编译原理 H15-1

第一题

C++ 关于 `delete` 算子的解释见 CPP Reference。现在为了提高程序设计的用户隐私性，准备在 SysYF 文法中加入**清除语句**，其可以清除不再需要的数据。具体做法为修改 $expr$ 的文法：

- $expr \to \ldots \mid \textbf{clear}\ expr$

其中 ... 为原来的 $expr$ 定义。$\textbf{clear}\ expr$ 语句的工作解释如下：

- 先对 $expr$ 求类型
  - 如果 $expr$ 的类型是 $void$，则什么也不做
  - 否则，将其置为类型的默认值（如 $integer$ 类型将置为 0）
- 语句返回值是执行上述操作后 $expr$ 的值
- 语句类型与 $expr$ 相同

请回答下列问题：

1. 写出 **clear** 语句的定型规则；
2. 记类型 T 的默认值为 D_T，域类型 $v = X(a_1 : T_1 = l_1, \ldots, a_n : T_n = l_n)$，用 $[\alpha_i/\alpha]S$ 可以将 S 中自由出现的 α 替换为 α_i，用 $e \to T, S$ 表示整个语句（T 为类型、S 为内容），请写出对 $void$ 类型和域类型 v 做 **clear** 操作的推理规则；
3. 为对域类型语句操作的 **clear** 语句写一个语法制导的翻译方案，并且说明你定义的函数；
4. 如果引入 $const$ 类型（不可变），修改上述 **clear** 语句的翻译翻译方案以支持这种类型的处理；
5. 回答下面两个有关 **clear** 语句具体实现的问题。
   - **停止-复制垃圾回收**的做法是，先暂停程序的运行，然后将所有存活的对象从当前堆复制到另一个堆，此时没有复制的全部都是垃圾。如果新的 SysYF 语言编译器使用停止-复制垃圾回收方法，请评价 **clear** 语句清除数据的效率；
   - **标记-清扫垃圾回收**的做法是，从堆栈和静态存储区出发遍历所有的引用，进而找出所有存活的对象，如果活着就标记。全部标记完毕后开始清理，没有标记的对象将会被释放，不会发生任何赋值动作。如果新的 SysYF 语言编译器使用标记-清扫垃圾回收方法，请评价 **clear** 语句清除数据的效率。

> **参考解答**
>
> 1. $$\frac{\Gamma \vdash e : T}{\Gamma \vdash \textbf{clear}\ e : T}$$
> 2. 对于 $void$ 类型，答案是 $\dfrac{\Gamma \vdash e \to void, S}{\Gamma \vdash \textbf{clear}\ e \to void, S}$；对于域类型，答案是
>
> $$\frac{\begin{array}{l}\Gamma \vdash e \to v, S \\ v = X(a_1 : T_1 = l_1, \ldots, a_n : T_n = l_n) \\ S' = [D_{T_1}/l_1]\ldots[D_{T_n}/l_n]S\end{array}}{\Gamma \vdash \textbf{clear}\ e \to v, S'}$$
> 3. 考虑文法 $expr \to \textbf{clear}\ expr_1$，翻译方案如下：

$$
\begin{aligned}
&\textbf{assert } expr_1.type == T_1 \times \ldots \times T_n \\
&expr \rightarrow \textbf{clear } expr_1 \\
&\qquad \{expr.type = expr_1.type; \\
&\qquad\quad \textbf{for } i \textbf{ in } [1...n]: \\
&\qquad\qquad expr.T_i = expr_1.T_i; \\
&\qquad\qquad expr.a_i = expr_1.a_i; \\
&\qquad\qquad expr.l_i = expr_1.l_i; \\
&\qquad \}
\end{aligned}
$$

4. 现在类型中可能有 $const$，修改上述翻译方案如下：

$$
\begin{aligned}
&\textbf{assert } expr_1.type == T_1 \times \ldots \times T_n \\
&expr \rightarrow \textbf{clear } expr_1 \\
&\qquad \{expr.type = expr_1.type; \\
&\qquad\quad \textbf{for } i \textbf{ in } [1...n]: \\
&\qquad\qquad expr.T_i = expr_1.T_i; \\
&\qquad\qquad \textbf{if } (expr.T_i \neq const) \{ \\
&\qquad\qquad\quad expr.a_i = expr_1.a_i; \\
&\qquad\qquad\quad expr.l_i = expr_1.l_i; \\
&\qquad\qquad \} \\
&\qquad \}
\end{aligned}
$$

