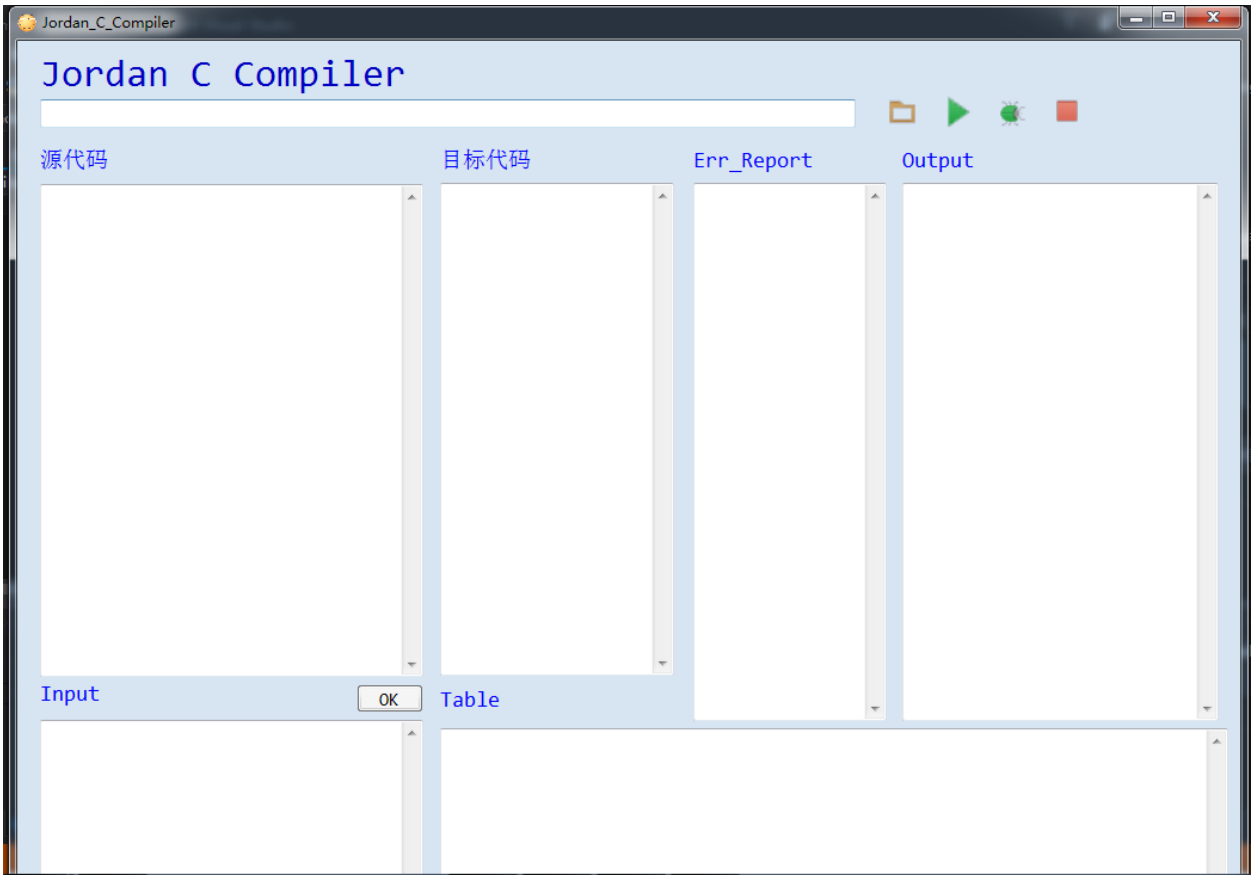
Jordan_C_Compiler
软件设计说明书

作者:乔丹

版本:v1.0

日期:2016/12/6

1. 介绍



本编译器为 Small_C 语言的扩展编译器，可编译扩展后的 Small_C 语言，完成基本的词法分析、语法分析、出错处理、中间代码生成和解释程序功能，并完成了取模、异或、块注释处理、行注释处理、odd、while-do 语句、do-while 语句、switch-case 语句、for 语句、bool 类型、常量、continue、break、exit 等扩展功能。