1.词法分析

1.1整体流程

->从文件中获取一个合法字符递给词法分析程序

->词法分析状态机程序改变

->词法分析以词为单位向外提供结果

1.2程序内容

1.2.1文件组织

文件名	作用
main.cpp	主函数
lexical_analysis.h	词法分析头文件
lexical_analysis.cpp	词法分析文件

1.2.2主要常量

主要常量即为两个hashmap，保存单词内容向单词类型的映射关系

常量	作用
static map<string,string> KeyWord	关键字的有关映射
static map<string,string> Delimiter	分隔符的有关映射

1.2.3主要变量

变量	作用
static ifstream *inFile	指向要打开文件对应的文件流
static char c	预读字符
static string lexical_str	词法分析的结果之一，表示词内容。
static string lexical_type	词法分析的结果之一，表示词类型

1.2.4主要函数

函数	作用
char file_getChar()	每次从文件中获取一个字符
void skip_blank()	筛选从文件中不合法字符
bool belongTo_Keyword(string word);	判断一个字符串是否属于关键字
bool belongTo_Delimiter(string word);	判断一个字符串是否属于分隔符
pair<string, string> get_a_word();	核心函数 根据读入字符进行分支判断并最终返回一个合法的词的内容与类型
void wrong_handle();	错误处理函数

1.2.5核心函数(get_a_word)设计

1.全局变量初始化——将c、 lexical_str、lexical_type进行清空处理。

2.初状态选择——在全局变量为空的状态下，根据读入的第一个字符选择主要分支

第一个字符内容	分支方向
'\''	字符常量
'\"'	字符串常量
['0'~'9']	整数
['_''a'~'z']	标识符或关键字
其它属于分隔符的字符	分隔符

3.根据接下来字符继续选择分支或者进行正确性判断

4.返回这次分析的词的值与类型

2.语法分析

2.1流程

1. 整体流程

   ->语法词法分析器初始化

   ->语法分析器自顶向下分析文法，每次向词法分析器发出获得一个词的请求

   ->词法分析器接受请求开始运行，并返回一个次的结果

   ->语法分析器继续运行

2. 语法分析器流程

   自顶向下从程序开始分析语法

   ->从当前程序进入到更小的子程序之前，预读一个符号

   ->将符号与子程序的第一个词相匹配，决定分支（若一个词无法决定，则继续预读，直至决定分支方向）

->进入子程序，打印之前的预读符号；子程序运行到最后，打印当前子程序信息，并预读一个符号后，退出当前子程序。

2.2主要内容设计

1. 全局变量

变量	作用
static string SYM;	保存当前词法分析器返回词的词内容
static string TYPE;	保存当前词法分析器返回词的词类型
static queue<pair<string,string>> BACKSTRACK;	当预读一个词无法决定分支时，需要将预读词存入该全局队列；子程序中需从该队列中取出之前预读的词
static map<string, string> FUNC_TYPE;	函数声明时保存函数名与函数类型的对应关系。用于语句调用时判断是有返回值函数调用还是无返回值函数调用

2. 宏

宏名	作用
PRE_READ_NEW_WORD_FROM_LEX	从语法分析器中预读一个词
PRE_READ_NEW_WORD_FROM_ALL	若预读队列里有词，则从队列弹出队首词；若没有，则从词法分析器中预读一个词
READ_NEW_WORD	若队列中有词，则读取队首词并打印；若没有，则从词法分析器中读一个词并打印
PRINT_PRE_WORD	打印所预读的词
CALL类宏定义	判断是否满足进入某个子程序的条件

注：预读与非预读的区别是前者不能打印当前词，后者需要打印当前词

3. 语法分析函数

函数名	作用
Program	'程序'程序
Constant_description	'常量说明'子程序
Variable_description	'变量说明'子程序
Constant_define	'常量定义'子程序...
... ...	

