南開大學
Nankai University

计算机学院

编译原理大作业报告

从零开始的编译器冒险

姓名：朱璟钰

学号：2013216

专业：计算机科学与技术

2023 年 1 月 15 日

摘 要

最终实现的编译器可以通过 level1level2 **全部**样例，除基本要求外，实现了函数、语句块、**数组**、**浮点**等 SysY 特性，并完成了代码优化，包括：**寄存器分配优化方法**（主要参考 Second-chance allocation）、**mem2reg**、**公共子表达式删除**、**常量折叠**及一些细节上的**代码优化**。

关键字：编译原理; 公共子表达式消除; 寄存器分配优化

目录

1 指路牌：概述

1.1 特性清单及分工

编译器特性实现清单如下：

主要特性	内容	本人分工
数据类型	int	共同完成
	float	
基础 track	变/常量声明及初始化	主要负责
	赋值、语句块、if、while、return	主要负责
	算术运算、关系运算和逻辑运算	
	注释	主要负责
	输入输出	
进阶 track	函数、语句块	主要负责
	数组	主要负责
	寄存器分配优化方法	主要负责
	公共子表达式删除	主要负责
	mem2reg	
	常量折叠	

表 1: 特性清单

本学期的编译器工作由本人与孟笑朵同学共同完成。在小组工作中，基础 track 中，本人主要负责变/常量声明及初始化、赋值、及注释，进阶 track 中主要负责**函数**、语句块分支及**数组**、寄存器分配优化、**公共子表达式删除**及**数组批量赋值**，**浮点数为二人共同完成**，各自负责对应部分的浮点处理。

需要说明的是，这里注明的**分工并不是绝对的**，我们在实现编译器时时常出现互相帮助对方完善对方负责部分工作的情况。而在实际实现中也基本不可能出现一个人只需要关注自己的部分，不需要修改/调整队友代码的情况，所以这里的分工情况只表示**主要**负责，相关部分工作在我们二人的报告中都有可能提及，只是侧重点不同。

1.2 总体设计

总体而言，在词法分析及语法分析阶段，我们虽然对框架代码进行了不同程度的改动，但是整体差异不大；但在中间代码生成阶段，我们并没有在 funcDef 中遍历指令完成前驱后继的设置，而是将这些前驱后继的连接分散到了对应的 genCode 中进行处理，省去了一次额外的遍历。之后，由于中间代码控制流设计带来的影响，我们生成的目标代码也相应地与一般的实现有所差异 (主要差异在于布尔表达式)。而之后，由于框架并没有给出优化部分的基本逻辑，我们主要参考去年编译大赛中科大的作品进行了实现。[1]

2 第一站：词法分析

词法分析阶段主要利用了 Flex 工具实现词法分析器，依据输入的 SysY 语言程序识别其中所有单词，并将其转化为单词流，按照符号表需求输出每一个文法单元的类别、词素，及相应的属性。

2.1 符号表

符号表表设计如下：

表项	Token	Lexeme	Lineno	Offset	Scope	Addr
说明	单词	词素	行号	列偏移	作用域	符号表项地址

表 2: 符号表设计

词法分析阶段实现的符号表共有 6 列，包含单词的类别、位置等相关信息，词法分析结果示例如下：

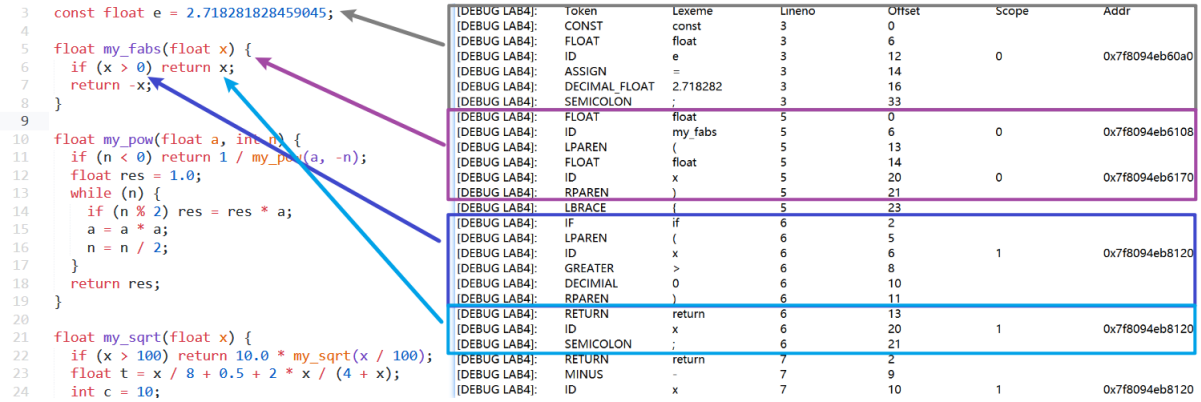

图 2.1: 词法分析示例

2.2 程序设计

此处重点说明作用域的判定：每当识别到左大括号 (’’) 时更新作用域，同时记录上一作用域当前的符号表大小，并在 globalFrame 中压入新的作用域，记录当前的作用域编号。每当识别到 ID 时，**优先在当前作用域的符号表中查找**，如果找到则进行对应的输出；如果未找到，则在当前作用域新建变量，并创建符号表表项。

3 第二站：语法分析

在语法分析阶段，我们完成了 SysY 语言的上下文无关文法及翻译模式的设计，并借助 Yacc 工具实现语法分析器，完成了语法树数据结构的设计。

语法分析生成语法树结果示例如下：

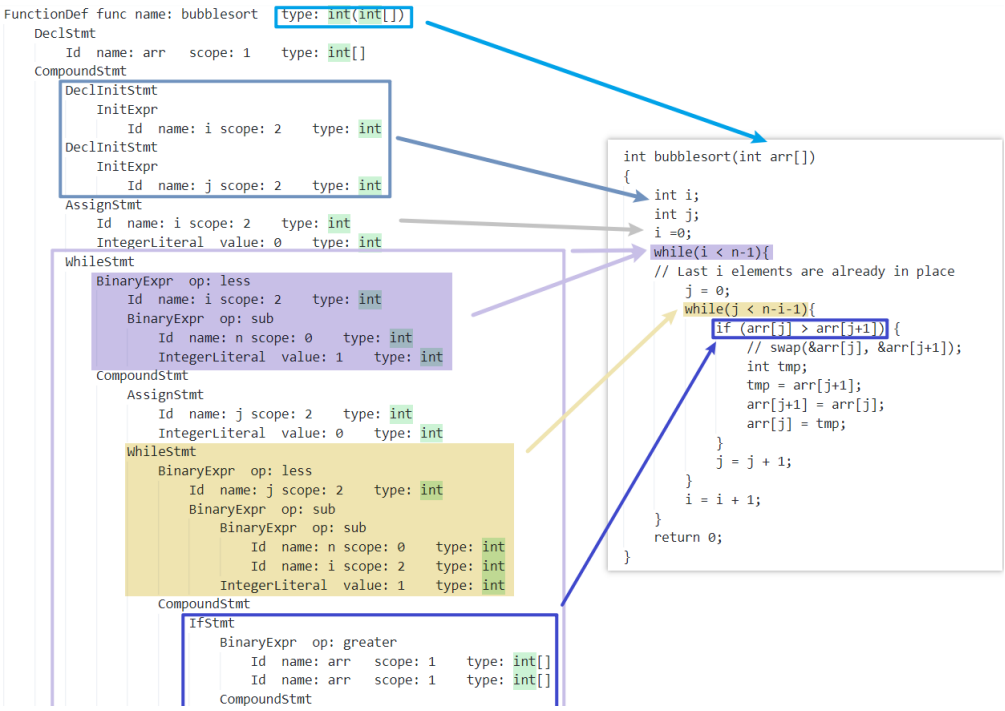

图 3.2: 语法分析示例

在框架代码中，有三种类型的符号表项：

- **ConstantSymbolEntry**：用于保存字面值常量属性值的符号表项。我们拓展了该类的成员变量用于存储并区分 int 及 float 常量，该类存储的数值为编译器阶段程序可确定的常量值，我们借助此类为常量折叠完成基础工作；
- **TemporarySymbolEntry**：用于保存编译器生成的中间变量信息的符号表项。同样，我们也拓展了该类的成员变量用以存储浮点的中间变量信息；
- **IdentifierSymbolEntry**：保存源程序中标识符相关信息的符号表项。通过作用域来区分 global 变量、param 变量及 local 变量，存储并区分相应的常量/变量标识符信息。

