

# PA1 下实验报告

首先对 PA1 的思考“cmd\_c()函数在调用 cpu\_exec()中传入了参数-1”做一点补充：cpu\_exec()的参数为 unsigned long long，传入-1 转换则会发生类型转换，得到 n 为最大正数，因此 cpu\_exec() 中的 for 循环会把余下所有指令全部执行，从而达到了 continue 的目的

## 表达式求值 2

### 表达式求值测试 1

利用助教给的 psmd.txt 测试时修复了 PA1 上 expr.c 中的一个问题：eval()函数中的寻找主运算符的代码考虑不周全（见下图）。对于遇到括号的情况时，应该直接循环跳过括号内部分，原来的方法是检测第一次遇到右括号作为结束条件，这显然是不全面的，因此修改为检测与第一个左括号匹配的右括号作为结束条件。然后把寻找主运算符的代码单独写成了一个函数 dominant\_operator()

### 表达式求值扩展

首先需要对 token 集合扩充，在 PA1 上寻找主运算符的算法（右图）的基础上，和 PA1 上一样把部分运算符按照优先级顺序排列。由于扩充运算符后，存在 3 个运算符同优先级的情况，需要在枚举中对特定 token 赋值，保证同优先级运算符对应值除 3 后相等。具体而言最高优先级的为数字、十六进制、寄存器，下面依次是负、解引用、逻辑非，乘、除，加、减，等于、不等于，与，或，左右括号，NOTYPE(空格)和 LETTER(字母)由于不会出现在 tokens 中，其顺序无影响，故放在最后。最后再把原算法中的 tokens[i].type/2 改为除 3。我在这部分开始把与和或归到了同一优先级，导致运算结果总出错。

```
int i, op = 0, pos = 0, val1, val2;
for (i = start; i <= end; i++)
{
    if (tokens[i].type == TK_LEFT_PAR)
    {
        do
            i++;
        while (tokens[i].type != TK_RIGHT_PAR);
        continue;
    }
    if (tokens[i].type / 2 >= op)
    {
        op = tokens[i].type / 2;
        pos = i;
    }
}
```

此部分为未修改前的代码

然后对 rules 数组扩充。在 rules 数组中重复 token 的正则表达式相同，因此负和减共用一个记号 TK\_MINUS（减），解引用和乘共用一个记号 TK\_MUL（乘），后续过程再将它们区分。不等于应该在逻辑非之前列出，否则!=中的!会被错误识别成逻辑非。

isa\_reg\_str2val()函数：函数的功能是识别合法的寄存器，首先对 pc 单独判断；由于 reg.h 中定义了相关寄存器的枚举，且 reg.c 中定义了 regsl[], regsw[], regsb[] 数组，因此可以使用 for 循环逐一比较寄存器名，相同则分别使用 reg\_l(), reg\_w(), reg\_b() 返回寄存器值。isa\_reg\_str2val()函数可以在我定义的 str2int()函数中调用，这个函数本来是用于计算十进制和十六进制数字的值的，现在可以再增加计算寄存器的值的功能，当传入的参数为寄存器类型直接用 isa\_reg\_str2val()函数计算。

重复 token 的区分：对于-号，表示负号时表达式的形式为<expr> op -<expr>，表示减号时表达式的形式为<expr> - <expr>，显然判断-号前面的表达式类型即可区分是负还是减。由于 op 的种类较多，故对前面的<expr>分析，如果是减号则-前面的<expr>必然是数字、十六进制数字、寄存器、右括号中的一种，否则为负号，然后负号还有一种特殊情况即位于整个表达式开头，因此如果运算符-位于表达式首位或者前一个 token 不是数字、十六进制数字、寄存器、右括号中的任意一个，则运算符为负号。对于\*号同理，故-和\*的判断可以统一为一个函数 judge\_op()。judge\_op()可以在 make\_token()函数中调用，若当前识别到

的类型为减或乘，则其在存入 tokens[] 数组后，检查存入的最后一个 token 的真实类型，然后对该 token 的 type 修正。

双目运算符和单目运算符的计算：基本思路是对双目和单目运算符分别处理，具体而言，在 PA1 上的双目运算符的 switch（图 1）结构前先对单目运算符处理（图 2）。由于单目运算符表达式的形式为 op <expr>，且出现的三个单目运算符-、!、\*同优先级，因此只需要计算<expr>即 eval(start + 1, end)，然后用 switch 对 op 判断即可。对于新增的双目运算符，直接在原来的 switch 中添加即可。由于&&和||的特殊性（例如&&左边为 0 则不会计算右边），对于像 0&&1/0 这样的表达式是可以计算的，尽管右边是未定义类型。因此不能再先计算双目运算符两边的表达式再处理，而应采用图 2 的处理方式，这样可以避免出错。

```
val1 = eval(start, pos - 1);
val2 = eval(pos + 1, end);
switch (tokens[pos].type)
{
case TK_ADD:
    return val1 + val2;
case TK_SUB:
    return val1 - val2;
case TK_MUL:
    return val1 * val2;
case TK_DIV:
    return val1 / val2;
case TK_EQ:
    return val1 == val2;
default:
    Assert(0, "Please check you expression\n");
}
```

