TEMU 中主要功能实现思路
（控制台命令交互方式）

阅读源代码

代码中 temu 目录下的源文件组织如下：

```
temu
├── include                  # 存放全局使用的头文件
│   ├── common.h             # 公用的头文件
│   ├── debug.h              # 一些便于调试用的宏
│   ├── macro.h              # 一些方便的宏定义
│   ├── misc.h               # 杂项
│   ├── temu.h
│   ├── cpu                  # 一些和处理器相关的头文件，如寄存器的定义、操作数的定义、指令的定义等
│   ├── memory               # 一些和存储器相关的头文件，例如读写接口、dram的定义等
│   └── monitor              # 一些和调试命令相关的头文件，如监视点、表达式求值等
│
└── src
    ├── main.c
    ├── cpu
    │   ├── exec.c           # MiniMIPS32相关指令
    │   ├── i-type.c         # I型指令相关
    │   ├── reg.c            # 寄存器相关
    │   ├── r-type.c         # R型指令相关
    │   └── special.c        # 特殊指令相关
    │
    ├── memory
    ├── monitor
        ├── cpu-exec.c       # 指令执行的主循环
        ├── expr.c           # 表达式求值的实现
        ├── monitor.c
        ├── ui.c             # 交互命令相关实现
        └── watchpoint.c     #监视点的实现
```

为了给出一份可以运行的框架代码，代码中完整实现了 lui、ori 和 and 指令的功能，并附带一个用户程序(mips_sc/src/logic.S)。在你第一次阅读代码的时候，你需要尽快掌握 TEMU 的框架，而不要纠缠于细节。随着实验任务的进行，你会反复回过头来探究这些细节。大致了解上述的目录树之后，你就可以开始阅读代码了，至于从哪里开始，就不用多费口舌了吧。

TEMU 开始执行的时候，会进行一些初始化工作，包括打开日志文件，编译正则表达式，初始化监视点结构池。这些初始化工作你几乎一个也看不懂，但不要紧，因为你现在根本不必关心它们的细节，因此可以继续阅读代码。之后代码会调用 restart()函数(在 temu/src/monitor/monitor.c 中定义)，它模拟了"计算机启动"的功能，主要是进行一些和"计算机启动"相关的初始化工作，包括

- 读入测试程序代码段 inst.bin 和数据段 data.bin
- 设置$pc 的初值

- 初始化 DRAM 的模拟

在一个完整的计算机中，程序的可执行文件应该存放在磁盘里，但目前我们并没有实现磁盘的模拟。因此，入口代码 entry 的引入其实是一种简化。我们知道内存是一种 RAM，是一种易失性的存储介质，这意味着计算机刚启动的时候，内存中的数据都是无意义的；而 BIOS 是固化在 ROM 中的，它是一种非易失性的存储介质，BIOS 中的内容不会因为断电而丢失。因此在真实的计算机系统中，计算机启动后首先会把控制权交给 BIOS，BIOS 经过一系列初始化工作之后，再从磁盘中将有意义的程序读入内存中执行。对这个过程的模拟需要了解很多超出本课程范围的细节，我们在这里做了简化，让 monitor 直接把一个程序的代码段和数据段读入到一个固定的内存位置，其中代码段加载到 0x1FC00000，数据段加载到 0x00000000，并把 0xBFC00000 作为$pc 的初值。

TEMU 中从虚拟地址 0xBFC00000（即物理地址 0x1FC00000）开始运行测试程序。测试程序的代码段和数据段通过 objcopy 命令获得（可参考工程目录下的 Makefile 文件）。

restart()函数执行完毕后，TEMU 会进入用户界面主循环 ui_mainloop() (在 temu/src/monitor/ui.c 中定义)，代码已经实现了几个简单的命令，它们的功能和 GDB 是很类似的。键入 c 之后，TEMU 开始进入指令执行的主循环 cpu_exec() (在 temu/src/monitor/cpu-exec.c 中定义)。

cpu_exec()模拟了 CPU 的工作方式：不断执行指令。exec()函数(在 temu/src/cpu/exec.c 中定义)的功能是让 CPU 执行一条指令。已经执行的指令会输出到日志文件 log.txt 中，你可以打开 log.txt 来查看它们。