3.1 初始化 & 声明

在一开始的设计中，初始化 (InitStmt) 与声明 (DeclStmt) 的 CFG 设计被完全**分离**了开来，分别处理。一切都相安无事，直到出现了**初始化与声明混合**的样例，比如：

```
1  int a, b = 1;
```

这意味原本一连串初始化或一连串声明分别串起的模式需要做出调整，二者需要进行整合。最直接同时也是最容易想到的改进是用一个新的 token 来归约声明与初始化，用这个新的 token 来串起交叉的声明 & 初始化，但这样会使得初始化与声明发生 **reduce/reduce conflict**（如，在遇到新的 Type ID 时，不知是由 DeclStmt COMMA Type ID 规约到 DeclStmt 还是由 newToken COMMA Type ID 规约到 NewToken）。

此处有两种解决方案：其一，是不再将声明与初始化写成串起的形式（如，A COMMA Type ID），用一个新的 token 整合声明与初始化；其二，是直接整合当前的 DeclStmt 与 InitStmt 的串起方式，用其中一个 token。而最终，我们选择了第二种方案：

CFG 1. InitStmt

$$
\begin{aligned}
\textbf{InitStmt} \ : \ & ID \ ASSIGN \ Exp \\
| \ & ID \\
| \ & ArrInit \\
| \ & InitStmt \ COMMA \ ID \ ASSIGN \ Exp \\
| \ & InitStmt \ COMMA \ ID \\
| \ & InitStmt \ COMMA \ ArrInit \\
; &
\end{aligned}
$$

产生式由上至下分别表示初始化、声明、数组，串起初始化、串起声明、串起数组。这样就使得 InitStmt 拓展了 DeclStmt 的功能，同时避免了 conflict 的产生。

此外，可以发现，InitStmt 是不含类型 Type 的。在语法分析阶段，语法分析器自下而上地建立语法树，也就是说，当语法器开始处理标识符（ID）时，还没有储存类型信息，而 Type 将在当前结点的父节点才能取得，这就使得在声明/初始化时**建立 SymbolEntry 缺少 Type**。

一种比较常见且广为使用的解决办法是通过**全局变量**来传递类型信息：因为语法器是从左往右处理单词流的，故在处理标识符（ID）时，按照 SysY 语法规范一定已经遍历过 Type，此时可以通过一个全局变量保存该 Type，在建立标识符（ID）的 SymbolEntry 时可以使用该全局变量。

而我们当时选择了先置 Type 为空，在处理到父节点、获取到类型信息时，再通过 **setType** 的形式传递 Type。

一切都相安无事，直到我们遭遇了如下样例，即**同一句代码内的变量相互初始化**：

```
1  const int one = 1, two = 2,three = one + two;
```

其实一般而言，这样的代码在刚才描述的处理方法**并不会暴露出问题**。但我们在进行**常量折叠**时，需要获取标识符（ID）的 const 信息来判断是否能进行折叠。但在上面的样例中，同样会遇到在处理标识符（ID）、建立 SymbolEntry 时缺少 const 信息的问题。

此时，如果仍然采取在父结点处理的形式，则需要在父结点重新遍历一遍子结点处理过的初始化信息，比较繁琐，故我们选择使用全局变量来标识当前处理的标识符（ID）的 const 信息。即，在单词流中识别到 const 置全局变量。

3.2 函数定义 & 声明 & 调用

在实现函数的上下文无关文法设计时，我们区分了函数的定义与声明，并在函数调用部分对输入输出函数进行处理（输入输出部分主要由队友完成，此处省略，相关产生式用省略号代替）。

CFG 2. Function

FuncHead *Type ID* ;

FuncDecl :

FuncHead LPAREN FuncFParamList RPAREN SEMICOLON ;

FuncDef : *FuncHead LPAREN FuncFParamList RPAREN* {}

BlockStmt {} ;

FuncCall : *ID LPAREN FuncRParam RPAREN*

| *GETINT LPAREN FuncRParam RPAREN*

| .....

FuncFParam : *Type ID*

| *Type ID ArrayIndex* ;

FuncFParamList : *FuncFParam*

| *FuncFParamList COMMA FuncFParam*

| *%empty* ;

FuncRParam : *Exp*

| *FuncRParam COMMA Exp*

| *%empty*

;

针对此处的 CFG，主要有三点需要说明：

1. 首先，可以发现，此处额外设置了一个 token，即函数头 FuncHead。这个设置主要是为了**更新当前作用域**，以在处理形参及 BlockStmt 中变量时正确设置 scope，以便于区分出 gloabl/param/local。一般来说，其实不需要设置 FuncHead，直接将 FuncDef 的翻译模式在 Type ID 及 LPAREN FuncFParamList RPAREN 之间断开即可。但由于我们的语法器实现了**函数声明**，这就使得如果采用刚才描述的解决方案，函数声明与定义的 Type ID 就可能产生 **reduce/reduce conflict**，解析器不知道该规约到声明还是定义。因此，我们选择**提取了公共左因子**，将 Type ID 归约成 FuncHead，以在正确设置作用域同时避免冲突。

2. 其次，在函数头（FuncHead）的部分的 Type ID 实际上与 FuncFParam 的**前序是一致**的，均为 Type ID，不过此处**并未发生冲突**。这本质上是因为 bison 是 **LALR(1)** 的解析器，有 **1 个 lookahaed 的 token**，通过函数的左括号 LPAREN，以及 FuncFParamList 中，FuncFParam 后可能出现的逗号 COMMA 便可以区分出二者。

