# 需求说明

## 文法说明

**为了提高效率或消除回溯，对部分文法进行修改。修改后的文法以 <\*语法成分\*> 表示.**

<程序> ::= <分程序>.

<分程序> ::= [<常量说明部分>][<变量说明部分>]{[<过程说明部分>]|[<函数说明部分>]}<复合语句>

<常量说明部分> ::= const<常量定义>{,<常量定义>};

<变量说明部分> ::= var<变量说明>;{<变量说明>;}

<\*过程说明部分\*>::= <过程首部><分程序>;

<过程首部> ::= procedure<标识符>[<形式参数表>];

<\*函数说明部分\*>::= <函数首部><分程序>;

<函数首部> ::= function<标识符>[<形式参数表>]:<基本类型>;

<形式参数表> ::= '('<形式参数段>{;<形式参数段>}')'

<复合语句> ::= begin<语句>{;<语句>}end

<常量定义> ::= <标识符>＝<常量>

<常量> ::= [+|-] <无符号整数>|<字符>

<字符> ::= '<字母>'|'<数字>'

<变量说明> ::= <标识符>{,<标识符>}:<类型>

<类型> ::= <基本类型>|array'['<无符号整数>']'of<基本类型>

<基本类型> ::= integer|char

<字符串> ::= "{十进制编码为32,33,35-126的ASCII字符}"

<无符号整数> ::= <数字>{<数字>}

<标识符> ::= <字母>{<字母>|<数字>}

<形式参数段> ::= [var]<标识符>{,<标识符>}:<基本类型>

<\*语句\*> ::= <赋值语句>|<返回值语句>|<条件语句>|<当循环语句>|<过程调用语句>|<复合语句>|<读语句>|<写语句>|<for循环语句>|<空>

<赋值语句> ::= <标识符>:=<表达式>|<标识符>'['<表达式>']':=<表达式>

<\*返回值语句\*> ::= <函数标识符>:=<表达式>

<函数标识符> ::= <标识符>

<表达式> ::= [+|-]<项>{<加法运算符><项>}

<项> ::= <因子>{<乘法运算符><因子>}

<因子> ::= <标识符>|<标识符>'['<表达式>']'|<无符号整数>|'('<表达式>')'|<函数调用语句>

<函数调用语句> ::= <标识符>[<实在参数表>]

<实在参数表> ::= '('<实在参数>{,<实在参数>}')'

<实在参数> ::= <表达式>

<加法运算符> ::= +|-

<乘法运算符> ::= \*|/

<条件> ::= <表达式><关系运算符><表达式>

<关系运算符> ::= <|<=|>|>=|=|<>

<\*条件语句\*> ::= if<条件>then<语句>[else<语句>]

<当循环语句> ::= do<语句>while<条件>

<for循环语句> ::= for<标识符>:=<表达式>(downto|to)<表达式>do<语句>

<过程调用语句> ::= <标识符>[<实在参数表>]

<读语句> ::= read'('<标识符>{,<标识符>}')'

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

<字母> ::= a|b|c|d…x|y|z |A|B…|Z

<数字> ::= 0|1|2|3…8|9

**附加说明：**

（1）char类型的变量或常量，用字符的ASCII码对应的整数参加运算

（2）标识符区分大小写字母

（3）赋值语句中<函数标识符> := <表达式> 作为函数的返回值，其类型应与返回类型一致，此语句后面的语句可继续执行

（4）写语句中的字符串原样输出，表达式只有单个字符类型的变量或常量按字符输出，其他表达式均按整型输出

（5）数组的下标从0开始

（6）带var的参数为变量形参，实参与该类形参传递数据时是传地址

## 目标代码说明

目标代码为 x86 汇编代码．所有指令都使用32位指令．用到如下指令：

|  |  |
| --- | --- |
| **指令** | **功能** |
| PUSH | 压栈 |
| POP | 弹栈 |
| MOV | 复制数据 |
| ADD | 加法 |
| SUB | 减法 |
| IMUL | 乘法 |
| CDQ | 填充（为除法作准备） |
| IDIV | 除法 |
| NEG | 求负 |
| AND | 按位与（用于截断） |
| PROC | 声明函数/过程 |
| ENDP | 函数/过程的结束 |
| END | 整个程序的结束 |
| CMP | 比较 |
| JG | 大于则跳转 |
| JGE | 大于等于则跳转 |
| JL | 小于则跳转 |
| JLE | 小于等于则跳转 |
| JE | 等于则跳转 |
| JNE | 不等于则跳转 |
| JMP | 无条件跳转 |
| CALL | 函数/过程调用 |
| RET | 函数/过程返回 |
| LEAVE | 还原栈指针寄存器 |
| SHL | 左移（用于计算数组元素的偏移） |

