表达式求值

一、实现目标

实现一个简单的表达式求值功能，能够将所输入的表达式进行求值。

支持寄存器：如果表达式中包含寄存器名，需要根据寄存器名查找对应的值，并以十六进制的格式输出。

支持十六进制：如果表达式中包含十六进制数，需要将其转换为对应的十进制值。

二、实现思路

表达式求值思路的关键在于将复杂的数学表达式转化为具体的数值结果。

实现过程分以下两个步骤：

1. 首先识别出表达式中的单元
2. 根据表达式的归纳定义进行递归求值

1、词法分析（识别出表达式中的单元）

在 `rules` 中添加完对应的规则后，即可进入下一步，将输入的字符串的每一个Token解析出来，每个 token 可以是数字、运算符、括号、寄存器名等。

例如，对于输入字符串 `3 + 5 * (2 - 1)`，我们可以得到以下 token 列表：

```
1 3, +, 5, *, (, 2, -, 1, )
```

根据输入的字符串，逐个匹配正则表达式规则，生成对应的 token。如果匹配成功，将 token 的类型和字符串存储到 `tokens` 数组中。

2、根据表达式的归纳定义进行递归求值

递归求值

- 如果遇到括号，会递归处理括号内的子表达式。
- 如果遇到运算符，会根据运算符类型和运算符优先级计算子表达式的值。（若处于同一优先级，则从左到右按顺序计算）
- 对于输入字符串 `3 + 5 * (2 - 1)`，递归求值的过程如下：
  - 计算 `(2 - 1)`，得到 `1`
  - 计算 `5 * 1`，得到 `5`

- 计算 `3 + 5`，得到 `8`

三、表达式求值工具测试

测试思路：利用nemu里的随机生成表达式函数随机生成表达式，再将这些表达式进行求值，判断求出来的值与预期结果是否一致。

随机生成表达式函数

```c
// nemu > tools > gen-expr > C gen-expr.c
void gen_rand_expr() {
    switch (choose(3)) {
        case 0:
            gen_num();
            break;
        case 1:
            gen('(');
            gen_rand_expr();
            gen(')');
            break;
        default:
            gen_rand_expr();
            gen_rand_op();
            gen_rand_expr();
            break;
    }
}
```

利用 `./gen-expr 100 > input` 命令将结果存到 `input` 文件中

```
// nemu > tools > gen-expr > input
189 97+(92)
2237 (((66*34)+74)-(81))
-30 ((55/39+(77)-21-((((87))))))
82 82
45 45
60 76*(57)/72
69846 (((73/(1)*((76))-(46)+((24))+((60))*16*(67))))
16 (81/57/44/24)+16
-37 93-75-55+(17/44/70)*30
-2288 68+3-(2336)-(34/90)*(50-965838-38)-(51/19/7570+23)
-4906 (49*24/48+(6-(91))*(17+(28)+83/6))
135663 135663
-68 6438/71+64-(28)*((((15))))+3*66
19684 6438/71+64-(28)*((((15))))+3*6650
92 (((58))+34)
189 97+(92)
69846 (((73/(1)*((76))-(46)+((24))+((60))*16*(67))))
-121 (25)-(90)-(((56))-(((29)/67/((5))/(((((47)-(33)))))))
```

测试结果：

```c
// nemu > tools > gen-expr > C test.c
    while (fgets(line, sizeof(line), file) != NULL) {
        long result;
        char expression[256];
        if (sscanf(line, "%ld %255s", &result, expression) == 2) {
            f_len++;
            bool success = true;
            init_regex();
            long res = expr(expression, &success);
            if (!success) {
                puts("invalid expression!");
            } else {
                assert(result == res);
            }
        } else {
            printf("Invalid line: %s", line);
        }
    }
    fclose(file);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS        bash - gen-expr

lemonpa@lemonpa-virtual-machine:~/Desktop/proj241-PA_for_LA/nemu/tools/gen-expr$ gcc test.c -o test
lemonpa@lemonpa-virtual-machine:~/Desktop/proj241-PA_for_LA/nemu/tools/gen-expr$ ./test
lemonpa@lemonpa-virtual-machine:~/Desktop/proj241-PA_for_LA/nemu/tools/gen-expr$
```

