

Mao 语言运行器

C 语言课程项目

李晗

1552771

目录

一．项目说明

1. 项目介绍

(1) 项目构成

(2) 功能构成

2. 健壮性考虑

二．开发历程







三．收获及感想

项目说明

项目介绍

整个项目作为 Mao 语言的解释器，进行了变量的定义，运算，与最后的输出，定义部分采用字典树存储的方式存储变量，运算运用双栈进行运算，输出时正常输出变量或者常数。

项目构成

📁 头文件	
▷  expression.h	: 表达式相关函数。
▷  stack.h	: 栈的定义与函数。
▷  string_division.h	: 字符串处理的函数。
▷  tools.h	: 一些工具函数。
▷  trie_tree.h	: Trie 树的定义域函数。
▷  variable_store.h	: 变量存储的函数。
📁 源文件	
▷ ++ expression.c	
▷ ++ main.c	
▷ ++ stack.c	
▷ ++ string_division.c	
▷ ++ tools.c	
▷ ++ trie_tree.c	
▷ ++ variable_store.c	

功能模块

整个项目功能上分为以下几个部分：

1. 输入字符串的处理。
2. 声明语句时的变量存储。
3. 运算语句的计算。
4. 输出语句的输出。

输入字符串处理：

判断语句的类型，为之定义了如下枚举变量：

```
enum sentence_type  
{INT_DECLARATION_SENTENCE,  
DOUBLE_DECLARATION_SENTENCE,  
OUTPUT_SENTENCE,  
CALCULATE_SENTENCE,  
BLANK_SENTENCE};
```

分别表示了：声明int型变量语句、声明double型变量语句、输出语句、运算语句、空行。

由这个一个函数进行判断，这个函数设计考虑到了语句前可能出现不影响语义的空格。

```
enum sentence_type declaration_variable_type  
(char *_declaration_sentence_sentence);
```

如果一个语句是一个声明语句，我们将它的前几位去掉（返回变量名部分的子串）并去掉最后的分号，去掉所有不影响语义的空格：

```
char *declaration_variable_start(char *_sentence);
```

```
char *delete_space(char *_input_string);
```

```
char *delete_se(char *_string);
```

（该函数同时进行了判断是否

缺少分号的情况。）

下面得到的是一个由变量名与分号组成的字符串，且变量名由分号隔开。下面将这些变量名进一步拆成子串，并存在一个字符串数组里。

```
char **string_division(char *input_string);
```

这其中调用了于string.h中的库函数strtok。

变量存储

现在拿到了一个存着这句声明语句中出现的变量名的字符串数组，把他存入一个存储结构中。

最开始选择使用链表，应用其可以动态增添的特性。

首先规定了存储变量的类型

```
enum Type { INT, DOUBLE };
```

然后这样建立的链节：

```
typedef struct List
{
    char *variable_name;
    enum Type type;
    int int_value;
    double double_value;
    struct List *p_next;
}List;
```

记录了该链节存储的变量的名称，数据类型，分别用一个int_value和一个double_value存储可能的int或者double类型的变量。

分别设计了

链表初始化

```
List *initialize_list(void);
```

增添链节

```
void add_item(enum Type _type, char *_name, List *_head);
```

查找变量

```
List *find_variable(char *_name, List *_head);
```

删除整个链表

```
void clear_list(List *_head);
```

后来考虑到数据量会比较大的问题，放弃了这种做法，将链表存储改为使用字典树（Trie tree）

设计树节点如下,节点内存储的项与链表相似：

```
typedef struct Node {  
    int int_value;  
    double double_value;  
    enum Type type;  
    struct Node *p_next[256];  
    bool declared;  
}Trie;
```

为之设计了如下函数：

初始化一棵树：

```
Trie *initialize_tree(void);
```

增加一个变量：

```
void add_variable(enum Type _variable_type, char *_variable_name, Trie *_trie_tree);
```

查找一个变量：这里的第三个参数为了处理多重定义的情况。

```
Trie *find_variable(char *_variable_name, Trie *_trie_tree, int _reference_tiem);
```

销毁整棵树：

```
void destroy_TrieTree(Trie *_trie_tree);
```

在链表到 Trie 树的替换过程中，由于函数设计特点，替换相当迅速。

至此，变量的存储需要的功能基本完成，将其封装与一个函数中：

```
void store_variable_in_tree(Trie *_variable, char *_sentence);
```