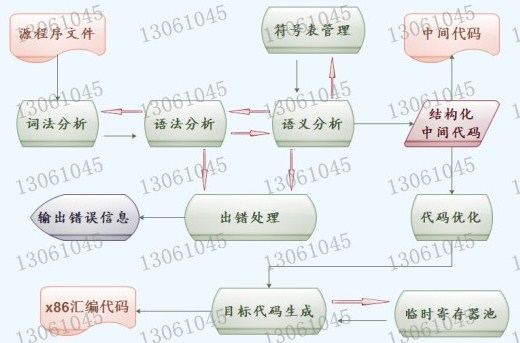
## 优化方案

临时寄存器池．将 eax, ebx, ecx, edx, esi, edi 这六个寄存器都当作临时寄存器来用，使用临时寄存器池分配．

# 详细设计

## 程序结构

图中粗箭头代表调用关系，细箭头代表信息传递。



## 类/函数功能

以下，只写比较重要的类，只写public函数。

1. **class LexicalAnalyzer**

词法分析器类。完成从文件中读取符号并分类的功能。

把语法分析器类等作为友元，以向其暴露符号类型symbolType、获取到的符号token、以及这个符号的行号lineNumber。

因为用到全局变量，所以须用单例模式。

**static LexicalAnalyzer &  
initialLexicalAnalyzer(ifstream &)**

由于使用单例模式，这个函数充当构造函数。如果多次调用这个函数，只返回同一个对象的引用。

**void nextSymbol()**

获取下一个符号，保存这个符号的类型及行号，如有必要，需要保存这个符号本身（比如标识符等）。

**void nextChar()**

获取下一个“字符”，指字符常量中单引号中间的部分。目前的设计是：

* 跳过空白符；
* 如果遇到非法字符，则将这个符号保存下来，并将类型置为“非法”；
* 如果遇到多个字符，只读一个，保存下来，并将类型置为“字符”。

以上设计有可能在不断的完善中更改。

**void n**extSt**ring()**

获取下一个字符串，指字符串常量中双引号中间的部分。目前的设计是：

* 允许有空串；
* 如果读到非法字符，则将非法字符之前的部分保存，将类型置为“字符串”；

以上设计中的第二条，有可能会在实现中不断完善，可能会有更改。

1. **class GrammarAnalyzer**

语法分析器。按照文法，使用递归下降子程序法，将词法分析读到的一个一个单词与文法匹配，并调用相应的语义处理函数。如果有错误，则调用错误处理函数。

用单例模式。

#### GrammarAnalyzer &initialGrammarAnalyzer(LexicalAnalyzer &, ErrorHandler &,SymbolTableManager &, SemanticAnalyzer &);

语法分析器的伪构造函数．

#### void grammarAnalyze();

这是对main函数提供的接口，也是编译过程前半程的入口。由此，进行语法分析、语义分析、错误处理、中间代码生成等过程。

**（各大语法成分的解析函数）**

这里函数太多，不逐一展开赘述。使用递归下降子程序法，每一个语法成分都有一个独立的处理函数，它们会调用词法分析程序、语义分析程序，如果出错，会调用错误处理程序。思路与教材相似，但是我的设计是，错误信息的输出由错误处理程序执行，这样程序的结构更加清晰。

1. **class SemanticAnalyzer**

语义分析器。与语法分析器配合，递归调用相应的函数，完成与语义相关的分析工作并生成结构化的四元式代码。与符号表、错误处理、四元式等类都有交互。

在设计上，语义分析器相当于给语法分析器提供的一个函数库，所以要用单例模式。

#### 各个语义动作函数如下：

void programEndmapInit();

void blockEndmapInit();

void blockEndmapClear();

void constDeclareEndmapInit();

void constDeclareEndmapClear();

void constDefineEndmapInit();

void constDefineEndmapClear();

void charEndmapInit();

void charEndmapClear();

void stringEndmapInit();

void stringEndmapClear();

void varDeclareEndmapInit();

void varDeclareEndmapClear();

void varDefineEndmapInit();

void varDefineEndmapClear();

void procedureDeclareEndmapInit();

void procedureDeclareEndmapClear();

void functionDeclareEndmapInit();

void functionDeclareEndmapClear();

void procedureHeadEndmapInit();

void procedureHeadEndmapClear();

void functionHeadEndmapInit();

void functionHeadEndmapClear();

void formalParameterEndmapInit();

void formalParameterEndmapClear();

void statementEndmapInit();

void statementEndmapClear();

void termEndmapInit();

void termEndmapClear();

void factorEndmapInit();

void factorEndmapClear();

void constDeclareSkip();

void charSkip();

void stringSkip();

以上这些函数用于出错处理，每个语法成分都对应一个Init和一个Clear,有的还需要一个Skip,遇到错误时,往后跳,读到一个符号就停,根据表中的符号与语法成分对应关系逐层返回到指定的语法成分,接着分析.

void constDefine(const string &, BasicType, int value);

string &stringDefine(const string &);

void varDefine(vector<string> &, BasicType);

void arrayDefine(vector<string> &, BasicType, int limit);

void procedureDefine(const string &procname, vector<Argument> &args);

void functionDefine(const string &funcname, vector<Argument> &args, BasicType type);

void formalParameterAdd(vector<string> &names, BasicType type,

bool vary, vector<Argument> &);

void getAddress(const string &base, const string &offset, string &addr);

void getContent(const string &base, string &value);

void assignFromAddress(const string &addr, const string &value,

BasicType addrType, BasicType valueType);

void assignToAddress(const string &addr, const string &value,

BasicType addrType, BasicType valueType);

void assignToArray(const string &arrname, const string &offset, const string &value,

BasicType arraytype, BasicType valuetype);

void assign(const string &name, const string &value,

BasicType nametype, BasicType valuetype);

void negative(const string &oper1, string &dest);

void addition(const string &oper1, const string &oper2, string &dest);

void subtraction(const string &oper1, const string &oper2, string &dest);

void multiplicatioin(const string &oper1, const string &oper2, string &dest);

void division(const string &oper1, const string &oper2, string &dest);

void intToString(int n, string &str);

void callFunction(const string &name, string &value);

void callProcedure(const string &name);

void functionAssign(const string &name, const string &value,

BasicType functype, BasicType valuetype);

void label(const string &lab);

void funcprocStart();

void funcprocReturn();

void push(const string &value);

void passParameter(const string &value, BasicType formaltype, BasicType realtype);

void passAddress(const string &addr, BasicType formaltype, BasicType realtype);

void compare(const string &value1, const string &value2);

void dissatisfyJump(SymbolType op, const string &lab);

void satisfyJump(SymbolType op, const string &lab);

void jumpAnyway(string lab);

string &newLabel();

string &newLabel(const string &forehead);

void readChar(const string &name);

void readInteger(const string &name);

void writeString(const string &name);

void writeInteger(const string &value);

void writeChar(const string &ch);

void newline();