3. FuncFParamList 及 FuncRParam 中的**%empty** 是为了处理**函数形参/实参为空**的情况。

在处理函数定义时，需要先检查符号表中是否已经**存在对应的表项**。如果不存在，则需要新建 SymbolEntry；如果存在，则需要检查是否已经定义，如果已经定义，则需要进行**重定义**的报错；如果没有定义，则需要检查函数**声明的参数是否与定义匹配**。

```
FuncDef :
FuncHead LPAREN FuncFParamList RPAREN{
    paramCnt = floatParamCnt = intParamCnt = 0;// 函数参数编号重置
    FuncHead* funchead=(FuncHead*)$1;
    SymbolEntry *se = identifiers->lookup(funchead->getName());
    FunctionType* funcType;
    // 如果第一次定义/不存在声明
    if(se==nullptr){
        std::vector<Type*> vec; // 保存参数类型
        std::vector<Operand*> vecSe; // 保存参数操作数
        DeclStmt* fparam = (DeclStmt*)$3;
        // 逐个处理形参
        while(fparam!=nullptr){
            IdentifierSymbolEntry * idse = (IdentifierSymbolEntry*)fparam
                    ->getId()->getSymbolEntry();
            vec.push_back(idse->getType());
            idse->setAddr(fparam->getId()->getOperand());
            vecSe.push_back(fparam->getId()->getOperand());
            fparam = (DeclStmt*)(fparam->getNext());
        }
        funcType = new FunctionType(funchead->getName(),funchead->getType(), vec);
        funcType->setParamSe(vecSe);
        retType = funcType->getRetType();
        // 加入符号表
        SymbolEntry* se = new IdentifierSymbolEntry(funcType,
                funchead->getName(), identifiers->getPrev()->getLevel());
        identifiers->getPrev()->install(funchead->getName(), se);
    }
    // 如果存在声明/重复定义
    else{
        funcType =(FunctionType*)se->getType();
        retType = funcType->getRetType();
        std::vector<SymbolEntry*> rparams;
        ExprNode* exp = (ExprNode*)$3;
        while(exp!=nullptr){
            rparams.push_back(exp->getSymbolEntry());
            exp=(ExprNode*)exp->getNext();
        }
        if(se->alreadyDef()){// 检查重复定义
            fprintf(stderr,"function redef!\n");
        }
```

```
            else se->setAlreadydef(1);
            // 声明与定义的参数类型匹配检查
            bool check_res= funcType->checkParam(rparams);
            if(check_res){
                /* 此处省略不匹配的报错信息输出 */
                /* ..... */
            }}
    }
    BlockStmt{
        FuncHead* funchead=(FuncHead*)$1;
        SymbolEntry *se = identifiers->lookup(funchead->getName());
        // 建立 FunctionDef 对象
        $$ = new FunctionDef(se, $6, (DeclStmt*)$3);
        SymbolTable *ident = identifiers;
        identifiers = identifiers->getPrev();
        FunctionType* func = dynamic_cast<FunctionType*>(se->getType());
        delete ident;
        /* 返回检查，将在类型检查一节讲解，此处省略 */
        /* ..... */
    };
```

3.3 数组

数组 CFG 设计的一个主要难点是多维数组的初始化赋值，这一点无论对语法分析还是后续的中间文法、目标代码都是相同的。

CFG 3. Array

$$
\begin{aligned}
\textbf{ArrInit} &: ID\ ArrayIndices \\
&\mid ID\ ArrayIndices\ ASSIGN\ ArrAssignExp; \\
\textbf{ArrAssignExp} &: ConstExp \\
&\mid LBRACE\ RBRACE \\
&\mid LBRACE\ ConstExpInitList\ RBRACE; \\
\textbf{ArrayIndices} &: LBRACKET\ ConstExp\ RBRACKET \\
&\mid ArrayIndices\ LBRACKET\ ConstExp\ RBRACKET \\
&\mid LBRACKET\ RBRACKET; \\
\textbf{ConstExpInitList} &: ArrAssignExp \\
&\mid ConstExpInitList\ COMMA\ ArrAssignExp \\
&;
\end{aligned}
$$

上述产生式一个重要特征在于：ConstExpInitList 与 ArrAssignExp 是**相互包含**的。这样的设计与

数组初始化本身的形式是分不开的，比如：

```
int array1[5][4] = {{1,2,3},{4},{},{5,6}}; // 式 (1)
int array2[5][4] = {1,2,3,4,5,6};          // 式 (2)
```

其中，最外层的大括号既可以像式 (1) 一样括起对 array1 不同切片的赋值，也可以像式 (2) 一样，括起对元素的赋值。这**对数组实际的赋值结果会产生影响**。比如，array1[0][3]=0,array2[0][3]=4。

ArrAssignExp 产生式中的 LBRACE RBRACE 与 ConstExp，可以表示数组初始化的等号右半部分中的，与赋值数值，这二者为数组初始化的基本元素；之后，ConstExpInitList 利用逗号 COMMA 连接起这些基本元素，而 ArrAssignExp 又可通过 LBRACE ConstExpInitList RBRACE 完成对数组某一维度初始化的归约。

实际上，数组初始化的核心也就在于 ArrAssignExp -> LBRACE ConstExpInitList RBRACE，它在**结束特定维度的赋值**的同时，也**将该维度的赋值内容作为基本元素规约**到 ArrAssignExp，这样一来，又可以利用 ConstExpInitList **串起**这些基本元素。

此外，对于形如 arr[4][6] 这样由中括号 [] 括起的数组切片，ArrayIndices 将在归约时判断 ConstExp 的 SymbolEntry 是否为常量。因为常量在程序的单词流中其实不一定紧跟着 const 这个单词，所以需要对符号表项类型进行判断，如果不是常量则需要报错。并且，ArrayIndices 结点处会计数 ID 后跟随的切片数量，以计算当前 ID 的使用性维度，进而判定后续操作数的类型是指针/数组/基本类型。

3.4 语句块

在语法分析阶段，语句块主要需要处理的问题是悬空 else 的二义性问题。

CFG 4. If-else

IfStmt : *IF LPAREN Cond RPAREN Stmt %prec THEN*
| *IF LPAREN Cond RPAREN Stmt ELSE Stmt*
;

此处的%prec 表示其所在的语法规则中最右边的运算符或终结符的优先级与%prec 后面的符号的优先级相同。而 ELSE 的优先级利用%precedence 设置为高于 THEN。这是为了在遇到多个 IfStmt 时，如果遇到新的 ELSE，则将该 ELSE 优先与最近的 If 进行匹配，即保证优先归约到 If-Else。

4 第三站：类型检查中间代码生成

在该阶段，我们完善了类型检查，并在语法分析阶段构建好的语法树的基础上，完成了中间代码的生成。需要说明的是，与框架不同，我们的**类型检查在语法树构建过程中进行**，在语法树构建完毕后不需要再通过 TypeCheck 进行检查。

4.1 类型检查

由于我们没有统一实现 TypeCheck 函数，为了表述更为清晰，在此处给出实现的类型检查清单及实现位置：

检查	内容	位置
变量声明	未声明/重复声明	语法分析过程对应单词翻译模式中
函数声明/定义	未声明/重复定义/声明定义不匹配	语法分析过程对应单词翻译模式中
函数返回类型	函数声明类型与返回类型匹配（含隐式类型转换）	语法分析过程对应单词翻译模式中
函数参数检查	函数实参与形参数目、类型匹配（含隐式类型转换）	语法分析过程中对应单词翻译模式中调用 checkParam
条件表达式	float/int 到 bool 的隐式类型转换	生成中间文法时对应条件表达式 genCode 中
表达式运算	操作符类型匹配/void 不参与计算	语法分析过程中表达式的翻译模式中
数组维度	声明时维度为常量/最高维缺省与计算/赋值及传参维度检查	语法分析/生成中间文法对应类型函数中
数组类型	数组声明类型及初始化的匹配（含编译器阶段常量折叠）	语法分析过程对应单词翻译模式中
break/continue 静态检查	检查二者是否仅出现在 while 语句中	语法分析过程对应单词翻译模式中

