

# PROGC Lab07 Calculator

## Inhalt

PROGC Lab07 Calculator .....	1
1 Einführung.....	1
2 Lernziele .....	1
3 Hintergrund Informationen.....	1
3.1 Tests .....	1
3.2 Was ist eine Lookup Tabelle .....	2
3.3 Verwendung der Anführungsstriche in der Bash Shell.....	2
3.4 Calculator Aufruf .....	3
3.5 Expression Syntax .....	3
3.6 Verwendete zusätzliche Sprach Elemente .....	4
4 Aufgaben.....	5
4.1 Lookup-Tabelle für unäre und binäre Operator Funktionen .....	5

## 1 Einführung

In diesem Praktikum lernen Sie eine Funktions-Lookup Tabelle für einen Rechner für einfache Ausdrücke zu implementieren. Der Rechner liest einen Ausdruck vom ersten Argument auf der Kommandozeile ein. In der Aufgabe implementieren Sie das Berechnen der Werte der Operatoren aus 4.1 (das Parsen der Syntax Elemente von 3.5 ist vorgegeben und muss somit nicht implementiert werden).

## 2 Lernziele

- Sie können einen **struct** Typen definieren der eine Assoziation zwischen einem Operator (z.B. "+") und einer Funktion (z.B. `add(double a, double b)`) herstellt.
- Sie wissen wie man aus obigem **struct** je eine Lookup-Tabelle für unäre (z.B. "-5") und binäre (z.B. "3-5") Operatoren anlegt und initialisiert.
- Sie können die Lookup-Tabellen nach passender Funktion für einen gegebenen Operator absuchen und die entsprechende Funktion aufrufen.

## 3 Hintergrund Informationen

### 3.1 Tests

Die Tests werden zu Beginn alle brechen. Ihre Aufgabe ist es, das Praktikumsprogramm so zu implementieren dass die Tests alle den Status „passed“ haben ohne den Test-Code oder deren Stimulus und erwarteten Resultat Daten zu manipulieren.

### 3.2 Was ist eine Lookup Tabelle

Eine Lookup Tabelle ist eine Datenstruktur mit welcher Werte nicht anhand des Index, sondern anhand eines Schlüssels nachgeschlagen werden (Lookup). Es gibt einfachere und optimiertere Varianten von Lookup Tabellen. Wir werden hier die einfachste Form verwenden: einen Array von Schlüssel-Wert Paaren.

Wir werden im Beispiel zwei Lookup Tabellen definieren: Eine erste für unäre Operationen, eine zweite für binäre Operationen.

D.h. der Schlüssel ist jeweils das Operator Symbol (z.B. '\*' für Multiplikation) und der Wert ist ein Pointer auf eine Funktion (welche z.B. die Multiplikation von zwei Argumenten ausführt).

Eine Lookup Tabelle für unäre Operatoren sähe z.B. so aus:

```
/// nop function
double nop(double value) { return value; }
/// neg function
double neg(double value) { return -value; }
...
/// function pointer type for unary function (only one argument)
typedef double (*unary_t)(double);
/// mapping of operator symbol to operator function
typedef struct {
    char op;    ///< operator symbol
    unary_t exec; ///< mapped operator function
} map_unary_t;
/// lookup table for unary operations
map_unary_t unary_lookup[] = {
    { EXPR, nop },
    { PLUS, nop },
    { MINUS, neg },
    ...
};
...
double evaluate_unary(char op, double value) {
    for(size_t i = 0; i < sizeof(unary_lookup)/(sizeof(*unary_lookup); i++) {
        if (unary_lookup[i].op == op) {
            return (*unary_lookup[i].exec)(value);
        }
    }
    error("unknown operator: %c", op);
    return 0.0;
}
```

### 3.3 Verwendung der Anführungsstriche in der Bash Shell

Die Bash Shell interpretiert die Kommandozeile grob nach folgendem Muster:

1. Die eingegebene Kommandozeile wird in Elemente aufgeteilt welche durch ein oder mehrere Spaces getrennt sind.  
*command arg1 arg2*
2. Das erste Element wird das Kommando angenommen, die restlichen Elemente als Argumente für dieses Kommando.
3. Will man Spaces in einem der Elemente haben, muss das entsprechende Element in einfache Anführungsstriche gesetzt werden.  
*command 'this is arg 1' arg2 '3rd arg has also spaces'*
4. Die Bash Shell wertet Text der mit einem \$ beginnt als Variablen aus. Text innerhalb von einfachen Anführungsstrichen setzt diese Auswertung aus.  
*echo 'shell = \$SHELL'*  
resultiert als erstes Argument an das *echo* Kommando  
*shell = \$SHELL*

5. Will man diese Auswertung erlauben muss man doppelte Anführungsstriche verwenden.

```
echo "shell = $SHELL "
```

resultiert als erstes Argument an das `echo` Kommando

```
shell = /bin/bash
```

#### Anmerkungen:

- Die Anführungsstriche werden durch die Bash Shell interpretiert und dann eliminiert. Sie sind nicht Teil des Elements welches schliesslich als Kommando interpretiert oder als Argument an das Kommando weitergegeben wird. **D.h. ein Kommando sieht die Anführungsstriche nicht.**
- Die Bash Shell kennt noch die `~` Substitution welche optisch leicht mit den einfachen Anführungsstrichen verwechselt werden kann. Auf diese Substitution gehen wir hier nicht weiter ein.
- Bash Shell Variablen sind ein weites Feld für eigene Entdeckungen ☺. Die sogenannten Environment Variablen können mit dem Kommando `env` aufgelistet werden. Mit `export var=value` kann eine neue angelegt oder eine existierende überschrieben werden. Die Environment Variablen dienen dazu, einem neuen Prozess (z.B. einem Kommando das aus der Shell aufgerufen wird), ein Satz von Variablen (neben den expliziten Argumenten) mit zu geben. Siehe *man 3 getenv*.
- Kommandos werden in der Bash Shell gefunden weil sie
  - entweder in Bash eingebaute Kommandos sind (z.B. `cd`)
  - oder weil sie User programmierte Bash Shell Funktionen sind
  - oder weil sie über die `$PATH` Environment Variable gefunden werden können
- Mittels `which command` kann gefragt werden wo in `$PATH` ein Kommando gefunden wird. Z.B.
 

```
which ls
```

