《编译原理》课程设计文档

学号：15061140

姓名：刘智辉

2018年1月13日

## 一．需求说明

### 1．文法说明

文法无需改写。

<加法运算符>::=+|-

<乘法运算符>::=\*|/

<关系运算符>::=<|<=|>|>=|!=|==

<字母>::=\_|a|...|z|A|...|Z

<数字>::=0|<非零数字>

<非零数字>::=1|...|9

<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}"

<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>

<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>}

<无符号整数>::=<非零数字>{<数字>}

<整数>::=[+|-]<无符号整数>|0

<标识符>::=<字母>{<字母>|<数字>}

<声明头部>::=int<标识符>|char<标识符>

<变量说明>::=<变量定义>;{<变量定义>;}

<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>‘]’)}

<类型标识符>::=int|char

<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>‘}’

<复合语句>::=[<常量说明>][<变量说明>]<语句列>

<参数表>::=<类型标识符><标识符>{,<类型标识符><标识符>}|<空>

<主函数>::=void main’(‘‘)’‘{‘<复合语句>‘}’

<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>‘]’|<整数>|<字符>|<有返回值函数调用语句>|’(‘<表达式>‘)’

<语句>::=<条件语句>|<循环语句>|<情况语句>|’{‘<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>;

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

<条件语句>::=if‘(‘<条件>‘)’<语句>

<条件>::=<表达式><关系运算符><表达式>|<表达式>//表达式为0条件为假，否则为真

<循环语句>::=do<语句>while‘(‘<条件>‘)’

<常量>::=<整数>|<字符>

<情况语句>::=switch‘(‘<表达式>‘)’‘{‘<情况表>‘}’

<情况表>::=<情况子语句>{<情况子语句>}

<情况子语句>::=case<常量>:<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>‘)’

<值参数表>::=<表达式>{,<表达式>}|<空>

<语句列>::={<语句>}

<读语句>::=scanf‘(‘<标识符>{,<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>‘)’]

附加说明:

（1）char类型的表达式，用字符的ASCII码对应的整数参加运算，在写语句中输出字符

（2）标识符不区分大小写字母

（3）写语句中的字符串原样输出

（4）数组的下标从0开始

（5）情况语句中，switch后面的表达式和case后面的常量只允许出现int和char类型；每个情况子语句执行完毕后，不继续执行后面的情况子语句

### 2．目标代码说明

目标代码为MIPS指令，包括伪指令，使用指令如下：

jal label

j label

jr rs

la label

li rt, imm

lw rt, rs, offset

sw rt, rs, offset

move rt, rs

add rd, rs, rt

addi rt, rs, imm

sub rd, rs, rt

subi rt, rs, imm

mul rd, rs, rt

div rt, rs

mflo rs

beq rt, rs, label

bne rt, rs, label

slt rd, rs, rt

sle rd, rs, rt

sgt rd, rs, rt

sge rd, rs, rt

seq rd, rs, rt

sne rd, rs, rt

syscall

### 3.优化方案\*

局部公共子表达式消除——DAG图

寄存器分配——引用计数法

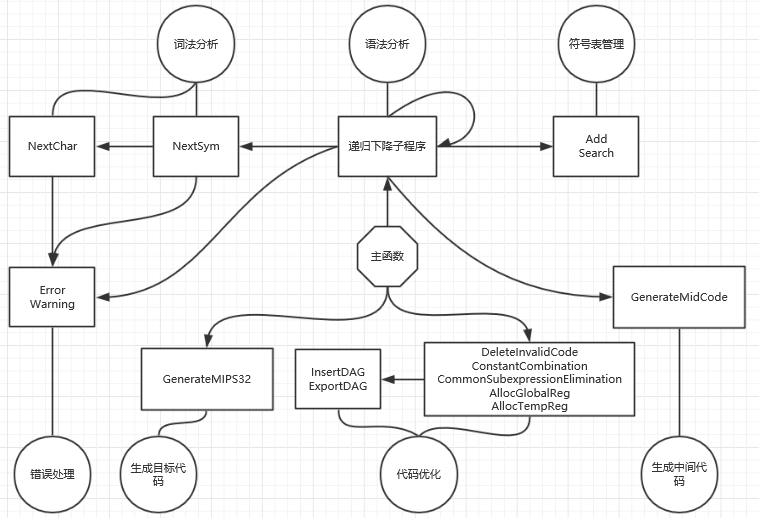
常数合并

无效代码删除

无用标识符删除

## 二．详细设计

### 1．程序结构



**图1 程序结构**

### 2．类/方法/函数功能

【词法分析】

主要函数

NextChar()从源文件中获取一个字符

NextSymbol()获取一个单词或符号的类别码及其值

【语法分析】

主要函数

Program()程序分析子程序

ConstDeclare(bool isglobal)常量声明分析子程序

ConstDefine(bool isglobal)常量定义分析子程序

VarDeclare(bool isglobal)变量声明分析子程序

VarDefine(bool isglobal)变量定义分析子程序

FuncDeclare()函数/过程声明分析子程序

ProcDefine()过程定义分析子程序

FuncDefine()函数定义分析子程序

ParaTable()参数表分析子程序

MainDefine()主函数定义分析子程序

CompoundStatement()复合语句分析子程序

Statements()语句列分析子程序

Statement()语句分析子程序

IfStatement()条件语句分析子程序

DoWhileStatement()循环语句分析子程序

SwitchStatement()情况语句分析子程序

AssignStatement()赋值语句分析子程序

ScanfStatement()读语句分析子程序

PrintfStatement()写语句分析子程序

ReturnStatement()返回语句分析子程序

FuncCallStatement()函数调用分析子程序

ProcCallStatement()过程调用分析子程序

Condition()条件分析子程序

SwitchTable()情况表分析子程序

ValueParaTable()值参数表分析子程序

Expression()表达式分析子程序

Term()项分析子程序

Factor()因子分析子程序

Constant()常量分析子程序

Integer()整数分析子程序

Character()字符分析子程序

【错误处理】

主要函数

Error()打印错误信息

Warning()打印警告信息

【符号表管理】

AddConst(char name[], int type,int value,bool isglobal)常量入表

AddArray(char name[], int type,int size,bool isglobal)数组入表

AddVar(char name[],int type,bool isglobal)变量入表

AddFunc(char name[],int type,int size,unsigned int paratype)函数入表

AddProc(char name[],int size,unsigned int paratype)过程入表

AddPara(char name[],int type)参数入表

Search(char name[],int code)查表

【生成中间代码】

主要函数

GenerateMidCode(int op, char op1[], char op2[], char res[])递归下降分析程序语义分析后生成中间代码

【代码优化】

主要函数

DeleteInvalidCode()删除中间代码中无效代码

ConstantCombination()常量运算结果合并

ClearUnusedVar()删除无用标识符

CommonSubexpressionElimination()消除公共子表达式

AllocGlobalReg()分配全局寄存器

AllocTempReg()分配临时寄存器

【DAG图】

主要函数

InsertDAG(Quadruple code)将一条中间代码插入DAG图中

ExportDAG()将DAG图导出生成中间代码

NewNode(char\* name, int op)在DAG图中新增一个结点

UpdateTab(char\* name, int index)更新结点表信息

SearchIndex(char\* name)查找结点标号

Link(int root, int lchild, int rchild)连接结点

【生成目标代码】

GenerateMIPS32()根据中间代码生成目标代码

AllocGlobal(Identity\* ident, char\* name, char\* value)全局变量分配空间

AllocLocal(Identity \*&ident)局部变量分配空间

### 3．调用依赖关系

参见程序结构

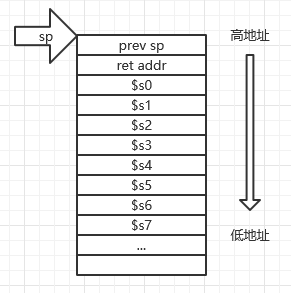
### 4．符号表管理方案

符号表中符号数量不太大，采用无序符号表比有序符号表实现更简单、操作更便捷。由于查表直接比较字符串复杂度较高，对符号名取Hash，在查找时大大减少字符串的直接比较，性价比较高。