表 3: 类型检查清单

下面对本人负责工作的重点进行介绍：

4.1.1 变量 & 函数声明检查

对于变量，其未声明使用与重复声明检查在语法分析过程完成，每次检查通过从符号表中查找对应 SymbolEntry，并依据查找结果来判断声明情况：

- 每次识别到**应该归约至 Lval** 中的 ID 时，如果没有查找到对应 SymbolEntry，则判定为变量未声明使用；
- 每次识别到**应该规约到 InitStmt** 中的 ID 时，如果 install 时查找符号表发现已存在相同表项，则判定为变量重定义。

对于函数，其未定义使用及重复声明的检查，与上述过程类似，只是实现该部分逻辑的位置不同。而定义及声明的类型匹配检查则主要包含：返回值类型及参数类型检查两个部分，二者与函数其他部分的类型检查工作有所重合，将在接下来的小节依次进行说明：

4.1.2 函数返回值检查

函数的返回值检查分为两个部分：1) **判断是否存在 return**; 2) **判断 return 类型是否匹配**。

首先，对于 return 存在性的判定需要考虑到：void 类型的函数可以省略 return，但除 void 外的函数至少有一个 return，并且一个函数内部可以存在多个 return，但 return 不能出现在函数定义的外部。

基于此，我们在语法分析阶段利用全局 int 变量 fin$_r$*eturn* *return*

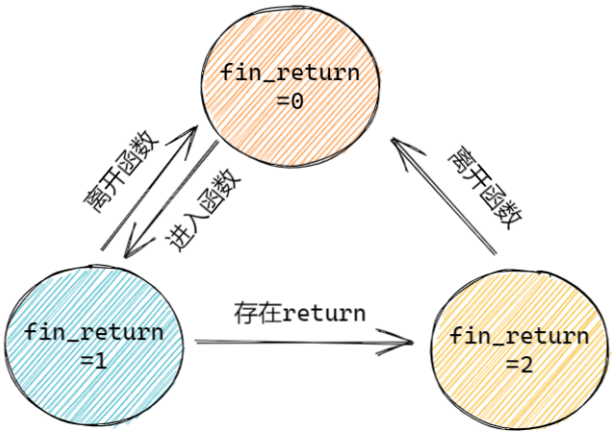

图 4.3: fin$_r$*eturn*

如果报错，则可能存在两种情况：

- 如果在离开函数定义时 fin_return!=2 并且函数的返回类型不为 void，则输出缺少 return 语句的报错；
- 如果在 fin_return==0 时识别到 return 语句，则输出 return 需要在函数定义内部的报错。

而对于 return 类型的匹配判断，可以将其抽象为两个 Type 的一致性判定。为此，我们定义了 pairTypeCheck 函数，比较过程大致为：

1. 首先，根据 Type 复合类型获取了对应的基本类型，并将其进行比较；
2. 如果**基本类型相同**，则再根据复合类型执行不同比较：(a) 如果复合类型是指针，则需判断二者 Type 均为指针; (b) 如果复合类型为数组，则需判断数组的**维度及维度大小**是否相同；如果再次经过比较后，结果仍为相同，则判定进行比较的两个 Type 为同一类型。
3. 如果**基本类型不相同**，则再根据所比较 Type 的实际类型来返回不同的数值：(a) 当第一个 Type 是 int 而第二个 Type 为 float 时，返回 2，表明此处存在 **float2int** 的隐式类型转换；(b) 当第一个 Type 是 float 而第二个 Type 为 int 时，返回 3，表明此处存在 **int2float** 的隐式类型转换；

```
1  inline int pairTypeCheck(std::vector<Type *> typeVec)
2  {
3  Type *basicType1=nullptr;
4  Type *basicType2=nullptr;
5  std::vector<Type *> basicTypeVec;
6  basicTypeVec.push_back(basicType1);
7  basicTypeVec.push_back(basicType2);
8  // 计算基本类型
9  for (int i = 0; i < 2; i++)
10 {
11     if (typeVec[i]->isArray()){ // 处理数组
12         basicTypeVec[i] = dynamic_cast<ArrayType *>(typeVec[i])->getEleType();
```

```
        }
        else if (typeVec[i]->isFunc()){ // 处理函数
            basicTypeVec[i] = dynamic_cast<FunctionType *>(typeVec[i])->getRetType();
        }
        else if (typeVec[i]->isPointer()){ // 处理指针
            basicTypeVec[i] = dynamic_cast<PointerType *>(typeVec[i])->getValueType();
        }else{ // 基本类型
            basicTypeVec[i] = typeVec[i];
        }
    }
    if (basicTypeVec[0] == basicTypeVec[1]){// 如果相等
        if (typeVec[0]->isArray()){ // 如果为数组则需进一步新检查
            auto arr1 = dynamic_cast<ArrayType *>(typeVec[0]);
            auto arr2 = dynamic_cast<ArrayType *>(typeVec[1]);
            if (arr1->getDimen() == arr2->getDimen()){
                auto len1 = arr1->getLenVec();
                auto len2 = arr2->getLenVec();
                for(int i=0;i<arr1->getDimen();i++){// 检查维度大小
                    if(len1[i]!=len2[i])
                        return 1;
                }
                return 0;
            }
            else
                return 1;
        }else if(typeVec[0]->isPointer()) // 确认基本类型相同后检查是否均为指针
            return typeVec[0]!=typeVec[1];
        return 0;
    }
    else{ // 如果不等
        /* f to i */
        if (basicTypeVec[0]->isInt() && basicTypeVec[1]->isFloat())
            return 2;
        /* i to f */
        else if (basicTypeVec[0]->isFloat() && basicTypeVec[1]->isInt())
            return 3;
    }
    return 0;
    }
```

在进行函数的返回类型匹配检查时，只需要在 ReturnStmt 的翻译模式中调用此函数即可，分别传入 FunctionType 的返回类型及 return 语句后的表达式类型，并根据 pairTypeCheck 的返回值进行

判断。需要说明的是，此处仅当返回值为 1 时才会进行报错，如返回 2/3，则会**进行对应的隐式类型转换**，不会报错。

4.1.3 函数参数检查

在该部分工作中，考虑到传参过程中的**隐式类型转换**，由于不同变量作为**不同函数**的实参 & 作为**不同位置**的参数传入，都会影响变量是否进行隐式类型转换的结果，所以此处建立一个新的结构体，用来记录函数、形参类型、对应参数位置及是否进行隐式类型转换的 bool 变量，并在 SymbolEntry 类中加入 unordered_multimap<string, transItem*> transitems; 成员变量。之所以使用 unordered_multimap，是因为该结构支持插入关键字相同的项，**与同一变量多次作为同一函数的参数传入，但传入参数位置不同**的场景相契合。

```
struct transItem{
    bool isTrans=0; // 是否转换
    Type *transType=nullptr; // 目标类型
    int paramno=-1; // 传参位置
};// 函数名利用 unordered_multimap 的关键字记录
```