一个语法分析函数运行的例子（以<常量说明为例>）

```
//<constant_description>--------------------------------------------
void constant_description() {//FIRST={CONST}
    PRINT_PRE_NEW_WORD   //打印进入该子程序前预读的词
```

```
    PRE_READ_NEW_WORD_FROM_LEX//从词法分析器预读一个词
    CALL_Constant_define //判断预读词是否满足进入常量定义子程序条件，满足则进入，不满足则报错
    if (TYPE == "SEMICN") PRINT_PRE_WORD else error();//从<常量定义>子程序退出后，判断预读的词是否满足接下来的文法，满足则打印预读词汇，不满足则报错
    PRE_READ_NEW_WORD_FROM_LEX//预读一个词汇（在不知道什么时候会退出的时候都要预读词汇）
    while (TYPE == "CONSTTK") {//判断是否继续在当前文法是则进入
        PRINT_PRE_WORD  //打印之前用于判断的预读词汇
        PRE_READ_NEW_WORD_FROM_LEX //预读
        CALL_Constant_define    //调用常量定义
        if (TYPE == "SEMICN") PRINT_PRE_WORD else error(); //判断文法
        PRE_READ_NEW_WORD_FROM_LEX //预读
    }

#ifdef TURN_ON
    file_out << "<常量说明>" << endl;  //打印
#endif
}
```

3.错误处理

3.1概述

本次错误处理的实现主要从两大方面考虑。

一是符号表的设计，因为错误类型中“非法符号或不合法词法”、“名字重定义”、“未定义的名字”、“函数参数个数不匹配”、“函数参数类型不匹配”、“条件判断中出现不合法的类型”、“数组下标只能使整型”、“不改变常量值”、“数组初始化个数不匹配”、“常量类型不一致”，这些方面都需要用到符号表的建表操作或者查表操作，因此符号表的设计占比较重要的成分。

二是在递归下降程序中插入错误处理处理语句，从而进行对错误处理条件的判断以及输出相应错误信息并进行错误处理（现阶段错误处理只是跳过导致错误的词）

3.2符号表的设计

1. 符号表结构

   分析需求后发现，我们的符号表设计需要满足：

   - 提供一个能够存储不同层级的符号表容器

```
map<int/**level**/, symbolTable/**符号表类**/> Tables;
```

   - 符号表中能够存储不同类型的符号

```
class symbolTable {
private:
    //there are many different kind map containers
    map<string, constant_line> constantLines;
    map<string, variable_line> variableLines;
    map<string, function_line> functionLines;
public:
    //some methods here...
    ...//略
};

```

- 符号表向外提供增删查方法

```
class symbolTable {
private:
    map<string, constant_line> constantLines;
    map<string, variable_line> variableLines;
    map<string, function_line> functionLines;
public:
    //insert
    void insert_constantLine(symbolLine &line) {...//略}

    void insert_variableLine(symbolLine &line) {...//略}

    void insert_functionLine(symbolLine &line) {...//略}

    //delete
    ...//略
    //judge if has
    bool has_Line(string name) {...//略}

    //get (needs check first)
    symbolLine &get_Line(string name) {...//略}
};
```

2. 符号的分类

因为不同符号可能要存入不同的特性信息，因此主要将要存入符号表的符号分为以下三类：

常量类`constant_line`、变量类`variable_line`、函数类`function_line`

这三个类都继承自一个公共父类，包含了这三个类所需的公共信息：

```
class symbolLine {
private:
    string name;//名称信息
    int kind;//种类信息，包括 K_CONSTANT, K_VARIABLE, K_FUNCTION,
    int type;//类型信息，包括 T_CHAR,T_INT
    ...//略
};
```

当然他们也含有自己所需要的特性信息：

```
class constant_line : public symbolLine {
private:
    //无特性信息
};
```

```
class variable_line : public symbolLine {
private:
    int dimension;//说明维度，普通常量为0，一维数组为1，二维数组为2
    int dimension1;//一维数组以及二维数组的一维分量个数
    int dimension2;//二维数组的二维分量个数
public:
    ...//略
};
```

```
class function_line : public symbolLine {
private:
    bool has_return;//是否含有返回值，区分void函数与有返回值函数
    vector<string> parameter_type;//参数类型表
    vector<string> parameter_name;//参数名称表
public:
    ...//略
};
```