此外，在经过语义分析需要插入临时常量时可以查找已有等值常量，返回已有常量。

### 5．存储分配方案

采用动态存储方式，扩充c0文法没有分程序，故运行栈中没有display区。运行栈结构如图2：



**图2 运行栈结构**

同时为了便于函数调用时参数传递，设$gp为参数栈栈顶，从低地址向高地址增长。

### 6.解释执行程序\*

扩充c0文法题采用编译执行，不采用解释执行

### 7.四元式设计\*

|  |  |
| --- | --- |
| 源代码 | 中间代码 |
| const int a = 10, b = -4; | const,int,10,a  const,int,-4,b |
| const char a = ‘a’, z = ‘z’; | const,char,97,a  const,char122z |
| int i, j; | var,int,,i  var,int,,j |
| char ch; | var,char,,ch |
| int t[100]; | arr,int,100,t |
| void test(int x, int y){} | proc,,,test  para,int,,x  para,int,,y |
| int add(char ch, int off){} | func,int,,add  para,char,,ch  para,int,,off |
| if(a < b)  c = d; | <,a,b,t0  jz,t0,l0,  =,d,,c  label,l0,, |
| do{  i = i + 1;  }while(i < 100); | label,l0,,  +,i,1,i  <,i,100,t0  jnz,t0,l0, |
| switch(a){  case 0 : b = 1;  case 1 : b = a \* 2;  } | ==,a,0,t0  jz,t0,l0,  =,1,,b  j,l2,,  label,l0,,  ==,a,1,t0  jz,t0,l1,  \*,a,2,b  j,l2,,  label,l1,,  label,l2,, |
| a[i] = b[i] + c \* d; | larr,b,i,t0  \*,c,d,t1  +,t0,t1,t2  sarr,t2,i,a |
| scanf(a, b, c); | scan,,,a  scan,,,b  scan,,,c |
| printf(“helloworld!”); | prints,“helloworld”,, |
| printf(“Number:”, a + b); | prints,“Number:”,,  +,a,b,t0  printi,t0,, |
| const char result = ‘a’;  printf(result); | const,char,97,result  printc,result,, |
| return(NULL); | ret,null,, |
| return; | ret,,, |
| error(FILE\_NOT\_EXIST); | val,FILE\_NOT\_EXIST,,  call,error,, |
| a = gcd(24, 36); | val,24,,  val,36,,  call,gcd,,t0  =,t0,,a |

**表1 四元式设计举例**

### 8.目标代码生成方案\*

目标代码依据中间代码生成，无需数据结构直接写入输出文件。

关键点：

1. 函数或过程声明中间代码处理：设置函数或过程的入口标签；参数栈栈顶按参数个数退栈；保存退栈信息，即保存$fp和$ra；运行栈中全局寄存器内存分配；运行栈中未分配寄存器局部变量和临时变量内存分配。
2. 参数传递中间代码处理：参数传递有关的中间代码操作符为para和val。para表示取参数，即从参数栈中取出参数至相应内存或寄存器；val表示压参数，即将参数压入参数栈中。
3. 函数或过程调用中间代码处理：压栈之前保存全局寄存器；压栈，即移动帧指针和栈指针；跳转并链接至相应函数或过程；函数调用时则从$v0取返回值至相应内存或寄存器。
4. 存取数组中间代码处理：获取数组元素基地址；根据数组下标乘四获取偏移量；由基地址和偏移量得到数组元素地址进行存取数组。
5. 函数或过程返回中间代码处理：函数返回时先将返回值存至$v0，后续操作和过程相同；退栈，先取返回地址至$ra再取栈指针至$sp；恢复全局寄存器。
6. 算术逻辑运算中间代码处理：由于设计符号表在语法分析完成后无法再扩充，常数合并仅限于操作结果为临时变量的中间代码，故此处仍可进行常数合并；无法常数合并则正常进行算术逻辑运算。

