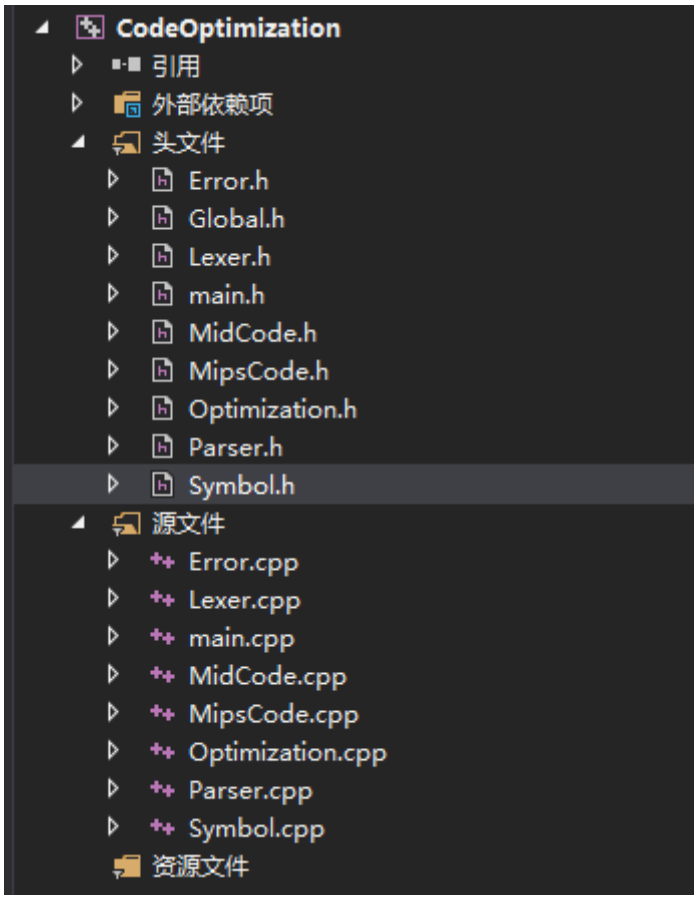
编译技术课程设计设计文档

编译技术课程设计设计文档
- 一、整体设计思路
- 二、词法分析
- 三、语法分析
- 四、错误处理及符号表管理
  - 4.1 错误处理概述
  - 4.2 符号表的建立和更新
- 五、中间代码生成
- 六、目标代码生成
- 七、代码优化
  - 7.1 常数优化
  - 7.2 内联函数优化
  - 7.3 临时寄存器优化
  - 7.4 全局寄存器优化
  - 7.5 其他优化

一、整体设计思路

本次编译课程设计的主要流程是按照理论课介绍的内容进行的。主要将编译器分为前后端两个部分，前者包括词法分析、语法分析、错误处理、中间代码生成及优化，后者则主要包括通过中间代码生成目标代码及优化。前后端以中间代码为分界，在一定程度上相互独立。

在完成了本学期的所有课程设计作业之后，我的项目工程目录如下：



其中每个阶段都有独立的模块及头文件，方便进行阶段性渐进式开发。

后文将按照前后端的各个模块依次进行介绍大致实现流程和重难点。

二、词法分析

对于词法分析，我的大致思路是对单词种类进行分类，对保留字和分界符采用一符一类的方法，之后根据状态转移图进行不同种类单词的识别。简单来说，就是根据词法规则分析出状态转移图，然后根据状态转移图进一步写出词法分析程序。

词法分析过程中，我使用的主要变量及函数如下：

```
int isFirst = 1;              // 标记是否为第一次调用

int symbol;                   // 保存当前所识别单词的类型
char CHAR;                    // 存放当前读入的字符
char TOKEN[256];              // 存放单词字符串
char buffer[256];             // 文件输入缓冲区

int cnt;                      // 记录下一个读取的字符的位置
int cnt_len;                  // 记录当前缓冲区字符串的长度
int cnt_line;                 // 记录当前缓冲区在源文件中的行数，以便进行错误输出
int lastCountLine;

int lastBegin;                // 上一个Token起始位置
int lastSymbol;               // 上一个Token的类型
char lastTOKEN[256];          // 上一个Token的字符串

int lastlastBegin;            // 上上个Token起始位置
int lastlastSymbol;           // 上上个Token的类型
char lastlastTOKEN[256];      // 上上个Token的字符串

void getsym();                // 获取下一个 symbol
void getsym(bool isOutput);
void trysym();                // 获取下一个 symbol，但不输出
void ungetsym();              // 回退一个 symbol
void outputsym();             // 输出当前的 symbol

int isDigit(char c);          // 判断字符是否为 数字
int isLetter(char c);         // 判断字符是否为 字母
int isChar(char c);           // 判断字符是否为 字符中的字符，即 '<加法运算符>'|'<乘法运算符>'|'<字母>'|'<数字>'
int isString(char c);         // 判断字符是否为 字符串中的字符，即 "{十进制编码为32,33,35-126的ASCII字符}"

void cat(char s[], char c); // 将字符插入到字符串的末尾
int getnbc();                 // 从字符缓冲区获取下一个非空字符，返回 1 表示成功，返回 0 表示失败
void getch();                 // 从字符缓冲区获取一个字符
void ungetch();               // 从字符缓冲区回退一个字符
void reserve(char s[]);       // 判断获取到的字符串是否为保留字
void tolowercase(char s[]); // 将字符串中的大写英文字符转化为小写
```