这样，在 genCode 生成中间代码的过程中，就可以通过获取该结构体来判断是否加入 ImplictCastExpr，并设置转换的目标类型。

而在进行函数的参数检查时，首先需要比较实参与形参的个数，如果相同再逐个按序比较参数的 Type，此处直接调用上一小节介绍的 pairTypeCheck 函数即可。检查时调用 checkParam 函数代码如下：

```
bool FunctionType::checkParam(std::vector<SymbolEntry*> rParamsSE)
{
std::vector<Type*> rparams;
for (size_t i = 0; i < rParamsSE.size(); i++)
    rparams.push_back(rParamsSE[i]->getType());
if(paramsType.size()!=rparams.size()){ // 比较形参与实参的个数
    assert(paramsType.size()==rparams.size());
    return false; // 不一致直接返回
}
std::vector<Type*> typeVec;
for (size_t i = 0; i < paramsType.size(); i++){
    typeVec.clear();
    typeVec.push_back(paramsType[i]);
    typeVec.push_back(rparams[i]);
    transItem * item;
    int retCode = pairTypeCheck(typeVec); // 逐个比较形参与实参 Type
    switch(retCode){ // 根据返回值进行处理
        case 0: // 类型一致
```

```
19              item=new transItem();
20              rParamsSE[i]->appendTransItem(this->func_name,item);
21              break;
22          case 1: // 存在类型不一致的实参，直接返回
23              return 0;
24          /* 隐式类型转换: f to i */
25          case 2:
26              item=new transItem();
27              item->isTrans=1; item->paramno=i;
28              item->transType=paramsType[i];
29              rParamsSE[i]->appendTransItem(this->func_name,item);
30              if(rParamsSE[i]->isConstant()){
31                  int value
32                      =dynamic_cast<ConstantSymbolEntry*>
33                      (rParamsSE[i])->getFloatValue();
34                  dynamic_cast<ConstantSymbolEntry*>(rParamsSE[i])
35                      ->setIntValue(value);
36              }
37              break;
38          /* 隐式类型转换: i to f */
39          case 3:
40              item=new transItem();
41              item->isTrans=1; item->paramno=i;
42              item->transType=paramsType[i];
43              rParamsSE[i]->appendTransItem(this->func_name,item);
44              if(rParamsSE[i]->isConstant()){
45                  float value
46                      =dynamic_cast<ConstantSymbolEntry*>
47                      (rParamsSE[i])->getIntValue();
48                  dynamic_cast<ConstantSymbolEntry*>(rParamsSE[i])
49                      ->setFloatValue(value);
50              }
51              break;
52      }}
53  return true;
54  }
```

4.1.4 数组检查

数组检查同样分为两个部分：(1) 数组类型检查; (2) 数组维度检查。

对于数组类型检查来说，这部分的工作与变量的类型检查基本类似，即如果数组初始化时，大括号内的常量如果存在与声明的类型不符的情况（比如声明为 int 但初始化的值为 float），则将在编译器

期完成类型转换，直接将相应的浮点值转换为整型。

而对于数组维度检查，则包括 (1) 检查**声明时的维度为常量自然数**、(2) 数组初始化时**只允许最高维缺省**，(3) 与缺省时的**最高维大小计算**，(4) **不允许赋值个数大于数组大小**，以及 (5) **数组赋值、传参时的维度检查**；

对于 (1)，只需要在语法分析数组声明时检查 [] 内表达式的类型，判断是否满足常量自然数即可；对于 (2)，需要统计数组声明时维度的缺省情况，如果缺省不是最高维或大于 1 处，则报错；对于 (3)，则需要统计数组初始化时的赋值个数，并根据除了最高维之外的数组大小进行计算；对于 (4)，同样需要统计数组初始化时的赋值个数，如果大于数组大小则报错；而对于 (5)，与上文提到的 Type 一致性检查基本一致 (其本质上也属于类型检查)，此处不再赘述。

4.2 中间代码生成

与框架不同，我们并没有在 FunctionDef::genCode 中调用 stmt->genCode() 之后，再重新遍历一遍刚才生成的指令 (instruction) 来构建控制流。而是**分别在需要构建控制流的地方直接建立相应的前驱/后继关系，这样就不再在最后对函数内的所有指令进行一次遍历，提高了编译效率**。

要实现这样的控制流设计，我们主要需要处理三个问题：

- 第一，需要处理 **If、If-Else、While 这类条件语句**的前驱后继关系，设置好布尔表达式为真 & 为假对应的跳转块 (then_bb、else_bb)，及条件分支结束后的结束块 (end_bb)。此外，还需要注意处理**函数在不同分支内分别 return，使得 end_bb 为空但被串入指令 list 的情况**。(比如样例 27)
- 第二，需要处理 **continue、break 这类短语句**的前驱后继关系，这类语句不像 if-else 一样，没有与自己的前驱块与后继块建立直接联系，也不知道自己之前/之后的语句是什么，我们需要为其建立相应的连接。
- 第三，需要处理**布尔表达式**的控制流关系，根据**逻辑短路**特性为关系表达式设置正确的前驱后继。

处理上述三个问题后，可以生成正确的中间文法如下图所示：

图 4.4: 中间文法生成结果示例

其中第三点主要由队友完成，本报告不作陈述。故下文将主要讲解第一点及第二点。

4.3 条件语句控制流

回填是控制流设计中非常重要的一个部分，其基本思想在于在生成一些跳转指令时，**暂时不指定这个指令跳转的目标位置，而是先将其暂时储存起来，在能确定正确的 br 标号之后，再去填充对应指令的目标标号**。

举个简单的例子：

```
while (a) {
    if (b) {
        break;
    }
in_while();
}
out_while();
```

以上述代码为例，在一趟式的翻译中，我们必须在处理 out$_w hile()$ $while$ $(Stmt)$ $break$ br
当经过一段分析后，我们开始处理 out$_w hile()$
此处以 IfElseStmt 的 genCode() 为例：

```
void IfElseStmt::genCode(){
```

```
    Function* func = builder->getInsertBB()->getParent();
    BasicBlock *then_bb, *end_bb,*else_bb;
    then_bb = new BasicBlock(func);
    end_bb = new BasicBlock(func);
    else_bb = new BasicBlock(func);

    // 建立前驱后继关系
    then_bb->addPred(builder->getInsertBB());
    builder->getInsertBB()->addSucc(then_bb);
    end_bb->addPred(then_bb);
    then_bb->addSucc(end_bb);

    else_bb->addPred(builder->getInsertBB());
    builder->getInsertBB()->addSucc(else_bb);
    end_bb->addPred(else_bb);
    else_bb->addSucc(end_bb);

    // 条件表达式生成指令
    cond->genCode();

    // 回填
    backPatch(cond->trueList(), then_bb);
    backPatch(cond->falseList(), else_bb);

    // 条件表达式 bool 类型的隐式类型转换
    Operand *tem = new Operand(new TemporarySymbolEntry
        (TypeSystem::boolType, SymbolTable::getLabel()));
    new CmpInstruction(
        CmpInstruction::NE, tem, cond->getOperand(),
        new Operand(new ConstantSymbolEntry(TypeSystem::intType, 0)),
        builder->getInsertBB());
    new CondBrInstruction(then_bb, else_bb, tem, builder->getInsertBB());

    // 获取到目标块，生成指令
    builder->setInsertBB(then_bb);
    thenStmt->genCode();
    then_bb = builder->getInsertBB();
    new UncondBrInstruction(end_bb, then_bb);

    builder->setInsertBB(else_bb);
    elseStmt->genCode();
    else_bb = builder->getInsertBB();
```

```
44      new UncondBrInstruction(end_bb, else_bb);
45      builder->setInsertBB(end_bb);
46  }
```

可以发现，像 if、if-else、while 这样的指令，其前驱后继关系完全可以在各自的 genCode() 函数中建立。在 then_bb、else_bb、end_bb 对应到真正的语句块之前，就可以将他们先连接起来，之后再利用回填将尚未填写的块号填写，即完成了相应的控制流生成。

一切都相安无事，直到样例 27 的出现：

```
1   int a;
2   int main(){
3       a = 10;
4       if( a>0 ){
5           return 1;
6       }
7       else{
8           return 0;
9       }
10  }
```

