南開大學
Nankai University

计算机学院和网络空间安全学院

编译原理大作业报告

sysY 语言编译器设计

姓名：魏伯繁

学号：2011269 2011395

专业：信息安全

2023 年 1 月 15 日

目录

1 sysY 编译器设计思想及理念 **3**
1.1 编译器简述 . . . 3
1.2 总体架构与设计思路 . . . 3
1.3 使用工具 . . . 4

2 分工 **4**
2.1 词法分析 . . . 4
2.2 语法分析 . . . 4
2.3 类型检查及中间代码生成 . . . 4
2.4 汇编代码生成 . . . 5

3 词法分析 **5**
3.1 正则表达式的设计 . . . 5
3.2 数制转换 . . . 6
3.3 符号表 . . . 6
3.4 状态处理 . . . 8
3.5 规则部分 . . . 9

4 语法分析 **11**
4.1 类型系统及符号表 . . . 11
4.2 抽象语法树 . . . 13
4.3 语法分析与语法树的创建 . . . 15
4.3.1 Lval 表达式 . . . 16
4.3.2 while 表达式 . . . 17
4.3.3 unary 表达式 . . . 17
4.3.4 VarDef 表达式与 initial 表达式 . . . 19
4.3.5 递归逻辑 . . . 22

5 类型检查及中间代码生成 **22**
5.1 类型检查 . . . 22
5.1.1 定义问题 . . . 22
5.1.2 函数问题 . . . 23
5.1.3 控制流语句问题 . . . 24
5.2 中间代码生成 . . . 25
5.2.1 控制流的翻译 . . . 25
5.2.2 unary 的翻译 . . . 26
5.2.3 调用指令打印 . . . 28

6 ARM 代码生成 **28**
6.1 线性扫描 . . . 29
6.2 ARM 汇编码生成 . . . 32
6.2.1 机器码指令类构造 . . . 32

6.2.2 打印 ARM 指令 36

7 总结 **39**

1 sysY 编译器设计思想及理念

1.1 编译器简述

编译器在计算机科学领域的地位举足轻重，几乎所有的高级编程语言都需要使用编译器来讲高级语言转换为计算机能识别的机器语言，以 C 语言为例，编译器的流程大体包括：

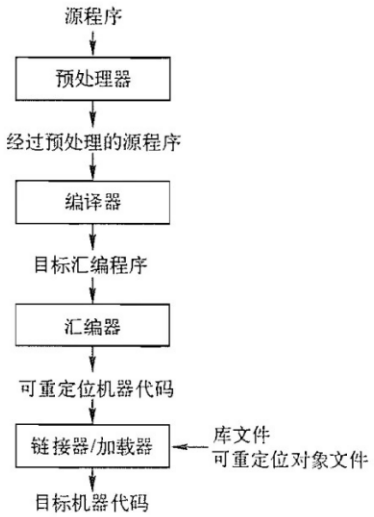

图 1.1: 汇编器流程图

其中：预处理器处理源代码中以 开始的预编译指令，例如展开所有宏定义、插入 include 指向的文件等，以获得经过预处理的源程序。编译器将预处理器处理过的源程序文件翻译成为标准的汇编语言以供计算机阅读。汇编器将汇编语言指令翻译成机器语言指令，并将汇编语言程序打包成可重定位目标程序。链接器将可重定位的机器代码和相应的一些目标文件以及库文件连接在一起，形成真正能在机器上运行的目标机器代码。

在本次实验中，我们将着重实现一个简单的 SysY 语言的编译器，即输入一段 SysY 代码，输出一段 ARM 指令。

1.2 总体架构与设计思路

本次实验的总体框架大概可以分为：词法分析、语法分析、语义分析（类型检查）、中间代码生成以及 ARM 汇编代码生成。

本次实验中每一个环节都是前后相互关联，且共同维护一些全局变量例如类型系统以及符号表。在词法分析阶段使用 flex 帮助我们完成词法分析的任务，通过编写正则表达式来对一些复杂的分词进行识别例如变量名、整数浮点数等，而对一些简单的 SysY 关键词则可以直接识别，例如 break、while 等，并将分词后每一个分词的标签传递给语法分析。

在语法分析阶段中，需要完成的最重要的任务就是构建抽象语法树，而构造的方式就是使用词法分析传递进来的词进行识别、汇总，判断出 sysY 语言的每一句分别在程序中起到什么样的作用，将其进行分类，例如 int a=10 就会被语法分析识别为包括赋值的声明语句，在语法树中会构造处一个左值节点以及一个表达式节点，这个表达式节点将会用常数 10 进行实例化。

语义分析阶段是贯穿整个 SysY 编译器的过程，有一些语义分析需要在语法分析完成，因为在语法分析中就需要在符号表中更新符号表项，由其便可以利用 vector 提供的函数来完成未定义和重定义

的错误检测，而诸如函数返回值检测则需要在中间代码生成时一并检测，因为只有当遍历语法树时才能确定一个函数返回了什么，才能判定一个函数到底有没有进行返回。

中间代码生成部分和 ARM 汇编代码的核心都是遍历语法树，只不过 ARM 汇编码需要在中间代码生成之后直接在中间代码的基础生生成 ARM 汇编代码。在这个过程中，就需要用到新的 instruction 类以及 MInstruction 类，他们分别负责保存一条指令所需要的基本信息，在打印时根据指令所保存的信息进行打印，在遍历语法树生成指令类时比较难处理的就是控制流指令的编写以及函数调用的编写，因为涉及到不同的分支，以及在函数调用时参数、寄存器的压栈和出栈，所以这部分是整个编译器的重点也是难点。

通过以上这些步骤，我们就可以将一个程序进行分词、构建语法树、遍历语法树生成中间代码以及最后的机器码，最终完成编译器的构造。

1.3 使用工具

LexFlex

Lex 是用来辅助生成词法分析程序的，它可以将大量重复性的工作自动计算，通过相对简单的代码可以生成词法分析程序。而 Flex 则是 Lex 的继承者，在 Linux 中被更广泛的使用，也更容易获得。

YaccBison

Yacc 是 Unix/Linux 上一个用来生成编译器的编译器（编译器代码生成器）。Yacc 生成的编译器主要是用 C 语言写成的语法解析器（Parser），（一般）需要与词法解析器 Lex 一起使用，再把两部分产生出来的 C 程序一并编译。Bison 是 Yacc 的 GNU 扩展。

gccllvm

gcc 与 llvm/clang gcc 和 llvm 是我们使用的主要编译工具。不仅用于编译我们的编译器，也将用于编译我们的编译器生成的汇编代码。clang 是 llvm 编译器工具集的前端，将源代码转换为抽象语法树（AST），由后端使用 llvm 编译成平台相关机器代码

2 分工

2.1 词法分析

王楠舟：符号表设计、数制转换、状态处理、词法分析规则编写

魏伯繁：正则表达式编写、状态处理、词法分析规则编写

2.2 语法分析

王楠舟：符号表设计、抽象语法树构造、完善、数组实现、浮点数实现、语法分析变量声明语句编写、while、break、continu 等节点设计

魏伯繁：类型系统设计、抽象语法树输出、语法分析常量声明语句编写、函数、算数、逻辑表达式节点设计、递归逻辑设计

2.3 类型检查及中间代码生成

王楠舟：更新维护符号表、类型检查–定义问题与函数问题、AST 中 gencode 的编写

魏伯繁：更新维护类型系统、类型检查–函数问题与控制流问题、llvm 中的语句打印

2.4 汇编代码生成

王楠舟：更新维护符号表、指令的 ARM 汇编代码翻译、内存加载、函数调用的机器码指令类编写

魏伯繁：更新维护类型系统、线性扫描、高层模块的汇编代码翻译、运算指令、控制流指令的机器码指令类编写

3 词法分析

构造编译器的第一步就是词法分析，在词法分析设计中，需要根据我们设计的编译器所支持的语言特性，设计正规定义。并利用利用 Flex 工具实现词法分析器，识别程序中所有单词，将其转化为单词流。

也就是说：在词法分析中，需要借助 Flex 完成这样一个程序，它的输入是一个 SysY 语言程序，它的输出是每一个文法单元的类别、词素，以及必要的属性。（比如，对于 NUMBER 会有属性它的“数值” 属性；对于 ID 会有它在符号表的 “序号”，有些标识符会有相同的 “序号”。）这需要设计符号表。当然目前符号表项还只是词素等简单内容，但符号表的数据结构，搜索算法词素的保存，保留字的处理等问题都可以考虑了。

综上所述，在词法分析中有两个重要的步骤需要考虑，第一个是正则式的编写以及对识别到的词素进行处理。第二个是符号表的实现，但是考虑到本次实验课的性质，在本次实验中我们将实现一个初步的符号表，当进入语法分析阶段时，我们会根据实际需要，结合模板以及本次实现的符号表做出调整以及修改。

