2. Ganze Zahlen

Auswertung arithmetischer Ausdrücke, Assoziativität und Präzedenz, arithmetische Operatoren, Wertebereich der Typen **`int`**, **`unsigned int`**

Celsius to Fahrenheit

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

int main()
{
  // Input
  std::cout << "Temperature in degrees Celsius =? ";
  int celsius;
  std::cin >> celsius;

  // Computation and output
  std::cout << celsius << " degrees Celsius are "
            <<  9 * celsius / 5 + 32  << " degrees Fahrenheit.\n";
  return 0;
}
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

`9 * celsius / 5 + 32`

- Arithmetischer Ausdruck,
- enthält drei Literale, eine Variable, drei Operatorsymbole

Wie ist der Ausdruck geklammert?

Präzedenz

Punkt vor Strichrechnung

```
9 * celsius / 5 + 32
```

bedeutet

```
(9 * celsius / 5) + 32
```

Regel 1: Präzedenz

Multiplikative Operatoren (`*`, `/`, `%`) haben höhere Präzedenz ("binden stärker") als additive Operatoren (`+`, `-`)

Assoziativität

Von links nach rechts

```
9 * celsius / 5 + 32
```

bedeutet

```
((9 * celsius) / 5) + 32
```

Regel 2: Assoziativität

Arithmetische Operatoren (`*`, `/`, `%`, `+`, `-`) sind linksassoziativ: bei gleicher Präzedenz erfolgt Auswertung von links nach rechts

Stelligkeit

Regel 3: Stelligkeit

Unäre Operatoren `+`, `-` vor binären `+`, `-`.

```
-3 - 4
```

bedeutet

```
(-3) - 4
```

Klammerung

Jeder Ausdruck kann mit Hilfe der

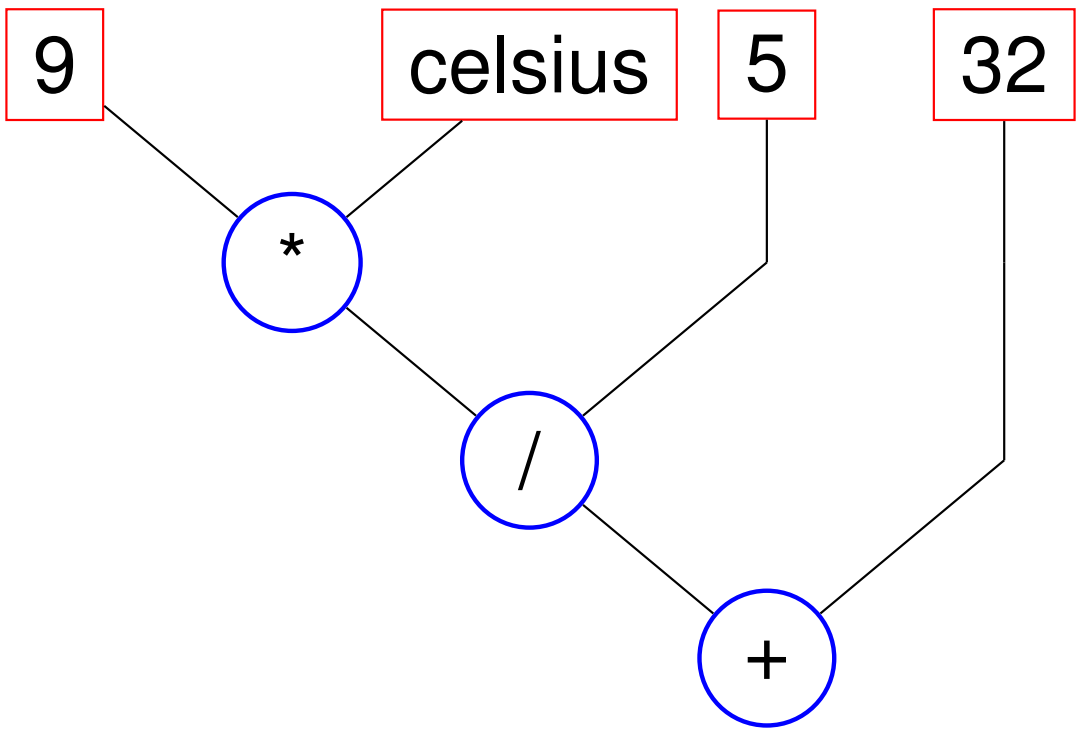
- Assoziativitäten
- Präzedenzen
- Stelligkeiten (Anzahl Operanden)

der beteiligten Operatoren eindeutig geklammert werden (Details im Skript).