可以看到，这个函数只有读入的字符串与所需的字典树根节点。

在这个函数中判断了句子中出现连续逗号的情况与不合法的变量命名。

运算语句

首先进行运算语句的读入，遇到的第一个问题还是不影响语义的空格与末尾的分号，仍然用 `delete_space` 于 `delete_se` 进行删除。

为了更好的拆分与计算，设计了这样一个预处理函数处理运算语句

```
Char *pretreatment_expression(char *_expression);
```

它进行了这样一些操作，首先，在式子中每一项（包括变量名，操作符，数字）后边插入一个空格，再在式子开头加入一个@字符，结尾加一个#字符。

预处理完成后，设计这一个函数，将式子拆成子串并存在一个字符串数组中，由于空格的存在让这样的拆分很方便。

```
char **expression_division(char *_expression);
```

现在得到了一个运算语句的各个项，下面进行运算。运算处理分别建立了操作数栈 `operated_number_stack` 与操作符栈 `operator_stack`。

栈的建立时，由于当时刚了解栈这种数据结构，模仿网上的链栈代码建立。

开始时的想法是栈中内容只存指针，如下

```
typedef struct stack
{
    void *item;
    enum item_type item_type;
    struct stack *p_next;
}stack;
```

有一个枚举变量表示栈中存的元素类型。

```
enum item_type
{INT_CONST,DOUBLE_CONST,VARIABLE,OPERATOR};
```

可能是 `int` 型常数，`double` 型常数，变量，或者操作符。

后来发现计算过程中的问题，改成了下面在当时看来比较麻烦的做法。但是对于变量做左值得情况有很好的效果。

```
typedef struct stack
{
    Trie *variable;
    int int_value;
    double double_value;
    char operator;
    enum item_type item_type;
    struct stack *p_next;
}stack;
```

为它写了如下函数

新建一个栈

```
stack *initialize_stack(void);
```


压栈

```
void push(Stack *_head, Trie *_item,  
int_int_value, double _double_value, char  
_operator, enum item_type _item_type);
```

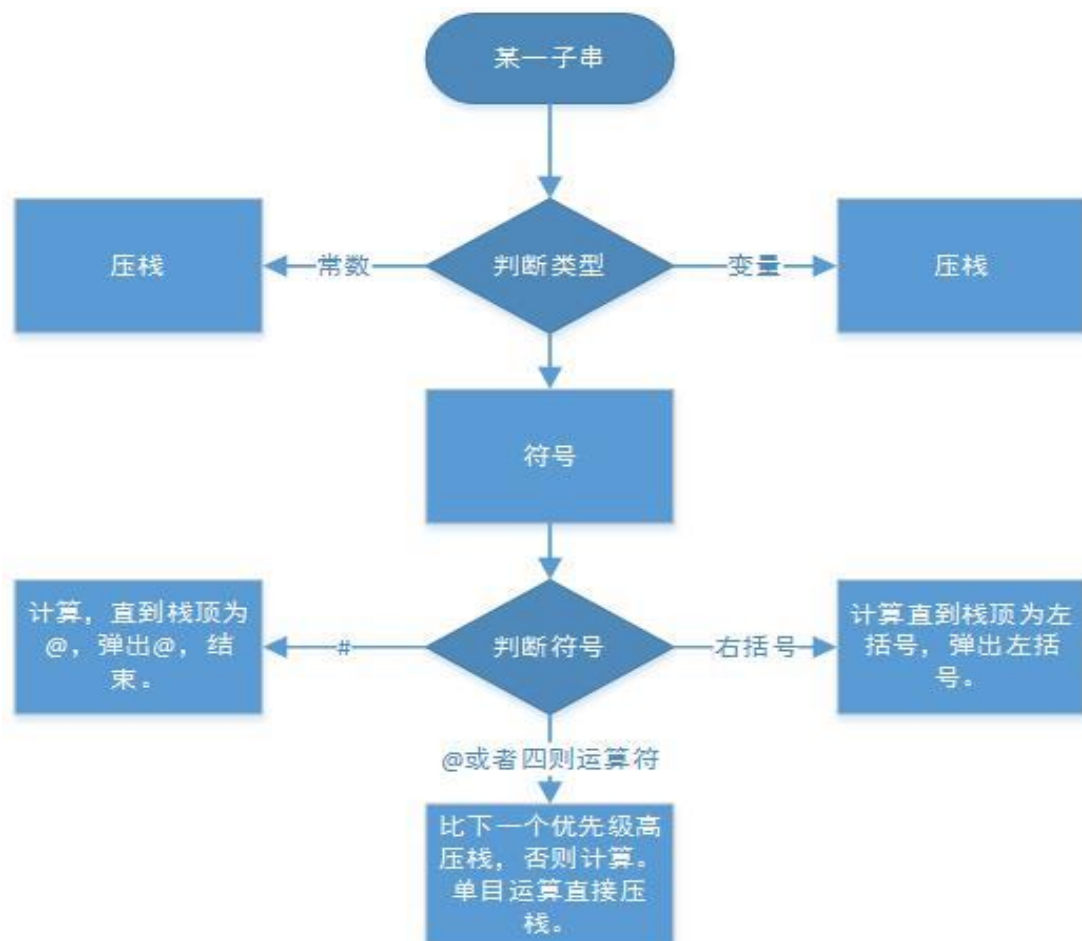
弹栈并返回指针

```
Stack *pop(Stack *head);
```