源代码读入方式是通过图形用户界面选中源代码 txt 文件，（若程序需要输入数据，则需在图形用户界面的 Input 框中输入需要的数据，点击”OK”按钮）然后点击编译运行按钮，编译后可以 GUI 对应各框内获得错误报告、目标代码、符号表、输出结果，同时，错误报告、目标代码、符号表、输出结果也被存入本地的 txt 文件中，路径需在 JCC_UI.cs 和 DLLtest.cpp 中调整。

2. 编译器系统结构

2.1 编译器

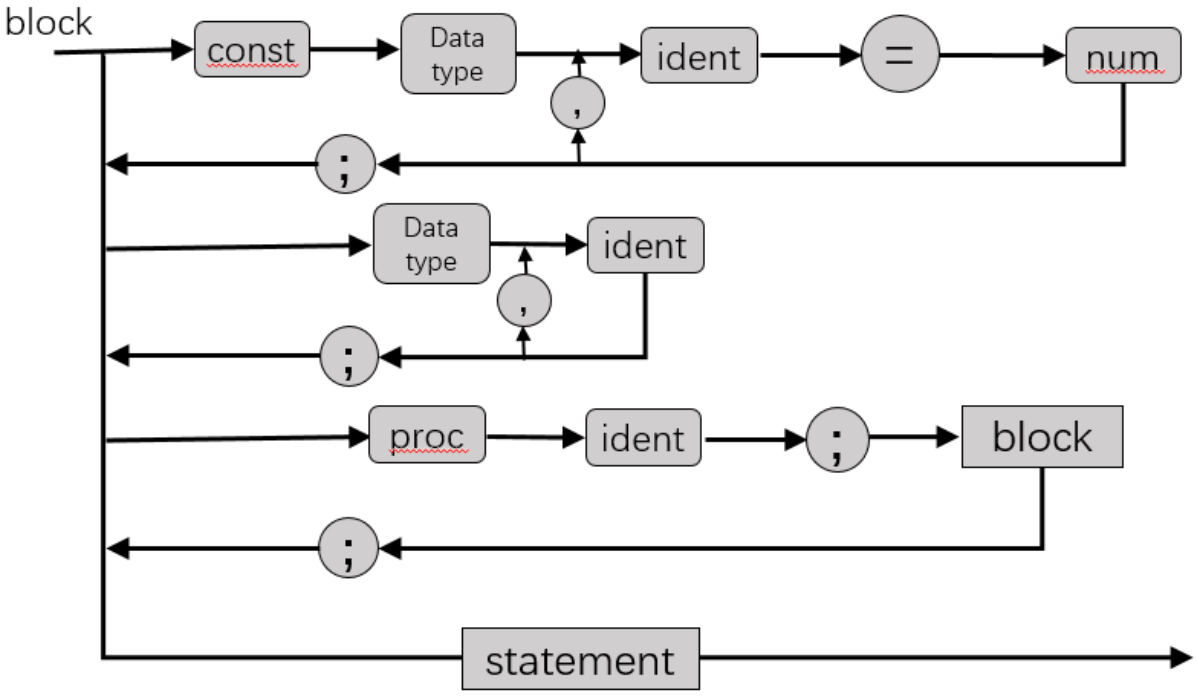
2.1.1 Small 语言语法图

<program>::=<block>.

语法扩展说明：program 由 block 构成，program 结束符为“.”

