南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

编译原理期末实验报告

编译原理

管昀玫

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2023 年 1 月 15 日

摘要

本次实验内容为构造一个编译器，主要工作为：词法分析、语法分析、类型检查、中间代码生成、目标代码生成。其中，前端由 lexer 和 parser 组成，中端由 TypeCheck 和 GenCode 构成，后端主要由生成汇编代码和寄存器分配两部分构成。本次小组成员为管昀玫和石家琪同学。

关键字：编译原理

目录

一、 总体设计

本课程实现的编译器 Compile-SysY 的总体架构分为前端、中端、后端三端。其中中端与后端所执行的任务为传统意义上的后端部分，将前端分析的抽象语法树转换为代码。这里区分中端的原因在于本编译器将导出的中间代码输出为 LLVM IR，并在其上进行类型检查后再由后端转换为对应的 arm 目标代码。

前端：由 lexer 和 Parser 构成：

- Lexer：将源码文本字符序列转换为 Token 流
- Parser：将 Token 流转换为 AST

中端：由 TypeCheck、Gencode 构成。其 Gencode 需要搭建 IR 的数据结构，即由 IRBuilder、Unit、Function、Basic Block、Instruction 和 Operand 构成。

- TypeCheck：在建立语法树的过程中进行相应的识别和处理。在建树完成后，自底向上遍历语法树进行类型检查。
- GenCode：代码生成的脚手架，执行代码生成的流程。从根节点深度优先遍历整棵语法树，导出 LLVM IR 代码。

后端：主要由生成汇编代码和寄存器分配两部分构成。

- AsmBuilder：汇编代码构造辅助类。其主要作用就是在中间代码向目标代码进行自顶向下的转换时，记录当前正在进行转换操作的对象，以进行函数、基本块及指令的插入。
- MachineCode：汇编代码构造相关的框架，大体的结构和中间代码是类似的，只有具体到汇编指令和对应操作数时有不同之处。
- LiveVariableAnalysis: 活跃变量分析，用于寄存器分配过程。
- LinearScan: 线性扫描寄存器分配算法相关类，为虚拟寄存器分配物理寄存器。

二、 词法分析

词法分析器的功能为输入源程序，按照构词规则分解成一系列单词符号。单词是语言中具有独立意义的最小单位，包括关键字、标识符、运算符、界符和常量等。

(1) 关键字是由程序语言定义的具有固定意义的标识符。例如，Pascal 中的 begin，end，if，while 都是保留字。这些字通常不用作一般标识符。

(2) 标识符用来表示各种名字，如变量名，数组名，过程名等等。

(3) 常数常数的类型一般有整型、实型、布尔型、文字型等。

(4) 运算符如 +、-、*、/等等。

(5) 界符如逗号、分号、括号、等等。

词法分析器所输出单词符号常常表示成如下的二元式：(单词种别，单词符号的属性值) 单词种别通常用整数编码。标识符一般统归为一种。常数则宜按类型（整、实、布尔等）分种。关键字可将其全体视为一种。运算符可采用一符一种的方法。界符一般用一符一种的方法。对于每个单词符号，除了给出了种别编码之外，还应给出有关单词符号的属性信息。单词符号的属性是指单词符号的特性或特征。

在本次实现中，词法分析器的输出格式为：**(token, lexeme, lineno)**

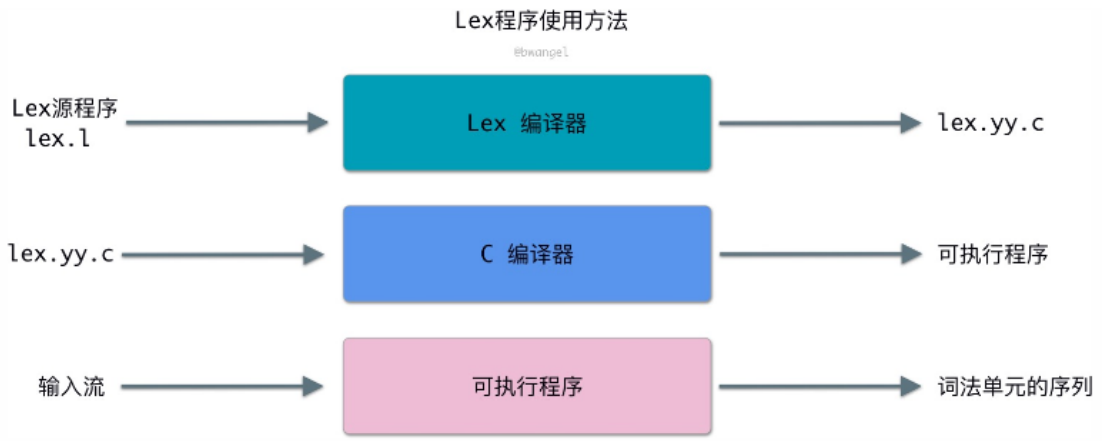

图 1: Syntactic Analysis

(一) 分工

在此部分，我主要完成:

- 八进制数字、十六进制数字的规则定义，将其保存为十进制输出。
- while/break/continue 语句的识别
- BLOCKCOMMENT

(二) 八进制/十进制/十六进制

1. 规则定义

正则表达式（Regular Expression）是用于描述一组字符串特征的模式，用来匹配特定的字符串。通过特殊字符 + 普通字符来进行模式描述，从而达到文本匹配目的的工具。其基本要素如下:

- 字符类
- 数量限定符
- 位置限定符
- 特殊符号

八进制、十进制、十六进制的规则定义如下所示:

八进制、十进制、十六进制的规则定义

```
HEX (0x[1-9|A-F|a-f][0-9|A-F|a-f]*|0x0)
OCT (0[1-7][0-7]*|00)
DECIMIAL ([1-9][0-9]*|0)
```

2. 转换为十进制存储

在这里，八进制和十进制都通过数学运算转换为十进制。需要注意的是，八进制由 0 开头，因此计算时需要从第 1 位开始；十进制由 0x 开头，计算时需要从第 2 位开始。

这里以十六进制转十进制代码为例：

八进制、十六进制转换十进制存储

```
{HEX} {
    #ifdef ONLY_FOR_LEX
        string a  = yytext;
        int len = a.length() - 2; // 由0x开头，因此减两位
        int all = 0;
        for(int i = 0; i < len; i++)
        {
            // 转化为十进制存储
            int temp1 = (pow(16, (len - i - 1)));
            int temp2;
            if(a[i + 2] >= '0' && a[i + 2] <= '9'){
                temp2 = (a[i + 2] - '0');
            }
            else if(a[i + 2] >= 'A' && a[i + 2] <= 'F'){
                temp2 = a[i + 2] - 'A' + 10;
            }
            else if(a[i + 2] >= 'a' && a[i + 2] <= 'f'){
                temp2 = a[i + 2] - 'a' + 10;
            }
            all = all + temp1 * temp2;
        }
        string temp = "HEX\t" + to_string(all) + "(DEC)" + "\tlinenum:" +
            to_string(yylineno);
        DEBUG_FOR_LAB4(temp);
    #else
        return HEX;
    #endif
}
```

在 lab4 中的实现如上方代码块所示。然而在最终的目标代码生成实验完成时，上述代码都将简化，由 **sscanf(yytext, "%o", &temp);** 和 **sscanf(yytext, "%x", &temp);** 分别完成八进制和十六进制的识别。

(三) 块注释

在 lab4 中的块注释的识别如下代码所示。声明部分的%x 声明了一个新的起始状态，而在之后的规则使用中加入 < 状态名 > 的表明该规则只在当前状态下生效。需要注意的是，%x 声明的为独占的起始状态，当处在该状态时只有规则表明为该状态的才会生效，而%s 可以声明共享的起始状态，当处在共享的起始状态时，没有任何状态修饰的规则也会生效。

块注释

```
BLOCKCOMMENTBEGIN "/*"
```

```
BLOCKCOMMENTELEMENT .|\n
BLOCKCOMMENTEND "*/"
%x BLOCKCOMMENT

LINECOMMENTBEGIN "//"
LINECOMMENTELEMENT .
LINECOMMENTEND \n
%x LINECOMMENT

{BLOCKCOMMENTBEGIN} {BEGIN BLOCKCOMMENT;}
<BLOCKCOMMENT>{BLOCKCOMMENTELEMENT} {}
<BLOCKCOMMENT>{BLOCKCOMMENTEND} {BEGIN INITIAL;}
{LINECOMMENTBEGIN} {BEGIN LINECOMMENT;}
<LINECOMMENT>{LINECOMMENTELEMENT} {}
<LINECOMMENT>{LINECOMMENTEND} {BEGIN INITIAL;}
```

（四） while/break/continue

此处的识别较为简单，直接识别关键字即可，然后在 chars 上加上字符长度即可。

while/break/continue

```
"while" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("WHILE\twhile");
    chars += strlen("while");
    return WHILE;
}
"break" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("BREAK\tbreak");
    chars += strlen("break");
    return BREAK;
}
"continue" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("CONTINUE\tcontinue");
    chars += strlen("continue");
    return CONTINUE;
}
```

（五） 实验效果

构建出的词法分析结果案例如图所示。

```
[DEBUG LAB4]:   INT int linenum:1
[DEBUG LAB4]:   ID  a   linenum:1   IDcount:1
[DEBUG LAB4]:   SEMICOLON   ;   linenum:1
[DEBUG LAB4]:   INT int linenum:3
[DEBUG LAB4]:   ID  main    linenum:3   IDcount:2
[DEBUG LAB4]:   LPAREN  (   linenum:3
[DEBUG LAB4]:   RPAREN  )   linenum:3
[DEBUG LAB4]:   LBRACE  {   linenum:4
[DEBUG LAB4]:   INT int linenum:5
[DEBUG LAB4]:   ID  a   linenum:5   IDcount:3
[DEBUG LAB4]:   SEMICOLON   ;   linenum:5
[DEBUG LAB4]:   ID  a   linenum:6   IDcount:3
[DEBUG LAB4]:   ASSIGN  =   linenum:6
[DEBUG LAB4]:   DECIMIAL    1   linenum:6
[DEBUG LAB4]:   ADD +   linenum:6
[DEBUG LAB4]:   DECIMIAL    2   linenum:6
[DEBUG LAB4]:   SEMICOLON   ;   linenum:6
[DEBUG LAB4]:   IF  if  linenum:7
[DEBUG LAB4]:   LPAREN  (   linenum:7
[DEBUG LAB4]:   ID  a   linenum:7   IDcount:3
[DEBUG LAB4]:   LESS    <   linenum:7
[DEBUG LAB4]:   DECIMIAL    5   linenum:7
[DEBUG LAB4]:   RPAREN  )   linenum:7
[DEBUG LAB4]:   RETURN  return  linenum:8
[DEBUG LAB4]:   DECIMIAL    1   linenum:8
[DEBUG LAB4]:   SEMICOLON   ;   linenum:8
[DEBUG LAB4]:   RETURN  return  linenum:9
[DEBUG LAB4]:   DECIMIAL    0   linenum:9
[DEBUG LAB4]:   SEMICOLON   ;   linenum:9
[DEBUG LAB4]:   RBRACE  }   linenum:10