// 要在生成四元式后，记录每个函数或过程用了多少个临时变量，所以提供这个函数，

// 查询 temporaryCount 的值，在 函数/过程 开头 get 一次，结尾 get 一次，

// 两次之差就是这个 函数/过程 用到的临时变量的个数。

int getTemporaryCount() { return temporaryCount; }

以上这些都是对每个语义动作提供的小函数,从名字就可以看出其功能.不一一介绍了.

1. **class SymbolTableManager**

符号表管理器。符号表全程只有一个，这里用作全局变量以提高效率，所以这里必须使用单例模式。

对外提供建表、查表、填表、删表等操作。由于查表函数会返回表项的指针，所以不需要再提供填表函数。

符号表使用树形符号表，用到map，里面的值为表项的指针。

#### static SymbolTableManager & initialSymbolTableManager()

单例模式中对外提供的“构造函数”，多次调用只返回同一个引用。完成符号表数据结构的构建。

#### void home();

将符号表的＂当前位置＂置为root.

#### void goIn(const string &);

将 position 往里推进一层

#### void setTempcount(int count);

将 position 的 multicount设为 count

#### int　getTempcount();

得到 position 的 multicount

#### void goOut();

将 position 往外退回一层

#### void comeHere();

将 position 指向 ste

#### bool isHere(const string &);

判断表中当前层是否有这个符号

#### bool find(const string &);

查表，查到返回 true，若查到，将 ste 指向它

#### void insert();

填表

#### void defineArguments(STE \*te, vector<Argument> &args);

定义参数表，将参数再复制一份作为局部变量

#### string &encodeKey();

把 ste 转化为唯一标识

#### string &encodePosition();

把 position 转化为唯一标识

#### void decodeKey(const string &);

把唯一标识转化为 ste

#### void addString(const string &name, const string &str);

把 str 命名为 name 并存入表中

#### map<string, string> &getStringMap();

返回 stringMap 给代码生成使用

#### map<string, STE\*> \*getChilds() { return position->childs; }

返回 childs 指针

1. **class Quadruples**

四元式列表。保存结构化的四元式，作为语义分析器、代码优化器、目标代码生成器三个类相互之间传递四元式代码的媒介。链表实现，对外提供一个迭代器．

#### void addTail()

往结尾加一个四元式．

#### iterator begin()

供迭代器使用，返回第一条指令的迭代器．

#### iterator end()

供迭代器使用，表示迭代器中不再有任何元素．

**void output()**

把当前所有的四元式输出到文件。参数为文件名。

1. **class RegisterPool**

临时寄存器池。给临时变量以及没有得到全局寄存器的变量分配和管理临时寄存器。这里的算法貌似没有最优的，因为没有人能预测未来，知道后面哪个变量用得多。只能随用随分。

#### RegisterPool(const vector<string> &registers, StackManager &stack, ofstream &out);

构造函数。第一个参数为临时寄存器的个数，后面为临时寄存器的名称．

#### void initialRegisterPool();

每进入一个新的过程或函数，都要调用这个函数！

#### void home(const string &registerName);

让 regist 寄存器回家（放回栈上）

void home();

让所有寄存器回家（放回栈上）

#### string getDestRegister(const string &temporary, bool wantValue = false);

#### string getDestRegister(const string &variable, int offset, int baseoffset = 0, bool wantValue = false);

参数为一个临时变量，返回它所在的寄存器名。如果这个临时变量在栈上，则给它分配一个寄存器；如果所有寄存器中都有有效值，则让某个临时变量回家，然后把腾出来的寄存器给要的临时变量；如果已给临时变量分配栈空间，没有问题；如果还没有给这个临时变量分配栈空间，则会给它分配空间，这时会改变 pushCount 的值．要求要的值在寄存器里

#### string getDest(const string &temporary);

#### string getDest(const string &variable, int offset, int baseoffset = 0);

不要求变量的值在寄存器里，可以在栈上

#### string getRegister();

获取一个空的寄存器，不与变量相关联

#### void unlock(const string &registerName);

#### void lock(const string &registerName);

锁定，锁定后的寄存器不参与寄存器池的调度

#### void setLastOffset(int);

设置 lastOffset 的值

### class ErrorHandler

错误处理器．完成两个事情：输出错误信息；出错时往后跳．

#### void addEndmap(SymbolType, ErrorEnd);

往 endmap 里加一对

#### void removeEndmap(SymbolType);

从 endmap 里移除 SymbolType，如果没有，无操作

#### static ErrorHandler &initialErrorHandler(LexicalAnalyzer &, bool warning);

单例模式

#### ErrorEnd skip();

错误处理函数，根据endmap中的符号往后跳,直到跳到了其中的符号为止.

