编译原理大作业实验报告

序章

项目简介

本项目基于flex、yacc、llvm和c++，实现了自定义语言`C-minus`的简易编译器，可以生成抽象语法树、中间代码、目标代码以及可执行文件。

小组分工

周承扬： lex + yacc +报告撰写

蒋昀昊：ast + code_gen

彭博：可视化 + 测试

第一章 词法分析

本实验的词法分析使用Flex完成，FLex读入.lex文件中定义的词法分析规则，输出C语言词法分析器源码。

1.1 具体实现

标准的lex文件由三部分组成，分为定义区、规则区和用户子过程区。实验中的`lex.l`代码如下：

1.1.1 定义区

`lex.l` 的定义区包括需要引入的头文件、某些选项设置以及语言关键字以及标识符，具体如下：

头文件：

```
%{
    #include<string.h>
    #include"ast.h"
    #include"grammar.hpp"
%}
```

设置：

```
%option yylineno    //用于输出错误信息（行数）
```

语言标识符。lex使用的是正则表达式匹配：

```
not "!"|"~"

if "if"
else "else"
while "while"
return "return"

type "int"|"float"|"char"|"boolean"

integer 0|[1-9][0-9]*
float {integer}\.[0-9]+
string \"(\\.|[^"\\])*\"
char \'.\'
bool "true"|"false"|"True"|"False"

a_c_op "+="|"-="|"*="|"/="|"%="

id [a-zA-Z_][a-zA-Z0-9_]*

relop ">"|"<"|">="|"<="|"=="|"!="
space [ \t]+
enter [\n]

commits \/\*([^\*]|(\*)*[^\*/])*(\*)*\*\/
commit  \/\/[^\n]*

else_sym .
```

1.1.2 规则区

规则区规定了如上操作符对应的语义动作。代码如下：

```
{not}       {yylval.node = new Node(yytext, "NOT", 0);
```

```
return NOT;}
{type}      {yylval.node = new Node(yytext, "TYPE", 0);
return TYPE;}

{char}      {yylval.node = new Node(yytext, "CHAR", 0);
return CHAR;}
{bool}      {yylval.node = new Node(yytext, "BOOLEAN",
0); return BOOLEAN;}
{float}     {yylval.node = new Node(yytext, "FLOAT", 0);
return FLOAT;}
{string}    {yylval.node = new Node(yytext, "STRING", 0);
return STRING;}
{integer}   {yylval.node = new Node(yytext, "INTEGER",
0); return INTEGER; }

{if}        {yylval.node = new Node(yytext, "IF", 0);
return IF;}
{else}      {yylval.node = new Node(yytext, "ELSE", 0);
return ELSE;}
{while}     {yylval.node = new Node(yytext, "WHILE", 0);
return WHILE;}
{return}    {yylval.node = new Node(yytext, "RETURN", 0);
return RETURN;}

{id}        {yylval.node = new Node(yytext, "ID", 0);
return ID;}

{relop}     {yylval.node = new Node(yytext, "RELOP", 0);
return RELOP;}
{a_c_op}    {yylval.node = new Node(yytext, "ACOP", 0);
return ACOP;}

{space}     {}
{enter}     {}
{commit}    {}
{commits}   {}

"=" {yylval.node = new Node(yytext, "ASSIGNOP", 0);
return ASSIGNOP;}
"+" {yylval.node = new Node(yytext, "PLUS", 0); return
PLUS;}
"-" {yylval.node = new Node(yytext, "MINUS", 0); return
MINUS;}
"*" {yylval.node = new Node(yytext, "STAR", 0); return
STAR;}
"/" {yylval.node = new Node(yytext, "DIV", 0); return
DIV;}

"%" {yylval.node = new Node(yytext, "MODULO", 0); return
```

```
MODULO;}

"||" {yylval.node = new Node(yytext, "OR", 0); return
OR;}
"&&" {yylval.node = new Node(yytext, "AND", 0); return
AND;}

";" {yylval.node = new Node(yytext, "SEMI", 0); return
SEMI;}
//Left Bracket
"(" {yylval.node = new Node(yytext, "LBRACKET", 0);
return LBRACKET;}
")" {yylval.node = new Node(yytext, "RBRACKET", 0);
return RBRACKET;}
//Left Square Bracket
"[" {yylval.node = new Node(yytext, "LSB", 0); return
LSB;}
"]" {yylval.node = new Node(yytext, "RSB", 0); return
RSB;}
"," {yylval.node = new Node(yytext, "COMMA", 0); return
COMMA;}
//Left Brace
"{" {yylval.node = new Node(yytext, "LBRACE", 0); return
LBRACE;}
"}" {yylval.node = new Node(yytext, "RBRACE", 0); return
RBRACE;}

{else_sym}  {cout << "sym error" << endl;}
```

