PA1 第二阶段 实验报告

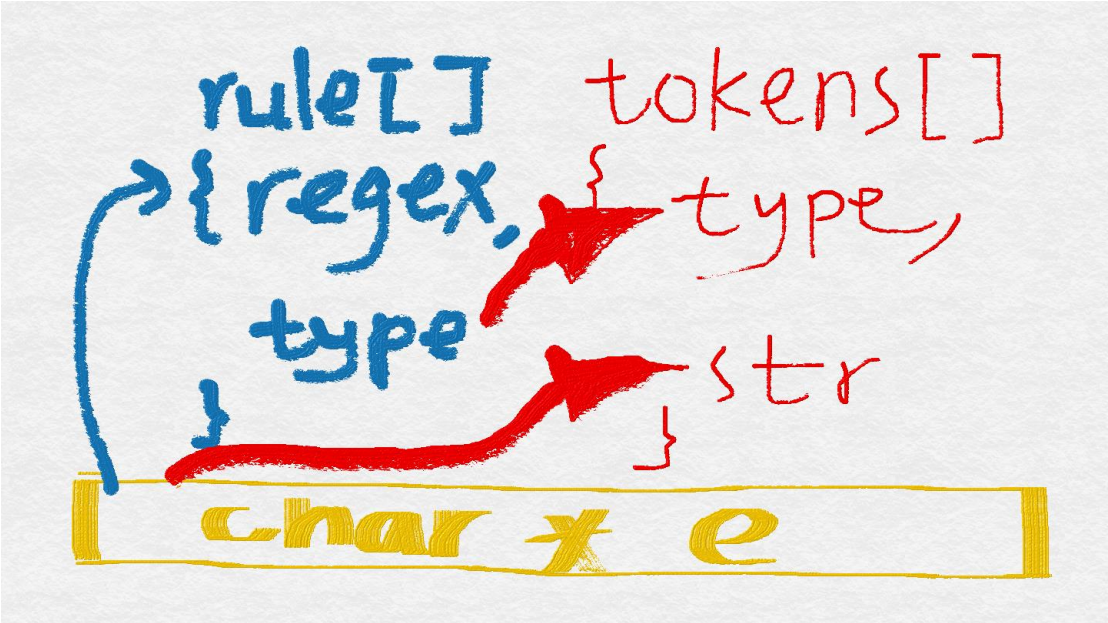
实验进度：

完成 PA1 第二阶段的实验：实现表达式求值，即 **p expr** 命令

必答题：

1 实现词法分析

为了实现这个功能，必须要理解 **expr.c** 的代码框架：



如上图，**char * e** 被所有的正则规则 **rules[]**检查、匹配，匹配成功以后：
成功匹配的字符串部分被传送到 **tokens[].str**
匹配出来的正则表达式类型被传送到 **tokens[].type**
这样，就可以完成词法分析。

这个部分的注意点：

rules[]的顺序会对正则匹配有影响，例如：

```
24      {"0x[0-9a-fA-F]{1,}", HEX},
26      {"[0-9]{1,}", DEC},
```

假如将 16 进制 **HEX** 识别和普通十进制 **DEC** 的识别顺序交换，就无法正确地识别 **HEX**，譬如识别：

0x2341

如果是 **DEC** 规则先匹配，将会匹配出**“0”**，而随后的**“x”**的无法匹配任何规则，这样就会产生错误。

所以，安排规则顺序的原则是将具体的规则放在前面。

相同的例子还有“!”和“!=”：

```
34      {"!=", NEQ},
37      {"\\!", NOT}
```

也遵循上述的原则。

另外，原来的代码安排的 type 并不合理，譬如
{"\\+", ‘+’}

这样会为后续的寻找 dominant operator 带来困难，因为 ASCII 下，‘+’等运算符的编号较大，无法方便地以其类型为下标构造数组，并处理运算符和优先级（具体将在后续的报告中详细说明），因此在运算符的枚举中统一处理更好：

```
9       enum {
10          NOT, PTR, NEG, MUL, DIV, ADD, SUB, EQ, NEQ, AND, OR, DEC,
        HEX, REG, LPA, RPA, NOTYPE = 256
11      };
12      int prior[16] = { 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 5, -3, -3, -
        3, -1, -2 };
```

在后续的说明中将看到，这样的处理又凑巧地处理了负号和解指针的区分。

此外，只要将正则匹配出的内容传入 tokens[].str 即可。这里注意到信息输出：

```
86      Log("match rules[%d] = \"%s\" at position %d with len %d: %.*s",
        i, rules[i].regex, position, substr_len, substr_len,
        substr_start);
```

因此将内容传入 tokens 只需要依葫芦画瓢即可：

```
101     sprintf( tokens[nr_token].str, "%.*s", substr_len,
        substr_start );
```

这里只有“值”需要传入，如果是运算符则可省略此步骤，指定类型即可。

2 实现简易的递归求值

求值函数的框架已经由手册给出，即按照巴科斯范式(BNF)，调用函数

```
150     uint32_t swaddr_read(swaddr_t, size_t);
151     int eval( int p, int q )
```

进行处理。

除了抛出错误的情况 **if(p > q)**以外，先简单地解释 **else if(p == q)**情况：

这个情况下说明 **tokens[p: q]**已经不可再分割为两个表达式，而从递归的方法看，这个不可分割不可能是一个运算符，只可能是 **DEC**，**HEX**，**REG** 三种类型，使用 **switch case** 分别用 **sscanf** 处理即可

```
166     case HEX: sscanf( tokens[p].str, "%x", &val ); break;
167     case DEC: sscanf( tokens[p].str, "%d", &val ); break;
168     case REG:
169     for( i = 0; i < 8; i ++ ){
170         if( strcmp( tokens[p].str, reg_name[i] ) == 0 ){
171             val = cpu.gpr[i]._32; break;
```

```
172         }
173     } break;
```

