2020编译课程设计文档

18373187

程序环境

- 语言：C++
- IDE：VS2019
- 测试环境：VS自带编译器、C++ 14、Windows10
- 目标环境：Clang 8.0.1、C++ 11、Linux

词法分析阶段

输入输出要求

- 输入：`testfile.txt`
- 输出：`output.txt`

程序设计

编码前设计

设计单一工具类，由主函数读取文件后，传字符串给工具类并进行分析，分析得到单词及其类别码后直接输出到文件。

编码后设计

- `TypeEnum.h`
  枚举类，用于记录所有的单纯类别码；
- `WordInfo.h`；`WordInfo.cpp`
  单词信息封装类；内含：
  - 单词类别码
  - 单词对应的字符串
  - 单词所在行数
  - 单词位于所在行的位置
- `Lexer.h`；`Lexer.cpp`
  对词法分析的工具类，对读到的单词用`WordInfo`封装并存储在`vector`中；
  传入输入文件路径开始分析并存储，传入输出文件路径对读取到的单词信息进行输出。
  - `read_in(string file_name)`
    从`file_name`指向的文件读取要解析的程序
  - `print_to_file(string file_name)`
    把读取到的词法内容输出到`file_name`

语法分析阶段

输入输出要求摘录

- 输入：`testfile.txt`
- 输出：`output.txt`
- 把读入的词法内容，分析出程序中包含的语法部分（参照给定语法），并进行输出（不包括错误处理）

程序设计

编码前设计

1. 设计基类（`Base`），包含分析语法的方法(`read_in()`)、输出成字符串的方法(`to_string()`)以及存储该语法包含所有基类的属性（`vector<Base>`）。
2. 每个语法成分公有继承基类，重写上述两个方法，存储自己包含的内容到vector。

存在问题

我们并不需要输出所有语法成分，意味着部分语法成分不需要封装而可以沿用`WordInfo`的封装。因而统一设计基类会比较麻烦。

编码后设计

- 对原有`Lexer`进行增量改造，便于进行语法分析：
  - `int word_pos`

    初始化为0，用于充当读取被存储的`vector<WordInfo>`的指针，便于语法分析时依次取用读取到的词法成分。
  - `WordInfo get_next()`

    返回位于`word_pos`处的词法成分，并把`word_pos`更新为`word_pos+1`
  - `WordInfo peek_next()`

    返回位于`word_pos`处的词法成分
  - `void set_pos(int pos)`

    设置`word_pos`为传入值
  - `int get_pos()`

    获取当前`word_pos`值
- 设置用于记录程序中包含的函数的表格`FunctionTable`，暂时只记录函数名称对应的返回值类型
  - `map<string, int> func_to_type`

    记录函数名称-返回值类型的键值对。

    0：`void`

    1：`int`

    2：`char`
  - `void add(WordInfo w, int type)`

    功能：用于登记新读入到的函数

    输入：`WordInfo`，要求其类型为`IDENTF`，即函数名称；`type`，函数类型，如上。
  - `bool have_return(WordInfo w)`

    功能：返回指定函数是否有返回值

输入：`WordInfo`，要求其类型为`IDENTF`，即函数名称。

- 对于每个需要输出的语法成分，单独设计一个`class`来封装，其属性因其需要存储的词法成分而不同，但都有读入（分析）和输出成字符串两个方法。
  - `int read_in(Lexer& lexer)`

    功能：传入已读取到的词法成分开始分析

    输入：读取完成的词法成分

    返回：正确得到语法成分返回-1；错误时返回出现错误处的词法成分的位置
  - `string to_string()`

    功能、返回值：把语法成分变成字符串格式以供输出
  - `int word_pos`

    记录该成分保存的（或读取到的）第一个词法成分在整个读入的程序的词法程序中的位置（全局排序）。这个数值可以用于预读入之后的回溯，也可以用于之后的错误分析。
- 采用自顶向下的递归语法分析方法

错误处理阶段

输入输出要求

- 输入：`testfile.txt`
- 输出：`error.txt`
- 输出分析得到的错误行号以及类型

程序设计

符号表构造

- `IdentifyTable`类，

  设置为全局、静态类，整个程序通过直接调用其中设置为`static`的函数可以直接向符号表登记新的函数、常量、变量，或查询`IDENTIFY`标识符是否存在，或查询已登记的函数的参数表，常量、变量的类型。
  - `static void init()`

