编译课设申优文档

19231056 康矗

前言

按照教材的方式，典型的编译器设计共分为七个阶段：词法分析、语法分析、错误处理、语义分析、中间代码生成、目标代码生成以及代码优化。其中代码优化是整个编译器设计中最有挑战性的部分。

本文将按照编译器的设计流程依次介绍每一部分的设计方案，并着重介绍**代码优化**方案。

一、词法分析

关于词法分析程序的实现，我采用的是**有限自动机DFA**的方式。该状态机的状态和转移规则如下。

- **所有状态**

  1. INITIAL : 初始状态

  2. IDENFR : 标识符状态，当前读入字符串为**标识符或关键字**

  3. INTCON: 整数状态，当前读入字符串为**整数（INT）类型**

  4. STRCON: 格式字符串状态，当前读入字符串为**格式字符串**

  5. NOTE_ONE : 单行注释状态，当前读入字符串为**单行注释（即'//'所在行）**

  6. NOTE_MULT: 多行注释状态，当前读入字符串为**多行注释（即'/ *... */' 所包含内容）**

  7. END：结束状态，当前读入指针移动到文件末尾

- **状态转移规则**

  1. INITIAL--> INITIAL

  > 这种转移有三种情况
  >
  > ① 读入字符为空格、换行转义符等
  >
  > ② 读入字符为单字符分界符
  >
  > ③ 读入字符串为双字符分界符

  2. INITIAL --> IDENFR

  > 当前读入字符为字母或下划线

  3. INITIAL --> INTCON

  > 这种转移有两种情况
  >
  > ① 当前读入字符为非零数字
  >
  > ② 当前读入字符为0，且后一个字符为非数字

  4. INITIAL --> STRCON

  > 当前读入字符为双引号 "

  5. INITIAL --> NOTE_ONE

  > 当前读入字符为 '/' ，且后一个字符为 '/'

  6. INITIAL --> NOTE_MULT

> 当前读入字符为 '/'，且后一个字符为 '*'

7. INITIAL --> END

> 当前读入字符指针移动到文件末尾

8. IDENFR --> INITIAL

> 这种转移有两种
>
> ① 当前读入字符为空格，转义符
>
> ② 当前读入字符为单字符或双字符分界符

9. IDENFR --> IDENFR

> 当前读入字符为**字母、下划线或数字**

10. INTCON --> INITIAL

> 这种转移有两种
>
> ① 当前读入字符为空格，转义符
>
> ② 当前读入字符为单字符或双字符分界符

11. INTCON --> INTCON

> 当前读入字符为数字

12. STRCON --> INITIAL

> 当前读入字符为双引号 ‘ " ’

13. STRCON --> STRCON

> 当前读入字符为非双引号的合法字符

14. NOTE_ONE --> INITIAL

> 当前读入字符为**换行符（即 '\n'）**

15. NOTE_ONE --> NOTE_ONE

> 当前读入字符为非换行符的合法字符

16. NOTE_MULT --> INITIAL

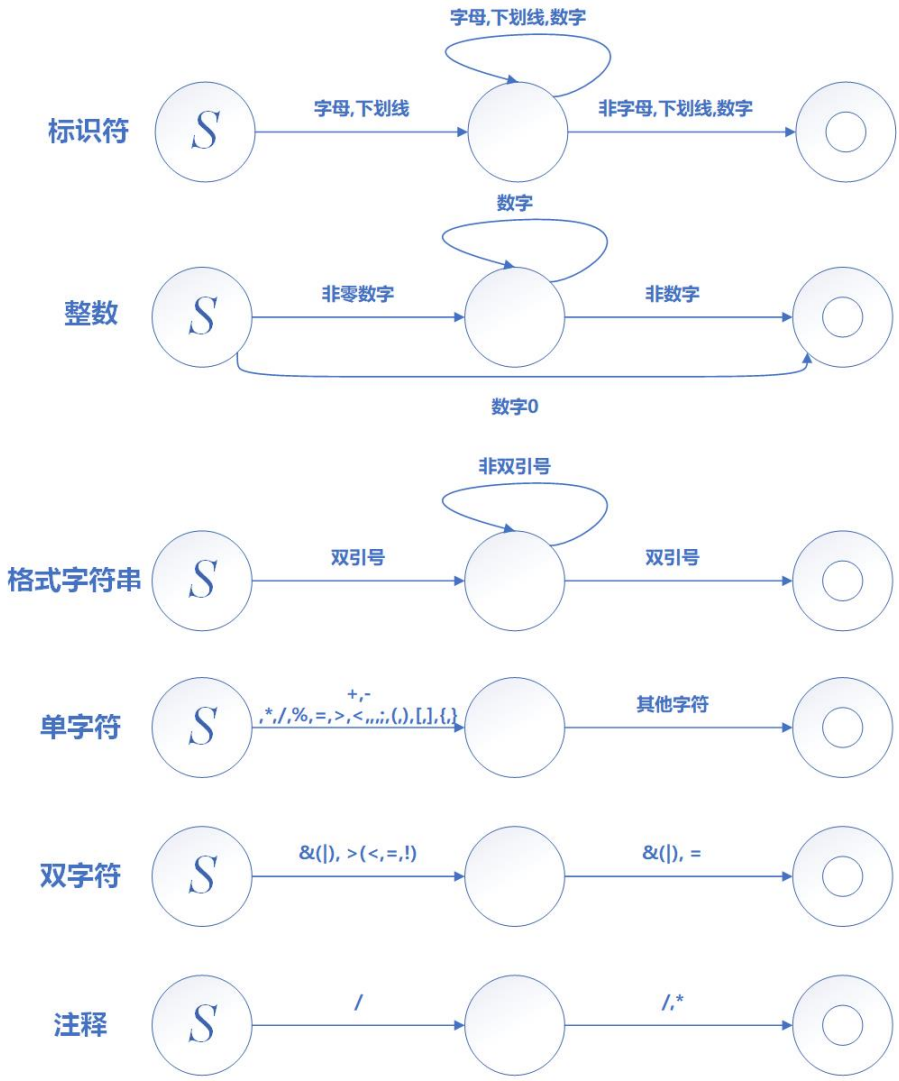
> 当前读入字符为 '*'，且后一个字符为 '/'

