编译原理设计文档

赵悦鸣 19375032

词法分析

设计思路

在这次作业中，我将词法分析的所有过程都写到了一个`Lexical`类中。进行词法分析的思路是通过遍历一遍文件中所有的字符串，在遍历的过程中将每个单词识别出来，将识别出来的单词单独用一个`SymItem`类存放，单词的属性包括名字和类别（将所有类别写成了一个`enum`类`Typename`），再将所有的单词都存放到数组当中，最后遍历数组输出所有的单词。

实现流程

首先介绍一下词法分析的**大致流程**：

1. 读入文件
2. 去除注释
3. 逐个分析出单词

其中重点介绍去除注释和逐个分析单词的过程。实现的过程中比设计的过程中多了去除注释的部分。

取得字符

由于我们经常需要获得下一个字符，所以专门写了一个叫做nextch的函数，用于获取下一个字符以及记录当前的行数，记录行数是为了便于后续的错误处理。

```
    private void nextch(){
        if (changeline = true) {
            changeline = false;
        }
        while (lineptr == linelen) {
            if (linenum == linestr.size()-1 || linestr.size() == 0){
                isend = true;
                return;
            }
            linenum++;
            changeline = true;
            lineptr=0;
            linelen=linestr.get(linenum).length();
        }
        tmpchar = linestr.get(linenum).charAt(lineptr);
        lineptr++;
    }
```

去除注释

为了方便后续的词法分析，我首先将注释部分替换成了空格。由于注释有单行注释（`//blabla`）和多行注释（`/*blabla*/`）两种，需要分别去除。

在处理注释的时候需要注意的是`<FormatString>`的影响，因为在`<FormatString>`中也会出现`/`和`*`符号，所以我们在处理注释的时候需要单独设置一个是否在`<FormatString>`中的标记，如果在`<FormatString>`中，不会识别为注释，只有不在`<FormatString>`中时我们才将其识别为注释。以下两种注释的处理的处理都是建立在它们不在格式字符串中的。

1. 单行注释

   在一行中识别到`//`之后直接将这一行后面的所有字符包括`//`全部替换成空格就行。

2. 多行注释

   在识别到`/*`之后，将到`*/`处的字符全部替换成空格。如果在注释中出现了格式字符串，我们不将其识别为格式字串。

总的来说，格式字符串和注释**谁先被识别，后面那个就不会被识别。**

逐个分析单词

在逐个分析单词的时候，我针对识别单个单词书写了`get_tpw`的方法，针对识别所有的单词实现了`gene_words`方法，每当识别一个单词就调用一个`get_tpw`方法。文件末尾返回`EOF`或者`NULL`类型的单词，标志分析过程的结束。

在分析单词的时候，我针对每一个类型都写了一个状态机。单词主要分为四大类，运算符号，格式字符串，数字，还有其他的字符串。

针对运算符号的识别，我将开头相同的字符串合并识别，比如`<`和`<=`，`=`和`==`等，一些识别开头就能识别类型的运算符号如`(`和`*`等直接进行识别。

针对格式字符串的识别，首先识别`"`，然后将结尾的`"`之前的字符全部加入到格式字符串。

针对数字的识别，从第一个数字识别到第一个非数字即可。

针对其它字符串的识别，它们或是`IDENFR`或是一些系统的关键字，我们将系统的关键字事先写好，在识别完字符串之后对这些关键字进行比较，如果能比较成功则是关键字，不能比较成功则是`IDENFR`

方法简介

下面展示`Lexical`和`SymItem`包含的方法：

`Lexical`类包含的方法如下：

```java
public class Lexical {

    private String fname;
    private File fin;
    private InputStreamReader istream;
    private BufferedReader bfread;
    private int linenum = 0;
    private int lineptr = 0;
    private int linelen = 0;
    private ArrayList<SymItem> items = new ArrayList<>();
    private ArrayList<String> linestr = new ArrayList<>();
    private char tmpchar = '\0';
    private boolean isend = false;
    private boolean changeline = false;

    public Lexical() throws IOException {...}

    public ArrayList<SymItem> getItems() { return this.items; }

    private void rep_anno(){...}

    private void nextch(){...}

    private int get_nowline() {...}

    private SymItem get_tpw(){...}

    public void gene_words() {...}

    public void print_table() throws IOException {...}
}
```