```

三、 语法分析

如果把词法分析看作为字母组合成单词的过程，那么语法分析就是一个把单词组合成句子的过程。正如在词法分析中使用正则表达式来描述词法的规则一样，我们在语法分析中使用一种比 RE 表达能力更强的工具——上下文无关文法，来描述语言的语法规则。我们可以把某一种语言看成无数个符合语法规则的句子的集合，根据给定的上下文无关文法我们可以判断某一个 Token 串是否符合某个语法规则；如果符合，那么我们可以把此文法和对应输入的 Token 串组合起来生成一个句子。整个流程如下图所示:

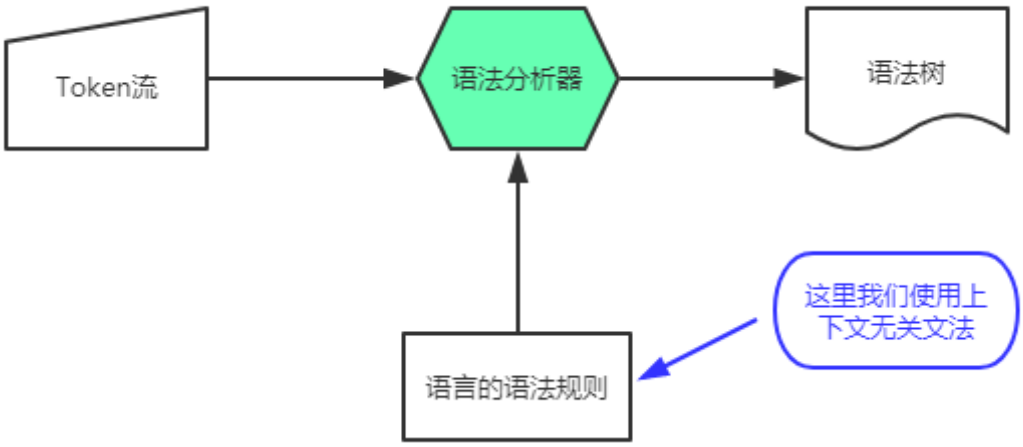

图 2: Syntactic Analysis

在本次实验中，我们采用自顶向下的方法构造语法树。
递归下降分析算法本质也是一种自顶向下分析算法，其基本思想如下：

- 对每一个非终结符构造一个分析函数
- 用前看符号指导产生式规则的选择

递归下降分析算法使用“分治法”来提高分析的效率，对于每一个产生式规则，都应该定义一个自己函数。因为在上下文无关文法中，终结符不可能出现在产生式的左边（可以在产生式左边出现终结符的文法叫做上下文有关文法），上下文无关文法中所有的产生式左边只有一个非终结符。所以我们在调用产生式规则的函数后，就分为两种情况：

1. 遇到终结符，因为终结符本质上是 token，所以直接把这个终结符和句子中对应位置的 token 进行比较，判断是否符合即可；符合就继续，不符合就返回
2. 遇到非终结符，此时只需要调用这个非终结符对应的函数即可。在这里函数可能会递归的调用，这也是算法名称的来源。

简单来说，就是遇到非终结符就调用函数，遇到终结符就比较。
在本次实验中，需要注意的重点有：符号表、类型系统、语法树、语法分析器。

(一) 分工

在这一部分，我负责的工作有：

- 符号表的修改
- 抽象语法树：Constant、Array、String、变量声明、函数定义、WhileStmt、BreakStmt、ContinueStmt、AssignStmt
- Parser: SEMICOLON、LBRACKET、RBRACKET、COMMA、初级表达式、关系表达式、相等性表达式、逻辑与或表达式、单目表达式、IfStmt、WhileStmt、BreakStmt、变量声明与定义、左值、函数定义、形参

(二) 符号表

符号表是编译器用于保存源程序符号信息的数据结构。符号表主要用于作用域的管理，我们为每个语句块创建一个符号表，块中声明的每一个变量都在该符号表中对应着一个符号表条目。

符号表的作用为：

- 收集符号属性；（词法分析）
- 上下文语义的合法性检查的依据；如果该标识符用于声明，那么语法分析器将创建相应的符号表条目，并将该条目存入当前作用域对应的符号表中，如果是使用该标识符，将从当前作用域对应的符号表开始沿着符号表链搜索符号表项。（语法分析）
- 作为目标代码生成阶段地址分配的依据；（语义分析）

符号表中的标识符一般设置的属性项目有：

- 符号名
- 符号的类型
- 符号的存储类别
- 符号的作用域及可视性
- 符号变量的存储分配信息
- 符号的其它属性

在本次的代码中，我们定义了三种类型的符号表项：用于保存字面值常量属性值的符号表项、用于保存编译器生成的中间变量信息的符号表项，以及保存源程序中标识符相关信息的符号表项。

在本次实验中，我负责符号表 lookup 函数的实现。loopup 函数意在返回标识符的符号表项的指针。符号表是一个栈，需要先查找当前的符号表，若未找到，则需要沿着 prev 指针查找上一级的符号表；若找到则返回指向该符号表项的指针，若查找失败则返回 nullptr。

Lookup

```
SymbolEntry* SymbolTable::lookup(std::string name) {
    map<string,SymbolEntry*>::iterator it;
    SymbolTable* p = this;
    it = p->symbolTable.find(name);
    while(it == p->symbolTable.end() && p->level != 0){
        p = p->prev;
        it = p->symbolTable.find(name);
    }
    if(it != p->symbolTable.end()){
        return it->second;
    }
    else
        return nullptr;
}
```

(三) 类型系统

在这一部分中，我主要负责数组的编写。

ArrayType 类继承于 Type，拥有 2 个私有变量：

- std::vector<int> indexs：数组的索引范围
- Type *baseType：数组的基本类型

其初始化函数为：

ArrayType

```
ArrayType(std::vector<int> indexs, Type *baseType = TypeSystem::intType)
        : Type(Type::ARRAY), indexs(indexs), baseType(baseType){
        this->size = 1;
        for (auto index : indexs){
            this->size *= index;
        }
        this->size *= 32;
    };
```

(四) Parser

我们 Parser 部分的总体设计如下：

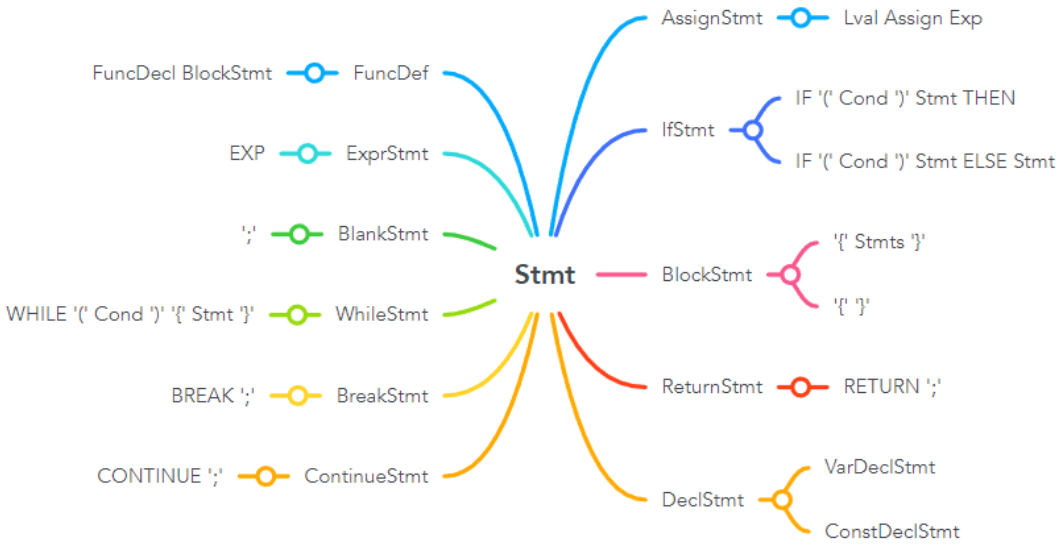

图 3: Parser1

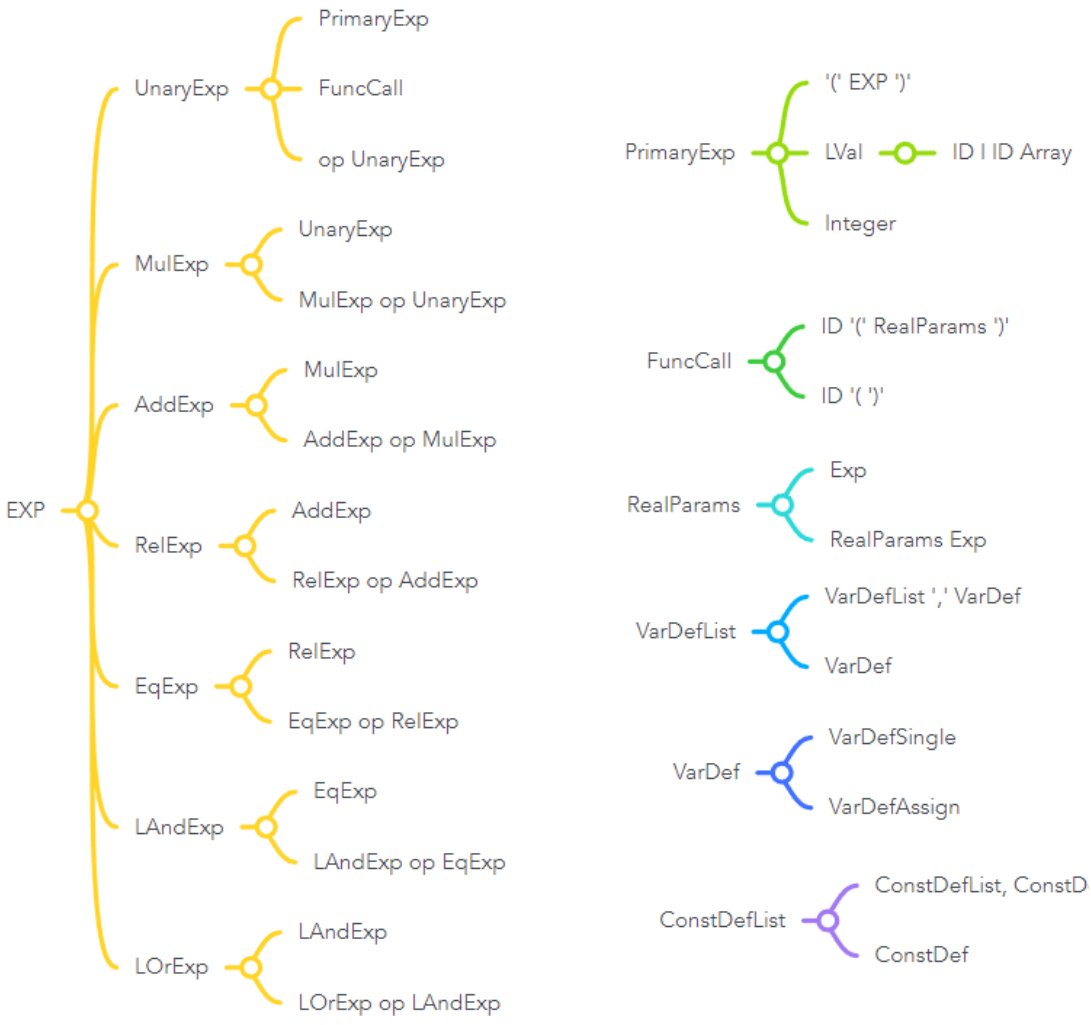
EXP
UnaryExp
PrimaryExp
FuncCall
op UnaryExp
MulExp
UnaryExp
MulExp op UnaryExp
AddExp
MulExp
AddExp op MulExp
RelExp
AddExp
RelExp op AddExp
EqExp
RelExp
EqExp op RelExp
LAndExp
EqExp
LAndExp op EqExp
LOrExp
LAndExp
LOrExp op LAndExp
PrimaryExp
'(' EXP ')'
LVal
ID | ID Array
Integer
FuncCall
ID '(' RealParams ')'
ID '(')'
RealParams
Exp
RealParams Exp
VarDefList
VarDefList ',' VarDef
VarDef
VarDef
VarDefSingle
VarDefAssign
ConstDefList
ConstDefList, ConstD
ConstDef