    初始化。
  - `static bool add_func(WordInfo* id, IdentifyType ret_type, ParameterTable* paras);`
    `static bool add_const(WordInfo* id, WordInfo* type);`
    `static bool add_var(WordInfo* id, WordInfo* type, int dimension);`

    向符号表添加新的符号信息。
  - `static bool have_var_const(WordInfo* id);`
    `static IdentifyType get_type_by_name(WordInfo* id);`
    `static IdentifyProperty get_property_by_name(WordInfo* id);`

    查询登记过的常量、变量信息。
  - `static bool have_func(WordInfo* func_id);`
    `static bool have_return(WordInfo* func_id);`
    `static bool check_func_para_num(WordInfo* func_id, ParameterValue* values);`
    `static bool check_func_para_type(WordInfo* func_id, ParameterValue*`

`values);`
`static IdentifyType get_return_type(WordInfo* func_id);`

查询登记过的函数信息。

- `IdentifyBlock`类

  全局变量以及函数存储在全局`block`中，每增加一个作用域（推理到本次设置是每增加一个函数），则增加一个`block`存放该作用域中的符号名称。
  详细内容略。

- `IdentifyInfo`类

  存放每个符号的`WordInfo`信息，小写名称（因要求不区分大小写），属性（常量、变量或函数），类型（常量、变量为`int`或`char`，函数存返回值类型），函数参数表或常量、变量的维数。
  详细内容略。

错误表构造

- `Error`类
  登记每个错误信息，记录错误所在行号和类型。
  详细内容略。

- `ErrorTable`类

  设置为全局、静态类，整个程序通过直接调用其中设置为`static`的函数可以向表中增加新的错误、查询是否有错误被登记以及输出错误到文件。

  - `static void log_error(int line, string type)`

    登记新的错误。

  - `static void print_to_file(string file_name)`

    输出错误到文件。

  - `static bool have_error()`

    检查是否有错误被登记。

整体流程

1. 程序进行词法分析，若出现错误直接登记到错误表。
2. 程序进行语法分析，若出现错误直接登记到错误表。
3. 程序检查错误表是否为空，是则输出语法分析结果，否则输出错误内容，输出时首先对错误进行排序。

值得一提的错误及处理

错误1

- 错误：存在非`void`函数不是所有路径都有返回值。
- 表现：本地测试与测评机测试结果不一致。
- 分析：不设置返回值则返回值不确定，是危险的。亦可能不同编译器、编译环境对默认返回值的处理不同。
- 处理：增设返回值。

错误2

- 错误：对出现的错误信息按行号进行排序时，使用C++的`sort()`对`vector`中内容进行排序后排序结果不稳定。
- `sort()`的使用方法：在需要排序的`Error`类中对操作符"<"、">"、"=="、"!="、进行了定义。
- 表现：运行后排序结果有时升序有时降序。
- 处理：放弃使用`sort()`函数，手写冒泡排序。

错误3

- 错误：在测评机环境编译`translate()`函数出现问题。
- 表现：仅在测评机环境编译错误。
- 分析：`translate()`函数传入的`tolower`函数在Linux环境下有宏定义与该函数重名，导致编译器分析失败。
- 处理：更改命名空间设置让编译器成功链接到该函数。

代码生成阶段

输入输出要求

- 输入：`testfile.txt`
- 输出：`mips.txt`
- 输出生成的mips代码。

程序设计

整体生成流程

1. 词法分析，并进行错误检查；
2. 对词法分析结果进行语法分析，并进行错误检查；
3. 若存在错误，输出错误信息并终止程序，否则继续执行4；
4. 对语法分析结果进行分析，并生成中间代码。
5. 按照中间代码生成mips汇编代码。

生成代码专用符号表

尽管在错误处理阶段已经设置了符号表对符号进行记录，但错误处理用符号表设置时并未考虑中间代码生成的需求，故在生成中间代码时再次单独设置新的符号表。

常量表

- 常量信息

```
class ConstInfo
{
public:
    int type = 0;                       //0:int, 1:char
    string name_in_low = "\0";
    int value_int = 0;                  //if type is int
    char value_char = '\0';             //if type is char
    int get_value();                    //return both int and char value in
int type
};
```

由于最后在mips中，所有字符型最终会作为整数处理，故这里之间返回字符对应的ASCII值；

由于本程序整体不区分字母大小写，这里只记录小写状态下名称。

- 常量表

```
class ConstTable
{
private:
    vector<ConstInfo*> consts;
public:
    ConstTable();
    void add_in(ConstInfo* info);
    bool have_name(string name);
    ConstInfo* get_info_by_name(string name);
};
```

