

“程序设计实践”上机作业文档

2019211304 班 2019211285 窦天杰

目录

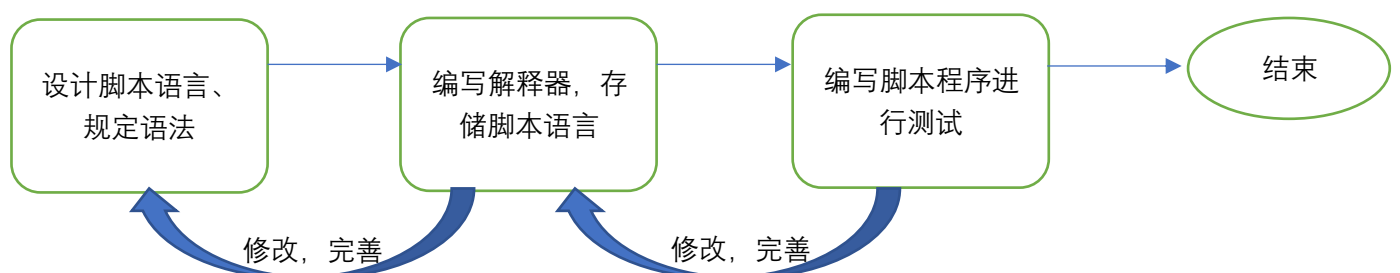
“程序设计实践”上机作业文档	1
一、 题目及要求	2
二、 设计思路	2
三、 语法设计（脚本语言部分）	3
i. 关键字（保留字）	3
a) robot_name	3
b) window_size	4
c) begin	4
d) jump	5
e) else	5
f) default	6
ii. 运算符（操作符）	7
a) 加号（+）	7
b) 减号（-），大于号（>）	8
c) 引号（“”）	8
d) 等号（=）、括号、冒号（:）	8
e) ※逗号（,）、分号（;）	8
iii. 变量	12
iv. 注释	14
四、 解释器设计	14
i. 整体框架	14
ii. 关键实现	14
a) Request 类	14
b) DSL 类	16
c) ※interpreter 类	18
d) 自动化测试程序的添加	22
五、 样例测试	24
六、 实验总结	29

一、 题目及要求

1. 标题：一种领域特定脚本语言的解释器的设计与实现
2. 描述：领域特定语言 (Domain Specific Language, DSL) 可以提供一种相对简单的文法，用于特定领域的业务流程定制。本作业要求定义一个领域特定脚本语言，这个语言能够描述在线客服机器人（机器人客服是目前提升客服效率的重要技术，在银行、通信和商务等领域的复杂信息系统中有广泛的应用）的自动应答逻辑，并设计实现一个解释器解释执行这个脚本，可以根据用户的不同输入，根据脚本的逻辑设计给出相应的应答。
3. 基本要求：
 - a) 脚本语言的语法可以自由定义, 只要语义上满足描述客服机器人自动应答逻辑的要求。
 - b) 程序输入输出形式不限，可以简化为纯命令行界面。
 - c) 应该给出几种不同的脚本范例, 对不同的脚本范例解释器执行之后会有不同的行为表现。

二、 设计思路

从题目和要求得知，不论是解释器的设计，还是最后的应用，首先应该归根于脚本语言的设计，如果能够设计出脚本语言的语法格式，并写出一个测试样例，根据测试样例设计规定数据结构，设计出解释器，最后才可以继续写出其他的脚本语言进行测试。因此，本次的作业流程就十分清晰了：



三、 语法设计（脚本语言部分）

i. 关键字（保留字）

脚本语言中一共有 6 个关键字，分别是：

"robot_name", "window_size", "begin", "jump", "else", "default"

下面将对它们的含义及使用方式进行介绍。

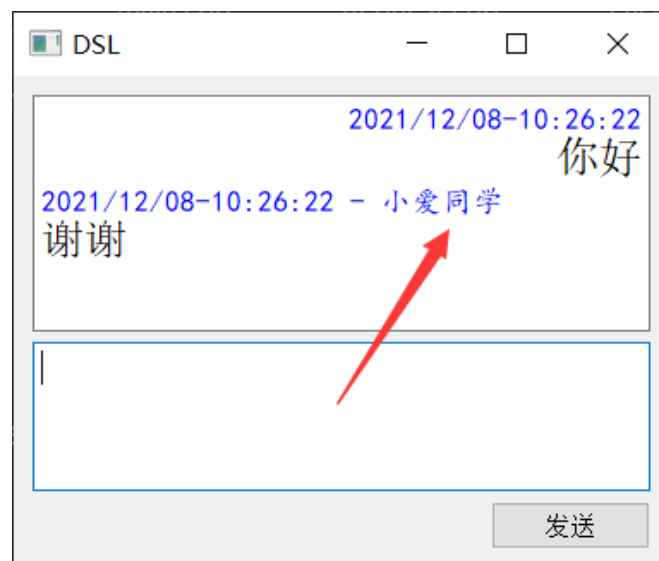
a) robot_name

robot_name 是在程序开头（即第一个 begin 关键字之前）添加的能够为机器人命名的关键字，所接受的数据类型为字符串类型（详见“语法结构”——“运算符”部分），如果程序中没有为 robot_name 赋值，即不存在 robot_name 时，默认机器人的名字为“robot”。

例：

```
robot_name = “小爱同学”;  
  
begin 0:  
    “你好”->“谢谢”;  
    ...  
  
begin 1:  
    ...
```

结果为：



b) window_size

`window_size` 是在程序开头（即第一个 `begin` 关键字之前）添加的能够为人机交互窗口调整初始大小的关键字，所接受的数据类型为两个整型数字，第一个数字表示窗口宽度，第二个数字表示窗口高度。如果程序中没有为 `window_size` 赋值，即不存在 `window_size` 时，默认窗口大小为 `1100*688`。

例：

```
window_size = (800,600);  
  
begin 0:  
    ...  
  
begin 1:  
    ...
```

c) begin

`begin` 在整个程序中起到了关键性作用，它表示的是一个**对话块**，整个对话块从 `begin` 的下一行开始，一直到下一个 `begin` 的上一行，或者到程序结束时为止；在使用 `begin` 时，应该在 `begin` 后添加一个整型数字表示当前处于哪一个对话块，并以冒号结尾，表示对话块的开始，**程序中物理顺序靠前的对话块在程序运行时首先进入**。

需要注意的是：在同一个时间，程序只会执行同一个对话块，除非根据用户的输入，激活了在某一个对话块中的 `jump` 指令时才会发生对话块的跳转，不然会一直执行同一个对话块中的内容。

例：

```
robot_name = “小爱同学”;  
  
window_size = (800,600);  
  
begin 0:  
    ...  
    “你好”->jump 1;  
    “谢谢”->“不客气”;  
    ...  
  
(未写出 begin 1 的具体内容)
```

其含义为：程序开始运行时，进入对话块 0，当用户输入“你好”，程序会进入对话块 1 继续执行，如果用户输入“谢谢”，程序会回复“不客气”并停留在对话块 0 中而不进行跳转。

d) jump

在介绍 begin 时大致介绍了 jump 的使用方法，的确 jump 只能用来进行语句块的跳转，需要在 jump 后面使用空格并加上想要跳转的语句块编号，这里我们简单说明 jump 可能出现的位置：