3.1 正则表达式的设计

首先，我们针对一些非终结符设计了正则表达式。第一，我们对不同的数制进行了识别，包括八进制、十进制以及十六进制，通过对其可能涉及的字符的约束设置为大小写不敏感识别，并规定以 0x 为开头识别十六进制，以 0 开头识别八进制。第二，针对标识符，我们根据 sysY 语言的约束进行设计，只有字母以及下划线可以作为标识符开头，但其他部分可以由字母、数字、下划线共同组成。

特别要注意在讨论浮点数的时候要明确对于十进制的浮点数而言可以进行普通的方式进行表示，也可以使用科学计数法的方式进行表示，同样，由于 SysY 语言支持十六进制的运算，那么我们也应该为十六进制设计浮点数的正则表达式

正则表达式示例

```
1
2   HEX (0[xX][0-9a-fA-F]+)
3   OCT (0[0-7]+)
4   DECIMIAL_INT ([1-9][0-9]*|0)
5   DECIMIAL_FLOAT ((([0-9]*[.][0-9]*([eE][+-]?[0-9]+)?)|([0-9]+[eE][+-]?[0-9]+))[fLlL]?)
6   HEX_FLOAT (0[xX](([0-9A-Fa-f]*[.][0-9A-Fa-f]*([pP][+-]?[0-9]+)?)|
7   ([0-9A-Fa-f]+[pP][+-]?[0-9]+))[fLlL]?)
8   ID [[:alpha:]_][[:alpha:][:digit:]_]*
9   EOL (\r\n|\n|\r)
```

```
10  WHITE [\t ]
```

3.2 数制转换

在进行不同进制数字的识别后，为了将数制进行统一，我们统一将各种数制转换为十进制，并利用 C 语言自带的函数 sscanf 进行转换，其类别将保存在符号表中，符号表的具体内容将在下一小节详细展开，再次只需明确，八进制使用 OCT 识别、十六进制使用 HEX 识别，十进制使用 NUM 识别。具体类别使用 type 作为函数参数进行传递。

```
1
2  float changeStr2Num(string s,string type){
3      if(strcmp(type.c_str(),"OCT")==0){
4          int oct=0;
5          sscanf(s.c_str(),"%o",&oct);
6          return oct;
7      }
8      if(strcmp(type.c_str(),"HEX")==0){
9          int hex=0;
10         sscanf(s.c_str(),"%x",&hex);
11         return hex;
12     }
13     if(strcmp(type.c_str(),"NUM")==0){
14         float num=0;
15         sscanf(s.c_str(),"%f",&num);
16         return num;
17     }
18     return 0;
19 }
20
```

3.3 符号表

符号表是本次实验的相对重点，符号表的作用在于在变量或函数被声明时记录其标识符、作用域以及属性，以便在后续调用时从符号表中查询相对应的属性。

根据实验设计，在词法分析阶段的实验中仅需自行给出一个自己设计的符号表，其结构、内容均可由自己指定，而到语法分析实验中中会给出符号表框架。所以在本次实验中，我们以实验指导手册中的一个略复杂测试样例可能的输出结果为蓝本进行符号表的编写，符号表类（SymbolTable）包括一个 map 代表符号表，表中内容包括 string 类型的标识符、以及一个指向标识的指针（Symbol*），还包括一个 Symbol 对象中包括两个内容：他的类型以及作用域，其中作用域用一个指向保存该对象的符号表标识。

当我们遇到一个左大括号的时候，就认为应该创建一个新的符号表，并将指向当前作用域的符号表指向新创建的符号表，因为在其中的符号可能会属于新的作用域，而当遇到一个右大括号时，我们

认为一个作用域已经结束，并把指向当前作用域的指针指向前一个作用域符号表。

符号表的定义：其中包括的函数有构造函数、查找指定符号的函数，没找到则返回空指针以及添加新元素的函数。

```

class SymbolTable
{
private:
    map<string,Symbol*>symbolTable;
    SymbolTable* prevTable;
public:
    SymbolTable(SymbolTable* prev=nullptr){
        this->prevTable=prev;
    }

    Symbol* lookUpSymbol(string symbolName){
        if(symbolTable.find(symbolName)!=symbolTable.end()){
            return symbolTable[symbolName];
        }
        return nullptr;
    }
    void addSymbol(string symbolName,Symbol* symbol){
        symbolTable.insert(pair<string,Symbol*>(symbolName,symbol));
    }
    SymbolTable*getPrev(){
        return prevTable;
    }
};

```

符号的定义，包括他的类型（用一个 int 表示多种不同的类型）以及构造函数，包括为一个变量指定作用域（指定其所在的符号表）

```

class Symbol
{
private:
    /* data */
    int type;
    SymbolTable* inTable;
public:
    Symbol(SymbolTable* addr,int type=0){
```

```
10          this->inTable=addr;
11          this->type=type;
12      }
13      ~Symbol();
14
15      void setInTable(SymbolTable* addr){
16          this->inTable=addr;
17      }
18  };
19
```

在遇到一个 ID 后，要先判断他是否在当前的符号表中，如果不在就将它加入当前符号表。

```
1
2   {ID} {
3   #ifdef ONLY_FOR_LEX
4       string s=yytext;
5       string temp;
6       Symbol* addr;
7       if(currentTable->lookUpSymbol(s)==nullptr)
8           currentTable->addSymbol(s,new Symbol(currentTable));
9
10      addr=currentTable->lookUpSymbol(s);
11      temp="ID\t\t"+s+"\t"+to_string(yylineno)+"\t"+to_string(columnno)+"\t"
12      +to_string(long((void*)addr));
13      columnno+=strlen(yytext);
14      DEBUG_FOR_LAB4(temp);
15  #else
16      return ID;
17  #endif
18
19
```

3.4 状态处理

由于在注释中可能也会出现各种 id 以及正则表达式，但是由于注释中的内容不对编译结果造成影响，所以不能对注释中的元素加入到符号表中，而 lex 给出了一个解决办法就是新声明一个起始状态，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。而状态的切换可以看出通过在之后附加的语法块中通过定义好的宏 BEGIN 来切换，初始状态默认为 INITIAL，因此在结束该状态时我们实际写的是切换回初始状态。还有额外的一点说明%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。

所以我们可以对应编写出下面的正则表达式来识别注释：

```

commentLine (\/\/.*)
commentBegin "/*"
commentElement (.|\n)
commentEnd "*/"
\%x BLOCKCOMMENT

```

在规则段中，通过添加状态来使该规则只对 comment 状态生效：

```

<BLOCKCOMMENT>{commentEnd} {
#ifdef ONLY_FOR_LEX
    columnno+=2;
    DEBUG_FOR_LAB4("COMMENTELEMENT\t\t*/\t"+commentbuffer);
    commentbuffer="";
    DEBUG_FOR_LAB4("COMMENTEND\t\t*/\t"+to_string(yylineno)+"\t"+to_string(columnno-2));
    BEGIN INITIAL;
    #else
        return COMMENTEND;
    #endif
}
<BLOCKCOMMENT>{commentElement} {
    #ifdef ONLY_FOR_LEX
        commentbuffer+=yytext;
        columnno=0;
    #else
        return COMMENTELEMENT;
    #endif
}
```

3.5 规则部分

有了以上内容作为基础保障，在规则部分我们只需要根据识别到的单词进行识别标记、分词后传递给语法分析进行分析即可，大部分的词法分析步骤都是如出一辙的，所以在此不展示全部代码，只展示有代表性的一段代码。

下面的代码很好的展示了词法分析中“分词”的要点，根据输入的词语不同为其打上不同的标签并传递给语法分析进行组合识别后统一处理。

```

"break" {
```

```
    if(dump_tokens)
        DEBUG_FOR_LAB4("BREAK\tbreak");
    return BREAK;
}
"continue" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("CONTINUE\tcontinue");
    return CONTINUE;
}
"const" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("CONST\tconst");
    return CONST;
}
"==" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("EQUAL\t==");
    return EQUAL;
}

"<=" {
    if(dump_tokens)
        DEBUG_FOR_LAB4("LESSOREQUAL\t<=");
    return LESSOREQUAL;
}

```

当然在这个时候我们要特别注意，有一种特殊函数的出现，也就是一些可以不用声明而直接使用的函数的调用，例如 getint、putint 函数等，当我们发现程序中出现这种字符串时要首先将其记录入符号表并做好“函数类型的标记”，这样我们在后面可以直接检测其库函数的特殊性质并对其进行特殊处理。

```

"getint" {//添加库函数的声明，避免语法分析无法识别的情况出现
    if(dump_tokens)
        DEBUG_FOR_LAB4(yytext);
    char *lexeme;
    lexeme = new char[strlen(yytext) + 1];
    strcpy(lexeme, yytext);
    yylval.strtype = lexeme;
    std::vector<Type*> vec;
    std::vector<SymbolEntry*> vec1;
```

```
11      Type* funcType = new FunctionType(TypeSystem::intType, vec,vec1);
12      SymbolTable* st = identifiers;
13      st->install(yytext, se);
14      return ID;
15
```