在整体框架下，一个常量表存储全局或一个函数内部的所有常量，因此不存在重名问题。

变量表

- 变量信息

```
class VarInfo
{
public:
    int type = 0;                       //0:int, 1:char
    string name_in_low = "\0";
    int dimenation = 0;                 //变量维数，可能为0，1，2
    int d1 = 0;                         //变量第一维大小，在维数为1，2时有意义
    int d2 = 0;                         //变量第二维大小，在维数为2时有意义
    bool have_initial = false;
    int size_in_byte = 0;               //变量在内存中占用的空间
    int real_offset = 0;
    bool is_func_arg = false;           //变量是不是函数调用时传递的参数
};
```

`real_offset`用来记录变量在内存中的位置，有三种情况：

  - 全局变量：相对data段顶的位置。
  - 函数参数：相对当前`$fp`的偏移量。
  - 局部变量：相对当前`$sp`的偏移量。

- 变量表

```
class VarTable
{
private:
    vector<VarInfo*> vars;
    int tmp_count = 0;
    int total_stack_size = 0;
    int para_stack_size = 0;
public:
    VarTable();
    string get_new_tmp();
    bool have_name(string name);
    void add_in(VarInfo* info);
    VarInfo* get_info_by_name(string name);
```

```
    string to_mips();
    void count_arg_stack();
    int get_stack_size();
    int get_para_size();
};
```

在整体框架下，一个变量表存储全局或一个函数内部的所有变量，因此不存在重名问题；

变量表同时还可以计算整个表中变量在内存中所需空间，包括两部分——函数参数占用空间与局部变量占用的空间。

字符串表

字符串表用于记录整个程序中所有**输出语句**中出现的字符串。

- 字符串信息

```
class StringInfo
{
public:
    string name_in_low;
    string value;
};
```

在存储字符串时需要考虑“`\`”的转义问题。

- 字符串表

```
class StringTable
{
private:
    vector<StringInfo* > strings;
    int count = 0;
public:
    string get_label();
    void add_in(StringInfo* info);
    string to_mips();
};
```

所有字符串在mips中会被存储在`.data`段，调用时通过其标签调用；标签编译过程中自动生成。

中间代码·四元式设计

四元式设计

四元式标志	说明	ARG1	ARG2	ARG3
ADD	ARG3 <= ARG1 + ARG2	变量	变量	变量，输出
ADDI	ARG3 <= ARG1 + INT	变量	整数	变量，输出
SUB	ARG3<= ARG1 - ARG2	变量	变量	变量，输出
SUBI	ARG3 <= ARG1 - INT	变量	整数	变量，输出
MULT	ARG3<= ARG1 * ARG2	变量	变量	变量，输出
MULI	ARG3 <= ARG1 * INT	变量	整数	变量，输出
DIV	ARG3<= ARG1 / ARG2	变量	变量	变量，输出
DIVI	ARG3 <= ARG1 / INT	变量	整数	变量，输出
NEG	ARG3 <= -ARG1	变量	空	变量
ASSIGN	ARG3 <= ARG1	变量	空	变量
INIT	变量初始化	变量	空	常量
P_STR	输出字符串	字符串标识	空	空
P_CHAR	输出字符	输出值	空	空
P_INT	输出整数	输出值	空	空
S_INT	输入整数	空	空	目标变量
S_CHAR	输入字符	空	空	目标变量
START_FUNC	标记函数调用开始（注释，无实意）	函数名	空	空
SAVE_PARA	函数调用时压入参数	被存储值	空	空
JAL	开始函数调用	函数名	空	空
JUMP	跳转到标志	标志	空	空
SAVE_RET	存储返回值到特定寄存器	待存储值	空	空
LOAD_RET	得到函数返回值	待赋值变量	空	空
BEQI	当ARG1与INT相等时跳转到标志	变量	整数	标志
BEQ	当ARG1与ARG2相等时跳转到标志	变量	变量	标志
BNE	当ARG1与ARG2不相等时跳转到标志	变量	变量	标志
BGE	当 `ARG1 => ARG2` 时跳转到标志	变量	变量	标志
BGT	当 `ARG1 > ARG2` 时跳转到标志	变量	变量	标志
BLE	当 `ARG1 <= ARG2` 时跳转到标志	变量	变量	标志
BLT	当 `ARG1 < ARG2` 时跳转到标志	变量	变量	标志
LABEL	记录标志位置，生成标志	标志	空	空

