语义实验

功能支持情况

- 完整支持的局部(全局)变量(常量)(数组)的声明和使用
- 完整支持各类运算，包括算术运算和逻辑运算
- 支持 if-else/while 语句
- 支持多层 break/continue
- 支持函数(递归)调用
- 支持函数参数传递变量
- 支持函数参数传递数组
- 支持函数参数传递部分数组（例如传递 `x[2][3][4]` 的 `x[1]` 到 `p[][4]`）
- 支持函数参数传递常数组为实参到数组形参上，不进行类型转换保护（同 C 语言），不允许函数形参为常数组（同 sysY 定义）
- 支持比较复杂的数组定义，比如 `int b[2][N*2]={1+1, a[1][0] * a[0][1]}` 其中 `a` 是常数组
- 支持就近原则的符号表
- 支持变量和函数重名（同 sysY 定义中所述）
- 不支持复杂的初值列表，比如 `{{1,2}, {3,4}}`，这个比较繁琐所以没有实现

以上，除了最后一点细节，基本完整地实现了 sysY。

设计思路

本阶段有两类难点，一类是语义翻译设计，一类是编写汇编代码。以下将交叉讨论。

基本构思

每个节点的内容都在栈上开辟一个临时空间存储，操作时移动到 `r8,r9` 这两个比较安全的无关的寄存器上，操作结束后再存回去。

临时空间在每个 BLOCK 结束时才释放。

节点的数据结构

语义阶段特有的数据结构如下

```
int value;
bool isConst;
bool isArr;
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;
vector<int> trueList, falseList;
vector<Node*> values;
```

`offset` 对于一般变量指的是其在栈上的偏移量，对于数组中的某个元素则是其数组在栈上的偏移量。

`offsetInArray` 对于一般变量是 0，对于数组中的某个元素则是其在数组中的下标（未乘4）。

values

比较特别的是 `values` 字段，其用在如参数列表，数组下标等场所，用于存储一系列的值，用 `Node*` 存意味着其存储的事实上是其一些孩子节点，比如参数列表中的参数（其中又存了参数名 `text`），方便统一处理。

另外，为了保留语法树的结构，values 实际上是从子结点复制一份随后再追加一个。

更好的实现方法是传递指针，但增加了维护内存的复杂度而且不便于准确反应语法树，不便于debug。

变量和函数的存储结构（在符号表中）

他们统一用 `Var` 存储

```
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
};
```

全局变（常）量（数组）的实现

汇编代码实现

```
const int array_const_int[2] = {1, 2};
int array_int[2];
const int var_const_int = 1;
int var_int;
```

全局数组统一用 `.long` 给出 0 初始值，符合 C 语言的语义。

```
    .section    .rodata
    .align  4
    .type   array_const_int, @object
    .size   array_const_int, 8
array_const_int:
    .long   1
    .long   2
    .text
    .globl  array_int
    .data
    .align  4
    .type   array_int, @object
```

```
    .size   array_int, 8
array_int:
    .long   0
    .long   0
    .text
    .section    .rodata
    .align  4
    .type   var_const_int, @object
    .size   var_const_int, 4
var_const_int:
    .long   1
    .text
    .globl  var_int
    .data
    .align  4
    .type   var_int, @object
    .size   var_int, 4
    .long   0
    .text
```