图 4: Parser2

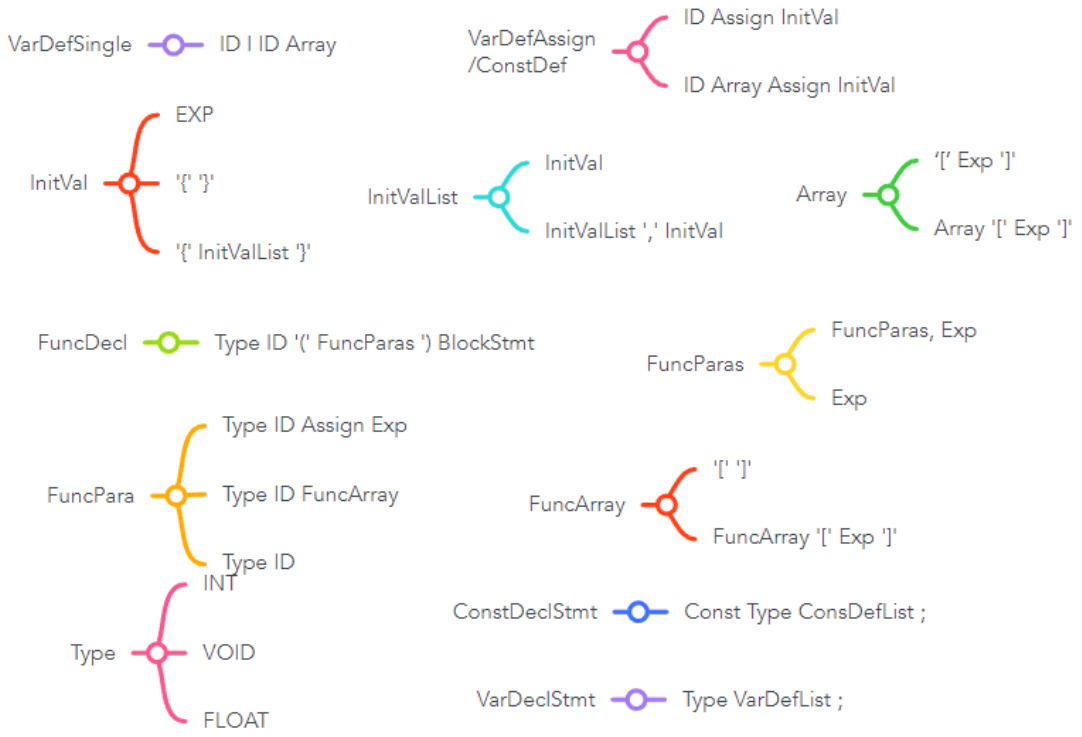

图 5: Parser1

在这一部分中，我主要负责的工作有：赋值语句、控制流（IF/IF-ELSE/WHILE）语句、部分表达式（初级表达式、关系运算、与/或表达式）、定义语句的变量定义、函数定义、数组相关。

1. 赋值语句

赋值语句由非终结符 LValStmt 表示。根据赋值语句的结构，赋值语句将表达式 Expr 的值赋给左值 LVal，因此可如下定义：

赋值语句 $$LValStmt \rightarrow LVal = Expr; \tag{1}$$

AssignStmt

```
AssignStmt
    : LVal ASSIGN Exp SEMICOLON {
        $$ = new AssignStmt($1, $3);
    }
    ;
```

其中 LVal 为左值。左值由非终结符 LVal 表示，作为左值的变量应当有唯一一个确定的地址，从而可以将赋值表达式的右值赋给这个地址上的变量。

在 SysY 语言中，左值可以为 int 类型变量或者 int 类型多维数组。而变量实质上是标识符，由终结符 Ident 表示。

由于 int 类型多维数组的维数无法确定，故应用递归定义的方法，因此用非终结符 ArrayLVal 表示多维数组类型左值。

具体定义如下:

左值 $LVal \rightarrow Ident \mid ArrayLVal$ (2)

多维数组类型左值 $ArrayLVal \rightarrow ArrayLVal\ [Number] \mid Ident\ [Number]$ (3)

可以改变的左值 $lval \rightarrow id \mid id\ [expr]$ (4)

LVal

```
LVal
    : ID {
        SymbolEntry* se;
        se = identifiers->lookup($1);
        $$ = new Id(se);
        delete []$1;
    }
    | ID Array{
        SymbolEntry* se;
        se = identifiers->lookup($1);
        $$ = new Id(se, $2);
        delete []$1;
    }
    ;
```

注意，在类型检查中，我们需要对 ID 进行检查，所有的 ID 必须是已经定义的 ID。ASSIGN 语句中的 Exp 是算数表达式，这部分将在算数表达式中详细说明。

2. 控制流语句

包括 If 语句、If Else 语句、While 语句。

分支语句 $CoDStmt \rightarrow if\ (\ Expr\ )\ Stmt$ (5)

$\mid if\ '('\ Expr\ ')'\ Stmt\ ['else'\ Stmt]$ (6)

IfStmt

```
IfStmt
    : IF LPAREN Cond RPAREN Stmt %prec THEN {
        $$ = new IfStmt($3, $5);
    }
    | IF LPAREN Cond RPAREN Stmt ELSE Stmt {
        $$ = new IfElseStmt($3, $5, $7);
    }
    ;
```

%prec 的使用是为了声明优先级。发生移进/规约冲突时，正确的做法是将 else 移入。SysY 语言中的 if 语句并没有终结符 then，在 yacc 中我们可以使用%prec 关键字，将终结符 then 的优先级赋给产生式。

While 语句:

循环语句 $LoopStmt \rightarrow while\ '('\ Expr\ ')'\ Stmt$ (7)

WhileStmt

```
WhileStmt
    : WHILE LPAREN Cond RPAREN{
        WhileStmt *whileNode = new WhileStmt($3);
        $<stmttype>$ = whileNode;//嵌入语义，将stmttype细化到whileNode类型
        whileStack.push_back(whileNode);//将当前whileNode节点压入保留栈
    }
    Stmt {
        StmtNode *whileNode = $<stmttype>5;
        ((WhileStmt*)whileNode)->setStmt($6);
        $$ = whileNode;
        whileStack.pop_back();//将当前whileNode节点弹出保留栈
    }
    ;
```

3. 算数表达式

1. PrimaryExp：基本表达式由非终结符 PrimaryExp 表示，实质上是每个表达式中最先计算的部分，也即其运算级最高。在 SysY 语言中，基本表达式可以有三种，一是用 () 括起来的表达式 Exp，二是左值 LVal，三是数值 Number。数值 Number 又可以分为整数类型和浮点数类型。

基本表达式 $PrimaryExp \rightarrow (Exp) \mid LVal \mid Number$ (8)

PrimaryExp

```
PrimaryExp
    : LPAREN Exp RPAREN {
        $$ = $2;
    }
    | LVal {
        $$ = $1;
    }
    | INTEGER {
        SymbolEntry* se = new ConstantSymbolEntry(TypeSystem::intType, $1);
        $$ = new Constant(se);
    }
    | FLOATINT {
        SymbolEntry* se = new ConstantSymbolEntry(TypeSystem::floatType, $1);
        $$ = new Constant(se);
    }
    ;
```

2. RelExp：关系表达式。相等性表达式由非终结符 EqExp 表示，计算 == 和!= 表达式的值。SysY 语言中，由于 == 和!= 运算的优先级低于 <、>、<= 和 >= 运算，因此计算完关系表达式的值之后就应当计算相等性表达式，也即其运算顺序低于关系表达式。

关系表达式 $RelExp \rightarrow AddExp$

$$
\begin{aligned}
&| \ RelExp \ < \ AddExp \\
&| \ RelExp \ > \ AddExp \\
&| \ RelExp \ <= \ AddExp \\
&| \ RelExp \ >= \ AddExp
\end{aligned}
$$

在这里，需要判断两个操作数中是否有 float 类型。如果有，则符号表项应初始化为 float 类型；否则就初始化为 int 类型

3. EqExp: 相等性表达式。相等性表达式由非终结符 EqExp 表示，计算 == 和!= 表达式的值。SysY 语言中，由于 == 和!= 运算的优先级低于 <、>、<= 和 >= 运算，因此计算完关系表达式的值之后就应当计算相等性表达式，也即其运算顺序低于关系表达式。

   相等性表达式 $EqExp \rightarrow RelExp|EqExp == RelExp|EqExp! = RelExp$

   这里将符号表项直接初始化为 floatType。

4. LAndExp: 逻辑与表达式由非终结符 LAndExp 表示，计算逻辑与 && 表达式的值。SysY 语言中，由于逻辑与 && 运算的优先级仅低于按位与或运算，因此计算完按位与或表达式的值之后才应当计算逻辑与表达式，也即其运算顺序低于按位与或表达式。

$$\textbf{逻辑与表达式} \quad LAndExp \rightarrow BitExp \ LAndExp \ \&\& \ BitExp \tag{9}$$

5. LOrExp: 逻辑或表达式由非终结符 LOrExp 表示，计算逻辑或 || 表达式的值。SysY 语言中，由于逻辑或 || 运算的优先级仅低于逻辑与 && 运算，因此计算完逻辑与表达式的值之后才应当计算逻辑或表达式，也即其运算顺序低于逻辑与表达式。

$$\textbf{逻辑或表达式} \quad LOrExp \rightarrow LAndExp \mid LOrExp \parallel LAndExp \tag{10}$$