存储四元式

- `MiddleSentence`

```
class MiddleSentence
{
private:
    Operation op;
    Arg* arg1;
    Arg* arg2;
    Arg* out;
    string load_arg_to_reg
        (Arg* arg, ConstTable* l_const, ConstTable* g_const,
        VarTable* l_var, VarTable* g_var);
    string save_to_stack
        (Arg* arg, ConstTable* l_const, ConstTable* g_const,
        VarTable* l_var, VarTable* g_var);
public:
    MiddleSentence(Operation o, Arg* a1, Arg* a2, Arg* a3);
    string to_string();
    string to_mips(ConstTable* l_const, ConstTable* g_const,
        VarTable* l_var, VarTable* g_var, StringTable* strings,
        MiddleCode* all_code, int& func_para_size);
    Arg* get_arg1();
    Arg* get_arg2();
    Arg* get_arg_out();
    Operation get_operation();
};
```

每个此类实体对应一个四元式。

- `Arg`

```
class Arg
{
private :
    ArgType type;                //分为整数、字符、标识符、数组标识符、空5种
    int value_int;               //类别为整数时有效
    char value_char;             //类别为字符时有效
    string identify;             //类别为数组标识符或标识符时有效
    Arg* offset;                 //类别为数组时有效，表示本参数相对数组头部的偏移量
    int target_reg = 0;
    bool need_stack = true;
public :
    ArgType get_type();
    int get_value();
    string get_id();
    Arg* get_offset();
    string to_string();
    void set_target_reg(int reg, bool n_s);
    void set_need_stack(bool b);
    bool operator==(const Arg& a);
    bool check_need_to_stack();
};
```

每个此类实体对应一个参数。

四元式分区

四元式按照函数进行划分，每个函数的四元式存储在一起，与该函数的变量表、常量表组合在一起存储在一个类（`MiddleFunction`）种。

部分重要逻辑

取数组值

1. 在中间代码生成过程中计算要用到的变量相对数组头部的偏移量，作为临时变量记录在变量表中，对应参数记录在`Arg`类中；计算过程直接作为中间代码存储；
2. 生成mips时直接通过偏移量找到对应的变量位置。

函数调用

1. 当前函数把参数压入内存中；所有参数均存入内存，不通过寄存器传值；过程中不改变`$sp`的值
2. 更改`$sp`到真实栈底，保存`$fp`到栈中，并设置`$fp = $sp`；
3. 进入被调用函数；
4. 维护寄存器；
5. 为局部变量开设栈中空间；
6. 运行被调用函数；
7. 处理返回值；
8. 从栈中推出局部变量；
9. 恢复寄存器；
10. 返回原函数；
11. 恢复`$fp`寄存器值；
12. 从栈中推出函数调用用的参数；
13. 继续原函数内容。

优化阶段

新增四元式

四元式标志	说明	ARG1	ARG2	ARG3
SLL	ARG3 <= ARG1 << ARG2(向左逻辑位移)	变量	整数	变量，输出

优化操作

对于已经生成的中间代码

- 优化`ASSIGN`：如果赋值之前为加减乘除运算等，运算结果直接改为下一条待赋值变量。
- 窥孔优化：检查可以直接计算得出的固定值，对中间代码中对应变量直接赋值。
- 优化乘法：对于可以改为逻辑左移的乘法操作，改为逻辑左移（对于负数为逻辑左移加取反操作）。

附录

词法定义

单词名称	类别码	单词名称	类别码	单词名称	类别码	单词名称	类别码
标识符	IDENFR	else	ELSETK	-	MINU	=	ASSIGN
整形常量	INTCON	switch	SWITCHTK	*	MULT	;	SEMICN
字符常量	CHARCON	case	CASETK	/	DIV	,	COMMA
字符串	STRCON	default	DEFAULTTK	<	LSS	(	LPARENT
const	CONSTTK	while	WHILETK	<=	LEQ	)	RPARENT
int	INTTK	for	FORTK	>	GRE	[	LBRACK
char	CHARTK	scanf	SCANFTK	>=	GEQ	]	RBRACK
void	VOIDTK	printf	PRINTFTK	==	EQL	{	LBRACE
main	MAINTK	return	RETURNTK	!=	NEQ	}	RBRACE
if	IFTK	+	PLUS	:	COLON		