词法分析是整个编译器的第一步，也是较为简单的一个环节。根据此次词法分析作业的要求，单词主要可以分为五类。

- 以<字母>开始的，包括标识符、const、int.....return等
- 以<数字>开始的，主要是整形常量
- 以单引号开始的字符常量
- 以双引号开始的字符串
- 以特殊字符开始的特殊符号

三、语法分析

语法分析的主要任务是根据文法规则，从源程序单词符号串中识别出语法成分，并进行语法检查。

语法分析过程中使用到的主要变量和函数如下：

```
// 以下为内部定义的变量和函数
int returnedType;                                   // 记录函数已经返回值的类型，0为无返回值，1为整型，2为字符，3为return;
map <string, int> functionReturnType;               // 记录函数返回值类型，0为无返回值，1为整型，2为字符
map <string, int> functionParameterCount;           // 记录有返回值函数的参数数量
map <string, int[100]> functionParameterType;       // 记录有返回值函数的参数类型

// 递归调用子函数
void getAdditionOperator(int& additionOperator);                    // <加法运算符>
void getMultiplicationOperator(int& multiplicationOperator);        // <乘法运算符>
void getRelationalOperator(int& tmpRelationalOperator);             // <关系运算符>
void getChar(string& c);                                            // <字符>
void getString(string& s);                                          //*<字符串>

void getProgram();                                      //*<程序>
void getConstantDeclaration();                          //*<常量说明>
void getConstantDefination();                           //*<常量定义>
void getUnsignedInteger(int& tmpUnsignedInteger);       //*<无符号整数>
void getInteger(int& tmpInteger);                       //*<整数>

void getIdentifier(string& identifier);                 // <标识符>
void getConstant(int& tmpConstType, int& constInteger, string& constChar);  //*<常量>

void getVariableDeclaration();                          //*<变量说明>
void getVariableDeclarationForProgram(int tmpTypeIdentifier);
void getVariableDefination();                           //*<变量定义>
void getVariableDefinationForProgram(int tmpTypeIdentifier);
void getVariableDefinationWithoutInitialization(int tmpTypeIdentifier);  //*<变量定义无初始化>
void getVariableDefinationWithInitialization(int tmpTypeIdentifier, int dimension);     //*<变量定义及初始化>
```

```
void getTypeIdentifier(int& typeIdentifier);
// <类型标识符>

void getFunctionDefinationWithReturnValueForProgram(int returnType);
//*<有返回值函数定义>
void getFunctionDefinationWithoutReturnValueForProgram(int returnType);
//*<无返回值函数定义>
void getCompoundStatements();
//*<复合语句>
void getParameterList();
//*<参数表>
void getMainForProgram();
//*<主函数>

void getExpression(int& type, string& ansTmp);          // 表达式
void getTerm(int& type, string& ansTmp);                // 项
void getFactor(int& type, string& ansTmp);              // 因子

void getStatement();
void getAssignmentStatement();
void getConditionalStatement();
void getCondition(string& tmpCondition);
void getLoopStatement();
void getStep(int& tmpStep);
void getSwitch();
void getCaseTable(int tmpExpressionType, string tmpExpression, string
endLabel);
void getCase(int tmpExpressionType, string tmpExpression, string endLabel);
void getDefault(string endLabel);
void getFunctionCallStatementWithReturnValue(int& type);
void getFunctionCallStatementWithoutReturnValue();
void getValueParameterTable(string tmpFunctionCall);
void getStatementList();
void getScanf();
void getPrintf();
void getReturn();

// 检查symbol类型
void checkSymbol(int symType, char errorType);
void checkSymbolandGetSym(int symType, char errorType);

// 检查Identifier
void checkUndefinedIdentifier(string tmpIdentifier);
```

