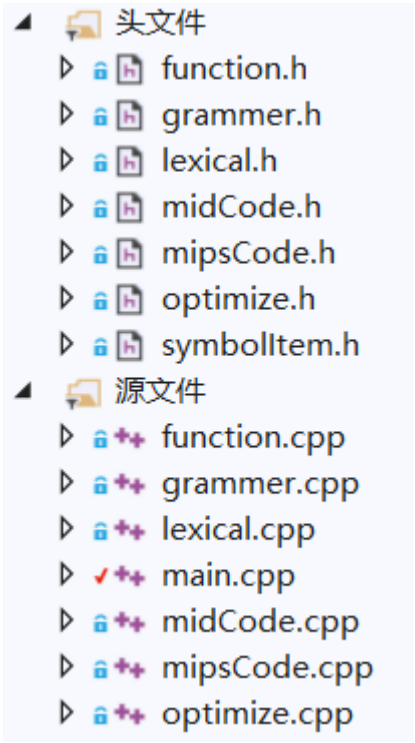
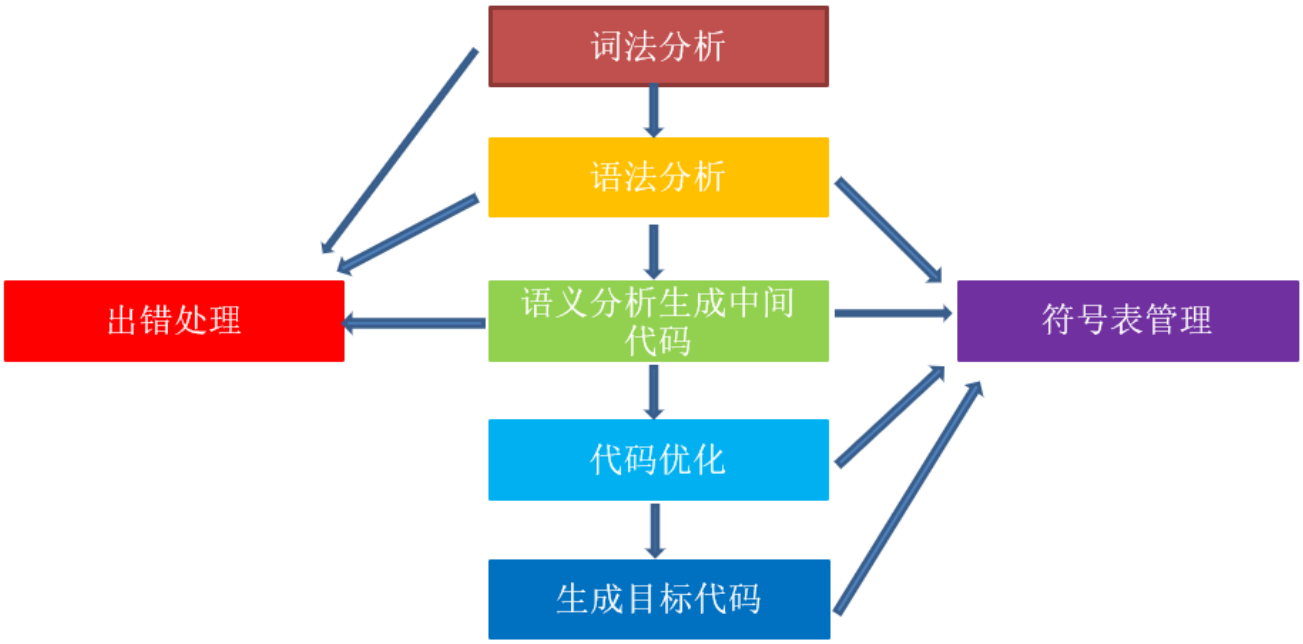
编译课设申优文章

一、整体架构：





二、具体设计与难点分析

（一）词法分析+错误处理的词法错误：`lexical.cpp`

1.1 主要操作

```
void clearToken();  //清空token
void catToken(); //连接到token后边
void get_ch();  //读一个字符
void retract(); //回退一个字符
void retractString(int oldIndex);  //回退到某一个位置
int getsym(int out=1);  //读一个符号
```