需要分析的错误类型

错误类型	错误类别码	解释及举例
非法符号或不符合词法	a	例如字符与字符串中出现非法的符号，符号串中无任何符号
名字重定义	b	同一个作用域内出现相同的名字（不区分大小写）
未定义的名字	c	引用未定义的名字
函数参数个数不匹配	d	函数调用时实参个数大于或小于形参个数
函数参数类型不匹配	e	函数调用时形参为整型，实参为字符型；或形参为字符型，实参为整型
条件判断中出现不合法的类型	f	条件判断的左右表达式只能为整型，其中任一表达式为字符型即报错，例如 `'a'==1`
无返回值的函数存在不匹配的return语句	g	无返回值的函数中可以没有 `return` 语句，也可以有形如 `return;` 的语句，若出现了形如 `return(表达式);` 或 `return();` 的语句均报此错误
有返回值的函数缺少return语句或存在不匹配的return语句	h	例如有返回值的函数无任何返回语句；或有形如 `return;` 的语句；或有形如 `return();` 的语句；或 `return` 语句中表达式类型与返回值类型不一致
数组元素的下标只能是整型表达式	i	数组元素的下标不能是字符型
不能改变常量的值	j	这里的常量指的是声明为 `const` 的标识符。例如 `const int a=1;` 在后续代码中如果出现了修改 `a` 值的代码，如给 `a` 赋值或用 `scanf` 获取 `a` 的值，则报错。
应为分号	k	应该出现分号的地方没有分号，例如 `int x=1` 缺少分号 （7种语句末尾， `for` 语句中，常量定义末尾，变量定义末尾）
应为右小括号')'	l	应该出现右小括号的地方没有右小括号，例如 `fun(a,b;` ，缺少右小括号（有/无参数函数定义，主函数，带括号的表达式， `if` ， `while` ， `for` ， `switch` ，有/无参数函数调用，读、写、 `return` ）
应为右中括号']'	m	应该出现右中括号的地方没有右中括号，例如 `int arr[2;` 缺少右中括号（一维/二维数组变量定义有/无初始化，因子中的一维/二维数组元素，赋值语句中的数组元素）
数组初始化个数不匹配	n	任一维度的元素个数不匹配，或缺少某一维的元素即报错。例如 `int a[2][2]={{1,2,3},{1,2}}`
<常量>类型不一致	o	变量定义及初始化和switch语句中的<常量>必须与声明的类型一致。 `int x='c';int y;switch(y){case('1') ….}`
缺少缺省语句	p	`switch` 语句中，缺少<缺省>语句。

文法

< 加法运算符 > ::= + | -

< 乘法运算符 > ::= * | /

< 关系运算符 > ::= < | <= | > | >= | != | ==

< 字母 > ::= _ | a | . . . | z | A | . . . | Z

< 数字 > ::= 0 | 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} "

```
        //字符串中要求至少有一个字符
```

< 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 > ::= < 数字 > { < 数字 > }

< 整数 > ::= [+ | -] < 无符号整数 >

< 标识符 > ::= < 字母 > { < 字母 > | < 数字 > }

```
        //标识符和保留字都不区分大小写，比如if和IF均为保留字，不允许出现与保留字相同的标识符
```

< 声明头部 > ::= int < 标识符 > |char < 标识符 >

< 常量 > ::= < 整数 > | < 字符 >

< 变量说明 > ::= < 变量定义 > ;{ < 变量定义 > ;}

< 变量定义 > ::= < 变量定义无初始化 > | < 变量定义及初始化 >

< 变量定义无初始化 > ::= < 类型标识符 > (< 标识符 > | < 标识符 > '[' < 无符号整数 > ']' | < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']'){,(< 标识符 > | < 标识符 > '[' < 无符号整数 > ']' | < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']')}

```
        //变量包括简单变量、一维、二维数组，＜无符号整数＞表示数组各维元素的个数，其值需大于0，数组元素按行优先存储
        //变量没有初始化的情况下没有初值
```

< 变量定义及初始化 > ::= < 类型标识符 > < 标识符 > = < 常量 > | < 类型标识符 > < 标识符 > '[' < 无符号整数 > ']'='{' < 常量 > {, < 常量 > }'}' | < 类型标识符 > < 标识符 > '[' < 无符号整数 > ']''[' < 无符号整数 > ']'='{''{' < 常量 > {, < 常量 > }'}'{, '{' < 常量 > {, < 常量 > }'}'}'

```
        //简单变量、一维、二维数组均可在声明的时候赋初值，＜无符号整数＞表示数组各维元素的个数，其值需大于0，数组元素按行优先存储，＜常量＞的类型应与＜类型标识符＞完全一致，否则报错；每维初值的个数与该维元素个数一致，否则报错，无缺省值；
```

