M4: C Real-Eval-Print-Loop (crepl)

截止日期

Soft Deadline: 5月5日23:59:59

收到的作业

引子

如果你想计算一些表达式的数值，不会还在用Windows的计算器或者自带的搜索框吧？在终端输入 `python3`，再输入一些表达式(回车)，有惊喜。基于“解释执行”的程序设计语言(包括shell本身)天生就具有这种交互式的工作模式。

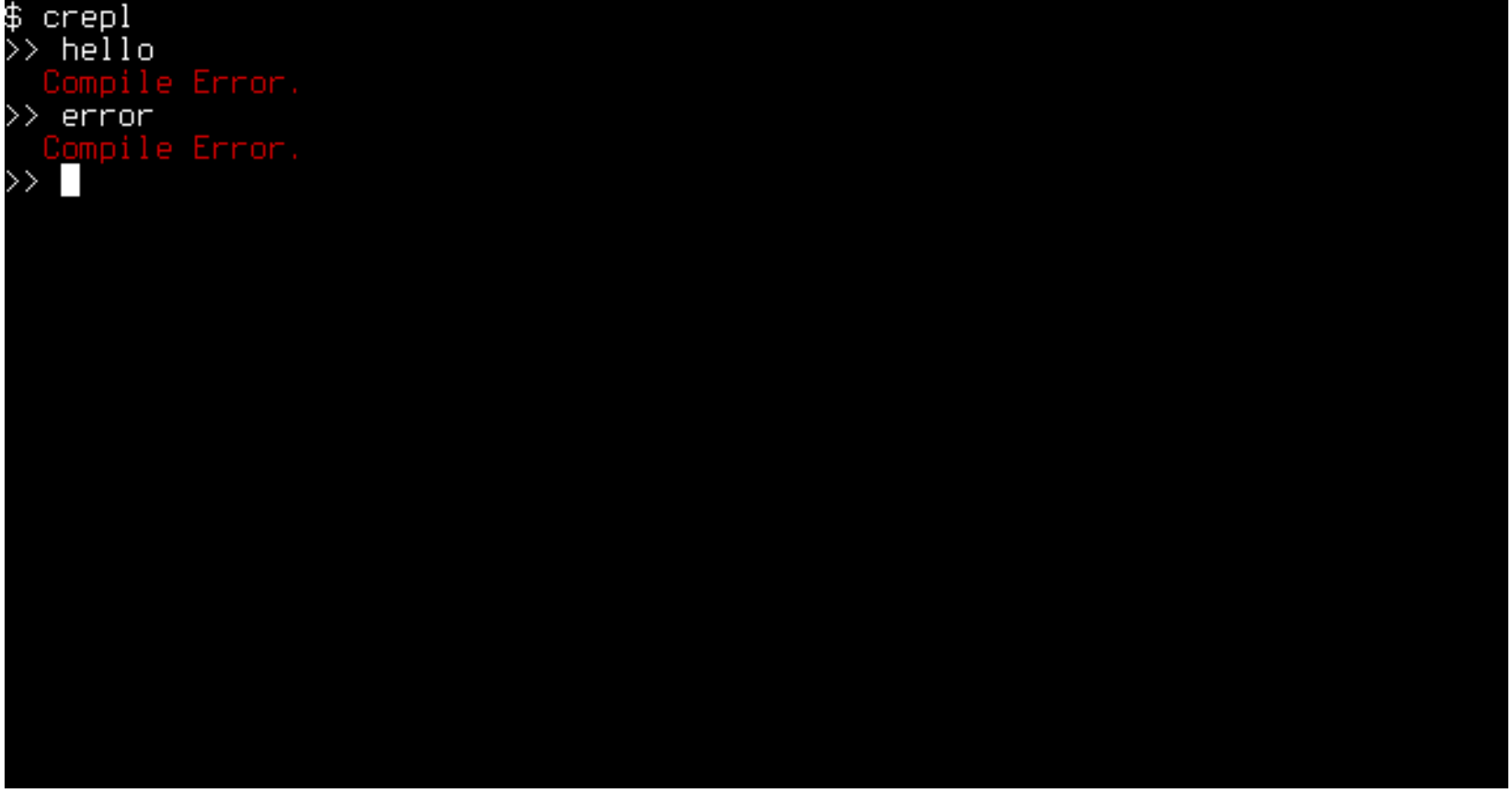
你有没有想过，C这种编译型的语言，同样也可以实现“交互式”的shell？——支持即时定义函数，而且能计算C表达式的数值。如果你输入一行代码，比如 `strlen("Hello World")`，这段代码会经历gcc编译、动态加载、调用执行，最终把代码执行得到的数值 `11` 打印到屏幕上！

完成这个实验，你对编译、链接、加载的理解将会上一个台阶。

背景

很多编程语言都提供了交互式的read-eval-print-loop (https://en.wikipedia.org/wiki/Read–eval–print_loop) (REPL)，更俗一点的名字就是“交互式的shell”。比如你在命令行输入 `python`，就可以和Python shell交互了。比如你想知道是多少，直接输入 `2**100` 回车就可以了。现代程序设计语言都提供类似的设施，即便是非解释型的程序设计语言也提供类似的设施，例如Scala REPL、go-eval等。