找到栈顶并返回栈顶指针

```
Stack *find_stack_top(Stack *_head);
```

下面是计算的过程



这里设计了一个计算的函数，计算的是已弹栈顶和现在栈顶，其中分为如下几种情况，并作为处理了单目运算符的+-号，完成了隐式

转换。

```
void calculate_stack_top  
(Stack*_operated_number_top,  
    stack**_operated_number, char _operator);
```

输出部分

这一部分比较简单，设计了一个输出的函数

```
void print_value(char *_variable_name, Trie  
*_variable_tree);
```

传入了要输出的变量名，与存储变量的字典树，

它的参数由一个解析输出语句中括号内部分的函数传入

```
Char *find_name_in_print_sentence(char  
*_print_sentence);
```

查找变量名，即可输出。其中注意了无法搜索到的变量的情况，
以及输出一个数字的情况。

健壮性考虑

首先，为可能的出错数据进行了报错，并定义了一个全局变量来记录行数。其次，为一些极端的正确数据进行了处理，保证代码在可预知范围内正确运行。

有如下可能出现的错误数据：

1. 一行的结尾缺少分号。
2. 一行只有分号。
3. 变量名未被定义。
4. 变量名重复定义。
5. 变量名不合法（使用了 `python` 语言保留字）。
6. `print` 语句中缺少右括号。
7. `print` 语句中无变量名，（如：`print();`）

有如下极端数据：

1. 多组括号出现的。
 2. 多个负号/正号连续出现的。
 3. `print` 一个常数的。
- 等。

开发历程

开发进度

2015.11.24 尝试用栈进行计算

2015.12.02 写出第一版计算操作

2015.12.07 正式开始项目，写下第一个字符串解析函数

2015.12.11 完成链表式的变量存储

2015.12.15 建好栈完成栈的有关函数

2015.12.21 完成第一版运算表达式的解析

2015.12.25 推倒表达式解析重构

2015.12.27 开始写利用栈的计算部分

2015.12.29 完成第一版计算

2015.12.31 完成第一版项目

2015.01.01 00 : 00 : 00 实验成功字典树

2015.01.01 15 : 00 重构变量存储部分，改链表为字典树

2015.01.01 19 : 30 添加健壮性考虑，重新优化计算部分。

2015.01.02 02:15 完成

收获及感想

Coding skill

1. heap 上的内存若越界，free 时会出事情。
2. 不在 heap 上开的内存，不要乱 free。
3. 作为承接有返回值的函数的指针就不用分配内存了。
4. 避免使用局部的指针变量，防止函数栈释放后被覆盖。
5. 先在逻辑上解决问题再去实现。
6. 尽量让代码易于修改。
7. 体会:高内聚，低耦合。

Time management

1. 先动手去做，不要纠结好久去学一些针对性不强的东西。
2. 不要往后拖。
3. 尽量利用不写代码的时间思考。
4. 每次完成一个功能块要及时测试。