6. Cond: 条件表达式。条件表达式由非终结符 Cond 表示，计算逻辑或 ||、逻辑与 && 和逻辑非! 表达式的值，因此用逻辑或表达式来定义条件表达式。

$$\textbf{条件表达式} \quad Cond \rightarrow LOrExp \tag{11}$$

由于此部分代码较为重复，故不再进行展示。

4. 变量定义

分为变量声明和变量定义两种情况。

变量声明 $VarDecl \rightarrow TypeArray \mid TypeVarlist$

变量定义 $VarDef \rightarrow \{ \ '[' \ ConstExp \ ']' \}$

$$| \ '\{' [\ InitVal \ \{',' \ InitVal\} \ ] \ '\}'$$

变量初值 $InitVal \rightarrow expr \ |\{\}| \ '\{' InitValList '\}'$

变量初值列表

$$
\begin{aligned}
Varlist \rightarrow \ & identifier, Varlist \mid identifier \\
& | \ identifier = identifier, Varlist \\
& | \ identifier = identifier \\
& | \ identifier = ConstExpr, Varlist \\
& | \ identifier = ConstExpr
\end{aligned}
$$

1. VarDef 用于定义变量。当不含有‘=’和初始值时，其运行时实际初值未定义。
2. VarDef 的数组维度和各维长度的定义部分不存在时，表示定义单个变量。存在时，和 ConstDef 类似，表示定义多维数组。
3. 当 VarDef 含有‘=’和初始值时，‘=’右边的 InitVal 和 CostInitVal 的结构要求相同，唯一的不同是 ConstInitVal 中的表达式是 ConstExp 常量表达式，而 InitVal 中的表达式可以是当前上下文合法的任何 Exp。
4. VarDef 中表示各维长度的 ConstExp 必须是能求值到非负整数，但 InitVal 中的初始值为 Exp，其中可以引用变量。

VarDef

```
VarDeclStmt
    : Type VarDefList SEMICOLON {$$ = $2;}
    ;
VarDefList
    : VarDefList COMMA VarDef {
        $$ = $1;
        $1->setNext($3);
    }
    | VarDef {$$ = $1;}
    ;
VarDef
    : VarDefSingle {
        $$=$1;
    }
    | VarDefAssign{$$=$1;}
    ;
VarDefSingle //声明语句
    : ID {
        SymbolEntry* se;
        se = new IdentifierSymbolEntry(TypeSystem::intType, $1, identifiers->
            getLevel());
        identifiers->install($1, se);
        $$ = new DeclStmt(new Id(se));
        delete []$1;
    }
    | ID Array {//数组定义
        SymbolEntry* se;
        se = identifiers->lookup($1, true);
        ExprNode *expr = $2; //
        std::vector<int> indexs; //存储数值中的维度值信息
        while (expr) { // 循环把数组维度值压入
            indexs.push_back(expr->getValue());
            expr = (ExprNode*)expr->getNext();
        }
        Type *arrType = new ArrayType(indexs, recentVarType); //创建数组类型
```

```
35          se = new IdentifierSymbolEntry(arrType, $1, identifiers->getLevel());
36          identifiers->install($1, se);
37          $$ = new DeclStmt(new Id(se));
38          delete []$1;
39      }
40      ;
41
42  VarDefAssign // 定义语句
43      : ID ASSIGN InitVal {// 有赋初值的情况
44          SymbolEntry* se;
45          se = new IdentifierSymbolEntry(TypeSystem::intType, $1, identifiers->
                getLevel());
46          identifiers->install($1, se);
47          ((IdentifierSymbolEntry*)se)->setValue($3->getValue());
48          $$ = new DeclStmt(new Id(se), $3);
49          delete []$1;
50      }
51      | ID Array ASSIGN {
52          SymbolEntry* se;
53          se = identifiers->lookup($1, true);
54          ExprNode *expr = $2;
55          std::vector<int> indexs;
56          while (expr) { // 循环把数组维度值压入
57              indexs.push_back(expr->getValue());
58              expr = (ExprNode*)expr->getNext();
59          }
60          Type *arrType = new ArrayType(indexs, recentVarType);
61          se = new IdentifierSymbolEntry(arrType, $1, identifiers->getLevel());
62          $<se>$ = se;
63          identifiers->install($1, se);
64          // 为数组分配空间, initArray是当前初始化数组的基地址
65          initArray = new ExprNode*[arrType->getSize() / TypeSystem::intType->
                getSize()];
66          memset(initArray, 0, sizeof(ExprNode*) * (arrType->getSize() /
                TypeSystem::intType->getSize()));
67          idx = 0;
68          std::stack<std::vector<int>>().swap(dimesionStack); // 清空栈
69          dimesionStack.push(indexs); // 压入该数组的维度信息
70          delete []$1;
71      } InitVal { //数组的值进行初始化
72          $$ = new DeclStmt(new Id($<se>4, $2));
73          ((DeclStmt*)$$)->setInitArray(initArray);
74          initArray = nullptr;
75          idx = 0;
76      }
77      ;
```

在 typeCheck 中，也应实现对定义语句数组 ID 的检查。

5. 函数定义

函数定义：

函数定义 $FuncDef \rightarrow FuncType$ **Ident** $'('\ [FuncFParams]\ ')'Block$ (12)

1. FuncDef 表示函数定义。其中的 FuncType 指明返回类型。
2. 当返回类型为 int 时，函数内所有分支都应当含有带有 Exp 的 return 语句。不含有 return 语句的分支的返回值未定义。
3. 当返回值类型为 void 时，函数内只能出现不带返回值的 return 语句。

形参为：

函数形参表 $FuncFParams \rightarrow FuncFParam\ \{\ ','\ FuncFParam\ \}$ (13)

函数形参 $BType \rightarrow$ **Ident** $['['']'\ \{\ '['\ Exp\ ']'\}]$ (14)

FuncCall

```
FuncDef
    :
    Type ID {
        recentFuncRetType = $1;//记录函数的返回类型
        identifiers = new SymbolTable(identifiers);
        spillPos = -1;
        intArgNum = 0;
        floatArgNum = 0;
    }
    LPAREN FuncParams RPAREN {
        Type* funcType;
        std::vector<Type*> vec;
        DeclStmt* temp = (DeclStmt*)$5;
        while(temp){//遍历参数列表，获取参数类型
            vec.push_back(temp->getId()->getSymPtr()->getType());
            temp = (DeclStmt*)(temp->getNext());
        }
        funcType = new FunctionType($1, vec);
        SymbolEntry* se = new IdentifierSymbolEntry(funcType, $2, identifiers
            ->getPrev()->getLevel());
        identifiers->getPrev()->install($2, se);
    }
    BlockStmt {
        SymbolEntry* se;
        se = identifiers->lookup($2);
        assert(se != nullptr);
        $$ = new FunctionDef(se, (DeclStmt*)$5, $8);
        SymbolTable* top = identifiers;
        identifiers = identifiers->getPrev();
        delete top;
        delete []$2;
```

```
    }
    ;
FuncParams
    : FuncFParams {$$ = $1;}
    | %empty {$$ = nullptr;}
FuncFParams
    : FuncFParams COMMA FuncFParam {
        $$ = $1;
        $$->setNext($3);
    }
    | FuncFParam {$$ = $1;}
    ;
```

6. 数组

数组 $Array \rightarrow ArrayVar = ArrayValue$

$ArrayVar \rightarrow Identifier\ '['ConstExpr']'\ ArrayVar \mid [ConstExpr] \mid \epsilon$

$ArrayValue \rightarrow '\{'\ ArrayValue\ '\}' \mid num, ArrayValue \mid num \mid \epsilon$

变量初值 $InitVal \rightarrow expr \mid \{\ \} \mid '\{'InitValList'\}'$ 未显式初始化的局部变量，其值是不确定的；而未显式初始化的全局变量，其（元素）值均被初始化为 0。

Array

```
InitVal
    : Exp {
        $$ = $1;
        if (initArray != nullptr) {
            initArray[idx++] = $1;
        }
    }
    | LBRACE RBRACE {//空数组
        std::vector<int> dimesion = dimesionStack.top();//获取当前数组的维度信息
        int size = 1;
        for (auto dim : dimesion) {//计算当前数组的大小
            size *= dim;
        }
        idx += size;
    }
    | LBRACE {//数组的初始化
        std::vector<int> dimesion = dimesionStack.top();//获取当前数组的维度信息
        // {-1, idx} 是vector的匿名构造，初始化为-1和idx
        dimesionStack.push({-1, idx});//压入一个标记，用来标记数组初始化的开始
        dimesion.erase(dimesion.begin());//删除第一个元素，因为数组初始化的时候，第一个元素是数组的大小
        if (dimesion.size() <= 0) {
            dimesion.push_back(1); // 一维数组的情况
```

```
        }
        dimesionStack.push(dimesion);
    } InitValList RBRACE {
        while (dimesionStack.top()[0] != -1) {
            dimesionStack.pop();
        }
        idx = dimesionStack.top()[1];//获取数组初始化的开始位置
        dimesionStack.pop();
        std::vector<int> dimesion = dimesionStack.top();
        int size = 1;
        for (auto dim : dimesion) {//计算当前数组的大小
            size *= dim;
        }
        idx += size;
    }
    ;
```

7. IOStream

SysY 运行时库提供一系列 I/O 函数，支持对整数、字符以及一串整数的输入和输出。为便于在 SysY 程序中控制输出的格式，诸如 putf 这样的 I/O 函数会超出 Sys Y 语言支持的数据类型的参数，如格式字符串。

SysY 运行时库提供如下的 I/O 函数，其中各个参数为整数值、变量、数组元素访问表达式:

1. int getint( )
   输入一个整数，返回对应的整数值。
   示例：int n; n = getint();

2. int getch()
   输入一个字符，返回字符对应的 ASCII 码值。
   示例：int n; n = getch();

3. int getarray(int[])
   输入一串整数，第 1 个整数代表后续要输入的整数个数，该个数通过返回值返回；后续的整数通过传入的数组参数返回。
   注：getarray 函数获取传入的数组的起始地址，不检查调用者提供的数组是否有足够的空间容纳输入的一串整数。
   示例：int a[10][10];int n; n = getarray(a[0]);

4. void putint(int)
   输出一个整数的值。
   示例：int n=10; putint(n); putint(10);putint(n);
   将输出:101010

5. void putch(int)
   将整数参数的值作为 ASCII 码，输出该 ASCII 码对应的字符。
   示例：int n=10; putch(n);
   将输出换行符

6. void putarray(int n,int a[])
   输入元素个数和数组，输出该数组。

基于前文的函数定义以及声明，输入输出函数的定义 IO 如下:

getint	$GetInt \rightarrow "getint()"$	(15)
getch	$GetCh \rightarrow "getch()"$	(16)
getarray	$GetArray \rightarrow "getarray(" \ LVal \ ")"$	(17)
putint	$PutInt \rightarrow "putint(" \ int \ ")"$	(18)
putch	$PutCh \rightarrow "putch(" \ Ident \ ")"$	(19)
putarray	$PutArray \rightarrow "putarray(num, " \ LVal \ ")"$	(20)
IO	$IO \rightarrow GetInt \vert GetCh \vert GetArray \vert PutInt \vert PutCh$	(21)

