代码生成阶段设计文档

输入输出要求

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

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

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。