语义翻译中的维护方式

全局

全局变量只需要用 `%rip` 加上其名字即可访问，所以只需要记录其名字，以及可能的相对数组头偏移量。

局部

要创建他们，只需要移动 `%rsp` 即可。需要维护的信息即此时其相对于这层栈帧的偏移量。

在我的实现中，我维护一个 `offset` 表示相对于栈帧的偏移量。

这里，局部常量也被创建在栈上。

var2reg

从内存加载到寄存器，需要讨论常量，数组，全局，局部，是否是用地址指向等。

```
// addr means put the address of the variable into the register, otherwise put
the value
void var2reg(Node* node, const char* reg, bool addr = false) {
    if (node->isConst) {
        this->append("\tmovl\t$%d, %s\n", node->value, reg);
    } else if (node->isGlobal) {  // 全局
        if (node->isArr) {
            this->append("\tmovl\t%d(%%rbp), %%ebx\n", node->offsetInArray);
            this->append("\tcltq\n");
            this->append("\tleaq\t0(, %%rbx, 4), %%rdx\n");
            this->append("\tleaq\t%s(%%rip), %%rbx\n", node->text);
            if (!addr) {
                this->append("\tmovl\t(%%rdx, %%rbx), %s\n", reg);
            } else {
                this->append("\tleaq\t(%%rdx, %%rbx), %s\n", reg);
            }
```

```
        } else {
            if (!addr) {
                this->append("\tmovl\t%s(%%rip), %s\n", node->text, reg);
            } else {
                this->append("\tleaq\t%s(%%rip), %s\n", node->text, reg);
            }
        }
    } else {
        if (node->isArr) {
            if (node->isAddr) {
                this->append("\tmovl\t%d(%%rbp), %%ebx\n", node->offsetInArray);
                this->append("\tcltq\n");
                this->append("\tmovq\t%d(%%rbp), %%r10\n", node->offset);
                if (!addr) {
                    this->append("\tmovl\t(%%r10, %%rbx, 4), %s\n", reg);
                } else {
                    this->append("\tleaq\t(%%r10, %%rbx, 4), %s\n", reg);
                }
            } else {
                this->append("\tmovl\t%d(%%rbp), %%ebx\n", node->offsetInArray);
                this->append("\tcltq\n");
                if (!addr) {
                    this->append("\tmovl\t%d(%%rbp, %%rbx, 4), %s\n", node-
>offset, reg);
                } else {
                    this->append("\tleaq\t%d(%%rbp, %%rbx, 4), %s\n", node-
>offset, reg);
                }
            }
        } else {
            if (!addr) {
                this->append("\tmovl\t%d(%%rbp), %s\n", node->offset, reg);
            } else {
                this->append("\tleaq\t%d(%%rbp), %s\n", node->offset, reg);
            }
        }
    }
}
```

reg2var 类似

```
void reg2var(const char* reg, Node* node) {
    if (node->isGlobal) {  // 全局
        if (node->isArr) {
            this->append("\tmovl\t%d(%%rbp), %%ebx\n", node->offsetInArray);
            this->append("\tcltq\n");
            this->append("\tleaq\t0(, %%rbx, 4), %%rdx\n");
            this->append("\tleaq\t%s(%%rip), %%rbx\n", node->text);
            this->append("\tmovl\t%s, (%%rdx, %%rbx)\n", reg);
        } else {
            this->append("\tmovl\t%s, %s(%%rip)\n", reg, node->text);
        }
    } else {
        if (node->isArr) {
            if (node->isAddr) {
```

```
                this->append("\tmovl\t%d(%%rbp), %%ebx\n", node->offsetInArray);
                this->append("\tcltq\n");
                this->append("\tmovq\t%d(%%rbp), %%r10\n", node->offset);
                this->append("\tmovl\t%s, (%%r10, %%rbx, 4)\n", reg);
            } else {
                this->append("\tmovl\t%d(%%rbp), %%ebx\n", node->offsetInArray);
                this->append("\tcltq\n");
                this->append("\tmovl\t%s, %d(%%rbp, %rbx, 4)\n", reg, node-
>offset);
            }
        } else {
            this->append("\tmovl\t%s, %d(%%rbp)\n", reg, node->offset);
        }
    }
}
```

LVal 中数组的访问和记录方法

> 需要指出：所有关于维度的存储，都被反转存储，也就是 values[0] 事实上是最后一维，这方便我们计算后缀积

语法

```
LVal : IDENT ArrayDim
```

翻译核心思路

在栈上即时创建一个临时变量计算和存储偏移量，对应以下代码的 `$$->offsetInArray = offset;`，如我们之前所述，数组首地址被存储在 `var.offset(%ebp)` 里，所以将 `offset` 也转存到 `$$` 中。

至此整个 `LVal` 就可以被两个 `offset` 确定。

这里的细节包括：

1. var.offset 指向一个 8 bytes 的地址，当 var.type == Addr，也就是这个变量是地址间接定位的（出现在数组传参中），但 `LVal` 不负责取值，所以只需要忠实地复制和标记即可。
2. 同时，因为部分数组的存在，变量的维度和使用时的维度大小可能不对应，作为参数的第一维大小可能为空，这些需要简单处理一下。

```
offset -= 4;
assemble.append("\tsubq\t$4, %%rsp\n");
assemble.append("\tmovl\t$0, %d(%%rbp)\n", offset);
int curSize = 1;
int delta = var.values.size() - $2->values.size();
for(int i = 0; i < var.values.size(); i++){
    if(i >= delta){
        assemble.var2reg($2->values[i - delta], "%r8d");
        assemble.append("\timull\t$%d, %%r8d\n", curSize);
        assemble.append("\taddl\t%d(%%rbp), %%r8d\n", offset);
        assemble.append("\tmovl\t%%r8d, %d(%%rbp)\n", offset);
    }
    if(var.values[i]){
        curSize *= var.values[i]->value;
    }
}
```

```
$$->isArr = true;
$$->offset = var.offset;
$$->isAddr = var.type == Addr;
$$->offsetInArray = offset;
```