针对每个方法的介绍如下：

1. `getItem()`

   将词法分析的结果传入语法分析

2. `rep_anno()`

   去除注释

3. `nextch()`

   获取下一个字符

4. `get_nowline()`

   获取当前的行号（最后发现这个函数没什么用）

5. `get_tpw()`

   识别单个单词

6. `gene_words()`

   识别所有的单词

7. `print_table()`

   打印词法分析的结果

`SymItem`类包含的方法如下：

```
public class SymItem extends Treenode{
    private Typename type;
    private String name = null;
    private int line = 0;

    public SymItem(Typename type, String name, int line){...}

    public String getName() { return this.name; }

    public Typename getType() { return this.type; }

    public int getLine() { return this.line; }

    public int getformatc() {...}

    @Override
    public String toString() { return type.toString()+" "+name; }
}
```

针对每个函数的作用介绍如下：

1. `getName()`

   获取当前单词的名字

2. `getType()`

   获取当前单词的种类

3. `getLine()`

   获取当前单词的行号

4. `getformatc()`

   在错误处理时用的

语法分析

设计思路

在这次作业中，将语法分析的所有过程都写到了`Parser`类中。语法分析的过程为递归下降分析法，并且将所有的语法成分都建成了一颗语法树，最后打印的时候只需要调用根节点`CompUnit`的`toString()`方法即可。

按照递归下降的写法，我们针对每一个语法成分都写了一个方法对其进行分析，在识别完语法成分之后就将其作为一个节点加入语法树中，建树的方法可以说是深度优先。

树的节点分为两类，一类是终结符节点，这些节点就是词法分析中分析出来的词，这类节点属于`SymItem`类，是叶子节点；另一类是非终结符节点，这类节点属于`MidItem`类，是中间节点，有一个专门存放叶子节点的数组。

```java
public class SymItem extends Treenode{
    private Typename type;
    private String name = null;
    private int line = 0;
```

```java
public class MidItem extends Treenode{
    private String type;
    private ArrayList<Treenode> nodes;
```

实现流程

语法分析的详细流程和教材上介绍的递归下降的流程差不多，读取每一个词并进行分析其是什么语法成分，然后再进去相应的递归程序进行分析。实现的过程中没有对之前的设计作出修改。可以从读取和分析两个部分来说明。

读取

对于语法分析中的单词读取，我采用的方法是直接把词法分析读取完之后的所有单词输入到语法分析的`Parser`类中，然后一个一个读取。

由于在语法分析的过程中有一部分语法的识别需要预处理，所以我实现了`index`变量用于指示现在分析到的单词和作为预处理时的基准，以及`getword()`函数用于获取下一个单词。

分析

分析的过程基本按照教材上递归下降的写法就行，需要注意的只有两个地方。下面分别进行说明：

1. 在`Stmt`语法成分分析的时候需要进行预读。

   `Stmt`的文法如下：

   ```
   语句  Stmt → LVal '=' Exp ';' // 每种类型的语句都要覆盖
     | [Exp] ';'  //有无Exp两种情况
     | Block
     | 'if' '(' Cond ')' Stmt [ 'else' Stmt ] // 1.有else 2.无else
     | 'while' '(' Cond ')' Stmt
     | 'break' ';' | 'continue' ';'
     | 'return' [Exp] ';' // 1.有Exp 2.无Exp
     | LVal '=' 'getint''('')'';'
     | 'printf''('FormatString{','Exp}')'';' // 1.有Exp 2.无Exp
   ```

   对于以下文法：

   ```
   Stmt -> Lval '=' Exp ';'
         | Lval '=' 'getint' '(' ')' ';'
         | [Exp] ';'
   ```

   当我们在分析`Stmt`的时候识别出`Lval`是，它有可能是以下三个成分之一，由于前两个成分是可以通过`=`后面的语法成分来进行分析的，所以我们需要先预读这一行判断是否存在`=`，如果存在等号，那么我们就识别他们为前两个文法，否则识别为第三个文法。