4 语法分析

语法分析就是根据高级语言的语法规则对程序的语法结构进行分析。语法分析的任务就是在词法分析识别出正确的单词符号串是否符合语言的语法规则，分析并识别各种语法成分，同时进行语法检查和错误处理，为语义分析和代码生成做准备。

语法分析在编译过程中处于核心地位。而语法分析最终想要达到的目的就是对给定的输入符号串根据文法的规则建立起一颗语法树。

在本次实验中，我们使用上下文无关文法来完成语法分析部分的实现。上下文无关法用递归的方式来描述语法规则。语法分析的过程就是按照文法规则对读入的 token 串进行分析的过程。token 串中的每个单词符合对应与文法的一个符号。

根据文法可以手工或自动生成一个有效的语法分析程序，用来判断输入的符号串在语法上是否正确。判断的一句就是对给定的输入符号串能否根据文法规则建立起一颗语法树。上述用推导的方式来考察文法定义语言的过程，但是推导不能表示句子的各个组成部分间的结构关系。语法树能够表示句子的构成和层次关系。语法树就是用一棵树来表示一个句型的推导过程，有时也称为语法分析树。语法树是一棵倒立的树，根在上，枝叶在下。根节点由开始符号标记。随着推导的展开，当某个非终结符被它的候选式所替换时，这个非终结符就产生下一代新结点。

4.1 类型系统及符号表

类型系统将在语法分析以及整个编译器的构造中起到非常重要的作用，类型系统为符号表中的每一个元素（包括函数）都分配了相应的类型，这些类型将在后续起到重要的作用：例如当 int 被当做条件语句进行判定、整形与浮点型相互转换以及语义分析中查看函数返回值、函数参数与声明时是否一致等等都需要用到类型系统。

在本次实验中，我们定义的类型总共有 7 中：包括整型、浮点型、函数类型、数组类型、浮点型以及指针类型，并且针对其中的数值类型还设计了 const 类型与其搭配，尽最大可能覆盖 SysY 语言特性

```
1
2   private:
3       int kind;//描述这个类型的种类
4   protected:
5       enum {INT, VOID, FUNC, FLOAT, ARRAY,BOOL,PTR};
6       int size;
```

其中较为简单明确的实现内容包括 int、float 以及 ptr 类型，他们所表示的内容单一且有限，实现他们难度不大

```

class IntType : public Type
{
private:
    bool isConst;
public:
    //这个 size 的改法可能有点草率
    IntType(int size, bool isConst=false) : Type(Type::INT, size), isConst(isConst){};
    std::string toStr();
};
```

而其中较为困哪复杂的部分则为 func 以及数组 array，fun 因为需要保存的东西较多，且需要和符号表进行配合，所以会略微复杂。

在 func 中需要对函数的返回值、参数进行保存，而保存的方式其实也就是对参的 Symbolentry 进行保存，等需要时迭代访问 vector 即可，而对返回类型的保存则采用 Type 类型的保存即可。而由于多维数组需要考虑数组套数组的情况，所以需要一个指针来表示本次数组的元素所表示的内容，其内容可能为一个个 int 或 float 的值，也就是 int*，float*，也可能是指向一维数组的指针，这样就可以完成对多维数组的保存以及读取。

```

class FunctionType : public Type
 {
    private:
     Type* returnType;
     std::vector<Type*> paramsType;
     std::vector<SymbolEntry*> paramsSe;

    public:
     FunctionType(Type* returnType,
                  std::vector<Type*> paramsType,
                  std::vector<SymbolEntry*> paramsSe)
         : Type(Type::FUNC),
           returnType(returnType),
           paramsType(paramsType),
           paramsSe(paramsSe){};
     void setParamsType(std::vector<Type*> paramsType) {
         this->paramsType = paramsType;
     };
     std::vector<Type*> getParamsType() { return paramsType; };
     std::vector<SymbolEntry*> getParamsSe() { return paramsSe; };
     Type* getRetType() { return returnType; };
```

```
23      std::string toStr();
24  };
25
26  class ArrayType : public Type {
27     private:
28      Type* elementType;
29      int length;
30      bool isConst;
31      Type* arrayType =nullptr;//要考虑多维数组的情况，数组套数组
32
33     public:
34      ArrayType(Type* elementType,
35                int length,
36                bool constant = false)
37          : Type(Type::ARRAY,elementType->getSize()*length),
38            elementType(elementType),length(length),isConst(constant) {};
39      std::string toStr();
40      int getLength() const { return length; };
41      Type* getElementType() const { return elementType; };
42      void setArrayType(Type* arrayType) { this->arrayType = arrayType; };
43      Type* getArrayType() const { return arrayType; };
44      int getSize() const {return size;}
45  };
46
```

4.2 抽象语法树

语法分析的目的是构建出一棵抽象语法树（AST），因此我们需要设计语法树的结点。结点分为许多类，除了一些共用属性外，不同类结点有着各自的属性、各自的子树结构、各自的函数实现。我们可以简单用 struct 去涵盖所有需要的内容，也可以设计复杂的继承结构。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。

其中一些抽象语法树的功能比较简单，实现也较为轻松，例如 If、Ifelse、while、break、continue 等，他们通常格式比较单一，能够完成的功能也有限。例如下面的两个节点的定义，其中 ifelse 需要的就是把两个分支语句的节点传入然后分别进行打印即可，while 也是一样，把判断条件、需要执行的代码全部录入后输出即可，因为二者都具有固定的调用格式，所以其代码实现也较为容易。

```
1
2  class IfElseStmt : public StmtNode
3   {
4   private:
5       ExprNode *cond;
```

```
6       StmtNode *thenStmt;
7       StmtNode *elseStmt;
8   public:
9       IfElseStmt(ExprNode *cond, StmtNode *thenStmt, StmtNode *elseStmt) : cond(cond), thenStmt(
10      void output(int level);
11  };
12
13  class WhileStmt : public StmtNode
14  {
15  private:
16      ExprNode *cond;
17      StmtNode *stmt;
18
19      BasicBlock* cond_bb, *end_bb;
20
21  public:
22      WhileStmt(ExprNode *cond, StmtNode *stmt) : cond(cond), stmt(stmt){cond_bb=end_bb=nullptr;
23      void setStmt(StmtNode* stmt){this->stmt=stmt;};
24      void output(int level);
25  };
26
```

较为复杂的结点包括 ID、Assign、Decl 等等，因为他们通常都没有固定的套路，例如 ID 可能是代表着 func、int、float、数组等等任意一种，所以其可能性非常丰富，需要考虑很多种可能性。

例如在 ID 中就需要完成非常多的任务，例如保存这个 ID 是否为左值使用，还是只是用作赋值的在等号右侧进行使用的，如果是一个数组类型那么就需要对 arrayIndices 进行初始化，同时还需要提供判断其类型、取出其大小的对外接口，这都让 ID 的节点变的较为复杂。

```
1
2   class Id : public ExprNode
3   {
4   private:
5       bool left;
6       ExprNode *arrayIndices;
7   public:
8       Id(SymbolEntry *se ,ExprNode *arrayIndices=nullptr) :
9       ExprNode(se),arrayIndices(arrayIndices){
10          left=false;
11          if(se){
12              if(getType()->isInt()){
13                  SymbolEntry* temp = new TemporarySymbolEntry(TypeSystem::intType,
14                  SymbolTable::getLabel());
```

```
                dst = new Operand(temp);
            }else if(getType()->isFloat()){
                SymbolEntry* temp = new TemporarySymbolEntry(TypeSystem::floatType,
                SymbolTable::getLabel());
                dst = new Operand(temp);
            }else if(getType()->isArray()){
                SymbolEntry* temp = new TemporarySymbolEntry(new
                PointerType(dynamic_cast<ArrayType*>(getType())->getElementType()),
                SymbolTable::getLabel());
                dst = new Operand(temp);
            }
        }
    };
    SymbolEntry* getSymbolEntry(){return symbolEntry;};
    void output(int level);
    std::string getValue(){return symbolEntry->toStr();}
    int getIValue(){return (int)getFValue();};
    //int getIValue(){return 0;};
    float getFValue(){
        SymbolTable* nowTable=identifiers;
        while(identifiers){
            if(identifiers->lookup(getValue())){
                return ((IdentifierSymbolEntry*)(identifiers->lookup(getValue())))
                ->getValue();
            }
            else{
                nowTable=nowTable->getPrev();
            }
        }
        return 0.0;//这里就是没找到，但不知道该怎么处理
    }
    void setLeft(){left=true;};
};
```

当然，在语法分析阶段最重要的其实是构造一个正确的语法树，所谓正确的语法树，就是这棵树的父子节点关系都是正确的，因为接下来的中间代码生成、ARM 代码生成都需要基于对 AST 的遍历后得到的，而在本次实验中，只需要对 AST 的各节点有一个基本的描述，并能输出这个节点的基本信息即可。

4.3 语法分析与语法树的创建

词法分析得到的，实质是语法树的叶子结点的属性值，语法树所有结点均由语法分析器创建。在自底向上构建语法树时（与预测分析法相对），我们使用孩子结点构造父结点。在 yacc 每次确定一个

产生式发生归约时，我们会创建出父结点、根据子结点正确设置父结点的属性、记录继承关系.