在语法分析过程中使用到的函数主要是递归子程序法所使用的子函数，再进行语法分析的过程中也穿插着进行符号表管理、错误处理和一定的代码优化。

在本次作业中，我的程序使用的主要思路是**自顶向下的递归子程序法。**具体来说，就是给语法的每一个非终结符都编写一个分析程序，当根据文法和当前的输入符号预测到要用到某个非终结符去匹配输入串时，就调用该非终结符的分析程序。

本次作业的文法规则中没有左递归的文法，但是部分非终结符的规则右部的多个选择会有一些相交的首符号集合。因此，遇到这些非终结符时（例如`<程序>`右部的`<变量说明>`和`<有返回值函数定义>`）我在本次作业中使用了**超前扫描**的方法来规避回溯的操作。

关于词法分析程序和语法分析程序的关系，我这里采用了**词法分析程序作为单独的子程序**的实现方案。在分析语法成分的同时不断从词法分析子程序中取单词，从而提高程序的效率。

如果将整个程序视作一棵语法树，那么我们的根节点就是`<程序>`，从`<程序>`这个语法成分出发，不断根据文法规则调用递归子程序，直至次法分析的得到的所有单词都能根据文法规则找到对应的语法成分为止。

四、错误处理及符号表管理

4.1 错误处理概述

错误处理的主要任务是给根据给定的文法设计实现错误处理程序，能诊察出常见的语法和语义错误，进行错误局部化处理，并输出错误信息。完成错误处理主要分为两个步骤：第一个是建立符号表，第二个是根据不同类型错误可能出现的位置判断不同的错误类型。

我们的作业中较多涉及错误的诊察，但是对出错之后的处理考察并不多。

错误处理阶段使用的主要变量和函数如下：

```
// 以下为外部引用的变量和函数
extern ofstream errorFile;
extern int cnt_line;
extern int lastCountLine;

// 以下为内部定义的变量和函数
void error(char errorCode);
```

以下按照作业要求的错误类型，依次介绍各种错误类型。

- 错误类型a
  - 错误类型a是本次作业中唯一的文法错误，所以需要在文法分析阶段进行处理。主要分为以下两个情况。
    - 在有单引号包围的字符情况下：检查读取到的字符是否有非法字符。
    - 在有双引号包围的字符串情况下，检查读取到的字符是否有非法字符；并且在读取到右双引号之后检查读取到的字符串是否为空集。
- 错误类型bc
  - 由于根据我们的文法，我们并不会有太多的函数嵌套，常量定义和变量定义也只有在`<程序>`、`<有返回值函数定义>`、`<无返回值函数定义>`中才会出现。因此在查询符号表的过程中，只需要查询当前所在函数的符号表和最外层的符号表（全局变量）即可。所以我们在常量定义和变量定义的过程中向符号表中增加符号，在使用到标识符的地方查询符号表是否有相应的符号。需要注意的是：建立符号表，如果是变量及初始化，每行仅能有一个；若变量无初始化，每行可以有多个。
- 错误类型de
  - 在函数定义的`<参数表>`部分，我们可以更新该函数的参数表。在函数调用`<值参数表>`部分，我们可以将实际的输入参数和参数表中的需要的参数进行比对，进而识别并输出错误。
- 错误类型f
  - 该错误可能发生的位置较为单一，在条件判断语句中判断等号两边的表达式是否为整型，进而识别并输出错误。
- 错误类型g和h
  - 要判断函数的返回类型是否和定义相同，我们需要建立一个函数表。