2. 左递归的语法分析需要改写文法。

   以`AddExp`为例：

   `AddExp`的原始文法如下：

```
AddExp -> MulExp | AddExp ('+'|'-') MulExp
```

改写后的文法如下：

```
AddExp -> MulExp {('+'|'-') MulExp}
```

但是需要注意的是改写文法之后需要对树的输出进行相应的修改。

错误处理

设计思路

在错误处理的过程中，我们需要新增的功能其实包括符号表管理和错误处理两个部分。为此我们实现了符号表管理的`IdentTable`类和错误处理的`Error`类，以及符号表的符号类`IdentItem`，它们的代码结构如下：

`IdentTable`类：

```
import java.util.ArrayList;
import java.util.HashMap;

public class IdentTable {
    private ArrayList<IdentItem> tableitems = new ArrayList<>();
    private ArrayList<IdentItem> serachitems = new ArrayList<>();
    private ArrayList<FuncItem> funcs = new ArrayList<>();
    private int index;

    public IdentTable() { this.index = 0; }

    public boolean isconst(String name,int level){...}

    public ArrayList<Integer> funcrtype(String name) {...}

    public void additem(String name,int clas,int type,int level) {...}

    public void poplevel(int level) {...}

    public int findfunc(String name) {...}

    public int findvartype(String name) {...}

    public int findfunctype(String name) {...}

    public int findvar(String name,int level) {...}

    public int mulvar(String name,int level) {...}
```

主要实现了对函数和变量名字的插入，查找和清除。在符号表管理中最重要的就是插入新的符号，在每一个变量定义的时候都需要插入到符号表；对于查找的功能，主要是在错误处理中的一些判断；对于清除的功能，就在退出一个`Block`的时候清除局部变量的过程。

我在`IdentTable`中也设置了三个表，第一个是编译时的表，清除局部变量就是清除这个表的变量，第二个时所有的变量的搜索表，没有清除，是为了方便后续匹配函数参数类型的过程，第三个是函数表，函数有函数名称和其所在搜索表的位置两个属性。**（由于现在实现的符号表中对于数组类型没有标明长度，之后还需要进行修改）**

`IdentItem`类：

```java
public class IdentItem {
    private String name;
    private int clas;//1:int 2:void 3:1d-array 4:2d-array
    private int type;//1:const 2:var 3:param 4:func
    private int level;
    private int addr;

    public IdentItem(String name,int clas,int type,int level,int addr) {
        this.name = name;
        this.clas = clas;
        this.type = type;
        this.level = level;
        this.addr = addr;
    }

    public String getName() { return this.name; }

    public int getClas() { return this.clas; }

    public int getType() { return this.type; }

    public int getLevel() { return this.level; }

    public int getAddr() { return this.addr; }

}
```

存储了符号的名字，类型和数据类型以及所在代码段的层数，用于后续查找时的判断。

`Error`类：

```java
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.TreeMap;

public class Error {
    private static boolean haserror = false;
    private static HashMap<String,String> errormap = new HashMap<>();
    private static TreeMap<Integer,String> errors = new TreeMap<>();

    public Error() { init(); }

    private void init() {...}

    public boolean gethaserror() { return Error.haserror; }

    public static void adderror(String etype,int linenum) {...}

    public static void print_error() throws IOException {...}

}
```

实现了错误类型的加入和查找。由于加入错误的方法是一个静态方法，所有的类都可以调用这个方法，便于在词法分析和文法分析的过程中加入错误。

在初步设计完符号表管理和错误类型的管理之后，我们针对不同的错误类型进行不同的判断和处理，具体的错误处理流程在后面的部分。

实现流程

将错误分成了三种，第一种是词法分析中就能检查出的非法符号，第二种是语法分析中能够检查出的缺少分号，右小括号和右中括号，第三种是语义相关的错误，前两种错误在词法分析和语法分析中处理，第三种单独建立一个`MEJudge`类，遍历语法树来查找。实现的过程中没有对之前的设计作出修改。

`MEJudge`类的定义如下：

```java
public class MEJudge {
    private MidItem compunit;
    private int cirdepth;
    private int level;
    private String functypenow = "";
    private String rettypenow = "";
    private IdentTable table;
    private boolean funcfind = true;

    public MEJudge(MidItem compunit) {...}

    public void print_comp() throws IOException {...}

    public void search_error(Treenode now) {...}

    private void in_func(Treenode type) {...}

    private void in_main_func(Treenode type) {...}

    private void out_func(int line) {...}

    private void ejudge_i(ArrayList<Treenode> elements) {...}

    private void ejudge_m(int line) {...}
```