由于 if 与 else 分支内都存在 return 函数，并且 if-else 之后不再有其他指令，这就会使得 if-else 语句的 **end_bb 为空**，但该对象仍然被串入 block$_list$ **块号被输出，但其后没有任何一条指令**

对此，我们效仿 clang 的形式，选择在 function 中默认 alloc 一个变量，并在判断出末尾基本块为空时，插入额外的 store 及 return 指令，实现的 FunctionDef::genCode 如下：

```
1   void FunctionDef::genCode()
2   {
3   Unit *unit = builder->getUnit();
4   Function *func = new Function(unit, se);
5   BasicBlock *entry = func->getEntry();
6   builder->setInsertBB(entry);
7   Type *type = dynamic_cast<FunctionType *>(se->getType())->getRetType();
8
9   // 一个额外的变量，插入 AllocaInstruction 指令
10  TemporarySymbolEntry *temp = new TemporarySymbolEntry(type, SymbolTable::getLabel());
11  Operand *temp_op = new Operand(temp);
12  AllocaInstruction *alloca = new AllocaInstruction(temp_op, temp);
13  entry->insertFront(alloca);
14
15  if (decl)
16      decl->genCode();
17  if (stmt)
```

```
18      stmt->genCode();
19
20  BasicBlock *bb = builder->getInsertBB();
21  // 如果末尾块为空，额外插入指令
22  if (bb->empty()){
23      SymbolEntry *temp3 = new TemporarySymbolEntry(type, SymbolTable::getLabel());
24      SymbolEntry *temp2 = new TemporarySymbolEntry(new PointerType(type), temp->getLabel());
25      Operand *temp_op2 = new Operand(temp2);
26      Operand *temp_op3 = new Operand(temp3);
27      new LoadInstruction(temp_op3, temp_op2, bb);
28      new RetInstruction(temp_op3, bb);
29  }}
```

针对上文提到的样例 27，利用这种策略，可以生成中间文法如图所示：

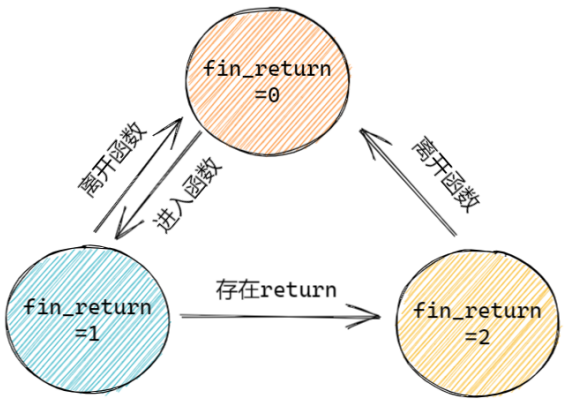

图 4.5: 样例 27 中间文法生成结果

4.4 短语句控制流

在之前的语法分析过程中，我们并没有记录 break、continue 等指令所在的 whileStmt。而这一点，则通过改写 yacc、设立一个全局的堆栈来记录 ast 建立过程中当前所处的 whileStmt 指针，由于可能存在 while 嵌套的情况，所以此处设立的不是一个简单的指针，而是**堆栈**，以更新 (压栈) 或回退 (弹栈) 到 break/continue 对应的 while 语句块内。

这样一来，我们只需要根据 break/continue 对应的 whileStmt 的 end_bb 及 then_bb，来设置 break/continue 的跳转块即可。

4.5 数组

在生成数组相关的中间代码时，需要增加 gep 指令。gep 指令的工作是 “计算地址”，本身并不进行任何数据的访问和修改。对于高维数组来说，在没有优化的过的情况下，往往需要对应维度数目的 gep 指令来帮助寻址，每条 gep 指令负责一个特定维度的寻址，此时上一条 gep 指令的 dst 为下一条 gep 指令的 src。

对于数组来说，生成的中间代码会根据变量不同情况而产生一些不同。比如，是否是函数的传入参数对指令的需求并不相同。如果是参数，则需要多加一条 load 指令。global 与 local 的数组跟其他变量一样生成的指令也存在不同。此处不再一一说明，仅给出 Id::genCode() 处理数组部分的代码作为示例。

```
ArrayType *arrType = (ArrayType *)this->getSymbolEntry()->getType();
ArrayType *eletype = arrType;
Operand *indice_dst = dst;
Operand *indice_src = addr;
bool first = 1; // 标记第一条数组取值指令
ExprNode *idx = index;
// 一维数组参数的 idx 为 nullptr
if (arrFlag || idx == nullptr)
{
    TemporarySymbolEntry *tse = new TemporarySymbolEntry
        (eletype, SymbolTable::getLabel());
    indice_dst = new Operand(tse);
    if (se->isParam()){ // 如果是函数传入参数，需要 load
        new LoadInstruction(indice_dst, addr, bb);
    }
    else
    {
        GepInstruction *gep = new GepInstruction(indice_dst, addr, bb, nullptr);
        gep->setIdxFirst(1);
    }
    dst = new Operand(new TemporarySymbolEntry(
        new PointerType(arrType->getEleType()), tse->getLabel()));
    return;
}
int idxCnt = 0;
std::vector<SymbolEntry*> idx_ses;
// 遍历所有数组切片，生成 gep 指令，
// 上一条指令的 dst 为下一条指令的 src
for (; idx != nullptr;)
{
    idx->genCode();
    idx_ses.emplace_back(idx->getSymbolEntry());
    // 如果是函数传入的参数，且为第一条 gep，需要 load
    if (se->isParam() && first){
        indice_src = new Operand(new TemporarySymbolEntry
            (eletype, SymbolTable::getLabel()));
        new LoadInstruction(indice_src, addr, bb, 8);
```

```
        }
        indice_dst = new Operand(new TemporarySymbolEntry
            (eletype, SymbolTable::getLabel()));
        indice_src->setType(eletype->getStripEleType());
        eletype = (ArrayType *)eletype->getStripEleType();
        GepInstruction *gep;
        if(se->isParam()) // 设置 param 标志
            gep = new GepInstruction(indice_dst, indice_src,
                bb, idx->getOperand(),1,addr->getSymbolEntry());
        else
            gep = new GepInstruction(indice_dst, indice_src,
                bb, idx->getOperand());
        if (first){ // 设置第一条 gep 标志
            gep->setIdxFirst(1);
            first = 0;
        }
        indice_src = indice_dst;
        idx = (ExprNode *)idx->getNext();
    }
    // 左值，且取到数组元素
    if (!isLeft && idxCnt == arrType->getDimen())
    {
        dst = new Operand(new TemporarySymbolEntry
          (arrType->getEleType(), SymbolTable::getLabel()));
        new LoadInstruction(dst, indice_dst, bb);
        dst->setIdxSe(idx_ses);
    }
    else{
        dst = new Operand(new TemporarySymbolEntry(new PointerType
            (arrType->getEleType()), SymbolTable::getStillLabel() - 1));
        dst->setIdxSe(idx_ses);
    }
```