特别注意，cpu_exec()在模拟 CPU 执行的过程中，首先需要将指令的虚拟地址转换为物理地址，即将 PC 的高三位清“0”，程序中通过代码“pc = cpu.pc & 0x1fffffff;”实现。

执行指令的相关代码在 temu/src/cpu/目录下，其中一个重要的部分是定义在 temu/src/cpu/exec.c 文件中的 opcode_table 数组，在这个数组中，你可以看到框架代码中都已经实现了哪些指令，其中 inv 的含义是 invalid，代表对应的指令还没有实现(也可能是 MiniMIPS32 中不存在该指令)。随着你实现越来越多的指令，这个数组会逐渐被它们代替。执行具体指令的代码请参考 temu/src/cpu/i-type.c，temu/src/cpu/r-type.c 和 temu/src/cpu/special.c，你可以根据实际情况在这些文件中添加新的指令，也可以定义新的文件。

温故而知新

opcode_table 到底是一个什么类型的数组？

如果你感到困惑，那么说明你需要马上复习程序设计的知识了。这里有一份 C 语言入门教程，如果你觉得你的程序设计知识比较生疏，请你务必阅读它。

TEMU 不断执行指令，直到遇到以下情况之一，才会退出指令执行的循环：

- 达到要求的循环次数。
- 用户程序执行了 temu_trap 指令。这是一条特殊的指令，opcode 为 0x12。MIPS 中并没有这条指令，它是为了指示程序的结束而加入的。我们可以借助这条指令实现一些无法通过程序本身完成的，需要 TEMU 帮助的功能。

退出 cpu_exec()之后，TEMU 将返回到 ui_mainloop()，等待用户输入命令。但为了再次运行程序，你需要退出 TEMU，然后重新运行。

究竟要执行多久?

在 cmd_c()函数中，调用 cpu_exec()的时候传入了参数-1，你知道为什么吗?

最后我们聊聊代码中一些值得注意的地方。

- 三个对调试有用的宏(在 temu/include/debug.h 中定义)
  - Log()是 printf()的升级版，专门用来输出调试信息，同时还会输出使用 Log()所在的源文件，行号和函数，当输出的调试信息过多的时候，可以很方便地定位到代码中的相关位置。
  - Assert()是 assert()的升级版，当测试条件为假时，在 **assertion fail** 之前可以输出一些信息。
  - panic()用于输出信息并结束程序，相当于无条件的 **assertion fail**。

代码中已经给出了这三个宏的例子，如果你不知道如何使用它们，阅读源代码。

- 访问模拟的内存
  - 在程序运行的过程中，总是使用 mem_read()和 mem_write()访问模拟的内存。位于 temu/src/memory/memory.c 中。

需要注意的是，上面描述的只是一个十分大概的过程，如果你对这个过程有疑问，**阅读源代码**。

TEMU 支持的主要功能

我们需要在 TEMU 中实现一个具有如下功能的简易调试器(相关部分的代码在 temu/src/monitor/目录下)，具体命令的格式请参考如下表格：

命令	格式	使用举例	说明
帮助*	help	help	打印命令的帮助信息
继续运行*	c	c	继续运行暂停的程序
退出*	q	q	退出 TEMU
单步执行	si [N]	si 10	程序单步执行 N 条指令后暂停，当 N 没有给出时，缺省为 1。
打印程序状态	info SUBCMD	info r info w	打印寄存器状态 打印监视点信息
表达式求值	p EXPR	p $s0 + 4	求出表达式 EXPR 的值,请见数学表达式求值小节。
扫描内存	x N EXPR	x 10 $t0	求出表达式 EXPR 值，将结果作为起始内存地址，以 16 进制形式输出连续的 N 个 4 字节。
设置监视点	w EXPR	w $at == 0x1010	当表达式 EXPR 的值发生变化时，暂停程序执行。
删除监视点	d N	d 2	删除序号为 N 的监视点

备注:

* 命令已实现

解析命令

TEMU 通过 readline 库与用户交互，使用 readline()函数从键盘上读入命令. 与 gets()相比，readline()提供了"行编辑"的功能，最常用的功能就是通过上，下方向键翻阅历史记录。事实上，shell 程序就是通过 readline()读入命令的. 关于 readline()的功能和返回值等信息，请查阅

```
man readline
```