#### void printWarning(ErrorType err);

#### void printWarning(ErrorType err, const string &msg);

输出警告信息．

#### void printError(ErrorType);

#### void printError(ErrorType, const string &);

输出错误信息．

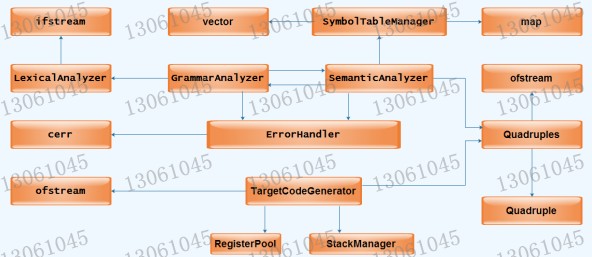
## 调用依赖关系

这里的依赖关系描述我认为比较关键的类和比较关键的函数。

此处的设计并不一定就是最终的实现，因为有好多问题，不实现是永远无法想到以及想清楚的。所以以后这个设计有可能会作改动。

下面两个图中，箭头A指向B意思是A调用B，或者A管理/支配B。

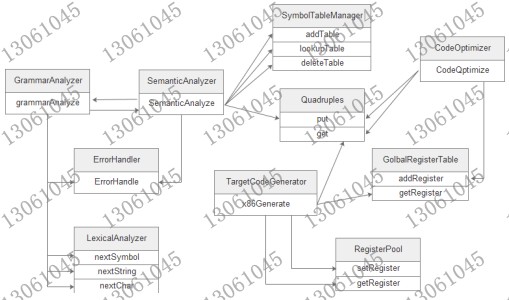
### 类关系



在编译过程的前半程，即词法分析、语法分析、语义分析、中间代码生成过程，程序的核心是语法分析器GrammarAnalyzer，它需要调用的类/对象有：词法分析器LexicalAnalyzer、语义分析器SemanticAnalyzer、错误处理器ErrorHandler等；而语义分析器又要调用错误处理器ErrorHandler、符号表管理器SymbolTableManager、以及四元式表Quadruples。

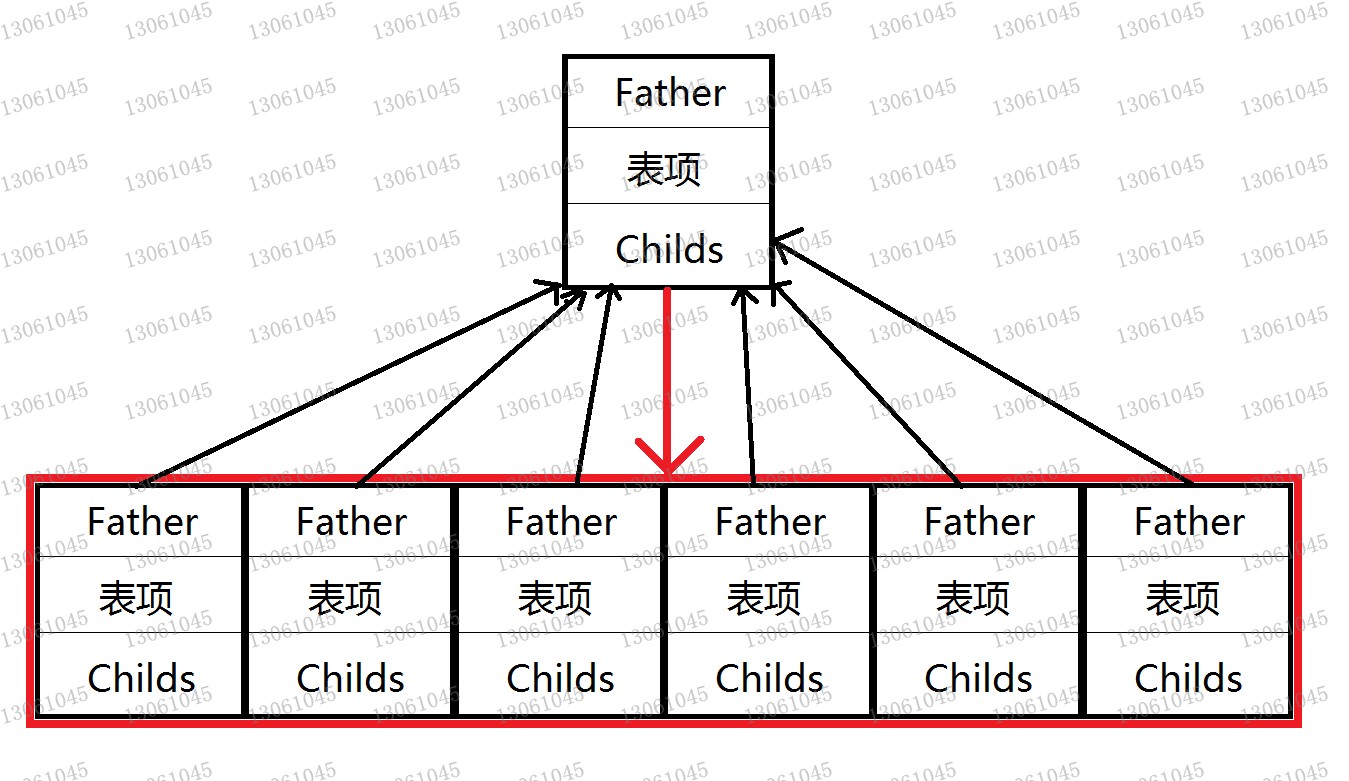
在编译过程的后半程,即目标代码生成，核心是TargetCodeGenerator，RegisterPool和StackManager为其服务．