在符号表中的使用时，传入引用`&symboline`并使用静态类型转变方法`dynamic_cast`实现父类转子类。

3. 符号表的使用方法

通过封装设计一些对符号表的操作方法，以便于文法的递归下降程序中去进行对符号表的调用。这些方法主要包括“建立符号表”、“删除符号表”、“填表”、“查表”、“获取符号”操作。

- 建表与删表

```
void create_table(int tableLevel/**输入符号表层级**/){
 Tables.insert(pair<int, symbolTable>(tableLevel, symbolTable()));//为对应层级建立一个新的空符号表
}
 void quit_table(int tableLevel) {
     try {
         Tables.erase(tableLevel);//删除指定层级的符号表
     } catch (exception e) {
         cout << "drop unexisted table!";
     }
 }


```

4. 填表

```
void set_table_symbol(int tableLevel/**待填入层级**/, int kind/**符号类型**/, symbolLine &line) {
 //检查是否跨表级填表，例如当前只有层级为0的符号表，要填入层级为2的符号表
 int checkLevel = tableLevel - 1;
 while (checkLevel >= 0) {
     if (Tables.find(checkLevel) == Tables.end()) {
         cout << "skip tables!";
     }
     checkLevel--;
 }

 //检查待填入的符号表是否存在，若不存在新建这个层级的符号表
 if (Tables.find(tableLevel) == Tables.end()) {
      create_table(tablelevel);
 }

 //检查当前层级符号表是否含有同名符号
 if (Tables.at(tableLevel).has_Line(lower_word(line.name))) {
     file_error << lastSymbolNum << " b" << endl;//含有同名符号报错
 } else {
     //不含有同名符号，根据kind类型填入符号表
     line.name = lower_word(line.name);
```

```
    switch (kind) {
        case K_CONSTANT://常量类型符号
            Tables.at(tableLevel).insert_constantLine(line);
            break;
        case K_VARIABLE://变量类型符号
            Tables.at(tableLevel).insert_variableLine(line);
            break;
        case K_FUNCTION://函数类型符号
            Tables.at(tableLevel).insert_functionLine(line);
        default:;
    }

 }
}

```

5. 查表

```
bool check_table_symbol(string name) {
 //从当前层级依次向下查找符号表
 int searchLevel = nowLevel;
 while (searchLevel >= 0) {
     if ...//判断条件 {
         return true;
     }
     searchLevel--;
 }

 //遍历所有符号表均未找到
 if (searchLevel < 0) {
     file_error << lastSymbolNum << " c" << endl;
 }
 return false;
}

```

6. 获取符号

```
symbolLine & get_table_symbol(string name) {
    if(check_table_symbol(string name)){//获取前先查找是否存在符号
         ...//若存在则获取符号
    }else{
        file_error<<lastSymbolNum<<"c"<<endl;//不存在输出错误信息
    }
}

```

3.3错误处理语句设计

3.3.1非法符号或不合法词法

直接在词法分析器中判断，若读到不合法的字符进行报错处理。需要注意的是词法分析器判断“字符”与“字符串”过程中即使遇到错误符号也要继续读至`'`与`"`,不然会影响后续的词法判断。

3.3.2名字重定义

在常量声明、变量声明与函数声明中，新建一个`symboline`子类对象，在文法分析的不同地方填入不同的信息，一个例子如下：

```
variable_line line;
line.kind = K_VARIABLE;//填入种类信息

if (TYPE == "INTTK")line.type = T_INT;//填入类型信息
else line.type = T_CHAR;

//...
line.name = SYM;//填入名称信息
//...
if (TYPE == "LBRACK") {
            line.dimension = 1;//填入特性信息
//...
set_table_symbol(nowLevel, line.kind, line);//填入符号表

```

3.3.3未定义的名字

只有在以下几种场合会引用标识符：

- 循环语句中`<标识符>=<表达式>;`与`<标识符>=<标识符>(+|-)<步长>')'`
- 有/无返回值调用函数语句`<标识符>'('<值参数表>')'`
- 赋值语句`<标识符>= ...`
- 读语句`scanf '('<标识符>')'`
- 因子