以上代码即实现了累乘后缀积，再乘上当前维度的下标累加到偏移量上。

LVal 中变量的访问和记录方法

同上，不再赘述。

```
 IDENT {
$$=newNode(L_VAL_TYPE);
addChildren($$, $1);
     // $$->text = $1->text;
     strcpy($$->text, $1->text);
     if(!isVarInTable($1->text)){
         yyerror("Reference Undefined Variable");
     }else{
         auto [depth, var] = getVar($1->text);
         $$->isGlobal = var.isGlobal;
         if(var.type == ConstInt){
             $$->isConst = true;
             $$->value = var.value;
         }else{
             $$->offset = var.offset;
         }
     }
}
```

定义函数

以以下语法为例

```
INT FuncName LEFTP FuncFParams RIGHTP EnterIntFuncBlock BlockWithoutNewLevel
FuncEnd
```

做法是，在对应的模块中，先将函数名，参数列表先暂存到全局变量中供之后使用；

`EnterFuncBlock` 时，压入新的函数和变量符号表，`FuncEnd` 时弹出，这本来是 `Block` 中做的事情，但是因为需要先压表，再插入新的变量，所以提前到 `EnterIntFuncBlock` 来。

插入变量时的 `offset` 为 `32+curSize` 其中 `32` 是每次进入函数维护寄存器，以及返回地址所需要的空间，`curSize` 即参数的累积大小，对应了每个参数的位置，因为有的是传 `int`，有的是传地址所以累加值不同。

```
FuncName:
    IDENT{
        $$ = $1;
        funcName = $1->text;
    }
;

EnterIntFuncBlock:
```

```
    InsertIntFuncName EnterFuncBlock {
        ;
    }
;

InsertIntFuncName:
    {
        if(isFuncInTable(funcName)){
            yyerror("duplicated function name");
        }else{
            insertFunc(funcName, Var(FuncInt, 0, 0, false, paramList));
        }
    }
;

EnterVoidFuncBlock:
    InsertVoidFuncName EnterFuncBlock {
        ;
    }
;

InsertVoidFuncName:
    {
        if(isFuncInTable(funcName)){
            yyerror("duplicated function name");
        }else{
            insertFunc(funcName, Var(FuncVoid, 0, 0, false, paramList));
            inVoidFunc = true;
        }
    }
;

EnterFuncBlock:
    /* empty */ {
        nextLevel();
        hasReturn = false;
        assemble.append("\t.globl\t%s\n", funcName.c_str());
        assemble.append("\t.type\tmain, @function\n");
        assemble.append("%s:\n", funcName.c_str());
        assemble.call();
        int curSize = 0;
        for(int i=0;i<paramList.size();i++){
            if(paramList[i]->isArr){
                insertVar(paramList[i]->text, Var(Addr, 0, 32+curSize, false,
paramList[i]->values));
                curSize += 8;
            }else{
                insertVar(paramList[i]->text, Var(Int, 0, 32+curSize, false));
                curSize += 4;
            }
        }
        paramList.clear();
    }
;
```

```
FuncEnd:
    {
        if(!hasReturn){
            assemble.append("\taddq\t$%d, %%rsp\n", -offset);
            assemble.ret();
        }
        offset = 0;
    }
;
```