（二）语法分析+语义分析与生成中间代码+错误处理的其他错误：`grammer.cpp`

2.1 主要操作

```
//<字符串>::="{十进制编码为32,33,35-126的ASCII字符}"
bool strings();
//<程序>::=[<常量说明>] [<变量说明>]{<有返回值函数定义>|<无返回值函数定义>}<主函数>
bool procedure();
//<常量说明>::= const<常量定义>;{ const<常量定义>;}
bool constDeclaration(bool isglobal);
//<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>}
bool constDefinition(bool isglobal);
//<无符号整数>::= <非零数字> {<数字>} | 0
bool unsignedInteger(int& value);
//<整数>::=[+|-]<无符号整数>
bool integer(int& value);
//<声明头部> ::=  int<标识符> |char<标识符>
bool declarationHead(string& tmp, int& type);
//<变量说明>::= <变量定义>;{<变量定义>;}
bool variableDeclaration(bool isglobal);
//<变量定义>::= <类型标识符>(<标识符>|<标识符>'['<无符号整数>']') {,(<标识符>|<标识符>'['<无符号整数>']' )}
bool variableDefinition(bool isglobal);
//<有返回值函数定义> ::= <声明头部>'('<参数表>')' '{'<复合语句>'}'
bool haveReturnValueFunction();
//<无返回值函数定义> ::= void<标识符>'('<参数表>')''{'<复合语句>'}'
bool noReturnValueFunction();
//<参数表> ::= <类型标识符><标识符>{,<类型标识符><标识符>}| <空>
bool parameterTable(string funcName, bool isRedefine);
//<复合语句> ::=[<常量说明>] [<变量说明>] <语句列>
bool compoundStatement();
//<主函数> ::= void main‘(’‘)’ ‘{’<复合语句>‘}’
bool mainFunction();
//<表达式> ::= [+|-]<项>{<加法运算符><项>}
bool expression(int& type, string& ansTmp);
//<项>::= <因子>{<乘法运算符><因子>}
bool item(int& type, string& ansTmp);
//<因子> ::= <标识符>|<标识符>'['<表达式>']'|'('<表达式>')'|<整数>|<字符>|<有返回值函数调用语句>
bool factor(int& type, string& ansTmp);
//<语句>::= <条件语句>|<循环语句>| '{'<语句列>'}'| <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>;|<空>;|<返回语句>;
bool statement();
//<赋值语句>::= <标识符>=<表达式>|<标识符>'['<表达式>']'=<表达式>
```

```
bool assignStatement();
//<条件语句> ::= if '('<条件>')'<语句> [else<语句>]
bool conditionStatement();
//<条件>::= <表达式><关系运算符><表达式>|<表达式>
bool condition();
//<循环语句> ::= while '('<条件>')'<语句>| do<语句>while '('<条件>')'|for'('<标识符>=<表达式>;<条件>;<标识符>=<标识符>(+|-)<步长>')'<语句>
bool repeatStatement();
//<步长>::= <无符号整数>
bool step(int& value);
//<有返回值函数调用语句> ::= <标识符>'('<值参数表>')'
bool callHaveReturnValueFunction();
//<无返回值函数调用语句> ::= <标识符>'('<值参数表>')'
bool callNoReturnValueFunction();
//<值参数表>::= <表达式>{,<表达式>}|<空>
bool valueParameterTable(string funcName);
//<语句列> ::= {<语句>}
bool statementList();
//<读语句> ::= scanf '('<标识符>{,<标识符>}')'
bool readStatement();
//<写语句> ::= printf '(' <字符串>,<表达式> ')'| printf '('<字符串> ')'| printf '('<表达式>')'
bool writeStatement();
//<返回语句> ::= return['('<表达式>')']
bool returnStatement();
//有关中间代码生成的函数:
void checkBeforeFunc();
void fullNameMap(map<string, string>& nameMap, vector<midCode> ve, string funcName);
void dealInlineFunc(string name);
```