1. 如同介绍 begin 时的例子，jump 关键字可以在用户输入相应的语句后直接进行跳转，而程序并不会回复任何内容，在此不再说明。
2. 然而常常我们需要在正常的对话过程中实现跳转，这就需要程序回复给用户一段话后，再进行跳转，如需要这种功能，应该在“->”符号后添加字符串，换行后再写出 jump 语句，以“;”结尾，这样就可以完成回复后进行跳转。（更多此种方式的使用细节详见“语法设计”——“运算符”部分）

例：

```
window_size = (800,600);  
begin 0:  
    ...  
    “你好”->jump 1;           /直接跳转到对话块 1 而不进行回复/  
    “谢谢”->“不客气”       /回复“不客气”并跳转到语句块 2/  
        jump 2;  
    ...  
(未写出 begin1 和 begin2 的具体实现)
```

e) else

else 是出现在语句块中的一个关键字。在一个语句块中，由于程序不可能做到完全匹配用户的每一句话而给出相应的回复，这时候我们可以在语句块的末尾（或者中间部分，但是写在末尾逻辑上较为通顺）添加 else 语句，如果用户没有输入语句块中的其他任意一句话，那么程序会自动输出 else 语句后的部分，假如不想程序在用户输入某些话后而让 robot 无动于衷，那么 else 语句几乎成为了每一个语句块中的必备语句，通常可以用来引导用户输

入，从而使程序正常运行下去。

例：

```
window_size = (800,600);

begin 0:

    ...

    “你好”->“你好呀”;

    “谢谢”->“不客气”;

    else->“现在我只听得懂“你好”和“谢谢”而已啦”;

    ...
```

f) default

`default` 是在程序开头（即第一个 `begin` 关键字之前）添加的关键字，它的作用是可以声明在程序运行开始时由 `robot` 说出的一段话，正确使用 `default`，可以让用户明确输入内容以及一些其他功效。

例：

```
window_size = (800,600);

default = “请输入“你好”或者“谢谢””;

begin 0:

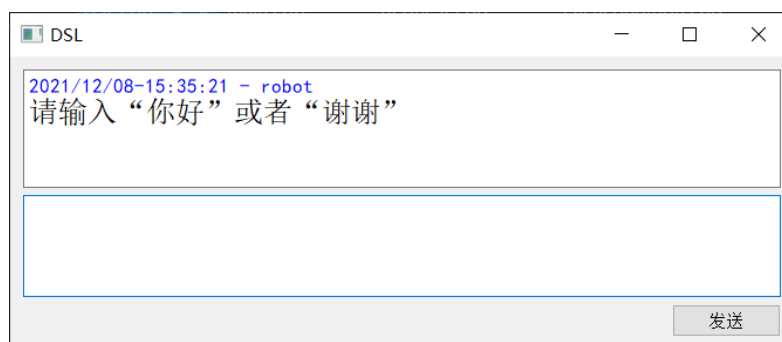
    ...

    “你好”->“你好呀”;

    “谢谢”->“不客气”;

    else->“现在我只听得懂“你好”和“谢谢”而已啦”;
```

其表现为：



ii. 运算符（操作符）

我们在上面关键字的介绍中已经看到了一些运算符的应用, 包括冒号、引号、小括号等, 现在对它们出现的位置或者用法做一一介绍:

(注: 这里出现的所有运算符都应为英文运算符)

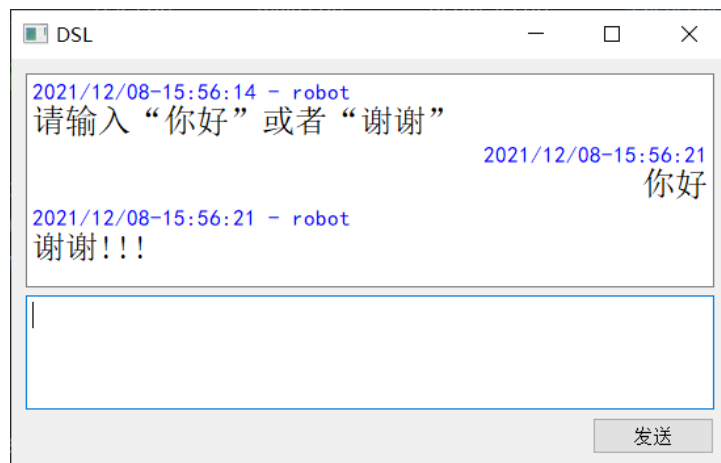
a) 加号 (+)

加号可以出现在程序中任意含有字符串的位置处 (除了声明 robot_name), 简要来说它的作用是连接两个字符串, 使这两个字符串连接为一个长字符串。

例:

```
default = “请输入“你好”或” + “者“谢谢””;  
begin 0:  
    ...  
    “你好”->“谢谢”+“!!!”;  
    ...
```

结果为:



这么看来加号好像并没有什么作用, 其实这里只是演示一种使用方式, 一般来说, 大多数情况下, 加号是配合变量来使用的, 如果有大量语句需要重复同一句话, 那么可以先设置变量, 再在需要使用的地方进行加法连接操作。(详见“语法设计——变量”部分)

b) 减号 (-), 大于号 (>)

减号在程序中会出现在两个位置:

1. 用于声明语句块的编号, 即 begin 之后, 表示当前语句块的编号为一个负整数。
2. 在之前的例子中已经频繁出现, 它和大于号(>)一并组成了一个运算符“->”, 在这个运算符左边表示的是用户输入, 右边表示的是 robot 回复的内容, 称作**回复块**, 也就是输出。“->”左右是对应的关系, 一个输入应对应着一组输出。

在整个程序中, 大于号 (>) 只会出现在上述位置中。

c) 引号 (“ ”)

程序中出现的引号一定为英文引号! 字符串中不允许出现英文引号!!!

引号可能是程序中出现最多的符号, 被它包括的就是我们一直再提的字符串, 不论是从用户方接受的, 或是期望回复给用户的语句, 除了变量之外, 均需要使用引号包裹, 从而告诉程序这里面的东西和程序逻辑无关, 按照其内容接收和输出就好。

当然, 我们看到的对 robot_name 和 default 的赋值, 也都用到了字符串。

d) 等号 (=)、括号、冒号 (:)

等号在程序中会出现在以下方面: 为 robot_name、default、window_size、变量赋值, 表示将等号右边的内容交给左边的变量或关键字, 而在对 window_size 进行赋值时, 需要将它接受的两个整型数字用括号包裹, 数字间使用逗号隔开。

冒号出现在程序中 begin 关键字后, 在使用 begin 时, 应该在 begin 后添加一个整型数字表示当前处于哪一个对话块, 并以冒号结尾。

e) **※**逗号 (,), 分号 (;)