17. NOTE_MULT --> NOTE_MULT

> 这种转移有两种情况
>
> ① 当前读入字符为非 '*' 字符
>
> ② 当前读入字符为 '*' , 但后一个字符为非 '/' 字符

补充说明：对于单字符分界符（即**'+、-、*、/、%、>、<、=、!、 ,、 ;、(、)、[、]、{、}'**）和双字符分界符（即**'&&、||、<=、>=、==、!='**），读入后直接解析处理生成类别码，不进行状态转移；而对于标识符和关键字，可以认为关键字是标识符的一种特殊情况而把它们归为一个状态处理，由于标识符不能使用关键字，可以穷举关键字进行解析生成类别码。

其状态转移图如下图所示。



二、语法分析

关于语法分析，直接使用**递归下降**程序按照所给文法进行处理即可。但由于所给的文法有些具有**回溯和左递归**的情况，对此分别采用**提前看和改文法（右递归）**进行处理，这里只讨论对于回溯和左递归情况的处理方案。

1. 回溯

- **CompUnit**

  > CompUnit → {Decl} {FuncDef} MainFuncDef

  由于Decl、FuncDef和MainFuncDef的FIRST集合都有交集，一个最极端的情况为

  > Decl → 'int' Ident { '[' Exp ']' } ['=' ConstInit] ';'
  >
  > FuncDef → 'int' Ident '(' [FuncFParams] ')' Block
  >
  > MainFuncDef → 'int' 'main' '(' ')' Block

  在这个极端情况下，Decl和FuncDef需要解析到 `Ident` 的后一个字符，判断是否为 `(` 才能解析；和FuncDef和MainFuncDef也需要解析到 `Ident` 的后一个字符，判断是否为 `main` 才能解析。

- **Stmt**

  > 语句 Stmt → LVal '=' Exp ';' // AssignStmt

```
| [Exp] ';'                                  // ExpStmt
| Block                                      // BlockStmt
| 'if' '(' Cond ')' Stmt [ 'else' Stmt ]     // IfStmt
| 'while' '(' Cond ')' Stmt                  // WhileStmt
| 'break' ';' | 'continue' ';'               // BreakStmt | ContinueStmt
| 'return' [Exp] ';'                        // ReturnStmt
| Lval '=' 'getint' '(' ')' ';'             // AssignStmt
| 'printf' '(' FormatString { ',' Exp} ')' ';'   // PrintfStmt
```

Stmt语句有很多个分支，可以看到IfStmt、WhileStmt、BreakStmt、ContinueStmt、ReturnStmt、PrintfStmt和BlockStmt的`FIRST集`彼此之间无交集，可以直接判断。而ExpStmt和AssignStmt的最为极端情况是`FIRST`交集为LVal，需要解析完LVal再看下一字符是否为`=`才能判断。

- **解决方案：**

  设置一个`record`和一个`back`的方法，用于记录指针位置和返回。当文法的分支`FIRST集`有交集导致回溯时，使用`record`方法可以保留偷看前指针和其他记录的信息，当偷看到可以判断走哪条分支时，使用`back`方法可以回到偷看前的位置开始继续解析。

```
ptr_record          // 指针缓存，
gram_record;

//  保留指针和其他信息
public void record() {
    ptr_record = ptr;
    gram_record = gramList.size();
}

//  指针倒退，用于向前偷看
public void back() {
    ptr = ptr_record;
    curSym = tokenList.get(ptr - 1).getSym();
    // gramList解析记录清除
    if (gramList.size() > gram_record) {
        gramList.subList(gram_record, gramList.size()).clear();
    }
}
```

2. 左递归

```
逻辑或表达式 LOrExp → LAndExp | LorExp '||' LAndExp

逻辑与表达式 LAndExp → EqExp | LAndExp '&&' EqExp

相等性表达式 EqExp → RelExp | EqExp ( '==' | '!=' ) RelExp

关系表达式 RelExp → AddExp | RelExp ( '<' | '>' | '<=' | '>=' ) AddExp

加减表达式 AddExp → MulExp | AddExp ( '+' | '-' ) MulExp

乘除模表达式 MulExp → UnaryExp | MulExp ( '*' | '/' | '%' ) UnaryExp
```

这六条文法均存在左递归问题，不能使用递归下降直接解析。为此改写文法为右递归，如下所示。

> 逻辑或表达式 LOrExp → LAndExp | LAndExp '||' LOrExp
>
> 逻辑与表达式 LAndExp → EqExp |EqExp '&&' LAndExp
>
> 相等性表达式 EqExp → RelExp | RelExp （'==' | '!=') EqExp
>
> 关系表达式 RelExp → AddExp | AddExp ('<' | '>' | '<=' | '>=') RelExp
>
> 加减表达式 AddExp → MulExp | MulExp ('+' | '-') AddExp
>
> 乘除模表达式 MulExp → UnaryExp | UnaryExp （'*' | '/' | '%') MulExp