为了方便，我们不再另外在 parser 和 lexer 部分处理这 6 个函数，而是在 main.cpp 内直接 install() 进这 6 个函数。

（五） 抽象语法树

抽象语法树（abstract syntax code，AST）是源代码的抽象语法结构的树状表示，树上的每个节点都表示源代码中的一种结构，这所以说是抽象的，是因为抽象语法树并不会表示出真实语法出现的每一个细节，比如说，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现。抽象语法树并不依赖于源语言的语法，也就是说语法分析阶段所采用的上下文无文文法，因为在写文法时，经常会对文法进行等价的转换（消除左递归，回溯，二义性等），这样会给文法分析引入一些多余的成分，对后续阶段造成不利影响，甚至会使合个阶段变得混乱。因些，很多编译器经常要独立地构造语法分析树，为前端，后端建立一个清晰的接口。

在本次实验中，我们需要设计语法树的结点。结点的类型大体上可以分为表达式和语句，每种类型又可以分为许多子类型，如表达式结点可以分为词法分析得到的叶结点、二元运算表达式的结点等；语句还可以分为 if 语句、while 语句和块语句等。

1. ExprNode 与 StmtNode

ExprNode 为表达式节点，继承于 Node，由 BinaryExpr、UnaryExpr、funcCall、Constant、Id、ImplicitCastExpr、CallExpr 等表达式类继承。

StmtNode 继承于 Node, 由 CompoundStmt、SeqNode、DeclStmt、BlankStmt、IfStmt、IfElstStmt、WhileStmt、BreakStmt、ContinueStmt、ReturnStmt、AssignStmt、ExprStmt、FunctionDef、funcParms、funcParm 继承。

2. Id

Id 为左值中使用。对于 Id，需要特别注意的是其第二个参数是 ExprNode* 类型的数组维度信息，在初始化时就要把数组与其他类型区分开讨论。其初始化函数如下:

Id

```
1 Id(SymbolEntry *se, ExprNode *index = nullptr) : ExprNode(se), index(index){
2     this->type = se->getType();
3     if (se->getType()->isArray()){
4         std::vector<int> indexs = ((ArrayType *)se->getType())->getIndexs();
```

```
        SymbolEntry *temp;
        ExprNode *expr = index;
        while (expr){
            expr = (ExprNode *)expr->getNext();
            indexs.erase(indexs.begin());
        }
        if (indexs.size() <= 0){
            // 如果索引和数组定义时候的维度一致，是引用某个数组元素
            if (((ArrayType *)se->getType())->getBaseType()->isInt()){
                this->type = TypeSystem::intType;
                temp = new TemporarySymbolEntry(TypeSystem::intType,
                    SymbolTable::getLabel());
            }
            else if (((ArrayType *)se->getType())->getBaseType()->isFloat()){
                this->type = TypeSystem::floatType;
                temp = new TemporarySymbolEntry(TypeSystem::floatType,
                    SymbolTable::getLabel());
            }
        }
        else
        {// 索引个数小于数组定义时候的维度，应该作为函数参数传递，传递的是一
            个指针
            isPointer = true;
            indexs.erase(indexs.begin());
            if (((ArrayType *)se->getType())->getBaseType()->isInt())
                this->type = new PointerType(new ArrayType(indexs, TypeSystem
                    ::intType));
            else if (((ArrayType *)se->getType())->getBaseType()->isFloat())
                this->type = new PointerType(new ArrayType(indexs, TypeSystem
                    ::floatType));
            temp = new TemporarySymbolEntry(this->type, SymbolTable::getLabel
                ());
        }
        dst = new Operand(temp);
    }
    else{
        this->type = se->getType();
        SymbolEntry *temp = new TemporarySymbolEntry(this->type, SymbolTable
            ::getLabel());
        dst = new Operand(temp);
    }
};
```

3. 双目表达式

双目表达式有两个操作数，其操作类型有：加、减、乘、除、取模、与、或、小于、小于等于、大于、大于等于、等于、不等于。

BinaryExpr

```
class BinaryExpr : public ExprNode {
private:
    int op;
    ExprNode *expr1, *expr2;

public:
    enum {ADD, SUB, MUL, DIV, MOD, AND, OR, LESS, LESSEQUAL, GREATER,
        GREATEREQUAL, EQUAL, NOTEQUAL};
    BinaryExpr(SymbolEntry* se, int op, ExprNode* expr1, ExprNode* expr2);
    void output(int level);
    int getValue();
    void genCode();
};
```

4. DeclStmt

在声明时，需要记录变量的 ID；而对于数组的声明，不仅要记录数组的 ID，还需要记录相关维度的信息。

声明、定义

```
class DeclStmt : public StmtNode {
private:
    Id* id;
    ExprNode* expr;
    ExprNode **exprArray; // 当Id是数组时，用来存数据的初始值
public:
    DeclStmt(Id* id, ExprNode* expr = nullptr) : id(id) {
        this->exprArray = nullptr;
        if (expr)
        {
            if (id->getType()->isFloat() && expr->getType()->isInt())
                this->expr = new ImplictCastExpr(expr, ImplictCastExpr::ITF);
            if (id->getType()->isInt() && expr->getType()->isFloat())
                this->expr = new ImplictCastExpr(expr, ImplictCastExpr::FTI);
        }
    };
    void output(int level);
    void genCode();
    void setInitArray(ExprNode **exprArray) ;
    Id* getId() { return id; };
};
```

5. while/continue/break

对于 WhileStmt，其继承于 StmtNode 类，私有变量 cond 记录条件语句，stmt 记录 then 语句。除此之外还需要两个 BasicBlock 供插入。在初始化时，需要把 cond 语句转为 bool 类型，

因此使用隐式转换。类定义代码如下所示:

while

```
class WhileStmt : public StmtNode {
private:
    ExprNode* cond;
    StmtNode* stmt;
    BasicBlock* cond_bb;
    BasicBlock* end_bb;
public:
    WhileStmt(ExprNode* cond, StmtNode* stmt=nullptr) : cond(cond), stmt(stmt
        ) {
        if (cond->getType()->isInt() && cond->getType()->getSize() == 32) {
            ImplictCastExpr* temp = new ImplictCastExpr(cond);
            this->cond = temp;
        }
    };
    void setStmt(StmtNode* stmt){this->stmt = stmt;};
    void output(int level);
    void genCode();
    BasicBlock* get_cond_bb(){return this->cond_bb;};
    BasicBlock* get_end_bb(){return this->end_bb;};
};
```

其 output 函数较为简单，只要将 cond 与 stmt 分别进行输出即可。

对于 continue/break 语句，需要有一个 StmtNode* 类型的私有变量来记录所在的 while 语句，在初始化时进行设置。其类定义较为简单，此处不再展示。

6. 赋值语句

对于赋值语句，其私有变量为两个元素：左值与右值表达式。其初始化较为简单，只需要将左值与右值的类型分别初始化即可。

7. 函数定义

函数定义也与上文的赋值语句类似，在初始化时记录 DeclStmt* 类型的 decl 和 StmtNode* 类型的 stmt，并记录 parser 传入的符号表项即可。

(六) 实验效果

这里着重展示数组的实验效果:

数组示例

```
int main(){
    const int a[4][2] = {{1, 2}, {3, 4}, {}, 7};
    const int N = 3;
    int b[4][2] = {};
    int c[4][2] = {1, 2, 3, 4, 5, 6, 7, 8};
    int d[N + 1][2] = {1, 2, {3}, {5}, a[3][0], 8};
```

```
    int e[4][2][1] = {{d[2][1], {c[2][1]}}, {3, 4}, {5, 6}, {7, 8}};
    return e[3][1][0] + e[0][0][0] + e[0][1][0] + d[3][0];
}
```

```
FunctionDefine function name: main, type: int()
    CompoundStmt
        Sequence
            Sequence
                Sequence
                    Sequence
                        Sequence
                            Sequence
                                ConstDecl_stmt
                                    Array_assign
                                        Decl_Array
                                            Decl_Array
                                                Id  name: a scope: 1    type: const
                                                IntegerLiteral  value: 4    type: int
                                            IntegerLiteral  value: 2    type: int
                                        ArrayInitVals
                                            ArrayInitVals
                                                ArrayInitVals
                                                    ArrayInitVals
                                                        Array_Initial
                                                            ArrayInitVals
                                                                ArrayInitVals
                                                                    IntegerLiteral  value: 1    type: int
                                                                IntegerLiteral  value: 2    type: int
                                                    Array_Initial
                                                        ArrayInitVals
                                                            ArrayInitVals
                                                                IntegerLiteral  value: 3    type: int
                                                                IntegerLiteral  value: 4    type: int
                                                Array_Initial
                                            IntegerLiteral  value: 7    type: int
                                ConstDecl_stmt
                                    Decl_Assign
                                        Id  name: N scope: 1    type: const
                                        IntegerLiteral  value: 3    type: int
                            varDecl_stmt
                                Array_assign
                                    Decl_Array
                                        Decl_Array
                                            Id  name: b scope: 1    type: int
                                            IntegerLiteral  value: 4    type: int
                                        IntegerLiteral  value: 2    type: int
                        varDecl_stmt
                            Array_assign
                                Decl_Array
                                    Decl_Array
                                        Id  name: c scope: 1    type: int
                                        IntegerLiteral  value: 4    type: int
                                    IntegerLiteral  value: 2    type: int
                                ArrayInitVals
                                    ArrayInitVals
```

图 6: ArrayResult

在 lab5 时, 我们的代码能够识别所有的基础要求, 并完成了区别常量与变量作用域、break/-continue 语句、非叶函数、数组的声明与初始化等级别而的要求。break 语句的识别效果如下所示:

break 示例

```
//test break
int main(){
    int i;
    i = 0;
    int sum;
    sum = 0;
```

```
    while(i < 100){
        if(i == 50){
            break;
        }
        sum = sum + i;
        i = i + 1;
    }
    return sum;
}
```


图 7: lab5$_{R}esult$

四、 类型检查

类型检查是编译过程的重要一步，以确保操作对象与操作符相匹配。每一个表达式都有相关联的类型，如关系运算表达式的类型为布尔型，而计算表达式的类型一般为整型或浮点型等。类型检查的目的在于找出源代码中不符合类型表达式规定的代码，在最终的代码生成之前报错，使

得程序员根据错误信息对源代码进行修正。

(一) 分工

在此部分，我负责的工作有:

- 数值运算表达式运算数类型是否正确 (可能导致此错误的情况有返回值为 void 的函数定义结果参与了某表达式计算)
- Assign 语句的类型检查
- 条件判断的类型检查