1.1.3 用户子过程区

定义了yywrap()函数:

```
int yywrap() {
    return 1;
}
```

第二章 语法分析

本实验的语法分析使用yacc和cpp实现，具体为`grammar.y`和`ast.cpp`文件的一部分。

`grammar.y`包括**定义段**与**规则段**。定义段包括**C语言部分**的申明与定义和文法中**终结符**的定义。规则段定义了文法中所有的的**非终结符**以及**产生式**。

`ast.cpp` 包括定义树结点的成员变量、构造函数、获取类型、赋类型函数

2.1 yacc

2.1.1 定义段

C语言部分的申明与定义如下：

```
%{
    #include<stdio.h>
    #include"ast.h"
    #include"type.h"
    extern int yylineno;
    extern char* yytext;
    extern Node *ROOT;

    extern int yylex();

    void yyerror(const char* msg) {
        printf("Error: %s Line:%d String:%s\n", msg,
yylineno, yytext);
    }
%}
```

终结符申明定义部分包括：

- 符号类型申明：

```
%union{
    struct Node* node;
}
```

- 终结符申明：

```
%token <node> INTEGER FLOAT CHAR STRING BOOLEAN
RETURN
%token <node> LBRACKET RBRACKET LSB RSB LBRACE
RBRACE
%token <node> TYPE ID COMMA WHILE IF ELSE SEMI
%token <node> ASSIGNOP ACOP
%token <node> RELOP
%token <node> AND OR NOT
%token <node> PLUS MINUS
%token <node> STAR DIV MODULO
```

- 操作符结合性与优先级申明：（从上到下操作符的优先级递增）

```
%left COMMA
%right ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV MODULO
%right NOT
%left LBRACKET RBRACKET LSB RSB
```

- 非终结符申明：

```
%type <node> program  declaration_funct_list
 declaration_funct  declaration  funct
%type <node> var_declaration  var_list  var
%type <node> funct_declaration  para_list  para
%type <node> comp_statement  statements
 statement  ret_statement  expr_statement
 loop_statement  select_statement
%type <node> arithmetic_statement
 declaration_statement  assignment  funct_call
%type <node> elem  args
```

2.1.2 规则段

规则段定义了所有非终结符以及产生式，并描述了产生式的语义动作（也就是建立抽象语法树的结点），例如如下语句：

```
program:
    decl_funct_list {
        $$ = new Node("", "program", 1, $1);
        ROOT = $$;
    }
    ;
```

建立结点program，并将它赋给全局变量ROOT（node类型的指针）

其余文法的实现，由于篇幅过长，详见附录或/src/grammar.y

2.2 结点定义

ast.h中定义了抽象语法树的结点。结点的成员变量如下：

```
class Node {
```

```
public:
  // Value or name of node, if type of node is int, the
value of nodeName is the
  // value of the integer, float, bool, char are similar
if type is var, the
  // value is the name of this variable
  string *node_name;
  // The type of the node
  string *node_type;
  // The type of exp, var or const
  int value_type;
  // The number of child of the node
  int child_num;
  // Child nodes of this node
  Node **children;
  // The number of rows of the node in the file
  int line_num;
};
```