除了使用逗号将为 window_size 赋值的两个整型数字分隔开, 他还和分号一并控制着程序的输出, 即“->”的右边部分。

1. 逗号的单独使用

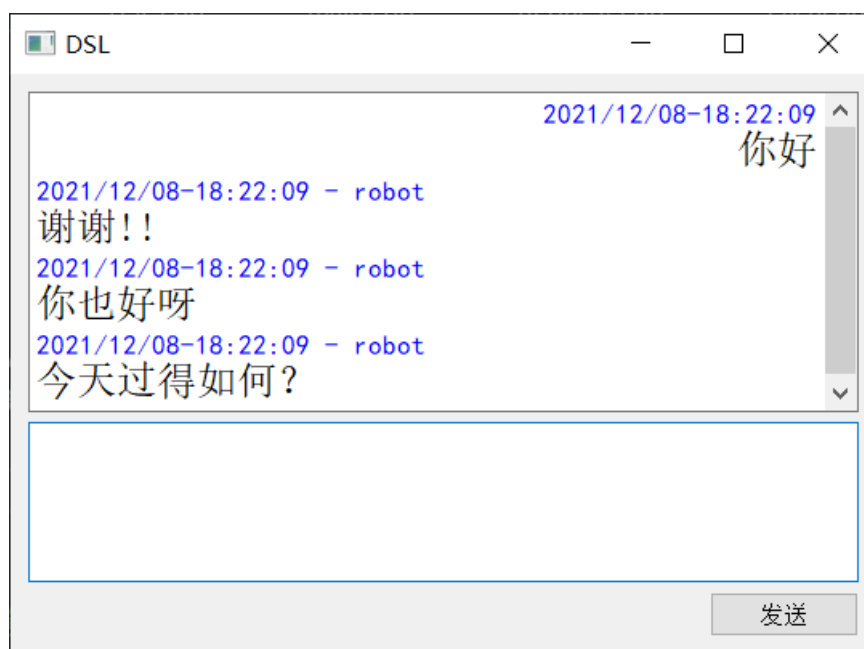
我们在之前已经看到了回复语句的基本使用方法, 即一个接受对应一个回复, 但是实际

上回复的语句可以不仅是一条内容，还可以是多条内容，只需要使用逗号将字符串进行分隔就可以了。

例：

```
begin 0:
...
“你好”->“谢谢”+“!!”，“你也好呀”，“今天过得如何？”；
...
```

结果为：



上述逗号的单独使用方法同样适用于 default 的赋值操作。

2. 分号的单独使用

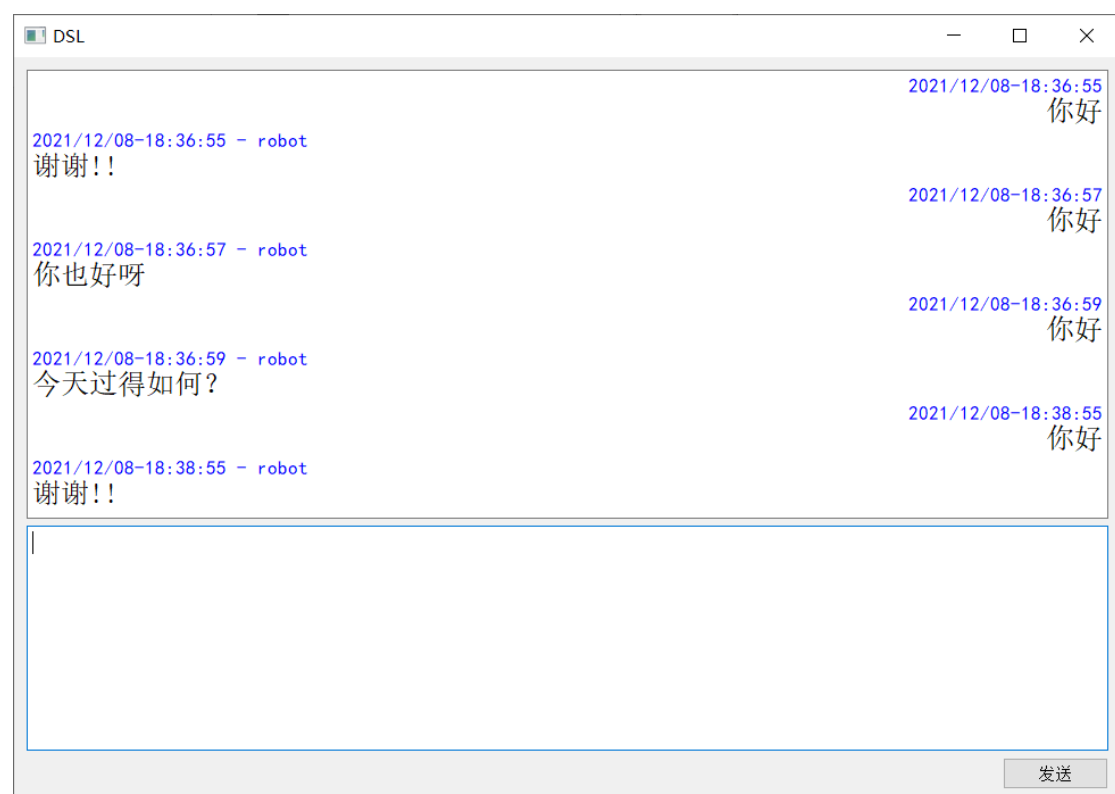
在程序中，为 robot_name、default、window_size、变量赋值这些操作都只会占用一行，因此，它们的末尾必然需要使用分号结尾。

而在程序的对话块中，若一条“接收->回复”语句只占用一行（包括“逗号的单独使用”情况），那么也只需要在行尾添加分号即可，这种类型我们在之前的例子中已经见到，但实际上，回复语句可以不只占用一行，如果具有多行回复语句（直到回复块结束前的换行均不需要加任何符号），它的含义是：如果接收到相同的字符串，即用户重复发送相同的句子，那么每重复一次，程序所回复的语句就向下查找一行，直到整个回复块结束（需使用分号结尾），若再发生重复，则回到第一行回复继续重新输出。

例：

```
begin 0:
    ...
    “你好”->“谢谢”+“!!”
        “你也好呀”
        “今天过得如何? ”;
    ...
```

结果为：



3. 逗号与分号的混合使用

我们看到在具有多条语句输出时一行中逗号的使用，还有循环输出时分号的使用，一个在行上有所扩展，一个在列上有所扩展，它们其实可以综合起来使用，只需要按照上述条件行列均扩展就可以实现。

例：

```
begin 0:

    ...

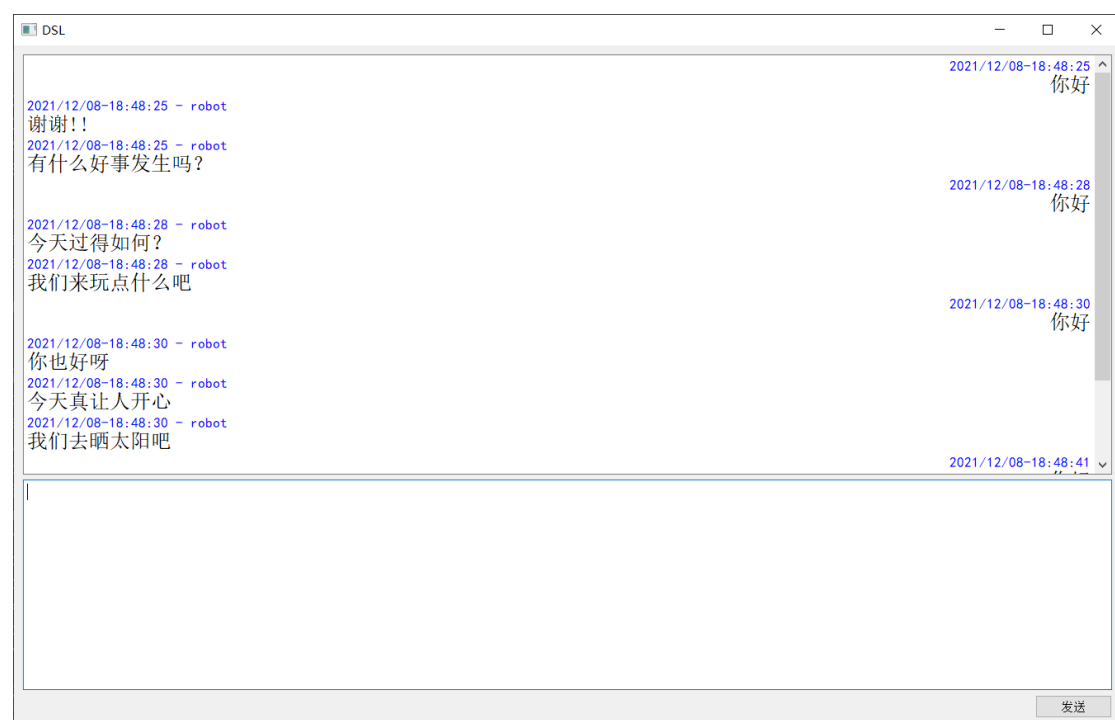
    “你好”->“谢谢”+“!!”，“有什么好事发生吗？”

        “今天过得如何？”，“我们来玩点什么吧”

            “你也好呀”，“今天真让人开心”，“我们去晒太阳吧”；

    ...
```