(二) 数值运算类型检查

在双目表达式的运算中，运算符左右两边的表达式类型须匹配，且左右两边都不能为空（否则就是单目表达式，或产生错误)。可以允许的情况有:

- 左右两边类型完全相同的情况
- int 型与 constant 型匹配
- int 型与 bool 型匹配
- bool 型与 constant 型匹配
- 函数返回值类型为 int 型与 int/bool 进行匹配

除此之外，其余的情况都会产生错误。具体实现代码如下所示:

BinaryExpr TypeCheck

```
void BinaryExpr::typeCheck()
{
    // 数值运算表达式运算数类型是否正确 (可能导致此错误的情况有返回值为 void
       的函数定义结果参与了某表达式计算)
    Type *type1 = expr1->getSymPtr()->getType();
    Type *type2 = expr2->getSymPtr()->getType();
    if(type2->toStr()=="void()" || type1->toStr()=="void()")
    {
        fprintf(stderr, "type %s 为空 不能赋值",
        type1->toStr().c_str());
        exit(EXIT_FAILURE);
        return;
    }
    if(type1 != type2)
    {
        if(
        !(
        (type1->toStr()=="i32" && type2->toStr()=="constant")
        || (type1->toStr()=="constant" && type2->toStr()=="i32")
        //int 至 bool 类型隐式转换
        || (type1->toStr()=="i1" && type2->toStr()=="constant")
```

```
        || (type1->toStr()=="constant" && type2->toStr()=="i1")

        || (type1->toStr()=="i32" && type2->toStr()=="i1")
        || (type1->toStr()=="i1" && type2->toStr()=="i32")

        || (type1->toStr()=="i32()" && type2->toStr()=="i32")//FunctionType
        || (type1->toStr()=="i32" && type2->toStr()=="i32()")

        || (type1->toStr()=="i32()" && type2->toStr()=="i1")
        || (type1->toStr()=="i1" && type2->toStr()=="i32()")
        //完全匹配情况
        ||(type1->toStr()=="i32" && type2->toStr()=="i32")
        ||(type1->toStr()=="i1" && type2->toStr()=="i1")
        ||(type1->toStr()=="i32()" && type2->toStr()=="i32()")
        ||(type1->toStr()=="constant" && type2->toStr()=="constant")
        ))
        {
            fprintf(stderr, "type %s and %s mismatch in line xx\n",
            type1->toStr().c_str(), type2->toStr().c_str());
            exit(EXIT_FAILURE);
            return;
        }
    }
    symbolEntry->setType(type1);
    expr1->typeCheck();
    expr2->typeCheck();
}
```

(三) Assign 语句类型检查

对于 Assign 语句，左值和右值的表达式类型必须匹配，可以允许的情况有：

- 左右两边类型完全相同的情况
- int 型与 constant 型匹配
- int 型与 bool 型匹配
- bool 型与 constant 型匹配
- 函数返回值类型为 int 型与 int/bool 进行匹配

代码主体部分与双目表达式的 TypeCheck 基本相同，此处不再赘述。

(四) 条件判断类型检查

对于 IfElse 语句的检查，首先条件语句必须是 INT 类型，并将其设置为 boolType；若是函数类型，则需要获取函数的返回值，并重新进行判断，最后设置 boolType。最后要对 thenStmt 和 elseStmt 逐一进行检查。而对于 If 语句，只是减少了 elseStmt 的检查，基于与 IfElseStmt 基本相同。

IfElseStmt TypeCheck

```
void IfElseStmt::typeCheck()//把cond的类型设置为bool
{
    cond->typeCheck();
    //1判断cond的类型 必须要是INT的(bool也是一种INT)
    if(cond->getSymPtr()->getType()->isInt()==1
    || cond->getSymPtr()->getType()->isConst()==1)
    {
        cond->getSymPtr()->setType(TypeSystem::boolType);
    }
    else{
        if(cond->getSymPtr()->getType()->isFunc()==1)//函数类型
        {
            FunctionType* temp=static_cast<FunctionType*>(cond->getSymPtr()->
                getType());
            Type*tmp=temp->getRetType();//获取返回值
            if(tmp->isInt()==1 || tmp->isConst()==1)
            {
                cond->getSymPtr()->setType(TypeSystem::boolType);
                thenStmt->typeCheck();
                elseStmt->typeCheck();
                return;
            }
        }
        fprintf(stderr,"ifelse语句条件类型不正确\n");
        exit(EXIT_FAILURE);
        return;
    }
    thenStmt->typeCheck();
    elseStmt->typeCheck();
}
```

五、 中间代码生成

在编译器的分析-综合模型中，前端对源程序进行分析并产生中间表示，后端在此基础上生成目标代码。在理想情况下，和源语言相关的细节在前端分析中处理，而关于目标机器的细节则在后端处理。基于适当定义的中间表示形式，可以把针对源语言的前端和针对目标机器的后端组合起来。本部分内容涉及中间代码表示、静态类型检查和中间代码生成。

为什么要生成中间代码? 快速编译程序直接生成目标代码，没有将中间代码翻译成目标代码的额外开销。但是为了使编译程序结构在逻辑上更为简单明确，常采用中间代码，并且可以在中间代码一级进行优化工作使得代码优化比较容易实现。

本部分的方法可以用于多种中间表示，包括抽象语法树和三地址代码。之所以命名为三地址代码，主要是因为这些指令的一般形式 x = y op z 具有三个地址：两个运算分量 y 和 z，以及结果变量 x。

中间代码生成的总体思路是对抽象语法树作一次遍历，遍历的过程中需要根据综合属性和继承属性来生成各结点的中间代码，在生成完根结点的中间代码后便得到了最终结果。

在本次实验中，我们将生成 llvm IR。

(一) 分工

在此部分,我将负责:Id/DeclStmt/ReturnStmt/ExprStmt/ContinueStmt/BreakStmt/WhileStmt/CallExpr/Unar
等部分的中间代码生成工作。

(二) Id

Id 的中间代码生成主要分为两种：Int 型和 Array 型。若为 Int 型，直接生成一条 Load 指令即可。若为 Array 型，则需要区分其是否作为函数参数。若是函数参数，例如 a[]，则该数组的类型为指针类型，与赋值语句左值的数组类型（如 a[2][3]）区分处理。

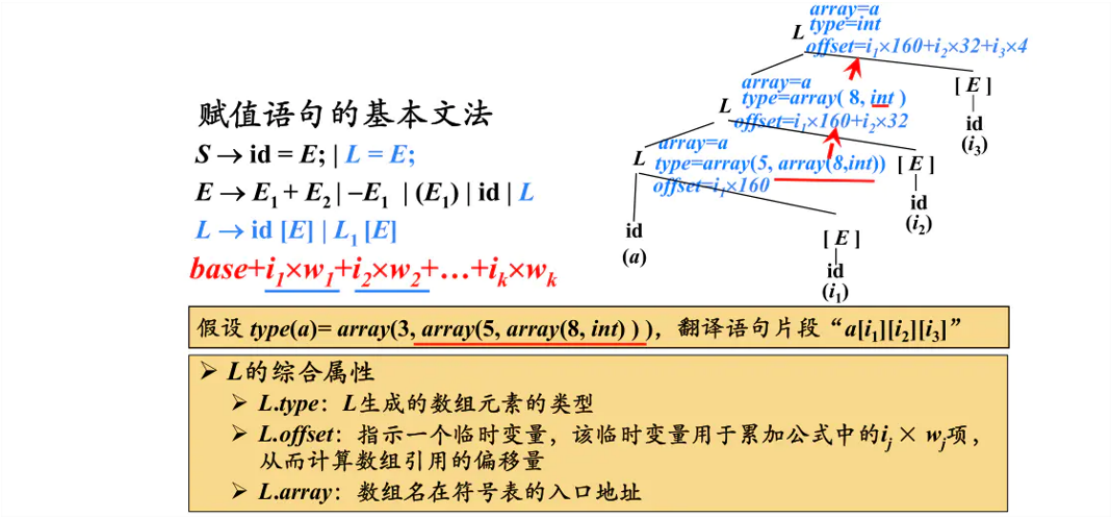

图 8: Array

Id::genCode() 对数组的处理

```
1   if (se->getType()->isArray()){
2       ExprNode *tmp = index;//数组下标
3       std::vector<Operand *> offs;
4       while (tmp){
5           tmp->genCode();
6           offs.push_back(tmp->getOperand());
7           tmp = (ExprNode *)tmp->getNext();
8       }
9       if (this->isPointer){
10          // 数组作为函数参数传递指针，取数组指针
11          // 生成一条gep指令返回
12          offs.push_back(new Operand(new ConstantSymbolEntry(TypeSystem::
                intType, 0)));
13          new GepInstruction(dst, addr, offs, now_bb);
14          return;
15      }
16      if (((ArrayType *)se->getType())->getBaseType()->isInt())
```

```
            addr = new Operand(new TemporarySymbolEntry(new PointerType(
                TypeSystem::intType), SymbolTable::getLabel()));
        else if (((ArrayType *)se->getType())->getBaseType()->isFloat())
            addr = new Operand(new TemporarySymbolEntry(new PointerType(
                TypeSystem::floatType), SymbolTable::getLabel()));
        new GepInstruction(addr, se->getAddr(), offs, now_bb);
    }
    else if (se->getType()->isPtr()){//指针
        ExprNode *tmp = index;
        if (tmp == nullptr){
            // 如果数组标识符没有索引，他应该是作为参数传递的，取数组指针就行
            new LoadInstruction(dst, addr, now_bb);
            return;
        }
        Operand *base = new Operand(new TemporarySymbolEntry(((PointerType *)
            (addr->getType()))->getType(), SymbolTable::getLabel()));
        new LoadInstruction(base, addr, now_bb);
        std::vector<Operand *> offs;
        while (tmp){
            tmp->genCode();
            offs.push_back(tmp->getOperand());
            tmp = (ExprNode *)tmp->getNext();
        }
        if (((ArrayType *)((PointerType *)se->getType())->getType())->
            getBaseType()->isInt())
            addr = new Operand(new TemporarySymbolEntry(new PointerType(
                TypeSystem::intType), SymbolTable::getLabel()));
        else if (((ArrayType *)((PointerType *)se->getType())->getType())->
            getBaseType()->isFloat())
            addr = new Operand(new TemporarySymbolEntry(new PointerType(
                TypeSystem::floatType), SymbolTable::getLabel()));
        new GepInstruction(addr, base, offs, now_bb, true);
    }
```

(三) 声明语句

对于声明语句，需要判断其是全局变量还是局部变量或参数。

1. 若为全局变量，则需要插入进 unit 中，单独进行处理。如果是全局变量且是数组，则用传递进来的数组初始值初始化数组。

2. 对于局部变量或参数，要先分配空间 (使用 AllocaInstruction)，且分配指令应该插入进该语句块的开头。如果有初始值，需要插入 store 指令。如果是数组同样也需要初始化，具体方法为递归数组的 index，从最高位一直到最低维，通过计算数组低地址来寻址，并生成 store 指令。