### 函数调用关系



## 符号表管理方案

使用树形符号表．如图：



代码如下：

struct STE {

string name; // 标识符名称

IdentifierType type; // 标识符类型

BasicType valueType; // 常量、变量：值的类型；

// 数组：元素的类型；

// 函数：返回值类型

int multiInt; // 常量：值；

// 数组：容量限

// 函数 / 过程：临时变量的个数

int level; // 当前处于第几层

int location; // 变量/数组在运行栈上的相对地址

vector<Argument> \*arguments; // 函数/过程的参数列表

////////////////////////////////////////////////////////////////

private: // 以下，为构建树形符号表而必须的指针

STE \*father; // 当前符号的父节点（所属函数/过程）

map<string, STE \*> \*childs; // 当前符号(函数/过程)下的符号

friend class SymbolTableManager;// 把数据结构相关的指针暴露给符号表管理器

////////////////////////////////////////////////////////////////

public:

STE(): arguments(NULL),

childs(NULL),

location(0),

father(NULL) {} // 构造函数：给参数列表置为 NULL

~STE() {

delete childs; // 先删除子树，再删除当前节点

delete arguments;

} // 析构函数：先删除参数列表

// 慎用！析构时会删除以当前节点为根的子树！！

STE(const STE &ste) {

name = ste.name;

type = ste.type;

valueType = ste.valueType;

multiInt = ste.multiInt;

level = ste.level;

location = ste.location;

// 对这个指针要格外小心！！！

arguments = ste.arguments == NULL ? NULL :

new vector<Argument>(\*ste.arguments);

// 数据结构相关的指针留着以后处理。

father = NULL;

childs = NULL;

} // 拷贝构造

};

## 存储分配方案

为了提高代码运行时的效率，使用ESP一个寄存器管理运行栈，而EBP作为全局寄存器使用。

运行栈中的结构如下：（由栈底到栈顶，即由高地址到低地址）

* display区：可能有多个值，保存外层调用者的栈基址。由调用者保存。
* 实在参数区：可能有多个值，这是显式参数，由调用者保存。
* 返回地址：只有一个值，这是隐式参数，由调用者保存。
* 全局寄存器保存区：可能有多个值，由被调用函数保存。被调用函数用到哪个全局寄存器，就要把哪个全局寄存器的值保存起来，并在返回之前恢复。
* 局部数据区：可能有多个值，保存未被分配到全局寄存器的局部变量。
* 临时变量区： 通常都有多个值，保存每个临时变量。在调用其他函数之前，一定要确保每个函数都“回家”了。

运行栈结构大致如下图：



## 四元式设计

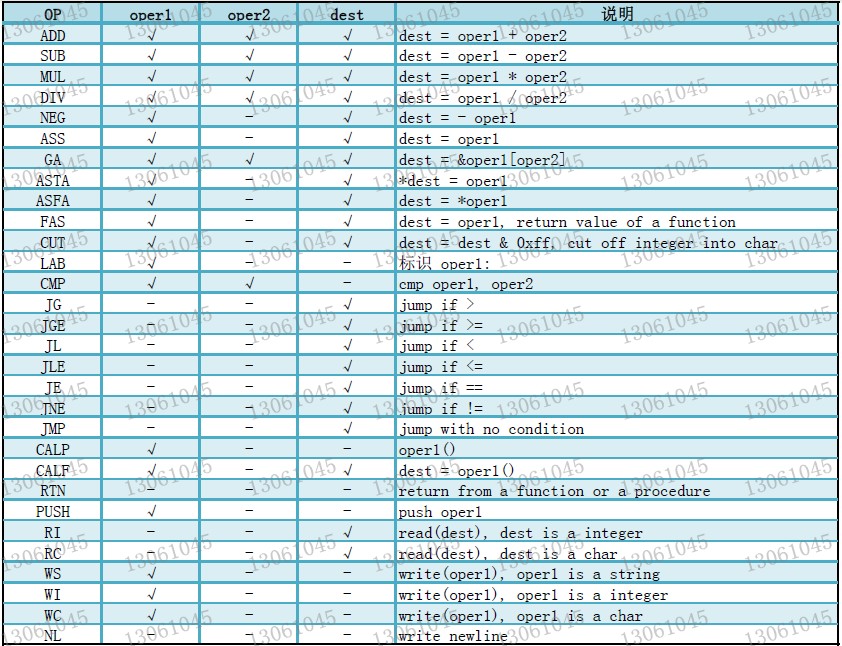
每一个四元式的输出形式都形如：

op oper1 oper2 dest

四个部分以空格分隔，op为操作符，两个操作数正常，结果dest放在四元式的最后一位。

操作数如果是源程序代码中有的标识符，则不变；如果是编译程序定义的临时变量，则用\_t\_1，\_t\_2等等表示，以与变量名区分。

语义分析生成的原始四元式暂不输出到文件，而是用结构化的方式保存在内存中，用于代码优化。所有四元式结构Quadruple保存在链表中．



## 目标代码生成方案

目标代码为x86汇编代码。

把优化之后的四元式转化为汇编代码时，要注意以下几点：

* 注意运行栈的管理，每个分程序在被调用时和返回时要进行运行栈的相关操作。具体如下：  
  调用函数时，必须要先把临时变量“回家”，然后先PUSH display区，再PUSH实参，再PUSH返回地址，然后跳转。上图中ESP0以上的位置都是调用方保存的。  
  被调用的函数如果要用全局寄存器，要先PUSH原来的全局寄存器的值，然后PUSH没有全局寄存器的局部变量，然后PUSH临时变量。
* 注意寄存器的分配问题。临时寄存器池在这个时间起关键作用。目标代码生成时，要时刻与临时寄存器池交互。

