2020编译设计文档汇总

——18373528 杨凌华

一、词法分析

1、针对本次实验的设计

为了支持更多的数据类型和调用库函数，本次作业采用 **C++** 语言编写。

因为不考虑字符串以及字符常量中出现转义字符的情况，所以可以判定源程序的任意两行之间相互独立，因此

我采用的基本思路是按行读取源码文件，每读取一行就处理一行，不必处理行于行之间存在耦合的情况，其次，每提取到一个词法成分就立即将其输出。

对于每一行源码，维护一个索引变量 ind 来定位当前处理的位置，并且在遍历每一行的时候采用 switch-case 分支结构。

step1：**提取字符串常量。**当检测到 " 双引号字符时，就清空 string 类型的变量 strcon ，然后用 strcon 存储字符串常量，直到遇到下一个 "

，没有考虑字符串内部出现 \" 的情况。

step2：**提取字符常量。**当检测到 ' 单引号字符时，就将紧随其后的字符作为字符常量并输出，并且让 ind++ 以跳过接下来的 ' ，也就是说，默认''中有且仅有一个字符，并不考虑其中出现 \' 的情况。

step3：**提取符号TK。**一堆类似的 case 语句，特判处理所有的 符号 。

step4：**提取整型常量。**此处的数字都是以 string 的形式提取并存储在变量 intcon 中的。

step5：**提取标识符。**标识符以 `_|a|...|z|A|...|Z` 开头，并以 `_|a|...|z|A|...|z|1|...|9` 作为组成。提取出标识符后，需要判断是否是保留字，我事先采用 map 数据结构用来存储了所有 <保留字，类别码>键值对，只需要将提取的标识符转换成小写后查找 map ，根据是否是保留字来对应输出相关的信息。

空白字符的处理：此处我考虑的空白字符只有 '\t' 和 ' ' 两种，专门写了这样一个函数来跳过多余的空白字符：

```
int getNewIndex(int start, std::string line)
{
    int i = start;
    while (line[i] == '\t' || line[i] == ' ') i++;
    return i;
}
```

2、编码后对设计的修改

本次作业的程序仅采用了简单的循环结构和分支结构，没有较为复杂的写法，也没有面向对象，并且代码长度相对较短（<200行），因此可修改的灵活性较高。再者，由于课程进度暂时还没讲到符号表管理和错误处理，因此，还不能准确预测今后具体要做哪些修改，断然预留一些接口显得很牵强，而且容易出错，于是在此只做一些简单的分析预测：

- **符号表管理**：需要根据之后的具体需求来建立合适的数据结构来保存标识符以及其对应的数据类型和值，这个数据需要保证便于添加和查找，可以考虑面向对象建类或者建立结构体，并且需要能够进行高效的查重，以进行错误处理。
- **错误处理**：
    1. 标识符命名是否符合规范
    2. 是否在单词之间有空白符分隔
    3. 是否有对不同数据类型的变量进行相同的命名
    4. ...还有很多细枝末节的小规范，需要根据之后实验的具体要求进行的添加处理，在此无法具体预测。

二、语法分析

1、针对本次实验的设计

本次实验能够继承上次的程序，基本思想就是课上讲的**递归向下分析法**，但是又与课上讲的有所不同，这里不能以字符为单位来进行分析，而应该以一个个单词为单位，这样就能充分利用上次词法分析作业的结果，不过这样一来就不能将词法分析和语法分析同步进行，而是先进行词法分析将每一个单词依次保存在数组（表）中，然后再进行语法分析，所以采用的是串行工作，事实上这也并不会改变复杂度O。

根据递归向下分析法，我为每一个文法成分设计了一个解析程序：

```
void parseProgram();
void parseConstDcrpt();
void parseConstDef();
void parseInteger();
void parseUnsignedInt();
void parseConst();
void parseVarDcrpt();
void parseVarDef();
void parseFuncDef();
void parseParamTable();
void parseDeclareHead();
void parseCompStmt();
void parseStmtList();
void parseStmt();
void parseLoopStmt();
void parseCondition();
void parseExpr();
void parseTerm();
void parseFactor();
void parseFuncCallWithReturn();
void parseFuncCallWithoutReturn();
void parseValueParamTable();
void parseStep();
void parseFuncCall();
void parseCondStmt();
void parseReadStmt();
```

```
27 void parseWriteStmt();
28 void parseSwitchStmt();
29 void parseCondTable();
30 void parseCondChildStmt();
31 void parseDefault();
32 void parseReturnStmt();
33 void parseAssignStmt();
34 void parseMainFunc();
35 void parseString();
36
37 void error();
```

其中，为下一次错误处理作业留下了一些接口，在其间穿插了`void error()`错误处理函数。

因为是递归的结构，所以在每一层解析函数的末尾都会将该`<语法成分>`进行输出，从而达到当某一语法成分分析结束前，另起一行输出当前语法成分的名字的效果。

另外，同时也要穿插输出每一个单词成分，我将单词成分的输出与 getSym() 读取进行绑定，也就是说，在每一次调用 getSym() 的同时，输出一个单词的解析，但这样一来遇到了一个问题，这也是困扰了我最长时间的一个**大问题**：

根据课上讲的方法，在我们每一层语法分析函数的最后都会多读一个单词，我也是按照这样的方式编写的语法分析函数，但是由于我将 getSym() 与 单词解析输出进行了绑定，这样就会导致最后多读的那个单词都会在`<语法成分>`之前输出，这样一来，`<语法成分>`输出之前总会多输出一个单词解析，导致了输出顺序的紊乱，对于种问题，大致有两种**解决思路**：

- 摒弃课上讲的那种多读一个单词的方式，每个函数只读到自己语法成分内的单词。
- 仍然保留多读一个单词的方式，而调整 getSym() 与 单词解析输出 的绑定。

我刚开始有想采用前一种方法，但是因为无法预测下一个读的是本语法成分的单词还是下一个语法成分的单词，这就势必导致一次回退，很不优雅！因此，我朝着第二种思路努力挖掘潜在的改进可能，经过较长时间的思考和讨论，我发现了一个具有较小修改量、不违背递归向下的原则的一种优雅的解决方式：

仍然将 getSym() 与 单词解析输出 进行绑定，但采用错位绑定！具体来说，就是在 getSym() 获取新的单词的同时，将上一个单词的解析进行输出，而读到的新的单词则在下一次 getSym() 的时候才输出。这样一来，每一次语法分析程序结束时多读的单词都不会立马进行输出，而要等到下一语法成分解析的开头才会输出。这样，不仅完美解决了输出顺序紊乱的问题，而且修改的代码量也只有仅仅两三行，极其优雅！

解决了上面的一大问题，剩下的就好办了，每个文法成分的 parse 函数基本都是按照文法定义，依葫芦画瓢，只稍微需要在某些地方注意一下是否需要多读一个或者少读一个单词。