图1

```
if(start == pos || tokens[pos].type == TK_DEREF || tokens[pos].type == TK_NEG || tokens[pos].type == TK_NOT)
{
    int val = eval(start + 1, end);
    switch(tokens[start].type)
    {
        case TK_NEG:
            return -val;
        case TK_DEREF:
            return paddr_read(val, 4);
        case TK_NOT:
            return !val;
    }
}

switch (tokens[pos].type)
{
case TK_ADD:
    return eval(start, pos - 1) + eval(pos + 1, end);
case TK_MINUS:
    return eval(start, pos - 1) - eval(pos + 1, end);
case TK_MUL:
    return eval(start, pos - 1) * eval(pos + 1, end);
case TK_DIV:
    return eval(start, pos - 1) / eval(pos + 1, end);
case TK_EQ:
    return eval(start, pos - 1) == eval(pos + 1, end);
case TK_NOT_EQ:
    return eval(start, pos - 1) != eval(pos + 1, end);
case TK_AND:
    return eval(start, pos - 1) && eval(pos + 1, end);
case TK_OR:
    return eval(start, pos - 1) || eval(pos + 1, end);
default:
    Assert(0, "Please check you expression\n");
}
```

图2

对于提供的前两种思路的考虑

单目变双目：首先应该需要修改 tokens 数组插入空值，数组插入会大大增加运算的时间复杂度，然后对于“空值”的考虑，如果要把单目运算符统一为 PA1 上的双目运算符处理方法，那么对于 val1 即空值的计算应该不返回任何结果，因此空值可以为表达式中不会出现的类型，然后在 eval() 中对这种特殊类型判断，遇到则不计算直接返回，并且在 switch 结构中单目运算符的计算不再需要 val1，只用 val2 即可。个人觉得这种方法的弊端为计算过程可能会比较慢。

统计连续单目运算符个数，直接将其变为具体值：可以在计算之前扫描一遍表达式，遇到单目运算符则用一个函数提取单目运算符连接的表达式进行计算，然后把单目运算符的计算结果替换掉 tokens 数组中内容，个人觉得这个方法可行且实现也挺简单，运算增加的开销也不多，但对于出现未定义类型的表达式会计算出错，比如 0&&-(1/0)，一开始就扫描计算的话会出错。

## 表达式求值测试 2

根据给出的 gen\_rand\_expr() 结构，首先需要定义相关函数

choose()：这个函数用于根据给定参数 n 返回小于 n 的伪随机正数，使用 rand() % n 即可实现。

`gen_blank()`: 这个函数用于在 `buf` 数组中生成一串空格再生成一个表达式, 可以使用 `sprintf()` 函数直接把空格写入 `buf`, 之后的所有生成函数把内容写入 `buf` 的方法都和这里一样, 由于空格的数量对计算无影响, 我这里就限制最多生成 3 个空格

`gen_num()`: 这个函数用于在 `buf` 数组中生成一个数字, 数字由 `rand() % n` 产生, 写入 `buf` 使用 `sprintf()` 函数, 使用 `switch(choose(2))` 决定写入 `buf` 中的数字为十进制表示还是十六进制表示, 用 `sprintf()` 的格式输出区别即可。但由于在 `main()` 函数内计算表达式正确结果时编译器会把形如 `0x...e-1` 等以 `e` 结尾的十六进制数字和后面的运算符 `-` 和 `+` 识别为指数, 在整数计算中会报错, 因此 `gen_num()` 中默认全部产生十进制数字。在我的程序中默认只产生 1000 以内的数字。