### 9.优化方案\*

1. 无效代码删除

无需数据结构。

可删除无效代码的两种情况：

1. ret和proc、func、label之间的代码为死代码，因为没有可以进入这段代码的入口。
2. 多个label可合并为单个label。

死代码删除可以略微减少代码内存。

1. 常数合并

无需数据结构。

常数合并有两个方面：

1. 算术逻辑运算的两个操作数均为常量时，若操作结果为临时变量，则该临时变量可被置为临时常量（因为只被定义一次）且值为两操作数的运算结果；若操作结果不为临时变量，则在生成目标代码时将值嵌入目标代码中而不是分配内存存储常量。
2. 当可查到等值常量时可用该常量替换。

常数合并可以略微减少指令数和运行内存。

1. 无用标识符删除

无需数据结构。

在进行消除公共子表达式之后，可以删除不再被使用的临时变量，也可以删除源程序中声明却未使用的变量。

无用标识符可以略微减少运行内存。

1. 局部公共子表达式消除

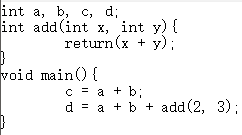
采用DAG图。

需要解决的问题：

1. 基本块该如何划分，只将连续的算术逻辑运算作为基本块处理还是按照跳转入口和跳转划分处理。
2. 若基本块以跳转入口和跳转划分，则基本块内夹杂着各种非算术逻辑运算中间代码，这些中间代码需要保证其相对顺序。
3. 重新从DAG图导出到中间代码可能会碰到一些问题。

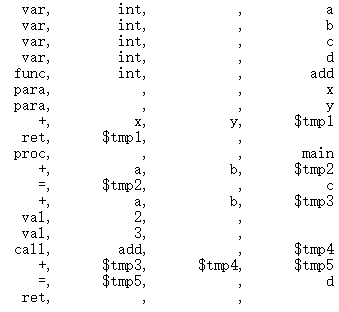
关键点：

1. 如果仅以连续的算术逻辑运算作为基本块处理，可以消除的局部子表达式相当有限，收效甚微（不是刻意制造可消除的情况，局部消除公共子表达式的情况本身少见）。划分基本块时以跳转入口和非函数调用跳转划分，至于为什么这么划分，原因如下：

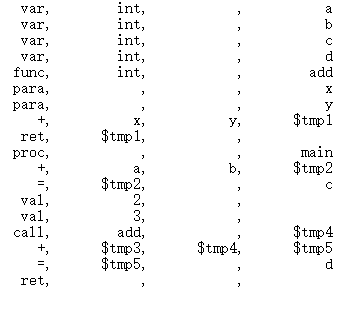


**图3 源代码**

遇到图3源代码中的情况时，若函数调用处划分基本块，那么在处理函数调用前会导出DAG图，将会产生像图5一样的中间代码。



**图4 优化前中间代码**



**图5 优化后中间代码**

对比图4和图5发现优化导致了错误，因为函数调用之前的DAG图仍未$tmp2和$tmp3可以复用而消除了$tmp3，但是函数调用之后$tmp3仍需要复用$tmp2，函数调用处划分基本块会导致这个棘手的问题，因此不在此处划分基本块。

1. 合理改变算术逻辑运算顺序不会改变程序逻辑，但是非算术逻辑运算改变顺序会出现问题。DAG结点生成的顺序就是按照中间代码的顺序生成的，利用这一点就可以保证非算术逻辑运算的相对顺序。
2. 仅靠结点下标顺序重新导出中间代码不太合理，而采用教材中所述启发式算法又不能保证非算术逻辑运算的相对顺序。改变策略，修改成如下启发式算法：
3. 初始队列为空。
4. 从下标最大向下标最小的结点中找到未加入队列且无父结点的非叶子结点p。
5. 将结点p加入队列。若其左子结点为非叶子结点且操作符为算术逻辑运算，将其左子结点置为结点p并重复步骤3，否则重复步骤2直至DAG图中非叶子结点全部加入队列。
6. 队列结点逆序即是正确的中间代码导出顺序。
7. 在DAG图还可以寻找某些特殊结构做窥孔优化。