在引用的过程中增加查表操作，即`check_symbol_table(nowlevel)`即可

3.3.4函数参数类型不匹配/函数参数个数类型不匹配

在有/无返回值函数调用语句中，符号表中取出对应名字的符号，然后对值参数表进行比对判断即可

```
//从符号表中取出对应符号
function_line line;
if (check_table_symbol(SYM)) {
    line = dynamic_cast<function_line &>(get_table_symbol(SYM));
}
...
//到值参数表步骤时，判断值参数表表达式个数与类型是否与上述符号内存储的参数类型和个数一致
while (...) {
        parameter_position++;
        //值参数类型与符号表中参数类型不一致
        if (line.parameter_type.at(parameter_position)!=expressionValue) {
        file_error << lastSymbolNum << " e" << endl;
}

//判断个数是否一致
if (parameter_position != line.parameter_type.size()) {
    file_error << lastSymbolNum << " d" << endl;
}

```

3.3.5无返回值的函数存在不匹配的return语句 / 有返回值的函数缺少return语句或存在不匹配的return语句

在`return`语句中增加对当前函数类型与return返回类型的匹配判断：

```
if (/**return返回值类型为char**/) {//char{
                if (/**当前函数类型为void**/)//报错
                if(/**当前函数类型为int**/)//报错
} else if(/**return返回值类型为int**/){
     if (/**当前函数类型为char**/)//报错
     if(/**当前函数类型为void**/)//报错
}else {//return无返回值
    if(/**当前函数类型为int**/)//报错
    if (/**当前函数类型为char**/)//报错
}

```

3.3.6数组下标只能使整型表达式

"因子"与"赋值语句"的递归子程序中有可能引用数组元素，在这些地方读到数组下标时：

```
//check array subscript
if (expressionValue == CHAR/**表达式类型为char**/) {
    file_error << lastSymbolNum << " i" << endl;
}

```

3.3.7.不能改变常量的值

"for语句"、"赋值语句"、"读语句"这三个地方可能会对标识符进行赋值，因此增加对标识符种类的判断即可：

```
temp = get_table_symbol(SYM);
if (temp.kind == K_CONSTANT) {
     file_error << lastSymbolNum << " j" << endl;
}

```

3.3.8.应为分号、应为右小括号、应为右中括号

在可能出现这些符号的地方增加是否缺失相应符号的判断即可。

```
/**以分号为例**/
if (TYPE == "SEMICN") {
     //这里需要注意，只有在读到分号的时候才需要预读，不是分号的情况相当于已经预读过了
     PRE_READ_NEW_WORD
} else {
    //不是分号输出错误信息
     file_error << lastSymbolNum << " k" << endl;
}


```

3.3.9. 数组初始化个数不匹配

"变量定义及初始化"中，对变量符号的`variable_line.dimension`、`variable_line.dimension1`、`variable_line.dimension2`进行是否一致判断即可。

3.3.10.常量类型不一致

判断常量与符号的类型是否一致即可：

```
if (symboline.type != constant_value_type)file_error << lastSymbolNum << " o"
<< endl;


```

3.3.11.缺少缺省语句

swicth语句中检查是否含有缺省语句即可。

4.代码生成

4.1生成中间代码

在语法分析的基础上，生成四元式形式的中间代码。

4.1.1四元式结构

四元式：

```
class fourOp{
    int op;//表示四元式类型，会在接下来的'中间代码格式'中展示
    //表示四元式的三个操作数
    operand item1;
    operand item2;
    operand item3;
}


```

操作数：

```
class operand {
    int kind;//表示操作数的种类，包括constant,variable,function,string四类

    int value;//在kind为constant的前提下，表示常数的值
    string stringValue;//在kind为string的前提下，表示字符串常量的值
    int valueType;//在kind为variable的前提下，表示变量的类型，包括int与char与
string

    string label;//若该操作数所属四元式类型为"put label"，则存储要设置的标签值

    //在kind为variable的前提下
    int address;//表示变量所存的地址
    bool direct_address;//表示所存地址是直接地址还是间接地址
    int addressBaseType;//表示所存地址的基地址是哪些，包括‘局部变量基地址’，‘全局变量
基地址’，‘临时变量基地址’
};