    - 当我们读取到 `＜有返回值函数定义＞` 的 `＜声明头部＞` 以及 `＜无返回值函数定义＞` 的 `void＜标识符＞` 部分，我们就可以判断出该函数的返回值类型和函数名称。因此我们在这里将函数名称和返回值类型加入到函数表中，当读取到 `<语句列>` 中的 `<返回语句>` 时与函数表进行比较判断，进而识别并输出错误。
- 错误类型i
    - 该错误只会出现在<因子>和<赋值语句>中。在这些语句可能使用到数组的地方，如果接收到的数组下标不是整型，则识别并输出该型错误。
- 错误类型j
    - 该错误只会出现在赋值语句和scanf语句中。在赋值语句和scanf语句出现标识符的地方，通过符号表查询是否为常量，从而识别错误并输出。
- 错误类型klm
    - 该错误相对来说比较容易识别。在原来的语法分析的基础上，需要接收分号、右小括号和右中括号的地方，如果未能成功接收该符号，则捕捉并输出该错误。需要注意的是，如果缺少的符号位于一行的最后一个，那么在输出错误的时候要注意行数需要定位到上一个单词所在的行数。以缺少分号为例，调用 `getsym()` 函数会获取到下一行的第一个单词，相应的行数计数器也会增加，但是我们缺少分号的错误行数应该是上一行。
- 错误类型o
    - 该错误主要出现在变量定义及初始化和switch语句中的 `<常量>`，需要判断等号两边的类型是否契合，否则捕捉o型错误。
- 错误类型p
    - 该错误需要在 `<情况语句>` 中没能成功接受 `DEFAULTTK` 的symbol时捕捉。

4.2 符号表的建立和更新

关于符号表的建立，由于我们的文法规则中不存在函数循环调用的情况，也就是说自定义函数只会在主函数中被调用，因此我使用了 `map<string, map<string, struct sym>>` 来存储符号表。

符号表管理阶段使用到的主要变量和函数如下：

```
int globalAddr = 0;                 // 记录全局变量在栈中的地址
int localAddr = 0;                  // 记录局部变量在栈中的地址

string curLevel;                    // 记录当前所在的符号表层次名称
struct sym curSym;                  // 记录当前正在使用的符号
map<string, struct sym> curMap;     // 记录当前所在的符号表层次的map
map<string, map<string, struct sym>> symList;

void initCurrentSymbol();
void insertCurrentSymbol();
void insertCurrentLevel();
void insertSymbol(string name, int kind, int type);
```

符号表的建立和更新则主要分为两步：其一是遇见新的函数定义，则需要添加新的层次；其二是遇见新的符号，则需要在当前层次下添加符号。

需要向符号表中增加层次主要有以下四种函数，我使用了函数名来作为该层次的key值。

- `<程序>`，暂时使用 `Program` 作为该层次的名称
- `<有返回值函数>`，使用函数名的标识符作为该层次的名称
- `<无返回值函数>`，使用函数名的标识符作为该层次的名称
- `<主函数>`，暂时使用 `Main` 作为该层次的名称

需要在同一层次向符号表中添加项的主要有如下情况。

- 常量定义（kind=0）：相对变量定义较为简单，只有整数和字符两种情况，不会有数组或者字符串。
  - `<程序>`中的`<常量说明>`
  - `<有返回值函数定义>`和`<无返回值函数定义>`中的`<复合语句>`中的`<常量说明>`
- 变量定义（kind=1）
  - `<程序>`中的`<变量说明>`
  - `<有返回值函数定义>`和`<无返回值函数定义>`中的`<复合语句>`中的`<变量说明>`
- 有返回值函数定义（kind=2）
  - `<程序>`中的`<有返回值函数定义>`
- 无返回值函数定义（kind=3）
  - `<程序>`中的`<无返回值函数定义>`
- 参数表（kind=4）
  - `<有返回值函数定义>`和`<无返回值函数定义>`中的`<参数表>`
- 主函数定义（kind=5）
  - `<主函数>`

五、中间代码生成

在生成中间代码阶段，我使用到的变量和函数如下。

```
// 以下为外部引用的变量和函数
extern ofstream midCodeFile;                                    // 中间代码输出文件
extern vector <struct midCode> midCodes;                        // 保存中间代码的容器
extern map<string, map<string, struct sym>> symList;            // 符号表

// 以下为内部定义的变量和函数
void insertMidCode(midOp op, string result, string left, string right);
void insertMidCode(midOp op, string result, string left, string right, string backup);
void outputMidCodes();
void outputMidCode(struct midCode m);
```

关于中间代码的格式，我使用的是课程组推荐的中缀表达式的结构，包含了结果、左操作数、操作符、右操作数和备用操作数四个成分。其中的备用操作数仅用于二维数组的相关操作。

```
struct midCode {
    string result;      // 结果
    string left;        // 左操作数
    midOp op;           // 运算符
    string right;       // 右操作数
    string backup;      // 备用操作数
};
// midOp是自己定义的包含中间代码操作符的枚举类型
```

