# TinyExpr

## https://github.com/codeplea/tinyexpr

TinyExpr 是一个非常小巧的递归下降解析器和数学表达式计算引擎。当你想在运行时添加计算数学表达式的功能,而不想给项目增加太多额外负担时,它会很有用。

除了标准的数学运算符和优先级之外,TinyExpr 还支持标准 C 数学函数和运行时变量绑定。

## 特性

- 纯 C99 实现,无依赖
- 仅包含一个源文件和一个头文件
- 简单快速
- 实现标准运算符优先级
- 支持标准 C 数学函数(sin、sqrt、ln 等)
- 可以轻松添加自定义函数和变量
- 可以在计算时绑定变量
- 使用 zlib 许可证发布 – 几乎可以免费用于任何用途
- 易于使用和集成到你的代码中
- 线程安全(前提是你的 malloc 是线程安全的)

## 构建

TinyExpr 仅包含两个文件: `tinyexpr.c` 和 `tinyexpr.h` 。要使用 TinyExpr,只需将这两个文件添加到你的项目中即可。

## 简单示例

下面是一个在运行时计算表达式的最小示例:

```c
#include "tinyexpr.h"
printf("%f\n", te_interp("5*5", 0)); /* 打印25 */
```

## 用法

TinyExpr 只定义了四个函数:

```c
    double te_interp(const char *expression, int *error);
    te_expr *te_compile(const char *expression, const te_variable *variables,
```

```
    int var_count, int *error);
    double te_eval(const te_expr *expr);
    void te_free(te_expr *expr);
```

### te_interp

```c
    double te_interp(const char *expression, int *error);
```

`te_interp()` 接受一个表达式并立即返回其结果。如果存在解析错误, `te_interp()` 返回 `NaN` 。

如果错误指针参数不为 `0` ,则在失败时 `te_interp()` 会将 *`error`* *设置为解析错误的位置,在成功时将* `error` 设置为 `0` 。

示例用法:

```c
double te_interp(const char *expression, int *error);
int error;
double a = te_interp("(5+5)", 0); /* 返回10 */
double b = te_interp("(5+5)", &error); /* 返回10,error设置为0 */
double c = te_interp("(5+5", &error); /* 返回NaN,error设置为4 */
```

### te_compile、te_eval、te_free

```c
    te_expr *te_compile(const char *expression, const te_variable *lookup,
    int lookup_len, int *error);
    double te_eval(const te_expr *n);
    void te_free(te_expr *n);
```

向 `te_compile()` 传入一个带有未绑定变量的表达式以及变量名称和指针列表。 `te_compile()` 将返回一个 `te_expr*` ,可以稍后使用 `te_eval()` 计算。如果失败, `te_compile()` 将返回 `0` ,并可选择将传入的 `*error` 设置为解析错误的位置。

你也可以通过将 `te_compile()` 的第二和第三个参数设为 `0` 来编译不带变量的表达式。

向 `te_eval()` 传入一个来自 `te_compile()` 的 `te_expr*` 。 `te_eval()` 将使用当前变量值计算表达式。

完成后,请确保调用 `te_free()` 。

### 示例:

```c
    double x, y;
    /* Store variable names and pointers. */
    te_variable vars[] = {{"x", &x}, {"y", &y}};

    int err;
    /* Compile the expression with variables. */
```

```
    te_expr *expr = te_compile("sqrt(x^2+y^2)", vars, 2, &err);

    if (expr) {
        x = 3; y = 4;
        const double h1 = te_eval(expr); /* Returns 5. */

        x = 5; y = 12;
        const double h2 = te_eval(expr); /* Returns 13. */

        te_free(expr);
    } else {
        printf("Parse error at %d\n", err);
    }
```

## 更长的示例

这是一个完整的示例，可以计算从命令行传入的表达式。

它还进行错误检查，并将变量 $x$ 和 $y$ 分别绑定为 $3$ 和 $4$。

C

```c
#include "tinyexpr.h"

#include <stdio.h>

int main(int argc, char *argv[])

{

  if (argc < 2) {

    printf("用法: example2 \"表达式\"\n");

    return 0;

  }

  const char *expression = argv[1];

  printf("正在计算:\n\t%s\n", expression);

  /* 这个示例展示了如何在计算时绑定变量
    - x 和 y */

  double x, y;

  te_variable vars[] = {{"x", &x}, {"y", &y}};

  /* 编译表达式并检查错误 */

  int err;
```

```c
  te_expr *n = te_compile(expression, vars, 2, &err);

  if (n) {

    /* 变量可以在这里更改，eval 可以被调用多次
      - 这种方式很高效，因为解析已经完成 */

    x = 3; y = 4;

    const double r = te_eval(n);

    printf("计算结果:\n\t%f\n", r);

    te_free(n);

  } else {

    /* 向用户显示错误位置 */

    printf("\t%*s^\n错误位置在这里", err-1, "");

  }

  return 0;
```

这会产生以下输出：

```bash
$ example2 "sqrt(x^2+y2)"
    Evaluating:
            sqrt(x^2+y2)
                      ^
    Error near here


$ example2 "sqrt(x^2+y^2)"
    Evaluating:
            sqrt(x^2+y^2)
    Result:
            5.000000
```