### 3.4 Calculator Aufruf

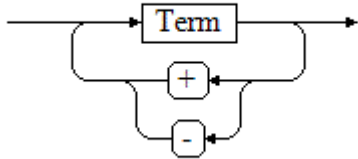
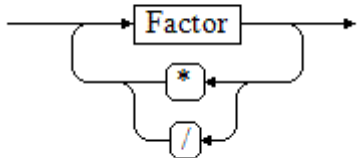
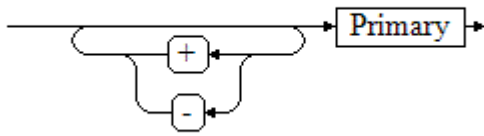
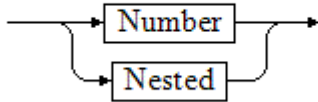
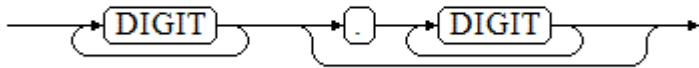
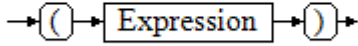
Eine Calculator Kommandozeile kann wie folgt aussehen (siehe 3.2 betreffend Anführungsstriche), d.h. es ist ein normaler „in-fix“ Arithmetik Ausdruck wie wir das von den Primarschule her kennen.

```
bin/calculator '2 + 3 * -(4 + 5)'
```

Hier wird das erste Argument als auszuwertender Ausdruck angenommen.

### 3.5 Expression Syntax

Syntax Element	Operator (siehe 4.1)
<p>Calculator</p> <p>→ Expression →</p>	<p>Unary: EXPR</p>

Syntax Element	Operator (siehe 4.1)
<p>Expression</p> 	Binary: +, -
<p>Term</p> 	Binary: *, /
<p>Factor</p> 	Unary: +, -
<p>Primary</p> 	n/a
<p>Number</p> 	Unary: NUM
<p>Nested</p> 	Unary: NESTED

### 3.6 Verwendete zusätzliche Sprach Elemente

Sprach Element	Beschreibung
<code>double (*func) (double)</code>	Funktions-Pointer auf eine Funktion die ein double Argument entgegen nimmt und einen double Return Wert hat.

Sprach Element	Beschreibung
<code>double (*func)(double, double)</code>	Funktions-Pointer auf eine Funktion die zwei double Argumente entgegen nimmt und einen double Return Wert hat.
<code>double add(double a, double b)</code> { return a + b; }	Funktionsdefinition einer <u>binären</u> Operator Funktion: die Werte werden addiert.
<code>double nop(double v)</code> { return v; }	Funktionsdefinition einer <u>unären</u> Operator Funktion: der Wert wird unverändert zurückgegeben.
<code>double neg(double v)</code> { return -v; }	Funktionsdefinition einer <u>unären</u> Operator Funktion: der Wert wird negiert zurückgegeben.
<code>typedef struct {   char op;   double (*exec)(double); } unary_map_t; unary_map_t unary_lookup[] = {   { EXPR, nop },   { '+', nop },   { '-', neg },   ... };</code>	Lookup Tabelle ist ein Array von Map-Elementen. Ein Map-Element ist ein Schlüssel-Wert Paar.
<code>...sizeof(table)/sizeof(*table)...</code>	Idiomatisches C: Angabe der Anzahl Elemente eines Arrays (z.B. der <code>table</code> Variable)

## 4 Aufgaben

### 4.1 Lookup-Tabelle für unäre und binäre Operator Funktionen

Ergänzen Sie in `lab07-calculator/src/evaluate.c` den Code so dass die Tests erfolgreich durchlaufen.

- Definieren Sie die unären Operator Funktionen `double nop(double)` und `double neg(double)`.
- Definieren Sie die binären Operator Funktionen `double add(double, double)`, `double sub(double, double)`, `double mul(double, double)` und `double div(double, double)`.
- Definieren Sie eine Lookup Tabelle für **unäre** Operatoren EXPR, NESTED, NUM, '+' und '-'. Initialisieren Sie diese mit den entsprechenden Operator-zu-Funktion Assoziation (siehe 3.6).
- Definieren Sie eine Lookup Tabelle für **binären** Operatoren '+', '-', '\*' und '/'. Initialisieren Sie diese mit den entsprechenden Operator-zu-Funktion Assoziation (siehe 3.6).
- Ergänzen Sie die beiden Funktionen `double evaluateUnaryOp(char op, double right)` und `double evaluateBinaryOp(char op, double left, double right)` so dass die zum Argument `op` passende Funktion in der entsprechenden Lookup-Tabelle gefunden wird.

- Rufen Sie die gefundene Funktion mit den/dem gegebenen Argument(en) auf und geben Sie das Resultat davon zurück.