5. （1）停等-复制垃圾回收对于 **clear** 语句有致命缺陷：如果需要 **clear** 的对象在之前被这种方式垃圾回收过，原来的数据将仍然留在旧的存储空间中，**clear** 语句只会处理新对象的数据；

   （2）标记-清扫垃圾回收没有上述缺陷，不过此过程后，剩下的存储空间是不连续的，垃圾回收器要是希望得到连续空间的话，就得重新整理剩下的对象，这涉及对象的移动（即还是要考虑上述缺陷，在移动时清理原有数据）。

第二题

当程序调用一个函数的时候，开销是很大的，比如保存原来的栈指针、保存某些寄存器的值、保存返回地址、设置参数，等等。其中很多都是内存读写操作，速度比较慢。如果能做一些优化，减少这些开销，那么带来的优化效果理论上来说就很显著，尾递归优化就是一种思路。

对函数调用在**尾位置的递归或互相递归的函数**，由于函数自身调用次数很多，递归层级很深，尾递归优化则使原本 $O(n)$ 的调用栈空间只需要 $O(1)$。

考虑如下数学函数 $g(n)$ 的计算：

$$
\begin{aligned}
f(n) &= n \bmod 3 + 1 \\
g(n) &= \begin{cases} 1, & n = 1 \\ f(n)g(n-1), & \text{other} \end{cases}
\end{aligned}
$$

1. `test/foo_linear.c` 与 `test/foo_tail.c` 提供了这个函数计算的线性递归版本和尾递归版本。如果尾部函数调用复用调用者的存储空间，说明两种情况下（调用 `foo(10)`）理论上各需要**开辟**多少个函数调用栈帧；
2. 使用 `clang [foo_*.c] -S` 产生汇编代码，观察你所用的 clang 版本是否对 `test/foo_linear.c` 的汇编实现做了尾递归优化。你可以使用提供于 `test/` 下的 `foo_linear.s` 作为参考，它是使用 `clang version 3.7.0` 生成的，目标机器是 `x86_64-w64-windows-gnu`；
3. 有人提出，假设编译器不优化尾部调用，如何在不让栈向上增长的前提下实现尾部递归。有一些支持函数嵌套的语言通过一种叫做**弹跳床**（trampoline）的位置，也就是一块不断进行函数调用的代码来实现。所有函数代码的加载过程都透过这个弹跳床。当一个函数需要调用另一个函数时，它不是直接调用该函数，而是将该函数的位置、该调用使用的参数等信息传递给弹跳床，让其代为执行。请评价这种做法，并在 `test/` 下给出范例。

参考答案

1. 对 `foo(10)` 的调用，线性递归版本的计算过程如下：

```
foo(10)
2 * foo(9)
2 * (1 * foo(8))
2 * (1 * (3 * foo(7)))
2 * (1 * (3 * (2 * foo(6))))
2 * (1 * (3 * (2 * (1 * foo(5)))))
2 * (1 * (3 * (2 * (1 * (3 * foo(4))))))
2 * (1 * (3 * (2 * (1 * (3 * (2 * foo(3)))))))
2 * (1 * (3 * (2 * (1 * (3 * (2 * (1 * foo(2))))))))
2 * (1 * (3 * (2 * (1 * (3 * (2 * (1 * (3 * foo(1)))))))))
...
```

   可见线性递归版本开辟了 10 个函数调用栈帧；

   尾递归版本的计算过程如下：

```
foo(10)
_foo(10, 1)
_foo(9, a9)
_foo(8, a8)
_foo(7, a7)
_foo(6, a6)
_foo(5, a5)
_foo(4, a4)
_foo(3, a3)
_foo(2, a2)
_foo(1, a1)
...
```

   可见尾递归版本只开辟了 1 个函数调用栈帧。

2. 使用 `clang version 3.7.0` 生成目标机器是 `x86_64-w64-windows-gnu` 的汇编文件 `foo_linear.s` 。观察第 37 行发现仍然使用了 `callq` 命令而不是 `jmp` 命令，这会导致开辟额外的函数调用栈帧，所以使用题目的命令时，这个版本的 clang 没有对尾递归做优化。

3. 如果所有函数调用都使用这种做法，相比常规的调用方式，这种调用额外增加了一些函数体内容。这也是一种额外的开销。一种用尾递归优化过的、使用弹跳床方法的代码实例见 `test/foo.py`，其可扩展性保证新加入的函数都可以调用适当的 `trampoline` 来实现。