5 第四站：目标代码生成

终于，到了这个阶段我们实现了一个简陋但完整的编译器。在生成目标代码时，我们只需要从前往后遍历一边之前生成的中间代码，并将其逐句翻译成目标代码。

5.1 寄存器分配算法及优化

本小节实现的寄存器优化代码算法主要参考了 **Second-chance allocation**[2]，而原始的线性扫描分配器，大致分为下面几个步骤：

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

由于线性扫描算法只使用了局部的冲突信息，只关注了当前进行分配的 live interval 和当前 active 列表中或者 inactive 列表中的冲突，没有像图着色分配器一样反复地构建冲突图，所以比图着色分配器效率更高。并且，因为线性扫描分配算法按照活跃区间的 start/end 进行排序，再进行扫描，不会进行任何回溯，所以线性扫描寄存器分配的时间复杂度也是线性的。

线性扫描算法在计算活跃区间之前，需要对指令进行编号。之后，通过框架已经实现的活跃分析（live analysis），我们得以确定每个虚拟寄存器在程序中的活跃点。然后通过选出编号最大（i）和最小（j）的活跃点，组成活跃区间 [i,j]。**活跃区间只需要保证在此区间之外没有活跃点，不需要保证在这个区间内处处活跃的**，这里就为我们的优化留出了空间。如果两个虚拟变量的活跃区间发生了重叠，则称其存在冲突，产生了冲突的两个变量不可以使用同一寄存器。

需要说明的是，**对指令的编号会影响到代码生成的质量** [3]。因为在有些编号下活跃区间会更短，更短的活跃区间会降低冲突的可能性。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 先编号会得到图中 (a) 的结果，而给 else 先编号则会得到图中 (b) 的结果。

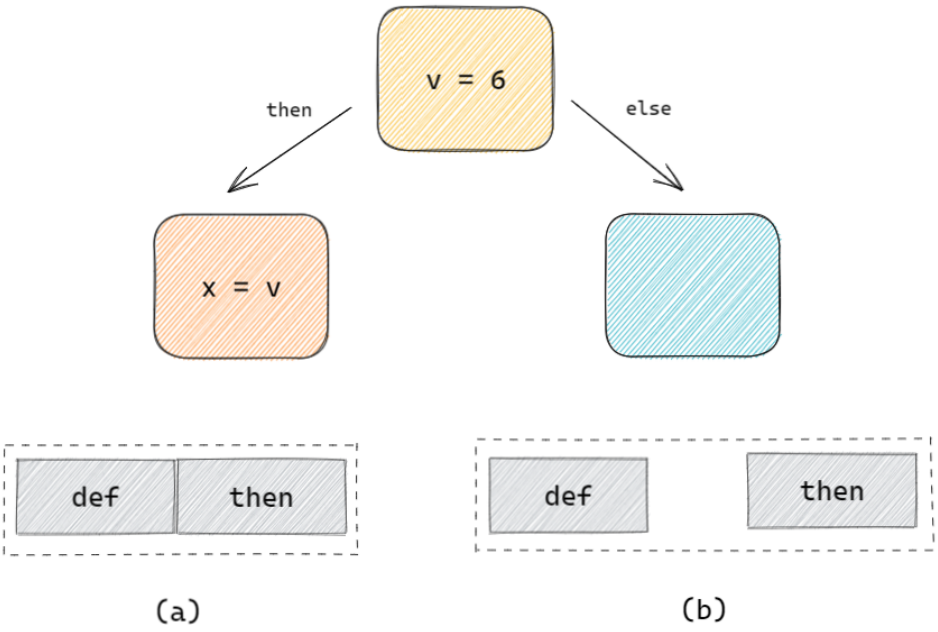

图 5.6: 编号影响目标代码生成结果示例

这里就可以发现，在计算活跃区间时，有些活跃区间会出现类似于图中 (b) 一样的**空洞** (interval hole)。事实上，如果另外一个活跃区间正好在这个空洞界限之内，就不会发生冲突。但是，由于原始的线性扫描算法在计算活跃区间时只考虑了最小及最大使用点，故其无法利用该空洞。

接下来本报告将针对这样的空洞对寄存器分配算法做出优化。

空洞是一个 interval 中变量不存在使用点的片段。基于 interval 建立时的算法流程，我们可以发现，如果出现了写操作，就有可能造成空洞的产生，因为写操作会使得变量原本的值被更新 (kill)。

为了利用空洞，我们需要将 interval 进行进一步的划分 (split)，将其切分为更小的 range，使得一个 interval 可以由一个或多个 range 组成。

这样一来，对于下图这种典型的情况来说：我们可以让 interval1 保存在两个位置，一个备份在寄存器 r1，而另一个备份则在内存中，这样就使得 interval2 有机会使用 r1。其中，由于 interval1 中的变量被保存在内存中时，存在空洞，即没有使用点，可以不使用寄存器。这样一来，我们就在只使用 1 个寄存器的情况下满足了两个 interval 的需要。

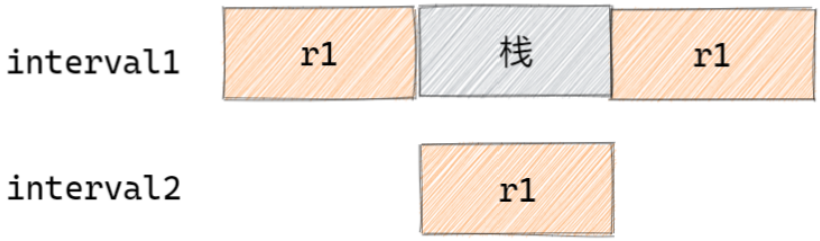

图 5.7: 典型空洞

这表明，如果我们能够使用 hole 以及对 interval 进行 split，那么就有机会**提高寄存器的利用率**。被处理过的 interval 被截下了一个片段，并成为了一个新的 interval，由于这样的 interval 更小，被二次处理时更有机会在寄存器分配时得到一个寄存器，而不是触发一个 spill。因此，这个方法被叫做 Second-chance allocation。

核心代码如下，这里基于框架构建，将 range 视作 interval 来构建：

```
std::vector<int> tmp_end;
for (auto &du_chain : du_chains) // 计算 range 的 start 与 end
{
    tmp_end.emplace_back(du_chain.first->getParent()->getNo());
    int maxno = -1;
    for (auto use : du_chain.second)
    {
        maxno = std::max(maxno, use->getParent()->getNo());
        if (use->getRange()){
            tmp_end.emplace_back(use->getParent()->getNo());
        }
    }
    tmp_end.emplace_back(maxno);
    std::vector<std::set<MachineOperand *>> ranges_uses(tmp_end.size());
    // 重新分配原本 interval 的 uses
    for (auto use : du_chain.second){
        for (size_t i = 0; i < tmp_end.size(); i++){
            if (use->getParent()->getNo() <= tmp_end[i])
                ranges_uses[i].insert(use);
        }
```

```
    }
    for (size_t i = 1; i < tmp_end.size(); i++){
        if ((i != 1 && ranges_uses[i].size() <= 1)) // hole
            continue;
        // 构建 range
        Interval *interval = new Interval(
                                    {tmp_end[i - 1],tmp_end[i],
                                    false, 0, 0, 0,
                                    {du_chain.first},ranges_uses[i]});
        intervals.emplace_back(interval);
    }
    tmp_end.clear();
    ranges_uses.clear();
}
```

