Documentation of a Simple Flex/Bison Compiler

Salar Rahnama

January 25, 2025

1 Introduction

This document explains how our small Flex (`lexer.l`) and Bison (`parser.y`) compiler processes an input, such as:

```
a = 30+21/6*14;
```

and produces both:

- Three-address code (TAC)
- The final computed result (e.g., `461`)

We do **not** enforce code blocks to stay on a single page; LaTeX will naturally flow them and may split a long listing across pages.

2 Lexical Analysis — `lexer.l`

Listing 1: lexer.l – Lexical Analyzer

```
%option noyywrap

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

```
#include "parser.tab.h"

/* 'extern' reference to reverseNumber defined in parser.y */
extern int reverseNumber(int num);
%}

/* Rules section */
%%
[ \t\r\n]+ { /* skip whitespace */ }
[a-zA-Z_][a-zA-Z0-9_]* {
    yylval.sval = strdup(yytext);
    return ID;
}
"=" { return ASSIGN; }
";" { return SEMICOL; }
"(" { return LPAREN; }
")" { return RPAREN; }
"+" { return PLUS; }
"-" { return MINUS; }
"*" { return MUL; }
"/" { return DIV; }

[0-9]+ {
    int val = atoi(yytext);
    /* Reverse digits if not multiple of 10 */
    if (val % 10 != 0) {
        val = reverseNumber(val);
    }
    yylval.ival = val;
    return NUM;
}

. {
    fprintf(stderr, "Lexical Error: unknown token '%s'\n", yytext);
    exit(1);
}
%%
```

Explanation (lexer.l)

- `[0-9]+`: Matches an integer. If it is *not* a multiple of 10, we call `reverseNumber`.
- Identifiers, operators, parentheses, semicolons, etc. are matched and returned as appropriate tokens (`ID`, `ASSIGN`, `PLUS`, etc.).
- Any unrecognized character leads to a lexical error message.

3 Syntax Analysis and Code Generation — `parser.y`

Listing 2: parser.y – Bison Grammar and Intermediate Code Generation

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

/* Declare yylex to avoid warnings */
int yylex(void);

/* Error function for Bison */
void yyerror(const char *s) {
    fprintf(stderr, "Parse␣Error:␣%s\n", s);
}

/* reverseNumber is defined here, so lexer can call extern. */
int reverseNumber(int num) {
    int rev = 0;
    int sign = (num < 0) ? -1 : 1;
    num = abs(num);
    while (num != 0) {
        rev = rev * 10 + (num % 10);
        num /= 10;
    }
    return rev * sign;
}

```

```
/* Generate fresh temp variables (t0, t1, etc.) */
static int tempCount = 0;
char* newTemp() {
    char buf[16];
    sprintf(buf, "t%d", tempCount++);
    return strdup(buf);
}
%}

%union {
    int ival;
    char* sval;
    struct {
        int val;
        char* addr;
    } info;
}

%token <sval> ID
%token <ival> NUM
%token ASSIGN SEMICOL LPAREN RPAREN PLUS MINUS MUL DIV

%left MUL DIV /* lower precedence */
%right PLUS MINUS /* higher precedence */

%type <info> E
%start S

%%
S : ID ASSIGN E SEMICOL
    {
      printf("\n--- Three-Address Code ---\n");
      printf("%s = %s;\n", $1, $3.addr);
      printf("Result: %d\n", $3.val);
    }
  ;

/* Expression rules with reversed precedence */
E : E PLUS E
    {
```

```
67      int r = $1.val + $3.val;
68      if (r % 10 != 0) r = reverseNumber(r);
69      char* t = newTemp();
70      printf("%s␣=␣%s+%s;\n", t, $1.addr, $3.addr);
71      $$.val = r;
72      $$.addr = t;
73    }
74  | E MINUS E
75    {
76      int r = $1.val - $3.val;
77      if (r % 10 != 0) r = reverseNumber(r);
78      char* t = newTemp();
79      printf("%s␣=␣%s-%s;\n", t, $1.addr, $3.addr);
80      $$.val = r;
81      $$.addr = t;
82    }
83  | E MUL E
84    {
85      int r = $1.val * $3.val;
86      if (r % 10 != 0) r = reverseNumber(r);
87      char* t = newTemp();
88      printf("%s␣=␣%s*%s;\n", t, $1.addr, $3.addr);
89      $$.val = r;
90      $$.addr = t;
91    }
92  | E DIV E
93    {
94      if ($3.val == 0) {
95          fprintf(stderr, "Error:␣divide␣by␣zero!\n");
96          exit(1);
97      }
98      int r = $1.val / $3.val;
99      if (r % 10 != 0) r = reverseNumber(r);
100     char* t = newTemp();
101     printf("%s␣=␣%s/%s;\n", t, $1.addr, $3.addr);
102     $$.val = r;
103     $$.addr = t;
104   }
105 | LPAREN E RPAREN
106   {
```

```
      $$.val = $2.val;
      $$.addr = $2.addr;
    }
  | NUM
    {
      char* t = newTemp();
      printf("%s = %d;\n", t, $1);
      $$.val = $1;
      $$.addr = t;
    }
  ;
%%
int main(void) {
    printf("Enter something like: a = 30+21/6*14;\n");
    return yyparse();
}
```

Explanation (parser.y)

- `%left MUL DIV` (lowest precedence) and `%right PLUS MINUS` (highest precedence) inverts normal arithmetic precedence, so `+`/`-` parse after `*`/`/`.
- Each rule like `E PLUS E` or `E MUL E` prints a line of three-address code and updates `$$.val` (the computed integer result).
- If a partial result is not a multiple of 10, we apply `reverseNumber(...)`.
- `S : ID ASSIGN E SEMICOL` handles the final assignment and prints the final numeric result.

4 Example Input: `a = 30+21/6*14;`

Lexing

1. `"a"` → `ID`.
2. `"="` → `ASSIGN`.

3. "30" is a multiple of 10, so stays 30; → NUM(30).
4. "+" → PLUS.
5. "21" reversed to 12; → NUM(12).
6. "/" → DIV.
7. "6" reversed is 6; → NUM(6).
8. "*" → MUL.
9. "14" reversed to 41; → NUM(41).
10. ";" → SEMICOL.

Parsing and TAC Output

- 30 + 12 = 42 → reversed to 24
- 24 / 6 = 4
- 4 * 41 = 164 → reversed to 461
- Final result = 461

Therefore, the compiler prints lines like:

```
t0 = 30;
t1 = 12;
t2 = t0+t1; // 42 => 24
t3 = t2/6;  // 4
t4 = 41;
t5 = t3*t4; // 164 => 461
a = t5;
Result: 461
```

5 Conclusion

By using:

- **Flex** (`lexer.l`): to tokenize the input and apply partial digit-reversal on integers,
- **Bison** (`parser.y`): to parse with reversed operator precedence and generate three-address code,

we construct a simple educational compiler that both *generates* the intermediate code and *calculates* the final integer. The code above may split across pages naturally, so the code listings can flow without forcing them onto a single page.