

# 1 Rechenoperationen

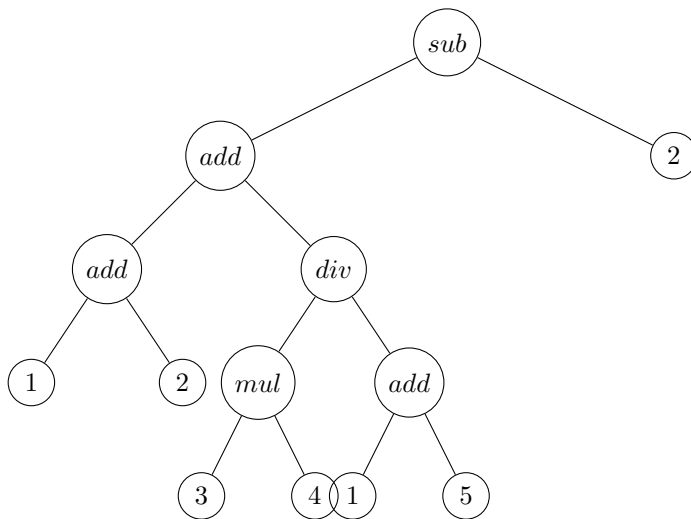
1. Baum besteht aus Knoten (Kreise) und Kanten (Pfeile)
2. Kanten verbinden Knoten mit ihren Kind-Knoten
3. jeder Knoten (außer der Wurzel) hat genau ein Elternteil
4. Knoten ohne Kinder heißen “Blätter“ (leaf-nodes)
5. Teilbaum
  - (a) wähle beliebigen Knoten
  - (b) entferne temporär dessen Eltern-Kante
    - i. der Knoten wird temporär zu einer Wurzel
    - ii. dieser Knoten mit allen seinen Nachkommen bildet wieder seinen Baum - “ Teilbaum des Originalbaums“
  - (c) Tiefe: Abstand des Knotens zur Wurzel
  - (d)

Infix-Notation:

$$1 + 2 + 3 * 4 / (1 + 5) - 2$$

Präfix-Notation:

*sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)*



### Präfix Notation aus dem Baum rekonstruieren

1. Wenn die Wurzel ein Blatt ist, dann "Drucke die Zahl"
2. sonst (Operator):
  - (a) Drucke Funktionsnamen
  - (b) Drucke "("
  - (c) wiederhole ab 1) für das linke Kind
  - (d) Drucke ","
  - (e) wiederhole den Algorithmus ab 1) für das rechte Kind
  - (f) Drucke ")"

Beachte Reihenfolge: Wurzel - Links - Rechts (Pre-Order Traversal) Ergebnis:  
 $sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)$

**Definition: Rekursion** Rekursion meint Algorithmus für Teilproblem von vorn

### Infix Notation

1. wie bei Präfix
2. sonst
  - (a) entfällt
  - (b) wie bei Präfix
  - (c) wie bei Präfix
  - (d) Drucke Operatorsymbol
  - (e) wie bei Präfix
  - (f) wie bei Präfix
  - (g) wie bei Präfix

Beachte Reihenfolge: Links - Wurzel - Rechts (In-Order Traversal)

Ergebnis:  
 $((1 + 2) + ((3 * 4) / (1 + 5))) + 2$

### Berechne den Wert mit Substitutionsmethode

1. Wenn Wurzel ein Blatt hat, gib die Zahl zurück
2. sonst
  - (a) entfällt
  - (b) entfällt
  - (c) wiederhole ab 1) für linken Teilbaum und speichere Ergebnis als "left-result"

- (d) entfällt
- (e) wiederhole ab 1) für rechten Teilbaum, speichere Ergebnis als “right-result“
- (f) berechne  $fkt\_name(left - result, right - result)$  und gib Ergebnis zurück