2.2 难点分析

（1）**变量定义与有返回值函数定义的冲突问题**；`int a;`，`int a,b;`，`int a[10];`与`int a();`

即处理到标识符之后，需要向后**预读**一个单词，判断如果是左括号，则认为是函数定义，这时应该退回到`int`之前，转入对函数定义的分析。

解决方法：设计了一个函数`void retractString(int oldIndex);`用于**回退**到某一个位置。

```
void retractString(int old) {
    for (int i = old; i < indexs; i++) {
        if (filecontent[i] == '\n') {
            line--;
        }
    }
    indexs = old;
}
```

类似的情况：无返回值函数与主函数`void f();`与`void main();`；因子中标识符、标识符[]、有返回值函数调用语句三者`a`、`a[]`、`a()`；语句中函数调用语句与赋值语句`a();`与`a=10,a[1]=10;`我都采用了预读，然后判断属于哪一种，然后回退，然后进入到对应的递归子程序。

（2）**错误处理的行号问题**

当读到一个符号，不符合当前语法分析结果时，应该把当前符号退掉，输出错误信息，然后把`symbol`改成正确的以正确执行后边的分析。在词法中设置一个`oldindex`，表示每次`getsym()`之前的`index`，也就是读入这个符号之前的位置，然后发生错误回退到`oldindex`，这样输出的行号就跟要求是一致的了。例如：

```
if (symbol != SEMICN) {
    retractString(oldIndex);
    errorfile << line << " k\n";  //缺少分号
    symbol = SEMICN;  //修正symbol
}
```

（三）符号表管理：`symbolItem.h`

3.1 数据结构

```
class symbolItem {
    string name;
    int kind; //var const function array
    int type; //int char void
    int constInt;
    char constChar;
    int length;  //数组长度  对于函数用于记录这个函数有多少变量（参数+局部变量+临时）
    vector<int> parameterTable;  //参数类型
    int addr;   //地址
}
map<string, symbolItem> globalSymbolTable;  //全局符号表
map<string, symbolItem> localSymbolTable;   //局部符号表(函数内部的) 函数结束会清空
map<string, map<string, symbolItem>> allLocalSymbolTable;  //保存所有的局部符号表
vector<string> stringList;  //保存所有的字符串
```

`symbolItem`中`name`为变量名/常量名/数组名/函数名；`kind`代表类别（变量/常量/数组/函数）；`type`对于函数代表其返回值类型，对于变量、常量、数组代表类型；`constInt`和`constChar`代表常量的值（其实可以只用一个`constInt`）；`length`对于数组表示数组长度，对于函数表示这个函数有多少个变量（参数+局部变量+临时变量）；`parameterTable`对于函数存储其参数类型；`addr`表示地址，常量/函数没有地址，变量/数组有地址，其实就是变量定义的顺序。

全局符号表`globalSymbolTable`：存储全局常量、变量、数组、函数、以及函数参数的类型。

局部符号表`localSymbolTable`：存储局部常量、变量、数组、函数的参数。局部符号表每次分析完一个函数定义后会清空，所以需要在此之前保留到`allLocalSymbolTable`中。