结果为：



4. 逗号、分号与 jump

虽然逗号、分号、jump 同属于回复语句块，但是也不是都可以交叉使用的。

当程序中接受到用户的某一句话（以下称作 accept）需要进行跳转时，除了 jump 的第一种“无回复跳转”使用方法之外，在第二种“有回复的跳转”中，只能使用“逗号的单独使用”这种方法，即只能在第一行填写想要回复的语句，必要时可以使用逗号分隔，在第二行则必须使用 jump 指令进行跳转，在 jump 语句后使用分号表示回复块的结束。

原因： jump 语句实际上与 accept 一一对应，并不是回复块中的某一行具体回复对应，因此在接受到 accept 的时候就已经完成了跳转；并不是说这种写法不可以，只是在再一次接受到 accept 时，程序执行的已经不是当前对话块了，因此除了第一行的回复，剩下的回复永远不可能被访问到。

例：

```
begin 0:
    ...
    “你好”->“谢谢”+“!!”，“有什么好事发生吗？”
        “今天过得如何？”，“我们来玩点什么吧”
        “你也好呀”，“今天真让人开心”，“我们去晒太阳吧”；
    jump 2;
    ...
begin 2:
    ...
    “你在干什么”->“想吓你一跳，居然被你发现了！”；
    ...
```

结果为：



iii. 变量

程序中只允许出现字符串变量一种变量，且变量的声明只能出现在程序开头（即第一个 `begin` 关键字之前）。

变量的命名方式：以字母或下划线开头，其后可以是有限个字母、数字、下划线的任意组合。

变量的初始化：“变量名=字符串;”

已初始化的变量可以使用加号与其他已初始化的变量或字符串连接从而初始化其他变量。

注意：default 和 robot_name 的声明不能使用变量！

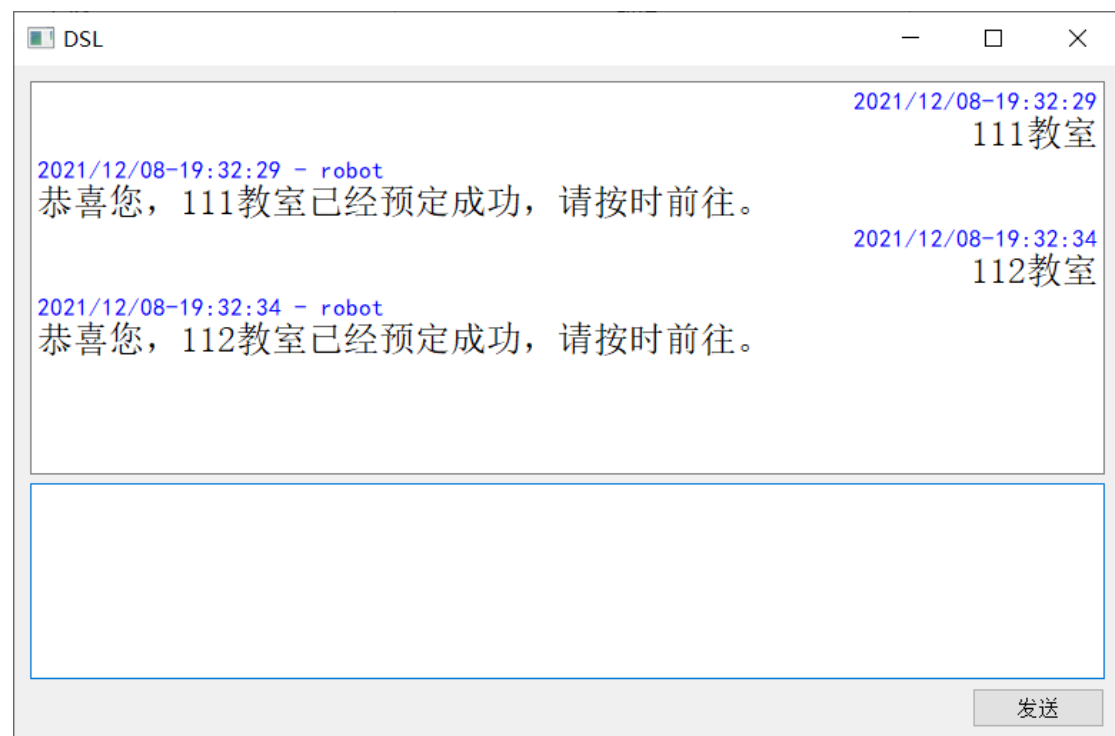
变量的作用域：所有对话块内部。

变量的作用方式：使用加号与其他变量或字符串连接，从而简化重复的输出。

例：

```
a = “恭喜您， ”;
b = “已经预定成功，请按时前往。”;
begin 0:
    ...
    “111 教室”->a + “111 教室” + b;
    “112 教室”->a+“112 教室”+b;
    ...
```