调用函数

同样地，需要区分传地址和传值。最后，如果是 `int` 函数，则这个节点还需要一个位置用来记录返回值，同样即时在栈上临时创建一个即可。最后函数的返回值会保存在 `%eax` 中，在返回后我们转存到这个临时变量中即可。

```
alignStack();
auto [depth, func] = getFunc($1->text);
for(int i = $3->value - 1; i >= 0 ; i--){
    if(!func.values[i]->isArr){
        assemble.var2reg($3->values[i], "%r8d");
        assemble.append("\tsubq\t$4, %%rsp\n");
        offset -= 4;
        assemble.append("\tmovl\t%%r8d, %d(%%rbp)\n", offset);
    }else{
        assemble.var2reg($3->values[i], "%r8", true);
        assemble.append("\tsubq\t$8, %%rsp\n");
        offset -= 8;
        assemble.append("\tmovq\t%%r8, %d(%%rbp)\n", offset);
    }
}
assemble.append("\tcall\t%s\n", $1->text);
if(func.type == FuncInt){
    offset -= 4;
    assemble.append("\tsubq\t$4, %%rsp\n");
    assemble.append("\tmovl\t%%eax, %d(%%rbp)\n", offset);
    $$->offset = offset;
}
```

if-else

其中的大部分核心内容只需要借鉴 PPT 或者现有的语义翻译样板即可，用到的主要是回填技术。

唯一值得注意的是关于临时变量（比如 Cond 中产生的）的空间回收，
这里 `EnterStmt` 和 `ExitStmt` 分别起到记录当前的 `offset`，压入符号表以及还原的作用，事实上，更符合 C 的做法是
在整个 `IF` 的外部加上 `EnterStmt` 和 `ExitStmt`，但是这样会导致意料之外的语法冲突。以下的实现的本质是第一层 IF 是被视为当前
活跃的栈帧的一部分，下一层开始才涉及到回收等。

```
| IF LEFTP Cond RIGHTP NewLabel EnterStmt Stmt ExitStmt %prec IFX { //
EnterStmtand ExitStmt has no effect, but can solve conflict
    $$=newNode(STMT_TYPE);
```

```
    addChildren($$, $1, $2, $3, $4, $7);
    assemble.backpatch($3->trueList, $5->quad);
    int end = assemble.newLabel();
    assemble.backpatch($3->falseList, end);
}
| IF LEFTP Cond RIGHTP NewLabel EnterStmt Stmt ExitStmt ELSE AfterElse
NewLabelEnterStmt Stmt ExitStmt NewLabel %prec ELSEX {
    $$=newNode(STMT_TYPE);
    addChildren($$, $1, $2, $3, $4, $7, $9, $13);
    assemble.backpatch($3->trueList, $5->quad);
    assemble.backpatch($3->falseList, $11->quad);
    assemble.backpatch($10->trueList, $15->quad);
}
```

while

while 略有不同，最主要的问题在于如果仍然沿用 if 的结构，Cond 中的临时变量会被反复创建，所以需要在进入 While 前记录当前的 offset，然后在 `ExitWhile` 即退出判断时立刻回收。事实上 if 也可以用类似的结构。

如果用三地址等方法可以更好地优化这部分代码，比如不需要反复移动栈指针等。

比较特别地，我们还需要给 break 和 continue 进行回填，因为 while 之间天然构成栈的结构，所以用全局的栈维护当前要回填的位置集，这样代码比较简单。具体方法即： break 的位置，回填到 while 结束的位置，continue 的位置，回填到 while 开始的位置。同时，二者都需要还原栈。

```
| WHILE EnterWhile EnterStmt LEFTP Cond RIGHTP ExitWhile NewLabel Stmt ExitStmt
{
  $$=newNode(STMT_TYPE);
  addChildren($$, $1, $4, $5, $6, $9);
  assemble.backpatch($5->trueList, $8->quad);
  assemble.append("\tjmp\t.L%d\n", $2->quad);
  int whileEnd = assemble.newLabel();
  assemble.comment("while end");
  assemble.backpatch($5->falseList, $7->quad);
  assemble.backpatch($7->trueList, whileEnd);
  for(auto [line, of]: breakStack.back()){
      sprintf(tmp, "\taddq\t$%d, %%rsp\n", offset - of);
      assemble[line - 1] = tmp;
      assemble[line] += to_string(whileEnd) + "\n";
  }
  breakStack.pop_back();
  for(auto [line, of]: continueStack.back()){
      sprintf(tmp, "\taddq\t$%d, %%rsp\n", offset - of);
      assemble[line - 1] = tmp;
      assemble[line] += to_string($2->quad) + "\n";
  }
  continueStack.pop_back();
}
```

```
| BREAK SEMI {
  $$=newNode(STMT_TYPE);
  addChildren($$, $1, $2);
  assemble.append("");
```

```
  assemble.append("\tjmp\t.L");
  breakStack.back().push_back({assemble.line, offset});
}
| CONTINUE SEMI {
  $$=newNode(STMT_TYPE);
  addChildren($$, $1, $2);
  assemble.append("");
  assemble.append("\tjmp\t.L");
  continueStack.back().push_back({assemble.line, offset});
}
```