（四）中间代码设计：`midCode.cpp midCode.h`

4.1 四元式设计

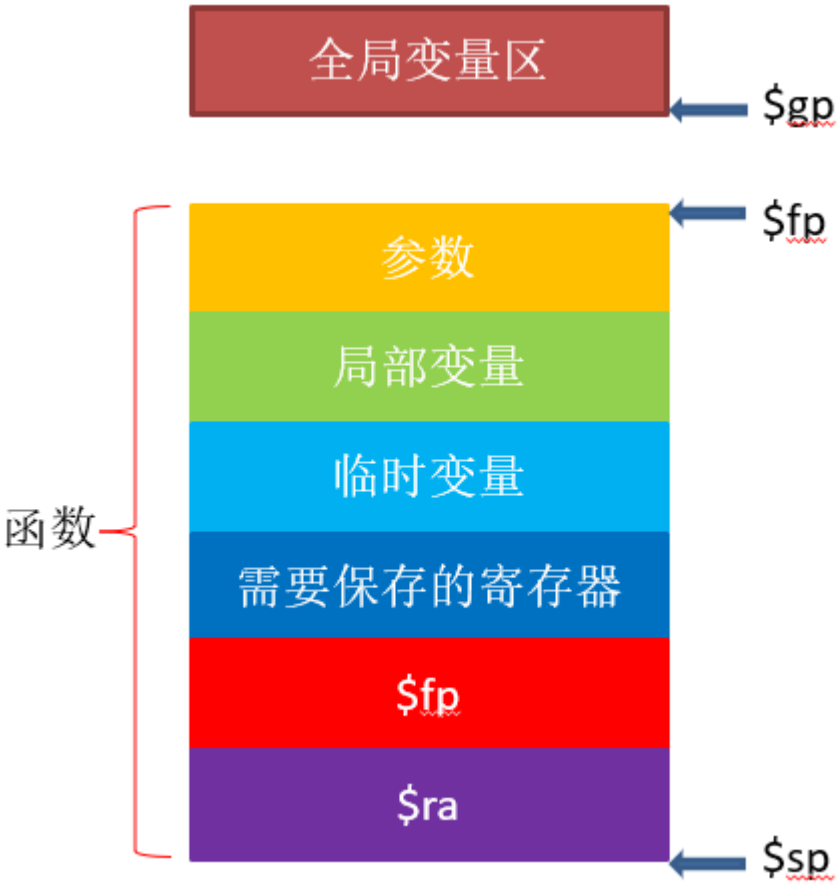
```
class midCode {  //z = x op y
public:
    operation op; // 操作
    string z;     // 结果
    string x;     // 左操作数
    string y;     // 右操作数
};
```

4.2 操作类别

操作	四元式	备注
PLUSOP	mc.z = mc.x + mc.y	+
MINUOP	mc.z = mc.x - mc.y	-
MULTOP	mc.z = mc.x * mc.y	*
DIVOP	mc.z = mc.x / mc.y	/
LSSOP	mc.z = (mc.x < mc.y)	<
LEQOP	mc.z = (mc.x <= mc.y)	<=
GREOP	mc.z = (mc.x > mc.y)	>
GEQOP	mc.z = (mc.x >= mc.y)	>=
EQLOP	mc.z = (mc.x == mc.y)	==
NEQOP	mc.z = (mc.x != mc.y)	!=
ASSIGNOP	mc.z = mc.x	=
GOTO	GOTO mc.z	无条件跳转
BZ	BZ mc.z (mc.x = 0)	不满足条件跳转
BNZ	BNZ mc.z (mc.x = 1)	满足条件跳转
PUSH	PUSH mc.z (mc.y)	函数调用时参数传递
CALL	CALL mc.z	函数调用
RET	RET mc.z	函数返回语句
RETVALUE	RETVALUE mc.z = mc.x	有返回值函数返回的结果
SACN	SCAN mc.z	读
PRINT	PRINT mc.z mc.x	写
LABEL	mc.z :	标号
FUNC	FUNC mc.z mc.x ()	函数定义
PARA	PARA mc.z mc.x	函数参数定义
GETARRAY	mc.z = mc.x [mc.y]	取数组值
PUTARRAY	mc.z [mc.x] = mc.y	给数组赋值
EXIT	EXIT	退出main
INLINEEND	INLINEEND mc.z mc.x	内联函数的结束标志
INLINERET	INLINERET mc.z	内联函数的返回值

（五）存储分配

5.1 存储分配示意图



5.2 存储分配

常量在优化后已经传播成数字了，不优化也可以直接查符号表得到；

字符串使用`.asciiz`存储在`.data`段；

全局变量通过`$gp`寄存器存取；

局部变量通过`$fp`寄存器存取；

进入`main`函数后，`$sp`直接到`main`的栈底，`$fp`在栈顶用于对局部变量的存取。每当调用函数时，`$sp`都在栈底，直接通过`$sp`进行参数传递(`PUSH`)，之后`$sp`变为被调用函数的栈底，将`$ra`、`$fp`存储到`$sp`偏移4和8的位置，`$fp`变为将函数被调用函数的栈顶，跳转需要保存的寄存器的值也通过`$sp`保存起来。这样跳转到新的函数后，就可以通过`$fp`进行参数和局部变量、临时变量的存取了。跳转返回后，通过`$sp`恢复`$ra`、`$fp`以及其他需要恢复的寄存器的值，通过`$fp`进行原函数的参数和局部变量、临时变量的存取。