在这个类中，通过遍历语法树，当识别到可能出错的元素时进入判断出错的方法查找。

1. 非法符号

   非法符号错误的识别在词法分析时进行，在识别`<FormatString>`的过程中，如果我们识别到了非法字符，就直接往`Error`类里面添加一个错误信息就行。

2. 名字重定义

   为了判断一个新的符号是否是重定义，我们只需要在当前层判断是否有符号和这个新的符号叫一样的名字即可，如果没有则可以添加，否则是重定义。属于符号表管理的内容。

3. 未定义的名字

   这里实际包含了两种情况，一种是查找变量名称是否定义过，另一种是查找函数名称是否定义过，我们分别在符号表管理中实现相应的判断方法即可。

4. 函数参数个数不匹配和函数参数类型不匹配

   在判断函数参数个数和类型是否匹配的时候，由于不存在不合法类型相加的错误，我们判断的类型包括`int`，一维数组，二维数组三类，没有判断二维数组的第二维长度是否符合。判断的过程是在`Exp`类中递归进行的。我们将符号表中对应函数的参数取出作为一个数组，再将调用函数过程中的参数取出作为一个数组，对它们进行比较，就能判断这两个错误。

5. 函数与返回值不匹配

   在进入函数的时候，我们根据函数的类型设置以下两个变量：`FuncTypeNow`用于指示当前的函数类型以及`RetTypeNow`用于指示当前返回值的类型。对于无返回值的函数，如果出现了有返回值的返回语句，则报错，若没有出现有返回值的返回语句或者没有返回语句都行；对于有返回值的参数，如果出现了无返回值的返回语句或者没有返回语句则报错，在出现返回语句的时候和函数的末尾对`RetTypeNow`和`FuncTypeNow`进行判断。

6. 改变常量的值

   去符号表中查找赋值的变量是否是常量即可。

7. 缺少分号，右小括号，右中括号

   在语法分析的时候判断即可。

8. `printf`格式字符串与表达式不匹配

   在读到`printf`的时候判断即可。

9. 再非循环块中使用`break`和`continue`语句

   标记当前循环所在层数，如果在第0层使用`break`和`continue`语句则报错。

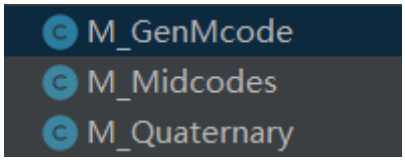
代码生成

设计思路

代码生成整体可以分为两个步骤，第一个是从原始代码生成中间代码，也就是四元式的部分，第二个是从中间代码生成目标代码的部分。其中四元式需要我们自己进行设计和翻译。下面我分为这两个部分讲一下我的设计思路。

生成中间代码

这里我才用的也是递归下降遍历语法树，每一个成分生成它们对应的四元式。由于在我的架构中，我没有将每一个语法成分都单独抽离出来成为一个单独的类，所以我新增了三个类，一个用来进行语法树的遍历和语法成分的分析，一个用来产生和组织四元式的形式，还有一个是四元式。