<block>::=
const<datatype><ident>=<num>{,<ident>=<num>};
|<datatype><ident>{,<ident>};
|{<proc><ident>;<block>;}

语法扩展说明：由于加入了 bool 型的数据类型，因此添加了变量声明语句。也添加了扩展常量的语句。block 内部语法设计参考了 PL/0 语言的语法。



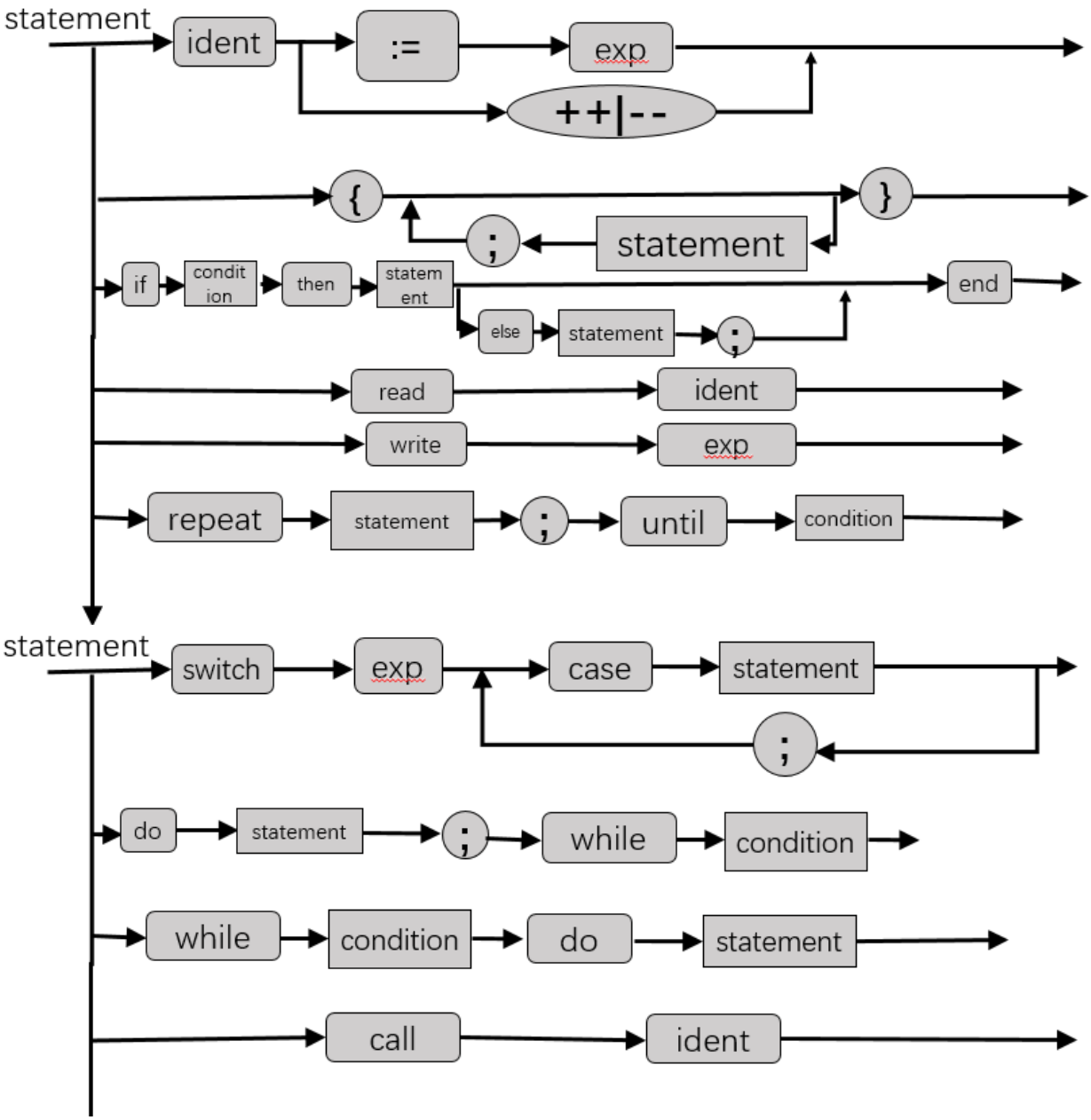
<statement>::=
<assign-stmt>
|<stmt-sqnc>
|<if-stmt>

|<read-stmt>
|<write-stmt>
|<repeat-stmt>
|<while-do-stmt>
|<do-while-stmt>
|<switch-case-stmt>
|<for-stmt>
|<call-stmt>