从键盘上读入命令后，TEMU 需要解析该命令，然后执行相关的操作. 解析命令的目的是识别命令中的参数，例如在 si 10 的命令中识别出 si 和 10，从而得知这是一条单步执行 10 条指令的命令. 解析命令的工作是通过一系列的字符串处理函数来完成的，例如框架代码中的 strtok()。strtok()是 C 语言中的标准库函数，如果你从来没有使用过 strtok()，并且打算继续使用框架代码中的 strtok()来进行命令的解析，请务必查阅

```
man strtok
```

另外，cmd_help()函数中也给出了使用 strtok()的例子. 事实上，字符串处理函数有很多，键入以下内容：

```
man 3 str<TAB><TAB>
```

其中<TAB>代表键盘上的 TAB 键. 你会看到很多以 str 开头的函数，其中有你应该很熟悉的 strlen()，strcpy()等函数. 你最好都先看看这些字符串处理函数的 manual page，了解一下它们的功能，因为你很可能会用到其中的某些函数来帮助你解析命令。当然你也可以编写你自己的字符串处理函数来解析命令。

另外一个值得推荐的字符串处理函数是 sscanf()，它的功能和 scanf()很类似，不同的是 sscanf()可以从字符串中读入格式化的内容，使用它有时候可以很方便地实现字符串的解析。如果你从来没有使用过它们，请阅读源代码，或者到互联网上查阅相关资料。

单步执行

单步执行的功能十分简单，而且框架代码中已经给出了模拟 CPU 执行方式的函数，你只要使用相应的参数去调用它就可以了。如果你仍然不知道要怎么做，请阅读源代码。

打印寄存器

打印寄存器就更简单了，执行 info r 之后，直接用 printf()输出所有寄存器的值即可。如果你不知道要输出什么，你可以参考 GDB 中的输出。

扫描内存

扫描内存的实现也不难，对命令进行解析之后，先求出表达式的值。但你还没有实现表达式求值的功能，现在可以先实现一个简单的版本：规定表达式 EXPR 中只能是一个十六进制数，例如

```
x 10 0xBFC00000
```

这样的简化可以让你暂时不必纠缠于表达式求值的细节。解析出待扫描内存的起始地址之后,你就使用循环将指定长度的内存数据通过十六进制打印出来。如果你不知道要怎么输出，同样的，你可以参考 GDB 中的输出。

实现了扫描内存功能之后，你可以打印 0xBFC00000（对应物理地址 0x1FC00000）附近内存，你应该会看到程序的代码，和用户程序的 objdump 结果进行对比（此时用户程序是 logic ，其 dump 结果在 mips_sc/build/logic.asm 中），看看你的实现是否正确。

数学表达式求值

给你一个表达式的字符串

```
"5 + 4 * 3 / 2 - 1"
```

你如何求出它的值？表达式求值是一个很经典的问题，以至于有很多方法来解决它. 我们在所需知识和难度两方面做了权衡，在这里使用如下方法来解决表达式求值的问题：

- 首先识别出表达式中的单元
- 根据表达式的归纳定义进行递归求值

词法分析

"词法分析"这个词看上去很高端，说白了就是做上面的第 1 件事情, "识别出表达式中的单元"。这里的"单元"是指有独立含义的子串，它们正式的称呼叫**token**。具体地说，我们需要在上述表达式中识别出 5, +, 4, *, 3, /, 2, -, 1 这些 token。你可能会觉得这是一件很简单的事情，但考虑以下的表达式：

```
"0xc0100000+   ($eax +5)*4 - *(  $ebp + 8) + number"
```

它包含更多的功能，例如十六进制整数(0xc0100000)，小括号，访问寄存器($eax)，指针解引用(第二个 *)，访问变量(number)。事实上，这种复杂的表达式在调试过程中经常用到，而且你需要在空格数目不固定(0 个或多个)的情况下仍然能正确识别出其中的 token。当然你仍然可以手动进行处理(如果你喜欢挑战性的工作的话)，一种更方便快捷的做法是使用**正则表达式**。正则表达式可以很方便地匹配出一些复杂的 pattern，是程序员必须掌握的内容，如果你从来没有接触过正则表达式，请查阅相关资料。在实验中，你只需要了解正则表达式的一些基本知识就可以了(例如**元字符**)。