语法分析与语法树的创建是本次实验的重点实现部分，有了上面关于符号表以及 AST 节点的类定义以及基本实现，我们就可以在语法分析的阶段根据读取到的内容进行节点类的声明并将他们插入到 AST 中。在这里我会对一些比较重要、具有代表性的语法分析过程进行介绍。

4.3.1 Lval 表达式

Lval 表达式的功能是处理那些左值表达式，也就是被赋值的那些变量，这个变量只能是 ID，或者数组，并且我们需要在符号表中记录出这些 ID 的出现属于左值，并且 new 一个对应的 AST 节点

```
1
2   //左值表达式，暂时只包括 id
3   LVal
4       : ID {
5           SymbolEntry *se;
6           se = identifiers->lookup($1);
7           if(se == nullptr)
8           {
9               fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1);
10              delete [](char*)$1;
11              assert(se != nullptr);
12          }
13          Id *id= new Id(se);
14          id->setLeft();
15          $$ = id;
16          delete []$1;
17      }
18      |
19      ID ArrayIndices{
20          SymbolEntry *se;
21          se = identifiers->lookup($1);
22          if(se == nullptr)
23          {
24              fprintf(stderr, "identifier \"%s\" is undefined\n", (char*)$1);
25              delete [](char*)$1;
26              assert(se != nullptr);
27          }
28          Id *id = new Id(se,$2);
29          id->setLeft();
30          $$ = id;
31          delete []$1;
32      }
```

```
33      ;
34
```

4.3.2 while 表达式

while 表达式需要做的就是对输入的词法分析进行切割，因为我们在之前提及过，whilestmt 的过程其实比较区域标准化，我们只需要分割出判断的条件是哪一部分，while 循环体的 stmt 是哪一部分然后分别进行初始化，并将其作为参数传入构造函数就可以。

```
1
2   WhileStmt
3   : WHILE LPAREN Cond RPAREN {
4       if(!$3->getType()->isBool()){
5           SymbolEntry *se_temp = new TemporarySymbolEntry(TypeSystem::boolType,
6           SymbolTable::getLabel());
7           SymbolEntry *se_zero = new ConstantSymbolEntry(TypeSystem::intType, 0);
8           $3 = new BinaryExpr(se_temp, BinaryExpr::UNEQUAL, $3, new Constant(se_zero));
9       }
10      WhileStmt* whileStmt = new WhileStmt($3,nullptr);
11      $<stmttype>$ = whileStmt;
12      whileStack.push(whileStmt);
13  }
14  Stmt {
15      StmtNode *whileStmt = $<stmttype>5;
16      ((WhileStmt*)whileStmt)->setStmt($6);
17      $$=whileStmt;
18      whileStack.pop();
19  }
20
```

4.3.3 unary 表达式

unary 表达式是语法分析中比较复杂的一个部分，因为其可能性比较多，他可能是一个简单的表达式，这就直接调用其他的表达式判断，或者是函数表达式，又或者是对 unary 自身的加、减、取反的递归操作，这些都可以统一被归结于 unary 表达式

对于不同的情况，也应该给予不同的考虑，最基本的就是如果是一个函数表达式则需要提取将这个函数保存在符号表中所应该必须具备的所有信息，比如函数参数、函数返回值等等。而如果是表达式的话，则需要 new 对应的 unary 将其插入到语法树中，注意在 new 新的节点时需要注意他的运算类型以及传入正确的 symboltable 的 label，然后根据后面的 unary 将其插入到语法树中。

```
1   UnaryExp
2   :
```

```
PrimaryExp {
    $$ = $1;
}
//调用函数是一个一元表达式
| ID LPAREN FuncRParams RPAREN{

    SymbolEntry* se;
    se = identifiers->lookup($1);
    if(se == nullptr)
    {
        fprintf(stderr, "function \"%s\" is undefined\n", (char*)$1);
        delete [](char*)$1;
        assert(se != nullptr);
    }
    std::vector<Type*> paramsType = ((FunctionType*)(se->getType()))
    ->getParamsType();
    ExprNode* callParamsType = $3;
    int paramCount = 0;

    while(callParamsType){
        paramCount++;
        callParamsType = (ExprNode*)callParamsType->getNext();
    }

    while(se){
        if(((FunctionType*)(se->getType()))->getParamsType().size()
        == paramCount){
            break;
        }
        se = se->getNext();
    }
    if(se == nullptr){
        fprintf(stderr, "function \"%s\" having %d params,
        is undefined\n", (char*)$1, paramCount);
        delete [](char*)$1;
        assert(se != nullptr);
    }

    callParamsType = $3;

    for(auto &params:paramsType){
        if(params != callParamsType->getSymbolEntry()->getType()){
```

```
45              fprintf(stderr, "your param TYPE should be %s,
46              but when you call func you use %s\n",
47                      params->toStr().c_str(),callParamsType->
48                      getSymbolEntry()->getType()->toStr().c_str());
49          }
50          callParamsType = (ExprNode*)callParamsType->getNext();
51      }
52      $$ = new FuncExpr(se,$3);
53  }
54  | ADD UnaryExp{
55      SymbolEntry *se;
56      if($2->getType()->isFloat())
57          se = new TemporarySymbolEntry(TypeSystem::floatType,
58          SymbolTable::getLabel());
59      else
60          se = new TemporarySymbolEntry(TypeSystem::intType,
61          SymbolTable::getLabel());
62
63      $$ = new UnaryExpr(se,UnaryExpr::ADD, $2);
64  }
65  | SUB UnaryExp{
66      SymbolEntry *se;
67      if($2->getType()->isFloat())
68          se = new TemporarySymbolEntry(TypeSystem::floatType,
69          SymbolTable::getLabel());
70      else
71          se = new TemporarySymbolEntry(TypeSystem::intType,
72          SymbolTable::getLabel());
73      $$ = new UnaryExpr(se, UnaryExpr::SUB, $2);
74
75  }
76  | NOT UnaryExp{
77      SymbolEntry *se;
78      se = new TemporarySymbolEntry(TypeSystem::boolType,
79      SymbolTable::getLabel());
80      $$ = new UnaryExpr(se, UnaryExpr::NOT, $2);
81  }
```

4.3.4 VarDef 表达式与 initial 表达式

函数定义与研与初始化语言是语法分析中较为复杂的部分，这部分复杂的语言主要因为他的可能性比较丰富，而且变量的声明可能包含数组，而多维数组的出现则让程序的编写难度极大程度提升，所

以这两个部分是语法分析中的难点。

首先，变量的声明包括变量只声明不赋值、变量声明且赋值、数组声明、数组声明且赋值的情况，其中前两种比较简单，因为只需要对一个 SymbolEntry 进行操作，需要做的操作也是首先判断变量的类型，根据类型首先初始化一个符号表项出来，如果不存在赋值则直接插入到语法树中，但是如果包括赋值语句的话那么就还需要取出他的具体值并赋值后再插入到抽象语法树中。

对于数组的初始化就显得更加复杂，因为需要把数组一层一层的剥离开然后进行存储，之后再一层一层自底向上进行指针的不断指向，让最底层真正存储 int 以及 float 的表格项存储初始化的内容后再调取对应的数组层数、维度等信息进行初始化，随后将其插入到语法树中。