四、开发过程中的Bug与处理方式

Bug 1：十六进制解析错误

现象：表达式求值显示的十六进制数值结果均为0

问题分析与定位：显示 `Log` 后发现程序将十六进制的 (例) `0xa0` 识别成了 `0` 和 `xa0` ，因此推测是在 `Token` 识别类型的时候出了问题。通过查询 `make_token` 函数，再到 `rules` 数组，最终定位到问题在 `rules` 数组中， `TK_NUMBER` 在 `TK_HEX` 前方，导致每次解析时候都先将 `0xa0` 的第一个 `0` 处理成 `TK_NUMBER` 类型了。

```
Welcome to loongarch32r-NEMU!
For help, type "help"
(nemu) p 0xa0
[src/monitor/sdb/expr.c:92 make_token] match rules[10] = "[0-9]+" at position 0 with len 1: 0
[src/monitor/sdb/expr.c:92 make_token] match rules[12] = "[a-zA-Z_][a-zA-Z0-9_]*" at position 1 with len 3: xa0
res = 0
```

解决方式：将 `rules` 中 `TK_NUMBER` 的位置放在 `TK_HEX` 后方即可解决。一个细节问题，时刻提醒我们执行顺序的重要性。

Bug 2：溢出处理bug

现象：在发现溢出数据时，则需对溢出的数值进行一个标注或提示，显示这个数据是有溢出情况并做出截断处理的，最终的结果将与预期值不符合。

因此我们很容易可以在代码中写下这样的判断逻辑：

```
int val1 = eval(p, op - 1);
int val2 = eval(op + 1, q);
switch (tokens[op].type) {
    case '+':
        if ((val1 > 0 && val2 > INT_MAX - val1) ||
            (val1 < 0 && val2 < INT_MIN - val1)) {
            printf("error: Integer overflow detected in addition\n");
            return -1; // 返回-1或进行其他处理
        }
        .....
}
```

但这样会遇到一个问题，只有一个数值或一端数值溢出时并不会提示出现溢出情况，如下图：

```
Welcome to loongarch32r-NEMU!
For help, type "help"
(nemu) p 123456789123
123456789123 = -1097262461
(nemu) p 123456789123 + 1
123456789123 + 1 = -1097262460
```

问题分析与定位：通过调试与分析得知，其根本原因在于，没有在选择 `Token` 类型的时候就判断溢出处理，而是等到 `val1` 和 `val2` 已经截断后才去判断。简单来说，由于 `val1` 和 `val2` 的值都是通过 `eval` 函数获得，而 `eval` 函数的返回值是一个 `int` 类型，所以此时获取到的 `val1` 和 `val2` 都是经过截断处理的，上述的代码逻辑就仅判断计算的结果是否超过 `INT_MAX` 或 `INT_MIN` 。

解决方式：在选择 `Token` 类型之后，计算结果之前也做判断溢出处理。

Bug 3：解析表达式时出现段溢出

现象：在解析较长的表达式时，如：`(66+87*(77+(48-100))+(43*15/(25)+(92))/55*16-42+((65/85+90*42+21*5)*63)+92/49)`。会发送段溢出，导致无法正常计算。

```
[src/monitor/monitor.c:32 welcome] Build time: 12:58:01, Apr 27 2024
Welcome to loongarch32r-NEMU!
For help, type "help"
(nemu) p (66+87*(77+(48-100))+(43*15/(25)+(92))/55*16-42+((65/85+90*42+21*5)*63) +92/49)
make: *** [/home/lemonpa/Desktop/proj241-PA_for_LA/nemu/scripts/native.mk:38: run] Segmentation fault (core dumped)
```

问题分析与定位：通过调试与分析得知，问题的关键在于这个表达式长度太长，因此由于存放表达式解析结果的数组放不下，在访问 `index = sizeof(str)` （第 `sizeof(str)` +1 位）的位置时，便出现了数组访问越界，导致发送段溢出。

解决方式：最简单的做法，增加 `str` 的长度。（从原本的32 改 64）

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

static Token tokens[32] __attribute__((used)) = {};
static int nr_token __attribute__((used))  = 0;
```

```c
typedef struct token {
  int type;
  char str[64];
} Token;

static Token tokens[64] __attribute__((used)) = {};
static int nr_token __attribute__((used))  = 0;
```

同时还需要进行风险处理，将可能造成越界访问的情况进行处理应对。

```c
if(nr_token >= 64){
  printf("The expression is too long!\n");
  return false;
}
```