在之后的过程中还遇到了一些**小问题**，比较典型的是：

<有返回值函数调用语句> ::= <标识符>'('<值参数表>')'
<无返回值函数调用语句> ::= <标识符>'('<值参数表>')'

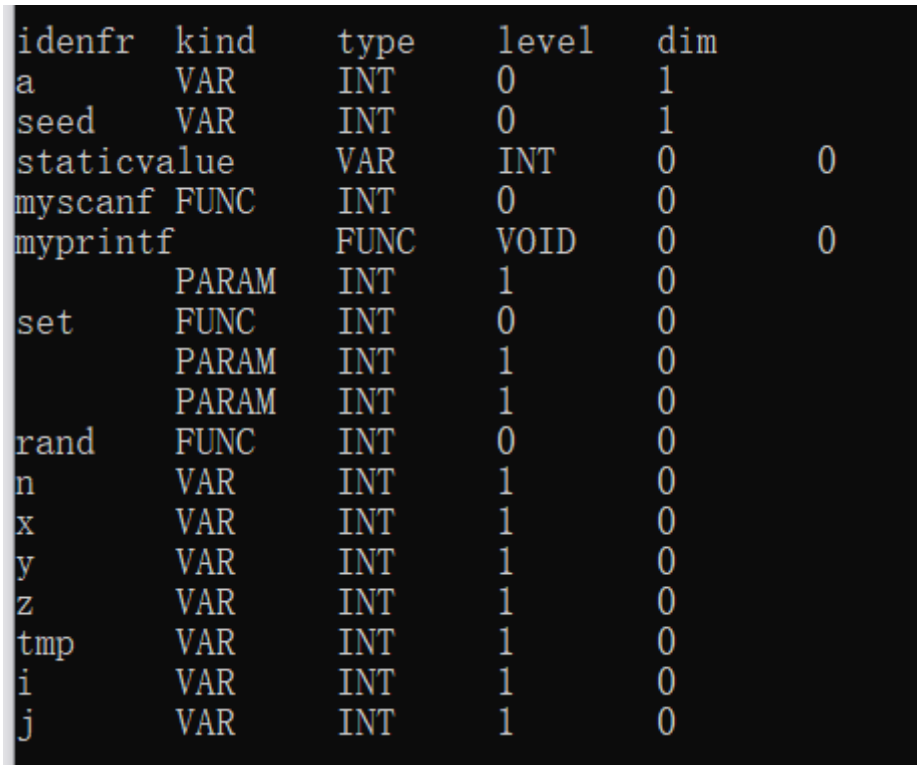
显然，这两个特殊的语法成分，其等号右边是完全一致的，单从语法树的角度来看，没有任何区别，要将其进行区分就得看后来读出来的这个`<标识符>`所指代的具体函数是否有返回值，这就需要我们提前对函数的类型进行额外的记录，我建立了一个map来保存某个函数名对应的函数是 RETURNABLE 还是 UNRETURNABLE , 然后利用读出的`<标识符>`进行查表，从而进入不同的分支子程序。

2、编码后对设计的修改

下一次实验是错误处理，必然是一个棘手的问题，如果不做限定的话，那可能出现的错误根本很难数得过来，因此得看下次具体需要做哪些类型的错误处理，这次编码完成之后，在语法解析函数的某些比较典型的错误中预留了error()错误处理函数的接口，具体实现就交给下次作业了，

同时，这次实验开始感觉到代码量的恐怖了，上次还将代码控制在了200行以内，这次就直接飙到了1000行，已经可以提前感知到之后代码量的凶残了，因此在这次编码完成之后，对源代码进行了分割而不是只写一个.cpp文件了。

我还将符号表信息输出到了控制台，便于调试时进行对照纠察



idenfr	kind	type	level	dim	
a	VAR	INT	0	1	
seed	VAR	INT	0	1	
staticvalue	VAR	INT	0		0
myscanf	FUNC	INT	0	0	
myprintf		FUNC	VOID	0	0
	PARAM	INT	1	0	
set	FUNC	INT	0	0	
	PARAM	INT	1	0	
	PARAM	INT	1	0	
rand	FUNC	INT	0	0	
n	VAR	INT	1	0	
x	VAR	INT	1	0	
y	VAR	INT	1	0	
z	VAR	INT	1	0	
tmp	VAR	INT	1	0	
i	VAR	INT	1	0	
j	VAR	INT	1	0	

三、错误处理

1、针对本次实验的设计

1.1 新增数据结构

1）符号表

对于每一个符号表项，建立一个结构体：

```
struct Symbol
{
    string idenfr; // 标识符
    int kind;
    int type;
    int level; // 作用域的层数
    int dim; // 维数
};

enum KindEnum { // kind的枚举
    CONST, // 常量
    VAR, // 变量
    FUNC, // 函数
```

```
    PARAM // 参数
};

enum TypeEnum { // type的枚举
    CHAR,
    INT,
    VOID
};
```

符号表则为 Symbol 的数组 `Symbol symTab[10000];`，采用栈式结构

同时为符号表建立一个 [子程序索引表] 记录每一层域的索引 `int subProcIndexTable[1000];`

其中特殊的，对于函数，其函数名属于外层域，函数参数以及函数体里定义的量属于内层域，当出函数体之后，将函数体里定义的变量全部从符号表中弹栈，仅保留函数名和参数，其中参数符号表项的idenfr 清空为 空串

2）错误信息打印

专门建立一个 ErrorProcessing类，为每条错误信息建立一个ErrorOutput结构体：

```
struct ErrorOutput
{
    int lineID; // 错误所在行号
    string errType; // 错误类别码
};
```

为 ErrorOutput 建立比较函数

建立一个错误信息数组 `vector<ErrorOutput> errOutVec;`，为其建立一个按行号递增排序的 `void sortErrOutVec();` 函数和一个去重插入函数 `void insertError(int lineID, string errType);`

在所有主程序执行的最后，调用 `void outputErrors()` 函数将错误信息表中的所有错误信息项按行号递增依次输出到 error.txt 文件中

1.2 错误解析与处理

a、**【非法符号或不符合词法】**例如字符与字符串中出现非法的符号，符号串中无任何符号

相关文法定义：

- <字符> ::='<加法运算符>'|'<乘法运算符>'|'<字母>'|'<数字>'
- <字符串> ::="{十进制编码为32,33,35-126的ASCII字符}" 且 非空

  方法：在**词法分析**阶段判断

b、**【名字重定义】**同一个作用域内出现相同的名字（**不区分大小写**）

相关文法定义：

- <常量定义> ::= int<标识符>=<整数>{,<标识符>=<整数>} | char<标识符>=<字符>{,<标识符>=<字符>}
- <声明头部> ::= int<标识符> |char<标识符>