但需要注意的是，这样解析会使得语法分析的输出结果有改变，所以需要调整一下输出。正常情况下应该是解析结束时输出，在左递归文法下，要移动到解析完第一个元素后输出。

```
public void parseLOrExp() throws ParseException {
    parseLAndExp();
    getGram(GRAMTYPE.LOrExp);           // 左递归情况下，解析完第一个直接输出再递归
    if (curSym.equals(SYM.OR)) {
        getSym();
        parseLOrExp();
    }
    //getGram(GRAMTYPE.LOrExp)          // 正常情况下，递归解析结束时输出
}
```

三、错误处理

1. 错误概述

本次错误处理类型共有13种（a ~ m）

- 错误类型 `i、j、k` 是符号缺失，属于文法错误，可以在递归下降过程中直接判断处理
- 错误类型 `a、l` 是只有在 `printf` 语句中才会出现，可以在 `printfStmt` 类中定义一个处理该类错误的方法
- 错误类型 `f、g` 都是关于函数返回值问题，可以在 `FuncDef` 类中设置成员变量 `type` 和 `block` ，根据函数类型进行判断
- 错误类型 `m` 是关于break和continue语句使用是否得当，可以设置一个循环块的栈，解析到一个while使这个栈数量就加1，而while紧跟的语句解析结束时栈数量则减1。解析到break或continue时若栈数量不为0则是在循环块内，若栈数量为0则在非循环块内非法。
- 错误类型 `b、c、d、e、h` 都需要查看符号表，需建立符号表进行处理。

2. 符号表技术

符号表管理共创建了三个类：`VarItem`、`ConstItem` 和 `FuncItem` ，并建立了三个容器Map来存储。其中depth表示当前block块的嵌套层数，而 `constItemMap` 和 `varItemMap` 的键K对应的是当前嵌套深度值。进入一个block块时，depth加1，而出一个block块时，depth减1，同时删去当前深度的符号表。

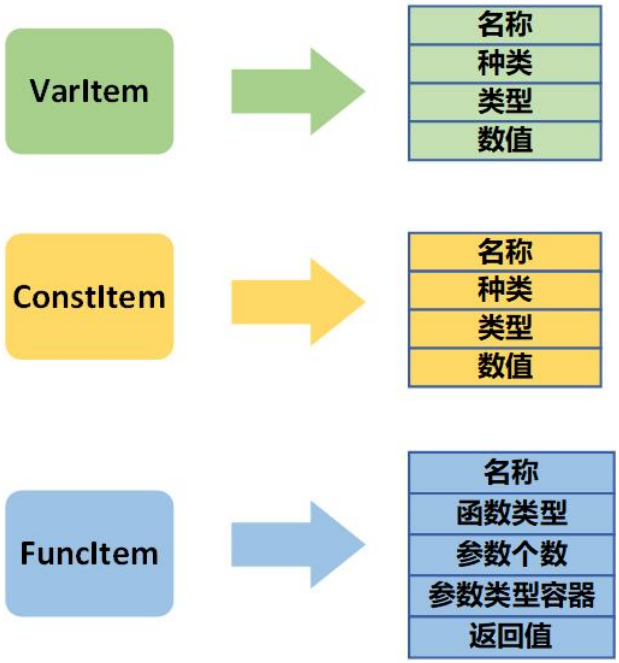
```
private int depth = 0;      //当前block块的嵌套深度

private HashMap<Integer, HashMap<String, ConstItem>> constItemMap;
private HashMap<Integer, HashMap<String, VarItem>> varItemMap;
private HashMap<String, FuncItem> funcItemMap = new HashMap<>();

public void inBlock() {
    depth++;
```

```
    constItemMap.put(depth, new HashMap<>());
    varItemMap.put(depth, new HashMap<>());
}

public void outBlock() {
    constItemMap.remove(depth);
    varItemMap.remove(depth);
    depth--;
}
```