下面再来讲讲这三个类我都是怎么设计的，首先是用来进行语法树的遍历和语法成分分析的类，由于之前已经进行了一次语法分析，这里的遍历和语法分析差不多，主要是加上了四元式。然后是产生和组织四元式的类，首先是我自己的四元式的设计，我的四元式分为操作名称和三个操作数，再最开始的设计中，我大概分类别设计了如下的操作和其对应的操作数，其中由于所有的常量都需要事先计算出来，所以我对他们写了专门的方法，展示如下（设计时的修改在实现流程中介绍）：

```
61      //const int
62      //const int x = a
63      public void AddConst(String name,int value) {...}
76
77      //const int[]
78      //constarr1 int x[n]
79      //x[0] = a1....x[n-1]=an
80  @   public void AddConst1d(String name,ArrayList<Integer> value1d,int len1) {...}
100
101     //const int[][]
102     //constarr2 int x[m][n]
103     //x[0][1]=a11...x[m-1][n-1]=amn
104 @   public void AddConst2d(String name,ArrayList<ArrayList<Integer>> value2d,
105                            int dim2len,int dim1len) {...}
141
142     //int
143     //int a
144     public void Addint(String name) {...}
154
155     //int[]
156     public void Addint1d(String name,int len1) {...}
163
164     //int[][]
165     public void Addint2d(String name,int dim2len,int dim1len) {...}
172
173     public void Addlocalval(String name,String num) {...}
190
191 @   public void Addlocalval1d(String name,ArrayList<String> exp1d,int len1) {...}
210
211 @   public void Addlocalval2d(String name,ArrayList<ArrayList<String>> exp2d,int dim2len,int dim1len) {...}
246
```

```
266     //int[] =
267 @   public void Addintval1d(String name,ArrayList<Integer> exp1d,int len1) {...}
286
287     //int[][] =
288 @   public void Addinival2d(String name,ArrayList<ArrayList<Integer>> exp2d,int dim2len,int dim1len) {...}
323
324     //funcname
325     public void AddFuncName(String name,String type) {...}
334
335     //funcpara0d
336     public void AddFuncPara0d(String name) {...}
345
346     //funcpara1d
347     public void AddFuncPara1d(String name) {...}
356
357     //funcpara2d
358     public void AddFuncPara2d(String name,int len) {...}
368
369     //printf
370 @   public void AddPrintf(String fstr,ArrayList<String> exps) {...}
402
403     //return val
404     public void AddRetVal(String final_exp) {...}
412
413     //x = exp
414     public void AddAssign0d(String name,String fexp) {...}
423
424     //x[i] = exp
425     public void AddAssign1d(String name,String fexp,String index1) {...}
```

```
433         //x[i][j] = exp
434         public void AddAssign2d(String name,String fexp,String index1,
435                                 String index2,int len2,int level) {...}
450
451         // x= y +/- z
452 @       public void AddAddExp(String op,String num1,String num2,String num3) {...}
470
471         // x = y */% z
472 @       public void AddMulExp(String op,String num1,String num2,String num3) {...}
496
497         //func call
498         public void AddFuncCall(String name) {...}
504
505         //func call ret
506         public void AddFuncCallRet(String name) {...}
516
517         //f(x,y)
518         public void AddFuncRP(String name) {...}
528
529         //- ! x
530 @       public void AddUnaryop(String op,String name1,String name2) {...}
547
548         //a[]
549         public void AddgetArr1d(String arrname,String index1,String target) {...}
559
560         //a[][]
561         public void AddgetArr2d(String arrname,String index1,String index2,
562                                 String len,String target,int level) {...}
580
```

```
581         //a[] -> a
582         //a[][] -> a
583         public void AddArrAddr1d(String arrname,String target) {...}
592
593         //a[][]  -> a[]
594         public void AddArrAddr2d(String arrname,String index1,
595                                  String len,String target,int level) {...}
609
610
611         //第二次作业新加的
612         //lable:
613         public void AddLable(String lablename){...}
621
622         //beq/bne/ a,b,lable
623 @       public void AddLogic(String calop,String var1,String var2,String lablename){...}
648
649         //jump lable
650         public void AddJump(String lablename) {...}
658
```

可以看出来我的四元式种类分的比较细，对于一维数组，二维数组，单个元素都采用了不同的处理，对常量，全局变量，和局部变量也是采用了不同的处理。因为在我认为如果中间代码就将操作尽可能独立出来，可以让我在生成目标代码的时候工作量尽可能地小。

最后是四元式的类，就是一个非常平平无奇的类。记录了四元式的信息，并且对第三个操作数提供了一个修改方法（后面优化的时候要用）。

```java
public class M_Quaternary {
    private String op;
    private String operant1;
    private String operant2;
    private String operant3;

    public M_Quaternary(String op, String operant1, String operant2, String operant3){...}

    public String getOp() { return this.op; }

    public String getOperant1() { return this.operant1; }

    public String getOperant2() { return this.operant2; }

    public String getOperant3() { return this.operant3; }

    public void setOperant3(String printflable) { this.operant3 = printflable; }


    @Override
    public String toString() {
        String ans = op + " " + operant1 + " " + operant2 + " " + operant3;
        return ans;
    }
}

