Programming Assignment I

15-李晓畅-凌之寒

1. stack.cl 的代码简述

主要思路就是先构建一个栈类，再在主函数里调用对应的方法就可以了。代码结构大概设计成这样：

```
  1 > class Stack inherits Lib{…
112   };
113
114 > class Lib{…
133   };
134
135   (*
136 > class Main{…
163   };
164   *)
165
166 > class Main{…
189   };
```

Stack 是栈类。

Lib 是一些 COOL 没有提供但或许有用的方法。

注释掉的是一个单独测试 Stack 类的主函数。

在最后的 Main 函数里实现栈语言的交互。

先根据 Cool 语言提供的类与方法构建出栈类：

```
  1  class Stack inherits Lib{
  2      top : Int ;
  3      s : String ;
  4
  5  >   init() : Stack{…
 11      };
 12
 13  >   push(c : String ) :Stack
 19      };
 20
 21  >   top() : String {…
 28      };
 29
 30  >   pop() : String {…
 47      };
 48
 49  >   plus() : Stack{…
 60      };
 61
 62  >   swap() : Stack{…
 73      };
 74
 75  >   e() : Stack{…
 83      };
 84
 85  >   display() : Object{…
102      };
```

因为可能需要压入栈中的只有字符或者字符串，所以用字符串来存储其实就可以了，因而将栈设计为一个字符串。又因为在栈语言中很多字符都是不会出现的，所以可以任意选择一个符号作为元素之间的分隔符。

考虑显示的方便性，这里选择换行符\n 作为分隔符，在初始化时压入一个分隔符做栈底可以方便操作。

```
init() : Stack{
    {
        top <- 0 -1;
        s <- "\n";
        self;
    }
};
```

栈顶指针其实没那么必要，因为假定栈语言输入都是合理的，不过这里还是设计上了。

Init, push, top, pop, plus, swap, e, display 分别实现栈类的初始化、压栈、查看栈顶、出栈、栈顶相加、栈顶交换、评估和展示功能。

Pop, push 和 top 是栈类的基础功能，使用常规方法即可，比如在 push 中：

```
13      push(c : String ) :Stack{
14          {
15          s <- s.concat(c).concat("\n");
16          top <- top + 1;
17          self;
18          }
19      };
```

将新符号连接在字符串后面，并加一个分隔符即可。

```
pop() : String {
    let t : String ,
        p : Int <- s.length()-2
    in
    {
    while (not (s.substr(p,1) = "\n")) loop p <- p-1 pool;
    t <- s.substr(p+1, s.length()-p-2);
        s <- s.substr(0,p+1);
        top <- top-1;
        t;
    }
};
```

如果假定输入的 int 都是一位的话其实会简单很多，这里实现的是不限定位数的一般方法，找到前一个分隔符，输出子串就是栈顶了。

Top 其实可以一次 pop()一次 push()实现。

Plus, swap，display 是基于 push 和 pop 构建的，但是 push 或者 pop 在具体实现上的改动并不会影响其功能,比如在 swap 中：

```
62      swap() : Stack{
63          let t : String <- pop(),
64              t1 : String <- pop()
65          in
66          {
67              push(t);
68              push(t1);
69              self;
70          }
71      };
```

取栈顶两个元素，交换顺序入栈即可。

Plus 也是类似的，取栈顶两个元素，相加再入栈即可。

于是 E 方法只需要按照栈顶元素类型进行处理调用就可以了。

虽然这里约定没有违法输入，但还是写了判断栈空和报空栈的方法。

一些常用的但语言规范没有提供的方法或许可以收集一下?

```
class Lib{
    and(x : Bool, y : Bool) : Bool{…
    };

    or(x : Bool, y : Bool) : Bool{…
    };
};
```

再调用上述的栈类中实现的方法：

```
class Main{
    i : IO <- new IO;
    main() : Object{
        let flag : Bool <- true,
            c : String ,
            stack : Stack <- (new Stack).init()
        in
        while (flag) loop
        {
            i.out_string(">");
            c <- i.in_string();
            if ( c="e") then stack.e()
            else
                if ( c="d" ) then stack.tdisplay()
                else
                    if ( c="x") then flag <- false
                        else stack.push(c)
                    fi
                fi
            fi;
        }
        pool
    };
};
```

输入 x 要正常介绍程序可以类似上面提到的那样实现，即设置循环标记，若不满足则退出循环即可。

就可以实现栈语言了。总体代码可参后附录。

2.实验结果

按照 handout 第 2 部分执行代码，实验结果如下：

```
wojtek@wojtek-virtual-machine:~/test$ spim stack.s
SPIM Version 6.5 of January 4, 2003
Copyright 1990-2003 by James R. Larus (larus@cs.wisc.edu).
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: ../lib/trap.handler
>1
>+
>2
>s
>d
s
2
+
1
>e
>e
>d
3
>x
COOL program successfully executed
```

