语义实验报告

基础函数及定义

节点属性

- 节点属性在语法的基础上加上了一些其他值，包括下图从value到values的变量
  - values对于不同的节点存储存储不同的特定值，比如它可以存储函数的参数的节点，或者在数组定义是存储下标以方便计算。

```
struct Node {
    char label[114];                // 节点标签
    int children[20];           // 子节点索引列表
    int n_children;             // 子节点数量
    int CHD_NO;                 // 画图的时候判断是父亲的第几个儿子。
    int Node_NO;                // 画图的时候判断父亲是谁。

    int value;
    bool isConst;
    bool isArr;
    bool isError;
    bool isGlobal;
    bool isAddr;                // 是否是根据地址定位的，如果是，则 offset 指向的是存储地址的指针，而非数组本身
    int offset;                 // offset to ebp
    int offsetInArray;          // offset in array
    int quad;

    std::vector<int> trueList, falseList;
    std::vector<Node*> values;
};
```

符号

- 用于存放符号的属性的

```
enum VarType { Int,
               ConstInt,
               Arr,
               ConstArr,
               FuncInt,
               FuncVoid,
               Addr };

struct Var {
    VarType type;
    int value;              // 用于常量
    int offset;             // 用于指示变量在栈中的位置
```

```
    bool isGlobal;          // 用于指示变量是否是全局变量，以区分处理方式
    vector<Node*> values;   // 用于函数参数，数组维度等
    vector<int> values2;    // 仅用于保存常量数组的值
    Var() {}
    Var(VarType type)
        : type(type) {}
    Var(VarType type, int value, int offset, bool isGlobal)
        : type(type), value(value), offset(offset), isGlobal(isGlobal) {}
    Var(VarType type, int value, int offset, bool isGlobal, vector<Node*> values)
        : type(type), value(value), offset(offset), isGlobal(isGlobal), values(values) {}
};
```

输出

- 类Assemble的主题是一个string的vector，存放输出至.s文件的指令
  - 里面集成了常用的指令包括调用函数，退出函数，栈上变量存入寄存器，寄存器变量存入栈，控制流回填，控制流生成新label，注释（在汇编代码中生成注释语句方便我调试）。

```
class Assemble : public SPrintfBuffer {
   public:
    Assemble() {

    }
    int labelCount = 1;
    void call()
    void ret()

    //addr代表是否存入地址，否的话就存入值。is_c代表是int类型还是char类型
    void var2reg(Node* node, const char* reg, bool addr = false,int is_c = false)

    void reg2var(const char* reg, Node* node)
    void backpatch(const vector<int>& list, int label)
    int newLabel()
    void comment(const char* str)
    void comment(const string& str)
};
```

其他定义

- level是指当前的深度，全局是0，随着函数调用而增加
- VarTable和funcTable分别是符号表和函数名表
- breakStack和continueStack是用于break和continue回填的
- offset是与当前函数ebp的值
- offsetStack：当调用函数时，把caller的offset存入然后offset=0
- tmp：用来临时存放字符串
- s_and_p_index用来计算是第几个scanf或printf以方便知道传参时用.LC几
- funcName：当前正在编译的函数的名字
- paramList：当前正在编译的函数的参数
- arr_dim，定义数组时记录其维度，例如a[1][2][3]，则arr_dim为{1，2，3}，定义新数组时先清空
- arr_level：数组嵌套赋值时用，指明当前区域在第几个大括号中

```
Assemble assemble;
int level = 0;  // 0: global
vector<map<string, Var>> varTable(1), funcTable(1);
vector<vector<pii>> breakStack(1), continueStack(1);
int offset;
vector<int> offsetStack(1);
char tmp[128];
int s_and_p_index = 0;
string funcName;
bool inVoidFunc = false;
bool hasReturn = false;
vector<Node*> paramList;
std::vector<int> arr_dim;       //数组赋值用来传递数组维度
int arr_level = 0;              //数组赋值用
```