（六）目标代码生成：`mipsCode.cpp mipsCode.h`

6.1 数据结构

```
class mipsCode {
    mipsOperation op; // 操作
    string z;     // 结果
    string x;     // 左操作数
    string y;     // 右操作数
    int imme;     // 立即数
};
```

6.2 mips指令

```
enum mipsOperation {
    add,
    addi,
    sub,
    mult,
    mul,
    divop,
    mflo,
    mfhi,
    sll,
    beq,
    bne,
    bgt, //扩展指令 相当于一条ALU类指令+一条branch指令
    bge, //扩展指令 相当于一条ALU类指令+一条branch指令
    blt, //扩展指令 相当于一条ALU类指令+一条branch指令
    ble, //扩展指令 相当于一条ALU类指令+一条branch指令
    blez,  //一条branch
    bgtz,  //一条branch
    bgez,  //一条branch
    bltz,  //一条branch
    j,
    jal,
    jr,
    lw,
    sw,
    syscall,
    li,
    la,
    moveop,
    dataSeg,  //.data
    textSeg,  //.text
    asciizSeg,  //.asciiz
    globlSeg,  //.globl
    label,  //产生标号
};
```

6.3 主要操作

```
void genMips();  //中间代码生成mips
void outputMipsCode();  //输出mips语句
void loadValue(string& name, string& regName, bool gene, int& va, bool& get, bool assign);  //取变量name的值到寄存器regName，后边的参数主要涉及到对常值的处理方式
void storeValue(string &name, string &regName);  //将regName的值存储到name的空间中
```

未优化版本：逐一针对中间代码翻译成`mips`代码，使用`$t0`,`$t1`,`$t2`作为运算的寄存器，对应中间代码的3个操作数。由于每个变量的地址是固定可以计算的，所以每次使用任何变量时，都直接从内存中用`lw`指令取出来；修改任何变量时，都用`sw`指令向内存中保存修改。

6.4 一个小坑点

`.global`**指令不能使用！！！**记忆犹新的是，这个导致我代码生成二作业改了一下午，交上去一直全错，结果是这个指令的问题，开头改成`j main`就可以了。

（七）代码优化

7.1 常数合并与传播

（1）**对表达式中出现的常数计算进行合并，在生成中间代码时优化**

优化前：b = 3 + 5 + a;

优化后：b = 8 + a;

（2）**常量在生成中间代码时，直接替换成数值**

优化前：const int a=10; b = 3 + 5 + a;

优化后：const int a=10; b = 18;

7.2 临时寄存器分配

7.2.1 基本原理

产生的所有的中间变量`#Ti`，都会**在第一次出现时被赋值，第二次出现时被使用掉**，即都只出现两次。根据这个规律，给所有的中间变量`#Ti`，分配t寄存器：**当它被赋值的时候，给它分配t寄存器，而当它被使用时，释放它占用的寄存器。**

7.2.2 数据结构

上文6.3中提到保留`$t0`,`$t1`,`$t2`三个寄存器用于进行表达式的运算。所以`$t3-$t9`用于临时寄存器分配。

使用以下数据结构记录各个寄存器是否被占用，以及它存储的是哪个中间变量。

```
int tRegBusy[10] = {0,};  //有t3-t9共7个临时寄存器供分配 用于记录临时寄存器是否被占用
string tRegContent[10];   //记录每一个临时寄存器分配给了哪一个中间变量 #T0,#T1...
```

使用以下两个函数用于查找是否有空闲的t寄存器和判断当前中间变量是否被分配了t寄存器

```
int findEmptyTReg();  //查找空闲的t寄存器
int findNameHaveTReg(string& name);  //判断当前中间变量name是否被分配了t寄存器
```

7.2.3 具体实现

t寄存器分配是在生成目标代码时完成的：当对某一个变量赋值时，如果发现它是中间变量`#Ti`，则调用`findEmptyTReg`方法，如果找到了空闲的寄存器`ti`，则给它分配寄存器`ti`，同时标记`tRegBusy`，`tRegContent`两个数组。当使用一个变量时，如果发现它是中间变量`#Ti`，则调用`findNameHaveTReg`方法，如果发现有某一个寄存器`ti`存储着`#Ti`，就直接使用寄存器`ti`生成目标代码，同时取消对`tRegBusy`，`tRegContent`两个数组的标志。如果寄存器不够了，那么只能通过内存进行存取了。