- ＜变量定义无初始化＞ ::= ＜类型标识符＞(＜标识符＞|＜标识符＞'['＜无符号整数＞']'|＜标识符＞'['＜无符号整数＞']''['＜无符号整数＞']'){,(＜标识符＞|＜标识符＞'['＜无符号整数＞']'|＜标识符＞'['＜无符号整数＞']''['＜无符号整数＞']')}
- ＜变量定义及初始化＞ ::= ＜类型标识符＞＜标识符＞=＜常量＞|＜类型标识符＞＜标识符＞'['＜无符号整数＞']'='{'＜常量＞{,＜常量＞}'}'|＜类型标识符＞＜标识符＞'['＜无符号整数＞']''['＜无符号整数＞']'='{''{'＜常量＞{,＜常量＞}'}'{,'{'＜常量＞{,＜常量＞}'}'}'}'
- ＜无返回值函数定义＞ ::= void＜标识符＞'('＜参数表＞')''{'＜复合语句＞'}'
- ＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

  方法：查同一层的符号表，转换为小写进行比较

c、**【未定义的名字】**引用未定义的名字

相关文法定义：

- ＜因子＞ ::= ＜标识符＞|＜标识符＞'['＜表达式＞']'|＜标识符＞'['＜表达式＞']''['＜表达式＞']'|'('＜表达式＞')'|＜整数＞|＜字符＞|＜有返回值函数调用语句＞
- ＜赋值语句＞ ::= ＜标识符＞=＜表达式＞|＜标识符＞'['＜表达式＞']'=＜表达式＞|＜标识符＞'['＜表达式＞']''['＜表达式＞']'=＜表达式＞
- ＜循环语句＞ ::= while '('＜条件＞')'＜语句＞|for'('＜标识符＞=＜表达式＞;＜条件＞;＜标识符＞=＜标识符＞(+|-)＜步长＞')'＜语句＞
- ＜有返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'
- ＜无返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'
- ＜读语句＞ ::= scanf '('＜标识符＞')'

  方法：检查整个符号表，转换为小写进行比较

d、**【函数参数个数不匹配】**函数调用时实参个数大于或小于形参个数

e、**【函数参数类型不匹配】**函数调用时形参为整型，实参为字符型；或形参为字符型，实参为整型

相关文法定义：

- ＜有返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'
- ＜无返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'
- ＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}|＜空＞

  统一处理方法：先逐个依次检查参数类型是否匹配，同时进行参数计数，当检测到类型不匹配时就将 ‘e’ 错误信息存放到错误信息数组中，放之前先检查上一条错误信息是否与之重复，重复则不再放入；当计数完毕后再检查参数数目是否匹配，如果不匹配就将 ‘d‘ 错误信息加入到错误信息数组中，加入之前先检查上一条错误信息是否是同一行的 ‘e‘ 错误，如果是的话，就将其覆盖，意图是：当同一行同时出现类型、数目不匹配时，优先输出数目不匹配。

f、**【条件判断中出现不合法的类型】**条件判断的左右表达式只能为整型，其中任一表达式为字符型即报错，例如‘a’==1

相关文法定义：

- ＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞

方法：在表达式解析递归子程序中添加一个返回值type，返回该表达式的类型，然后判断 '==' 两边表达式解析递归子程序的返回值是否都为 INT。

g、**【无返回值的函数存在不匹配的return语句】**无返回值的函数中可以没有return语句，也可以有形如return;的语句，若出现了形如return(表达式);或return();的语句均报此错误

相关文法定义：

- <无返回值函数定义> ::= void <标识符> '(' <参数表> ')''{' <复合语句> '}'
- <复合语句> ::= [<常量说明>] [<变量说明>] <语句列>
- <语句列> ::= {<语句>}
- <语句> ::= <循环语句> | <条件语句> | <有返回值函数调用语句>; | <无返回值函数调用语句>; | <赋值语句>; | <读语句>; | <写语句>; | <情况语句> | <空>; | <返回语句>; | '{' <语句列> '}'
- <返回语句> ::= return['(' <表达式> ')']

方法：

因为从<无返回值函数定义>到<返回语句>之间存在多层函数递归调用，因此势必要进行跨函数状态处理，有两种解决方法：一是函数传参，二是设置宏观全局状态变量。前者容易维护但是参数传递需要改动较多，后者写起来简单，不过容易忘记维护。根据个人喜好，我选择了后者，设置了

`int funcRetFlag = -1;// -1:不在函数里 0：在无返回值函数里 1：在有返回值函数里`

当进入void函数中时，将funcRetFlag设置为0，出函数时设置为-1

用来存储当前处理的单词是否处于函数中，如果是，又是处于哪种类型的函数中。

那么在进入解析<返回语句>的递归子程序中时，如果检测到 '(' 就判断 `funcRetFlag == 0` 是否成立，如果成立则报错

h、**【有返回值的函数缺少return语句或存在不匹配的return语句】**例如有返回值的函数无任何返回语句；或有形如return;的语句；或有形如return();的语句；或return语句中表达式类型与返回值类型不一致

相关文法定义：

- <有返回值函数定义> ::= <声明头部> '(' <参数表> ')' '{' <复合语句> '}'
- <复合语句> ::= [<常量说明>] [<变量说明>] <语句列>
- <语句列> ::= {<语句>}
- <语句> ::= <循环语句> | <条件语句> | <有返回值函数调用语句>; | <无返回值函数调用语句>; | <赋值语句>; | <读语句>; | <写语句>; | <情况语句> | <空>; | <返回语句>; | '{' <语句列> '}'
- <返回语句> ::= return['(' <表达式> ')']

方法：

同样采用宏观全局状态变量，

```
int funcRetFlag = -1;// -1:不在函数里 0：在无返回值函数里 1：在有返回值函数里
int hasRet = 0; // 当前带返回值的函数 是否 有return语句
int retType = -1; // 当前带返回值函数 的 返回值类型 0: CHAR 1: INT
```

当进入有返回值的函数中时，将funcRetFlag设置为1，出函数时设置为-1

当进入返回语句解析递归子程序时，判断如果funcRetFlag == 1，则将hasRet置为1

首先检测是否有

1、return;

2、return();

3、表达式类型与返回值类型不一致

有则报错

之后检测hasRet是否为0，为0则报错

i、**【数组元素的下标只能是整型表达式】**数组元素的下标不能是字符型

相关文法定义：

- < 因子 > ::= < 标识符 > | < 标识符 > '[' < 表达式 > ']' | < 标识符 > '[' < 表达式 > ']''[' < 表达式 > ']' | '(' < 表达式 > ')' | < 整数 > | < 字符 > | < 有返回值函数调用语句 >
- < 赋值语句 > ::= < 标识符 > = < 表达式 > | < 标识符 > '[' < 表达式 > ']'= < 表达式 > | < 标识符 > '[' < 表达式 > ']''[' < 表达式 > ']' = < 表达式 >