学会使用简单的正则表达式之后，你就可以开始考虑如何利用正则表达式来识别出 token 了。我们先来处理一种简单的情况 -- 算术表达式，即待求值表达式中只允许出现以下的 token 类型：

- 十进制整数
- +, -, *, /
- (,)
- 空格串(一个或多个空格)

首先我们需要使用正则表达式分别编写用于识别这些 token 类型的规则。在框架代码中，一条规则是由正则表达式和 token 类型组成的二元组。框架代码中已经给出了+和空格串的规则，其中空格串的 token 类型是 NOTYPE，因为空格串并不参加求值过程，识别出来之后就可以将它们丢弃了；+的 token 类型是 '+'，事实上 token 类型只是一个整数，只要保证不同的类型的 token 被编码成不同的整数就可以了；框架代码中还有一条用于识别双等号的规则，不过我们现在可以暂时忽略它。

这些规则会在 TEMU 初始化的时候被编译成一些用于进行 pattern 匹配的内部信息，这些内部信息是被库函数使用的，而且它们会被反复使用，但你不必关心它们如何组织。但如果正则表达式的编译不通过，TEMU 将会触发 assertion fail，此时你需要检查编写的规则是否符合正则表达式的语法。

给出一个待求值表达式，我们首先要识别出其中的 token，进行这项工作的是 make_token()函数（temu/src/monitor/expr.c）。make_token()函数的工作方式十分直接，它用 position 变量来指示当前处理到的位置，并且按顺序尝试用不同的规则来匹配当前位置的字符串。当一条规则匹配成功，并且匹配出的子串正好是 position 所在位置的时候，我们就成功地识别出一个 token，Log()宏会输出识别成功的信息。你需要做的是将识别出的 token 信息记录下来(一个

例外是空格串)，我们使用 Token 结构体来记录 token 的信息：

```
typedef struct token {
  int type;
  char str[32];
} Token;
```

其中 type 成员用于记录 token 的类型。大部分 token 只要记录类型就可以了，例如+, -, *, /，但这对于有些 token 类型是不够的：如果我们只记录了一个十进制整数 token 的类型，在进行求值的时候我们还是不知道这个十进制整数是多少，这时我们应该将 token 相应的子串也记录下来，str 成员就是用来做这件事情的。需要注意的是, str 成员的长度是有限的，当你发现缓冲区将要溢出的时候，要进行相应的处理(思考一下，你会如何进行处理？)，否则将会造成难以理解的 bug。tokens 数组用于按顺序存放已经被识别出的 token 信息，nr_token 指示已经被识别出的 token 数目。

如果尝试了所有的规则都无法在当前位置识别出 token，识别将会失败。make_token()函数将返回 false，表示词法分析失败。

递归求值

把待求值表达式中的 token 都成功识别出来之后，接下来我们就可以进行求值了。需要注意的是，我们现在是在对 tokens 数组进行处理，为了方便叙述，我们称它为"token 表达式"。例如待求值表达式

```
"4 +3*(2- 1)"
```

的 token 表达式为

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| NUM | '+' | NUM | '*' | '(' | NUM | '-' | NUM | ')' |
| "4" |     | "3" |     |     | "2" |     | "1" |     |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

根据表达式的归纳定义特性，我们可以很方便地使用递归来进行求值。首先我们给出算术表达式的归纳定义：

```
<expr> ::= <number>          # 一个数是表达式
    | "(" <expr> ")"     # 在表达式两边加个括号也是表达式
    | <expr> "+" <expr>     # 两个表达式相加也是表达式
    | <expr> "-" <expr>     # 接下来你全懂了
    | <expr> "*" <expr>
    | <expr> "/" <expr>
```

上面这种表示方法就是大名鼎鼎的巴克斯范式（BNF）。根据上述 BNF 定义，一种解决方案已经逐渐成型了：既然长表达式是由短表达式构成的，我们就先对短表达式求值，然后再对长表达式求值。这种十分自然的解决方案就是分治法的应用，就算你没听过这个高大上的名词，也不难理解这种思路。而要实现这种解决方案，递归是你的不二选择。