对于 initial 语句，同样具有很多种可能，可能是一个表达式，而这个表达式包括函数调用以及平常我们理解的加减乘除表达式，同样也可能是一个常数值。较为复杂的部分也是出现在对数组的初始化上，其初始化可能仅仅包括两个大括号，大括号之间也可能具有表达式列表，我们需要将这些表达式按照位置放置在数组声明时对应的位置上即可完成对数组内容的初始化。

```
VarDefStmt
    :
//只定义一个变量，不赋值
ID {
    SymbolEntry* se;
    if(nowType->isInt()){
        se = new IdentifierSymbolEntry(TypeSystem::intType, $1,
        identifiers->getLevel());
    }else{
        if(nowType->isFloat()){
            se = new IdentifierSymbolEntry(TypeSystem::floatType, $1, identifiers->getLevel())
        }
    }
    if (!identifiers->install($1, se)){
        fprintf(stderr, "identifier \"%s" is already defined\n", (char*)$1);
        assert(false);
    }
    $$ = new DeclStmt(new Id(se));
    delete []$1;
}
//定义一个变量并赋值
|
ID ASSIGN InitVal {
    SymbolEntry*se;
        if(nowType->isInt()){
            se=new IdentifierSymbolEntry(TypeSystem::intType,
            $1,identifiers->getLevel());
            ((IdentifierSymbolEntry*)se)->setValue($3->getIValue());
    }else{
```

```
30          if(nowType->isFloat()){
31              se = new IdentifierSymbolEntry(TypeSystem::floatType, $1, identifiers->getLevel())
32              ((IdentifierSymbolEntry*)se)->setFValue($3->getFValue());
33          }
34      }
35      if (!identifiers->install($1, se)){
36          fprintf(stderr, "identifier \"%s\" is already defined\n", (char*)$1);
37          assert(false);
38      }
39      $$ = new DeclStmt(new Id(se),$3);
40      delete []$1;
41  }
42  ;
43  InitVal
44  :
45  Exp{
46      //需要补充
47      $$=$1;
48      if(!arrayStack.empty()){
49          //初始化数组
50          arrayStack.top()->addExpr($1);
51      }
52
53  }
54  | LBRACE RBRACE{
55      SymbolEntry* se;
56      if(!arrayStack.empty()){
57          currentArray = (ArrayType*)(((ArrayType*)(arrayStack.top()->
58          getSymbolEntry()->getType()))->getElementType());
59      }
60      se = new ConstantSymbolEntry(currentArray);
61      ArrayInit *arrayNode = new ArrayInit(se,true);
62      if(!arrayStack.empty())
63          arrayStack.top()->addExpr(arrayNode);
64
65      if(!arrayStack.empty())
66          currentArray=(ArrayType*)(((ArrayType*)(arrayStack.top()->
67          getSymbolEntry()->getType()))->getElementType());
68      $$=arrayNode;
69
70  }
71  ;
```

72

4.3.5 递归逻辑

除此之外，也有一些复杂的实现如函数参数的确定之所以复杂是基于数组的，在这里就不做过多的赘述，但有一些基于逻辑的比较复杂的语法分析实现还是要在这里有所提及，因为语法分析很多时候都涉及到递归的思想进行处理，尤其是在处理表达式时这种思想体现的更为明显，例如对于不同表达式类型的递归：UnaryExp–>PrimaryExp–>AddExp–>MulExp–>UnaryExp，这样的递归链条保证了代码可以识别 SysY 语言中大部分的复杂表达式语句。

5 类型检查及中间代码生成

在该部分需要在之前构造好的语法树的基础上，进行类型检查，检测出代码中的一些错误并进行错误信息的打印，之后需要进行中间代码的生成，在中间代码的基础上，就可以进行一系列代码的优化工作。

5.1 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。

具体的需要在类型检查工作中进行判定的内容有：
- 变量未声明，及在同一作用域下重复声明
- 条件判断表达式 int 至 bool 类型隐式转换
- 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 的函数调用结果参与了某表达式计算)
- 函数未声明，及不符合重载要求的函数重复声明
- 函数调用时形参及实参类型或数目的不一致
- return 语句操作数和函数声明的返回值类型是否匹配
- 实现了数组要求的同学，还可以对数组维度进行相应的类型检查
- 实现了 break、continue 语句要求的同学，还可以对 break、continue 语句进行静态检查，判断是否仅出现在 while 语句中

类型检查困难的地方在于他可能位于整个工程的任意一个位置，比如语法分析的.y 文件中或者 AST 中都可能成为类型检查的位置，选取一个好的位置进行对应逻辑的代码编写很可能达到事半功倍的效果。

5.1.1 定义问题

定义问题是类型检查中非常重要的一部分，比如变量名的重定义、使用未定义的变量名等等，这部分的判定放在.y 文件是最合适的，因为在语法分析阶段我们需要将这些变量或者函数插入 symboltable 中，在这个过程中，我们可以检查符号表中是否存在对应的变量名来判定重定义或未定义问题：

```

  if (!identifiers->install($1, se)){
          fprintf(stderr, "identifier \"%s\"
          is already defined\n", (char*)$1);
          assert(false);
      }

```

5.1.2 函数问题

函数相关的问题比如函数的未定义或使用未定义的函数问题可以直接参考变量的定义问题，而这里所说的函数问题主要包括函数参数不匹配以及返回值与期望不一致等。函数参数的问题同样可以在语法分析解决，因为在构造语法树时我们可以获取函数的参数，这样就可以在符号表中进行查找与比对。

而函数返回值的问题则放在遍历语法树时实现最好，因为这样我可以明确的知道 return 语句后面 return 了什么，还是这个函数压根没有 return 任何东西，这些信息只有在遍历语法树时才能够高效的获得，而在语法分析时是不能准确的知道的。

```

  if(se == nullptr){
          fprintf(stderr, "function \"%s\" having %d params,
          is undefined\n", (char*)$1, paramCount);
          delete [](char*)$1;
          assert(se != nullptr);
      }

      callParamsType = $3;

      for(auto &params:paramsType){
          if(params != callParamsType->getSymbolEntry()->getType()){
              fprintf(stderr, "your param TYPE should be
              %s, but when you call func you use %s\n",
                      params->toStr().c_str(),callParamsType
                      ->getSymbolEntry()->getType()->toStr().c_str());
          }
          callParamsType = (ExprNode*)callParamsType->getNext();
      }
      $$ = new FuncExpr(se,$3);

  void FunctionDef::typeCheck()
  {

```

```
24      ret = false;
25
26      Type* funcRetType=((FunctionType*)(this->se->getType()))
27      ->getRetType();
28      if(this->stmt==nullptr && funcRetType!=TypeSystem::voidType){
29          fprintf(stderr, "function %s must return a vaild
30          value\n",this->se->toStr().c_str());
31          return;
32      }
33
34      retType = funcRetType;
35      if(this->stmt != nullptr)
36          this->stmt->typeCheck();
37
38      if(!funcRetType->isVoid() && !ret){
39          fprintf(stderr, "function \"%s\" shoule return a vaild
40          type\n",this->se->toStr().c_str());
41
42          assert(!funcRetType->isVoid() && !ret == false);
43      }
44  }
45
```

5.1.3 控制流语句问题

控制流语句问题是最简单的一种判断，只需要检查 continue、break 函数的出现是否在 while 函数内就可以。

```
1
2   BreakStmt
3       : BREAK SEMICOLON {
4           if(whileStack.empty()){
5               fprintf(stderr,"break stmt is not in a while stmt\n");
6               assert(!whileStack.empty());
7           }
8           $$ = new BreakStmt(whileStack.top());
9       }
10      ;
11  ContinueStmt
12      : CONTINUE SEMICOLON {
13          if(whileStack.empty()){
14              fprintf(stderr,"continue stmt is not in a while stmt\n");
```

```
15              assert(!whileStack.empty());
16          }
17          $$ = new ContinueStmt(whileStack.top());
18      }
19      ;
```

5.2 中间代码生成

词法分析和语法分析是编译器的前端，中间代码是编译器的中端，目标代码是编译器的后端，通过将不同源语言翻译成同一中间代码，再基于中间代码生成不同架构的目标代码，我们可以极大的简化编译器的构造。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。

5.2.1 控制流的翻译

控制流的翻译是中间代码生成的难点，实验通过回填技术来完成控制流的翻译。我们为每个结点设置两个综合属性 true_list 和 false_list, 它们是跳转目标未确定的跳转指令的列表，回填是指当跳转的目标基本块确定时，设置列表中跳转指令的跳转目标为该基本块。

