LL(1) Parser Implementation for Syntax Checking

Compiler Construction

Submitted By:

Anoosha Ali – 22i-1242

Ali Usman – 22i-0926

Section: J

Date of Submission: 28th April, 2025

Introduction

In this assignment, we were tasked with implementing an LL(1) parser to parse a sequence of space-separated terminal symbols from an input string. The primary objective was to build a parsing stack and demonstrate the step-by-step parsing process. The parser should also handle error recovery and output meaningful error messages when the input string doesn't conform to the grammar.

The LL(1) parser was based on the context-free grammar (CFG) provided in the assignment. Using the grammar, we generated the FIRST and FOLLOW sets and constructed the LL(1) parsing table to guide the parsing process. The program was designed to read strings from an input file, parse them using the LL(1) parsing table, and show detailed steps involved in parsing.

Approach

1. **Grammar Design**: The grammar for this assignment was simple and designed to handle basic variable declarations and assignments, including expressions and control flow using if-statements.
2. **Generating FIRST and FOLLOW Sets**: Using the given CFG, we calculated the FIRST and FOLLOW sets for all non-terminal symbols. These sets were crucial for constructing the LL(1) parsing table and ensuring correct symbol matching during the parsing process.
3. **LL(1) Parsing Table**: With the FIRST and FOLLOW sets in hand, we constructed the LL(1) parsing table. The table maps each combination of a non-terminal symbol and terminal symbol to a corresponding production rule. This table was used by the parser to expand non-terminals and match terminals.
4. **Parsing Process**: The parser was implemented using a stack to simulate the LL(1) parsing process. The stack is initialized with the start symbol and the end-of-input marker ($). The program iterates over the input symbols and performs the following steps:
    - If the top of the stack is a terminal symbol, it matches the input symbol and advances the input pointer.
    - If the top of the stack is a non-terminal, the program looks up the corresponding production in the LL(1) parsing table and pushes the right-hand side of the production onto the stack.
    - The program displays the stack at each step to illustrate the progress of the parsing.
    - Error handling was implemented to catch syntax errors, such as unexpected tokens, mismatched symbols, and missing entries in the parsing table.
5. **Error Handling**: If the parser encounters an error, such as a terminal mismatch or an empty entry in the parsing table, an appropriate error message is displayed. The program then either skips the error-causing token or stops parsing, depending on the nature of the error. The error messages help users understand what went wrong during parsing.

Challenges Faced

1. **Error Handling**: Implementing error handling was a key challenge. It was important to not only detect errors but also ensure the parser could recover from them and continue parsing the rest of the input when possible. This was handled by skipping over erroneous tokens or stopping parsing when necessary.