在这个实验中，我们实现非常简单的C交互式shell，就像下面图中所示那样：



这个技术和现代虚拟机中的即时编译(just-in-time)技术是非常相关的：在程序运行时(而非程序执行前)进行编译，并将编译得到的二进制代码(指令/数据)动态加载。其中最成功的案例之一是Sun的Java虚拟机hotspot (https://wiki.openjdk.java.net/display/HotSpot/Main) (现在是OpenJDK的一部分)，它使Java彻底摆脱了“性能低下”的诟病，也是引领Web热潮的核心技术。另一个最成功的案例是JavaScript的V8引擎 (https://v8.dev/)。借助Webkit/v8，Chrome成功地把IE拖下神坛，并且奠定了Google在互联网技术的霸主地位。

实验描述

获取实验代码与提交

本学期的所有代码(minilab, OSlab)都在同一个目录中完成。请参考代码获取与提交 (OS2019_Code)。

在你原先的os-workbench基础上，执行

```
git pull origin M4
```

将本实验的框架代码下载到本地。在 `crepl/` 下编译能得到二进制文件 `crepl-32` 和 `crepl-64` 。

`crepl` - 逐行从stdin中输入，根据内容进行处理：

- 如果输入的一行定义了一个函数，则把函数编译并加载到进程的地址空间中；
- 如果输入是一个表达式，则把它的值输出。

总览

crepl

描述

解释执行每一行标准输入中的C代码(假设所有输入的表达式都是整数；定义函数的返回值也永远是整数)，分如下两种情况：

- 函数总是以 `int` 开头，例如

```
int fib(int n) { if (n ⩽ 1) return 1; return fib(n - 1) + fib(n - 2); }
```

接收若干参数，返回一个 `int` 数值。如果一行是一个函数，它将会经过gcc编译，并被加载到当前进程的地址空间中。函数可以引用之前定义过的函数。

- 否则，一行是一个合法的C语言表达式，其类型为int，例如 `fib(3) * fib(4)`, `"0123456789abcdef"[fib(6) % 16]` 都是合法的表达式。

函数和表达式均可以调用之前定义过的函数，但不访问全局的状态。如果一行既不是函数，也不是表达式， `crepl` 的行为未定义；重复定义重名函数行为未定义。你可以做出非常友好的假设——我们没有刁难你的测试用例，都和demo中的差不多。

解释

你的输出不必和demo一致，只要在输入一行是表达式的前提下输出表达式的值即可，例如以下程序实现也是完全没问题的：

```
$ ./crepl-64 << EOF
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }
gcd(256, 144) * gcd(56, 84)
EOF
[crepl] = 448.
```

看起来很好玩吧，从前可能大家都觉得只有解释型的语言能这么做。但实际上，编译和解释并没有明确的边界——在openjdk的实现中，即便是“解释器”也是编译的(只是没有经过优化)；而传统的所谓解释型的语言，Python, JavaScript, ……都有编译到机器代码的过程。

Shell Tricks

上面的Shell语法似乎有些奇怪，但也不难读懂——两个EOF之间的文字。被作为输入传递给 `crepl` 。但万一输入的文字里有EOF，不就麻烦了么？

为什么demo那么酷炫?

JYY才不会告诉你demo调用了一些外挂程序在终端里实现了语法高亮。你完全没有做这件事情的必要——使用外挂很可能导致你的程序在助教的环境中不能运行，从而损失分数。

当然另一个更好的办法(也是编写软件常见的做法)是在运行时检查外挂是否存在。如果存在则调用；如果不存在则直接输出。

检查外挂是否存在

JYY使用了pygmentize工具：

```
echo 'int main() {}' | pygmentize -l c
```

所以你的程序里可能会写：

```
if (pygmentize_exists()) {
  // call pygmentize
  ...
}
```

但注意到操作系统里的进程是并发的。你可能在

- 执行 `pygmentize_exists()` 时，pygmentize是存在且好用的。
- 就在检查完之后，pygmentize被删除了。

这就可能导致TOCTTOU (https://en.wikipedia.org/wiki/Time_of_check_-to_time_of_use)的bug，甚至安全漏洞。刚才的连接里有一段让人不禁想笑的例子，请戳。因此，把代码写对真不是件容易的事情，如果有朝一日操作系统能提供事务支持，也能缓解这些问题。

实验指南

想自己尝试?

