实验报告

课程：编译原理

姓名：李锦源
学号：20238131022 数据结构：`Stack` 日期：2025年9月27日 星期六 13:31:21

GitHub 仓库链接：
https://github.com/awa-Orz-Ljy/1coolc.git

摘要

本实验使用 COOL 语言实现了一个通用的栈（Stack）数据结构。
栈是一种 **后进先出（LIFO）** 的线性结构，常用于函数调用、括号匹配等场景。
本报告详细介绍了代码的设计与实现过程，包括 `StackNode` 和 `Stack` 两个核心类，以及 `Main` 测试类，并通过运行结果验证了实现的正确性。

设计与实现

类设计

1. **StackNode 类**
   - 属性：
     - `item : Object` —— 节点存储的数据
     - `next : StackNode` —— 指向下一个节点
   - 方法：
     - `init(i:Object, n:StackNode)` —— 初始化节点
     - `getItem()` —— 获取节点存储的数据
     - `getNext()` —— 获取下一个节点

   → `StackNode` 相当于链表的一个节点。

2. **Stack 类**
   - 属性：
     - `top : StackNode` —— 指向栈顶节点
   - 方法：
     - `isEmpty()` —— 判断栈是否为空
     - `push(item:Object)` —— 入栈
     - `peek()` —— 查看栈顶元素（不移除）
     - `pop()` —— 出栈并返回栈顶元素
     - `print()` —— 打印栈中所有元素

   → 内部由 `StackNode` 串联，形成链式存储。

3. **Main 类**
    - 主要用于测试栈的功能。
    - 在 `main()` 方法中：
        - 测试栈是否为空
        - 推入不同类型的元素（`String` 与 `Int`）
        - 使用 `peek()` 测试栈顶元素
        - 打印栈中所有元素

核心算法解释

StackNode 类方法介绍

- **初始化节点 init(i:Object, n:StackNode)**

```
init(i : Object, n : StackNode) : StackNode {
    {
        item <- i;
        next <- n;
        self;
    }
};
- i → 节点存储的数据
- n → 指向下一个节点
- 返回 self，便于链式操作
- 每次创建新节点时都调用这个方法初始化节点内容
- 时间复杂度 O(1)
```

- **获取节点数据 getItem()**

```
getItem() : Object {
    item
};
- 返回节点存储的数据
- 栈操作（如 peek() 或 pop()）需要通过它访问节点的值
- 时间复杂度 O(1)
```

- **获取下一个节点 getNext()**

```
getNext() : StackNode {
    next
};
- 返回指向下一个节点的引用
- 栈操作（如 pop() 更新 top 或 print() 遍历）需要用它来访问链表
- 时间复杂度 O(1)
```

Stack类的方法解释

- **入栈 push**

```
push(item : Object) : SELF_TYPE {
    {
        let new_node : StackNode <- (new StackNode).init(item, top) in
            top <- new_node;
        self;
    }
};
- 创建新节点，指向原栈顶，再更新 top 时间复杂度为O(1)。
```

- **出栈 pop**

```
pop() : Object {
    if isEmpty() then
        { out_string("Error: pop from empty stack.\n"); abort(); new Object; }
    else
        let item_to_return : Object <- top.getItem() in
        { top <- top.getNext(); item_to_return; }
    fi
};
- 取出 top.item，再更新 top = top.next 时间复杂度为O(1)。
```

- **判断栈是否为空 isEmpty**

```
isEmpty() : Bool {
    top = null
};
- 判断栈顶是否为空 时间复杂度为O(1)。
```

- **查看栈顶元素 peek**

```
peek() : Object {
    if isEmpty() then
        { out_string("Error: peek from empty stack.\n"); new Object; }
    else
        top.getItem()
    fi
};
- 返回栈顶元素但不删除 时间复杂度为O(1)。
```

- **打印栈中所有元素 print**

```
print() : SELF_TYPE {
    let current : StackNode <- top in
    {
        out_string("----- Top of Stack -----\n");
        while current != null loop
            case current.getItem() of
                s : String => out_string(s); out_string("\n");
                i : Int => out_int(i); out_string("\n");
                o : Object => out_string("Object\n");
            esac;
            current <- current.getNext();
        pool;
        out_string("----- Bottom of Stack -----\n");
        self;
    }
};
```

- 遍历整个栈并打印元素 时间复杂度 O(n)。

测试与结果

Main 类测试代码片段

```
out_string("Pushing 'Alice', 100, 'Bob'...\n");
miku_stack.push("Alice");
miku_stack.push(100);
miku_stack.push("Bob");
miku_stack.print();

out_string("Peeking top element: ");
case miku_stack.peek() of
    s : String => out_string(s);
    i : Int => out_int(i);
    o : Object => out_string("Object");
esac;
```

运行输出示例

```
--- Stack Demo ---

Is stack empty? Yes
Stack is empty.

Pushing 'Alice', 100, 'Bob'...
----- Top of Stack -----
Bob
100
Alice
```

```
----- Bottom of Stack -----

Peeking top element: Bob
```

从结果可以看出：

- 初始时栈为空；
- 入栈后，元素顺序符合 后进先出；
- peek() 正确返回栈顶元素 Bob；
- print() 打印出栈中全部元素。

结论

本实验实现了一个基于链表的通用栈结构。

熟悉了 COOL 的面向对象语法；

理解了 链式存储栈 的实现原理；

掌握了在 COOL 中使用 多态（Object + case 语句）来处理不同数据类型。