Beachte Reihenfolge: Links - Rechts - Wurzel (Post-Order Traversal)

$$\begin{aligned}
 & sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2) \\
 &= sub(add(add(1, 2), div(12, 6)), 2) \\
 &= sub(add(3, 2), 2) \\
 &= sub(5, 2) \\
 &= 3
 \end{aligned}$$

## 2 Maschinensprache

- optimiert für die Hardware (viele verschiedene)
- Gegensatz: höhere Programmiersprache ( $C++$ ) ist optimiert für Programmierer
- Compiler oder Interpreter übersetzen Hoch- in Maschinensprache

### Vorgang des Übersetzens

1. Eingaben (und Zwischenergebnisse) werden in Speicherzellen abgelegt  $\Rightarrow$  jeder Knoten im Baum bekommt eine Speicherzelle (Maschinensprache: durchnummeriert ; Hochsprache: sprechende Namen)
2. Speicherzellen für die Eingaben *initialisieren* ; Notation:  $SpZ \leftarrow Wert$
3. Rechenoperationen in der Reihenfolge des Substitutionsmodells ausführen und in der jeweiligen Speicherzelle speichern ; Notation:  $SpZ\_Ergebnis \leftarrow fkt\_name\ SpZ\_Arg1\ SpZ\_Arg2$
4. alles in Zahlencode umwandeln
  - Funktionsname  $\Rightarrow$  Opcodes
  - Speicherzellen: nur die Nummer
  - Werte sind schon Zahlen
  - Notation: Opcode    Ziel SpZ    SpZ\_Arg1    SpZ\_Arg2 oder Opcode    Ziel SpZ    Initialwert

## 3 funktionale Programmierung

(alles durch Funktionsaufrufe ausführen)

1. bei Maschinensprache wurden Zwischenergebnisse in Speicherzellen abgelegt

2. das ist auch in der funktionalen Programm. eine gute Idee

- (a) Speicherzellen werden durch Namen (vom Programmierer vergeben) unterschieden
- (b) Beispiel: Lösen einer quadratischen Gleichung:  $ax^2+bx+c=0$ , finde  $x_{1/2} \Rightarrow x^2-px+q=0$  mit  $p = -\frac{b}{2a}, q = \frac{c}{a} \Rightarrow x_1 = -\frac{b}{2a} + \sqrt{(-\frac{b}{2a})^2 - \frac{c}{a}}$   
 $\Leftarrow$  allgemein :  $x_{1/2} = p \pm \sqrt{p^2 - q}$
- (c) Präfix:  
 $x_1 \leftarrow add(div(div(b, a), -2), sqrt(sub(mul(div(div(b, a), -2), div(div(b, a), -2)), div(c, a))))$   
mit Zwischenergebnissen und Infix-Notation:  $p \leftarrow b/c/-2$  oder  $p \leftarrow -0,5*b/a$   $q \leftarrow c/a$   
 $discriminant \leftarrow sqrt(p * P - q)$   
 $x_{1/2} \leftarrow p \pm discriminant$

3. zwei Vorteile:

- (a) lesbar
- (b) redundante Berechnung verschieden  
Beachte: In der funktionalen Programmierung können die Speicherzellen nach der Initialisierung nicht mehr verändert werden
- (c) Speicherzellen mit Namen sind nützlich, um Argumente an Funktionen zu übergeben  $\Rightarrow$  Definition eigener Funktionen  
Bsp: 

function sq(x){ return x\*x }

## 4 funktionale Programmierung in C++

1. in C++ hat jede Speicherzelle einen Typ (legt Größe und Bedeutung der Speicherzelle fest)  
wichtigste Typen: "int" für ganze Zahlen, "double" für reelle Zahlen, "std::string" für Text  
zugehörige Literale (Konstanten): 12, -3 (int) -1.02, 1.2e-4 (double) "text text" (string)
2. Die Initialisierung wird geschrieben als

```
typ_name spz_name = initialwert
```