3. 如果是参数，则还需要得到其地址，以便后续 store 指令。参数是需要使用 getNext() 得到下一个参数，并再产生中间代码。

(四) 赋值语句

对于赋值语句，首先需要插入基本块，然后根据 expr 产生中间代码；之后需要分两种情况进行讨论：若为 int 型，直接得到其地址；若为数组，则将先计算其地址，最后进行 StoreInstruction，将 expr 的值存入得到的地址中。

Assign

```
void AssignStmt::genCode() {
    if (this->expr)
        this->expr->genCode();
    Operand* addr = nullptr;
    if (lval->getOriginType() == TypeSystem::intType || lval->getOriginType()
        == TypeSystem::constIntType)
        addr = dynamic_cast<IdentifierSymbolEntry*>(lval->getSymbolEntry())->
            getAddr();
    else if (lval->getOriginType()->isArray()) {
        dynamic_cast<Id*>(lval)->setLeft();
        lval->genCode();
        addr = lval->getOperand();
    }
    new StoreInstruction(addr, expr->getOperand(), builder->getInsertBB());
}
```

(五) Return 语句

Return 语句的处理较为简单，但需要判断是否有返回值。若有返回值，则返回值进行中间代码生成，最终使用 RetInstruction 即可。

ReturnStmt

```
void ReturnStmt::genCode() {
    // Todo
    BasicBlock* bb = builder->getInsertBB();
    Operand* src = nullptr;
    if (retValue) {
        retValue->genCode();
        src = retValue->getOperand();
    }
    else{
        src=nullptr;
    }
    new RetInstruction(src, bb);
}
```

(六) Continue/Break 语句

continue/break 语句较为相似，此处以 continue 语句为例。需要取当前所在 whileStmt 节点的 cond 块，并无条件跳转到那里，并插入 continue 之后的语句块。

ContinueStmt

```
void ContinueStmt::genCode() {
    Function* func = builder->getInsertBB()->getParent();
    BasicBlock* bb = builder->getInsertBB();
    new UncondBrInstruction(((WhileStmt*)whileStmt)->get_cond_bb(), bb);
    BasicBlock* continue_next_bb = new BasicBlock(func);
    builder->setInsertBB(continue_next_bb);
}
```

(七) While 语句

- 用$M_1.quad$来记录$while$循环的的第一条指令，用于回填$S_1.nextlist$
- 用$M_2.quad$来记录S_1的第一条指令，用于回填$B.truelist$

$S \rightarrow$ **while** M_1 B **do** M_2 S_1

{

backpatch(S_1.nextlist, M_1.quad);

backpatch(B.truelist, M_2.quad);

S.nextlist = B.falselist;

gen('goto' M_1.quad);

}

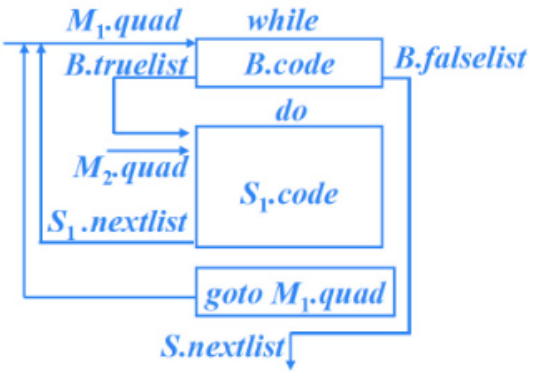

图 9: While 回填

需要创建出 cond_bb/then_bb 和 end_bb 两个基本块, then_bb 是 WhileStmt 结点生成的指令的插入位置，end_bb 为 while 语句后续的结点生成的中间代码的插入位置。需要生成 cond 结点的中间代码，cond 为真时将跳转到基本块 then_bb，cond 为假时将跳转到基本块 end_bb，进行回填。之后设置插入点为基本块 then_bb，然后生成 thenStmt 结点的中间代码。因为生成 thenStmt 结点中间代码的过程中可能改变指令的插入点，因此需要更新插入点，然后再次跳转到 cond_bb 去执行。最后设置后续指令的插入点为 end_bb。

(八) 函数定义

函数定义较为简单，首先设置该函数的插入点，然后 decl 和 stmt 分别进行 genCode()。之后为了建立控制流，我们需要设置前驱和后继节点：首先我们移除基本块中间的跳转指令，并获取该块的最后一条指令。

- 如果最后一条指令是条件指令, 则分别得到其 truebranch 和 falsebranch。若 truebranch 为空，则插入 RetInstruction(nullptr, truebranch)，反之则插入 RetInstruction(nullptr, falsebranch)。
- 如果为非条件指令，则获取跳转的 dst，如果 dst 为空，则根据函数返回类型插入 ret 指令。

- 如果最后一条语句不是返回以及跳转，且返回值类型为 void，则插入 ret void。

（九） 单目表达式

单目表达式有三种情况：ADD/NOT/SUB。它们都需要先插入基本块，在进行后一步操作。

- ADD: 此情况较为简单，直接调用 UnaryInstruction 即可。
- NOT: 首先判断如果是 int 型或者 float 型，与 0 进行比较，并存储结果；最后将结果与源操作数进行异或。
- SUB: 和 ADD 的情况相同，直接调用 BinaryInstruction

六、 目标代码生成

目标代码生成是编译程序的最后一个工作阶段。其任务是把先行阶段所产生的中间代码转换为相应的目标代码。在中间代码生成后，对中间代码进行自顶向下遍历，从而生成目标代码。其思路大致与中间代码相同，只是指令和操作数有所不同。在本次实验中，我们将生成汇编代码。

目标机器的主要指令有：

1. 加载指令：$LD\ \ dst, addr$
   - $LD\ \ r, x$
   - $LD\ \ r1, r2$
2. 保存指令$ST\ \ x, r$
3. 运算指令$OP\ \ dst, src1, src2$
4. 无条件跳转指令$BR\ \ L$
5. 条件跳转指令$Bcond\ \ r, L$
   - 例：$BLTZ\ \ r, L$

图 10: 主要指令

寻址模式大致如下所示：

1. 变量名a
   - 例: $LD\ \ R1, a$, 即$R1 = contents(a)$
2. $a(r)$
   - a是一个变量, r是一个寄存器
   - 例: $LD\ \ R1, a(R2)$, 即$R1 = contents(a + contents(R2))$
3. $c(r)$
   - c是一个整数
   - 例: $LD\ \ R1, 100(R2)$, 即$R1 = contents(contents(R2) + 100)$
4. $*r$
   - 在寄存器r的内容所表示的位置上存放的内存位置
   - 例: $LD\ \ R1, *R2$, 即$R1 = conents(contents(contents(R2)))$
5. $*c(r)$
   - 在寄存器r中内容加上c后所表示的位置上存放的内存位置
   - 例: $LD\ \ R1, *100(R2)$, 即$R1 = conents(contents(contents(R2) + 100))$
6. $\#c$
   - 例: $LD\ \ R1, \#100$, 即$R1 = 100$

图 11: 寻址模式

(一) 分工

在本部分，我将负责:

- Instruction: StoreInstruction 、BranchInstruction (Cond+Uncond+Ret) 、ZextInstruction /XorInstruction、GepInstruction
- LinearScan: linearScanRegisterAllocation、genSpillCode、expireOldIntervals、spillAtInterval
- MachineCode: BranchMInstruction、CmpMInstruction、MachineUnit 相关函数、Uxtb-MInstruction、GlobalMInstruction

(二) Instructions

1. StoreInstruction

参照框架代码进行 Store 指令的完善。访存指令分为以下三种情况:

1. 加载一个全局变量或常量:
   需要生成两条加载指令，首先在全局的地址标签中将其地址加载到寄存器中，之后再从该地址中加载出其实际的值。
2. 加载一个栈中临时变量:
   由于 AllocaInstruction 指令中，已经为所有的局部变量申请了栈内空间，并将其相对 FP 寄存器的栈内偏移存在了符号表中。因此只要以 FP 为基址寄存器，根据其栈内偏移生成一条加载指令即可。

3. 加载一个数组元素:
   数组元素的地址存放在一个临时变量中，只需生成一条加载指令即可。

2. BranchInstruction

1. UncondBrInstruction
   只需要生成一条无条件跳转指令。对于跳转目的操作数的生成，只需要调用 genMachineLabel() 函数即可，参数为目的基本块号;

2. CondBrInstruction
   对于 CondBrInstruction，该指令一定位于 CmpInstruction 之后。对 CondBrInstruction，需要取前一条 CmpInstruction 的条件码，从而在遇到 CondBrInstruction 时生成对应的条件跳转指令跳转到 True Branch，之后需要生成一条无条件跳转指令跳转到 False Branch。

3. RetInstruction
   首先，当函数有返回值时，需要生成 MOV 指令，将返回值保存在 R0 寄存器中；其次，需要生成 ADD 指令使 sp 指针移动来恢复栈帧，（如果该函数有 Callee saved 寄存器，还需要生成 POP 指令恢复这些寄存器），生成 POP 指令恢复 FP 寄存器；最后再生成跳转指令跳转到 lr 寄存器，即返回地址。

无条件跳转指令与条件跳转指令较为简单，此处不再展示。此处只展示返回指令:

RetInstruction

```
void RetInstruction::genMachineCode(AsmBuilder* builder) {
    auto cur_block = builder->getBlock();
    // 需要返回值的情况
    if (operands.size() > 0){
        auto ret_value = genMachineOperand(operands[0]);
        if (ret_value->isImm())
            ret_value = new MachineOperand(*immToVReg(ret_value, cur_bb));
        if (operands[0]->getType()->isFloat()){
            if (ret_value->isFReg())
                cur_bb->InsertInst(new MovMInstruction(cur_bb,
                    MovMInstruction::VMOV32, genMachineReg(0, true),
                    ret_value));
            else // 同样的，这种情况是返回立即数，把立即数放到r寄存器里了
                cur_bb->InsertInst(new MovMInstruction(cur_bb,
                    MovMInstruction::VMOV, genMachineReg(0, true), ret_value)
                    );
        }
        else
            cur_bb->InsertInst(new MovMInstruction(cur_bb, MovMInstruction::
                MOV, genMachineReg(0), ret_value));
    }
    //恢复栈空间
    auto cur_func = builder->getFunction();
    auto sp = new MachineOperand(MachineOperand::REG, 13);
    auto size = new MachineOperand(MachineOperand::IMM, cur_func->AllocSpace
        (0));//当前栈空间
```

```
    auto cur_inst = new BinaryMInstruction(cur_block, BinaryMInstruction::ADD
        , sp, sp, size);//sp指针移动 恢复栈空间
    cur_block->InsertInst(cur_inst);
    //跳转到lr寄存器(返回地址)
    auto lr = new MachineOperand(MachineOperand::REG, 14);
    auto cur_inst2 = new BranchMInstruction(cur_block, BranchMInstruction::BX
        , lr);//生成跳转链接指令
    cur_block->InsertInst(cur_inst2);
}
```

3. ZextInstruction

在 llvm 语法中，存在 zext...to 指令。

- 语法：<result> = zext <ty> <value> to <ty2> ; yields ty2
- 概述：zext 指令把操作数使用 0 扩展为 ty2 类型。
- 参数：zext 指令要转换的数值，必须是整型，目标类型也必须是整型。value 的位的大小必须比目标类型 ty2 的位的大小小。
- 语义：zext 指令用 0 填充 value 的高阶位，直到达到目标类型 ty2 的大小。

此处我们将其简化为使 i1 扩展为 i32 的 0 扩展指令。其主要思路为使用 mov 指令，将 bool 值写入 dst 中。

另一种思路：

$cmp\ \ src\,1\ \ \ moveqdst\ temp\_reg1; (temp\_reg1 = newMachineOperand(MachineOperand :: IMM, 1); movnedst, te$

4. XorInstruction

异或指令，用于 NOT 取反的情况：如果 a 是一个 bool 类型值，我们想得到!a，则需要 a xor 1.