```

生成目标代码

由于我生成中间代码之后得到的就是一个四元式的数组，所以我生成目标代码的实际上就是遍历了一次我的四元式数组。在初步设计的时候，我想的就是对每一个四元式，我都识别他的操作类型，然后不同的操作类型就翻译成不同的代码就行了，在设计的时候，我完全没有想到其实我对栈指针不太熟悉，在具体实现的时候其实这一块有非常大的改动（所以说设计时候想的简单那么实现的时候就一定会和设计有非常大的区别）。

实现流程

在这里我主要讲一下实现过程中和之前设计的时候不一样的地方。还是分成生成中间代码和生成目标代码两个部分来说明。

生成中间代码

我碰到的第一个问题就是所有的常量初始化时就必须把具体的值给算出来，那么就意味着在常量初始化的时候，我的表达式会有一个整数类型的返回值。对此我对常量初始化做了一个专门的标记，并且在这个标记下对表达式的计算中，用了一个量将表达式的值存起来，主要的区别就在于取得左值的时候，如果是常量初始化，我会直接将结果赋值为具体的值。

```java
    //LVal -> Ident {'[' Exp ']'}
    public void MidLVal(N_MidItem LVal) {
        ArrayList<N_Treenode> nodes = LVal.getNodes();
        N_SymItem ident = (N_SymItem) nodes.get(0);
        IT_IdentItem var = table.getvar(ident.getName(),level);
        int dim = cal_dim(nodes);
        if ((inconstinit || inglobalinit)) {
            if (dim == 0) {
                this.tmpmulans = var.getConstvalue();
                this.tmpmulexp = tmpmulans+"";
            } else if (dim == 1) {
                pushstore();
                MidExp((N_MidItem) nodes.get(2));
                int len = this.conans0d;
                popstore();
                this.tmpmulans = var.getConstvalue1d(len);
                this.tmpmulexp = tmpmulans+"";
            } else if (dim == 2) {
                pushstore();
                MidExp((N_MidItem) nodes.get(2));
                int len1 = this.conans0d;
                popstore();
                pushstore();
                MidExp((N_MidItem) nodes.get(5));
                int len2 = this.conans0d;
                popstore();
                this.tmpmulans = var.getConstvalue2d(len1,len2);
                this.tmpmulexp = tmpmulans+"";
            }
```

```
} else {
    int vardim = var.getClas();
    if (dim == 0 && vardim == 1) {
        this.tmpmulans = 0;
        this.tmpmulexp = ident.getName();
    } else if (dim == 0 && (vardim == 3 || vardim == 4)) {
        this.tmpmulans = 0;
        this.tmpmulexp = gentmpvar();
        midcodes.Addint(this.tmpmulexp);
        table.additem(this.tmpmulexp, clas: 1, type: 2,level);
        midcodes.AddArrAddr1d(ident.getName(),tmpmulexp);
    }else if (dim == 1 && vardim == 3) {
        pushstore();
        MidExp((N_MidItem) nodes.get(2));
        String index = this.expans0d;
        popstore();
        this.tmpmulans = 0;
        this.tmpmulexp = gentmpvar();
        midcodes.Addint(this.tmpmulexp);
        table.additem(this.tmpmulexp, clas: 1, type: 2,level);
        midcodes.AddgetArr1d(ident.getName(),index,this.tmpmulexp);
    } else if (dim == 1 && vardim == 4) {
        pushstore();
        MidExp((N_MidItem) nodes.get(2));
        String index1 = this.expans0d;
        popstore();
        this.tmpmulans = 0;
        this.tmpmulexp = gentmpvar();
        String len = var.getDim2len() +"";
        midcodes.Addint(tmpmulexp);
        table.additem(this.tmpmulexp, clas: 1, type: 2,level);
```

然后在其他的表达式计算中，我也采用了一个变量将计算出来的值记录下来。例如在`MulExp`中，记录的过程如下：

由于常量需要能够在获取左值的时候参与运算，我也对符号表进行了修改，对于常量，我存储了常量的值，如果是数组的话，我存储了数组的长度和对应的元素（作为一个数组存储）。

```
3   public class IT_IdentItem {
4       private String name;
5       private int clas;//1:int 2:void 3:1d-array 4:2d-array
6       private int type;//1:const 2:var 3:param 4:func
7       private int level;
8       private int addr;
9       private int dim1len = 0;
10      private int dim2len = 0;
11      private int constvalue = 0;
12      private String lablename = "";
13      private int localvaroff = 0;
14      private ArrayList<Integer> constdim1value = new ArrayList<>();
15      private ArrayList<ArrayList<Integer>> constdim2value = new ArrayList<>();
16
```