和预期一致。

3.实验总结

在本次实验中我们搭建了实验环境，实现了栈语言，在此基础上熟悉了 COOL 语言。

COOL 语言是面向对象的语言，和 C++或者 JAVA 有类似的地方，也有不同的地方，基于实验不难发现一些值得注意的细节，比如：

1. 每个**类方法**由一个表达式定义，这个表达式可能是一个**变量**、一个**代码块**{}，表达式的值就是方法的返回值，故经常出现**大括号内包含大括号**的情况。方法的结束大括号}后需要添加;。
2. if, while 等结构后需要跟 then, loop 等关键字，不能直接跟表达式。
3. if, while 等结构也是一种**表达式**，也有值。当需要包含多个表达式时，要使用{}代码块。
4. Local Variable 需要用 let 关键字定义，不能**直接在代码段中定义**。

虽然使用上很不方便，尤其是同其他流行高级语言相比，但我们也不难理解其意义，这些不太符合习惯的约定在实现编译器时似乎可以带来一些方便。

另外，实际使用 COOL 语言编程时，很容易感觉到它太小了。很多高级语言中都提供的基础特性都没有提供：语言支持的不是比较，而是大于；语法非常简单，“标准库”只包含几个基本类；虽然编译器支持多个源文件作为输入，但不支持单独编译；每个 Cool 程序都必须定义一个 Main 类，它必须有一个执行流程开始的 no-args main 方法；不支持命名空间——这无疑带来了一些障碍。

但同时，我们也会感觉到其囊括的内涵其实是很丰富的，拥有现代编程语言的很多特征，包括对象，自动内存管路，静态打印，简单的映射。

总之，我们可以看出设计者在其中的一些精妙的构思，这也是值得我们学习的地方。

附录

总体实现代码如下：

```
class Stack inherits Lib{
    top : Int ;
    s : String ;

    init() : Stack{
        {
            top <- 0 -1;
            s <- "\n";
            self;
        }
    };

    push(c : String ) :Stack{
        {
        s <- s.concat(c).concat("\n");
        top <- top + 1;
        self;
        }
    };

    top() : String {
        let t : String <- pop()
        in
        {
            push(t);
            t;
        }
    };

    pop() : String {
        let t : String ,
```

```
            p : Int <- s.length()-2
        in
        {
        while (not (s.substr(p,1) = "\n")) loop p <- p-1 pool;
        t <- s.substr(p+1, s.length()-p-2);
            s <- s.substr(0,p+1);
            top <- top-1;
            t;
        }
    };

    plus() : Stack{
        if (not gt2()) then {emptys();self;} else
            let t : String <- pop(),
                t1 : String <- pop(),
                trans : A2I <- new A2I
            in
            {
                push(trans.i2a((trans.a2i(t1)+trans.a2i(t))));
                self;
            }
        fi
    };

    swap() : Stack{
        let t : String <- pop(),
            t1 : String <- pop()
        in
        {
            push(t);
            push(t1);
            self;
        }
    };

    e() : Stack{
        let t : String <- top()
        in
        if (t = "+") then {pop(); plus();} else
            if (t = "s") then {pop();swap();} else
                self
            fi
        fi
    };

    display() : Object{
        let p : Int <- s.length() -2 ,
            e : Int <- s.length() -2,
            i : IO <- new IO,
            t : String
        in
            while (0 < p) loop
            {
                while (not (s.substr(p,1) = "\n")) loop
                    p <- p-1
                pool;
                t <- s.substr(p+1, e-p);
                p <- p-1;
                e <- p;
                i.out_string(t.concat("\n"));
            }
            pool
    };

    emptys(): Object{
        {
        (*(new IO).out_string("Empty stack!\n".concat(s));
        abort();
           *)
```

```
            self;
            }
        };

        isempty() : Bool{
            if (top = (0-1)) then true else false fi
        };

        gt2() :Bool{
            if (0 < top) then true else false fi
        };
    };

    class Lib{
        and(x : Bool, y : Bool) : Bool{
            if (x) then
                if (y) then true
                else false
                fi
            else
                false
            fi
        };

        or(x : Bool, y : Bool) : Bool{
            if (x) then true
            else
                if (not y) then false
                else true
                fi
            fi
        };
    };

    (*
    class Main{
        main() : Object{
            let stack : Stack <- (new Stack).init()
            in
            {
                stack.push("m");
                stack.push("t");
                stack.push("i");
                stack.push("n");


                stack.pop();

                stack.swap();

                -- stack.display();

                stack.push("s");
                stack.push("10");
                stack.push("3");

                stack.plus();

                stack.display();

            }
        };
    };
    *)

    class Main{
        i : IO <- new IO;
        main() : Object{
            let flag : Bool <- true,
```

```
                c : String ,
                stack : Stack <- (new Stack).init()
            in
            while (flag) loop
            {
                i.out_string(">");
                c <- i.in_string();
                if ( c="e") then stack.e()
                else
                    if ( c="d" ) then stack.display()
                        else
                            if ( c="x") then flag <- false
                                else stack.push(c)
                            fi
                    fi
                fi;
            }
            pool
        };
};
```