方法：

在解析相关语法成分时，判断parseExpr()的返回值是否为CHAR，是则报错

j、**【不能改变常量的值】**这里的常量指的是声明为const的标识符。例如 const int a=1;在后续代码中如果出现了修改a值的代码，如给a赋值或用scanf获取a的值，则报错。

相关文法定义：

- < 赋值语句 > ::= < 标识符 > = < 表达式 > | < 标识符 > '[' < 表达式 > ']'= < 表达式 > | < 标识符 > '[' < 表达式 > ']''[' < 表达式 > ']' = < 表达式 >
- < 读语句 > ::= scanf '(' < 标识符 > ')'

方法：

在相关子程序中，判断在符号表中查找相关的标识符，判断其kind值是否为 CONST，是则报错

k、【应为分号】应该出现分号的地方没有分号，例如int x=1缺少分号 （7种语句末尾，for语句中，常量定义末尾，变量定义末尾）

相关文法定义：

- < 常量说明 > ::= const < 常量定义 > ;{ const < 常量定义 > ;}
- < 变量说明 > ::= < 变量定义 > ;{ < 变量定义 > ;}
- < 语句 > ::= < 循环语句 > | < 条件语句 > | < 有返回值函数调用语句 > ; | < 无返回值函数调用语句 > ; | < 赋值语句 > ; | < 读语句 > ; | < 写语句 > ; | < 情况语句 > | < 空 > ; | < 返回语句 > ; | '{' < 语句列 > '}'
- < 循环语句 > ::= while '(' < 条件 > ')' < 语句 > | for'(' < 标识符 > = < 表达式 > ; < 条件 > ; < 标识符 > = < 标识符 > (+|-) < 步长 > ')' < 语句 >

方法：

此处需要注意输出的行号，如果 ' ; ' 在行末，如果缺少，则会读到下一行，此时输出的行号就会变成出错的行号的下一行。

解决方式：为每一个单词添加一个bool属性isAtHead，如果为真，则表示该单词在行首

那么当检测到缺少分号时，就判断当前读到的单词的isAtHead是否为真，为真则输出 当前行号 - 1 ，否则就输出当前行号

l、【应为右小括号')'】应该出现右小括号的地方没有右小括号，例如fun(a,b;，缺少右小括号（有/无参数函数定义，主函数，带括号的表达式，if，while，for，switch，有/无参数函数调用，读、写、return）

相关文法定义：

- < 有返回值函数定义 > ::= < 声明头部 > '(' < 参数表 > ')' '{' < 复合语句 > '}'
- < 无返回值函数定义 > ::= void < 标识符 > '(' < 参数表 > ')''{' < 复合语句 > '}'
- < 主函数 > ::= void main‘(’)‘ ‘{’ < 复合语句 > ‘}’
- < 因子 > ::= < 标识符 > | < 标识符 > '[' < 表达式 > ']' | < 标识符 > '[' < 表达式 > ']''[' < 表达式 > ']' | '(' < 表达式 > ')' | < 整数 > | < 字符 > | < 有返回值函数调用语句 >
- < 条件语句 > ::= if '(' < 条件 > ')' < 语句 > [else < 语句 >]
- < 循环语句 > ::= while '(' < 条件 > ')' < 语句 > | for'(' < 标识符 > = < 表达式 > ; < 条件 > ; < 标识符 > = < 标识符 > (+|-) < 步长 > ')' < 语句 >
- < 情况语句 > ::= switch ‘(’ < 表达式 > ‘)’ ‘{’ < 情况表 > < 缺省 > ‘}’
- < 有返回值函数调用语句 > ::= < 标识符 > '(' < 值参数表 > ')'
- < 无返回值函数调用语句 > ::= < 标识符 > '(' < 值参数表 > ')'
- < 读语句 > ::= scanf '(' < 标识符 > ')'
- < 写语句 > ::= printf '(' < 字符串 > , < 表达式 > ')' | printf '(' < 字符串 > ')' | printf '(' < 表达式 > ')'
- < 返回语句 > ::= return['(' < 表达式 > ')']

方法：

此处需要注意输出的行号，如果 ') ' 在行末，如果缺少，则会读到下一行，此时输出的行号就会变成出错的行号的下一行。

解决方式：为每一个单词添加一个bool属性isAtHead，如果为真，则表示该单词在行首

那么当检测到缺少 ')' 时，就判断当前读到的单词的isAtHead是否为真，为真则输出 当前行号 - 1 ，否则就输出当前行号

m、【应为右中括号']'】应该出现右中括号的地方没有右中括号，例如int arr[2;缺少右中括号（一维/二维数组变量定义有/无初始化，因子中的一维/二维数组元素，赋值语句中的数组元素）

相关文法定义：

- < 变量定义无初始化 > ::= < 类型标识符 > (< 标识符 > | < 标识符 > '[' < 无符号整数 > ']' | < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']'){,(< 标识符 > | < 标识符 > '[' < 无符号整数 > ']' | < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']')}
- < 变量定义及初始化 > ::= < 类型标识符 > < 标识符 > = < 常量 > | < 类型标识符 > < 标识符 > '[' < 无符号整数 > ']'='{' < 常量 > {, < 常量 > }'}' | < 类型标识符 > < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']'='{''{' < 常量 > {, < 常量 > }'}'{, '{' < 常量 > {, < 常量 > }'}'}'}'
- < 因子 > ::= < 标识符 > | < 标识符 > '[' < 表达式 > ']' | < 标识符 > '[' < 表达式 > ']''[' < 表达式 > ']' | '(' < 表达式 > ')' | < 整数 > | < 字符 > | < 有返回值函数调用语句 >
- < 赋值语句 > ::= < 标识符 > = < 表达式 > | < 标识符 > '[' < 表达式 > ']'= < 表达式 > | < 标识符 > '[' < 表达式 > ']''[' < 表达式 > ']' = < 表达式 >

方法：

此处需要注意输出的行号，如果 '] ' 在行末，如果缺少，则会读到下一行，此时输出的行号就会变成出错的行号的下一行。

解决方式：为每一个单词添加一个bool属性isAtHead，如果为真，则表示该单词在行首

那么当检测到缺少 ']' 时，就判断当前读到的单词的isAtHead是否为真，为真则输出 当前行号 - 1 ，否则就输出当前行号

n、【数组初始化个数不匹配】任一维度的元素个数不匹配，或缺少某一维的元素即报错。例如int a[2][2]={{1,2,3},{1,2}}

相关文法定义：

- < 变量定义及初始化 > ::= < 类型标识符 > < 标识符 > = < 常量 > | < 类型标识符 > < 标识符 > '[' < 无符号整数 > ']'='{' < 常量 > {, < 常量 > }'}' | < 类型标识符 > < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']'='{''{' < 常量 > {, < 常量 > }'}'{, '{' < 常量 > {, < 常量 > }'}'}'}'