结点的成员函数包括如下几类：

- 构造与析构函数
- 成员变量操作函数（赋值、查询成员变量）
- 语义分析函数（建立抽象语法树）
- json生成函数

其中与语法分析有关的是第一与第二部分，如下：

```
class Node {
public:
    //构造终结符结点
    Node(char * node_name, string node_type, int
line_num);
    //构造非终结符结点
    Node(string node_name, string node_type, int
child_num, ...);
    //析构函数
    ~Node();
};
```

第三章 语义分析

3.1 LLVM

本项目使用LLVM生成自定义语言c--的中间代码**LLVM-IR**。LLVM是一个用于建立编译器的基础框架，以C++编写。创建此工程的目的是对于任意的编程语言，利用该基础框架，构建一个包括编译时、链接时、执行时等的语言执行器。

LLVM-IR是LLVM的中间代码，可以转换为汇编代码和可执行文件。每一个IR文件是一个module，包含了IR对象中的所有信息（符号表、符号等）。

3.2 实际实现

LLVM为我们封装好了所有的功能函数，因此语义分析的主要任务是：在抽象语法树上根据不同的语义情况，调用llvm-api生成对应的中间代码。在实际实现中，我们采用自顶向下的策略生成IR，函数如下：

```
class Node {
public:
  llvm::Value *ir_build();
  llvm::Value *ir_build_declaration();
  llvm::Value *ir_build_funct();
  llvm::Value *ir_build_statement();
  llvm::Value *ir_build_comp_statement();
  llvm::Value *ir_build_exp_statement();
  llvm::Value *ir_build_funct_call();
  llvm::Value *ir_build_println();
  llvm::Value *ir_build_printf();
  llvm::Value *ir_build_scan();
  llvm::Value *ir_build_addr();
  llvm::Value *ir_build_arithmetic_statement();
  llvm::Value *ir_build_elem();
  llvm::Value *ir_build_relop();
  llvm::Value *ir_build_declaration_statement();
  llvm::Value *ir_build_assignment();
  llvm::Value *ir_build_loop_statement();
  llvm::Value *ir_build_select_statement();
  llvm::Value *ir_build_ret_statement();
};
```

如上的函数申明中，使用缩进来大致表明各个函数的“层级关系”。下面对ir_build作为示例进行介绍。

3.2.1 ir_build

`ir_build` 函数是分析整个抽象语法树的入口，对应于“program”结点，并根据结点的子结点情况判断执行`funct_build`、`decl_build`还是`ir_build`。最终会执行所有的结点。

3.2.1.1 ir_build_declaration

`ir_build_declaration`函数用于分析**单句变量申明**：首先获得变量的类型，接着获取变量名并判断变量是否为数组类型。确定信息后根据变量所在函数栈，将其插入对应位置

3.2.1.2 ir_build_funct

`ir_build_funct`函数用于处理函数实现，返回建立的函数类型的Value变量。处理函数包括如下几个过程：

1. 获取函数基本信息：返回类型、函数名、参数表等
2. 建立函数
3. 创建基本块
4. 设置函数参数
5. 执行函数体（comp_stmt）
6. 函数出栈

3.2.2 ir_build_statement

`ir_build_statement`用于分析单条语句，返回Value类型变量。语句包括如下类型：

- 复合语句（comp_stmt）
- 表达式语句（exp_stmt）
- 循环语句（loop_stmt）
- 选择语句（select_stmt）
- 返回语句（ret_stmt）

3.2.2.1 ir_build_comp_statement

`ir_build_comp_statement`包括大括号以及语句表，对于语句表中的每单条语句，执行`ir_build_statement`函数

3.2.2.1 ir_build_exp_statement

`ir_build_exp_statement`用以分析表达式语句，表达式有如下几种类型：

1. 申明语句（decl_stmt）
2. 运算语句（arit_stmt）
3. 赋值语句（assignment）
4. 函数调用（funct_call）

根据不同的语句类型，执行不同的`ir_build`函数：

- 申明语句：申明语句根据申明时是否赋予初值分为两类。

- 如不赋初值则调用之前实现的`ir_build_declaration`函数。
- 如需赋值，则需要先获得运算式的变量值，之后将该变量值。具体为：获取左值（等号的左值，下同）的地址，将右值的运算返回结果赋给它。