为了在 token 表达式中指示一个子表达式，我们可以使用两个整数 p 和 q 来指示这个子表达式的开始位置和结束位置。这样我们就可以很容易把求值函数的框架写出来了：

```
eval(p, q) {
  if(p > q) {
    /* Bad expression */
  }
  else if(p == q) {
    /* Single token.
     * For now this token should be a number.
     * Return the value of the number.
     */
  }
  else if(check_parentheses(p, q) == true) {
    /* The expression is surrounded by a matched pair of parentheses.
     * If that is the case, just throw away the parentheses.
     */
    return eval(p + 1, q - 1);
  }
  else {
    /* We should do more things here. */
  }
}
```

其中 check_parentheses()函数用于判断表达式是否被一对匹配的括号包围着，同时检查表达式的左右括号是否匹配，如果不匹配，这个表达式肯定是不符合语法的，也就不需要继续进行求值了。我们举一些例子来说明 check_parentheses()函数的功能：

"(2 - 1)" // true

"(4 + 3 * (2 - 1))" // true

"4 + 3 * (2 - 1)" // false, the whole expression is not surrounded by a matched pair of parentheses

"(4 + 3)) * ((2 - 1)" // false, bad expression

"(4 + 3) * (2 - 1)" // false, the leftmost '(' and the rightmost ')' are not matched

至于怎么检查左右括号是否匹配，就留给聪明的你来思考吧！

上面的框架已经考虑了 BNF 中算术表达式的开头两种定义，接下来我们来考虑剩下的情况(即上述伪代码中最后一个 else 中的内容)。一个问题是，给出一个最左边和最右边不同时是括号的长表达式，我们要怎么正确地将它分裂成两个子表达式？我们定义 dominant operator 为表达式人工求值时，最后一步进行运行的运算符，它指示了表达式的类型(例如当最后一步是减法运算时，表达式本质上是一个减法表达式)。要正确地对一个长表达式进行分裂，就是要找到它的 dominant operator。我们继续使用上面的例子来探讨这个问题：

```
"4 + 3 * ( 2 - 1 )"
/*********************/
case 1:
    "+"
   /   \
"4"     "3 * ( 2 - 1 )"


case 2:
        "*"
       /   \
"4 + 3"     "( 2 - 1 )"


case 3:
             "-"
            /   \
"4 + 3 * ( 2"     "1 )"
```

上面列出了3种可能的分裂，注意到我们不可能在非运算符的token处进行分裂，否则分裂得到的结果均不是合法的表达式。根据 dominant operator 的定义，我们很容易发现，只有第一种分裂才是正确的，这其实也符合我们人工求值的过程：先算 4 和 3 * (2 - 1)，最后把它们的结果相加。第二种分裂违反

了算术运算的优先级，它会导致加法比乘法更早进行。第三种分裂破坏了括号的平衡，分裂得到的结果均不是合法的表达式。

通过上面这个简单的例子，我们就可以总结出如何在一个 token 表达式中寻找 dominant operator 了：

- 非运算符的 token 不是 dominant operator。
- 出现在一对括号中的 token 不是 dominant operator。注意到这里不会出现有括号包围整个表达式的情况，因为这种情况已经在 check_parentheses()相应的 if 块中被处理了。
- dominant operator 的优先级在表达式中是最低的。这是因为 dominant operator 是最后一步才进行的运算符。
- 当有多个运算符的优先级都是最低时，根据结合性，最后被结合的运算符才是 dominant operator。一个例子是 1 + 2 + 3，它的 dominant operator 应该是右边的+。

要找出 dominant operator，只需要将 token 表达式全部扫描一遍，就可以按照上述方法唯一确定 dominant operator。

```
eval(p, q) {
  if(p > q) {
    /* Bad expression */
  }
  else if(p == q) {
    /* Single token.
     * For now this token should be a number.
     * Return the value of the number.
     */
  }
  else if(check_parentheses(p, q) == true) {
    /* The expression is surrounded by a matched pair of parentheses.
     * If that is the case, just throw away the parentheses.
     */
    return eval(p + 1, q - 1);
  }
  else {
    op = the position of dominant operator in the token expression;
    val1 = eval(p, op - 1);
    val2 = eval(op + 1, q);

    switch(op_type) {
      case '+': return val1 + val2;
      case '-': /* ... */
      case '*': /* ... */
      case '/': /* ... */
      default: assert(0);
    }
  }
}
```