7.3 函数内联

对于比较"短"的函数可以进行内联，减少函数调用时对寄存器的存取等。**内联在生成中间代码的同时实现。**

7.3.1 内联的条件

（1）内联函数内部没有调用其他函数，因为嵌套调用、递归调用不方便处理。

（2）内联函数内部不能有跳转语句（BZ，BNZ，GOTO），因为这样可能导致内联函数有多个返回的出口，不方便处理。

（3）内联函数内部可以定义局部变量，但是不能过多，我设置了2个的限制。过多的局部变量，会使得调用方函数内部的变量过多，使得其s寄存器不够用，这样可能会导致负优化。

（4）内联函数不能修改全局变量。（原因见后文）

7.3.2 数据结构

```
map<string, vector<midCode> > funcMidCodeTable;  //每个函数的中间代码
map<string, bool> funcInlineAble;  //函数是否可以内联
```

每次处理到函数定义语句时，说明前一个函数定义结束了，扫描前一个函数的中间代码，调用`checkBeforeFunc`方法，判断此函数是否符合内联条件，将结果登记到`funcInlineAble`中。

7.3.3 内联的具体处理

当遇到函数调用语句时，判断当前函数是否可以内联。如果不可以，则只是生成函数调用中间代码`CALL funcName`即可。如果可以内联，调用`dealInlineFunc`进行处理，主要步骤为：

（1）因为内联后需要**将内联函数的变量名进行修改，修改成一定不会跟调用方函数重名的名字**，所以调用`fullNameMap`函数，实现内联函数内部的变量名到内联后新的变量名的对应。

（2）把原中间代码中函数**传参**`PUSH`**的过程改成赋值**`ASSIGNOP`的过程，新的中间代码插入到调用方函数的中间代码中。

（3）把原中间代码中其他内容的变量名修改为新的名字，新的中间代码插入到调用方函数的中间代码中。

（4）上述(3)中，对返回语句`RET`要特别处理，如果是无返回值函数的`RET`则直接忽略掉，如果是有返回值函数的`RET`改为`INLINERET`供后续处理。

（5）把新的名字插入到调用方函数的符号表中。

```
void fullNameMap(map<string, string>& nameMap, vector<midCode> ve, string funcName);
void dealInlineFunc(string name);
```

7.3.4 难点分析

（1） 对内联函数返回值的处理

正常的调用有返回值函数时，返回值`value`会被存储到`$v0`中，返回值做因子时，会生成一条`RETURNVALUE op1 = RET`，表示将函数返回值从`$v0`传入到`op1`中。

对于内联的有返回值函数，可以仿照上述提到会生成中间代码`INLINERET value`，这时直接将`value`的值传递给`op1`即可，省略了对`$v0`的赋值。

（2）修改了全局变量的函数不能内联

因为实现内联时，需要把中间代码复制到原函数调用的位置，同时**需要对变量名进行修改，而对于全局变量名是不能修改的**，一旦修改会跟使用该全局变量的其他位置的代码出现数据不一致，但是如果调用函数定义了跟此全局变量**同名的局部变量**，在使用此全局变量时会把它当成局部变量使用，就会出错。例如：

```
int a;
void f() {
  a = 1;
  a = a + 1;
}
void main() {
  int a;  //局部变量跟全局变量同名，如果f内联过来，就会出错。
  a = 0;
  f();
  a = a + 1;
  printf(a);
}
```

(3) 函数内联导致原函数寄存器不够用问题

函数内联后，内联函数里边的临时变量、局部变量都加入到了调用方函数中，导致调用方函数的变量增多，而一旦过多，就会导致寄存器不够分配。尤其对于s寄存器，对于一个基本块内，多次内联函数，内联函数的局部变量占用的s寄存器没法得到及时的释放。

解决方案：根据上文知道，内联函数的中间代码需要改名字，而产生新名字的方法是固定的：

```
string genName() {
    nameId++;
    return "%INLINE_" + int2string(nameId); //%开头 跟正常的变量区分开
}
```