方法：

在解析<变量定义及初始化>时，记录中括号中的无符号整数，然后分一维、二维，建立循环遍历 '=' 右边的成分，出现异常即报错

o、【<常量>类型不一致】变量定义及初始化和switch语句中的<常量>必须与声明的类型一致。int x='c';int y;switch(y){case('1') ....}

相关文法定义：

- < 变量定义及初始化 > ::= < 类型标识符 > < 标识符 > = < 常量 > | < 类型标识符 > < 标识符 > '[' < 无符号整数 > ']'='{' < 常量 > {, < 常量 > }'}' | < 类型标识符 > < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']'='{''{' < 常量 > {, < 常量 > }'}'{, '{' < 常量 > {, < 常量 > }'}'}'}'
- < 情况语句 > ::= switch '(' < 表达式 > ')' '{' < 情况表 > < 缺省 > '}'
- < 情况表 > ::= < 情况子语句 > { < 情况子语句 > }
- < 情况子语句 > ::= case < 常量 > : < 语句 >

方法：

对于<常量>解析递归子程序parseConst()，返回值为常量的类型，INT和CHAR

对于switch语句，采用传递type参数进入情况子语句进行类型匹配检查

p、【缺少缺省语句】switch语句中，缺少<缺省>语句。

相关文法定义：

- < 情况语句 > ::= switch '(' < 表达式 > ')' '{' < 情况表 > < 缺省 > '}'
- < 缺省 > ::= default : < 语句 >

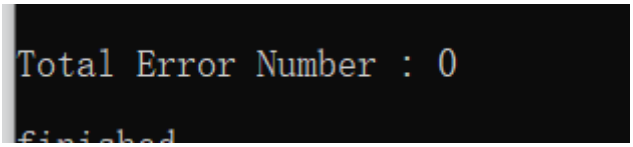
方法：直接在出了<情况表>子程序之后，看有没有"DEFAULTTK"

错误类型	错误类别码	解释及举例
非法符号或不符合词法	a	例如字符与字符串中出现非法的符号，符号串中无任何符号
名字重定义	b	同一个作用域内出现相同的名字（不区分大小写）
未定义的名字	c	引用未定义的名字 (不区分大小写)
函数参数个数不匹配	d	函数调用时实参个数大于或小于形参个数
函数参数类型不匹配	e	函数调用时形参为整型，实参为字符型；或形参为字符型，实参为整型
条件判断中出现不合法的类型	f	条件判断的左右表达式只能为整型，其中任一表达式为字符型即报错，例如'a'==1
无返回值的函数存在不匹配的return语句	g	无返回值的函数中可以没有return语句，也可以有形如return;的语句，若出现了形如return(表达式);或return();的语句均报此错误
有返回值的函数缺少return语句或存在不匹配的return语句	h	例如有返回值的函数无任何返回语句；或有形如return;的语句；或有形如return();的语句；或return语句中表达式类型与返回值类型不一致
数组元素的下标只能是整型表达式	i	数组元素的下标不能是字符型
不能改变常量的值	j	这里的常量指的是声明为const的标识符。例如 const int a=1;在后续代码中如果出现了修改a值的代码，如给a赋值或用scanf获取a的值，则报错。
应为分号	k	应该出现分号的地方没有分号，例如int x=1缺少分号 （7种语句末尾，for语句中，常量定义末尾，变量定义末尾）
应为右小括号')'	l	应该出现右小括号的地方没有右小括号，例如fun(a,b;，缺少右小括号（有/无参数函数定义，主函数，带括号的表达式，if，while，for，switch，有/无参数函数调用，读、写、return）
应为右中括号']'	m	应该出现右中括号的地方没有右中括号，例如int arr[2;缺少右中括号（一维/二维数组变量定义有/无初始化，因子中的一维/二维数组元素，赋值语句中的数组元素）
数组初始化个数不匹配	n	任一维度的元素个数不匹配，或缺少某一维的元素即报错。例如int a[2][2]={{1,2,3},{1,2}}
<常量>类型不一致	o	变量定义及初始化和switch语句中的<常量>必须与声明的类型一致。int x='c';int y;switch(y){case('1') ....}
缺少缺省语句	p	switch语句中，缺少<缺省>语句。

单词名称	类别码	单词名称	类别码	单词名称	类别码	单词名称	类别码
标识符	IDENFR	else	ELSETK	-	MINU	=	ASSIGN
整形常量	INTCON	switch	SWITCHTK	*	MULT	;	SEMICN
字符常量	CHARCON	case	CASETK	/	DIV	,	COMMA
字符串	STRCON	default	DEFAULTTK	<	LSS	(	LPARENT
const	CONSTTK	while	WHILETK	<=	LEQ	)	RPARENT
int	INTTK	for	FORTK	>	GRE	[	LBRACK
char	CHARTK	scanf	SCANFTK	>=	GEQ	]	RBRACK
void	VOIDTK	printf	PRINTFTK	==	EQL	{	LBRACE
main	MAINTK	return	RETURNTK	!=	NEQ	}	RBRACE
if	IFTK	+	PLUS	:	COLON		

2、编码后对设计的修改

将错误处理有关的所有操作和函数，都封装在ErrorProcessing类中，在其中建立存储错误信息的数据结构，通过error函数将错误信息保存在实例化后的ErrorProcessing对象中，之后将错误信息输出到error.txt文件中，同时对错误信息的统计数据，我会将其输出到控制台进行显示，方便在之后进行调试，同时加了错误处理开关，可随时开启和关闭错误处理。



四、代码生成

1、针对本次实验的设计

1.1 阶段一

1.1.1 本次实验涉及到的文法内容

输入文件：testfile.txt 输出文件：mips.txt 中文编码格式：UTF-8

1）**常量说明**

```
<常量说明> ::= const<常量定义>;{ const<常量定义>;}
<常量定义> ::= int<标识符>=<整数>{,<标识符>=<整数>} | char<标识符>=<字符>{,<标识符>=<字符>}

<标识符> ::= <字母>{<字母>|<数字>}
<整数> ::=[+|-]<无符号整数>
<字符> ::= '<加法运算符>'|'<乘法运算符>'|'<字母>'|'<数字>'

<无符号整数> ::= <数字>{<数字>}
<字母> ::= _|a|...|z|A|...|Z
<数字> ::= 0|1|...|9
<加法运算符> ::= +|-
<乘法运算符> ::= *|/
```

2）**变量说明**

```
<变量说明> ::= <变量定义>;{<变量定义>;}
<变量定义> ::= <变量定义无初始化>|<变量定义及初始化>

<变量定义无初始化> ::= <类型标识符>(<标识符>){,(<标识符>)}
<变量定义及初始化> ::= <类型标识符><标识符>=<常量>

<类型标识符> ::= int | char
<标识符> ::= <字母>{<字母>|<数字>}
<常量> ::= <整数>|<字符>

<整数> ::=[+|-]<无符号整数>
<字符> ::= '<加法运算符>'|'<乘法运算符>'|'<字母>'|'<数字>'
<字母> ::= _|a|...|z|A|...|Z
<数字> ::= 0|1|...|9

<无符号整数> ::= <数字>{<数字>}
<加法运算符> ::= +|-
<乘法运算符> ::= *|/
```