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

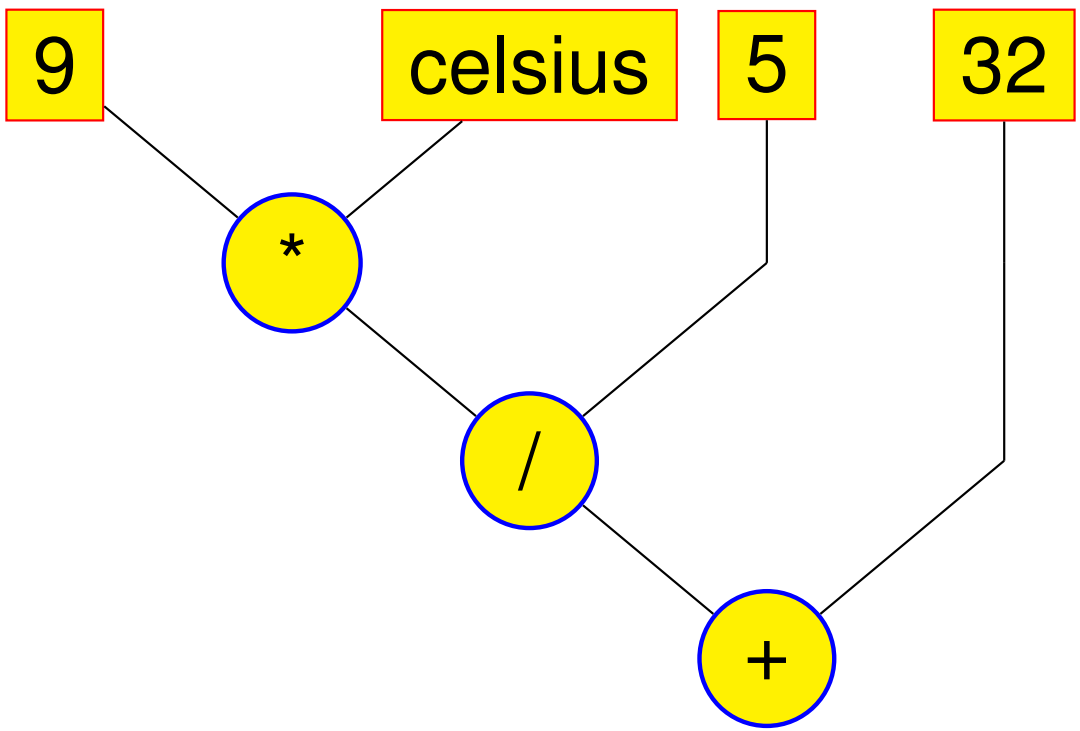
```
(((9 * celsius) / 5) + 32)
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

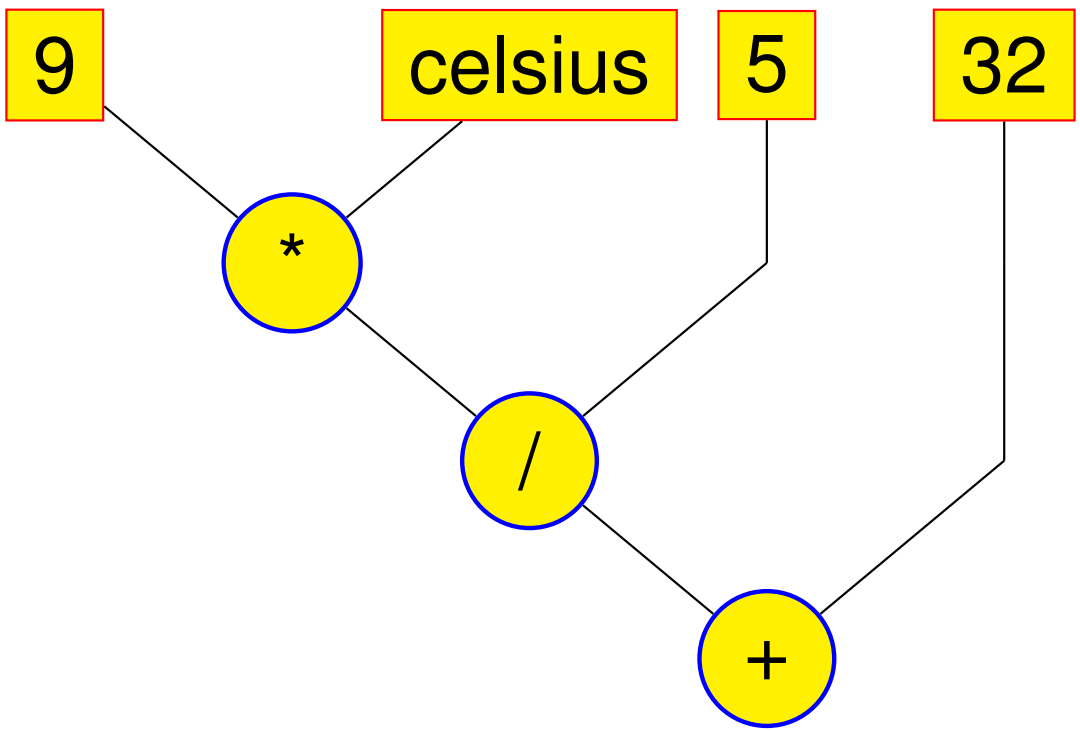