## 代码优化方案

临时寄存器池．用它来使临时寄存器分配更加合理一点．

## 出错处理

按书上的方法，报错之后跳几个东西，直到跳到编译工作可以继续执行时。然后继续正常的编译工作。

错误类型如下：

NO\_PERIOD, // 缺少句点

ILLEGAL\_CHAR, // 非法字符

NO\_FINISH, // 希望结束而没有结束

FINISH, // 程序尚不完整而文件结束

NO\_CONST, // 缺少 const 关键字

NO\_SEMICOLON, // 缺少分号

NO\_IDENTIFIER, // 缺少标识符

REDEFINE, // 标识符重定义

NO\_EQUAL, // 缺少等号

NO\_CHNUM, // 缺少字符或整数（<常量>）

NO\_SINGLEQUOTE, // 缺少单引号

NO\_DOUBLEQUOTE, // 缺少双引号

ILLEGAL\_STRING, // 非法字符串

NO\_NUMBER, // 希望是数字而不是

INT\_OVERFLOW, // 整数溢出

NO\_VAR, // 缺少 var 关键字

NO\_COLON, // 缺少冒号

NO\_LEFTSQUARE, // 缺少左方括号

NO\_RIGHTSQUARE, // 缺少右方括号

NO\_OF, // 缺少 of 关键字

NO\_BASICTYPE, // 缺少基本类型

NO\_PROCEDURE, // 缺少 procedure 关键字

NO\_FUNCTION, // 缺少 function 关键字

NO\_LEFTBRACKET, // 缺少左括号

NO\_RIGHTBRACKET,// 缺少右括号

CONSTANT, // 意外的常量标识符

NO\_ASSIGN, // 缺少赋值符，即希望是赋值语句，实际不是

NO\_VALUE, // 缺少值

UNDEFINED, // 标识符未定义

CANNOT\_CALL, // 标识符不能用于函数或过程调用

INT\_TO\_CHAR, // 将整型数转化为字符型

TOO\_LESS\_ARG, // 参数太少

TOO\_MANY\_ARG, // 参数太多

UNVARIABLE, // 需要变量表达式

NO\_RELATION, // 需要关系运算符而没有

NO\_IF, // 缺少 if 关键字

NO\_THEN, // 缺少 then 关键字

NO\_ELSE, // 缺少 else 关键字

NO\_DO, // 缺少 do 关键字

NO\_WHILE, // 缺少 while 关键字

NO\_FOR, // 缺少 for 关键字

NO\_DOWN\_TO, // 缺少 to 或 downto 关键字

NO\_BEGIN, // 缺少 begin 关键字

NO\_END, // 缺少 end 关键字

NO\_READ, // 缺少 read 关键字

NO\_WRITE // 缺少 write 关键字

不同的错误类型，在不同的语法成分中的对待方式是不同的，所以有一个表：

// skip 操作可结束时的 SymbolType 及返回值对照表

list<pair<SymbolType, ErrorEnd> > endmap;

这样往后跳时，就知道应该在哪个语法成分接着分析了．

# 操作说明

## 运行环境

编译器本身的运行环境就是win32，程序有几个命令行参数：第一个为源代码文件名，第二个为四元式文件名，第三个为目标代码文件名．如果还有其他选项，则应放在源代码文件名前面，以减号开头，一次性给出所有选项．选项有w和dw两种，分别用于打开警告和关闭警告．默认为打开警告．

汇编器使用masm32，汇编时必须将代码与masm32的根目录放在同一级．也就是如果masm32的主目录为C:\masm32，那么代码必须也放在C:\，与masm32在同一级．汇编和链接命令为：

masm32\bin\ml.exe /c /coff 文件名.asm

masm32\bin\link /subsystem:console 文件名.obj

## 操作步骤

在Visual Studio环境下，生成后按Alt+F7，在Debugging选项卡下设置命令行参数，按照上文所述的来设置．然后运行．

将生成的目标代码移动到上文所述的目录下，找开控制台，输入汇编和链接命令．然后执行．

# 测试报告

## 测试程序及测试结果

### 程序一（正确）

const TEN = 0010,

NINE = +9,

EIGHT1 = -8,

A = 'A',

ZERO = '0';

var int1, int2: integer;

ch1, ch2: char;

intArray: array[3] of integer;

chArray: array[1] of char;

procedure proc1;

begin

write("proc1 start...");

int1 := -(TEN + NINE) \* EIGHT1 + 1024;

ch1 := A;

ch2 := ZERO;

write("int1: ", int1);

write("ch1: ", ch1);

write("ch2: ", ch2);

write("chArray:");

for int2 := 0 downto 0 do

begin

chArray[int2] := ZERO;

write(chArray[int2]);

end;

write("intArray:");

for int2 := 0 to 2 do

begin

intArray[int2] := int2;

write(intArray[int2])

end;

begin

write("proc1 end.");

end

end;

function func1(num: integer): integer;

var steps: integer;

procedure mov(tstart, tend, tmid, num: integer; var count: integer);

begin

if num > 0 then

begin

mov(tstart, tmid, tend, num-1, count);

count := count + 1;

write("step ", count);

write("from: ", tstart);

write("to: ", tend);

mov(tmid, tend, tstart, num-1, count);

end

end;

begin

write("func1 start...");

write("argument is num: integer");

write(num);

write("Hanoi Tower with levels: ", num);

steps := 0;

mov(1, 3, 2, num, steps);

write("The total number of steps is: ", steps);

write("func1 end with: ", steps);

func1 := steps;

end;

function func2: char;

begin

write("func2 start...");

write("Please input ch1, ch2, int1...");

read(ch1, ch2);

read(int1);

if ch1 <= ch2 then

if ch1 < ch2 then

if 0 = int1 then

write("<=")

else

write("<")

else if int1 >= 0 then

if int1 > 0 then

write("<>")

else

write("=")

else

write("<>")

else

if 0 <> int1 then

write(">")

else

write(">=");

write("func2 end with: ", ZERO);

func2 := ZERO;

end;

procedure proc2(int1, int2: integer; var int3: integer);

begin

write("proc2 start...");

write("arguments are int1, int2: integer; var int3: integer");

write(int1);

write(int2);

write(int3);

int3 := 0;

do

begin

int3 := int3 + int2;

int1 := int1 + 1;

end

while int1 < int2;

write("int3 now is :", int3);

write("proc2 end.");

end;

begin

proc1;

proc2(int1, int2, intArray[3 \* 3 - 2 \* 5 + 1]);

int1 := func1(4);

ch1 := func2;

end.