结果为：



iv. 注释

在程序中的任意两个位置（不包括字符串内部）添加“/”，即可注释掉其中的内容。

注：即使是单行注释，也需要将需要注释的内容首尾添加“/”。

四、 解释器设计

i. 整体框架

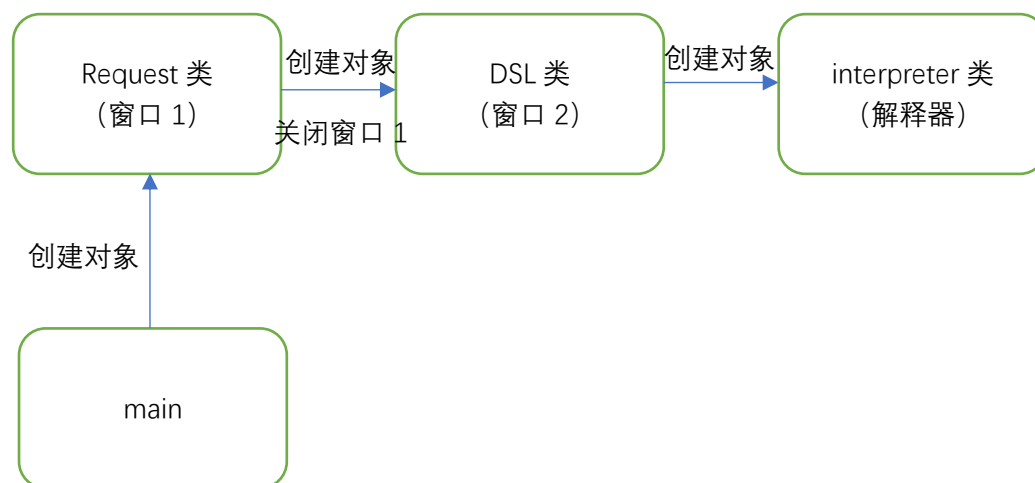
考虑到自动化测试程序，首先应该完成三个模块：

窗口 1：请求用户输入测试文件名，并告知是否采用自动化测试。

窗口 2：对话窗口，来完成客户与机器之间的交互。

解释器：读取用户输入的文件名，分析文法，并保存在相应的数据结构中。

即：



ii. 关键实现

a) Request 类

在 Request 类中完成的事情比较简单，它只需要创建窗口获取用户的输入即可，规定用户只能采用自动化测试和非自动化测试的方式，Request 类需要对两种方式加以区分，从而创建相应的 DSL 对象。

- 所用函数

```
Request(QWidget *parent = Q_NULLPTR); //构造函数, 创建窗口  
void getfile(); //获取用户在 Request 窗口的输入并分析
```

- 数据结构

```
string input; //存放用户在 Request 窗口的输入  
DSL* dialog = NULL; //创建 DSL 窗口对象
```

- 部分伪代码

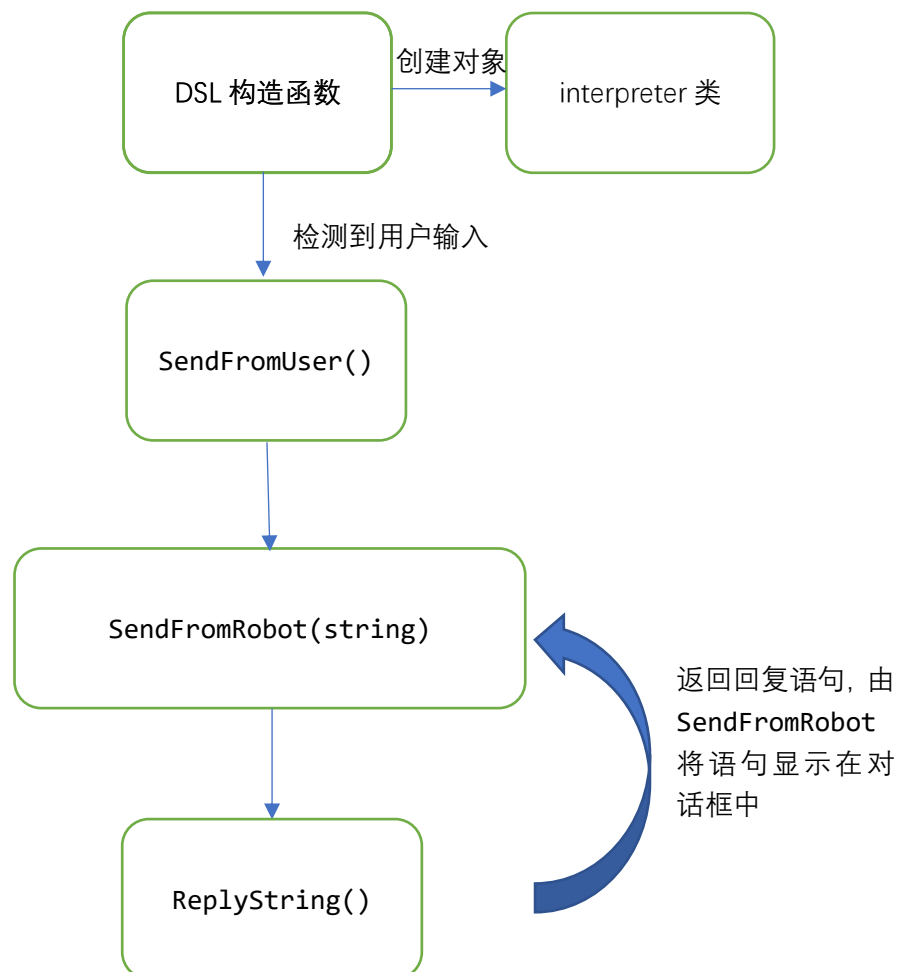
```
void getfile() //获取用户在 Request 窗口的输入并分析  
{  
    input = request.lineEdit->text() //获得用户输入  
    if ("auto" in input) {  
        输入中含有 auto, 说明为自动化测试  
        获取脚本文件名保存在 textfile 中  
        获取输入文件名保存在 testfile 中  
        /*使用 textfile 和 testfile 初始化并创建对话框*/  
        dialog = new DSL(Q_NULLPTR, textfile, testfile);  
        dialog->show();  
    }  
    else{  
        普通手动测试  
        获取脚本文件名保存在 textfile 中  
        testfile 置为空  
        /*使用 textfile 和 testfile 初始化并创建对话框*/  
        dialog = new DSL(Q_NULLPTR, textfile, testfile);  
        dialog->show();  
    }  
}
```

b) DSL 类

DSL 类的主要功能是维护一个对话窗口, 它初始化时创建了 interpreter 类的对象 inter, inter 中包括了脚本程序中的所有信息, 因此 DSL 只需要从对话框中接收用户输入, 根据用户输入在 inter 对象中寻找合适的输出, 并按照要求打印在对话框中即可。

● 所用函数

```
DSL(QWidget* parent, string text, string test); // 创建 interpreter 类的对象, 并创建对话窗口
void SendFromUser(); // 获得用户在对话窗口中的输入, 将它显示在对话框中, 并调用 SendFromRobot
void SendFromRobot(string); // 调用 ReplyString, 并将回复语句显示在对话框中
vector<string> ReplyString(); // 根据 user_send 分析回复内容, 返回回复语句
```



● 数据结构

```
string robot_name; // 对话机器人的名字
string user_send; // 用户在 DSL 窗口中的输入
string textfile; // 需要解释的 DSL 程序文本名
string testfile; // 自动化测试所需的输入文本名
```



```

vector<string> output;//记录 robot 在整个对话过程中回复的所有内容
interpreter inter;//创建一个 interpreter 对象，分析脚本程序
Ui::DSLClass dsl;//创建对话框
string last_from_user = "";//记录用户上一次输入的内容
int repeat_of_user = 0;//记录用户重复同一句话的次数
int current_state;//记录当前对话处于脚本程序中的哪一个对话块