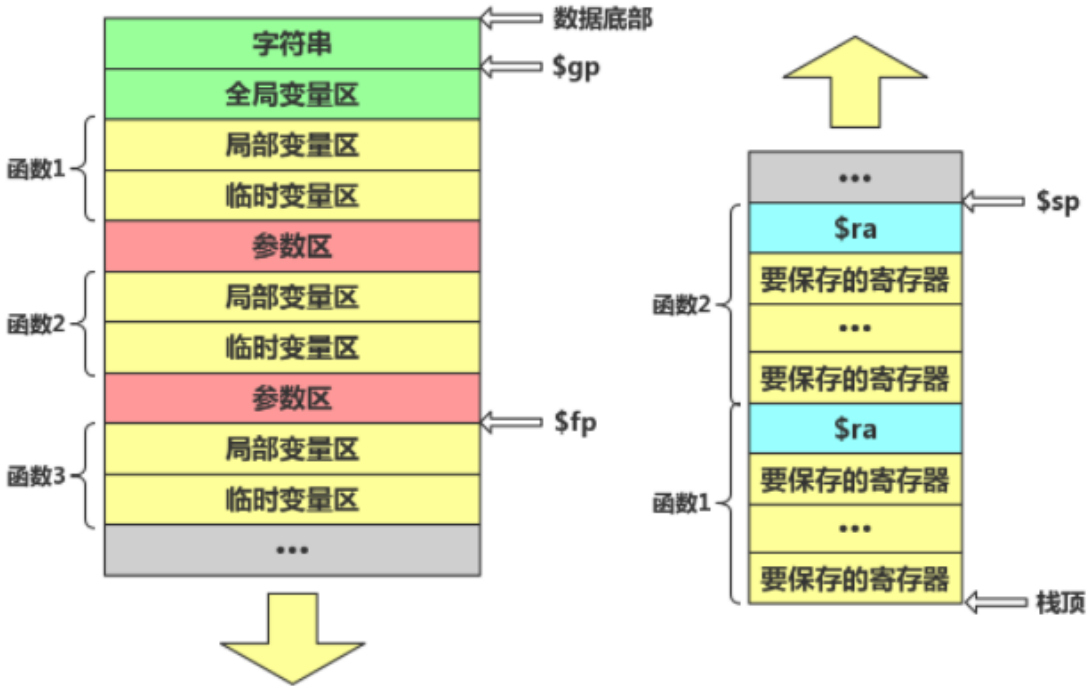


- **名字重定义**：这类错误只需查看当前函数名或变量名在当前作用域下是否有重复值
- **未定义名字**：这类错误要访问内层和外层的所有变量或函数名，查看是否定义过
- **对常量赋值**：这类错误只要在 `AssignStmt` 类中根据左值 `Lval` 的名称查找符号表看**最内层**的类型是否为常量即可
- **函数参数个数不匹配**：在 `FuncItem` 添加一个参数个数的成员变量即可
- **函数参数类型不匹配**：个人认为这个类型的错误处理是难度最大的。对于形参只需在 `FuncItem` 里添加一个**参数维数容器**的成员变量即可；但对于实参，其终结符只有**数字、变量（常量）、函数调用**三种可能。
    - **数字**：认为其维数是0
    - **函数调用**：如果是 `void` 类型，维数是-1，如果是 `int` 类型，维数是0
    - **变量（常量）**：找到对应的 `VarItem` (或 `ConstItem`) 的维数，减去引用时的中括号数量即可得到引用时的维数（如 `int a[2][2]` ，则 `a` 的维数是2，引用 `a[1][1]` 时，使用了两个括号，故引用时的维数为 `2 - 2 = 0` ）

3. 建立语法树AST

在语法分析程序中，我边递归下降边建立AST语法树，使代码不那么臃肿。错误处理时可以在语法树的每个节点递归处理，使得代码更为清晰直观。

四、存储分配方案



- 如图所示，将存储的数据类型归为字符串、全局变量、全局常量、局部变量、局部常量、临时变量和参数，则它们的存储分配分别为：
  - 字符串：存在数据底部，输出时直接读取
  - 全局变量：以$gp为栈底，向上存储
  - 全局常量：若为数组，以$gp为栈底，向上存储；否则不分配存储空间
  - 参数：前4个参数存储在参数寄存器$a0~$a3中，序号大于4的则以$fp为栈底，在内存中增长
  - 局部变量：以参数区的栈顶作为局部变量区的栈底，向上增长
  - 临时变量：以局部变量区的栈顶作为临时变量区的栈底，向上增长；每执行完一个语句后，释放临时变量区；
- 调用函数时，以调用前的临时变量区的栈顶为栈底，在此基础上不断向上增长；并将返回寄存器$ra和帧指针$fp存入栈内存$sp中

五、中间代码设计

(说明：中间代码中临时变量使用 #x, x=0,1,2...来表示)

1. 变量声明
   - @var a // 声明数值变量a
   - @array a [] 12 // 声明数组变量a，数组a的长度为12（若为二维展成一维形式）
2. 常量声明
   - 当常量不是数组型时，没有中间代码
   - @array a [] 12 // 声明数组常量a，数组a的长度为12（若为二维展成一维形式）
3. 函数声明
   - @func comp // 声明函数comp
   - @para a // 函数comp的参数a
4. 函数返回
   - @return #1 // 返回变量#1
   - @return 1 // 返回数值1
   - @return // void型，无返回值
5. 函数调用
   - @push #1 // 传入参数变量#1
   - @push 1 // 传入数值参数1

- @call comp // 调用函数comp
- @get #2 // 得到函数返回值#2

6. 运算表达式

   按运算符分类有：

   - @oper #1 = a ADD b // 加法
   - @oper #2 = a SUB #1 // 减法
   - @oper #3 = #1 MUL a // 乘法
   - @oper #4 = #3 DIV #2 // 除法
   - @oper #5 = a MOD b // 模运算
   - @oper #5 = a LSS b // 小于
   - @oper #1 = a LEQ b // 小于等于
   - @oper #2 = a GRE b // 大于
   - @oper #3 = a GEQ b // 大于等于
   - @oper #5 = #1 EQL 2 // 等于
   - @oper #4 = #1 NEQ 3 // 不等于
   - @oper #5 = #2 NOT 0 // 非运算

7. 赋值表达式

   - @assign #2 = a // 将变量a的值赋给#2
   - @assign a = #2 // 将变量#2的值赋给a
   - @assign #1 = 2 // 已经优化，不会出现这种情况

8. 输入

   @scanf #2 // 将输入数值赋值给#2

9. 输出

   - @printf string 1b2b // 输出字符串
   - @printf int #1 // 输出变量#1的数值

10. 取数组操作

    （1）考虑a为成员变量，且不作为参数传入函数的情况

    - @array_get_value #1 = a [] 2 // 将a[2]的数值赋值给#1
    - @array_get_value #2 = a [] #3 // 将a[#3]的数值赋值给#2

    （2）考虑a为成员变量，但作为参数传入地址**（只在FuncRParams）**中有可能出现

    - @array_get_ptr #1 = a // 将地址a传入函数
    - @array_get_ptr #1 = a [] 1 // 将地址a[1]传入函数
    - @array_get_ptr #2 = a [] #1 // 将地址a[#1]传入函数

    （3）考虑a为函数参数，取值

    - @param_get_value #1 = a [] 2 // 取 a[2] 的值
    - @param_get_value #2 = a [] #1 // 取 a[#1] 的值，**展成一维形式**