代码生成一的作业中仅涉及<常量说明>、<变量说明>、<读语句>、<写语句>、<赋值语句>五种成分，我们首先对这些元素进行一定分析：

- <常量说明>：将符号保存在符号表中，并且保存相应的常量值。
- <变量说明>：将符号保存在符号表中，特别的对于有初始值的变量的操作类似于常量，只不过还可以进行赋值等操作。
- <读语句>：根据读语句中的标识符，通过mips指令中的系统调用读取相关数据并保存到对应的内存地址。
- <写语句>：根据写语句中需要数据的字符串和标识符，通过mips指令中的系统调用输出相关数据。
- <赋值语句>：取出操作数的值，并保存到结果的内存地址即可。

除了这些成分，中间代码生成过程中难度较大的部分是关于<表达式>的中间代码生成过程。在这里我主要使用了符号栈的思想，当读取到运算符时，弹出并检查当前在栈顶的两个元素，如果两者都是符号（常量或者变量），则生成一个新的中间临时变量作为操作结果并添加到符号表中；倘若两者中的左操作数为中间临时变量，则将运算后的结果直接保存到该中间临时变量即可。一直这样操作下去，最后将符号栈中剩余的最后一个元素，就是保存了该表达式结果的标识符。返回该标识符即可、

代码生成二的作业中则包含了所有的语法成分，我分析之后发现相较于第一次作业，主要增加了如下五种成分的语义分析：

- 数组相关：对于数组的取用，主要难度在于mips代码中对数组元素地址的计算，需要在数组地址的地址的基础上加入元素的偏移。
- 函数相关：函数相对而言是较为容易实现的，因为main本身就是一个特殊的函数，我们只需要在函数调用语句的部分加入跳转指令，并在<返回语句>中跳转回跳转前的地址即可。
- <条件语句>：首先将条件语句中的条件转化为一个跳转指令，同时对if和else分别生成两个标签`ifLabel`和`elseLabel`，前者用于条件不符合时跳转至else语句或者跳出该条件语句，后者则用于条件符合时跳出else语句。
- <循环语句>：循环语句包含`while`和`for`两种情况。
  - `while`语句：在循环开始处进行条件判断是否需要跳出循环，在循环语句最后插入跳转指令，返回while语句的条件判断处。这里使用了两个标签`beginLabel`和`endLabel`，分别用于跳转回循环开始处，和条件不满足时跳出循环语句。
  - `for`语句：for语句的大致结构与while语句相似，主要是在for语句开始前增加了对循环变量的初始化操作，以及循环语句的最后需要单独增加修改循环变量的语句。与while语句相同的是，这里使用了两个标签`beginLabel`和`endLabel`，分别用于跳转回循环开始处，和条件不满足时跳出循环语句。
- <情况语句>：对于情况语句，我采用了比较笨的办法，就是将每个case转化为了一个if语句进行分析，如果case语句中的变量和主变量不同则跳过该case语句，在每个case语句的最后增加一个跳转出整个switch语句的指令，从而完成了整个情况语句的代码构建。

六、目标代码生成

在生成mips代码阶段，我使用到的变量和函数如下。

```
// 以下为外部引用的变量和函数
extern ofstream mipsCodeFile;
extern map<string, map<string, struct sym>> symList;
extern vector <struct midCode> midCodes;
extern vector <struct mipsCode> mipsCodes;

vector<string> stringList;          //保存所有的字符串

string curFuncName = "";            // 记录当前所在函数名称
extern int string2int(string s);
```

```
extern string int2string(int t);

// 以下为内部定义的变量和函数
void generateMipsCodes();
void loadValue(string symbolName, string regName, bool gene, int& va, bool&
get);
void storeValue(string symbolName, string regName);
void insertMipsCode(mipsOp op, string result, string left, string right,
int imm);
void insertMipsCode(mipsOp op, string result, string left, string right);
void outputMipsCodes();
void outputMipsCode(struct mipsCode mc);
```

关于mips代码的格式，我采用的是类似于mips指令的结构，包含了指令、结果、左操作数、右操作数和立即数。

```
struct mipsCode {
    mipsOp op;          // 操作
    string result;      // 结果
    string left;        // 左操作数
    string right;       // 右操作数
    int imm;            // 立即数
};
// mipsOp是自己定义的包含mips指令的枚举类型
```

在我的编译器设计中，代码逻辑上的跳转等已经在生成中间代码的过程中完成了设计，所以生成mips指令过程的主要工作就是为中间代码寻找对应的mips指令进行翻译即可。

从中间代码到mips代码，最大的变化就是运算的操作数从无限的变量到有限的寄存器中。因此我们在进行相应的运算之前需要将符号表或者内存地址中的值取出到寄存器中，并在运算结束后保存到内存地址中。在本次作业中，我没有对寄存器的使用进行优化，而是采用了每条语句后还原的思路，保证每条语句之间的寄存器使用互不影响。

