Compiler Project Report

Parsa Haghighatgoo

January 25, 2025

1 Introduction

This document provides the implementation details of a compiler project, including its lexical analyzer and parser. It breaks down the code into logical sections, explaining their functionality to enhance understanding.

2 Lexical Analyzer (Lexer)

The lexical analyzer, implemented in `flex`, is responsible for tokenizing the input into meaningful units.

2.1 Code

Listing 1: Lexer Code

```
%{
#include "parser.tab.h"
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdio.h>
%}

DIGIT    [0-9]
ID       [a-zA-Z_][a-zA-Z_0-9]*

%%

{DIGIT}+ {
    yylval.attributes.value = atoi(yytext);
    yylval.attributes.temp = 0;
    return NUMBER;
}

{ID} {
    yylval.id = strdup(yytext);
    return IDENTIFIER;
}

"+"      { return ADD; }
"-"      { return SUBTRACT; }
```

```
"*"       { return MULTIPLY; }
"/"       { return DIVIDE; }
"="       { return ASSIGN; }
"("       { return LPAREN; }
")"       { return RPAREN; }
";"       { return SEMICOLON; }
[ \t\n]+ {}  // Ignore whitespace
. {
    printf("Unexpected character: %s\n", yytext);
}

%%

int yywrap() {
    return 1;  // Signal end of input
}
```

2.2 Description

- **Headers:** The necessary libraries (`stdio.h`, `stdlib.h`, `string.h`, `parser.tab.h`) are included for input/output and string operations.
- **Definitions:**
  - `DIGIT` defines numeric tokens (e.g., `0-9`).
  - `ID` defines identifier tokens (e.g., variable names).
- **Rules:**
  - Numeric tokens (`DIGIT+`) are converted to integers and assigned as `NUMBER`.
  - Identifier tokens (`ID`) are copied into a string and assigned as `IDENTIFIER`.
  - Operators (`+`, `-`, `*`, `/`) and other symbols (`=`, `();`) are directly returned as tokens.
  - Whitespace is ignored.
  - Unexpected characters are printed with an error message.
- `yywrap()`: Signals the end of input.

3 Parser

The parser, implemented in `bison`, processes the tokens and applies grammar rules to validate the input and generate semantic actions.

3.1 Code

Listing 2: Parser Code

```
%{
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <math.h>

extern int yylex();
extern int yyparse();
extern FILE *yyin;

void yyerror(const char *s);

int temp_counter = 1;
int result;

void print_temp_code(const char *op, int left_temp, int right_temp, int t, int left_val, int right_val) {
    printf("t%d = ", t);
    if (left_temp > 0) printf("t%d ", left_temp);
    else printf("%d ", left_val);

    printf("%s ", op);

    if (right_temp > 0) printf("t%d;\n", right_temp);
    else printf("%d;\n", right_val);
}

int reverse_number(int num) {
    int reversed = 0;
    while (num != 0) {
        reversed = reversed * 10 + num % 10;
        num /= 10;
    }
    return reversed;
}

int is_multiple_of_10(int num) {
    return num % 10 == 0;
}
%}

%union {
    int num;          // Numeric values
    char *id;         // Identifiers
    struct {
        int value;
        int temp;
    } attributes;
}

%token <num> NUMBER
%token <id> IDENTIFIER
%token ADD SUBTRACT MULTIPLY DIVIDE ASSIGN LPAREN RPAREN SEMICOLON

%type <attributes> statement expression term factor

%right ASSIGN
%left ADD SUBTRACT
%left MULTIPLY DIVIDE

%%
```

```
61
62  program:
63      statements
64      ;
65
66  statements:
67      statements statement
68      |
69      ;
70
71  statement:
72      IDENTIFIER ASSIGN expression SEMICOLON {
73          printf("%s = t%d;\n", $1, $3.temp ? $3.temp : $3.value);
74          free($1);
75      }
76      ;
77
78  expression:
79      expression MULTIPLY term {
80          $$ = (typeof($$)){.value = $1.value * $3.value, .temp =
               temp_counter++};
81          print_temp_code("*", $1.temp, $3.temp, $$.temp, $1.value, $3.
               value);
82      }
83      | expression DIVIDE term {
84          $$ = (typeof($$)){.value = $1.value / $3.value, .temp =
               temp_counter++};
85          print_temp_code("/", $1.temp, $3.temp, $$.temp, $1.value, $3.
               value);
86      }
87      | term {
88          $$ = $1;
89      }
90      ;
91
92  term:
93      factor ADD term {
94          $$ = (typeof($$)){.value = $1.value + $3.value, .temp =
               temp_counter++};
95          print_temp_code("+", $1.temp, $3.temp, $$.temp, $1.value, $3.
               value);
96      }
97      | factor SUBTRACT term {
98          $$ = (typeof($$)){.value = $1.value - $3.value, .temp =
               temp_counter++};
99          print_temp_code("-", $1.temp, $3.temp, $$.temp, $1.value, $3.
               value);
100     }
101     | factor {
102         $$ = $1;
103     }
104     ;
105
106 factor:
107     NUMBER {
108         $$ = (typeof($$)){.value = $1, .temp = 0};
109     }
110     | LPAREN expression RPAREN {
```

```
        $$ = $2;
    }
    ;

%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main() {
    printf("Enter an expression:\n");
    if (!yyparse()) {
        printf("Parsing complete.\n");
    }
    return 0;
}
```

3.2 Description

- **Grammar Rules:** Includes rules for arithmetic expressions and assignments.
- **Semantic Actions:** Implements temporary variable generation and specific operations (e.g., reversing numbers).

4 How to Compile and Run

1. Install `flex` and `bison`.
2. Compile the lexer and parser:

```
flex lexer.l
bison -d parser.y
gcc lex.yy.c parser.tab.c -o compiler
```

3. Run the program:

```
./compiler
```

4. Enter arithmetic expressions when prompted.

5 Conclusion

This project demonstrates a functional compiler for arithmetic expressions, showcasing tokenization, grammar parsing, and semantic processing.