局部公共子表达式可以少量减少中间代码，从而减少指令数。

1. 寄存器分配

无需数据结构。

采用引用计数法，在语法分析阶段统计权重。以函数或过程为单位分配全局寄存器，全局寄存器仅分配给局部非临时变量；以基本块为单位分配临时寄存器，临时寄存器仅分配给临时变量。其中，s0-s7作为全局寄存器，a1-t7作为临时寄存器。

尽管这种方式实现简单易行，但同样会遇到问题：在消除局部公共子表达式的时候就遇到过了，函数调用将一个基本块分成两部分，从前半部分进入一个基本块再回到后半部分，又不能将这两部分分成两个基本块，因为这两部分可能存在临时变量的联系。本身以基本块来分配临时寄存器是没有问题的，因为临时变量的作用域仅在基本块内，出了基本块就再也不会用到。但是函数调用导致了这种特殊性。

想到两种解决方案：第一种是像全局寄存器一样，对于函数返回后还会用到的临时寄存器在函数调用时保存其值，返回后再恢复；第二种则是当基本块内存在函数调用时仅给最后一个函数调用之后产生的临时变量分配临时寄存器，之前的一律不分配临时寄存器。

寄存器分配可以将大量读写内存指令转变为算术逻辑指令，可以大幅度地减少指令运行时间。

### 10.出错处理

出错处理包含错误和警告，对于错误，是很可能导致程序逻辑错误甚至无法编译的严重问题；对于警告，是可能导致程序逻辑错误但是不影响编译的轻微问题。

出错处理方案：

词法、语法、语义分析出现错误时调用Error()并传递错误信息；依据错误信息打印出错信息和出错位置；通过出错信息和位置跳过部分源代码；跳过源代码依据当前调用的语法分析子程序而定，通过设定跳过符号的终止符号集进行跳读。警告同理但无需跳读。

出错信息：

目前共38种错误和6种警告。

|  |  |  |
| --- | --- | --- |
| 错误信息 | 错误含义 | 错误类型 |
| FILE\_NOT\_EXIST\_ERROR | 源文件不存在 |  |
| PROGRAM\_INCOMPLETE\_ERROR | 程序不完整 | 语法错误 |
| INVALID\_CHARACTER\_ERROR | 无效字符 | 语法错误 |
| CHARACTER\_LENGTH\_ERROR | 字符长度不符 | 语法错误 |
| INVALID\_SYMBOL\_ERROR | 无效符号 | 语法错误 |
| MISSING\_SEMICOLON\_ERROR | 缺失分号 | 语法错误 |
| MISSING\_ASSIGN\_ERROR | 缺失赋值符 | 语法错误 |
| MISSING\_IDENTITY\_ERROR | 缺失标识符 | 语法错误 |
| NOT\_CHARACTER\_ERROR | 非字符 | 语法错误 |
| MISSING\_TYPE\_ERROR | 缺失数据类型 | 语法错误 |
| MISSING\_RIGHT\_BRACKET\_ERROR | 缺失右方括号 | 语法错误 |
| MISSING\_INDEX\_ERROR | 缺失数组下标 | 语法错误 |
| SYNTAX\_ERROR | 语法有误 | 语法错误 |
| MISSING\_RIGHT\_BRACE\_ERROR | 缺失右花括号 | 语法错误 |
| MISSING\_LEFT\_BRACE\_ERROR | 缺失左花括号 | 语法错误 |
| MISSING\_RIGHT\_PARENTHESIS\_ERROR | 缺失右圆括号 | 语法错误 |
| MISSING\_LEFT\_PARENTHESIS\_ERROR | 缺失左圆括号 | 语法错误 |
| PARAMETER\_NUM\_OUTOFRANGE\_ERROR | 参数个数溢出 | 语义错误 |
| MISSING\_WHILE\_ERROR | 缺失while关键字 | 语法错误 |
| MISSING\_COLON\_ERROR | 缺失冒号 | 语法错误 |
| MISSING\_CASE\_ERROR | 缺失case关键字 | 语法错误 |
| EMPTY\_FACTOR\_ERROR | 因子为空 | 语法错误 |
| NOT\_INTEGER\_ERROR | 非整数 | 语法错误 |
| NOT\_CONSTANT\_ERROR | 非常量 | 语法错误 |
| DUPLICATE\_DEFINATION\_ERROR | 重复定义标识符 | 语义错误 |
| UNDEFINED\_IDENTITY\_ERROR | 未定义标识符 | 语义错误 |
| CANNOT\_ASSIGN\_IDENTITY\_ERROR | 不能赋值的标识符 | 语义错误 |
| NOT\_ARRAY\_ERROR | 非数组 | 语义错误 |
| NOT\_FUNCTION\_ERROR | 非函数 | 语义错误 |
| INDEX\_TYPE\_ERROR | 数组下标类型错误 | 语义错误 |
| INDEX\_OUTOFRANGE\_ERROR | 下标越界 | 语义错误 |
| CANNOT\_GETVALUE\_IDENTITY\_ERROR | 不能取值的标识符 | 语义错误 |
| UNMATCHED\_TYPE\_ERROR | 数据类型不匹配 | 语义错误 |
| UNMATCHED\_PARA\_NUM\_ERROR | 参数数量不匹配 | 语义错误 |
| UNMATCHED\_PARA\_TYPE\_ERROR | 参数类型不匹配 | 语义错误 |
| MISSING\_RETURN\_VALUE\_ERROR | 缺失返回值 | 语义错误 |
| FUNC\_RETURN\_VALUE\_ERROR | 函数返回空值 | 语义错误 |
| UNMATCHED\_RETURN\_TYPE\_ERROR | 函数返回类型不匹配 | 语义错误 |