`gen_rand_op()`: 使用 `switch(choose(n))` 决定生成运算符即可。

生成固定字符比如左右括号的函数可以直接用 `sprintf()` 替代

`gen_rand_expr()`: 根据实际情况, `gen_rand_expr()` 分为五种情况: 空格 `<expr>`、数字、`<expr> op <expr>`、`( <expr> )`、`op <expr>`。实际情况下由于 `op <expr>` 中的单目运算符只限制为 `!`, 导致会生成大量的 0 在表达式中, 一旦表达式较长非常容易出现除 0 错误, 而对于除 0 我在后续设置了检测机制, 检测函数每次都对除 0 检测使得生成一个表达式的时间过长, 这对于需要大量实例测试是非常不利的。为了解决上述问题, 我有两种思路: 减小表达式长度或降低 `!` 出现的频率。对于前一种思路, 表达式过长是因为 `gen_rand_expr()` 的五种情况中只有生成数字会结束递归, 其余情况至少会递归一次, 从而造成递归次数过多生成的表达式很长, 因此我把 `op <expr>` 改成 `op 数字`, 增加一个递归结束入口, 这样处理效果很明显, 原来的 `op <expr>` 形式基本上每 10 个里就有一个长达上百个字符的表达式, 修改后大部分都是不超过一行的表达式。对于后一种思路, 可以在第五种情况中使用 `if(choose(n) == k)` 决定是否生成一个逻辑非符号, 再默认生成一个 `<expr>`, 这样处理相比前一种思路可以正确生成较多的长表达式, 但相应花费时间也很长。综合时间因素我在代码中采用了前一种思路, 后一种思路也在注释中给出了相应代码。

除 0 检测: 基本思路是在 `gen_rand_expr()` 的开头先检测当前 `buf` 的末尾字符, 如果为 `/` 号, 则在 `switch` 结构后对新生成的部分表达式计算检测, 如果为 0 则把原来 `/` 号后面生成的部分清除, 重新生成一个表达式。对于如何计算当前新生成的表达式的值, `main()` 函数已经给出了相应代码, 稍微修改一下即可。首先我把 `main()` 函数中给出代码重写成一个函数 `int check_expr(char *buf, int x)`, 返回 0 表达检测到错误, 1 表示正确, 由于不是对 `buf` 整个表达式检测, `char *buf` 为需要计算的字符串头指针, 即 `/` 号后一个字符开始的表达式, `x` 为需要检测的值, 即如果计算结果为 `x` 则返回错误。最后, 由于计算结果为 `unsigned` 和 `int` 类型的表达式在 `gcc` 的内部优化不同, 经过测试在 `unsigned` 下某些除 0 表达式仍可以正确计算, 但在 `int` 下不能 (比如 `0==0/0!=0`, 在 `unsigned` 下会计算为 1, `int` 下无法计算), 因此需要把 `main()` 函数中 `code_format` 中的 `unsigned` 修改为 `int`, 得到新的计算规则 `cd_format` 用于 `check_expr()` 计算。

除 0 检测补充以及其他可能的错误检测: 上面的除 0 检测是有问题的, 由于是对 `/` 号后面生成的表达式检测, 这里忽视了运算符优先级问题, 例如 `1/2*0` 会被检测为错误, `1/0+1` 会被检测为正确, 因此有必要在表达式生成完成后再对整个表达式计算检测一次, 方法为 `check_expr(buf, 0x80000000)`, `buf` 就是生成表达式的头指针, 参数 `0x80000000` 是因为在 `check_expr()` 中保存计算结果的变量 `result` 初始值为 `0x80000000`, 如果表达式计算错误的话 `result` 值不会改变, 因此传入参数 `0x80000000`。由于要控制在表达式生成完成后再检测, 不能在生成过程中检测, 否则表达式不完整, 因此给 `gen_rand_expr()` 一个参数表示其是否为第一次递归, 生成一个新表达式使用 `gen_rand_expr(1)`, 在 `gen_rand_expr()` 内部递归时调用 `gen_rand_expr(0)`, 这样当检测到 1 时才会测试计算整个表达式。

思考 1: 递归的返回入口: 即 `gen_rand_expr()` 中 `switch` 内没有再递归的情况; 控制表达式的长度: 可以在 `gen_rand_expr()` 的 `switch` 中增加空情况, 调整进入返回入口的概率, 可以实现控制表达式的平均长度。暂时还没想到精确控制表达式长度的方法。避免除 0 情况: 见上文的除 0 检测

## 简易调试器 2

### 实现监视点

基本思路: 把 `wp_pool` 分成两条链: 已使用链和未使用链, 分别由 `head` 和 `free_` 控制, 增加、删除监视点通过链表的增删节点方式进行。

WP 结构成员: `bool is_free`、`int NO`、`char msg[100]`、`int val`、`next` 指针

`init_wp_pool()`: 需要在 `for` 循环中增加对 `is_free` 的初始化。

`new_wp()`: 首先判断 `free_` 是否为空, 为空则表示 `wp_pool` 数组已经存满。然后从 `free_` 链中拿出第一个表元接到 `head` 链末尾, 需要对 `head` 为空的特殊情况进行处理。