试试吧！这次不给任何提示了(也不要偷看下面哦)，加油！

你多多少少都应该对你的能力有自信——独立解决问题会让你感觉更棒！

函数的定义和加载

对于一个一行的函数，比如

```
int gcd(int a, int b) { return b ? gcd(b, a % b) : a; }
```

我们可以把它写到一个临时文件里，比如 a.c 中，然后在命令行里调用 gcc -c a.c，然后就能得到 a.o，这里就包含了函数的代码，我们希望**想办法把代码加载到本进程的地址空间中**。

想要直接使用系统的动态链接函数进行加载，就必须把这个 .c 文件编译成位置无关的共享库。你需要阅读手册、互联网上的文档找到正确的编译选项，把 .c 文件编译成正确的二进制文件，然后用 dlopen 函数动态载入。

RTFM

之前的实验已经给了足够多的引导，让你学会查阅资料了。在这个实验中，你会面对几个问题：

1. 如何创建一个文件？创建什么名字的文件？是否应该每次创建不同的临时文件？有相应的库函数能帮助你创建临时文件。当然我们不阻止你在当前目录创建 a.c ——但显然使用临时文件是更好的选择。
2. 我们用fork/exec/wait可以实现对gcc的调用，但用什么选项编译？你知道 -c，-g 这些选项的含义吗？什么选项能正确编译出能被载入的共享库？
3. dlopen 有 mode 选项，设置的不正确可能导致实验失败。应该用哪个(些)选项？

通过阅读手册和互联网上的文档独立解决这些问题，这是最好的锻炼——以后到了工作中，或者读研究生的时候，你的老板也许不会耐心地告诉你这些，而是直接把锅甩给你，你干不好就会失去老板对你的信任。

表达式的求值

对于一个表达式，我们也可以把它变成一个函数嘛！比如我们想求 gcd(256, 144)，我们可以定义一个匿名的函数(也就是给它一个没有重名的名字)，比如

```
int __expr_wrap_123() {
  return gcd(256, 144);
}
```

换句话说，我们的临时文件好像是一个“模板”：

```
int __expr_wrap_XXX() {
  return (EXPR);
}
```

我们只需要把这个函数编译，加载到当前进程的地址空间里，然后在当前进程里调用：

```
int (*func)() = func_lookup(cmd_id); // 查找XXX对应的函数
int value = func(); // 通过函数指针调用
printf(">> %s = %d.\n", cmd, value);
```

至于 func_lookup 应该怎么实现呢？还记得每个二进制文件里都有很重要的一部分是符号表么？所有的函数都是一个符号，所以应该能根据符号查找到我们所需的内容。因此怎么调用 __expr_wrap_123() 这个问题就留给大家啦。

另一个麻烦是，这个函数里调用了 gcd ，而 gcd 是之前动态加载的(程序在编译的时候甚至都不知道有 gcd 这个函数存在)。如何让 gcd 能被程序调用？

思考题：二进制文件中的符号

用 nm 和 readelf 都能查看二进制文件中的符号。你有兴趣了解这些符号都是用来做什么的吗？

自己实现二进制文件加载

注意

以下部分不计任何成绩也不需要实现，给闲得蛋疼的学霸玩用的。

实现成功，肯定会好奇 dlopen 等一系列函数到底做了什么。如果你有兴趣，可以自己hack一下——在这个实验里，我们的假设很简单，函数只访问局部变量，因此你可以通过解析ELF文件，把一个动态链接库中的代码部分提取出来(在PA中已经做过类似的事情了)。

因为这些代码是位置无关的，所以我们可以把它加载到内存里的任何位置，你只需要用 malloc 为它分配一些内存就行了(或者你可以使用 mmap ，这是实现动态链接的标准方式)。如果你需要将一段内存标记为可执行，可以使用 mprotect 。

你可以先用一些简单的程序尝试一下(例如死循环，或者一些简单计算的函数例如 gcd)，然后你会发现真的只要把文件的内容搬到正确的地方(无论是直接写入的还是使用 mmap)、设置了正确的访问权限，代码就被动态链接进了进程的地址空间。恭喜你——你已经迈出了实现加载器的第一步。

然后你会发现，如果函数中调用了其他函数，或者修改了全局变量，只加载代码就不够了，你还是得老老实实分析 ELF文件，按照ABI所规定的动作完成加载。除非你对链接和加载有异常的兴趣，否则你不会想实现的，哈哈哈！

代码、函数调用和数据

动态链接中最简单的部分就是本地的代码：它们被编译成了位置无关的。数据和向其他符号的调用就麻烦得多了，如果有一个全局变量，它是有地址的，但它也不能控制自己被加载到地址空间的哪个地方。有兴趣的同学可以深入阅读链接和加载相关的材料 (http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html)。

NOI规则的漏洞

在很长一段时间里，全国青少年信息学奥林匹克竞赛 (http://www.noi.cn) (NOI)相关的竞赛都不打开编译优化开关，而且有一系列规定禁止你使用 `mprotect` 把一段代码解释执行，从而把一段预先编译优化好的代码加载到地址空间里。(条例禁止使用除读写规定的输入/输出文件之外的其它系统调用，大家应该没有傻到敢去违反这个条例。)

看起来条例很好地保证了大家不能乱来。但他们没有想到的是，用Trampoline (https://gcc.gnu.org/onlinedocs/gccint/Trampolines.html)能打破这一限制，让进程在默认的编译选项下获得能够执行的数据。计算机系统就是这样：经过很多年的演进，各种东西混合在一起，以至于理解它的行为非常困难。适当地trick链接器和加载器就能达到意想不到的目的。

这有什么好办法能解决呢？似乎是个open problem...