<assign-stmt>::=<ident>[**:=**<exp>|**++**|**--**]
<stmt-sqnc>::=**{**<statement>;**}**
<if-stmt>::= **if**<condition>then<statement>
{**else** <statement>**;**}**end**
<read-stmt>::= **read**<ident>
<write-stmt>::= **write**<exp>
<repeat-stmt>::= **repeat**<statement>**;until**<condition>
<while-do-stmt>::= **while**<condition>**do**<statement>

<do-while-stmt>::= **do**<statement>**; while**<condition>
<switch-case-stmt>::= **switch**<exp>**case**<ident|num>
<for-stmt>::=**for(**<ident>**:=**<exp>**;**<condition>**;**
<ident>[**:=**<exp>|**++**|**--**]) <statement>
<call-stmt>::= **call**<ident>

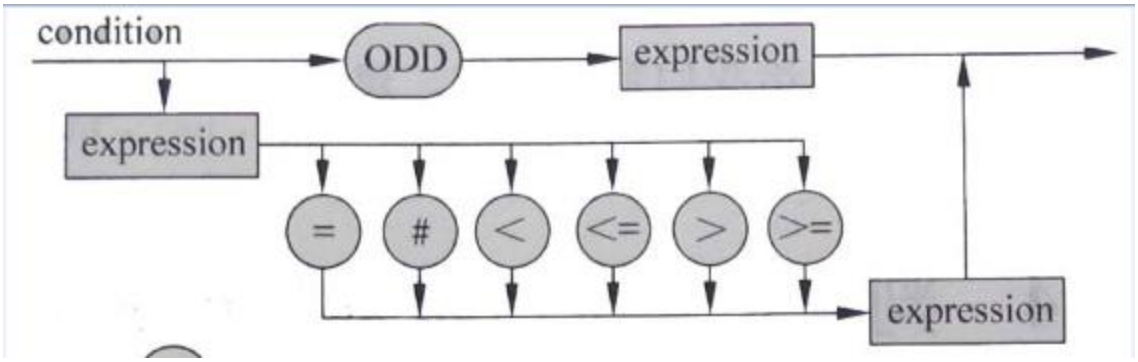
语法扩展说明：规定所有句子后继符号必须为”；”，删除句子序列的定义，增加复合语句的定义。由于扩展了 while-do 语句、for 语句、switch-case 语句、do-while 语句等，因此增加了相应的语法规则。



<condition>::=**odd**<exp>|<exp><cmp-op><exp>
<comparison-op>::=>|<|>=|<=|==|!=
<exp>::=<add-op><mul-term><or-op><xor-term>|<xor-term><or-op><xor-term>
<or-op>::=**|**
<xor-term>::=<and-term><xor-op><and-term>
<xor-op>::=**^**
<and-term>::=<add-term><and-op><add-term>
<and-op>::=&