3）**读语句**

```
<读语句> ::= scanf '('<标识符>')'

<标识符> ::= <字母>{<字母>|<数字>}
<字母> ::= _|a|...|z|A|...|Z
<数字> ::= 0|1|...|9
```

4）**写语句**

```
<写语句> ::= printf'('<字符串>,<表达式>')'| printf'('<字符串>')'|
printf'('<表达式>')'

<字符串> ::= "{十进制编码为32,33,35-126的ASCII字符}"
<表达式> ::= [+|-]<项>{<加法运算符><项>}

<项> ::= <因子>{<乘法运算符><因子>}
<加法运算符> ::= +|-

<因子> ::= <标识符> | '('<表达式>')'|<整数>|<字符>
<乘法运算符> ::= *|/
```

5）**赋值语句**

```
<赋值语句> ::= <标识符>=<表达式>

<标识符> ::= <字母>{<字母>|<数字>}
<表达式> ::= [+|-]<项>{<加法运算符><项>}

<字母> ::= _|a|...|z|A|...|Z
<数字> ::= 0|1|...|9
<项> ::= <因子>{<乘法运算符><因子>}
<加法运算符> ::= +|-

<因子> ::= <标识符> | '('<表达式>')'|<整数>|<字符>
<乘法运算符> ::= *|/
```

6）**无函数定义及调用**：nice

7）**无数组声明及引用**：nice

1.1.2 注意事项

a）需自行设计**四元式中间代码**，再从中间代码生成MIPS汇编，需设计实现**输出**中间代码的有关函数，本次作业不考核，后续会有优化前后中间代码的输出及评判(输出文件命名为 学号_姓名_优化前/后中间代码.txt)。

b）后续的作业需参加竞速排序，需提前预留**代码优化有关的接口**，并设计方便**切换开启/关闭优化的模式**

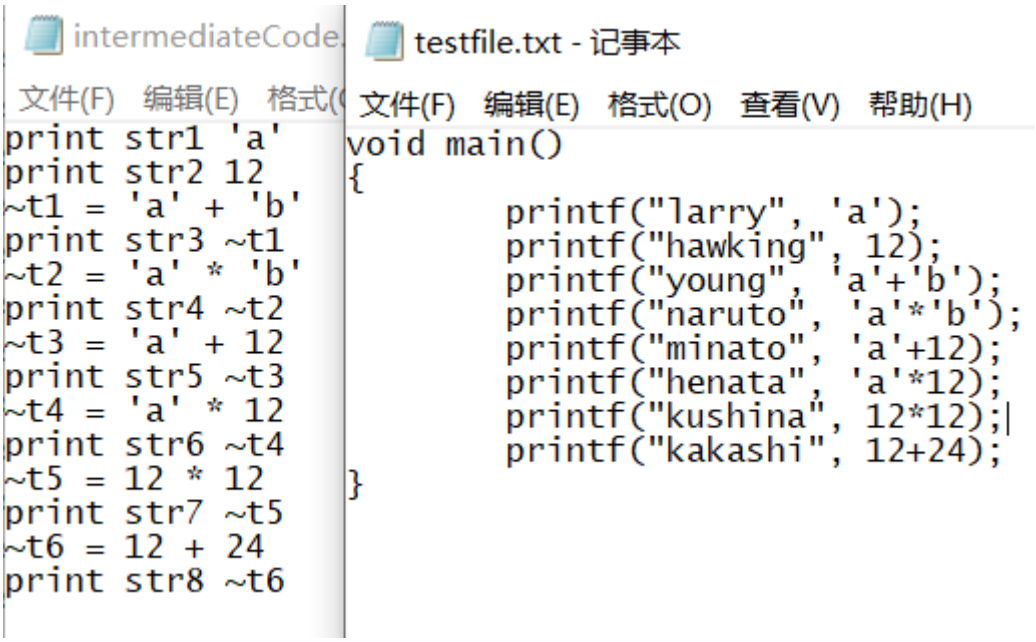
1.1.3 针对本次实验的设计

1）四元式格式设计：

	ans	x	op	y	基本块入口
赋值I	ans	x	op	y	
赋值II	ans	x			
赋值IV	ans	x	addi	y	
赋值III	ans		-	y	
if语句表达式	label	x	cmpOp	y	下一句
标签语句	label		:		√
语句跳转	label		j		下一句
函数调用	label		jal		下一句
返回值设置	<标识符>		return		
把$v0返还存内存	ans		$v0		
返回跳转			jr		下一句
开辟栈空间		size(数值)	spAlloc	(备注)	
回收栈空间		size(数值)	spFree	(备注)	
参数/变量压栈	value	offset	spPush	(备注)	
$ra压栈			raPush		
$ra出栈			raPop		
从栈中取值	ans	offset	spGET	(备注)	
向栈中存值	value	offset	spSET		
函数中读语句将$v0存入栈中	$v0	offset(数值)	spSET$v0		
数组赋值压栈	value	offset	spARRSET		
数组赋值出栈	ans	offset	spARRGET		
用数组赋值	ans	数组名	ARRGET	offset	

	ans	x	op	y	基本块入口
数组被赋值(by变量)	value	数组名	ARRSETbyVAR	offset(VAR)	
数组被赋值(by常量)	value	数组名	ARRSETbyCONST	offset(CONST)	
读语句	ans		read		
读语句（特定类型)		type	ReadByType		
写语句I		< 字符串 > (标识符)	print	< 表达式 >	
写语句II		< 字符串 > (标识符)	print		
写语句III			print	< 表达式 >	

部分中间代码对照图：



2）存储管理设计：

为了优先保证基本正确性得到满足，本次实验设计先暂且牺牲一定的效率，进一步的优化放在之后再进行，本次实验暂且没有充分利用寄存器的存储功能，而是将所有变量、常量、字符串都存储在以 0x10010000 (将该基地址值始终保存在 $t0 寄存器中) 为起始的内存空间中，只进行运算时，将其分别 lw 给 $t2 和 $t3 寄存器，并将运算结果保存在 $t1 寄存器中，然后将 $t1 寄存器的值 sw 到内存中或者输出。

3）符号表设计：

本次实验额外分别建立了一个**代码生成专用符号表** 和 一个**字符串记录表**，前者专门存储本次实验定义的常变量以及复杂表达式拆解过程中用到的中间变量，对于每一个表项，主要记录数据类型和在内存中相对于基地址0x10010000的地址偏移；后者记录本次所有printf语句中需要输出的字符串，对于每一个表项，主要记录该字符串的标识符、内容、在内存中的起始地址相对于基地址0x10010000的偏移值，字符串的长度(按字节计)。