XorInstruction

```
void XorInstruction::genMachineCode(AsmBuilder* builder) {
    auto cur_block = builder->getBlock();
    auto dst = genMachineOperand(operands[0]);
    auto trueOperand = genMachineImm(1);
    auto falseOperand = genMachineImm(0);
    auto cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
         trueOperand, MachineInstruction::EQ);
    cur_block->InsertInst(cur_inst);
    cur_inst = new MovMInstruction(cur_block, MovMInstruction::MOV, dst,
        falseOperand, MachineInstruction::NE);
    cur_block->InsertInst(cur_inst);
}
```

5. GepInstruction

Gep 指令为 GetElementPtr（GEP）。在 llvm 中，GEP 非常类似于数组 c 的索引和字段选择，在这里用作数组的寻址。主要是通过遍历数组当前的维度信息得到数组元素相对于栈帧的偏移，方便后续的操作。

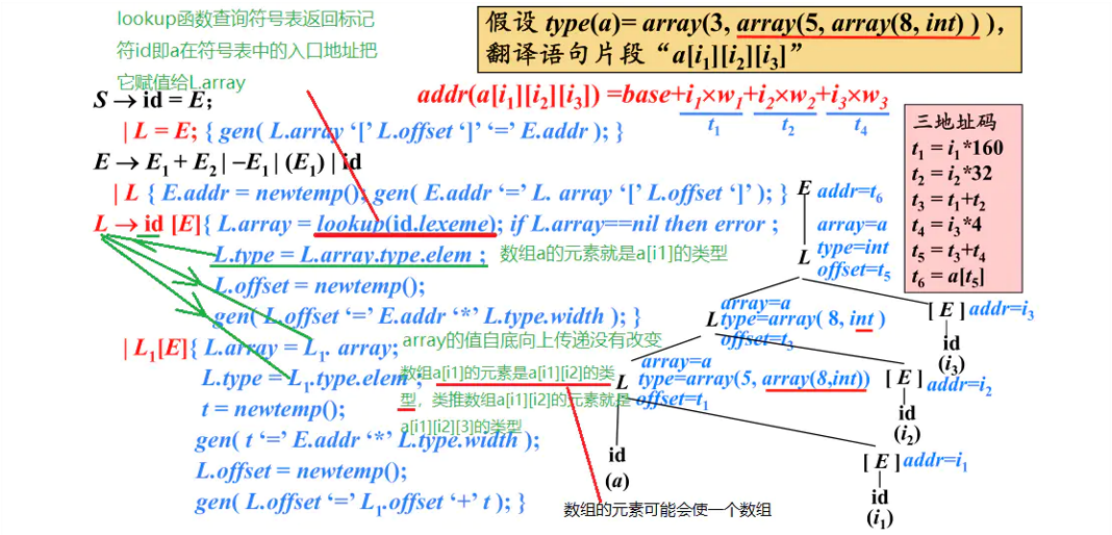

图 12: Array

在 GepInstruction 中我们定义了一个 bool 类型的值 thisType，用来标识该数组是不是传参这种形式的数组指针，为 true 则是，否则该数组就是局部变量或者全局变量。如果是全局变量/局部变量，则将他们的地址放入一个临时寄存器中，用一个 load 指令从 operand[1] 中取出 src，然后将其加载到 base 当中；而对于函数参数，operand[1] 就存有它的地址。

- 对于全局变量，则将数组的地址标签加载到寄存器中。
- 如果是局部变量, 则是将数组的首地址相对于函数栈帧的偏移量加载到寄存器中。这里需要判断偏移量是否是合法的立即数且大于-255: 如果是, 则直接插入一条 BinaryMInstruction, base = fp + offset; 如果不是，则需要将 offset 放入到一个虚拟寄存器中再计算。

之后，通过遍历数组的维度信息和数组元素的维度信息，获取**数组元素**相对于函数首地址的偏移量。通过 BinaryMInstruction::ADD 指令得到该数组元素相对于函数栈帧的偏移。

最后使用一条 MovMInstruction 指令将或得到的数组元素加载到目的寄存器中。

(三) LinearScan

线性扫描寄存器算法需要单趟遍历每个活跃区间, 为期分配物理寄存器。参考论文为《Linear Scan Register Allocation》[1]。

大致分为以下三个步骤:

1. 计算出每个虚拟寄存器的活跃区间（Live Interval），根据活跃区间的起点进行排序。
2. 从前到后进行扫描活跃区间列表。
3. 进行分配并处理 spill。

线性扫描伪代码

```
LinearScanRegisterAllocation:
    active := {}
    for i in live interval in order of increasing start point
        ExpireOldIntervals(i)
        if length(avtive) == R
            SpillAtInterval(i)
        else
            register[i] := a regsiter removed from pool of free registers
            add i to active, sorted by increasing end point
ExpireOldInterval(i)
    for interval j in active, in order of increaing end point
        if endpoint[j] >= startpoint[i]
            return
        remove j from active
        add register[j] to pool of free registers
SpillAtInterval(i)
    spill := last interval in active
    if endpoint[spill] > endpoint[i]
        register[i] := register[spill]
        location[spill] := new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] := new stack location
```

1. 计算活跃区间

首先对活跃区间进行计算。首先需要对指令进行一个编号，一个简单的方法是对指令进行深度优先标号（Deep First Numbering, DFN）。在进行了编号之后，通过活跃分析（live analysis）确定每个虚拟寄存器在程序的那些位置是活跃的，然后选出编号最大（j）和最小（i）的活跃点，组成活跃区间 [i,j]。v 的活跃区间只需要保证在这个区间之外 v 不会活跃就行了，不需要保证 v 在这个区间内是处处活跃的。

如果两个虚拟变量的活跃区间发生了重叠，那么可以称这两个虚拟变量存在冲突，产生了冲突的两个变量不可以使用同一个寄存器。

这里需要提到的是，对指令的编号会影响到代码生成的质量，因为在有些编号下活跃区间会更加短，更短的活跃区间会降低冲突的可能性 [3]。比如下面的例子中，then 分支使用了 v，但是 else 分支并没有使用 v，同时 then 中的使用是最后一次使用，给 then 先编号会得到??的结果，而给 else 先编号则会得到??的结果。

很容易发现，一些活跃区间会出现类似于??的空洞，如果另外一个活跃区间正好在这个空洞里面，就不会发生冲突，不过我们这里并不利用这种空洞，而是简单地将活跃区间从头到尾当成一个整体。

2. 对活跃区间进行线性扫描

接下来对活跃区间进行扫描，在进行扫描之前首先对所有的活跃区间进行排序，根据区间开始点从低到高。排序后的活跃区间很适合进行线性扫描，对活跃区间的迭代的顺序和标号顺序是

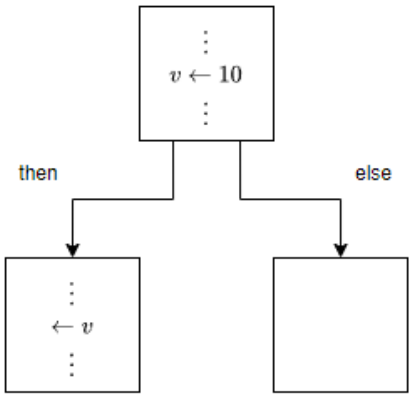

图 13: World Map

图 14: 活跃区间分析

一样的。

下面开始，假设一共有 R 个可供分配的寄存器。有一个链表 active，用于存放覆盖了当前位置的活跃区间，active 根据每个区间的终点从高到低进行排列，反过来也行，这样有利于减少查找的次数。

每次向前扫描可以遇到一个新的活跃区间 I_{new}，用 I_{new} 的起点 i 和 $active$ 链表中的每个区间 I_{old} 的终点 j 进行对比，如果出现了 $j\ i$，那么就将 I_{old} 从 $active$ 中移除，这个时候可以认为 I_{old} 以及被完全处理了，不需要再次进行考虑。

再进行了对 $active$ 的更新之后，如果 $size(active)$ R 那么此时必须要将链表中的某个 I_{old} 或者是 I_{new} 进行 spill，可以使用多种方式决定到底需要将那个区间进行 spill，这里给出一个 heuristic，就是使用结束位置最后的一个区间进行 spill，这么做的原因是，因为结束的位置最后，所以会延续很长因此可能会导致更多的区间被溢出，如果使用了区间的终点从高到低进行排列，那么 $active$ 中最后结束的就是第一个。除此之外还可以通过区间的使用频率的等信息进行判断。

3. 进行分配和处理 spill

一个区间进行了 spill 之后，就不再占有任何一个寄存器了，但是在使用点依旧需要对寄存器进行使用，所以在使用点可以使用任意一个寄存器，临时寄存器的选取只要保证即可不和指令内其他寄存器重复即可。此时，需要为虚拟变量在栈上分配一个空间，在对虚拟变量进行赋值之后要将值回存到栈上的空间。由于需要占用其他寄存器，那么可能会破坏寄存器中的值，所以还需要另外一个栈上的空间能够存放被临时占用的寄存器的值。

具体步骤为：

1. 为其在栈内分配空间，获取当前在栈内相对 FP 的偏移；
2. 遍历其 USE 指令的列表，在 USE 指令前插入 LoadMInstruction，将其从栈内加载到目前的虚拟寄存器中。此时还需要判断数据地址是否超出寻址空间，若超出则不能直接加载，需要分两步；
3. 遍历其 DEF 指令的列表，在 DEF 指令后插入 StoreMInstruction，将其从目前的虚拟寄存器中存到栈内。这里同样需要考虑寻址空间的问题。

参考文献 [?] [?]

多行公式

$$a+b=a+b \tag{22}$$

$$\frac{a+b}{a-b} \tag{23}$$

行内公式：$\sum_{i=1}^{N}$
超链接 YouTube
带标号枚举

1. 1
2. 2

不带标号枚举

- 1
- 2

切换字体大小

七、 代码链接

切换字体大小 Gitlab

参考文献

[1] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, sep 1999.