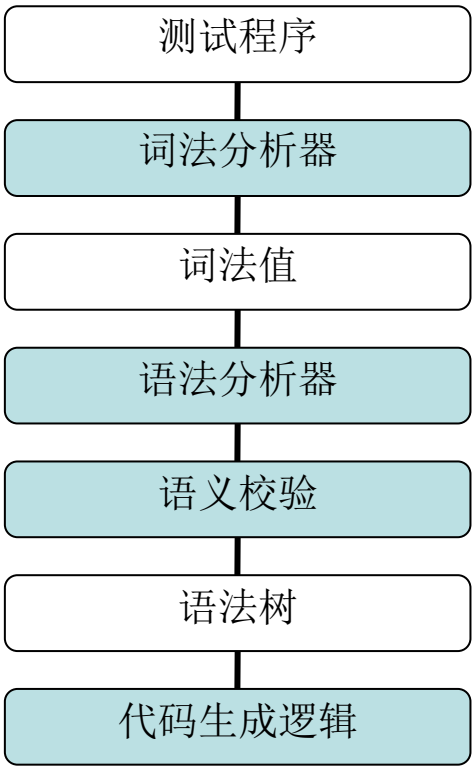
<add-term>::=<mul-term><add-op><mul-term>
<add-op>::=**+**|**-**
<mul-term>::=<not-term><mul-op><not-term>
<mul-op>::=*****|**/**|**%**
<not-term>::=**!**<factor>|<factor>
<factor>::=**(**<exp>**)**|<number>|<identifier>
<number>::=<digit>|<number><digit>
<digit>::=**0**|**1**|**2**|**3**|**4**|**5**|**6**|**7**|**8**|**9**
<identifier> ::=<letter>|<identifier><letter>|<identifier><digit>
<letter> ::=a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

语法扩展说明：为了扩展运算符，因此对表达式的下级项多了多层的扩展，扩展规则遵循 c 语言中运算符优先级。但是有一点失误是未将比较运算符加入表达式中。这一点还有待改进。

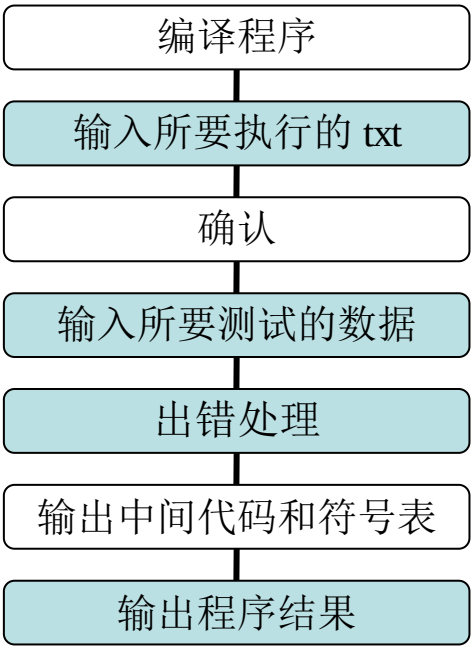


2.1.2 判断是否符合两条限制规则

2.1.3 过程调用相关图



2.1.4 程序总体结构



2.1.5 语法出错表定义

error(1):常量声明时的赋值符号写为了变量赋值符号（把=写成了:=）

error(2): 常量声明中的=后和类型不匹配

error(3): 常量声明时应直接赋值
error(4):数据类型或 proc 后不是标识符
error(5): 常量或变量声明的结尾漏掉了分号
error(6):proc 后继符号不合法
error(7):声明语句后继符号不合法
error(8):block 后继符号不合法
error(9):主函数结尾没有”.”
error(10):复合语句中的语句末尾缺少分号
error(11):标识符未声明,或过程名未找到
error(12):赋值语句左部不是变量
error(13):未检测到赋值符号或自增自减符号
error(14): call 后应为标识符
error(15): call 后标识符类型应为过程
error(16):if 语句后 缺少 then
error(17):if 语句末尾缺少 end
error(18):while 语句后缺少 do
error(19):语句结尾后继符号不合法
error(20):条件处理的第一个表达式后应该为关系运算符
error(21):表达式中的标识符不能为过程
error(23):因子结束的后继符号不合法
error(22):以左括号开头的表达式缺少右括号
error(26): 缺少右大括号
error(27):for 后面格式错误，应该是左括号
error(28):for 的第一部分应该是赋值语句，开头是标识符
error(29):for 后面括号里语句的前两句缺少分号
error(30):数字位数太多
error(31):常数越界
error(32):函数调用嵌套层数太多
error(34):常量变量声明时未定义数据类型
error(31):常数越界
error(35):read 语句括号中的标识符应该是声明过的变量
error(41):当前代码块不应出现 break
error(42):当前代码块不应出现 continue
error(43):当前循环或 switch 层 break 语句数量太多