那么到这里就基本解决了常数的计算问题。第二个之前设计的时候没有考虑到的是短路的问题。每一个子元素都需要一个跳转，在没有优化之前，我的短路设计的十分麻烦，每一个条件表达式的计算元素都有几个跳转标签，整个中间代码十分臃肿。

还有一点需要说明的就是我的表达式中，每两个操作数的运算都对应着一个中间变量，可以说是一个局部的SSA，在之后的寄存器分配过程中较为方便。

生成目标代码

在实现生成目标代码的过程中，和设计有着较大的差别。首先是符号表的部分，在设计的过程中，我以为目标代码生成时的符号表可以直接套用中间代码生成的符号表，但是仔细思考之后我发现如果直接套用中间代码生成的符号表，在查找变量的时候会出现很大的问题，所以我在目标代码生成的时候又重新生成了一个新的符号表，这就意味着我需要在目标代码生成的时候重复一次建表的过程。

在解决了符号表的问题之后，我对目标代码的结构做了初步设计，首先在data段进行常量和全局变量的定义，这样我可以直接使用 `la` 指令获取它们的地址，从而获取其上的数据。此时我又遇到了另一个问题，那就是局部的常量，它们按照我的设计，应该也要放到data段，但是那样的话不可避免地就会出现重名的问题。为了解耦合，我在中间代码生成之后，目标代码生成之前又多加了一个类用来将所有的名字变成特有的名字。包含的方法如下：

```
33      private void Gen_Newquas() {...}
82
83  @   private boolean isnum(String str) {...}
98
99  @   private String gennewname(String name) {...}
104
105     private void AddQuater(String op,String op1,String op2,String op3) {...}
109
110 @   private void Addconst(String op,String op1,String op2,String op3) {...}
124
125 @   private void Addvar(String op,String op1,String op2,String op3) {...}
137
138 @   private void Addparam(String op,String op1,String op2,String op3) {...}
150
151 @   private void Addfunc(String op,String op1,String op2,String op3) {...}
161
162     private void Addassign(String op,String op1,String op2,String op3) {...}
171
172     private void Addassignarr(String op,String op1,String op2,String op3) {...}
184
185 @   private void Addprintf(String op,String op1,String op2,String op3) {...}
193
194     private void Addop1(String op,String op1,String op2,String op3) {...}
200
201     private void Addop1_2_3(String op,String op1,String op2,String op3) {...}
213
214     private void Addop1_3(String op,String op1,String op2,String op3) {...}
223
224     private void Addop1_2(String op,String op1,String op2,String op3) {...}
```

为此我又写了一次建表的过程（个人感觉我的架构不是那么好，因为建表弄了好多次，虽然说是完全解耦合了）。

在解决了变量名字的问题之后，我将所有的常量和全局变量都放到了data段，包括 `printf` 的字符串。然后再跳转到 `main` 函数之前将所有的常量和可以初始化的全局变量进行了初始化。

```
//处理常量和全局变量定义
for (int i=0;i<codes.size();i++) {
    M_Quaternary tmp = codes.get(i);
    if (tmp.getOp().equals("BVarDecl")) {
        gen_Glolable_int( i: i+1);
    } else if (tmp.getOp().equals("CONST")) {
        gen_Glolable_const(tmp);
    } else if (tmp.getOp().equals("CONSTARR1")
            || tmp.getOp().equals("CONSTARR2")) {
        gen_Glolable_constarr(tmp);
    } else if (tmp.getOp().equals("BBLOCK")) {
        this.level++;
    } else if (tmp.getOp().equals("EBLOCK")) {
        this.level--;
    } else if (tmp.getOp().equals("FUNC")) {
        this.nowfuncname = tmp.getOperant1();
    }
}

//处理字符串
for (M_Quaternary qua:codes) {
    if (qua.getOp().equals("PRINTF")) {
        AddString(qua);
    }
}
```

```
//将常量的初始化放到.data段中
targetcode.add(".text");
for (int i=0;i<codes.size();i++) {
    M_Quaternary tmp = codes.get(i);
    if (tmp.getOp().equals("BConstDecl")) {
        gen_ConstDecl(i);
    } else if (tmp.getOp().equals("BVarDecl")) {
        gen_GolVarDecl(i);
    } else if (tmp.getOp().equals("BBLOCK")) {
        this.level++;
    } else if (tmp.getOp().equals("EBLOCK")) {
        this.level--;
    }
}
```