3.2.2.1 ir_build_loop_statement

`ir_build_loop_statement` 用以分析循环语句。本实验实现的语言支持while语句。处理while语句的过程大致如下：

1. 获取当前函数栈信息
2. 创建基本块
3. 执行语句（对stmt结点执行ir_build）
4. 返回分支类型的Vaule变量

3.2.2.1 ir_build_select_statement

`ir_build_select_statement` 用以分析选择语句，本实验实现的语言支持if-else语句。处理过程大致如下：

1. 获取条件判断值（arit_stmt的返回值）
2. 创建块
3. 根据条件判断的值，选择执行块，创建执行分支指令

返回执行分支指令类型的Vaule

3.2.2.1 ir_build_ret_statement

`ir_build_ret_statement` 较为简单，根据返回值（arit_stmt类型的变量）创建返回指令，并将返回指令类型的Vaule返回

第四章 代码生成

由于在语义分析中，我们已经自顶向下的分析了抽象语法树，并且完善了IR-module，因此我们已经得到了程序对应的完整的中间代码。

如果要进一步得到可执行程序，我们可以使用llvm-as指令生成.bc文件，之后使用clang将.bc文件编译为可执行文件即可

第五章 测试案例

简单语句的测试暂略。

5.1 快速排序测试

快速排序程序主体如下：

```
//quick sort [l, r]
int qsort(int l, int r) {
    if(l >= r)
        return 0;
    int i = l;
    int j = r;
    int p = a[i];
    while(i < j) {
        while(i < j && a[j] > p)
            j -= 1;
        if(i < j) {
            a[i] = a[j];
            i += 1;
        }
        while(i < j && a[i] < p)
            i += 1;
        if(i < j) {
            a[j] = a[i];
            j -= 1;
        }
    }
    a[i] = p;
    qsort(l, i - 1);
    qsort(i + 1, r);
    return 0;
}
```

运行结果如下：

```
pb@DESKTOP-375LM8H:/mnt/c/Users/pb/Desktop/c-compiler-master/test/test_1$ ./test ./test.out
fixed case 0 (size 0)...pass!
fixed case 1 (size 1)...pass!
fixed case 2 (size 2)...pass!
fixed case 3 (size 2)...pass!
fixed case 4 (size 3)...pass!
fixed case 5 (size 3)...pass!
fixed case 6 (size 3)...pass!
fixed case 7 (size 3)...pass!
fixed case 8 (size 3)...pass!
fixed case 9 (size 4)...pass!
fixed case 10 (size 9)...pass!
fixed case 11 (size 9)...pass!
fixed case 12 (size 10000)...pass!
fixed case 13 (size 10000)...pass!
fixed case 14 (size 4096)...pass!
randomly generated case 0 (size 10000)...pass!
randomly generated case 1 (size 10000)...pass!
randomly generated case 2 (size 10000)...pass!
randomly generated case 3 (size 10000)...pass!
randomly generated case 4 (size 10000)...pass!
randomly generated case 5 (size 10000)...pass!
randomly generated case 6 (size 10000)...pass!
randomly generated case 7 (size 10000)...pass!
randomly generated case 8 (size 10000)...pass!
randomly generated case 9 (size 10000)...pass!
---------------------------------------
2022-06-12 19:27:41.517
```