## 绑定自定义函数

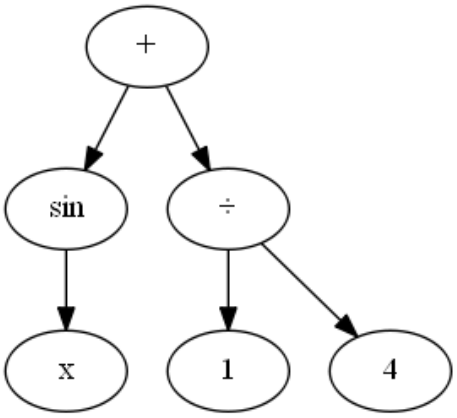
TinyExpr 还可以调用用 C 实现的自定义函数。这里是一个简短的示例：

```c
double my_sum(double a, double b) {
    /* 示例C函数,将两个数相加 */
    return a + b;
}

te_variable vars[] = {
```

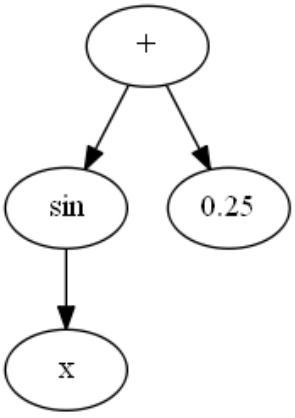
```
    {"mysum", my_sum, TE_FUNCTION2} /* 使用TE_FUNCTION2是因为my_sum接受两个参数 */
};

te_expr *n = te_compile("mysum(5, 6)", vars, 1, 0);
```

## 工作原理

`te_compile()` 使用简单的递归下降解析器将你的表达式编译成语法树。例如,表达式"`sin x + 1/4`"解析为:




`te_compile()` 还会自动剪枝常量分支。在这个例子中`,te_compile()` 返回的编译后表达式将变为:




`te_eval()` 将通过指针自动加载任何变量,然后计算并返回表达式的结果。

`te_free()` 在你完成编译表达式的使用后应该始终被调用。

## 速度

与 C 相比,TinyExpr 在以下情况下相当快:

- 表达式较短
- 表达式进行复杂计算(如指数运算)
- 部分工作可以被 `te_compile()` 简化

与 C 相比,TinyExpr 在以下情况下较慢:

- 表达式较长且只涉及基本算术运算

这里是一些来自包含的 `benchmark.c` 程序的性能数据示例:

| 表达式 | te_eval 时间 | 原生 C 时间 | 速度比较 |
| --- | --- | --- | --- |
| sqrt(a^1.5+a^2.5) | 15,641 ms | 14,478 ms | 慢 8% |
| a+5 | 765 ms | 563 ms | 慢 36% |
| a+(5*2) | 765 ms | 563 ms | 慢 36% |
| (a+5)*2 | 1422 ms | 563 ms | 慢 153% |
| (1/(a+1)+2/(a+2)+3/(a+3)) | 5,516 ms | 1,266 ms | 慢 336% |

## 语法

TinyExpr 解析以下语法:

```
<list>   = <expr> {"," <expr>}
<expr>   = <term> {("+" | "-") <term>}
<term>   = <factor> {("*" | "/" | "%") <factor>}
<factor> = <power> {"^" <power>}
<power>  = {("-" | "+")} <base>
<base>   = <constant>
         | <variable>
         | <function-0> {"(" ")"}
         | <function-1> <power>
         | <function-X> "(" <expr> {"," <expr>} ")"
         | "(" <list> ")"
```

此外,标记之间的空白将被忽略。

有效的变量名由一个字母开头,后跟任意组合的:字母、数字 0–9 和下划线。常量可以是整数或浮点数,可以是十进制、十六进制(如 *0x57CEF7*)或科学计数法(如 *1e3* 表示 1000)。不需要前导零(如.5 表示 0.5)。

## 支持的函数

TinyExpr 支持加法(+)、减法/取反(–)、乘法(*)、除法(/)、指数运算(^)和取模(%)等运算,具有正常的运算符优先级(唯一的例外是指数运算是从左到右计算的,但这可以更改 – 见下文)。

还支持以下 C 数学函数:

- abs(调用 fabs)、acos、asin、atan、atan2、ceil、cos、cosh、exp、floor、ln(调用 log)、log(默认调用 log10,见下文)、log10、pow、sin、sinh、sqrt、tan、tanh

TinyExpr 还内置并提供了以下函数:

- fac(阶乘,如 `fac 5 == 120`)
- ncr(组合数,如 `ncr(6,2) == 15`)
- npr(排列数,如 `npr(6,2) == 30`)

此外,还提供以下常量:

- `pi、e`

## 编译时选项

默认情况下,TinyExpr 从左到右计算指数运算。例如:

`a^b^c == (a^b)^c` 且 `-a^b == (-a)^b`

这是设计使然,这也是电子表格(如 Excel、Google Sheets)的计算方式。

如果你希望指数运算从右到左进行,需要在编译 tinyexpr.c 时定义 TE_POW_FROM_RIGHT。在该文件顶部附近有一个被注释掉的定义。启用此选项后,行为将变为:

`a^b^c == a^(b^c)` 且 `-a^b == -(a^b)`

这将匹配许多脚本语言(如 Python、Ruby)的做法。

另外,如果你希望 `log` 默认为自然对数而不是 `log10`,那么你可以定义 `TE_NAT_LOG`。

## 提示

- 所有函数/类型都以字母 te 开头
- 要允许常量优化,请将常量表达式用括号括起来。例如,"x+(1+5)"将在编译时计算"(1+5)"表达式,并将整个表达式编译为"x+6",节省运行时计算。括号很重要,因为 TinyExpr 不会改变求值顺序。如果你改为编译"x+1+5",TinyExpr 会坚持先将"1"加到"x"上,然后将"5"加到结果上。