因为本次作业是建立在错误处理通过的基础之上的，因此不会出现变、常量名未定义或者重定义的情况，因此当用到某一个量时，就在本次新建的符号表中由后往前遍历查找对应标识符的表项。

值得注意的是，因为在复杂表达式拆解过程当中会产生许多中间量，其标识符的命名需要避免与程序本身定义的常量、变量名冲突，唯一的办法就是采用与一般标识符不同的命名规范，我的做法是在标识符之前加一个'~'波浪符以示区分。

4）复杂表达式的处理

本次语义分析的过程与语法分析的流程相融合，采用课上所讲的语法制导属性翻译文法，对每个与表达式计算相关的递归子程序，添加参数传入与返回值，从而以递归的方式对复杂表达式进行拆解，实现起来改动较小，极为方便，稍加细心便不会出错。

1.2 阶段二

1.2.1 本次实验新增的文法内容

1）一维数组 2）二维数组 3）函数定义与调用

1.2.2 针对本次实验的设计

1）一维数组

之前不存在数组，因此在符号表中所有量都以4字节为单位存储，当引入数组之后，需要在语法分析阶段对数组的大小进行计算，从而确定需要开辟的内存空间大小，并将空间偏移量记录在符号表中，而对数组中某一元素进行存取的时候，如arr[i]，则计算偏移量为offset = arr.offset + 4 * i，相对来说较好处理

2）二维数组

二维数组的存取就复杂多了，我们需要提前记录每一个二维数组的列数col，在对于arr[i][j]进行存取的时候，offset = arr.offset + (i * col + j) * 4

3) 函数定义及调用

我采用将函数中所有的量，包括参数、定义的量、中间变量、$ra的值，都存在$sp 的栈中，每当调用一个函数的时候，就将所有参数压栈，出函数之后让所有参数出栈，进入函数之后将$ra的值存进栈里，每遇到一个函数中定义的量就将其存进栈中，在函数中调用其他函数之前，需要先将所有运算过程中产生的中间变量一起放进栈中，然后再将下一个函数的参数压栈，再jal到下一个函数。

因为函数中需要保存的量，其种类很多，而且需要和函数外面的量进行区分，因此我专门建立了一个函数类，用于保存每一个函数内各个量的类型、在$sp栈中的位置偏移等信息，当在函数语句中遇到一个量，优先在该函数的类中查找，若没找到，再到符号表中寻找，从而有效避免了冲突。

1.3 附录

1.3.1 四元式格式

	ans	x	op	y	基本块入口
赋值I	ans	x	op	y	
赋值II	ans	x			
赋值IV	ans	x	addi	y	
赋值III	ans		-	y	
if语句表达式	label	x	cmpOp	y	下一句
标签语句	label		:		√
语句跳转	label		j		下一句
函数调用	label		jal		下一句
返回值设置	<标识符>		return		
把$v0返还存内存	ans		$v0		
返回跳转			jr		下一句
开辟栈空间		size(数值)	spAlloc	(备注)	
回收栈空间		size(数值)	spFree	(备注)	
参数/变量压栈	value	offset	spPush	(备注)	
$ra压栈			raPush		
$ra出栈			raPop		
从栈中取值	ans	offset	spGET	(备注)	
向栈中存值	value	offset	spSET		
函数中读语句将$v0存入栈中	$v0	offset(数值)	spSET$v0		
数组赋值压栈	value	offset	spARRSET		
数组赋值出栈	ans	offset	spARRGET		
用数组赋值	ans	数组名	ARRGET	offset	

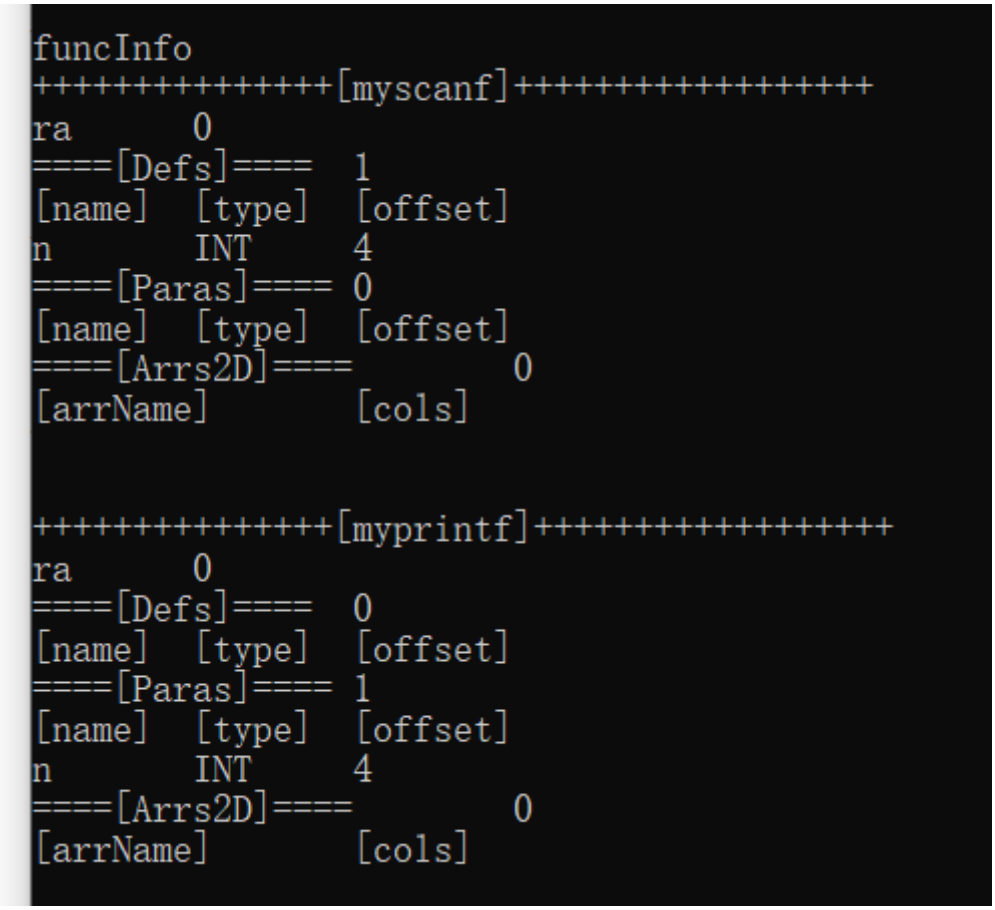
	ans	x	op	y	基本块入口
数组被赋值(by变量)	value	数组名	ARRSETbyVAR	offset(VAR)	
数组被赋值(by常量)	value	数组名	ARRSETbyCONST	offset(CONST)	
读语句	ans		read		
读语句（特定类型)		type	ReadByType		
写语句I		< 字符串 > (标识符)	print	< 表达式 >	
写语句II		< 字符串 > (标识符)	print		
写语句III			print	< 表达式 >	