逻辑与具有短路的特性，当第一个子表达式的值为假时，整个布尔表达式的值为假，第二个子表达式不会执行；当第一个子表达式的值为真时，根据第二个子表达式的值得到整个布尔表达式的值。在代码中，我们首先创建一个基本块 trueBB, 它是第二个子表达式生成的指令需要插入的位置，然后生成第一个子表达式的中间代码，在第一个子表达式生成中间代码的过程中，生成的跳转指令的目标基本块尚不能确定，因此会将其插入到子表达式结点的 true_list 和 false_list 中。在翻译当前布尔表达式时，我们已经能确定 true_list 中跳转指令的目的基本块为 trueBB，因此进行回填。我们再设置第二个子表达式的插入点为 trueBB，然后生成其中间代码。最后，因为当前仍不能确定子表达式二的 true_list 的目的基本块，因此我们将其插入到当前结点的 true_list 中，我们也不能知道两个子表达式的 false_list 的跳转基本块，便只能将其插入到当前结点的 false_list 中，让父结点回填当前结点的 true_list 和 false_list。

```

void IfElseStmt::genCode()
{
    // Todo
    Function *func;
    BasicBlock *then_bb, *else_bb, *end_bb;
    func = builder->getInsertBB()->getParent();
    then_bb = new BasicBlock(func);
    else_bb = new BasicBlock(func);
    end_bb = new BasicBlock(func);

    cond->genCode();
    backPatch(cond->trueList(), then_bb);
    backPatch(cond->falseList(), else_bb);
```

```
15
16      builder->setInsertBB(then_bb);
17      thenStmt->genCode();
18      then_bb = builder->getInsertBB();
19      new UncondBrInstruction(end_bb, then_bb);
20
21      builder->setInsertBB(else_bb);
22      elseStmt->genCode();
23      else_bb = builder->getInsertBB();
24      new UncondBrInstruction(end_bb, else_bb);
25
26      builder->setInsertBB(end_bb);
27  }
28
29  void WhileStmt::genCode()
30  {
31      Function* func = builder->getInsertBB()->getParent();
32      BasicBlock *cond_bb=new BasicBlock(func), *while_bb=new BasicBlock(func),
33                  *end_bb=new BasicBlock(func), *bb=builder->getInsertBB();
34
35      this->setCondBB(cond_bb);
36      this->setEndBB(end_bb);
37      new UncondBrInstruction(cond_bb, bb);
38      builder->setInsertBB(cond_bb);
39      this->cond->genCode();
40
41      backPatch(cond->trueList(),while_bb);
42      backPatch(cond->falseList(),end_bb);
43
44      builder->setInsertBB(while_bb);
45      this->stmt->genCode();
46      while_bb = builder->getInsertBB();
47      new UncondBrInstruction(cond_bb,while_bb);
48
49      builder->setInsertBB(end_bb);
50  }
51
```

5.2.2 unary 的翻译

另外一个非常常用也非常重要的语句翻译就是算数运算的翻译，在 unary 中我们需要根据具体的类型完成 gencode 的编写，在这里同样需要对操作数的类型进行判定，根据运算的不同分开讨论。

```
void UnaryExpr::genCode()
{
    BasicBlock* bb = builder->getInsertBB();
    expr->genCode();
    Type* unaryExprType = expr->getType();
    Operand* src = expr->getOperand();
    Operand* zero = nullptr;
    switch (op)
    {
    case SUB:
        if(unaryExprType->isFloat())
            zero = new Operand(new ConstantSymbolEntry
            (TypeSystem::floatType,0.0f));
        else if(unaryExprType->isBool()){
            zero = new Operand(new ConstantSymbolEntry
            (TypeSystem::intType,0));
            src = new Operand(new TemporarySymbolEntry
            (TypeSystem::intType, SymbolTable::getLabel()));
            new ZextInstruction(src, expr->getOperand(), bb);
        }
        else
            zero = new Operand(new ConstantSymbolEntry
            (TypeSystem::intType,0));
        new BinaryInstruction(BinaryInstruction::SUB,
        dst, zero, src, bb);
        break;
    case ADD:
        fprintf(stderr,"not finish yet\n");
        break;
    case NOT:
        if(unaryExprType->isInt() || unaryExprType
        ->isFloat()){
            Operand* temp = new Operand(new TemporarySymbolEntry
            (TypeSystem::boolType, SymbolTable::getLabel()));
            new CmpInstruction(CmpInstruction::E, temp,
            src,new Operand(new ConstantSymbolEntry
            (TypeSystem::intType, 0)),bb);
            src = temp;
        }
        new XorInstruction(dst, src, bb);
    }
```

```
42  }
```

5.2.3 调用指令打印

在本次实验中，还有一部分非常重要的工作就是完成 ll 的打印，打印工作则在 instruction 中完成，我们需要做的就是根绝在 AST 中生成的关于指令的构造函数构造的指令类所提供的信息真正完成对于 llvm 指令的打印

```
1  void CallInstruction::output() const
2  {
3      fprintf(yyout, "  ");
4      if (operands[0])
5          fprintf(yyout, "%s = ", operands[0]->toStr().c_str());
6      FunctionType* type = (FunctionType*)(func->getType());
7      fprintf(yyout, "call %s %s(", type->getRetType()->toStr().c_str(),
8              func->toStr().c_str());
9      for (long unsigned int i = 1; i < operands.size(); i++) {
10         if (i != 1)
11             fprintf(yyout, ", ");
12         fprintf(yyout, "%s %s", operands[i]->getType()->toStr().c_str(),
13                 operands[i]->toStr().c_str());
14     }
15     fprintf(yyout, ")\n");
16 }
17
18 void CondBrInstruction::output() const {
19     std::string cond, type;
20     cond = operands[0]->toStr();
21     type = operands[0]->getType()->toStr();
22     int true_label = true_branch->getNo();
23     int false_label = false_branch->getNo();
24     fprintf(yyout, "  br %s %s, label %%B%d, label %%B%d\n", type.c_str(),
25             cond.c_str(), true_label, false_label);
26 }
27
```

6 ARM 代码生成

在最后一步中需要完成的工作有两项，第一项就是完成线性扫描的代码补全，第二部分是完成 genMachinecode 的代码填写，第二部分中又分为两部分，需要先在遍历 AST 时完成中间代码向最终 ARM 机器码的转换，但只是通过构造函数完成类的构造以及排序，具体的将不同的指令类进行输出打印则是最终的步骤。

6.1 线性扫描

在本次实验中我们需要完成线性扫描寄存器分配算法，单趟遍历每个活跃区间 (Interval)，为其分配物理寄存器。其具体分为以下三个步骤:

第一部分为活跃寄存器分析，在前一步的目标代码生成过程中，已经为所有临时变量分配了一个虚拟寄存器 VREG。在这一步需要为每个 VREG 计算活跃区间。这一步实验所给的框架代码已经给出，我们只需要使用其结果即可

第二部分就是寄存器分配，intervals 表示还未分配寄存器的活跃区间，其中所有的 interval 都按照开始位置进行递增排序；active 表示当前正在占用物理寄存器的活跃区间集合，其中所有的 interval 都按照结束位置进行递增排序。

现在开始遍历 intervals 列表进行寄存器分配，对任意一个 unhandledinterval 都进行如下的处理:
1. 遍历 active 列表，看该列表中是否存在结束时间早于 unhandledinterval 的 interval（即与当前 unhandled interval 的活跃区间不冲突），若有，则说明此时为其分配的物理寄存器可以回收，可以用于后续的分配，需要将其在 active 列表删除；
2. 判断 active 列表中 interval 的数目和可用的物理寄存器数目是否相等:
(a) 若相等，则说明当前所有物理寄存器都被占用，需要进行寄存器溢出操作。具体为在 active 列表中最后一个 interval 和当前 unhandledinterval 中选择一个 interval 将其溢出到栈中，选择策略就是看谁的活跃区间结束时间更晚，如果是 unhandled interval 的结束时间更晚，只需要置位其 spill 标志位即可，如果是 active 列表中的 interval 结束时间更晚，需要置位其 spill 标志位，并将其占用的寄存器分配给 unhandled interval，再按照 unhandled interval 活跃区间结束位置，将其插入到 active 列表中。
(b) 若不相等，则说明当前有可用于分配的物理寄存器，为 unhandled interval 分配物理寄存器之后，再按照活跃区间结束位置，将其插入到 active 列表中即可。

```
1
2  bool LinearScan::linearScanRegisterAllocation() {
3          bool ret=true;
4          doubleClear();
5          for(int i=4;i<=10;i++){
6              regs.push_back(i);
7          }
8          for(auto& i :intervals){
9              expireOldIntervals(i);
10             if(regs.empty()){
11                 spillAtInterval(i);
12                 ret=false;
13             }
14             else{
15                 i->rreg=regs.front();
16                 regs.erase(regs.begin());
17                 active.push_back(i);
18                 sort(active.begin(), active.end(), compareEnd);
```

```
            }
        }
        return ret;
}
```

第三部分就是生成溢出代码，在上一步寄存器分配结束之后，如果没有临时变量被溢出到栈内，那寄存器分配的工作就结束了，所有的临时变量都被存在了寄存器中；若有，就需要在操作该临时变量时插入对应的 LoadMInstruction 和 StoreMInstruction，其起到的实际效果就是将该临时变量的活跃区间进行了切分，以便重新为其进行寄存器分配