所以可以记录内联前后`nameId`的范围，在这个范围内的就是这个内联函数用到的局部变量，这样在内联函数结尾处，增加一个中间代码标识，当遇到这条标识的时候，尽可能的释放`nameId`范围内的局部变量所占用的寄存器即可（即便这是在一个基本块的内部，因为一个内联函数里用到的局部变量在后边是绝对不会再次用到的，只在这个内联函数内部是有效的）。

7.4 基本块划分与活跃变量分析：`optimize.cpp optimize.h`

7.4.1 基本块划分原则

对于每一个函数，基本块划分原则：

（1）函数的第一条语句属于基本块的入口（`FUNC`）

（2）能够跳转到的语句属于基本块入口（`LABEL`）

（3）跳转语句下一条语句输入基本块入口（跳转语句包括`BZ`,`BNZ`,`GOTO`,`RET`,`EXIT`）

7.4.2 数据结构

```
class Block {
    int start;  //中间代码开始的序号
    int end;    //中间代码结束的序号
    int nextBlock1;  //后继1
    int nextBlock2;  //后继2
    vector<midCode> midCodeVector;  //块内的中间代码
    vector<string> use;
    vector<string> def;
    vector<string> in;
    vector<string> out;
}
map<string, vector<Block> > funcBlockTable;   //每个函数的基本块列表
```

7.4.3 活跃变量分析

划分块的同时计算每个块的use,def集合，全部划分之后采用书中的算法，从后向前计算每一个块的out,in集合。out = 所有后继的in的并集；in = use ∪ (out - def)；直到所有的in不再变化。

```
void calUseDef(Block& bl, string funcName);
void calInOut();  //计算in,out
```

至此，就求出每个基本块的use,def,in,out集合。

7.4.4 难点分析

（1）**基本块数据结构的构造**

在当前文法下，每个块最多两个后继。`BZ`和`BNZ`有两个后继，`GOTO`**只有一个后继**，`RET`没有后继。所以可以记录每一个块的两个后继分别是什么，这样就形成了一个基本块间的图了。

7.5 全局寄存器分配

7.5.1 数据结构

全局寄存器用于存储函数的局部变量，在函数调用时需要对使用的全局寄存器进行保存，函数调用返回后再把值取出来。

使用以下数据结构记录各个寄存器是否被占用，以及它存储的是哪个局部变量。

```
int sRegBusy[10] = {0,};  //有s0-s7共8个全局寄存器供分配 用于记录全局寄存器是否被占用
string sRegContent[10];   //记录每一个全局寄存器分配给了哪一个局部变量
```

使用以下两个函数用于查找是否有空闲的t寄存器和判断当前中间变量是否被分配了s寄存器

```
int findEmptySReg();  //查找空闲的s寄存器
int findNameHaveSReg(string& name);  //判断当前中间变量name是否被分配了s寄存器
```

7.5.2 全局寄存器分配的基本原则

s寄存器分配是在生成目标代码时完成的：局部变量跟中间变量不同，不再满足第一次出现时被赋值，第二次出现时被使用掉的规则，所以每当出现局部变量，不管是被使用还是被赋值，都要检查它是否被分配过寄存器，调用`findNameHaveTReg`，如果被分配过，就直接使用寄存器的值，但是并不取消对`sRegBusy`，`sRegContent`两个数组的标记；如果没有被分配过就调用`findEmptyTReg`，尝试着去分配寄存器，分配成功则需要标记`sRegBusy`，`sRegContent`两个数组。如果寄存器不够了，那么只能通过内存进行存取了。

7.5.3 全局寄存器释放的基本原则

（1）跨函数时，s寄存器全部释放。

（2）函数内部，跨基本块时，根据基本块的in集合，集合内的元素代表此变量是活跃的，不在集合内的变量说明它是不活跃的，可以释放全局寄存器。

7.5.4 难点分析

s寄存器的释放——结合活跃变量分析

函数内部，跨基本块时，根据基本块的in集合，集合内的元素代表此变量是活跃的，不在集合内的变量说明它是不活跃的，可以释放全局寄存器。

上述方案存在的一些问题：

比如，对于条件语句`if-else`，每次在执行时只会进入到其中一个分支，但是在数据流分析时，进入到`if-else`的代码后继有两个，在下边例子中，a会被分配`$s0`寄存器，b被分配`$s1`寄存器，在进入第6行的块之前，就会把a,b的寄存器都释放掉，后边又给b分配寄存器`$s0`，这就会导致输出的b的值实际上是a的值，导致了错误。同样的，对于`while`和`for`循环也有类似问题。