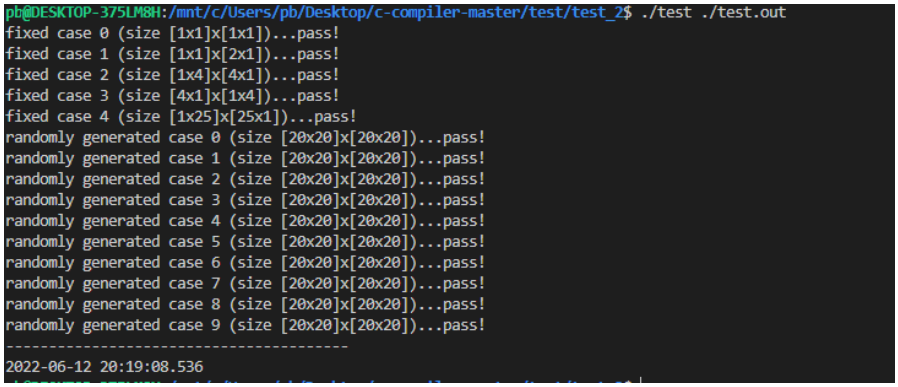
5.2 矩阵乘法测试

矩阵乘法函数主体如下：

```
int A[1001];
int B[1001];
int C[1001];

int main()
{
    int p = 0;
    int q = 0;
    int r = 0;
    int a = 0;
    int i = 0;
    int j = 0;
    int k = 0;

    scan(p, q);
    while(i < p) {
        j = 0;
        while(j < q) {
            scan(A[i * q + j]);
            j += 1;
        }
        i += 1;
    }
    i = 0;
    scan(a, r);
    while(i < a) {

        j = 0;
```

```
        while(j < r) {
            scan(B[i * r + j]);
            j += 1;
        }
        i += 1;
    }

    if(a != q) {
        println("Incompatible Dimensions");
        return 0;
    }

    i = 0;
    while(i < p) {
        j = 0;
        while(j < r) {
            k = 0;
            while(k < q) {
                C[i * r + j] += A[i * q + k] * B[k * r +
j];
                k += 1;
            }
            j += 1;
        }
        i += 1;
    }

    i = 0;
    j = 0;
    while(i < p) {
        j = 0;
        while(j < r) {
            printf("%10d", C[i * r + j]);
            j += 1;
        }
        i += 1;
        println("");
    }

    return 0;
}
```