5.2 目标代码生成注意事项

对于目标代码生成来说，关注点主要在于目标汇编语言的特性，包括寄存器数目、功能等。比如，在 arm 中，没有现成的指令可用于取模，我们需要手动地进行除、乘、减。

编写代码过程中主要遇到的巨坑如下：

- 没有意识到压栈要对齐；
- 没有意识到立即数有大小限制，超过需要先加载到虚拟寄存器；
- 没有意识到参数编号要浮点整型分开算。

6 终点站：代码优化

6.1 公共子表达式删除

公共子表达式删除 (CSE, Common Subexpression Elimination) 的主要思想在于利用可用表达式分析，计算出每个基本块的可用表达式集合 (use)，同时记录每一条可用表达式第一次出现时对应的指令 (def)。之后，遍历每一条指令，如果识别出该指令含有对应的公共子表达式时，则将其后任何引用到该指令的地方，全部用该公共子表达式第一次出现时对应的指令值来替换，同时将该含有公共子表达式的指令加入到可删除指令集中，最后在完成所有替换之后进行统一的删除。

CSE 可以根据删除的指令域分为 global 与 local 两个部分，对于 local，也即同一基本块中的公共子表达式删除基本流程与上文所述一致，而 global 域的公共子表达式删除则相对来说更加复杂，不过核心流程是相似的。

首先来看看 local 域的公共子表达式消除：参考经典样例 107，这里利用其缩减版进行演示。

```
int a[5][20000];
int main() {
    int ans = a[2 * 2][20000 - 1] + a[2 * 2][20000 - 1];
```

```
return 0;
}
```

优化前的汇编代码：

```
.L5:
mov r4, #4
ldr r5, addr_a0
ldr r6, =80000
mul r7, r4, r6
add r4, r5, r7
mov r5, r4
ldr r4, =19999
mov r6, #4
mul r7, r4, r6
add r4, r5, r7
ldr r5, [r4]
mov r4, #4
ldr r6, addr_a0
ldr r7, =80000
mul r8, r4, r7
add r4, r6, r8
mov r6, r4
ldr r4, =19999
mov r7, #4
mul r8, r4, r7
add r4, r6, r8
ldr r6, [r4]
add r4, r5, r6
str r4, [fp, #-4]
mov r0, #0
add sp, sp, #8
pop {r3, r4, r5, r6, r7, r8, fp, lr}
bx lr
```

优化后的汇编代码：

```
.L5:
mov r4, #4
ldr r5, addr_a0
ldr r6, =80000
mul r7, r4, r6
```

```
add r4, r5, r7
mov r5, r4
ldr r4, =19999
mov r6, #4
mul r7, r4, r6
add r4, r5, r7
ldr r5, [r4]
ldr r6, [r4]
add r4, r5, r6
str r4, [fp, #-4]
mov r0, #0
add sp, sp, #8
pop {r3, r4, r5, r6, r7, r8, fp, lr}
bx lr
```

可以发现，**对于相同的数组寻址的 gep 指令，以及二维数组寻址的 add 指令都进行了删除**。(并且这里也可以看出常量进行了折叠)。

实际实现 CSE 的过程中，利用 cmp_inst 结构体实现了指令之间严格小于的关系，便于公共子表达式的对比与识别。此外，还定义了四个 std::map<BasicBlock *, std::set<Instruction *, cmp_inst» 结构, 分别用于：

- bb_gen[B]：块 B 产生的可用表达式集合;
- bb_kill[B]：块 B 注销的可用表达式集合;
- bb_in[B]：块 B 入口的可用表达式集合;
- bb_out[B]：块 B 出口的可用表达式集合。

local 部分核心代码可见于下：

```
for (auto bb : f->getBlockList()){
std::vector<Instruction *> delete_list = {};
auto head = bb->end();
for (auto instr = head->getNext(); instr != head; instr = instr->getNext()){
    if (!is_valid_expr(instr))
        continue;
    // 如果子表达式的某一变量值发生更新，则对应的子表达式失效，从 bb_gen 中移除
    if (instr->isStore() && instr->getOpse().size()){
        auto change_se = instr->getOpse()[0];
        for (auto src = bb_gen[bb].begin(); src != bb_gen[bb].end();){
            bool find = 0;
            std::vector<SymbolEntry *> opse = (*src)->getOpse();
            for (auto se : opse){
```

```
                if (se == change_se){
                    bb_gen[bb].erase(src++);
                    find = 1;
                    break;
                }
            }
            if (!find)
                src++;
        }}
    if (is_valid_expr(instr)){
        auto res = bb_gen[bb].insert(instr);
        if (!res.second){
            auto old_instr = bb_gen[bb].find(instr);
            std::vector<Operand *> opd = instr->getOp();
            int cnt = (int)opd.size() - 1;
            Operand *dst = instr->getOp()[0];
            Operand *src1 = instr->getOp()[1];
            Operand *src2 = nullptr;
            // 查找操作数后续所有被使用的地方，并进行替换
            for (auto use : dst->getUses()){
                std::vector<Operand *> ops = use->getOp();
                for (size_t i = 0; i < ops.size(); i++){
                    if (ops[i] == nullptr)
                        continue;
                    if (ops[i]->toStr() == dst->toStr()){
                        use->getOp()[i] = (*old_instr)->getOp()[0];
                        if (is_valid_expr(use) && i > 0)
                            use->getOpse()[i - 1] = use->getOp()[i]->getSymbolEntry();
                        break;
                    }}}
            delete_list.emplace_back(instr);
        }
        else{
            // U 中插入子表达式
            auto u_res = U.insert(instr);
        }}}
// 删除所有不是第一次出现的公共子表达式
for (auto instr : delete_list)
    bb->remove(instr);
}
```

针对 global 的公共子表达式删除，需要利用数据流等式进行迭代计算，对基本块 B 计算 bb_in[B]

和 bb_out[B]。若 bb_out[B] 集合大小不变则结束迭代，得到最终结果。

$$bb_out[B] = bb_gen[B] \bigcup (bb_in[B] - bb_kill[B])$$

除此之外，与 local 相比 global 只是遍历及替换范围更广，其核心代码与 local 相差不大，此处不再赘述。

6.2 其他优化

除了 CSE 跟寄存器分配优化之外，还进行了一些细微的优化。

- **寄存器分配优化启发式**：考虑到寄存器分配会耗费大量的时间，那么在程序本身不存在任何浮点变量/函数时再去遍历浮点寄存器,其实十分不划算。于是,我们在词法分析阶段引入了 float_check 全局变量，用来判断程序中是否存在 float 的使用，如若没有，则在分配寄存器时，不遍历浮点寄存器。
- **数组初始化优化**：对于数组来说，如果 size 较大，又需要全部赋值为 0，则需要生成许多的数组赋值指令，而这同样也会大幅影响编译时间，因为需要逐一地进行 output()。于是，我们选择了在需要全部赋值为 0 时，调用 memset 函数，以缩减数组的初始化指令数目。

7 总结

这次报告继承了并行的不优良传统，又爆页数了... 给批阅的老师或者学长学姐们道歉... 总地来说，本学期的编译原理课程及实验确实让我收获良多。同时，在随着实验推进的过程中，也不断地想要回溯然后重写代码。这是因为前期在完成实验时由于缺乏对编译器具体实现的了解、以及实操经验，导致很多前期代码会为后期实现带来不变，或者甚至是埋下隐患。这一点，在编写 CSE 代码时给我带来了尤为惨痛的教训，教会了我前瞻性以及规划的重要性。

参考文献

[1] mlzeng. CSC2020-USTC-FlammingMyCompiler. (2023 年,1 月 15 日).

[2] Omri Traub, Glenn Holloway, and Michael D Smith. Quality and speed in linear-scan register allocation. *ACM SIGPLAN Notices*, 33(5):142–151, 1998.

[3] Christian Wimmer and Michael Franz. Linear scan register allocation on ssa form. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 170–179, 2010.