```

4.1.2中间代码格式

其中，{}中的内容表示生成操作数时必须要给定值的变量。

op	item1	item2	item3
VAR	address: addressbasetype:	value: 0	\
PLUS/MINU /MULT/DIV	address:	{kind,direct_address, addressBaseType,value}	{kind,direct_address, addressBaseType,value}
ASSIGN	{direct_address, addressBaseType}	{kind,direct_address, addressBaseType,value}	\
bgt/bge/blt/ ble/beq/bne	{kind,direct_address, addressBaseType,value}	{kind,direct_address, addressBaseType,value}	\
j	{label}	\	\
putlabel	{label}	\	\
SCANF	{valueType,addressBaseType}	\	\
PRINTF	\	{kind,valueType,value, direct_address,addressBaseType}	\
SaveBackValue	{kind,direct_address, addressBaseType,value}	\	\
EXIT	\	\	\
back	\	\	\
addSpace	{size}	\	\
jumpToFunction	{label}	\	\
giveparameters	{value}	{kind,direct_address, addressBaseType,value}	\
getReturnValue	address	\	\
SaveBackRA	\	\	\

4.1.3一些格式约定

- 表达式的计算结果一定保存在临时变量内存中。也就是即使该表达式仅有一个局部变量标识符组成的情况下，也会将该标识符的值存储在一块临时内存中，作为表达式结果的地址。
- 进行函数调用的时候，需要做以下步骤：

  调用前：

  1. 对当前块所需的内存大小做出统计并记录，得到函数调用时基地址需要增加的大小，以及函数调用后的基地址值，假设为k。
  2. 将上述所要增长内存大小存储在 (k+0)的地方，以便于函数退出时将基地址回退只函数调用处。
  3. 将要传递的参数从（k+8）的地方开始存储
  4. 将基地址增加到函数调用后的基地址，即k
  5. 调用jal指令

  调用后：

  1. 要将ra中存储的地址存到（k+4）的地方，以便于从当前函数退出

  返回：

  1. 存储函数返回值，约定返回值存储在a0
  2. 从(k+0)处取出基地址要减小的空间，从（K+4）处取出要跳回的指令地址，并进行回退。

4.2目标代码生成

完成四元式到目标代码的翻译工作。因为由语法分析到中间代码做的事情比较多，所以中间代码到目标代码的翻译过程就比较简单与机械。所以接下来仅展示几个例子：

```
//op为var
if (four.op == O_VAR) {
        if (four.item2.kind == K_CONSTANT) {
            mips_file << "li $s2, " << four.item2.value << endl;
            if (four.item1.addressBaseType == A_GP) {
                mips_file << "sw $s2, " << four.item1.address << "($gp)" << endl;
            } else if (four.item1.addressBaseType == A_VAR) {
                mips_file << "sw $s2, " << four.item1.address << "($k0)" << endl;
            }

        }
    }

```

```
//op为assign
if (four.op == O_ASSIGN) {
        if (four.item2.kind == K_CONSTANT) {
            mips_file << "li $s2, " << four.item2.value << endl;
        } else if (four.item2.kind == K_VARIABLE) {
            if (four.item2.direct_address) {
                ......
            } else {
               ......
            }

        }

        if (four.item1.direct_address) {
           ......
        } else {
           ......
        }
    }

```

```
//op为j
if (four.op == O_j) {
        mips_file << "j " + four.item1.label << endl;
}

```

```
//op为putlabel
if (four.op == O_putlabel) {
        mips_file << four.item1.label + ": " << four.notation << endl;
    }


```

。。。。。。

5.代码优化

5.1 全局寄存器的分配

1.分基本块并添加控制流

要完成优化，首先要做的就是分基本块。在我的中间代码中，以下四元式是可能会**划分基本块**的：

第一类**表示，截止到该句且不包含该句，以上内容可以分为一块，**从该句开始要划分新块。包括$O_putlabel$

第二类表示，截止到该句且包含该句，以上内容可以分为一块，从下一句开始要划分新块。判断代码如下：

```
bool belong_to_branch(int op) {
    if (op == O_bgt || op == O_bge || op == O_blt ||
        op == O_ble || op == O_beq || op == O_bne ||
        op == O_j )
        return true;
    else return false;
}
```