具体分为以下三个步骤：

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；

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；插入结束后，会迭代进行以上过程，重新活跃变量分析，进行寄存器分配，直至没有溢出情况出现。

```

void LinearScan::spillAtInterval(Interval* interval) {
    auto spill = active.back();
    if (spill->end > interval->end) {
        spill->spill = true;
        interval->rreg = spill->rreg;
        active.push_back(interval);
        sort(active.begin(), active.end(), compareEnd);
    } else {
        interval->spill = true;
    }
}
```

了解了寄存器线性扫描法分配的具体流程，我们就可以完成对完整的线性扫描代码的编写。

```

void LinearScan::genSpillCode() {
    for (auto& interval : intervals) {
        if (!interval->spill)
            continue;

        interval->disp=-func->myAllocSpace(0,4,8);
        MachineOperand* mo = lsgetImmInt(interval->disp);
        MachineOperand* mo2 = lsgetReg(11);

```

```
11          for(MachineOperand* it:interval->uses){
12              MachineOperand* mo3 = new MachineOperand(*it);
13              MachineOperand* mo4=nullptr;
14              if(abs(interval->disp)>=256){
15                  mo4=lsgetVreg(SymbolTable::getLabel());
16                  MachineInstruction*mi=new LoadMInstruction
17                  (lsgetBlock(it),mo4,mo);
18                  lsInsertBefore(it,mi,0);
19              }
20              if(mo4!=nullptr){
21                  MachineInstruction*mi=new LoadMInstruction
22                  (lsgetBlock(it),mo3,mo2,new MachineOperand(*mo4));
23                  lsInsertBefore(it,mi,0);
24              }
25              else{
26                  MachineInstruction*mi = new LoadMInstruction(lsgetBlock(it),mo3,mo2,mo);
27                  lsInsertBefore(it,mi,0);
28              }
29          }
30          for(MachineOperand*it:interval->defs){
31              MachineOperand*mo3 = new MachineOperand(*it);
32              MachineOperand*mo4=nullptr;
33              MachineInstruction*mi=nullptr;
34              MachineInstruction* mi1=nullptr;
35              if(abs(interval->disp)>=256){
36                  mo4=lsgetVreg(SymbolTable::getLabel());
37                  mi1=new LoadMInstruction(lsgetBlock(it),mo4,mo);
38                  lsInsertAfter(it,mi1,0);
39              }
40              if(mo4!=nullptr){
41                  mi=new StoreMInstruction(lsgetBlock(it),mo3,mo2,new MachineOperand(*mo4));
42              }
43              else{
44                  mi = new StoreMInstruction(lsgetBlock(it),mo3,mo2,mo);
45              }
46              if(mi1!=nullptr){
47                  mi1->insertAfterBlock(mi);
48              }
49              else{
50                  lsInsertAfter(it,mi,0);
51              }
52          }
```

```
53
54      }
55  }
```

6.2 ARM 汇编码生成

6.2.1 机器码指令类构造

由于机器码指令类较为繁多，于是在此找出三个比较具有代表性的进行分析。

首先是访存指令，对于访存指令，具体可以分为以下三种情况：

1. 加载一个全局变量或者常量

对于这种情况，同学们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. 加载一个栈中的临时变量

由于在 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，只需要以 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。

store 和 load 的情况分类基本相同，只需要模仿并稍作修改就可以得到对应的机器码指令类的构造方式：

```
1
2   void LoadInstruction::genMachineCode(AsmBuilder* builder) {
3       IdentifierSymbolEntry*ise = (IdentifierSymbolEntry*)
4       (operands[1]->getEntry());
5       if(operands[1]->getEntry()->isVariable() && fastIsGlobal(ise)){
6           MachineOperand* d = genMachineOperand(operands[0]);
7           MachineOperand* t = genMachineVReg();
8           MachineOperand* s = genMachineOperand(operands[1]);
9           MachineInstruction* mi = new LoadMInstruction
10          (builder->getBlock(),t,s);
11          builder->getBlock()->InsertInst(mi);
12          mi = new LoadMInstruction(builder->getBlock(),d,t);
13          builder->getBlock()->InsertInst(mi);
14          return ;
15      }
16      if(operands[1]->getEntry()->isTemporary() &&  fastDef(operands[1]) && operands[1]->getDef(
17          MachineOperand* d = genMachineOperand(operands[0]);
18          int o = ((TemporarySymbolEntry*)(operands[1]->getEntry()))
19          ->getOffset();
20          MachineOperand* s2 = getImmInt(o);
21          MachineOperand* s = getReg(11);
```

```
22          if(abs(o)>=256){
23              MachineOperand* t= genMachineVReg();
24              builder->getBlock()->InsertInst(new
25              LoadMInstruction(builder->getBlock(),t,s2));
26              s2 = getImmInt(256);
27              builder->getBlock()->InsertInst(new
28              LoadMInstruction(builder->getBlock(),t,s2));
29              s2 = getImmInt(255);
30              builder->getBlock()->PopInst();
31              s2 = t;
32          }
33          MachineInstruction* mi = new LoadMInstruction
34          (builder->getBlock(),d,s,s2);
35          builder->getBlock()->InsertInst(mi);
36          return ;
37      }
38
39      auto d = genMachineOperand(operands[0]);
40      auto s = genMachineOperand(operands[1]);
41      MachineInstruction* mi = new LoadMInstruction(builder->getBlock()
42      , d, s);
43      builder->getBlock()->InsertInst(mi);
44
```

控制流指令的实现

1. UncondBrInstruction

对于 UncondBrInstruction，只需要生成一条无条件跳转指令即可，至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，首先明确在中间代码中该指令一定位于 CmpInstruction 之后，对 CmpInstruction 的翻译比较简单。对 CondBrInstruction，首先需要在 AsmBuilder 中添加成员以记录前一条 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 FalseBranch。

3. RetInstruction

对于 RetInstruction 需要考虑的情况比较多。首先，当函数有返回值时，我们需要生成 MOV 指令，将返回值保存在 R0 寄存器中；其次，我们需要生成 ADD 指令来恢复栈帧，(如果该函数有 Callee saved 寄存器，我们还需要生成 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来返回到 Caller。

```
1
2  void UncondBrInstruction::genMachineCode(AsmBuilder* builder) {
3      std::string temp="";
```

```
    temp.append(".L");
    temp.append(int2String(branch->getNo()));
    MachineOperand* mo = getLabel(temp);
    MachineInstruction* mi = new BranchMinstructionWithoutCond
    (builder->getBlock(),BranchMInstruction::B,mo);
    builder->getBlock()->InsertInst(mi);
}

void RetInstruction::genMachineCode(AsmBuilder* builder) {
    if(!retVoid()){
        setGlobald(getReg(0));
        setGlobals1(genMachineOperand(getOperands(0)));
        MachineInstruction* mi =new MovMInstruction(builder->getBlock(),MovMInstruction::MOV,
        getGlobald(),getGlobals1());
        builder->getBlock()->InsertInst(mi);
    }
    MachineFunction* mf = builder->getFunction();
    MachineOperand* sp = getReg(13);
    int i = mf->AllocSpace(0);
    MachineOperand* size =getImmInt(i);
    MachineInstruction* cur_inst = new BinaryMInstruction(builder->getBlock(),
    BinaryMInstruction::ADD,sp, sp, size);
    builder->getBlock()->InsertInst(cur_inst);
    MachineOperand* mo2 = getReg(14);
    MachineInstruction* mi = new BranchMinstructionWithoutCond(builder->getBlock(),
    BranchMInstruction::BX, mo2);
    builder->getBlock()->InsertInst(mi);
}