1.3.2 寄存器表

REGISTER	NAME	USAGE
$0	$zero	常量0
$1	$at	保留给汇编器
$2-$3	$v0-$v1	函数调用返回值
$4-$7	$a0-$a3	函数调用参数
$8-$15	$t0-$t7	临时寄存器
$16-$23	$s0-$s7	全局寄存器
$24-$25	$t8-$t9	临时寄存器
$28	$gp	全局指针(Global Pointer)
$29	$sp	堆栈指针(Stack Pointer)
$30	$fp	帧指针(Frame Pointer)
$31	$ra	返回地址(return address)

2、编码后对设计的修改

编码完成后基本不存在错误了，因此主要的修改是添加一些可视化的调试工具，将许多重要的关键数据结构的信息，在程序末尾进行输出，比如，对于每一个函数我建了一个类，其中记录了该函数栈空间的数据存储信息，我会在程序结束时将它们输出：



由上往下一次是$ra寄存器栈地址、函数内部定义的量、函数传入的参数、函数中的二维数组的信息。

五、代码优化

1、针对本次实验的设计

主要做了如下几部分的优化：

一、对中间代码的优化

1) 寄存器分配

我对如下寄存器进行了全局分配：

```
vector<string> regPool = {"$v1", "$a1", "$a2", "$a3", "$t3",
                          "$t4", "$t5", "$t6", "$t7", "$t8", "$t9",
                      "$s0", "$s1", "$s2", "$s3", "$s4", "$s5", "$s6", "$s7", "$fp" };
```

分配方式，采用的是引用计数分配，首先统计一遍中间代码中每一个量被引用的次数，然后将它们按照次数由多到少进行排序，然后依次将上述寄存器分配给排序靠前的量，对于循环语句块中的量添加了一定的权重，为了实现的方便起见，我给每一个变量分配好了寄存器之后就没有再对它们回收寄存器，也就是说分配好了的寄存器就由该变量专享了，当然这不是最好的解决方式，但相对于我有限的时间和精力的情况下，这是最划算的解决方式，不仅能减去大量的mem操作，还能不应担心出玄学bug，所以最终我就只优化到这一程度了。

2) <项>的常数预算

在递归分析项的每个因子的乘除时，我首先建立了一个变量来存储乘积中的常量，特别的是在连乘中，先将常数提前计算好，然后在计算每一个标识符的运算，介绍了一部分常量相乘的指令，又因为除法是取整运算，因此会存在精度损失，所以不能够将此方法运用到除法中。

3) <表达式>的常数预算

表达式是若干个项进行加减来得到的所以其中可能会遇到很多常数的加减，我于是将所有常数的加减，全部计算得到一个值，然后再将其与标识符进行加减运算，减少了大量加减的指令。

4）常量传播

对于全局常量和函数中定义的常量，我分别建立全局常量表和局部常量表，局部常量表存储在函数类中，当在符号表中查找某一个标识符之前，现在常量表中查找是否存在同名的常量，然后将其替换为该常量的具体值。

5）循环优化

对于For循环和While循环，我将判断条件复制一次，在循环开头和循环结尾各判断一次，这样一来，每一轮循环结束，如果判断不满足下一轮循环的条件了，就直接按顺序执行出循环，而不用先跳转到开头再判断是否满足条件，这样的优化能够减少将近一半的branch指令条数。

6）窥孔优化

例如对于形如 + $t1 $t2 $t3

= $t3 $t4

这样的中间代码，我直接转化为 + $t1 $t2 $t4

二、对汇编代码的优化

1）对部分乘法用位移替换

对于标识符与2的幂次常数之间的乘法，我采用的是sll左移幂来替换该乘法，减少了一个数量级的mul指令的数目

2）对于除法，同样可以采用sra右移来替换，但值得注意的是，当被除数如果为负数，需要着重特判，如果能整除，则可以直接sra，如果不能整除就必须sra后再加1，这部分特判还必须写成汇编代码，因此会增加一些跳转指令，但这样的替换还是划算的因为div指令的权重很高，替换后还是会提高汇编指令的性能。

3）用加法替换乘2，乘除1或者-1时不用乘除法，加减0时取消该操作

4）j label

label:

这样的情况，我直接将前面的j指令去除

5）对于sw $t1 4($t0)

lw $t1 4($t0)

这样的情况，直接将lw指令去除

2、编码后对设计的修改

同样是加入一些代码优化操作的调试信息的输出，如寄存器分配前后，变量的存储地址，分配前是位于内存中的哪个位置，以及分配之后，是位于哪个寄存器中

分配前:

```
CodeGenSymbol_Before
[idenfr]        [kind]  [type]  [level] [offset]        [dim]
a       VAR     INT     0       0       1
seed    VAR     INT     0       4000    1
staticvalue     VAR     INT     0       4012    0
~t1     VAR     INT     1       4016    0
~t2     VAR     INT     1       4020    0
~t3     VAR     INT     1       4024    0
~t4     VAR     INT     1       4028    0
~t5     VAR     INT     1       4032    0
~t6     VAR     INT     1       4036    0
~t7     VAR     INT     1       4040    0
~t8     VAR     INT     1       4044    0
~t9     VAR     INT     1       4048    0
~t10    VAR     INT     1       4052    0
~t11    VAR     INT     1       4056    0
~t12    VAR     INT     1       4060    0
~t13    VAR     INT     1       4064    0
~t14    VAR     INT     1       4068    0
~t15    VAR     INT     1       4072    0
```

分配后:

```
CodeGenSymbol_After
[idenfr]        [kind]  [type]  [level] [offset]        [dim]
a       VAR     INT     0       0       1
seed    VAR     INT     0       4000    1
staticvalue     VAR     INT     0       $v1     0
~t1     VAR     INT     1       4016    0
~t2     VAR     INT     1       4020    0
~t3     VAR     INT     1       4024    0
~t4     VAR     INT     1       4028    0
~t5     VAR     INT     1       $t6     0
~t6     VAR     INT     1       $s2     0
~t7     VAR     INT     1       4040    0
~t8     VAR     INT     1       4044    0
~t9     VAR     INT     1       4048    0
~t10    VAR     INT     1       $s0     0
~t11    VAR     INT     1       $s1     0
~t12    VAR     INT     1       4060    0
~t13    VAR     INT     1       4064    0
~t14    VAR     INT     1       4068    0
~t15    VAR     INT     1       $s5     0
~t16    VAR     INT     1       $s6     0
```

同时，为了便于常量传播的调试，我将全局常量表的统计信息进行的输出查看:

```
[ConstMap]: 0
```