2.活跃变量分析与构建冲突图

def与use的计算：

```
    /***
     * 1.def与use
     * ①def与use概念
     * def：基本块内定义先于使用的变量
     * use：基本块内使用先于定义的变量
     * ②可能出现def的四元组
     * var,+,-,*,/,=,scanf-------item1.address
     * ③可能出现use的四元组(先忽略函数调用)
     * var,+,-,*,/,=,b,print
     *
     */
```

in与out的计算：

```
    /***
     * 2.in与out
     * in = use并(out-def)；out = in的并
     * 循环直到in与out均不再变化
     */
```

由于上述的变量分析单位为基本块，我们可以采用按四元式为单位的方法去对每一个四元式进行活跃变量分析，从而提高分析精度，方法如下：

```
/***
 * 每一个四元式引入:
 * live,def,use
 * 其中live表示在该四元式之后仍活跃的变量，def与use的含义大致等同于基本块中的含义。
 *
 * 计算方式如下:
 * ①若四元式是基本块的最后一个四元式，令 live=out[B];反之，live[i]=live[i+1];
 * ②为def中每一个变量与live中每一个变量添加到冲突图中，并添加冲突边
 * ③更新live=useU(live-def)
 * ④转至①
 */
```

3.染色

构建好冲突图后，就可以按照理论课所讲的启发式算法来为冲突图染色了。染色分为两步：

1. **启发式算法从冲突图中删点**：点分为两类，若所连边数小于颜色个数，则为满足染色条件的点，否则为不满足染色条件的点。若满足颜色条件，则可从图中删去，进入结点栈；否则需要将点标记为不需要染色，再从图中删去，不进入结点栈。

```
//2.移除结点直至剩一个结点
    while (dynamic_graph.size() > 1) {
        bool flag = false;//判断是否有冲突边小于sRegNum的结点

        //①寻找所有冲突边小于sRegNum的结点
        for (map<pair<int, int>, set<pair<int, int>>>::iterator iter = dynamic_graph.begin();
             iter != dynamic_graph.end(); iter++) {
            //找到了
            if ((*iter).second.size() < sRegNum) {
                //删除与该点连接的其它点的这条边
                for (set<pair<int, int>>::iterator conflict_set = (*iter).second.begin();
                     conflict_set != (*iter).second.end(); conflict_set++) {
                    map<pair<int, int>, set<pair<int, int>>>::iterator other_point;
                    other_point = dynamic_graph.find((*conflict_set));
                    if (other_point != dynamic_graph.end()) {
                        //找到并删除
                        (*other_point).second.erase((*iter).first);
                    }
                }

                //将变量加入待染色栈
                uncolored_stack.push((*iter).first);

                //将自己从图中删除
                dynamic_graph.erase(iter);
                flag = true;
                break;
            }
        }

        //②没有小于sRegNum的结点，则随便挑选一个删除
        if (flag == false) {
```

```
            map<pair<int, int>, set<pair<int, int>>>::iterator
toDeletePoint = dynamic_graph.begin();
            //删除与该点连接的其它点的这条边
            for (set<pair<int, int>>::iterator other_point =
(*toDeletePoint).second.begin();
                 other_point != (*toDeletePoint).second.end();
other_point++) {
                //对于每一个连接待删除点的其它点，从他们的set中删除该点
                map<pair<int, int>, set<pair<int, int>>>::iterator
other_point_graph = dynamic_graph.find(*other_point);
                if (other_point_graph != dynamic_graph.end()) {
                    //找到了其它点1

(*other_point_graph).second.erase((*toDeletePoint).first);
                }
            }

            //大于sRegNum的不进行染色处理，不加入待染色栈

            //将自己从图中删除
            dynamic_graph.erase(toDeletePoint);
        }
    }
```