error(44):当前循环 continue 语句数量太多

error(45):for 括号内第三部分应以标识符开头

error(46):for 第三部分应是自增自减或赋值运算

error(47)for 语句格式错误，缺少右括号

error(48):case 后应该是标识符或数字

error(49)repeat 语句缺少 until

error(50):do while 语句缺少 while

error(54):case 的标识符或数字后缺少冒号

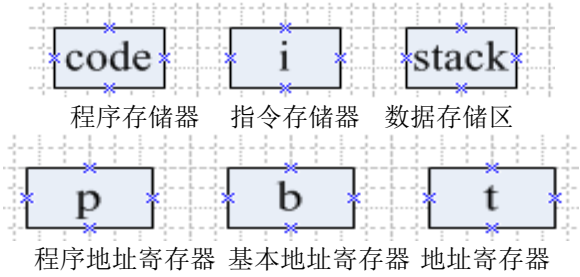
error(56)switch 后面缺少左括号

error(57)switch 的表达式后缺少右括号

error(60):exit 语句数量太多

2.2 虚拟机

2.2.1 虚拟机组织结构



2.2.2 虚拟机指令格式

Instruction(fct,l,a)

fct 虚拟机指令代码，l 为引用层与声明层层次差，a 为虚拟机指令对应操作数

2.2.3 虚拟机指令系统及其解释

lit 指令： 把一个常数置入栈顶。

lit 0,a： 将常量 a 置入栈顶

lod 指令：把一个变量置入栈顶.

lod l,a：1 决定了变量在过程段的起始地址；a 是变量在过程段的位移地址；L 和 a 决定了变量在数据栈的绝对地址，从而将变量置入栈顶。

sto 指令：从栈顶将一个数置入变量单元。

sto l,a：l 决定了变量在过程段的起始地址；a 是变量在过程段的位移地址。从栈顶把一个数置入变量单元里。

jmp 指令： 无条件转移指令。

jmp 0,a ： 无条件转移到程序地址 a。

jpc 指令： 有条件转移指令。

jpc 0,a ：当栈顶数据 s(t)=0 时，转移到程序地址 a。

opr 指令： 一组算术和关系运算指令。

opr 0,a ： 参数 a 决定了对栈顶数据执行的具体算术或关系运算。

注：当第二个操作数等于 1 时，表示 bool 型变量的读写。

ini 指令： 预留数据存储位置。

ini 0,a：a 是为此过程预留的数据栈空间

3. 模块架构

- 3.1 翻译模块
- 3.2 解释模块
- 3.3 图形用户界面

4. 模块功能介绍

- 4.1 翻译模块：将测试程序代码逐行读入，通过词法分析、语法分析、语义分析几个阶段生成中间代码，并将中间代码存入 code 栈中。
- 4.2 解释模块：读取 code 栈中的中间代码，按照中间代码执行程序，直至结束。
- 4.3 图形用户界面：将编译程序整体封装为动态链接库，供图形用户界面调用。界面实现了选择输入文档、显示输入文档、读入程序需要的输入数据、编译并运行、显示错误报告、显示中间代码、显示输出结果等功能。使编译器更美观，使用更方便。

5. 模块接口

- 翻译模块与解释模块接口：gen 函数

```
void gen(enum fct x, int y, int z )
{
        if (cx >= cxmax)
```

```
        {
                printf("Program is too long!\n");      /* 生成的虚拟机代码程序过长 */
                exit(1);
        }
        if ( z >= amax)
        {
                printf("Displacement address is too big!\n"); /* 地址偏移越界 */
                exit(1);
        }
        code[cx].f = x;
        code[cx].l = y;
        code[cx].a = z;
        cx++;
}
```

