SeedCup2016-糖分

实现功能

1. 能够正确输出顺序结构的执行顺序
2. 能够正确输出分支结构的执行顺序
3. 能够正确输出循环结构的执行顺序

优点

1. 逻辑清晰，架构良好
   - 程序主体逻辑清晰，目录结构一目了然
2. 可扩展性高
   - 模块之间相对独立，耦合性较低
3. 可处理无限次循环和分支嵌套（在资源允许的情况下）
   - 因为处理过程采用递归调用实现，变量存取采用栈实现
4. 野路子，想象力丰富
   - 没有按照编译原理的套路来
5. 每条顺序语句都有返回值，可处理 while (printf("..")) {...} 的情况

编译运行

编译环境：

操作系统	编译器	编译参数
macOS Sierra 10.12.1	Apple LLVM version 8.0.0 (clang-800.0.42.1)	-I include/ -Wno-c++11-extensions
Arch Linux	gcc	
Windows10		

使用GNU Make工具编译整个工程可执行下列命令：

```
make realclean
make
```

若有环境问题，请编辑 `Makefile` 文件，把第5行 `CPP = g++` 改为测试机上的编译器。

若没有问题可以通过如下命令运行：

```
make run
```

或

```
../Bin/SeedCup2016.exe
```

我们也准备了三个版本的可执行文件，分别为：`SeedCup2016(macOS).exe`，`SeedCup2016(Linux).exe` 和 `SeedCup2016(Windows).exe`，分别对应 macOS，Linux 和 Windows 操作系统。

程序设计

目录结构

- Src
    - include - 存放所有头文件
        - code.h - 定义了存储代码的结构体
        - const.h - 常量定义及声明
        - functions.h - 所有全局函数的声明
        - global.h - 全局变量的声明
        - headers.h - 包含了其他头文件，方便cpp引用
        - variable.h - 变量类的声明

        - branch.cpp - 处理分支结构，函数 `do_if` 所在位置
        - global.cpp - 全局变量定义，通过控制宏定义实现自动定义
        - KMP.cpp - KMP串匹配算法
        - loop.cpp - 处理循环结构，函数 `do_loop` 所在位置
        - main.cpp - 程序入口，主循环 `do_block` 所在位置
        - Makefile - 编译运行工程
        - preprocess.cpp - 预处理文件
        - sequence.cpp - 处理顺序结构，函数 `do_sequence` 所在位置
        - util.cpp - 通用工具类函数定义，如读写文件等
        - variable.cpp - 变量类的实现

- Bin
    - SeedCup2016.exe
    - SeedCup2016(macOS).exe
    - SeedCup2016(Linux).exe
    - SeedCup2016(Windows).exe

- Doc
    - README.pdf

模块说明

整体逻辑

- 首先对输入文件进行预处理
- 定义一个全局指针指向待执行代码的语句，就像IP寄存器一样，初始为第一行第一句
- 建立一个全局栈(GVS, Global Variable Stack)来保存全局变量表，初始创建一个空表入栈，这个栈被封装成一个变量类(Variable)
- 建立一个全局列表保存要输出的行号，初始为空，取名为OUTPUT
    - 对OUTPUT操作的函数是 `do_output`，里面包含判断的逻辑

- 函数 `do_block()` - 处理一个代码段
    - while (没有到BLOCK_END)
        - `do_output()`
        - 匹配IP指向的语句
            - 如果是 BLOCK_END

- `do_exit()` - 释放资源
- `return` - 退出

- 如果匹配到顺序结构，调用 `do_sequence()`
- 如果匹配到分支结构，调用 `do_if()`
- 如果匹配到循环结构，调用 `do_loop()`

预处理

1. 获取正确的行数
2. 去掉空行和注释
3. 一行一句排列
4. 若if或loop只有一条语句没写 `{}`，再加上
5. 设置 `break;` 和 `}` 的属性为 BLOCK_END
6. 在输入程序最后加上BLOCK_END

   预处理结果示意：

```
行号 属性 代码
1    0    int apple;
1    0    int orange;
4    0    apple = 1;
5    0    int done;
6    0    int asdfor;
7    0    apple = apple * 1;
7    0    orange = apple / orange;
8    5    if (orange!= 2){
9    0    printf("%d", apple);
9    3    }
11   5    else if(orange <= 3){
11   0    printf("%d", apple);
12   3    }
12   5    else{
13   0    printf("%d", orange);
13   3    }
...
```

---

do_sequence()

- 声明语句，设置变量
- 初始化语句
- 赋值语句，更新变量
    - 自增自减
    - 表达式计算
    - 处理时可以把等号左右分开
- 输出语句
    - 要考虑一下printf里面表达式对变量值的影响，如a++
- IP++

do_if()

- 判断条件，匹配括号
- 条件匹配成功，找到要执行的代码段：
    - 设置IP = 该段段首
    - push一张新的变量表
    - 调用do_block
- 继续匹配括号，直到该分支彻底结束，设置IP为段尾后一句

do_loop()

- for / while
    - while(判断循环条件)
        - 注：for 的循环条件在for的下一行
        - 设置IP = 该段段首
        - push一张新的变量表
        - 调用do_block
    - 设置IP为循环结束后一句
- do...while
    - 记录do的位置
    - 设置IP = 该段段首(do + 1)
    - push一张新的变量表
    - 调用do_block
    - while(判断循环条件)
        - 设置IP = 该段段首
        - push一张新的变量表
        - 调用do_block

- 设置IP为循环结束后一句

Variable 类

类的声明

```
class Variable
{
    // "global" variable stack
    vector<map<string, int>*> variable_stack;
public:
    Variable();
    ~Variable();
    // 取得某个变量的值，变量值放在value中，返回成功失败
    bool get(const string& name, int& value);
    // 设置(初始化)某个变量的值
    void set(const string& name, int value);
    // 更新某个变量的值
    void update(const string& name, int value);
    // push一张变量表
    void push_table();
    // pop一张变量表
    void pop_table();
    // 打印变量表
    void print_table();
};
```

存储变量策略

- 每进入一层 `{}` 就 push 一张空表入栈，每出一层 `{}` 就 pop 一张表并释放资源；
- 设置（初始化）变量：在栈顶的表中记录变量的值，一张表就是一个字典；
- 查询变量：从栈顶到栈底的表中依次查询该变量，直到找到为止；
- 更新变量：从栈顶到栈底的表中依次查询该变量，直到找到为止，然后更新该变量的值。

代码的数据结构

存储代码结构体

```
typedef struct code_struct {
    string code;        // 代码字符串
    int line_number;    // 行数
    char attribute;     // 代码属性，详见const.h
} Code;
```

属性的定义

```
/* code attribute */
#define     EXEC_CODE           0   // 可执行语句，可能需输出行号
#define     DECLAIR_CODE        1   // 声明语句，不需声明行号
#define     INIT_CODE           2   // 初始化语句，可能需输出行号
#define     BLOCK_END           3   // 块结束，如果是break; 则可能需输出行号
#define     OTHER_CODE          4   // 其他不输出行号语句
#define     IF_ELSE_CODE        5   // 分支语句
#define     LOOP_CODE           6   // 循环语句，for, while..
```