`if-else`的例子：

```
1  void ccc() {
2      int a,b,c,d,e,r,t,y,u,i,o,p;
3      a=-3;
4      b=1;
5      if (a>0) {
6          b=1;c=1;d=1;e=1;r=1;t=1;y=1;u=1;i=1;o=1;p=1;
7      }
8      else {
9          a=1;
10     }
11     printf(b);
12 }
```

`for`循环的例子：

```
1  void cc(int a1,int a2,int a3,int a4,int a5,int a6,int a7,int a8,int a9) {
2      int i;
3      for (i=0;i<10;i=i+1) {
4          a1=a9;
5          if (a1>0) {
6              a2=2;
7          }
8          else {
9              a2=1;
```

```
        }
        a9=10;
        printf(a2);
    }
}
```

`while`循环的例子：

```
void c() {
    int a,b,c,d,e,r,t,y,u,i,o,p;
    a=-10;b=2;
    while(a>0) {
        b=1;c=1;d=1;e=1;r=1;t=1;y=1;u=1;i=1;o=1;p=1;
        a=-1;
    }
    a=1;
    printf(b);
}
```

解决方案：

根据条件语句和循环语句的特点，中间代码会以某`LABEL`开头，某`LABEL`结束，这样就可以设置标志，当遇到第一个`LABEL`时，关闭释放s寄存器的开关，不再释放s寄存器，直到遇到后一个**匹配**的`LABEL`时，打开释放s寄存器的开关，可以正常释放s寄存器。这样就实现了全局寄存器的释放。

7.6 其他优化

主要针对ALU，涉及到指令选择、跳转优化、循环优化等。

（1）四则运算指令，如果两个操作数都是数字，直接运算将结果赋给结果操作数，不需要取到寄存器中。

（2）加减法时，一个常数，一个寄存器，则直接使用addi，省略一步取寄存器。

（3）乘除法时，注意x*1=x，1*x=1，0/x=0，x/1=x，可以省略取寄存器和`mflo`。

（4）比较运算符，如果两个操作数都是数字，直接比较，然后无条件跳转`j`。

（5）上述运算时，使用$0寄存器表示常数0，节约一步取寄存器。

（6）取数组值和给数组赋值时，数组下标如果是常数，直接计算出正确的地址去取值即可。

（7）`#T1=#T2+#T3`；`a=#T1`；这种比较简单的赋值直接合并为`a=#T2+#T3`；在生成中间代码时处理。

（8）来自讨论区的方法，修改了`for`和`while`循环中间代码顺序，减少了`jump`。

（9）**无用代码删除**，对于无用的赋值语句，直接删除。例如在本次竞速排序中，`x1 = (j/i) * i;`但是`x1`并没有在其他地方使用到，所以可以优化掉。

三、总结反思

一学期下来，终于完成了这个"简单的"编译器，文法已经简化了很多很多，但是实现起来还是各种困难，体会到了写编译器的难度。总结下来，这一学期，首先直观的感受前期（期中考试、错误处理之前），节奏要比后期慢，后半学期的节奏和难度明显增大。针对优化，优化可以说是做多错多，但是优化更重要的是保证正确性为前提的，优化首先要做好设计，中间代码跟目标代码不是割裂的，其次要注意细节，同时需要大量测试，一次优化可能带来很多在你考

虑之外的问题，通过大量测试才能保证正确性。通过编译器代码的编写，我学会了c++的使用，同时使用git做版本控制，我认为这对一个大项目有很大帮助。同时，课程的截止日期的固定的，做好时间规划就很重要，坚决不能拖延！充足的时间才能保证代码的质量。这学期引入了自动化测评，希望以后的课设越做越好。