- 翻译、解释模块与图形界面的接口

  DLLtest.dll

  ```
  void __stdcall solve()
  ```

  DLLtest.h

  DLLtest.def

  JCC_UI.cs

6. 全局数据结构、常量和变量

- 全局数据结构：

```
struct instruction
{
        enum fct f; /* 虚拟机代码指令 */
        int l;      /* 引用层与声明层的层次差 */
        int a;      /* 根据 f 的不同而不同 */
};
struct tablestruct
{
        char name[al];      /* 名字 */
        enum object kind;       /* 类型：const，var 或 proc */
```

```
        int type;           /*数据类型，1 为 int，2 为 boolean，仅 proc 不使用*/
        int val;            /* 数值，仅 const 使用 */
        bool bl;            /*bool 型数值，仅 const 使用*/
        int level;          /* 所处层，仅 const 不使用 */
        int adr;            /* 地址，仅 const 不使用 */
        int size;           /* 需要分配的数据区空间，仅 proc 使用 */
};
```

- 全局常量

```
public string fDirPath = "D:\\Program\\small_c_final\\";/*存储输出结果等文档的上级目录*/


std::string fileDir = "D:\\Program\\small_c_final\\";/*存储输出结果等文档的上级目录*/

#define norw 29        /* 保留字个数 */
#define txmax 100      /* 符号表容量 */
#define nmax 14        /* 数字的最大位数 */
#define al 10          /* 标识符的最大长度 */
#define maxerr 30      /* 允许的最多错误数 */
#define amax 2048      /* 地址上界*/
#define levmax 3       /* 最大允许过程嵌套声明层数*/
#define cxmax 200      /* 最多的虚拟机代码数 */
#define stacksize 500 /* 运行时数据栈元素最多为 500 个 */
#define breakmax 10
```

- 全局变量

```
char strtmp[al + 1];
int lastcase;           /*上一个 case 生成的条件跳转指令地址（待回填的指令）*/
int iniexit[breakmax];
int inicont[breakmax];
int inibreak[breakmax];/*初始存在的 break 对应无条件指令代码地址*/
bool listswitch;    /* 显示虚拟机代码与否 */
bool tableswitch;   /* 显示符号表与否 */
char ch;            /* 存放当前读取的字符，getch 使用 */
enum symbol sym;    /* 当前的符号 */
char id[al + 1];      /* 当前 ident，多出的一个字节用于存放 0 */
```

```
int num;              /* 当前 number */
int cur;             /*当前 bool 值*/
int cc, ll;           /* getch 使用的计数器，cc 表示当前字符(ch)的位置 */
int cx;               /* 虚拟机代码指针，取值范围[0, cxmax-1]*/
char line[81];        /* 读取行缓冲区 */
char a[al + 1];         /* 临时符号，多出的一个字节用于存放 0 */
struct instruction code[cxmax]; /* 存放虚拟机代码的数组 */
char word[norw][al];          /* 保留字 */
enum symbol wsym[norw];       /* 保留字对应的符号值 */
enum symbol ssym[256];        /* 单字符的符号值 */
char mnemonic[fctnum][5];     /* 虚拟机代码指令名称 */
bool declbegsys[symnum];      /* 表示声明开始的符号集合 */
bool statbegsys[symnum];      /* 表示语句开始的符号集合 */
bool facbegsys[symnum];       /* 表示因子开始的符号集合 */
int err;          /* 错误计数器 */
```

7. 函数原型

函数原型	void __stdcall solve()
参数描述	无
函数描述	原 c 文件中的 main 函数，现作为动态链接库和 GUI 接口函数
返回值	NULL

函数原型	void init()
参数描述	无
函数描述	初始化函数，做好编译程序开始前的准备工作
返回值	NULL

函数原型	int inset(int e, bool* s)
参数描述	e 为要查找的 symbol，s 为查找的集合
函数描述	在 s 中查找 e 是否存在，是返回 1，否则返回 0
返回值	0 或 1