    （4）考虑a为函数参数，取地址

    - @param_get_ptr #1 = a // 取参数a的地址
    - @param_get_ptr #2 = a [] 1 // 取参数a，偏移量为1的地址
    - @param_get_ptr #3 = a [] #2 // 取参数a，偏移量为#2的地址

11. 存数组操作

    （1）考虑a为成员变量，即不作为形参的情况

    - @array_set a [] 2 = 3 // 将数值3赋值给a[2]
    - @array_set a [] #1 = 3 // 将数值3赋值给a[#1]
    - @array_set a [] 2 = #1 // 将变量#1赋值给a[2]
    - @array_set a [] #1 = #2 // 将变量#2赋值给a[#1]

（2）考虑a为形参的情况

- @param_set a [] 2 = 1　　// 将数值1赋值给形参a[2]
- @param_set a [] #2 = 1　　// 将数值1赋值给形参a[#2]
- @param_set a [] 2 = #1　　// 将临时变量#1赋值给形参a[2]
- @param_set a [] #1 = #2　　// 将临时变量#2赋值给形参a[#1]

12. 进出基本块
    - @inblock　　// 进入基本块
    - @outblock　　// 出基本块
13. 跳转语句
    - @bz #1 label1　　// 若 #1 == 0 跳转到标签label1
    - @j label2　　// 无条件跳转到label2
14. 标签
    - @label if_1_or_2　　// 插入标签if_1_or_2
    - @save
    - @init
    - @save_init
15. 临时寄存器池清零
    - @free　　// 释放所有临时寄存器
16. 程序退出

    @exit

六、中间代码生成

目标代码的生成只是简单翻译一下中间代码即可，因此目标代码的生成关键在于如何构建并生成中间代码，这里仅就本人觉得比较重要的几个方面来说

1. 变量标识序号

变量名覆盖是符合Sys文法的，因此测试代码中可能会出现多个同名变量在不同的语句块中出现，会给代码生成中变量分配空间造成一些问题，对于代码优化中的数据流分析也十分不友好。因此生成中间代码时给每个变量都赋予唯一的标识名，**便于后续构建冲突图、常数传播等优化**，如下面代码：

```
int i;
int main()
{
    int i;
    {
        int i;
    }
}
```

则生成的中间代码中它们将分别对应为

```
@var i#g
@func main
@var i#1
@var i#2
```

全局变量在变量名后加后缀 `#g`，局部变量在变量名后加后缀 `#i`，其中i表示该变量的序号。

2. 运算表达式

运算表达式按照文法给出的优先级关系依次递归解析即可，每解析一次分配一个临时变量赋值。

```
i = a + b * (c + d)
```

生成中间代码后表示为

```
t1 = c + d
t2 = b * t1
t3 = a + t2
i = t3
```

3. 短路运算

如果条件表达式中含有短路运算符号（&&和||），则按照优先级最小的符号将其拆成多个判断语句处理。

- 在每一个&&且语句后面使用`bz`语句跳转，如果此**且语句不成立，则跳转到下一个（||）或语句**，如果成立继续往下解析
- 在每一个||或语句后面使用`j`语句跳转，如果能顺利地解析下来，则说明所有的&&且语句都满足，**跳转到语句块开始begin处**
- 在语句块开始标签前面添加一个`j`语句跳转，如果程序能解析到此处每发生跳转，说明前面**所有的或语句都不满足，跳转到语句块结束end处**

```
if (a || b && c || d) {
    // do...
}
```

上述代码可以展开成如下语句

```
if (!a) {
    goto IF_OR_1
}

goto IF_BEGIN
IF_OR_1 :

if (!b) {
    goto IF_OR_2
}
if (!c) {
    goto IF_OR_2
}

goto IF_BEGIN
IF_OR_2:

if (!d) {
    goto IF_OR_3
}

goto IF_BEGIN
IF_OR_3:
```

```
goto IF_END
IF_BEGIN:
    // do...
IF_END:
```

七、代码优化方案

1. 基本块划分和流图

大部分优化方案都离不开数据流分析，因此需要先将中间代码划分成基本块并构建流图。

基本块的划分和流图连接遵循的原则如下：

- 基本块只在函数内部进行连接，不能在两个函数之间连接基本块
- 开始时先初始化一个基本块，向下解析，直到结束
  - 碰到`无条件跳转`语句结束，将当前基本块与即将跳转的基本块连接，再新开一个基本块
  - 碰到`条件跳转`语句结束，将当前基本块与即将跳转的基本块连接，再新开一个基本块，**并将当前基本块与新开的基本块进行连接**
  - 碰到`标签`语句结束，新开一个基本块，并将当前基本块与其连接
- 连接基本块时，给每个基本块设置了`prev`和`next`指针数组来记录此基本块的前驱或后继基本块，以便后面的数据流分析

```
public void cutBlocks() {
    if (mediStr[0].equals("@j")) {
        linkCodeBlocks(curBlock, getCodeBlock(mediStr[1]));
        String tempLable = setTempLabel();
        turnNextBlock(tempLable);
    } else if (mediStr[0].equals("@bz")) {
        linkCodeBlocks(curBlock, getCodeBlock(mediStr[2]));
        String tempLabel = setTempLabel();
        linkCodeBlocks(curBlock, getCodeBlock(tempLabel));
        turnNextBlock(tempLabel);
    } else if (mediStr[0].equals("@label")) {
        linkCodeBlocks(curBlock, getCodeBlock(mediStr[1]));
        turnNextBlock(mediStr[1]);
}
```

