Parser and Lexer Design Report

Cabdifatah Mohamed

October 1, 2024

This report explains the design of a lexical analyzer and a parser for a simple programming language inspired by Fortran (F_{15}), utilizing Flex for lexical analysis and Bison for syntax analysis. This parser recognizes basic language constructs, including program definitions, arithmetic expressions, conditionals, and loops. The program reads input files written in this language and processes them according to the grammar specified in the parser.

1 Lexical Analyzer (Lexer)

The lexical analyzer is responsible for breaking down the input source code into a series of tokens. This is achieved by matching regular expressions for each token type using Flex. The lexer also stores some token values (e.g., integers and strings) in the `yylval` union.

1.1 Token Definitions

We define various tokens for the programming constructs. Here is the list of token types used:

- `PROGRAM, INTEGER, READ, PRINT, IF, THEN, ELSE, DO, END`: Reserved keywords.
- `GT, LT, EQ`: Relational operators (greater than, less than, equal to).
- `PLUS, MINUS, TIMES, DIVIDE`: Arithmetic operators.
- `LPAREN, RPAREN, LBRACKET, RBRACKET`: Parentheses and brackets.
- `COMMA, SEMICOLON, COLON, ASSIGN`: Punctuation symbols.
- `IDENTIFIER, INTEGER, STRING`: Non-keyword tokens such as variable names, integers, and string literals.

1.2 Lexical Rules

Each token type is defined by a corresponding regular expression in the lexer file `lexer.l`. For instance:

- `[0-9]+` matches integer literals and returns the `INTEGER` token while assigning the value to `yylval.intVal`.
- `[a-z][a-z]*` matches identifiers and assigns them to `yylval.strVal`.
- Reserved keywords such as `PROGRAM`, `IF`, etc., are directly matched and returned as their corresponding token.

Comments (starting with `!`) and whitespace are ignored. Unrecognized tokens produce an error message and terminate the program.

2 Parser

The parser is implemented using Bison, which constructs a syntax tree based on the grammar provided. The grammar is designed to recognize arithmetic expressions, conditionals, loops, and variable assignments.

2.1 Grammar Design

The grammar is defined in the file `parser.y` and includes the following constructs:

- `stmt_list`: A sequence of statements separated by semicolons.
- `stmt`: A single statement, such as an assignment (`ID ASSIGNOP expression`).
- `expression`: Arithmetic or relational expressions, composed of terms and factors.
- `simple_expression`: Allows for addition or subtraction of terms.
- `term`: A factor or multiplication/division of factors.
- `factor`: The smallest unit in the expression (identifiers, numbers, or expressions within parentheses).

2.2 Precedence and Associativity

Operators have the following precedence, as declared in the parser:

- `ADDOP` (addition and subtraction) have left associativity.
- `MULOP` (multiplication and division) also have left associativity.
- Relational operators (`RELOP`) are non-associative.

This ensures that multiplication and division have higher precedence than addition and subtraction.

3 Handling of Test Cases

This section outlines how the parser and lexer handle three different test cases, including a valid case, a duplicate case, and an error case. We analyze the tokens generated by the lexer, how the parser processes them, and how errors are handled.

3.1 First Test Case: `test1.f15`

`test1.f15` is a program that calculates the sum of the first `n` natural numbers.

```
PROGRAM
INTEGER n
INTEGER i
INTEGER sum
READ *, n
sum = 0
DO i = 1, n
    sum = sum + i
END DO
PRINT *, "Sum is", sum
END PROGRAM sum
```

This program defines three variables (`n`, `i`, `sum`) and uses a loop (`DO-END DO`) to compute the sum of the first `n` natural numbers. The program also includes input and output operations using `READ` and `PRINT` statements.

The lexer successfully recognizes the tokens for the program structure, such as `PROGRAM`, `INTEGER`, and operators like `+` and `=`. The parser constructs the syntax tree according to the grammar, identifying the `DO-END DO` loop as a repetition statement and handling the assignment of the sum.

The output of the parser for this test case is as follows:

```
KEYWORD:  PROGRAM
KEYWORD:  INTEGER
IDENTIFIER: n
KEYWORD:  INTEGER
IDENTIFIER: i
KEYWORD:  INTEGER
IDENTIFIER: sum
KEYWORD:  READ
IDENTIFIER: n
ASSIGN
INTEGER: 0
KEYWORD:  DO
IDENTIFIER: i
ASSIGN
INTEGER: 1
```

```
IDENTIFIER: sum
PLUS
IDENTIFIER: i
END DO
KEYWORD:  PRINT
STRING: "Sum is"
IDENTIFIER: sum
KEYWORD:  END
KEYWORD:  PROGRAM
```

3.2 Second Test Case: `test2.f15`

The second test case `test2.f15` is essentially the same as the first, designed to verify the consistency of the parser and lexer over repeated inputs.

```
PROGRAM
INTEGER n
INTEGER i
INTEGER sum
READ *, n
sum = 0
DO i = 1, n
    sum = sum + i
END DO
PRINT *, "Sum is", sum
END PROGRAM sum
```

Since the input is identical to `test1.f15`, the output of the parser is the same as described in the first test case. This verifies the consistency of the lexer and parser when processing the same code multiple times.

3.3 Error Test Case: `error_test.f15`

The `error_test.f15` case introduces an error in the program, specifically a missing `END DO` statement in the loop.

```
PROGRAM looperror
INTEGER i
DO i = 1, 10
    PRINT *, i
! Missing END DO here
END PROGRAM looperror
```

In this case, the parser will raise an error due to the missing `END DO`, which is necessary to close the loop. The lexer identifies the tokens correctly, but when the parser expects an `END DO` to complete the loop, it encounters `END PROGRAM` instead. This causes a syntax error.

The output of the parser is as follows:

```
KEYWORD:  PROGRAM
IDENTIFIER: looperror
KEYWORD:  INTEGER
IDENTIFIER: i
KEYWORD:  DO
IDENTIFIER: i
ASSIGN
INTEGER: 1
INTEGER: 10
KEYWORD:  PRINT
STRING: "*"
IDENTIFIER: i
error: syntax error, unexpected END, expecting END DO
```

This output shows the tokens generated up to the point where the error is detected. The error message highlights that the parser expected an `END DO` statement but encountered `END PROGRAM` instead.

4 Makefile

The Makefile automates the compilation process, ensuring that the lexer and parser are generated and linked correctly.

```
# Generate the parser source files from the Bison grammar file (name_A3.y)
bison name_A3.y --defines=name_A3.tab.h -o name_A3.tab.c

# Generate the lexer source file from the Flex specification file (name_A3.l)
flex -o name_A3.yy.c name_A3.l

# Compile and link the lexer, parser, and main C program into the final executable
gcc -o parser name_A3.yy.c name_A3.tab.c name_A3.c -lfl -Werror
```

This Makefile first uses Bison to generate the parser source code and the accompanying header file. Then, Flex is used to generate the lexer source code. Finally, all the components are compiled and linked together to create the executable `parser`.

5 Conclusion

This report outlined the design choices for a simple programming language lexer and parser using Flex and Bison. The lexer efficiently tokenizes the input, while the parser correctly constructs a syntax tree based on the grammar. Test cases demonstrate the parser's ability to handle both valid programs and error cases.