找到了正确的 dominant operator 之后，事情就变得很简单了，先对分裂出来的两个子表达式进行递归求值，然后再根据 dominant operator 的类型对两个子表达式的值进行运算即可。于是完整的求值函数如上所示。

实现算术表达式的递归求值

我们已经把递归求值的思路和框架都列出来了，你需要做的是理解这一思路，然后在框架中填充相应的内容. 实现表达式求值的功能之后，p 命令也就不难实现了。

需要注意的是，上述框架中并没有进行错误处理，在求值过程中发现表达式不合法的时候，应该给上层函数返回一个表示出错的标识，告诉上层函数"求值的结果是无效的"。例如在 check_parentheses()函数中，(4 + 3)) * ((2 - 1) 和 (4 + 3) * (2 - 1)这两个表达式虽然都返回 false，因为前一种情况是表达式不合法，是没有办法成功进行求值的；而后一种情况是一个合法的表达式，是可以成功求值的，只不过它的形式不属于 BNF 中的 "("<expr> ")"，需要使用 dominant operator 的方式进行处理，因此你还需要想办法把它们区别开来。

当然，你也可以在发现非法表达式的时候使用 assert(0)终止程序，不过这样的话，你在使用表达式求值功能的时候就要十分谨慎了。

选做任务 1：实现带有负数的算术表达式的求值

在上述实现中，我们并没有考虑负数的问题，例如“1 + -1”。

此时会被判定为不合法的表达式。为了实现负数的功能，需要考虑两个问题：

- 负号和减号都是 - ，如何区分它们?
- 负号是个单目运算符，分裂的时候需要注意什么?

你可以选择不实现负数的功能，但你很快就要面临类似的问题了.

调试中的表达式求值

实现了算术表达式的求值之后，你可以很容易把功能扩展到复杂的表达式。我们用 BNF 来说明需要扩展哪些功能：

```
<expr> ::= <decimal-number>
    | <hexadecimal-number>    # 以"0x"开头
    | <reg_name>              # 以"$"开头
    | "(" <expr> ")"
    | <expr> "+" <expr>
    | <expr> "-" <expr>
    | <expr> "*" <expr>
    | <expr> "/" <expr>
    | <expr> "==" <expr>
    | <expr> "!=" <expr>
    | <expr> "&&" <expr>
    | <expr> "||" <expr>
    | "!" <expr>
    | "*" <expr>              # 指针解引用
```

它们的功能和 C 语言中运算符的功能是一致的，包括优先级和结合性，如有疑问，请查阅相关资料。需要注意的是指针解引用(dereference)的识别，在进行词法分析的时候，我们其实没有办法把乘法和指针解引用区别开来，因为它们都是 *。在进行递归求值之前，我们需要将它们区别开来，否则如果将指针解引用当成乘法来处理的话，求值过程将会认为表达式不合法。其实要区别它们也不难，给你一个表达式，你也能将它们区别开来。实际上，我们只要看 * 前一个 token 的类型，我们就可以决定这个 * 是乘法还是指针解引用了，不信你试试？我们在这里给出 expr()函数的框架：

```
if(!make_token(e)) {
  *success = false;
  return 0;
}

/* TODO: Implement code to evaluate the expression. */

for(i = 0; i < nr_token; i ++) {
  if(tokens[i].type == '*' && (i == 0 || tokens[i - 1].type == certain type) ) {
    tokens[i].type = DEREF;
  }
}

return eval(?, ?);
```

其中的 certain type 就由你自己来思考啦！其实上述框架也可以处理负数问题，如果你之前实现了负数，*的识别对你来说应该没什么困难了。

另外和 GDB 中的表达式相比，我们做了简化，简易调试器中的表达式没有类型之分，因此我们需要额外说明两点：

- 为了方便统一，我们认为所有结果都是 uint32_t 类型。
- 指针也没有类型，进行指针解引用的时候，我们总是从内存中取出一个

uint32_t 类型的整数，同时记得使用 mem_read()来读取内存。

实现更复杂的表达式求值

你需要实现上文 BNF 中列出的，除**指针解引用**之外的功能. 一个要注意的地方是词法分析中编写规则的顺序，不正确的顺序会导致一个运算符被识别成两部分，例如 != 被识别成 ! 和 = 。关于变量的功能，它需要涉及符号表和字符串表的查找，本实验不需要实现它。