```

- 部分伪代码

```

vector<string> DSL::ReplyString() {
    vector<string> str;//存放查找出的回复语句
    for (int i = 0; i < inter.state[current_state].size(); i++) {
        if (inter.state[current_state][i].accept == "") {
            记录序号 i, i 为对话块的 else 的序号
        }
        /*找到用户输入对应的回复块*/
        if(inter.state[current_state][i].accept == user_send){
            /*用户重复发送相同语句*/
            if (last_from_user == user_send){
            }
            else{
            }
            /*如果引发了跳转*/
            if (inter.state[current_state][i].jump){
            }
        }
    }
    /*如果对话块全部遍历完依然未找到回复块，说明需要恢复 else 的回复块*/
    if (str.size() == 0 && else_index != -1){
        /*依然进行判断：是否重复发送，是否引发跳转*/
    }
}

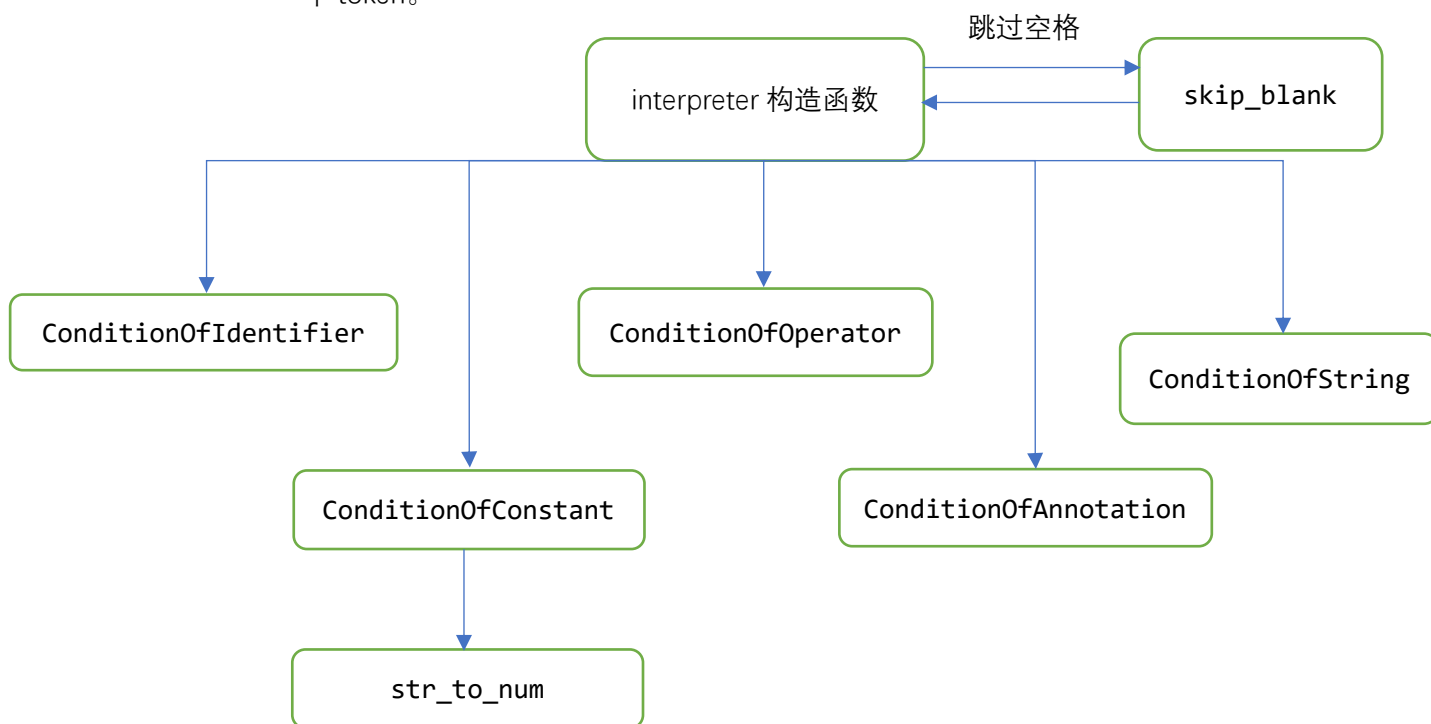
```

c) *interpreter 类

interpreter 类只要负责读取脚本程序，并保存在合适的数据结构中。

如：由于对话块的索引都是唯一的，我们可以使用 map 来保存对话块索引以及对话块，这样能很快根据索引找到对话块的内容。

读取脚本程序的方法：类比编译原理中的词法分析过程，按照每一行读取，根据读取到的内容不同转移至不同状态（运算符、关键字、数字等），直到读取结束，所读取到的内容应该为一个 token，根据 token 的类型进行对应的操作，然后再读取这一行中的下一个 token。



● 所用函数

```
interpreter(string file); // 构造一个解释器对象，解释 file 中的 DSL 程序，并保存在上述数据结构中
int str_to_num(string); // 将一个全部为数字的字符串类型转变为整型
void skip_blank(int& , string); // 跳过 DSL 程序中的空格
void ConditionOfIdentifier(int&, string); // 为 IDENTIFIER 状态时进入该程序，读取一个关键字或变量名作为 token
void ConditionOfConstant(int&, string); // 为 CONSTANT 状态时进入该程序，读取一个常量作为 token
void ConditionOfOperator(int&, string, int&); // 为 OPERATOR 状态时进入该程序，读取一个运算符作为
```

token

```
void ConditionOfString(int&, string);//为 STRING 状态时进入该程序, 读取一个字符串作为 token
```

```
void ConditionOfAnnotation(int&, string&, ifstream&);//为 ANNOTATION 状态时进入该程序, 读取注释并跳过
```

- 数据结构

```
public:

    /*存储对话信息的数据结构*/
    typedef struct grammer {

        string accept = "";//接收到的用户输入

        vector<vector<string>> response;//输入对应的回复块

        bool jump = false;//accept 是否对应跳转

        int next_state = -1;//调转的语句块编号

    }gram;

    string filename;//所要进行分析的脚本程序文件名

    unordered_map<int, vector<gram>> state;//对话块—对话内容结构*/
    unordered_map<string, string> str_variate;//变量名—变量内容结构*/
    pair<int, int> window_size = { 0,0 };//保存脚本程序中定义的窗口尺寸
    vector<string> default_response; //保存脚本程序中定义的初始回复
    string robot_name = "robot";//保存脚本程序中定义的 robot_name

private:

    /*

        以下为解释 DSL 程序过程中用到的标识,

        如 isRobotName == true

        那么下一个读取的字符串将赋值给 robot_name

    */

    bool isDefault = false;

    bool isRobotName = false;

    bool isWindowSize = false;

    bool isBegin = false;
```

```

bool isAccept = false;

bool isResponse = false;

int begin_index = 0;//记录对话块的索引

int dialog_index = 0;//记录对话的索引

int string_index = 0;//记录回复的索引

string broken_str = "";//由于支持字符串之间的连接操作，故需要设置
broken_str 保存当前读取到的字符串

string variate = "";//保存当前读取的用户自定义的变量

```

- 部分伪代码

```

interpreter::interpreter(string file)
:filename(file)
{
    while (getline(f, str)){//读取脚本程序每一行
        while (i < (int)str.size()){//读取每一行的每一个字符
            switch (state) {
                case INITIAL://初状态，根据首字符判断 token 类型
                    /*跳过空格*/
                    skip_blank(i, str);
                    /*如果读取到字母或下划线，则进入标识符状态*/
                    if (isalpha(str[i]) || str[i] == '_') {
                        state = IDENTIFIER;
                    }
                    /*如果读取到数字，则进入常量状态*/
                    else if (isdigit(str[i])) {
                        state = CONSTANT;
                    }
                    /*如果读取到运算符，则进入运算符状态*/
                    else if (OPERATE.count(string(1, str[i])) == 1) {
                        state = OPERATOR;
                    }
                    /*如果读取到'/'，则进入注释状态*/
                    else if (str[i] == '/') {
                        state = ANNOTATION;
                    }
                    else {
                        state = ERROR;
                    }
                    break;
                /*根据不同状态跳转进入每一种 token 的分析函数中*/
                case IDENTIFIER:

```

```

        ConditionOfIdentifier(i, str); //当 token 为标识符时
        state = INITIAL;
        break;
    case CONSTANT:
        ConditionOfConstant(i, str); //当 token 为常量时
        state = INITIAL;
        break;
    case OPERATOR:
        ConditionOfOperator(i, str, state); //运算符
        if (state != STRING && state != CONSTANT)
            state = INITIAL;
        break;
    case ANNOTATION:
        ConditionOfAnnotation(i, str, f); //注释
        state = INITIAL;
        break;
    case STRING:
        ConditionOfString(i, str); //字符串
        state = INITIAL;
        break;
    default:
        printf("ERROR");
    }
}
}
}
}

```

我们以字符串状态函数 ConditionOfString(i, str)为例，它的伪代码如下：

```

/*为 STRING 状态时进入该程序，读取一个字符串作为 token*/
void interpreter::ConditionOfString(int& i, string str) {
    string token = "";
    /*直到下一个引号为止，期间均为字符串内容*/
    for (; i < (int)str.size() && str[i] != '"'; i++) {
        token += str[i];
    }
    i++;
    /*如果当前 string 为 robot_name 或是 default_response*/
    /*在读到 robot_name 或 default_response 时，isRobotName 或 isDefault 就已经赋值为 true*/
    if (isRobotName) {

```

```

        robot_name = token;

        isRobotName = false;
    }
    else if (isDefault) {
        broken_str += token; //default 支持字符串连接，故优先存放在变量 broken_str 中
    }

    /*其他情况下，首先做字符串的连接，若已经到达行尾，且属于 response，则将字符串加入相应位置的 response
    中，接着为下一个 response 开辟空间*/

    else {
        broken_str += token;

        if (isResponse && i >= (int)str.size()) {
            //将字符串加入相应位置的 response 中，接着为下一个 response 开辟空间
        }
    }
}

```

d) 自动化测试程序的添加

我们刚刚提到了，如果在 request 的窗口中输入“auto 脚本文件名 测试文件名”，那么程序将进入自动化测试模式，现在需要在已经写好的程序中添加这一功能，这个时候需要格外小心，一定要先考虑好哪些程序是共通的，并且应该在哪里添加什么。

首先，我理解的自动化测试就是让程序从已经写好的测试输入中读取输入，并且模拟用户发送给 robot，robot 作出回应，并且将回应保存下来，与已经写好的正确输出的文件进行对比，如果完全一致，则通过自动化测试。那么，至少可以确定应答逻辑是不用修改的，只需要在 robot 回应每一句话时将这句话保存到一个容器中，最后关闭窗口后写入文件并检查即可。那么就需要在发送逻辑中做文章，我们很快能想到，在 DSL 类中有一个 SendFromUser()函数，它的作用是：**获得用户在 DSL 窗口中的输入，将它显示在对话框中，并调用 SendFromRobot**，由此分析，我们只需要判断用户是否需要的是自动化测试，如果是，则从文件读取输入；如果不是，则按照原来的逻辑，获得用户在 DSL 窗口中的输入即可。

在 SendFromUser()中添加:

```
/*testfile 不为空, 则为自动化测试, 直接从 user_send 获取到输入*/
if (testfile != "") {
    str = pCodec->toUnicode(user_send.c_str(), user_send.length());
}
/*不为自动化测试, 从用户输入框获取输入*/
else {
    str = dsl.textEdit_2->toPlainText();
    user_send = pCodec->fromUnicode(str).data();
}
```

由于在一开始的逻辑中, SendFromUser()函数被绑定在“发送”按钮以及快捷键“ctrl+enter”上, 因此只有在监听到这两个信号才会跳转, 但是自动化测试并不会产生这两个信号, 所以除了以上改动还需要再 DSL 构造函数中手动调用 SendFromUser()函数:

```
/*如果 testfile 不为空, 说明用户采用的是自动化测试方式, 需要从文件读入输入语句*/
if (testfile != "") {
    ifstream f;
    f.open(testfile, ios::in);
    while (getline(f, user_send)) {
        SendFromUser();
    }
    f.close();
}
```

此外, 为了检验输出内容与正确的输出结果是否一致, 还需要在最后检查这两个文件的内容是否一致, 我将这个部分放在了整个程序的最后, 也就是 main 函数中, 如果正确, 则在输出文件最后打印 correct, 如果错误, 打印 incorrect, 并打印出不一致的行号, 至此, 自动化测试程序就补充完毕了。

五、 样例测试

程序同文件夹下有四个测试样例，这里选用第三个作为例子进行演示：

注意：凡是涉及到的需要读取或者输出的文件，均采用 GBK 编码格式。

测试脚本程序：

```
default = "您好，欢迎来到航班购票系统","请问您的目的地是哪里呢？（北京、广州）";
robot_name = "tickets";
window_size = (1200,800);

ask_des = "请问您的目的地是哪里呢？（北京、广州）";
ask_date = "请问您希望哪天出发呢？（明天、后天）";
ask_id = "请输入您期望的航班序号。";
success1 = "恭喜您，飞往";
success2 = "的航班预定成功，航班号：";

begin 0:
"北京"->ask_date
    jump 1;
"广州"->ask_date
    jump 2;

else->ask_des;

begin 1:
"明天"->"明天飞往北京的航班有：","1.上午十点","2.下午四点",ask_id
    jump 3;

"后天"->"后天飞往北京的航班有：","1.上午十一点","2.下午两点",ask_id
    jump 4;

"广州"->"请选择明天或后天飞往广州的航班"
    jump 2;

else->ask_date;

begin 2:
"明天"->"明天飞往广州的航班有：","1.上午十点","2.下午四点",ask_id
    jump 5;
"后天"->"后天飞往广州的航班暂未开售！";
"北京"->"请选择明天或后天飞往北京的航班"
    jump 1;
else->ask_date;
```



```

begin 3:
"1"->success1 + "北京" + success2 + "1.明天上午十点","程序已退出"
    jump 6;
"2"->success1 + "北京" + success2 + "2.明天下午四点","程序已退出"
    jump 6;
else->ask_id;

begin 4:
"1"->success1 + "北京" + success2 + "1.后天上午十一点","程序已退出"
    jump 6;
"2"->success1 + "北京" + success2 + "2.后天下午两点","程序已退出"
    jump 6;
else->ask_id;

begin 5:
"1"->success1 + "广州" + success2 + "1.明天上午十点","程序已退出"
    jump 6;
"2"->success1 + "广州" + success2 + "2.明天下午四点","程序已退出"
    jump 6;
else->ask-id;

begin 6:
else->ask_des
jump 0;

