: Boolean indicating if it's an accepting state - `transitions`: Vector of pairs containing input symbol and next state

- `Automaton`: Represents the complete finite state machine

- `id`: Unique identifier for the automaton
- `states`: Collection of all states - `initialState`: Starting state of the automaton

Abstract Syntax Tree Structures cpp struct noded {

```
char data[80];     noded* next;
}; struct nodeu {
char
data[80];     int l;     nodeu*
next; };

struct node {     char
label[80];     char p[80];
int n1, n;     noded* d1, * d;
nodeu* und;     node*
child1;     node* child2;
node* child3;     node*
child4; };
```

These structures form the Abstract Syntax Tree (AST):

- `noded`: Linked list node for declarations
- `nodeu`: Symbol table entry with scope level- `node`: Main AST node with:
- `label`: Node type identifier
- `p`: Additional properties
- `und`: Symbol table pointer
- `child1` to `child4`: Pointers to child nodes ### Function Management Structures

```
struct Parameter {     char type[80];     char
name[80]; };

struct Function {     char name[80];     char
returnType[80];     vector<Parameter> parameters;
node* body;     Function* next;     int scopeLevel;
};
```

These handle function definitions and parameters:

- `Parameter`: Stores function parameter information- `Function`: Complete function definition with:
- Name, return type, parameters
- Body as AST node – Scope level tracking

2. Global Variables

vector<Automaton> automatons; char UL[128];
char mot[128]; char* car; char* tokenStart;
node* root, * nd; Function* functionTable;
``` - `automatons`: Collection of all defined finite state machines

- `UL`: Buffer for current lexeme
- `mot`: Input buffer
- `car`: Current position in input- `tokenStart`: Beginning of current token
- `root`: Root of AST- `functionTable`: Symbol table for functions

3. Core Components

Main Menu System void menu() {   //
Implementation handles:
    // 1. Add Automaton
    // 2. Delete Automaton
    // 3. Search Automaton
    // 4. Lexical Analysis
    // 5. Syntax Analysis
    // 6. Semantic Analysis
    // 7. Draw Derivation Tree   // 8. Draw
Automaton by ID
}

Provides user interface for all compiler operations.

4. Lexical Analysis

Token Recognition const char* AL() {   // Main lexical analyzer function   //
Returns token types: "sep", "id", "kw", "number"
}

Implements lexical analysis through:

1. State-based token recognition
2. Keyword identification
3. Identifier validation
```

4. Number parsing
5. Separator handling

Automaton Management

- `addAutomaton()`: Creates new FSM
- `deleteAutomaton()`: Removes existing FSM
- `searchAutomaton()`: Finds FSM by ID- `drawAutomatonByID()`: Visualizes FSM structure

5. Syntax Analysis

Parser Components
```
void AP() {     // Main
parser entry point     // Builds AST from
tokens
}
```

Key parsing functions:

- `program()`: Handles overall program structure
- `declaration()`: Processes variable declarations
- `assignment()`: Handles assignment statements
- `expression()`: Parses expressions- `functionDeclaration()`: Processes function definitions

Control Structures

- `ifStatement()`: Parses if-else constructs
- `whileLoop()`: Handles while loops
- `forLoop()`: Processes for loops

6. Semantic Analysis

Symbol Table Management
```
void eval(node* n) {     // Performs semantic analysis on
AST     // Checks for:
   // - Variable declarations
   // - Scope rules
   // - Type consistency     // - Function
calls }
```

Key features:

1. Scope tracking

2. Variable declaration checking 3. Type verification

4. Function parameter validation

7. Function Handling

```
Function Declaration  void functionDeclaration() {     //
Processes function definitions including:
    // - Return type
    // - Parameters
    // - Function body     // - Scope
creation }
```

```
Function Calls void functionCall() {     // Handles
function invocation:
    // - Argument validation
    // - Parameter counting     // - Scope management
}
`
```

8. Control Structures

Each control structure implements:

1. Syntax validation
2. Scope management
3. AST node creation
4. Symbol table updates

Examples:

- If statements: Condition, then-block, optional else-block
- Loops: Initialization, condition, increment, body- Function calls: Name validation, parameter matching

9. Symbol Table Management

```
void printlist(nodeu* list) {
    // Displays symbol table contents
}
```

1. Variable tracking
2. Scope levels
3. Type information
4. Declaration checking

10. Utility Functions

Tree Operations

```
void traverse(node* n, int depth);
void drawTree();
```

- AST visualization
- Production rule display
- Node relationship showing

Helper Functions

- `copie()`: String copying
- `returns()`: Token rewinding
- `accept()`: Token validation

11. Usage Examples

1. Lexical Analysis:

```
lexicalAnalysis(); // Input: "int x = 5;" // Output: (int, kw)(x, id)(=, sep)(5,
number)(;, sep)
```

2. Syntax Analysis:

```cpp
syntaxAnalysis(); // Validates
program structure
```

// Builds AST

3. Semantic Analysis:

```cpp semanticAnalysis(); //

Checks variable usage

// Validates scopes // Ensures

type safety

```

12. Error Handling

The compiler implements error detection for:

1. Lexical errors (invalid characters)
2. Syntax errors (malformed statements)
3. Semantic errors (undefined variables, type mismatches)
4. Scope violations
5. Function-related errors

Each error type provides specific error messages and appropriate error recovery mechanisms.

13. All Functions :

menu(): Displays a menu for compiler operations and handles user input to call appropriate functions.

addAutomaton(): Adds a new automaton to the list by taking user input for states, transitions, and initial/final states.

deleteAutomaton(): Deletes an automaton from the list based on its ID.

searchAutomaton(): Searches for an automaton by ID and displays its details.

drawAutomatonByID(): Draws the structure of an automaton by its ID, showing states and transitions.

lexicalAnalysis(): Performs lexical analysis on an input string, tokenizing it and displaying the tokens.

syntaxAnalysis(): Performs syntax analysis on an input program, checking for syntactic correctness.

semanticAnalysis(): Performs semantic analysis on the syntax tree, checking for semantic errors and building the symbol table.

drawTree(): Traverses and prints the derivation tree for the parsed program.

AL(): Lexical analyzer function that identifies and returns the next token from the input.

copie(): Copies characters from the input to a buffer for token creation.

returns(): Resets the input pointer to the start of the current token.

AP(): Starts the syntax analysis process by initializing the root node and calling the program parser.

accept(): Ensures the next token matches the expected token and type, throwing an error if not.

program(): Parses the main program structure, including function declarations and the main function.

functionDeclaration(): Parses and validates a function declaration, adding it to the function table.

functionCall(): Parses and validates a function call, ensuring the function exists and arguments match parameters.

parseReturn(): Parses a return statement and its expression, adding it to the syntax tree.

evalFunction(): Evaluates a function’s body, managing scope and parameter handling.

ifStatement(): Parses an if statement, including its condition, then-block, and optional else-block.

forLoop(): Parses a for loop, including initialization, condition, increment, and body.

whileLoop(): Parses a while loop, including its condition and body.

declaration(): Parses a variable declaration, adding it to the symbol table.

assignment(): Parses an assignment statement, ensuring the variable is declared and evaluating the expression.

expression(): Parses an expression, handling operators and operands.

eval(): Recursively evaluates nodes in the syntax tree, performing semantic checks and scope management.

traverse(): Recursively traverses and prints the derivation tree with production rules.

printlist(): Prints the symbol table, showing variables and their scope levels.