`free_wp()`: 由于 `wp_pool` 本质上是数组, 可以直接根据序号对 `wp_pool[no]` 清零, 但是还需要调整链表结构, 即把 `head` 中需要删除的表元放入 `free_` 头部, 实现方法即一般的链表增删节点。`no` 的有效性根据 `NR_WP` 判断, `no` 是否在 `head` 中使用成员 `is_free` 判断。

`print_wp()`: 把 `head` 中所有表元依次输出即可

`delete_all_wp()`: 用 `for` 循环对 `wp_pool` 数组扫描, 对于 `is_free` 为 `false` 的表元使用 `free_wp()` 删除即可。

### 命令扩充

`info w`: 调用 `print_wp()` 函数即可

`w <expr>`: 使用 `expr()` 函数计算 `<expr>` 的值 `val`, 然后调用 `new_wp(args, val)` 新建监视点并把 `args` (即 `<expr>`) 存入 `msg`, `val` 存入 `val`

`d n`: `n` 存在时调用 `free_wp(n)`, `n` 不存在时调用 `delete_all_wp()`

### 实现监视功能

由于 `ui.c`、`cpu-exec.c` 中都包含了头文件 `watchpoint.h`, 监视函数 `check_wp()` 定义在 `watchpoint.c` 中则不需要再额外声明; 从其功能上而言, `check_wp()` 也属于监视器的必要函数; 从其实现过程上而言, `check_wp()` 需要 `expr.c` 的 `expr()` 函数, 而 `watchpoint.c` 中包含了头文件 `expr.h`, 因此把 `check_wp()` 定义在 `watchpoint.c` 中是最合适的。


`check_wp()`: 基本思路是对 `head` 中所有表元分别调用 `expr()` 函数计算 `msg` 的值 `new_val`, 然后和存储的旧值 `val` 比较, 如果不等于则更新 `val` 值并返回 1 表示触发监视点。由于需要对 `head` 的所有表元检测, 显然不能触发一次就返回, 因此可以定义一个变量 `hit` 并初始化为 0, 一旦发生触发监视点就把 `hit` 设置为 1, 这样只要触发一次监视点 `hit` 就是 1, 否则 `hit` 为 0, 最后返回 `hit` 即可。

`check_wp()` 的调用: 应该在 `cpu-exec.c` 中调用, 因为 `check_wp()` 的后续功能是设置 `nemu.state` 的状态, 相关代码大部分都在 `cpu-exec.c` 中, 而且 `check_wp()` 属于 debug 功能的函数, 正常运行是不需要的, 注意到 `cpu-exec.c` 中的 `cpu-exec()` 函数中有 `#ifdef DEBUG` 部分, 里面已经有相关调试代码, 由此可知 `check_wp()` 应在 `#ifdef DEBUG` 部分中调用。调用方式为若 `nemu_state.state` 为 `NEMU_RUNNING` 且 `check_wp()` 为 `true` 则把 `nemu_state.state` 设置为 `NEMU_STOP`。一开始我没有考虑 `nemu_state.state` 为 `NEMU_RUNNING` 这个条件, 导致出现了下图的错误。错误的原因在于我设置监视了 `pc`, 然后

一直 c 执行命令，由于 pc 每执行一条指令其值都会变，因此 check\_wp() 始终为真，nemu\_state.state 每次运行都被设置为 NEMU\_STOP。当进行到最后一条指令时，本来 nemu\_state.state 在执行完命令后为 NEMU\_END，然后在后续代码中可以正常退出程序，但是由于 check\_wp() 为真，因此 nemu\_state.state 的 NEMU\_END 值被替换成了 NEMU\_STOP，从而无法退出程序。当再次执行命令 c 时 pc 继续读取指令，导致读取了无效指令码。

```
(nemu) c
[src/monitor/debug/expr.c,120,make_token] match rules[14] = "\\$[a-zA-Z]+" at position 0 with len 3: $pc
Watchpoint 1: $pc
Old value = 0x00100026
New value = 0x00100027
(nemu) c
invalid opcode(PC = 0x00100027): 34 12 00 00 01 00 00 00 ...

There are two cases which will trigger this unexpected exception:
1. The instruction at PC = 0x00100027 is not implemented.
2. Something is implemented incorrectly.
Find this PC(0x00100027) in the disassembling result to distinguish which case it is.

If it is the first case, see

for more details.
```

NEMU\_STOP 的作用：cpu\_exec() 函数的 for 循环最后会根据 nemu\_state.state 是否为 NEMU\_RUNNING 决定是否结束循环，设置 NEMU\_STOP 后会触发结束循环，结束 cpu\_exec() 函数。而 ui\_mainloop() 的 cmd 指令函数负责调用 cpu\_exec()，当 cpu\_exec() 结束后会返回到 cmd 指令函数，cmd 指令函数结束后，返回到 ui\_mainloop() 中等待用户输入下一条指令。