Bsp:

```
double a = 10  
std::cout << "x_1" << x_1 << "\n" ;
```

3. eigene Funktion in C++:

```
type_ergebnis funktionsname (typ_arg1 name_arg1, typ_arg2 name_arg2)  
{  
    <code>  
    return ergebnis;  
}
```

4. zwei Funktionen mit gleichem Namen, aber unterschiedlichen Typen dürfen in C++ gleichzeitig definiert sein (“overloading”)  
⇒ C++ wählt automatisch die richtige Variante anhand des Argumenttypes (“overload resolution”)

5. jedes C++-Programm muss genau eine Funktion names “main” haben: Dort beginnt die Programm-Ausführung

Bsp:

```
int main() { <code> return 0 (erfolgreich abgearbeitet) }
```

6. Regel von C++ für erlaubte Namen (Speicherzelle & Funktion):

- (a) erste Zeichen: Klein- oder Großbuchstaben des englischen Alphabets oder \_
- (b) optional: weitere Zeichen: wie erstes Zeichen oder Ziffern 0 ... 9

7. vordefinierte Funktionen in C++

- (a) eingebaute Funktionen (immer vorhanden) z.B. Infix Operatoren
- (b) Funktionen der Standardbibliothek (Programmierer muss sie explizit auffordern)
  - i. z.B. algebraische Funktionen beginnend mit std::...
  - ii. sind in Module geordnet, z.B. `cmath`  $\hat{=}$  algebraische Funktionen, `iostream`  $\hat{=}$  Ausgabe, z.B. `std::cout`
  - iii. Um ein Modul zu benutzen, muss man zuerst (am Anfang des Programms) sein Inhaltsverzeichnis importieren  
`#include <module_name>` sprich “Header inkludieren“

```
# include <iostream>
# include <string>

int main() {

    std::cout << "Hello" << "\n";
    std::string >> ausgabe = "mein erstes Programm"
    std::cout << ausgabe;

    return 0
}
```

```
int a = 3;
int b = 4;
int c = a * b;
double x = 3.0;
double y = 4.0;
double z = x * y;
```

$3.0 * 4 \Rightarrow$  automatische Umwandlung in höheren Typ, hier: "double"  $\Rightarrow$  wird als  $3.0 * 4.0$  ausgeführt

### Integer-Division in C++ Konsequenzen:

1. Division unterscheidet sich nach dem Datentypen:  $(-12)/5 \Rightarrow -2 \neq -2.4 \Leftarrow (-12.0/5.0)$
2. negative Ereignisse werden aufgerund, positive abgerundet (truncating division)  
d.h. Nachkommastellen abschneiden, d.h. Richtung Null runden
3. Gegensatz (z.B. zu Python): floor division  $\hat{=}$  wird immer abgerundet
4. Divisionsrest:

```
int a = ...;
int b = ...;
int q = a/b;
(a/b)*b = q * b
```

ist im allgemeinen ungleich  $a \Rightarrow$

```
int rest = a - q*b;
```

1. wenn Division aufgeht  $\Rightarrow$  rest = 0 , sonst  $\neq 0$
2. Invariante:

$$(a/b) * b + \text{rest} = a$$

```
int rest1 = a % b; // äquivalent: a-(b/a)*b
```

**Anwendung** Wochentag für beliebiges Datum bestimmen: gegeben:  $d, m, y$  , gesucht:  $w \in \{0, \dots, b\}$   
`int weekday(int d, int w, int y)` ; `weekday(10,11,2016)  $\Rightarrow$  3 (Donnerstag)`  
Teilprobleme

1. finde den Wochentag vom 1. Januar y
2. finde den Abstand vom (d,m,y) zum (1,1,y)
3. setze beides zusammen

Schaltjahresregel: y ist Schaltjahr, wenn:

1. y durch 4 teilbar, aber nicht durch 100  $\Rightarrow$  2004, 2006, nicht 2100
2. y durch 400 teilbar  $\Rightarrow$  2000