输入：

4

d a 0

输出:

proc1 start...

int1: 1176

ch1: A

ch2: 0

chArray:

0

intArray:

0

1

2

proc1 end.

proc2 start...

arguments are int1, int2: integer; var int3: integer

1176

3

0

int3 now is :3

proc2 end.

func1 start...

argument is num: integer

4

Hanoi Tower with levels: 4

step 1

from: 1

to: 2

step 2

from: 1

to: 3

step 3

from: 2

to: 3

step 4

from: 1

to: 2

step 5

from: 3

to: 1

step 6

from: 3

to: 2

step 7

from: 1

to: 2

step 8

from: 1

to: 3

step 9

from: 2

to: 3

step 10

from: 2

to: 1

step 11

from: 3

to: 1

step 12

from: 2

to: 3

step 13

from: 1

to: 2

step 14

from: 1

to: 3

step 15

from: 2

to: 3

The total number of steps is: 15

func1 end with: 15

func2 start...

Please input ch1, ch2, int1...

>=

func2 end with: 0

### 程序二(错误)

const TEN = 0010,

NINE = +9,

EIGHT1 = -8,

A = 'A',

ZERO = '0';

var int1, int2: integer;

ch1, ch2: char;

intArray: array[3] of integer;

chArray: array[1] of char;

procedure proc1;

begin

write("proc1 start...");

int1 := -(TEN + NINE) \* EIGHT1 + 1024;

ch1 := A;

ch2 := ZERO;

write("int1: ", int1);

write("ch1: ", ch1);

write("ch2: ", ch2);

write("chArray:");

for int2 := 0 downto 0 do

begin

chArray[int2] := ZERO;

write(chArray[int2]);

end;

write("intArray:");

for int2 := 0 to 2 do

begin

ZERO := 0;

intArray[int2] := int2;

write(intArray[int2])

end;

begin

write("proc1 end.");

end

end;

function func1(num: integer): integer;

var steps: integer;

procedure mov(tstart, tend, tmid, num: integer; var count: integer);

begin

if num > 0 then

begin

mov(tstart, tmid, tend, num-1, count);

count := count + 1;

write("step ", count);

write("from: ", tstart);

write("to: ", tend);

mov(tmid, tend, tstart, num-1, count);

end

end;

begin

write("func1 start...");

write("argument is num: integer");

write(num);

write("Hanoi Tower with levels: ", num);

steps := 0;

mov(1, 3, 2, num, steps);

write("The total number of steps is: ", steps);

write("func1 end with: ", steps);

func1 := steps;

end;

function func2: char;

begin

write("func2 start...");

write("Please input ch1, ch2, int1...");

read(ch1, ch2);

read(int1);

if ch1 <= ch2 then

if ch1 < ch2 then

if 0 = int1 then

write("<=")

else

write("<")

else if int1 >= 0 then

if int1 > 0 then

write("<>")

else

write("=")

else

write("<>")

else

if 0 <> int1

write(">")

else

write(">=");

write("func2 end with: ", ZERO);

func2 := ZERO;

end;

procedure proc2(int1, int2: integer; var int3: integer);

begin

write("proc2 start...");

write("arguments are int1, int2: integer; var int3: integer");

write(int1);

write(int2);

write(int3);

int3 := 0;

do

begin

int3 := int3 + int2;

int1 := int1 + 1;

end

while int1 < int2;

write("int3 now is :", int3);

write("proc2 end.");

end;

begin

proc1;

proc2(int1, int2, intArray[3 \* 3 - 2 \* 5 + 1]);

int1 := func1(4);

ch1 := func2;

end.

报出来的错误:

Err 1: line 28: error E25: unexpected constant identifier. (ZERO)

Err 2: line 28: error E43: end keyword expected

Err 3: line 28: error E43: end keyword expected

Err 4: line 28: error E5: semilcolon expected

Err 5: line 28: error E42: begin keyword expected

Err 6: line 28: error E25: unexpected constant identifier. (ZERO)

Err 7: line 28: error E43: end keyword expected

Err 8: line 28: error E0: period expected

Err 9: line 28: error E2: file should be end

9 ERROR in total

### 程序三(正确)

const MAX=200;

var i :integer;

arr: array[20] of char;

procedure proc;

var a:integer;

procedure proc2;

var b:integer;

begin

a:=MAX;

b:=MAX-10;

i:=a-b;

write(" a= ",a);

write(" b= ",b);

end;

begin

proc2;

end;

function func(t:integer):integer;

begin

t := 2\*t;

func := t;

end;

begin

i:= func(10);

write(" i= ",i);

proc;

write(" i= ",i)

end.