其他函数

- merge：合并两个数组
- nextLevel&exitLevel：进入\退出函数时用，对level加减并存入\删除符号表的某一层
- alignStack：让offset能16字节对齐
- recoverStack函数退出时清空该函数的offset并恢复caller的的offset
- isVarInTable\isFuncInTable：通过名字查找是否在符号表\函数符号表
- insertVar\insertFunc：往符号表\函数符号表插入变量
- getVar\getFunc：通过名字在符号表\函数符号表中得到变量的Var

```
vector<T> merge(const vector<T>& a, const vector<T>& b)
void nextLevel()
void exitLevel()
void alignStack()
void recoverStack()
bool isVarInTable(const string& name)
bool isFuncInTable(const string& name)
void insertVar(const string& name, Var var)
void insertFunc(const string& name, Var var)
```

语义分析

变量定义及赋值

- 根据是否全局分别定义
  - 全局的直接定义为.globl，然后循环用.long赋值
  - 非全局的则定义在栈上，用后循环用var2reg函数赋值
- 嵌套赋值
  - 以下以const的赋值为例子，主要思想就是对赋值的每一个{}，检查里面的值的数量和数组定义的是否一致，不一致补0。
  - LEFTB这个非终结符直接推导为左括号，顺便把arr_level++，即每遇到一个左括号便arr_level++

- 例如对a[1][2][3] = {{{1},1}}
  - 先是规约至{1}，此时arr_level=3，第三维需要of = 3个值，所以补2个0变成{1,0,0}。
  - 然后规约至{{1}，1}，但由于补了0，所以实际上上规约{{1,0,0},1}，此时arr_level=2，第二维需要of = 2*3=6个值，所以继续补0为{{1,0,0},1,0,0}
  - 最后规约至{{{1},1}}，但由于补了0，所以实际上上规约{{{1,0,0},1,0,0}}，此时arr_level=1，需要of=1*2*3=6个值，不用补0

```
ConstInitVal:LEFTB OtherConstInitVal '}'{
    fprintf(fDetail, "ConstInitVal -> { OtherConstInitVal }\n");
    $$ = newNode("ConstInitVal");
    addChild($$, newNode("{"));
    addChild($$, $2);
    addChild($$, newNode("}"));
        nodes[$$].values = nodes[$2].values;
        nodes[$$].value = nodes[$2].values.size();

        int of = 1;
        for(int i=arr_level;i<=arr_dim.size();i++)
            of *= arr_dim[i-1];
        arr_level--;

        if(nodes[$$].value == of){
            break;
        }
        else if(of > nodes[$$].value){
            of -= nodes[$$].value;
            while(of--){
                Node* nNode = new Node();
                copyNode(nNode, nodes[$2].values[0]);
                nNode->value = 0;
                nNode->isConst = true;
                nodes[$$].values.push_back(nNode);
            }
        }
        else{
            yyerror("Wrong arrList");
        }
        nodes[$$].value = nodes[$$].values.size();

    }
```

函数调用

- 理论上应该先讲函数定义的，不过我决定要先讲函数调用
- 对普通函数:
  - 先字节对齐
  - 然后传参，传参只需要把参数传入栈中，有三种形式：
    - 首先计算flag，默认为false，若当前正在编译的函数为func1，被调用的函数为func2，若传给func2的参数是func1的参数，且这个参数是数组，则flag为true。
    - 如果是数组且flag为false，把数组的地址传入栈
    - 如果是数组且flag为true，把数组的“值”传入栈（如果此时传入数组的地址，实际传入的是地址的地址）
    - 如果是int，把数的地址传入