选做任务 2：实现指针解引用

按上文提示实现对指针解引用运算符‘*’的识别

监视点

监视点的功能是监视一个表达式的值何时发生变化。如果你从来没有使用过监视点，请在 GDB 中体验一下它的作用。

简易调试器允许用户同时设置多个监视点，删除监视点，因此我们最好使用链表将监视点的信息组织起来。框架代码中已经定义好了监视点的结构体(在 temu/include/monitor/watchpoint.h 中)：

```
typedef struct watchpoint {
  int NO;
  struct watchpoint *next;

  /* TODO: Add more members if necessary */


} WP;
```

但结构体中只定义了两个成员：NO 表示监视点序号，next 就不用多说了吧。为了实现监视点功能，你需要根据对监视点工作原理的理解在结构体中增加必要的成员。同时我们使用"池"的数据结构来管理监视点结构体，框架代码中已经给出了一部分相关的代码(在 temu/src/monitor/ watchpoint.c 中)：

```
static WP wp_pool[NR_WP];
static WP *head, *free_;
```

代码中定义了监视点结构的池 wp_pool，还有两个链表 head 和 free_，其中 head 用于组织使用中的监视点结构，free_用于组织空闲的监视点结构，init_wp_pool()函数会对两个链表进行了初始化。

实现监视点池的管理

为了使用监视点池，你需要编写以下两个函数(你可以根据你的需要修改函数的参数和返回值)：

WP* new_wp();

void free_wp(WP *wp);

其中 new_wp()从 free_链表中返回一个空闲的监视点结构，free_wp()将 wp 归还到 free_链表中，这两个函数会作为监视点池的接口被其它函数调用。

需要注意的是，调用 new_wp()时可能会出现没有空闲监视点结构的情况，为了简单起见，此时可以通过 assert(0)马上终止程序。框架代码中定义了 32 个监视点结构，一般情况下应该足够使用，如果你需要更多的监视点结构，你可以修改 NR_WP 宏的值。

这两个函数里面都需要执行一些链表插入，删除的操作，对链表操作不熟悉的同学来说，这可以作为一次链表的练习。

实现了监视点池的管理之后，我们就可以考虑如何实现监视点的相关功能了. 具体的，你需要实现以下功能：

- 当用户给出一个待监视表达式时，你需要通过 new_wp()申请一个空闲的监视点结构，并将表达式记录下来。每当 cpu_exec()执行完一条指令，就对所有待监视的表达式进行求值(你之前已经实现了表达式求值的功能了)，比较它们的值有没有发生变化，若发生了变化，程序就因触发了监视点而暂停下来，你需要将 temu_state 变量设置为 STOP 来达到暂停的效果。最后输出一句话提示用户触发了监视点，并返回到 ui_mainloop()循环中等待用户的命令。
- 使用 info w 命令来打印使用中的监视点信息，至于要打印什么，你可以参考 GDB 中 info watchpoints 的运行结果。

- 使用 d 命令来删除监视点，你只需要释放相应的监视点结构即可。

实现监视点

你需要实现上文描述的监视点相关功能，实现了表达式求值之后，监视点实现的重点就落在了链表操作上. 如果你仍然因为链表的实现而感到调试困难，请尝试学会使用 assertion。

在同一时刻触发两个以上的监视点也是有可能的，你可以自由决定如何处理这些特殊情况,我们对此不作硬性规定。

断点

断点的功能是让程序暂停下来，从而方便查看程序某一时刻的状态。事实上，我们可以很容易地用监视点来模拟断点的功能：

```
w $pc == ADDR
```

其中 ADDR 为设置断点的地址。这样程序执行到 ADDR 的位置时就会暂停下来。

调试器设置断点的工作方式和上述通过监视点来模拟断点的方法大相径庭。事实上，断点的工作原理，竟然是三十六计之中的"偷龙转凤"！如果你想揭开这一神秘的面纱，你可以阅读这篇文章。