运行结果:

i=20

a=200

b=190

i=10

### 程序四(错误)

const MAX=200;

var i integer;

arr: array[20] char;

procedure proc;

var a:integer;

procedure proc2;

var b:integer;

begin

a:=MAX;

b:=MAX-10;

i:=a-b;

write(" a= ",a);

write(" b= ",b);

end;

begin

proc2;

end;

function func(t:integer):integer;

begin

t := 2\*t;

func := t;

end;

begin

i:= func(10);

write(" i= ",i);

proc;

write(" i= ",i)

end.

错误信息输出:

Err 1: line 2: error E16: colon expected

Err 2: line 3: error E19: of keyword expected

2 ERROR in total

### 程序五(正确)

var a1,a2,a3,a4,a5:integer;

g :char;

procedure p1;

var l1,l2,l3,l4,l5:integer;

g2:integer;

procedure p2;

procedure p3(var tmp:integer);

procedure p4(var tmp:integer);

var tmp4: integer;

procedure p5(var tmp:integer);

begin

tmp:= 1024;

a1:=256;

write("p5 ")

end;

begin

p5(tmp)

end;

begin

p4(tmp)

end;

begin

p3(l1)

end;

begin

p2;

write(l1);

end;

begin

a1:=0;

write(" ",a1);

p1;

write(" ",a1)

end.

输出:

0p5

1024

256

### 程序六(错误)

var a1,a2,a3,a4,a5:integer;

g :char;

procedure p1;

var l1,l2,l3,l4,l5:integer;

g2:integer;

procedure p2;

procedure p3(var tmp:integer)

procedure p4(var tmp:integer);

tmp4: integer;

procedure p5(var tmp:integer);

begin

tmp:= 1024;

a1:=256;

write("p5 ")

end;

begin

p5(tmp)

end;

begin

p4(tmp)

end;

begin

p3(l1)

end;

begin

p2;

write(l1);

end;

begin

a1:=0;

write(" ",a1);

p1;

write(" ",a1)

end.

错误信息输出:

Err 1: line 8: error E5: semicolon expected

Err 2: line 9: error E42: begin keyword expected

Err 3: line 9: error E28: undefined identifier. (tmp4)

Err 4: line 10: error E43: end keyword expected

Err 5: line 10: error E5: semicolon expected

Err 6: line 20: error E28: undefined identifier. (p4)

Err 7: line 34: error E43: end keyword expected

Err 8: line 34: error E5: semicolon expected

Err 9: line 34: error E42: begin keyword expected

Err 10: line 34: error E43: end keyword expected

Err 11: line 34: error E5: semicolon expected

Err 12: line 34: error E42: begin keyword expected

Err 13: line 34: error E43: end keyword expected

13 ERROR in total

### 程序七(正确)

procedure pri;

function f(n:integer):integer;

begin

if n<3 then

f:=n

else

f:=n\*f(n-1)

end;

begin

write(f(5))

end;

begin

pri

end.

输出:

120

### 程序八(错误)

procedure pri;

function f(n:integer):integer;

begin

if n<<3 then

f:=n

else

f:=n\*f(n-1)

end;

begin

write(f(5))

end;

begin

pri

end

错误信息输出:

Err 1: line 4: error E27: value expected

Err 2: line 4: error E36: then keyword expected

Err 3: line 4: error E43: end keyword expected

Err 4: line 4: error E5: semicolon expected

Err 5: line 4: error E42: begin keyword expected

Err 6: line 4: error E43: end keyword expected

Err 7: line 4: error E5: semicolon expected

Err 8: line 4: error E42: begin keyword expected

Err 9: line 4: error E43: end keyword expected

Err 10: line 4: error E0: period expected

Err 11: line 4: error E2: file should be end

11 ERROR in total

### 程序九(正确)

var i:integer;

procedure h1(var i:integer);

procedure h2(var i:integer);

procedure h3(var i:integer);

begin

i:=9

end;

begin

h3(i)

end;

begin

h2(i)

end;

begin

h1(i);

write(i)

end.

输出:

9

### 程序十(错误)

var i:integer;

procedure h1(var i integer);

procedure h2(var i:integer);

procedure h3(var i:integer);

begin

i=9

end;

begin

h3(i)

end;

begin

h2(i)

end;

begin

h1(i);

write(i)

end.

错误信息输出:

Err 1: line 2: error E16: colon expected

Err 2: line 6: error E26: symbol = expected. Here is ecpected to be an assigning

statement

Err 3: line 17: error E43: end keyword expected

Err 4: line 17: error E5: semicolon expected

Err 5: line 17: error E42: begin keyword expected

Err 6: line 17: error E43: end keyword expected

Err 7: line 17: error E5: semicolon expected

Err 8: line 17: error E42: begin keyword expected

Err 9: line 17: error E43: end keyword expected

Err 10: line 17: error E5: semicolon expected

Err 11: line 17: error E42: begin keyword expected

Err 12: line 17: error E43: end keyword expected

12 ERROR in total

## 测试结果分析

以上测试程序已经覆盖了所有的语法成分．对错误的程序，由于跳过一些语句的原因，可能会报出来很多个错误．但是应该报出来的错误都报出来了，不会少报．

# 总结感想

## 学会了一门新的语言：C++

完成这次课程设计之前，从没有写过一行C++代码，这次正好通过写这个课程设计，熟练掌握了C++语言．

## 初步认识了x86

做这次课设之前，一直都是写MIPS汇编，这次我给自己增加了难度，完成以前没有接触过的x86汇编．还是有所收获的．

## 美中不足

做到目标代码生成这步时，由于有非常多的小细节之前没有想到，导致一直到测试三进行时，我的代码还在处理Debug状态，迟迟没有开始写代码优化．这也说明了自己的能力有所欠缺．不过在Debug的过程中，解决一个一个的小问题，还是有非常多的感悟的．