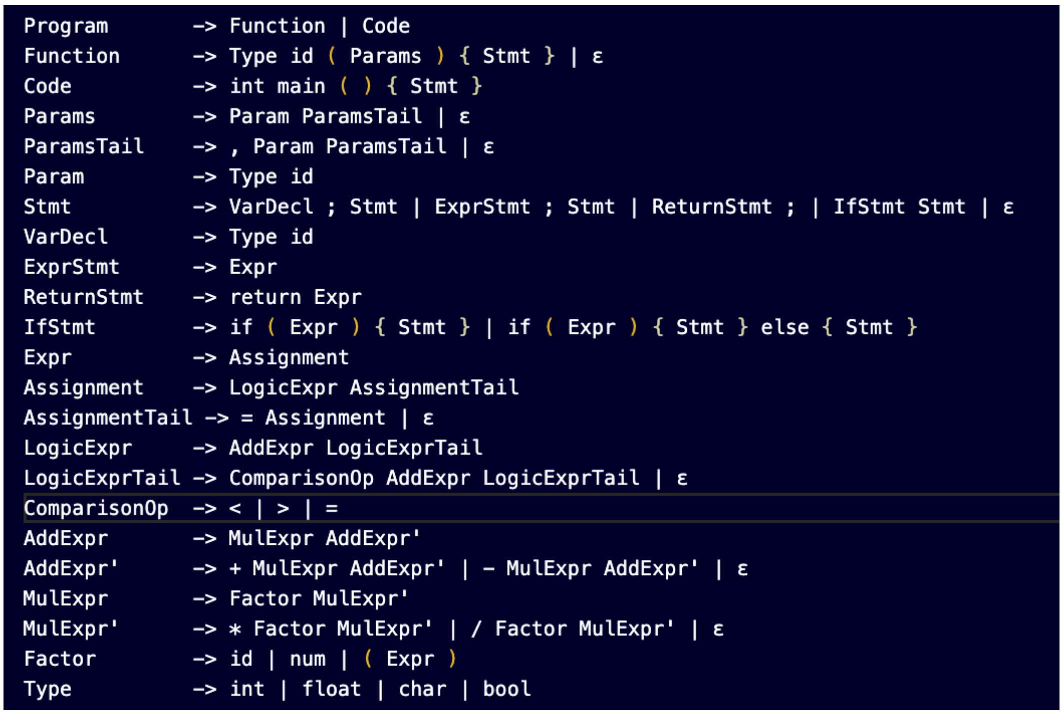
2. **Parsing Stack Implementation**: Managing the stack and ensuring the correct order of operations was another challenge. The stack needed to be updated at every step to reflect the correct state of the parser, and this required careful tracking of the input symbols and the parsing actions.

3. **Generating the Parsing Table**: Constructing the LL(1) parsing table manually required a deep understanding of the grammar and its productions. Ensuring there were no conflicts in the table was critical for the correctness of the parser.

Testing and Verification

The parser was tested with a variety of input strings, including valid and invalid code snippets. The following test case was used:

CFG Used:



```
Program       -> Function | Code
Function      -> Type id ( Params ) { Stmt } | ε
Code          -> int main ( ) { Stmt }
Params        -> Param ParamsTail | ε
ParamsTail    -> , Param ParamsTail | ε
Param         -> Type id
Stmt          -> VarDecl ; Stmt | ExprStmt ; Stmt | ReturnStmt ; | IfStmt Stmt | ε
VarDecl       -> Type id
ExprStmt      -> Expr
ReturnStmt    -> return Expr
IfStmt        -> if ( Expr ) { Stmt } | if ( Expr ) { Stmt } else { Stmt }
Expr          -> Assignment
Assignment    -> LogicExpr AssignmentTail
AssignmentTail -> = Assignment | ε
LogicExpr     -> AddExpr LogicExprTail
LogicExprTail -> ComparisonOp AddExpr LogicExprTail | ε
ComparisonOp  -> < | > | =
AddExpr       -> MulExpr AddExpr'
AddExpr'      -> + MulExpr AddExpr' | - MulExpr AddExpr' | ε
MulExpr       -> Factor MulExpr'
MulExpr'      -> * Factor MulExpr' | / Factor MulExpr' | ε
Factor        -> id | num | ( Expr )
Type          -> int | float | char | bool
```

CFG

Test Case Input:

```
int x;
x = 5 + ;
if (x > 0 {
    x = x - 1;
}
```

Output:

```
Syntax Errors:
Line 3: Syntax Error: Unexpected Token ; after +
Line 4: Syntax Error: Expected ) before {
```

The parser successfully handled the above input, detecting syntax errors and continuing after error recovery. It also printed the stack content at each parsing step, allowing me to trace the progress of the parsing process.

Conclusion

In this assignment, we successfully implemented an LL(1) parser that can read space-separated terminal symbols from an input file, parse the string using the LL(1) parsing table, and display the parsing steps and stack contents. The parser also handles errors gracefully, detecting mismatched symbols and other syntax errors. Overall, the assignment enhanced my understanding of recursive descent parsing and error handling in compilers.