- 再字节对齐（这是为了保证被调用的函数的初地址是16字节对齐的）
- call

```
UnaryExp | Ident '(' FuncRParams ')'{
    fprintf(fDetail, "UnaryExp -> Ident ( FuncRParams )\n");
    $$ = newNode("UnaryExp");
    addChild($$, newNode($1));
    addChild($$, newNode("("));
    addChild($$, $3);
    addChild($$, newNode(")"));

        if(!isFuncInTable($1)){
            yyerror("Call Undefined Function");
        }{
            alignStack();
            //funcName
            auto [depth, func] = getFunc($1);
            auto [depth1, func1] = getFunc(funcName);

            for(int i = nodes[$3].value - 1; i >= 0 ; i--){
                bool flag = false;
                for(auto cur:func1.values){
                    if(!strcmp(cur->label, nodes[$3].values[i]->label)){
                        flag = true;
                        break;
                    }
                }

                std::string tmpstr = "main";
                if(funcName == tmpstr)
                    flag = false;

                if(func.values[i]->isArr && !flag){
                    assemble.var2reg(nodes[$3].values[i], "%r8", true);
                    assemble.append("\tsubq\t$8, %%rsp\n");
                    offset -= 8;
                    assemble.append("\tmovq\t%%r8, %d(%%rbp)\n", offset);

                }else if(func.values[i]->isArr &&flag){
                    assemble.append("\tmovq\t%d(%%rbp), %%r8\n", nodes[$3].values[i]->offset);
                    assemble.append("\tsubq\t$8, %%rsp\n");
                    offset -= 8;
                    assemble.append("\tmovq\t%%r8, %d(%%rbp)\n", offset);
                }else{
                    assemble.var2reg(nodes[$3].values[i], "%r8d");
                    assemble.append("\tsubq\t$4, %%rsp\n");
                    offset -= 4;
                    assemble.append("\tmovl\t%%r8d, %d(%%rbp)\n", offset);
                }
            }
            alignStack();
            assemble.append("\tcall\t%s\n", $1);
            if(func.type == FuncInt){
                offset -= 4;
                assemble.append("\tsubq\t$4, %%rsp\n");
                assemble.append("\tmovl\t%%eax, %d(%%rbp)\n", offset);
                nodes[$$].offset = offset;
```

```
            }
        }

    }
```

- scanf和printf
  - 以下以scanf为例
  - 对非终结符STRING，直接存为.LC+s_and_p_index，s_and_p_index依次增加
  - 只支持一个参数，传入%rsi，然后.LC传入%rdi

```
UnaryExp| SCANF'(' STRING ',' PointParams ')'{
    fprintf(fDetail, "UnaryExp -> sacnf ( \"string\" , FuncRParams )\n");
    $$ = newNode("UnaryExp");
    addChild($$, newNode("scanf"));
    addChild($$, newNode("("));
    addChild($$, newNode($3));
    addChild($$, newNode(","));
    addChild($$, $5);
    addChild($$, newNode(")"));

        alignStack();
        fprintf(asm_file, ".LC%d:\n", s_and_p_index);
        fprintf(asm_file, "\t.string\t%s\n", $3);
        assemble.var2reg(nodes[$5].values[0], "%rsi", true);
        assemble.append("\tleaq\t.LC%d(%%rip), %%rdi\n",s_and_p_index);
        assemble.append("\tmovl\t$0, %%eax\n");
        assemble.append("\tcall\t__isoc99_scanf@PLT\n");
        s_and_p_index++;
    }
```

函数定义

- 以下以int带参函数为例
  - FuncName记录当前的函数名到全局变量funcName
  - FuncFParams把参数传入当前参数链paramList
  - EnterIntFuncBlock可以分为InsertIntFuncName和EnterFuncBlock
    - InsertIntFuncName把函数插入函数表
    - EnterFuncBlock主要是做进入函数的预处理，以及给传入本函数的参数赋基于当前函数ebp的offset
  - BlockWithoutNewLevel就是推导为{block}
  - FuncEnd结束函数，增加%rsp以恢复栈，并ret

```
FuncDef:INT FuncName '(' FuncFParams ')'  EnterIntFuncBlock BlockWithoutNewLevel FuncEnd
```

控制流语句

照着书上写就完事了，注意一下维护栈的正确

布尔表达式

照着书上写就完事了

基本运算

照着书上写就完事了，创建临时变量保存结果就行了

结果

通过了测试点1，2，3，4，6，8，10

其实控制流语句有点bug，会导致创建.L标签有一点问题，不过今天是最后一天了，写不完了