2. 到达定义分析

到达定义分析针对的元素是**中间代码语句**，因此我专门开了一个类`MediLine`进行操作，具体操作步骤如下：

- 计算每个基本块的`gen`集合，由于`gen`是针对某个变量而言，因此采用了`Map<String, MediLine>`的数据结构来存储，基本块内每个变量名对应一个中间代码语句
- 对于某个基本块，计算每个基本块的`gen`与该基本块的`gen`变量名相同的集合，进行并操作得到该基本块的`kill`集合
- 根据公式 $in[B] = \bigcup_{(B的每个前驱基本块P)} out[P]$ 和 公式 $out[B] = gen[B] \cup (in[B] - kill[B])$ 计算得到每个基本块的`in`和`out`集合
- 检查上一步的out集合是否有改变，若有重复上一步，否则结束循环

进行到达定义分析后，可以做**跨基本块的常量传播和复写传播**

3. 活跃变量分析

活跃变量分析针对的元素是**变量**，计算操作比到达定义数据流要容易一些，具体操作步骤如下：

- 计算每个基本块的`use`和`def`集合
- 根据公式 $out[B] = \bigcup_{(B的每个后继基本块P)} in[P]$ 和公式 $in[B] = use[B] \cup (out[B] - def[B])$ 计算得到每个基本块的`in`和`out`集合
- 检查上一步的in集合是否有改变，若有重复上一步，否则结束循环

进行活跃变量分析后，可以做**死代码删除，寄存器分配**等优化

4. 常量合并

此优化是在生成中间代码之前做的，无需用到数据流分析。

对于运算操作，碰到常数表达式，即运算元素中只涉及常量`const`和数字`number`，编译器可以直接给出结果，尽可能化到最形式。

采用exp、addExp、mulExp、unaryExp、primaryExp、Lval层层递归的方式实现表达式数值的计算，每个类都有对象 `boolean certain` 和 `int value` ，certain表示当前对象的数值是否为确定值，如果是，那么返回value表示此对象的数值。最底层的表达式只有变量、常量、数字和函数调用四种，判断它们数值是否确定的规则如下：

- 若为数字和常量，则数值可确定，直接给出结果
- 若为变量或函数调用，则数值不能确定

```
// 递归判断addExp的数值是否确定
if (addExp == null) {
    isCertain = mulExp.isCertain();
} else {
    isCertain = mulExp.isCertain() && addExp.isCertain;
}
```

优化前：b = 3 + 5 * (5 - 3) * a

优化后：b = 3 + 10 * a

5. 基本块内部的复写传播和常量传播

划分基本块后，就可以大胆地做变量的赋值传播了。对于基本块内部的复写传播，算法步骤如下：

- 初始化符号表，符号表中一个变量对于一个值（可能是常数也可能是变量）
- 顺序扫描，中间代码是如下几种语句时改变符号表中变量对应的值
    - 运算语句：如果操作数的值都为常数，则将变量的值设置为常数的

它的效果展示如下

假如有中间代码

```
@var a
@var b
a = 4
#1 = a * a
b = #1 / a
```

那么由于在第4行对a进行了赋值操作，其后#1的值和b的值都可以直接计算出来，优化后的代码为

```
@var a
@var b
a = 4
b = 4
```

对于复写传播原理也是一样，只是把常数换成变量进行传播

6. 跨基本块的复写传播和常量传播

进行到达定义数据流分析后，可以计算得到每个基本块的in集合。对于某个变量而言，如果此基本块的in集合中只有一条中间代码对此变量进行了赋值操作（若有多条，则前驱有循环或跳转语句，无法判断变量到达时的值），那么将此中间代码作为该基本块的前驱语句，再按照基本块内的复写传播进行操作。算法步骤大致为：

- 计算基本块B的in集合中能用于进行复写传播的语句，临时加入到基本块内
- 进行基本块内的复写传播操作，并删除无用的临时变量和中间代码
- 若此操作后中间代码发生变化，则重复执行，否则结束循环

其效果为可跨越基本块进行复写传播，优化前代码为

```
d = 2;
i = 1;
while (i < 2) {
    i = i + d;
}
b = 2 / d;
```

变量d在循环体内没有进行赋值操作，因此可以跨越基本块进行常数传播

```
d = 2;
i = 1;
while (i < 2) {
    i = i + 2;
}
b = 1;
```

复写传播看似是不节约代价的，但对于除数为变量的除法，此操作可以节约很大的时间开销，而且配合死代码删除后，威力会显得更大

7. 死代码删除

根据活跃变量数据流分析，如果变量x在某个定义点后不再被使用，则可以将它删除。值得注意的是，前面的基本块内部和跨基本块的常量传播优化完成后，再做活跃变量数据流分析可以删除很多无用的中间代码。比如样例testfile1中的代码

```
int main () {
    int i = 2,j = 5;
    i = getint();
    j = getint();
    int k = -+-5;
    int n = 10;
    while (n < k*k*k*k*k*k*k) {
        d = d * d % 10000;
        n = n + 1;
    }
    printf("%d, %d, %d\n", i, j, k);
    return 0;
}
```

做完死代码删除后可以将循环体内部的运算删除，此时循环体内部没有执行语句，故可以直接将循环体删除。但是需注意：

- 被定义变量是全局变量时不能删除
- 中间代码是函数调用或输入语句时不能删除

8. 寄存器分配策略

寄存器分配是整个代码优化中最重要的一个方面。现有的较为成熟的寄存器分配算法有图着色算法和引用计数法。这里我改进了引用计数法，综合使用临时寄存器池，改为一个更高效的分配方案。

（1）三种寄存器使用规范

在mips架构中，仅仅提供了两类寄存器，即s类和t类。s类用于保存局部变量（或全局变量），t类寄存器保存临时变量。经过多次测试后，我认为这种分配方案是不合理的，应当再加一类寄存器用于**保存全局变量**，因此总共有三类寄存器可供使用，其行为规范如下