```

接下来是函数定义，在目标代码中，在函数开头需要进行一些准备工作。首先需要生成 PUSH 指令保存 FP 寄存器及一些 CalleeSaved 寄存器，之后生成 MOV 指令令 FP 寄存器指向新的栈底，之后需要生成 SUB 指令为局部变量分配栈内空间。分配栈空间时需要注意，一定要在完成寄存器分配后再确定实际的函数栈空间，因为有可能会有某些虚拟寄存器被溢出到栈中。一种思路是不在目标代码生成时插入 SUB 指令，而是在后续调用 output() 函数打印目标代码时直接将该条指令打印出来，因为在打印时已经可以获取到实际的栈内空间大小；另一种思路是先记录操作数还没有确定的指令，在指令的操作数确定后进行设置 1 。至此，就可以继续转换函数中对应的其他指令了。

```

void CallInstruction::genMachineCode(AsmBuilder* builder) {
    for(unsigned long int i=1;i<operands.size()&&i<=4;i++){
        MachineOperand*mo=getReg(i-1);
```

```
5           MachineOperand*mo2=genMachineOperand(getOperand(i));
6           if(mo2->isImm()&&mo2->getVal()>=256){
7               MachineInstruction* mi =new LoadMInstruction(builder->
8               getBlock(),mo,mo2);
9               builder->getBlock()->InsertInst(mi);
10              continue;
11          }else{
12              MachineInstruction* mi = new MovMInstruction(builder->getBlock(),
13              MovMInstruction::MOV,mo,mo2);
14              builder->getBlock()->InsertInst(mi);
15          }
16      }
17      for(unsigned long int i=operands.size()-1;i>=5;i--){
18          MachineOperand* mo = genMachineOperand(getOperand(i));
19          if(mo->isImm()){
20              MachineOperand* mo2 = getVreg(SymbolTable::getLabel());
21              if(mo->getVal()<=255){
22                  MachineInstruction* mi = new MovMInstruction(builder->getBlock(),
23                  MovMInstruction::MOV,mo2,mo);
24                  builder->getBlock()->InsertInst(mi);
25              }else{
26                  MachineInstruction* mi = new LoadMInstruction(builder->getBlock(),mo2,mo);
27                  builder->getBlock()->InsertInst(mi);
28              }
29              mo=mo2;
30          }
31          MachineInstruction* mi =new BigParaStackMInstrcution(builder->getBlock(),
32          StackMInstrcuton::PUSH,this->vmo,mo);
33          builder->getBlock()->InsertInst(mi);
34      }
35      MachineOperand*mo=new MachineOperand(func->toStr().c_str());
36      MachineInstruction*mi=new BranchMinstructionWithoutCond(builder->getBlock(),
37      BranchMInstruction::BL,mo);
38      builder->getBlock()->InsertInst(mi);
39      if(operands.size()>5){
40          MachineOperand* mo2 = getImmInt(calOffset());
41          MachineOperand* mo3 = getReg(13);
42          mi = new BinaryMInstruction(builder->getBlock(),
43          BinaryMInstruction::ADD,mo3,mo3,mo2);
44          builder->getBlock()->InsertInst(mi);
45      }
46      if(dst!=nullptr){
```

```
47          mo = genMachineOperand(dst);
48          MachineOperand* mo2 = getReg(0);
49          mi = new MovMInstruction(builder->getBlock(),
50          MovMInstruction::MOV,mo,mo2);
51          builder->getBlock()->InsertInst(mi);
52      }
53  }
```

6.2.2 打印 ARM 指令

在本部分，需要完成根据之前分析得出的机器指令类来打印 ARM 指令的环节，可以通过查找 ARM 手册来进行翻译的包括关于 LOad、store、stack 等内容，这些类的翻译可以通过查看官方文档来完成。比较复杂的是对 basicblock、Function 以及 unit 类的翻译，我们需要对其中存储的机器指令类进行正确遍历并在适当的时候做出特殊的处理才能保证程序正常运行

下面的代码展示了对函数类的翻译过程，其中需要注意的是，我们添加了 LTORG 指令，LTORG 用于声明一个数据缓冲池，（也称为文字池）的开始。在使用伪指令 LDR 时，常常需要在适当的地方加入 LTORG 声明数据缓冲池，LDR 加载的数据暂时被编译器放于数据缓冲池中。并且对于栈空间过大的情况进行了处理，需要不断的通过 Load 将需要的内容加载入内存。

```
1
2  void MachineFunction::output()
3   {
4       std::string s = this->sym_ptr->toStr();
5       s = s.substr(1,s.size());
6       const char* func_name = s.c_str();
7       fprintf(yyout, "\t.global %s\n", func_name);
8       fprintf(yyout, "\t.type %s , %%function\n", func_name);
9       fprintf(yyout, "%s:\n", func_name);
10      (new BigParaStackMInstrcution(nullptr, BigParaStackMInstrcution::PUSH,
11      getSavedRegs(), getReg(11),getReg(14)))->output();
12      (new MovMInstruction(nullptr, MovMInstruction::MOV, getReg(11),
13      getReg(13)))->output();
14      int off = myAllocSpace(13,0,14);
15      MachineOperand* size = getImm(off);
16      if(abs(off)>=256){
17          MachineOperand* mo = getReg(4);
18          MachineInstruction* mi =new LoadMInstruction(nullptr,mo,size);
19          mi->output();
20          mi = new BinaryMInstruction(nullptr,BinaryMInstruction::SUB,
21          getReg(13),getReg(13),mo);
22          mi->output();
23      }else{
```

```
24          MachineInstruction* mi = new BinaryMInstruction(nullptr,BinaryMInstruction::SUB,getReg
25          getReg(13),size);
26          mi->output();
27      }
28      for(unsigned long int i= 0;i<block_list.size();i++){
29          block_list[i]->output();
30          this->allSize+=block_list[i]->getInstSize();
31          if(allSize > 200){
32              fprintf(yyout, "\tb .F%d\n", parent->getUnitLabel());
33              fprintf(yyout, ".LTORG\n");
34              parent->printGlobal();
35              fprintf(yyout, ".F%d:\n", parent->getUnitLabel()-1);
36              allSize = 0;
37          }
38      }
39      fprintf(yyout, "\n");
40  }
41
```

除却对高层抽象代码的指令翻译外，对全局变量、函数声明的打印也是至关重要要的，我们需要的就是从 unit 中读取需要进行全局声明的信息，从 vector 中拿到对应的信息并对其进行分类，有初始化的变量，没有初始化的变量，以及函数的声明或定义，通过分为不同的可能进行保存并对分类结果进行打印。

```
1
2  void MachineUnit::PrintGlobalDecl()
3   {
4       std::vector<int>choose;
5       if (!global_list.empty())
6           fprintf(yyout, "\t.data\n");
7       for (unsigned long long int i = 0; i < global_list.size(); i++) {
8           IdentifierSymbolEntry* se = (IdentifierSymbolEntry*)global_list[i];
9           if (se->getConst()) {
10              choose.push_back(1);
11              //constIdx.push_back(i);
12          } else if (se->isAllZero()) {
13              choose.push_back(2);
14              //zeroIdx.push_back(i);
15          } else {
16              choose.push_back(3);
17          }
18      }
```

```
19      for(unsigned long long int i=0;i<choose.size();i++){
20          if(choose[i]==3){
21              IdentifierSymbolEntry* se = (IdentifierSymbolEntry*)global_list[i];
22              fprintf(yyout, "\t.global %s\n", se->toStr().c_str());
23              fprintf(yyout, "\t.align 4\n");
24              fprintf(yyout, "\t.size %s, %d\n", se->toStr().c_str(),
25                      se->getType()->getSize());
26              fprintf(yyout, "%s:\n", se->toStr().c_str());
27              if (!se->getType()->isArray()) {
28                  fprintf(yyout, "\t.word %d\n", se->getValue());
29              }
30              //数组
31              if(se->getType()->isArray()){
32                  int n = se->getType()->getSize() / 32;
33                  int* p = se->getArrayValue();
34                  for (int i = 0; i < n; i++) {
35                      fprintf(yyout, "\t.word %d\n", p[i]);
36                  }
37              }
38          }
39      }
40      fprintf(yyout, "\t.section .rodata\n");
41      for(unsigned long long int i=0;i<choose.size();i++){
42          if(choose[i]==1){
43              IdentifierSymbolEntry* se = (IdentifierSymbolEntry*)global_list[i];
44              fprintf(yyout, "\t.global %s\n", se->toStr().c_str());
45              fprintf(yyout, "\t.align 4\n");
46              fprintf(yyout, "\t.size %s, %d\n", se->toStr().c_str(),
47                      se->getType()->getSize());
48              fprintf(yyout, "%s:\n", se->toStr().c_str());
49              if (!se->getType()->isArray()) {
50                  fprintf(yyout, "\t.word %d\n", se->getValue());
51              } else {
52                  int n = se->getType()->getSize() / 32;
53                  int* p = se->getArrayValue();
54                  for (int i = 0; i < n; i++) {
55                      fprintf(yyout, "\t.word %d\n", p[i]);
56                  }
57              }
58          }
59      }
60      for(unsigned long long int i=0;i<choose.size();i++){
```

```
61            if(choose[i]==2){
62                //数组
63                IdentifierSymbolEntry* se = (IdentifierSymbolEntry*)global_list[i];
64                if (se->getType()->isArray()) {
65                    fprintf(yyout, "\t.comm %s, %d, 4\n", se->toStr().c_str(),
66                            se->getType()->getSize());
67                }
68            }
69        }
70    }
71
```

7 总结

通过本次实验，我将理论课上学习的知识应用到了实践中，在已经提供的简单 sysY 编译器框架中根据自己在理论课上学习的知识动手实现一个编译器。例如对类型系统、符号表的实现更加丰富了我对理论知识的理解，让我更加清晰了编译器的细节以及不同阶段的连接承载方式令人受益匪浅。

最终提交的 gitlab 链接：https://gitlab.eduxiji.net/nku-jinmenhu/complier/-/tree/finallab7