值得说明的是，为了方便我在目标代码生成的过程中识别各个元素所对应的中间代码，我在一部分元素的中间代码对应的起点和终点之前加上了标记。

对于函数部分的目标代码生成。我在没有优化的时候，所有的变量（包括所有的中间变量，局部变量以及函数的参数）都保存在了栈上，这使得我需要维护`sp`寄存器和`fp`寄存器，我在函数的开头和函数的结尾对这两个寄存器进行了维护。

```
//函数头的目标代码
//设置一些寄存器
private void gen_FuncHead(String funcname) {
    int paracount = table.funcparanum(funcname);
    int fpoff = this.fpra;
    this.localoffset = this.fpra;
    int raoff = fpoff-4;
    String storefp = "sw $fp,"+fpoff+"($sp)";
    String addfp = "add $fp,$sp,"+fpoff;
    String storera = "sw $ra,"+raoff+"($sp)";
    targetcode.add(storefp);
    targetcode.add(addfp);
    targetcode.add(storera);
    //参数设置（call的时候弄）
}

//函数尾部的目标代码
private void gen_FuncTail() {
    targetcode.add("lw $ra, -4($fp)");
    targetcode.add("move $sp, $fp");
    targetcode.add("lw $fp, ($sp)");
    targetcode.add("move $sp, $s7");
    targetcode.add("jr $ra");
}
```

由于所有的变量都要保存在栈上，所以我在符号表中还需要记录一个当前变量在栈上的偏移，我在每个变量定义的时候就为他们分配了偏移。并且每个函数在初始化的时候都会初始化一下最初的偏移。代码如下：

```
//局部变量声明
private void gen_Local_var() {
    table.additem(codes.get(index).getOperant2(), clas: 1, type: 2,level);
    String varname = codes.get(index).getOperant2();
    //System.out.println(varname);
    IT_IdentItem var = table.getvar(varname,level);
    var.setLocalvaroff(this.localoffset);
    this.localoffset += 4;
    targetcode.add("sub $sp, $sp, 4");
}
```

还有一个难点就在于对于获取变量以及修改变量的那些操作，我们需要写加载变量到寄存器以及从寄存器把变量存回去的目标代码。为此我将变量大概分成了几类，对于常量和全局变量，我直接从`data`段加载，对于中间变量和函数的参数，直接从栈上加载，写回的过程就是加载的逆过程。由于没有进行寄存器分配，写入和加载内存的操作非常之多。

最后和之前设计的区别就在于函数调用的时候。我采用的方法是先将实参的值传到对应的形参的地址，然后再进入函数之后，再对形参进行声明和分配。虽然说起来就这么几句，但是当时写的时候调正确还是花了一点时间的。

代码优化

由于我的中间代码和目标代码部分实在是有太多的冗余，所以我的代码优化实际上分成了两个部分，第一个是去除了我的冗余，第二个是实现了一些附加的优化。比较具体的代码优化的过程写到申优文档里面去了，在这里只对所做的优化进行一个简单的说明。

1. 将中间变量分配到$t1-$t9
2. 优化条件表达式的目标代码指令
3. 优化条件表达式的跳转目标

   在完成上面一个优化之后，我发现我的中间代码存在一些连续的跳转，类似下面：

   ```
   j lable1
   lable1:
   j lable2
   lable2:
   ```

   我将这样的优化直接写成了一个跳转，采用的方式也是遍历一次中间代码。
4. 删除没用的中间变量
5. 对加载运算数和存储运算结果进行优化

   在没有优化之前，我发现即使中间变量现在已经保存在了寄存器，由于我将做运算的变量固定在了几个寄存器中，我也会使用一个`move`指令将中间变量移到寄存器中，于是去掉了这些`move`指令。
6. 除法优化

   将除以常数的除法全部用乘法替换。
7. 全局变量的寄存器分配

   简单的分析了一下每个变量的定义和使用周期