运行结果如下：



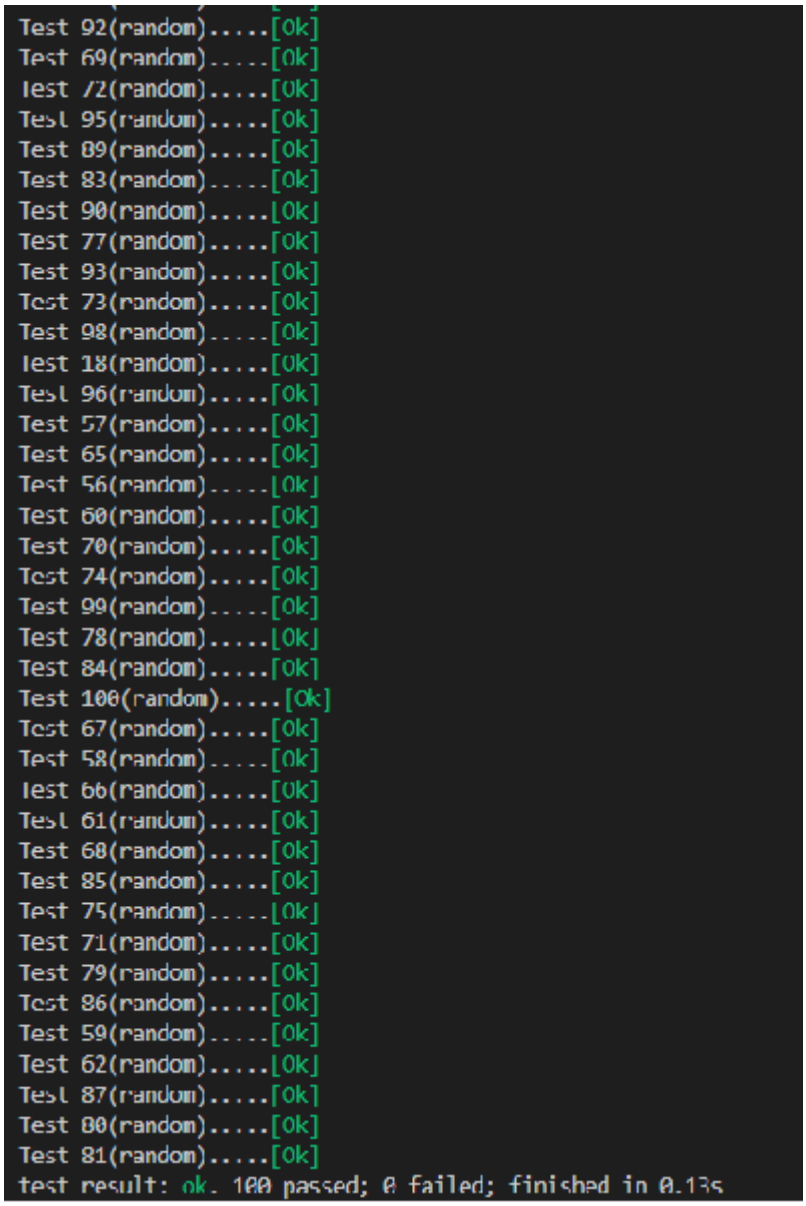
5.3 培养方案测试

培养方案主体函数如下：

```
...
int check(int j, int k) {
    x = 0;
    tj = j;
    tk = k;
    while(tk > 0) {
        if(tmp[x] != name[tj]) {
            return 0;
        }
        x += 1;
        tj += 1;
        tk -= 1;
    }
    return 1;
}
...
while(i < n) {
        j = 0;
        ok = 0;
        tok = 1;
        k = 0;
        while(lim[i * 500 + j] != ling) {
            ch = lim[i * 500 + j];
            if(ch == ';') {
                if(k != 0) {
                    tmp[k] = 0;
                    tmp[k + 1] = 0;
                    tmp[k + 2] = 0;
                    p = 0;
                    q = -1;
                    while(p < n) {
                        int cnt = 0;

                        if(check(p * 10, k + 2) == 1) {
```

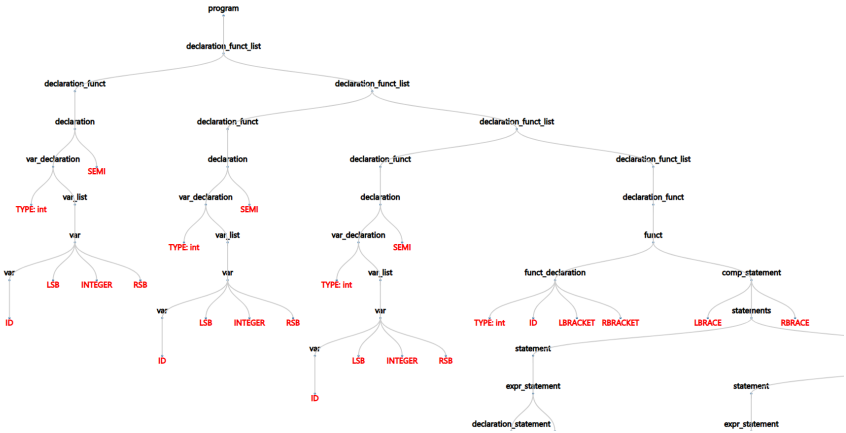
```
                            q = p;
                        }
                        p += 1;
                    }
                    if(q == -1)
                        tok = 0;
                    else {
                        if(r[q] <= 0) {
                            tok = 0;
                        }
                    }
                    k = 0;
                }
                if(tok == 1)
                    ok = 1;
                tok = 1;
                k = 0;
            } else {
                if(ch == ',') {
                    tmp[k] = 0;
                    tmp[k + 1] = 0;
                    tmp[k + 2] = 0;
                    p = 0;
                    q = -1;
                    while(p < n) {
                        if(check(p * 10, k + 2) == 1) {
                            q = p;
                        }
                        p += 1;
                    }
                    if(q == -1)
                        tok = 0;
                    else {
                        if(r[q] <= 0) {
                            tok = 0;
                        }
                    }
                    k = 0;
                }
                else {
                    tmp[k] = ch;
                    k += 1;
                }
            }
            j += 1;
        }
...
```

运行结果如下：



第六章 可视化

在运行test.out后，会自动生成.json文件并放置在visual目录下。在live server的环境下打开html即可查看可视化的抽象语法树，例如：



其中终结符是红色，非终结符是黑色（限于篇幅，没有截取整张图片）