2. **根据结点栈逆向添加删去的点，并为加入的点染色**：由于结点栈中只有需要被染色的点，所以每个加入的点都需要被染色。

```
//3.进行染色分配处理
    //分配剩余的一个颜色,要求s_pool必须大于1
    pair<int, int> last_node = (*dynamic_graph.begin()).first;
    var_color.erase(last_node);
    var_color.insert(pair<pair<int, int>, int>(last_node, s0));
    //将待染色栈的结点依次取出并染色
    while (!uncolored_stack.empty()) {
        //①从栈中去除该颜色
        pair<int, int> uncolored_node = uncolored_stack.top();
        uncolored_stack.pop();

        //②构造一个表来判断有哪些颜色还可以用
        map<int, bool> color_can_use;
        for (int i = 0; i < sRegNum; i++)color_can_use.insert(pair<int,
bool>(i, true));

        //③找该点连接的其它点，将他们的颜色设置为不可用
        set<pair<int, int>> conflict_set =
conflict_graph.at(uncolored_node);
        for (set<pair<int, int>>::iterator other_point =
conflict_set.begin();
             other_point != conflict_set.end(); other_point++) {
            int other_point_color = var_color.at(*other_point);
            if (other_point_color != -1) {
                //若其它颜色被染过色,则令该颜色不可用
                color_can_use.erase(other_point_color);
                color_can_use.insert(pair<int, bool>(other_point_color,
false));
```

```
            }
        }

        //⑥挑选数值最小的可用颜色给该结点
        for (int i = 0; i < sRegNum; i++) {//若没有可用颜色则不分配
            if (color_can_use.at(i) == true) {
                var_color.erase(uncolored_node);
                var_color.insert(pair<pair<int, int>, int>
(uncolored_node, i));
                break;
            }
        }
    }
```

4.分配$t及$s寄存器

在染好色之后，就可以分配**$s寄存器**了。寄存器池如下：

```
enum sRegPool{
    s0,s1,s2,s3,s4,s5,s6,s7,
    s_end
};
```

5.2临时寄存器的分配

我代码中临时寄存器的作用是用来保存表达式计算过程中产生的结果。并且由于中间结果通常只会被使用一次，即参与下一次运算，因此中间结果只有两种状态——“产生”与“消除”。

故提供两组映射关系：

```
map<temp_var, reg_num> var_temp;
map<reg_num, use> temp_use_case;
```

其中`var_temp`表示当前的临时变量产生式所对应的是哪个临时寄存器，以便于在下次使用该临时寄存器时找到对应的寄存器编号；`temp_use_case`表示当前有哪些临时寄存器是空余的，以便在产生临时结果时判断是否有空闲寄存器保存该临时结果。

我的中间代码中以下列出的四元式是有可能产生中间变量或者使用中间变量的：

```
//对于每一个'+ - * /'指令，进行临时变量的产生与消除
            if (now_fourOp.op == O_PLUS || now_fourOp.op == O_MINU ||
                now_fourOp.op == O_MULT || now_fourOp.op == O_DIV) {
                //(1)先消除
                now_fourOp.item2 = remove(now_fourOp.item2);
                now_fourOp.item3 = remove(now_fourOp.item3);

                //(2)再产生(根据运算法则，产生的可以占用消除的寄存器)
                if (!now_fourOp.isArraySubPlus)now_fourOp.item1 =
generate(now_fourOp.item1);
            }

            //对每一个消除指令，进行临时变量的消除
            if (now_fourOp.op == O_ASSIGN) {
                //item2
                now_fourOp.item2 = remove(now_fourOp.item2);
            }
```

```
            if (condition_branch(now_fourOp.op)) {
                //item1与item2
                now_fourOp.item1 = remove(now_fourOp.item1);
                now_fourOp.item2 = remove(now_fourOp.item2);
            }
            if (now_fourOp.op == O_GiveParameters) {
                //item2
                now_fourOp.item2 = remove(now_fourOp.item2);
            }
            if (now_fourOp.op == O_PRINTF) {
                //item2
                now_fourOp.item2 = remove(now_fourOp.item2);
            }
            if (now_fourOp.op == O_SaveBackValue) {
                //item
                now_fourOp.item1 = remove(now_fourOp.item1);
            }
```