- 全局寄存器（跨函数）：任何时刻都可以直接调用寄存器内的值，直接对此寄存器进行操作，任何时刻都不需要写回内存
- 局部寄存器（跨基本块）：在一个函数中可以直接调用该寄存器，在此函数内任何操作都不需要写回内存，但是在调用函数时，需保存现场，压到栈里，调用结束后再恢复现场，从栈中取回
- 临时寄存器（不跨基本块）：在一个基本块内可以直接调用该寄存器，但碰到跳转语句等跨基本块行为时，需要把寄存器的值写回内存，将寄存器池清空，没有保存恢复现场的操作。

（2）分配策略

本人认为内存访问最重要的两个方面在于**循环和函数调用**，三种寄存器访问内存的可能为：

- 全局寄存器：没有访问内存操作
- 局部寄存器：循环时不访问内存，**函数调用**时需压入压出栈，两次访问内存操作
- 临时寄存器：函数调用时可以直接清除，如果值发生改变则写入内存，**循环**时需从内存中读取值

根据以上几点，我的分配策略是：如果局部变量是在循环体内部，则分配局部寄存器，**以减小访问内存开销**；否则使用临时寄存器，**以减小函数调用压入压出栈开销。**此过程的局部寄存器数量是不定的，是根据函数内循环体中局部变量的数量来确定的。设分配的局部寄存器数量为`var_reg`，循环体内的局部变量数量为`var_num`，则局部寄存器数量`var_reg`的确定方式如下，确定局部寄存器数量后即可确定临时寄存器数量

```
const int VAR_MAX = 8;
const int REG_NUM = 18;

if (var_reg < VAR_MAX) {
    var_reg = var_num;
} else {
    var_reg = VAR_MAX;
}

temp_reg = REG_NUM - var_reg;
```

分配策略为：

- 全局寄存器（2个）：对于全局变量，引用计数挑选使用最多的全局变量进行分配，剩余的分配给临时寄存器（注意**不能分配给局部寄存器**，因为如果在函数内对全局变量进行赋值后调用另一个函数，那这个改变的值就无法写回内存了）
- 局部寄存器（不大于8个）：对于局部变量，如果是在循环体内出现且引用次数靠前的进行分配，剩余的局部变量全部扔给临时寄存器
- 临时寄存器（不小于10个）：对于临时变量，赋值后使用一次立即释放（所有的临时寄存器都只使用一次），如果是函数调用，则临时变量有可能也要写回内存；对于局部变量和全局变量，使用后可以不立即释放，如果临时寄存器池已满，则采用**最久不用LRU**替换策略进行替换，如果该寄存器的值发生改变，则需写回内存

（注意：关于数组存取的操作，其在内存中的地址只有编译器知道，不能将地址存在一个寄存器里（函数实参除外），数组的读存操作都是必须访问内存一次的，因此局部寄存器分配的局部变量不包括数组元素）

（3）进一步优化

- 临时寄存器进入基本块后需要保存并清除。可以记录每个寄存器的读写操作，若该寄存器的值发生改变，则写回内存，否则可以不用写回内存
- 局部寄存器碰到函数调用后需要保存和恢复现场。根据活跃变量分析，计算该基本块的outs得到后继语句需要使用到的变量，如果后续无需使用该变量，则对于该变量可以不需保存和恢复现场，减少写回内存的开销

9. 循环结构优化

将while语句改写成do-while语句可以减少跳转

```
while (cond) {
    // do...
}
```

该语句会被直观翻译成

```
COND:
if (!cond) {
    goto THEN
}
// do...
goto COND
THEN:
```

对于一个n次循环，这样的语句需要执行n次分支和n次跳转，如果转成do-while语句，可以翻译成

```
if (!cond) {
    goto THEN
}
BEGIN:
// do...
if (cond) {
    goto BEGIN
}
THEN:
```

这样对于一个执行了n次的循环，就只需执行n次分支，原本是 `2n+1` 次跳转，优化后只需 `n+1` 次跳转

10. 乘除优化

（1）乘法优化

优化规则如下

- 若乘数的绝对值为2的幂，可用一条移位指令
- 若乘数的绝对值为2的幂+1，可用一条移位指令和加法指令
- 若乘数的绝对值为2的幂-1，可用一条移位指令和减法指令
- 若乘数为负数，将结果取反即可

比如：

```
a * 4   //  a << 2
a * 5   //  (a << 2) + a
a * -7  //  -((a << 3) - a)
```

（2）除法优化

首先得明确一点，只有当除数是常数的时候才可以做除法优化，如果除数是变量，则无法优化。具体优化算法，本人是参照论文《Division by Invariant Integers using Multiplication》来实现的，优化规则如下：

- 若除数的绝对值是1，直接赋值
- 若除数的绝对值是2的幂，可用一条加法指令和三条移位指令
- 若除数的绝对值不是2的幂，则用两条加法指令、三条移位指令和一条乘法指令
- 若除数为负数，再加一条取反指令

