Parser Report

Ahmed Badr - 900202868
Youssef Khaled - 900213467

November 11, 2024

Course Information

- Course Name: CSCE4101 - Compiler Design (Fall 2024)
- Instructor: Dr. Ahmed Rafea

1 Objectives

The primary goals of this assignment are:

- To develop a recursive-descent parser for the C- language grammar provided in Appendix A of the course text.
- To handle syntax errors effectively and provide informative error messages.

2 Grammar

The parser is based on the following grammar rules:

1. program → `Program ID { declaration-list statement-list }`
2. declaration-list → declaration-list `var-declaration | var-declaration`
3. var-declaration → `type-specifier ID;` — `type-specifier ID [ NUM ] ;`
4. type-specifier → `int | float`
5. statement-list → `statement-list; statement | empty`
6. statement → `assignment-stmt | compound-stmt | selection-stmt | iteration-stmt`
7. assignment-stmt → `var = expression`
8. var → `ID | ID [ expression ]`
9. compound-stmt → { `statement-list` }
10. selection-stmt → `if ( expression ) statement | if ( expression ) statement else statement`
11. iteration-stmt → `while ( expression ) statement`
12. expression → `additive-expression relop additive-expression | additive-expression`
13. relop → `<= | < | > | >= | == | !=`
14. additive-expression → `additive-expression addop term | term`
15. addop → `+ | -`
16. term → `term mulop factor | factor`
17. mulop → `* | /`
18. factor → `( expression ) | var | NUM`

3 Implementation Details

3.1 Design and Implementation

The parser is implemented using a recursive-descent approach. Each grammar rule is represented by a function, and the parser recursively calls functions according to the production rules. Here are key design choices made during implementation:

- Recursive Structure: Each grammar rule is mapped to a specific function in the parser. This structure simplifies understanding and debugging, as each function is responsible for recognizing a single type of syntactic structure.
- Lookahead Token: A single lookahead token is used to decide which rule to apply next, which aligns with the predictive nature of a recursive-descent parser.
- Error Handling and Reporting: To meet the requirement of error handling, each function checks for the expected token(s). If the lookahead token does not match the expected type, an error is reported, specifying the line number, position, the received token, and the expected token(s). The parser then stops further parsing upon encountering the first syntax error, as error recovery is not required.

3.2 Edge Cases

Several edge cases are handled in this parser, as illustrated by the outputs in `parser_output.txt`:

- Missing Semicolons: One common error case involves missing semicolons after declarations or statements. For example, in 'invalid_missing_semicolon.c', the parser detects the missing semicolon and outputs an error indicating the expected token type (semicolon).
- Unmatched Braces: In cases where there are unmatched braces, such as in 'invalid_missing_brace.c', the parser accurately flags the syntax error at the point where a closing brace is expected but missing.
- Undefined Keywords: In 'invalid_undefined_keyword.c', an unexpected keyword (e.g., 'repeat') triggers an error message, allowing for the detection of invalid identifiers or undeclared keywords.
- Unexpected Token in Expression: For expressions that do not conform to the grammar, such as when encountering an unexpected semicolon in 'invalid_expression.c', the parser immediately flags the error and halts further parsing.
- Missing 'then' Keyword in 'if' Statements: In cases like 'invalid_missing_then.c', where the 'then' keyword is missing in an 'if' statement, the parser outputs an error message detailing the expected token.

4 Running the Parser

To compile and run the scanner and parser, follow these steps:

1. Compile the scanner (assumed to be in 'scanner.l'):

```
flex scanner.l
gcc -o scanner lex.yy.c -lfl
```

2. Compile the parser:

```
gcc -o parser parser.c
```

3. Use the bash script to run the parser on each test case:

```
./run_tests.sh
```

The script `run_tests.sh` performs the following steps:

- Initializes an output file `parser_output.txt`.
- Iterates over each test case file, executes the parser, and appends the output to `parser_output.txt`.

5 Results

An excerpt from the output file, `parser_output.txt`, is shown below, demonstrating successful parsing for valid inputs and detailed error messages for invalid inputs.

5.1 Sample Run Output

```
Starting parser tests...
==========================
Running parser on valid_simple.c...
Parsing completed successfully.
...
Running parser on invalid_missing_semicolon.c...
Syntax Error at line 3, position 6: Expected token type 59 but found 'x'
...
```

6 Conclusion

This report demonstrates the recursive-descent parser for the C- language with error handling and syntax checking. The results of the parser show its ability to parse valid programs successfully and provide meaningful error messages for invalid inputs.