布尔表达式

同样是借鉴 PPT 或现有的语义翻译样本即可。

自己写也很容易，只要把握“短路”的概念即可。

```
| LAndExp AND EqExp {
    $$=newNode(L_AND_EXP_TYPE);
    addChildren($$, $1, $2, $3);
    assemble.backpatch($1->trueList, $3->quad);
    $$->trueList = $3->trueList;
    $$->falseList = merge($1->falseList, $3->falseList);
    $$->quad = $1->quad;
}
```

基本运算

如前文所述，对于每个计算节点，在栈上创建临时变量，保存结果，将此时的 `offset` 保存到节点信息中备以后使用。

单目运算例子：`-x`

```
| MINUS UnaryExp {
    $$=newNode(UNARY_EXP_TYPE);
    addChildren($$, $1, $2);
    if($2->isConst){
        // copyNode($$, $2);
        $$->isConst = true;
        $$->value = -$2->value;
    }else{
        assemble.var2reg($2, "%r8d");
        assemble.append("\tneg %%r8d\n");
        offset -= 4;
        assemble.append("\tsubq $4, %%rsp\n");
        assemble.append("\tmovl %%r8d, %d(%%rbp)\n", offset);
        $$->offset = offset;
    }
}
```

双目运算例子：`x + y`

```
| AddExp PLUS MulExp {
    $$=newNode(ADD_EXP_TYPE);
```

```
    addChildren($$, $1, $2, $3);
    if($1->isConst && $3->isConst){
        $$->isConst = true;
        $$->value = $1->value + $3->value;
    }else{
        assemble.var2reg($1, "%r8d");
        assemble.var2reg($3, "%r9d");
        assemble.append("\taddl\t%%r9d, %%r8d\n");
        offset -= 4;
        assemble.append("\tsubq\t$4, %%rsp\n");
        assemble.append("\tmovl\t%%r8d, %d(%%rbp)\n", offset);
        $$->offset = offset;
    }
}
```

附录

自定义测验代码

以下代码的运行效果均在检查时演示过。

```
// 传递数组
// 传递部分数组
// 复杂的数组声明
const int N = 3, M = 4;
int sum(int x[][M], int n){
    int i = 0;
    while(i < n){
        int j = 0;
        int sum = 0;
        while(j < M){
            sum = sum + x[i][j];
            j = j + 1;
        }
        printf(sum);
        i = i + 1;
    }
}
void fill(int x[][M], int n){
    int i = 0;
    while(i < n){
        int j = 0;
        while(j < M){
            x[i][j] = -i * j;
            j = j + 1;
        }
        i = i + 1;
    }
}
int main(){
    const int tmp[4] = {1, 2, 3, 4};
    int a[2][N][M] =    {tmp[0], tmp[1], tmp[2], tmp[3],
                         2*tmp[0], 2*tmp[1], 2*tmp[2], 2*tmp[3],
                         3, 6, 9, 12,
```

```
                          -1};
    sum(a[0], N);
    sum(a[1], N);
    fill(a[1], N);
    sum(a[1], N);
}
```

```
// 就近符号表
void func(){
    int x = 1;
    {
        int x = 2;
        {
            int x = 3;
            printf(x);
        }
        x = -2;
        printf(x);
    }
    printf(x);
}
int x = 0;
int main(){
    func();
    printf(x);
}
```

```
// 常数组传参演示
void func(int x[], int i){
    printf(x[i]);
}
const int number[10] = {0,1,2,3,4,5,6,7,8,9};
int main(){
    func(number, 0);
    func(number, 5);
    func(number, 6);
}
```

之前阶段的更新内容

因为允许使用 cpp，所以重构了之前的代码

使用变长模板参数替代了 `stdarg.h`，这意味着新版本的代码需要运行在支持 C++17 的编译器上

```
template <typename... Args>
void addChildren(Node* parent, Node* child, Args... args) {
    child->kChild = parent->childNum;
    parent->children[parent->childNum++] = child;
    child->parent = parent;
    if constexpr (sizeof...(args) > 0) {
        addChildren(parent, args...);
    }
}
```