从上述描述中，不难发现，生成mips指令阶段的主要任务就是在匹配相应的mips指令的基础上分配内存地址和寄存器。

我在这里主要分析函数调用和数组取用两个过程进行分析。

- 函数调用：函数调用的过程首先将相关变量（包括函数参数、变量和生成中间代码过程中产生的临时变量）、`$ra`和`$fp`寄存器保存到栈中，之后就可以修改栈指针并跳转至相应的标签。
- 数组元素：数组元素的取用主要是要进行地址的计算，这里以获取二维数组中的元素为例，先将两个下标分别保存至`$t0`和`$t1`寄存器，之后根据`$t0 * column($t3) + $t1`计算出该数组元素相对于数组地址的偏移，进而找出该数组元素在内存地址中位置。

七、代码优化

7.1 常数优化

在代码生成的过程中，如果标识符对应的是常数，则在大多数的情况下可以进行一定的优化。

- 加减法运算中，遇到一个操作数为常数时，可以用addi和subi
- 乘除法运算中，遇到操作数中有0或1的情况可以进行简化，例如x*1、x*0、x/1、0/x等情况下可以不进行调用乘除操作就得到相应的运算结果

- 条件跳转语句中，如条件两边均为常数，可以直接得出结果以判断是否跳转
- 运算过程中可以使用$0寄存器替代数字0
- 数组调用以及数组赋值过程中，若数组下标为常数，可以直接计算出正确的地址

通过用常数替换从内存中获取数据并用寄存器参与运算的过程，可以节省大量的load操作的时间。

7.2 内联函数优化

对于没有跳转语句、函数调用语句、数组引用的函数，可以使用内联函数进行优化，从而减少函数调用带来的跳转等必然消耗。通过内联函数优化，可以节省很多保存现场的时间，此外还可以减少很多参数传递操作所需要的时间。当该函数位于循环语句中，内敛优化的效果会格外明显。

在进行函数内联之前，我们需要对原函数中的变量名进行预处理，如果是原函数的局部变量则需要重命名并加入到外层函数的符号表中，如果是全局变量则不需要进行处理。

7.3 临时寄存器优化

对于临时寄存器，可以建立寄存器池对临时寄存器进行统一分配。当遇到新的未分配寄存器的变量时从寄存器池中获取新的寄存器，当遇到所有的寄存器都已经被使用的情况时，则需要考虑释放寄存器。在释放寄存器的同时，要按照寄存器对应的变量是否被改写判断寄存器中的值是否需要写回内存。当一个基本块结束时，我们需要释放所有的临时寄存器。

7.4 全局寄存器优化

对于全局寄存器，我们可以通过理论课上所讲的引用计数法或者图着色算法进行全局寄存器分配。

首先我们先来介绍全局寄存器相较于临时寄存器的优点。临时寄存器的生存周期是一个基本块，当基本块结束时需要释放临时寄存器，而在下个基本块中如果需要再次使用到这个变量，则需要重新从内存中读取并写入寄存器。这种操作对于仅需使用一次的变量没什么影响，但是对于那些经常需要使用的变量，分配一个全局寄存器会比使用临时寄存器节省大量的存取内存的时间。

基于引用计数法分配全局寄存器的方法相对来说比较简单。以一个函数为一个全局域，统计该域内所有变量的引用次数，对于引用次数较多的变量则分配全局寄存器。在使用引用计数法统计变量的引用次数时，要注意调整循环变量的引用次数的权重，因为单纯的引用计数可能会减少循环变量的作用。

图着色法分配寄存器需要首先进行基本块划分与活跃变量分析。基本块的起始语句有三种情况：函数的第一条语句；跳转语句跳转到的语句；跳转语句之后的语句。活跃变量分析则是根据基本块的划分结果按照理论课上所学知识进行每个基本块的in、out、def、use集合，从而进行后续的图着色。

7.5 其他优化

以下都是一些细节之处的优化，故在此一并指出：

- 遇到 `#T1=#T2+#T3;` `a=#T1;` 这种情况可以简化为 `a=#T2+#T3;`
- 删除无用代码，对于结果操作数在后文中没有再使用到的情况下，可以直接删除
- 对于循环语句，尽可能地使用beq和bne语句替换跳转语句以提高性能