其中，寄存器的类型稍微麻烦一些，要根据寄存器名称匹配、调用 gpr[]:

```
164     char * reg_name[] = {"eax", "ecx", "edx", "ebx", "esp", "ebp",
        "esi", "edi"};
```

else 情况和函数 static bool check_parentheses(int p, int q, bool * legal)一起放在下一部分讨论。

3 实现更复杂的递归求值

根据手册的描述，else 情况的核心问题就是找到 dominant operator。最初完成 2nd 部分时，我用了比较粗糙的方法，所以导致扩展更多运算符的时候带来了很多麻烦。所以，现在统一用一种方法处理。

首先需要说明的是括号函数 static bool check_parentheses(int p, int q, bool * legal)

这个函数采用了一种类似于栈的简单方法，

```
137         for( i = p; i <= q; i ++ ){
138             if( tokens[i].type ==  LPA )
139                 legal_check ++;
140             if( tokens[i].type ==  RPA )
141                 legal_check --;
142             if( ( legal_check == 0 ) && ( i != q ) )
143                 match = false;
144         }
```

子表达式 tokens[p: q]从尾部向前检查，遇到左括号++(类似于进栈)，遇到右括号--(类似于出栈)，如果没有匹配到最后就出现了零和的情况，说明最外层的括号 tokens[p].type == LPA && tokens[q].type == RPA 虽然存在，但相互不对应。

通过这样的方法，可以检查括号的合理性。

需要说明的是，形参中的 bool * legal 是用来进行分支判断的。因为在接下来的寻找 dominant operator 中，我们希望能在不终止 nemu 进程的情况下调用 check_parentheses，因此需要一个额外的 bool * legal 指针来判断在函数返回 false 的情况下来区分这个子表达式的括号是否合法。

下面说明寻找 dominant operator 的方法：

注意，在 1st 部分提到的枚举处理：

```
9       enum {
10          NOT, PTR, NEG, MUL, DIV, ADD, SUB, EQ, NEQ, AND, OR, DEC,
        HEX, REG, LPA, RPA, NOTYPE = 256
11      };
12      int prior[16] = { 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 5, -3, -3, -
        3, -1, -2 };
```

实现了如下图所示的一个优先级映射：

优先级	识别的内容	enum 所对应的
-3	16 进制数;10 进制数;寄存器值	HEX; DEC; REG
-2	)	RPA
-1	(	LPA
0	!; *(ptr); -(negative)	NOT; PTR; NEG
1	*; \	MUL; DIV
2	+; -	ADD; SUB
3	==; !=	EQ; NEQ
4	&&	AND
5	||	OR

枚举的处理使得运算符的“值”分布在[0, 15]内，还有优先级映射使得我们将不同的运算符按照优先级“聚类”，也就是说能够直接通过数组来输出运算符的优先级：

prior[NOT] == 0, prior[DIV] == 1, prior[ADD] == 2 ……

这使得寻找 dominant operator 变得更为方便。

根据先前说明的 check_parentheses 函数，可以简单地跳过表达式中的小括号：

```
tokens[i].type == RPA
for( j = i - 1; check_parentheses( j, i, &legal ) == false; j -- );
```

对于需要寻找 dominant operator 的表达式 tokens[p: q]，在排除括号的干扰后，设置两个数组对每一个非数值型（即不是 HEX, DEC, REG）tokens[i]进行记录：

```
201    prior_arr[num] = prior[ tokens[i].type ];
202    locate[num] = i;
203    num ++;
```

扫描完整个数组以后，我们对得到的三个数据进行解释：

num	所有非数值型，且排除了括号的 tokens[j] 的数量
prior_arr[i]	i∈[0, num]，第 i 个被记录的符号的优先级
locate[i]	i∈[0, num]，第 i 个被记录的符号在 tokens[]中的位置

也就是：

prior_arr[i] == prior[tokens[locate[i]].type], i∈[0, num]

这样，我们只需要找到 prior_arr[]中最先出现（从 q 到 p）的最大的数（对应最低优先级）—— prior_arr[i]即可。它对应的具体的运算符为 tokens[locate[i]].type

对于单目运算符，我们只能先计算第二个表达式，然后根据是单目运算符还是双目运算符来决定是否计算第一个表达式，例如：

```
int val2 = eval( op + 1, q );
switch( tokens[op].type ){
    case ADD: return eval( p, op - 1 ) + val2; break;
    case NOT: return !(val2); break;
```

最后，我们说明一下如何根据优先级映射来处理负号和解指针。

仔细想一想，会出现减法运算符、解指针和乘法运算符的情况只有如下几种：

减法运算符- 和 负号-　　乘法运算符* 和 解指针*

tokens[i-1]	i	SUB OR NEG	tokens[i-1]	i	MUL OR PTR
DEC,HEX,REG	-	SUB	DEC,HEX,REG	*	MUL
)	-	SUB	)	*	MUL
(	-	NEG	(	*	PTR
!	-	NEG	!	*	PTR
*	-	NEG	*	*	PTR
/	-	NEG	/	*	PTR
+	-	NEG	+	*	PTR
-	-	NEG	-	*	PTR
==	-	NEG	==	*	PTR
!=	-	NEG	!=	*	PTR
&&	-	NEG	&&	*	PTR
||	-	NEG	||	*	PTR

我们在保证运算符优先级的情况下，刻意设计了 **DEC,HEX,REG,RPA,LPA** 的顺序，使得所有被识别为 **NEG** 或 **PTR** 的情况都满足

```
prior[ tokens[i - 1].type ] > -2
```

这样判断更为简洁