```

自动化测试输入：

```

北京
广州
北京
明天
1
132
广州
后天
?
明天
2

```

自动化测试的正确输出（**结尾无空行!!!!**）：

```

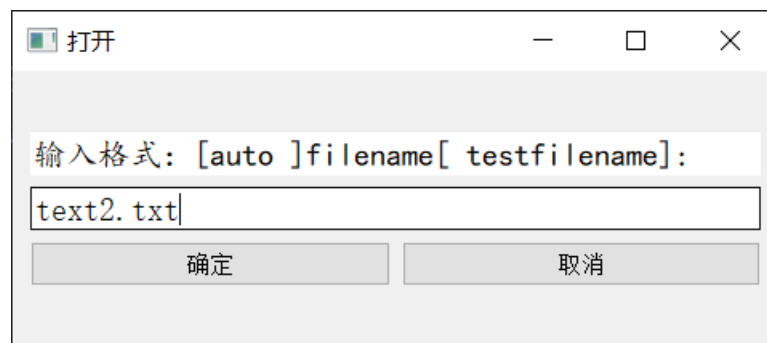
您好，欢迎来到航班购票系统
请问您的目的地是哪里呢？（北京、广州）
请问您希望哪天出发呢？（明天、后天）
请选择明天或后天飞往广州的航班
请选择明天或后天飞往北京的航班
明天飞往北京的航班有：

```

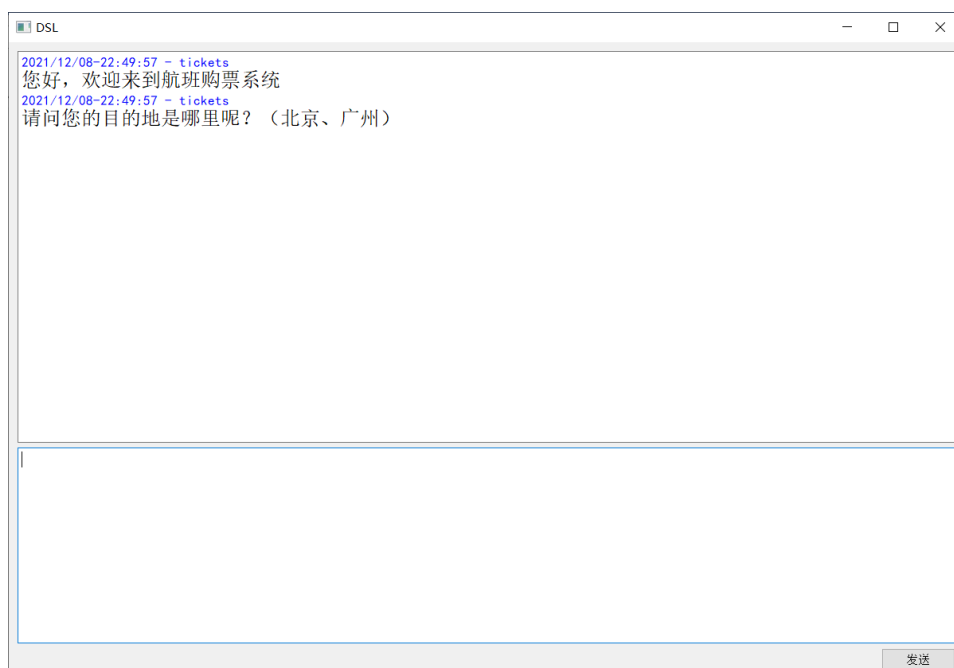
1.上午十点
2.下午四点
请输入您期望的航班序号。
恭喜您，飞往北京的航班预定成功，航班号：1.明天上午十点
程序已退出
请问您的目的地是哪里呢？（北京、广州）
请问您希望哪天出发呢？（明天、后天）
后天飞往广州的航班暂未开售！
请问您希望哪天出发呢？（明天、后天）
明天飞往广州的航班有：
1.上午十点
2.下午四点
请输入您期望的航班序号。
恭喜您，飞往广州的航班预定成功，航班号：2.明天下午四点
程序已退出

- 非自动化测试

首先运行程序，由于是非自动化测试，这里直接输入同文件夹下脚本程序文件名即可：



接着进入对话窗口界面：

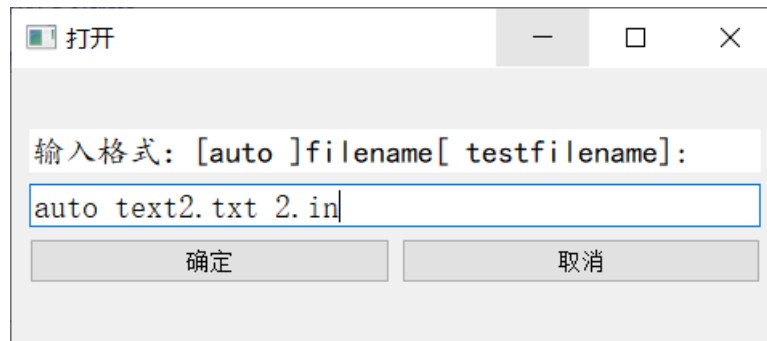


这个时候我们可以随意输入进行测试了（发送快捷键：ctrl+enter）：



- 自动化测试


在第一个窗口中依次输入“auto 脚本程序文件名 输入测试文件名”即可，注意应该提前写好正确输出文件，文件名应当为输入测试文件名的首字符连接上 result.out。



点击确定或者按下回车键：



这个时候关闭窗口，等待程序结束后，我们打开 output.out 进行查看：

 output.out - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

您好，欢迎来到航班购票系统
请问您的目的地是哪里呢？（北京、广州）
请问您希望哪天出发呢？（明天、后天）
请选择明天或后天飞往广州的航班
请选择明天或后天飞往北京的航班
明天飞往北京的航班有：
1. 上午十点
2. 下午四点
请输入您期望的航班序号。
恭喜您，飞往北京的航班预定成功，航班号：1. 明天上午十点
程序已退出
请问您的目的地是哪里呢？（北京、广州）
请问您希望哪天出发呢？（明天、后天）
后天飞往广州的航班暂未开售！
请问您希望哪天出发呢？（明天、后天）
明天飞往广州的航班有：
1. 上午十点
2. 下午四点
请输入您期望的航班序号。
恭喜您，飞往广州的航班预定成功，航班号：2. 明天下午四点
程序已退出
correct

最后输出 correct，说明与已经写好的 2result.out 文件内容一致，即通过自动化测试。

六、 实验总结

通过本次实验，无疑让我获得了巨大收获，不论是写程序之前的构思过程，还是写程序时不断推敲命名和结构，都让我有了全新的体验，虽然命名谈不上理想，但是我确实实在写程序的过程中感受到了命名的重要性，不论是声明后再次使用还是之后看程序时思考其含义，都有很大帮助；而且本次作业刚好能用上在编译原理课上学习到的词法分析技术，整体思想就是使用一个自动机来实现对整个程序的 token 划分，确实是一个很方便的方法，非常容易理解。

在设计中也遇到了不少困难，比如 Qt 的使用，虽然之前只简单学习了一点点理论知识，而且这一次也并不算需要做一个很复杂的图形界面，一开始以为只是用来熟悉熟悉 Qt 的使用而已，谁知道居然有各种问题，比如添加监听、两个窗口的切换……都造成了不小的困扰，还有就是脚本程序的分析办法，也模拟了很多次，耗费了相当长的时间才得以解决。

总体来说，有了完成这次实验的经历，确实让我对于注释、接口、命名、记法等一系列课上的内容融入代码从而重新审视代码，这种体验十分宝贵，我相信在以后的程序设计中一定还能够时时刻刻关注着这些问题，不断完善，每一次都会有所收获。