$\Rightarrow$  400-Jahres-Zyklus der Regeln: nach 400 Jahren beginnt die Schaltjahresregel von vorn

- Beobachtung: der 1.1.2001 ist der erste Tag eines neuen Zyklus und war Montag
- die Anzahl der Tage vom 1.1.y zum 1.1.2001 ist:  

$$z = y - 2001 \quad \Delta = 365 * z + z/4 - z/100 + z/400$$
- floor division ist wichtig, wenn  $z < 0$ , z.B.  $y = 2000, z = -1$

zu②: d.m. ist der x-te Tag im Jahr mit:

- kein Schaltjahr
  1.  $m = 1 \Rightarrow d$
  2.  $m = 2 \Rightarrow d + 31$
  3.  $m = 3 \Rightarrow d + 59$
  4.  $m = 4 \Rightarrow d + 90$
  5.  $m = 5 \Rightarrow d + 120$
  6.  $m > 2 \Rightarrow d + 59 + (153 * m - 457)/5$
- Schaltjahr
  1.  $m = 1 \Rightarrow d$
  2.  $m = 2 \Rightarrow d + 31$
  3.  $m = 3 \Rightarrow d + 60$
  4.  $m = 4 \Rightarrow d + 91$
  5.  $m = 5 \Rightarrow d + 121$
  6.  $m > 2 \Rightarrow d + 60 + (153 * m - 457)/5$

zu③: Wochentag von d, m, y:

$$w = (w\_11y + x - 1) \bmod 7$$

## Bedingungen

- Bei den meisten Algorithmen ist die Reihenfolge der Schritte nicht fix, sondern hängt von den Eingabedaten ab
- Beispiel: Auswahl der Offset  $d \rightarrow x$  hängt von m ab  
 dafür die Funktion:

cond ( bedingung , resultat\_wenn\_wahr , resultat\_we

- kanonische Beispiele: Absolutbetrag, Vorzeichenfunktion

Bedingungen programmieren:

- relationale Operatoren: Vergleich von zwei Argumenten  
 $<, >, <=, >=, !=$

- logische Operatoren: Verknüpfen von mehreren Bedingungen  
 $\&\&(und), ||(oder), != (nicht)$
- in  $C++$  gibt es keine Prefix-Variante für die  $cond()$ -Funktion, aber eine Infix-Variante:

```
( bedingung ) ? erg_wenn_wahr : erg_wenn_falsch

int abs (int x) {
    return (x >= 0) ? x : -x;
}
double abs (double x) {
    return (x >= 0.0) ? x : -x;
}
int sign (int x) {
    return (x == 0) ? 0 : ((x > 0) ? 1 : -1);
}
```

**Rekursion** bedeutet: eine Funktion ruft sich selbst auf (evtl. indirekt)

- kanonisches Beispiel: Fakultätsfunktion  $k! = 1 \cdot 2 \cdot \dots \cdot (k-1) \cdot k$
- in  $C++$  (rekursive Definition)

```
int fakultaet (int k) {
    return (k == 0) ? 1 : k * fakultaet(k-1) ;
}
```

- wichtige Eigenschaften:
  - jede rekursive Funktion muss mindestens einen nicht-rekursiven Zweig enthalten, der nach endlich vielen rekursiven Aufrufen erreicht wird “Rekursionsabschluss“- sonst: Endlosrekursion (Absturz)
  - bei jedem Aufruf werden dem Namen der Dateenelemente (Argumente & Zwischenergebnisse) neue Speicherzellen zugeordnet  
 $fakultaet(3) \rightarrow fakultaet(2) \rightarrow fakultaet(1) \rightarrow fakultaet(0) \Rightarrow$   
 $return\ 3*fakultaet(2) \leftarrow return\ 2*fakultaet(1) \leftarrow return\ 1*fakultaet(0) \leftarrow return\ 1$