此规则的具体原理在论文中有详细证明，此处不赘述，算法的计算过程如下所示

Inputs: **sword** d and n, with d constant and $d \neq 0$.
udword m;
int ℓ, sh_{post};
$(m, sh_{\text{post}}, \ell) = \text{CHOOSE\_MULTIPLIER}(|d|, N-1)$;
if $|d| = 1$ **then**
  Issue $q = d$;
else if $|d| = 2^{\ell}$ **then**
  Issue $q = \text{SRA}(n + \text{SRL}(\text{SRA}(n, \ell-1), N-\ell), \ell)$;
else if $m < 2^{N-1}$ **then**
  Issue $q = \text{SRA}(\text{MULSH}(m, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
else
  Issue $q = \text{SRA}(n + \text{MULSH}(m - 2^N, n), sh_{\text{post}}) - \text{XSIGN}(n)$;
  Cmt. Caution — $m - 2^N$ is negative.
end if

if $d < 0$ **then**
  Issue $q = -q$;
end if

11. 高效指令选择

这里涉及到指令选择方面的优化，较为细节，本人做的优化有：

- sub指令中如果减数是常数，可以先算出减数的相反数，再使用addi代替
- div指令的三操作数中会多出判断除数是否为0的分支指令，可替换为div+mflo或div+mfhi
- mul指令替换掉mult+mflo
- slti指令替换li+sgt指令（将小于运算调换成大于运算，下同）
- sge指令替换li+sle指令
- sle指令替换li+sge指令

这些指令替换看似只能减少一条other指令，但在实际中往往会有上万次的循环，每次减少一点，其效果也是十分显著的，尤其是在竞速排名靠前时，大家只有分毫之差时，其作用就更为突出了。

12. 窥孔优化

此部分主要实现了两种窥孔优化：

（1）删除冗余中间代码

递归产生四元式时，对于赋值语句，是先求出右表达式的值，再把此值赋值给左值，因此对于如下代码会产生中间代码如下所示：

```
// src_code
a = b + c;

// medi_code
t = b + c;
a = t;
```

可以看到这里的临时变量t的过渡是冗余的，可以在中间代码中进行替换，将过渡的中间代码删除。

对于输入语句，输出语句，返回值语句等，同理也可以进行优化

（2）多余跳转优化

先看如下代码

```
if (cond) {
    //do...
}
```

短路运算递归解析之后会生成如下指令

```
if (!cond) {
    goto IF_OR_1
}
goto IF_BEGIN
IF_OR_1 :
goto IF_END
IF_BEGIN:
//do...
IF_END
```

可以看到在cond条件满足时，还需要做一次分支和一次跳转，针对这类语句，我将其优化成

```
if (cond) {
    goto IF_BEGIN
}
goto IF_END
IF_BEGIN:
//do...
IF_END
```

这样对于每一次循环都可以减少一次跳转，效果堪比循环结构优化

八、遇到的困难和解决方案以及一点感想

- 词法分析阶段：形式语言课中胡老师已经讲完了DFA，所以我设计了一个有穷自动机来解析，没有太大的难点。
- 语法分析阶段：我参照课本上给出的递归下降程序来构造，难点在于所给的文法有些具有**回溯和左递归**的情况，对此分别采用**提前看和改文法（右递归）**进行处理。但是这里给后面代码生成留下了很大的隐患，改完文法后解析运算表达式会出现顺序错误的问题。所以这里我的设计还有待改进。
- 错误处理阶段：经过上学期OO第一单元的血的教训，我在词法分析和语法分析阶段就已经预留了错误处理接口。所以错误处理时，语法错误很轻松的就能解决，剩下的语义错误使用符号表，也不难处理。由于出现的错误类型在文法说明中已十分详尽，所以列举所有可能出现的错误后，**逐个构造测试样例**基本就没啥问题。
- 代码生成阶段：到目标代码生成，难度就有了质的飞跃了。前期三个阶段加起来也只用了不到五天就能写完，但代码生成我整整爆肝了一个星期。个人认为此部分的难点主要有以下几个方面：
    - 复杂表达式解析：这一部分在前面已给出解决方案，且OO第一单元也涉及到了，此处不再赘述
    - 存储分配方案：新开一个变量时要怎么分配空间？调用一个函数后再开一个变量又要如何分配空间？全局变量和局部变量是否该分配在一个区域内？调用函数时保存变量又该保存到哪里？

这些问题虽然在之前的计组和操作系统课程中都有讲过，但都是机械记住几个概念而已。直到自己实现一个编译器需要设计地址空间分配方案时，才真正理解了内存堆、运行栈、和静态数据区分别应该存哪些数据，内存分配又是如何实现动态变化的。

    - 函数实参传地址：这是今年课设新增的一个难点，我在思考处理方法时花费了大量时间。个人认为解决方案只有自己动手写c语言进行测试，观察正确的执行结果，才能知道哪些数据是编译器应该知道的，哪些工作应该是由编译器来完成的。由于大一c语言指针没学好，我在此耗费了很多精力，但还是十分值得的，让我从底层角度理解了地址操作。

- 代码优化阶段：这一阶段的优化部分是在中间代码上实现的。经过反复重构中间代码，让我更深信了**设计优先**的原则：**用于设计的精力一定要大于编码的精力！！！** 由于时间有限，我没能把课本上的优化全都做一遍，比如DAG图个人认为用处不大，就没有实现。这里仅就本人实现的优化谈几点认识：

    - 优化效果是叠加生效的：很多优化方案不是说做了一个优化就一定能达到效果，是需要**多个优化方案结合起来**才能发挥出威力。比如常量传播看似没有很大的功效，但它对于寄存器分配方案，死代码删除，以及除法优化等，都有不可小觑的作用
    - 寄存器分配方案无最优解：本人对比了图着色算法和改进的引用计数法两种方案，发现它们的作用效果差不多，对于某些测试点，改进的引用计数的效果甚至比图着色算法好。原因在于图着色算法可以降低循环上访问内存的开销，但容易增大在函数调用上的开销，当然这里配合函数展开实现后效果会更好，只是本人没时间去完善。所以在最终斟酌之下，我还是采用了自己改进的引用计数方案，~~自己写的不香吗~~。