< 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 > } //[+|-]只作用于第一个<项>

< 项 > ::= < 因子 > { < 乘法运算符 > < 因子 > }

< 因子 > ::= < 标识符 > | < 标识符 > '[' < 表达式 > ']'| < 标识符 > '[' < 表达式 > ']''[' < 表达式 > ']'|'(' < 表达式 > ')' | < 整数 > | < 字符 > | < 有返回值函数调用语句 >

```
        //char 类型的变量或常量，用字符的ASCII 码对应的整数参加运算
        //<标识符>'['<表达式>']'和<标识符>'['<表达式>']''['<表达式>']'中的<表达式>只能是整型，下标从0开始
        //单个<标识符>不包括数组名，即数组不能整体参加运算，数组元素可以参加运算
```

< 语句 > ::= < 循环语句 > | < 条件语句 > | < 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 > ; | < 情况语句 > | < 空 > ; | < 返回语句 > ; | '{' < 语句列 > '}'

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

```
        //<标识符>=<表达式>中的<标识符>不能为常量名和数组名
```

< 条件语句 > ::= if '(' < 条件 > ')' < 语句 > [else < 语句 >]

< 条件 > ::= < 表达式 > < 关系运算符 > < 表达式 >

```
        //表达式需均为整数类型才能进行比较
```

< 循环语句 > ::= while '(' < 条件 > ')' < 语句 > | for'(' < 标识符 > = < 表达式 > ; < 条件 > ; < 标识符 > = < 标识符 > (+|-) < 步长 > ')' < 语句 >

```
        //for语句先进行条件判断，符合条件再进入循环体
```

< 步长 > ::= < 无符号整数 >

< 情况语句 > ::= switch ‘(’ < 表达式 > ‘)’ ‘{’ < 情况表 > < 缺省 > ‘}’ */测试程序需出现情况语句/*

< 情况表 > ::= < 情况子语句 > { < 情况子语句 > }

< 情况子语句 > ::= case < 常量 > : < 语句 >

< 缺省 > ::= default : < 语句 >

```
        //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
```

< 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 > ')'

```
        //函数调用时，只能调用在之前已经定义过的函数，对是否有返回值的函数都是如此
```

< 值参数表 > ::= < 表达式 > {, < 表达式 > } | < 空 >

```
        //实参的表达式不能是数组名，可以是数组元素
        //实参的计算顺序,要求生成的目标码运行结果与Clang8.0.0 编译器运行的结果一致
```

< 语句列 > ::= { < 语句 > }

< 读语句 > ::= scanf '(' < 标识符 > ')'

```
        //从标准输入获取<标识符>的值，该标识符不能是常量名和数组名
        //生成的PCODE或MIPS汇编在运行时，对每一个scanf语句，无论标识符的类型是char还是int，末尾均需回车；在testin.txt文件中的输入数据也是每项在一行
        //生成MIPS汇编时按照syscall指令的用法使用
```

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

```
        //printf '(' ＜字符串＞,＜表达式＞ ')'输出时，先输出字符串的内容，再输出表达式的值，两者之间无空格
        //表达式为字符型时，输出字符；为整型时输出整数
        //＜字符串＞原样输出（不存在转义）
        //每个printf语句的内容输出到一行，按结尾有换行符\n处理
```

< 返回语句 > ::= return['(' < 表达式 > ')']

```
        //无返回值的函数中可以没有return语句，也可以有形如return;的语句
        //有返回值的函数只要出现一条带返回值的return语句（表达式带小括号）即可，不用检查每个分支是否有带返回值的return语句
```

另外

关于类型和类型转换的约定：

1. 表达式类型为char型有以下三种情况：
   1）表达式由<标识符>、< 标识符 > '[' < 表达式 > ']和 < 标识符 > '[' < 表达式 > ']''[' < 表达式 > ']'构成，且<标识符>的类型为char，即char类型的常量和变量、char类型的一维、二维数组元素。
   2）表达式仅由一个<字符>构成，即字符字面量。
   3）表达式仅由一个有返回值的函数调用构成，且该被调用的函数返回值为char型
   除此之外的所有情况，<表达式>的类型都是int
2. 只在表达式计算中有类型转换，字符型一旦参与运算则转换成整型，包括小括号括起来的字符型，也算参与了运算，例如('c')的结果是整型。
3. 其他情况，例如赋值、函数传参、if/while条件语句中关系比较要求类型完全匹配，并且 < 条件 > 中的关系比较只能是整型之间比，不能是字符型。