函数原型	int addset(bool* sr, bool* s1, bool* s2, int n) int subset(bool* sr, bool* s1, bool* s2, int n) int mulset(bool* sr, bool* s1, bool* s2, int n)
参数描述	sr 目标集合，s1 集合运算的第一个集合，s2 集合运算的第二个集合，n 集合大小
函数描述	用数组实现的集合的运算
返回值	0

函数原型	void error(int n)
参数描述	n 错误类型
函数描述	出错处理程序，打印错误位置和错误类型
返回值	无

函数原型	void getch()
参数描述	无
函数描述	获得要从源程序中读入的下一个字符串
返回值	无

函数原型	void getsym()
参数描述	无
函数描述	词法分析，获取一个符号（以及处理行注释块注释）
返回值	无

函数原型	void gen(enum fct x, int y, int z)
参数描述	x 虚拟机指令,y 虚拟机指令的第二个参数，z 虚拟机指令第三个参数
函数描述	生成虚拟机代码
返回值	无

函数原型	void test(bool* s1, bool* s2, int n)
参数描述	s1 需要的单词集合，s2 出错恢复的补充单词集合
函数描述	检查当前单词进入该语法单位的合法性
返回值	无

函数原型	void block(int lev, int tx, bool* fsys)
参数描述	Lev 当前分程序所在层，tx 符号表当前尾指针，fsys 当前模块后继符号集合
函数描述	编译程序主体
返回值	无

函数原型	void enter(enum object k, int* ptx, int lev, int* pdx, int type)
参数描述	k 常量、变量或过程，ptx 符号表尾指针，lev 标识符所在层，pdx 当前应分配的变量的相对地址，type 标识符的数据类型
函数描述	在符号表中加入一项
返回值	无

函数原型	int position(char* id, int tx)
参数描述	id 要查找的名字，tx 当前符号表尾指针
函数描述	查找标识符在符号表中位置
返回值	0~tx

函数原型	void constdeclaration(int* ptx, int lev, int* pdx, int type) void vardeclaration(int* ptx, int lev, int* pdx, int type)
参数描述	ptx 当前符号表尾指针，lev 当前常量或变量所在层，pdx 当前应分配的相对地址，type 数据类型
函数描述	常量声明处理 变量声明处理
返回值	无

函数原型	void listcode(int cx0)
参数描述	cx0 当前分程序的目标代码起始位置
函数描述	输出目标代码
返回值	无

函数原型	void listall()
参数描述	无
函数描述	输出所有目标代码
返回值	无

函数原型	void statement(bool* fsys, int* ptx, int lev, int *fbreak, int *fcont)
参数描述	fsys 当前语句后继符号集合，ptx 当前符号表尾指针，lev 当前语句所在层，fbreak 当前句子的 break 语句对应跳转指令位置，fcont 当前语句 continue 语句对应跳转指令的位置
函数描述	语句处理
返回值	无

函数原型	void expression(bool* fsys, int* ptx, int lev) void termxor(bool* fsys, int* ptx, int lev) void termand(bool* fsys, int* ptx, int lev) void termadd(bool* fsys, int* ptx, int lev) void termmul(bool* fsys, int* ptx, int lev) void termnot(bool* fsys, int* ptx, int lev) void factor(bool* fsys, int* ptx, int lev) void condition(bool* fsys, int* ptx, int lev)
参数描述	fsys 当前后继符号集合，ptx 当前符号表尾指针，lev 当前层
函数描述	表达式处理，项处理，因子处理，条件处理
返回值	无

函数原型	void interpret()
参数描述	无
函数描述	解释程序
返回值	无

函数原型	int base(int l, int* s, int b)
参数描述	l 层数差，s 数据栈，b 基址

函数描述	通过过程基址求上 1层过程的基址
返回值	无

GUI 中所使用的函数略。