**表2 错误说明**

|  |  |
| --- | --- |
| 警告信息 | 警告含义 |
| LEADING\_ZERO\_WARNING | 整数存在前导零 |
| ASSIGN\_INTTOCHAR\_WARNING | 整型赋值字符型 |
| MAY\_MISSING\_RETURN\_VALUE\_WARNING | 可能缺失返回值 |
| PROC\_RETURN\_VALUE\_WARNING | 过程返回非空值 |
| DUPLICATE\_CASE\_WARNING | 重复定义情况 |
| UNUSED\_IDENTITY\_WARNING | 未使用的标识符 |

**表3 警告说明**

## 三．操作说明

### 1．运行环境

Visual Studio 2010、Mars4.5

### 2．操作步骤

用Visual Studio打开项目开始执行，输入源代码路径进行编译，编译记录及编译结果保存在文件，编译信息输出到控制台。

词法分析、语法分析记录输出至record.txt

语法分析后符号表基本信息输出至table.txt

优化前中间代码输出至midcode\_origin.txt

优化前目标代码输出至midcode.txt

优化后中间代码输出至targetcode\_origin.asm

优化后目标代码输出至targetcode.asm

错误警告信息输出至控制台

用Mars运行目标代码

## 四．测试报告

### 1．测试程序及测试结果

> 代表输出

< 代表输入

【测试程序1】【正确】

> 1:passed

> 2:passed

< 5

> 3:passed

【测试程序2】【正确】

< r 999

> 999=3^3\*37

< t 50

> Prime numbers: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

【测试程序3】【正确】

< 24

< 36

> 12

< 105

< 98

> 7

【测试程序4】【正确】

> -5

> 0

> 6

> 1

> passed

【测试程序5】【正确】

> test passed

> 55

【测试程序6】【错误】

【测试程序7】【错误】

【测试程序8】【错误】

【测试程序9】【错误】

【测试程序10】【错误】

### 2．测试结果分析

【测试程序1】

覆盖所有语法，主要测试各个简单语法的正确性。

【测试程序2】

覆盖大部分语法，主要测试函数调用、递归、循环、情况语句等，可输出质数表或分解质因数。

【测试程序3】

测试函数递归、且参数传递复杂的情况，输出两数的最大公因数。

【测试程序4】

测试比较特殊的情况的正确性。

【测试程序5】

测试优化对于结果的正确性。

【测试程序6】

【测试程序7】

【测试程序8】

覆盖大部分语法错误，测试错误信息的准确性。

【测试程序9】

【测试程序10】

覆盖大部分语义错误，测试错误信息的准确性。