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

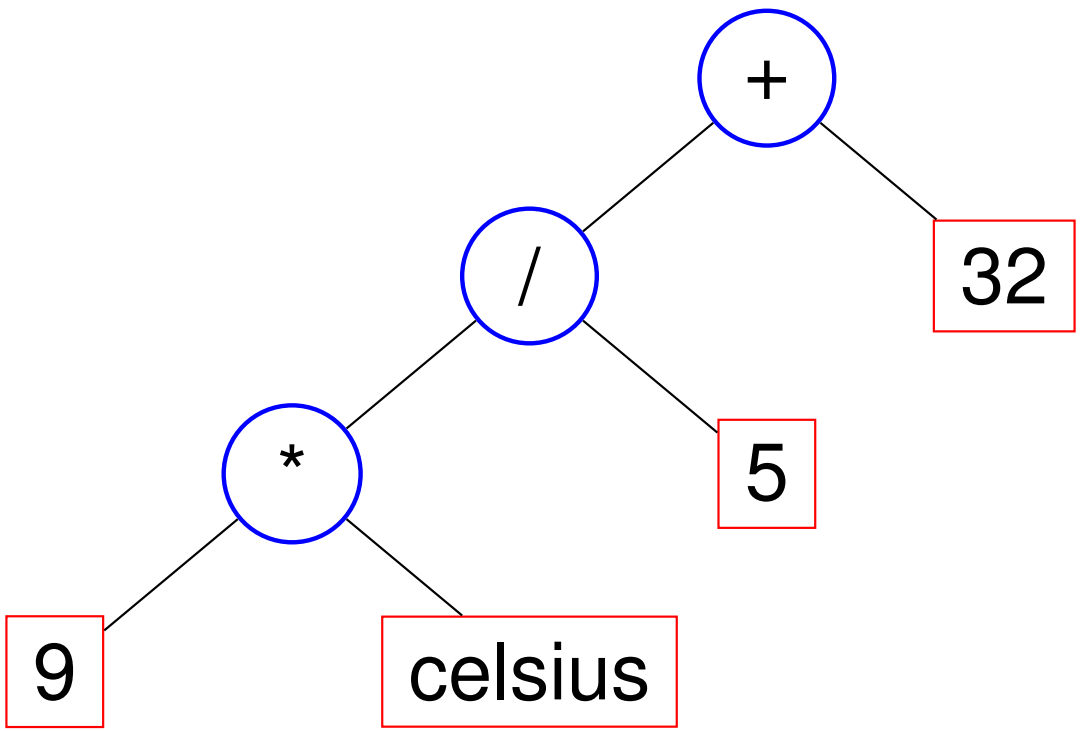
```
9 * celsius / 5 + 32
```



Ausdrucksbäume – Notation

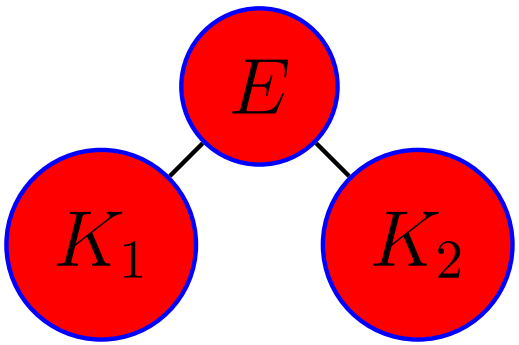
Üblichere Notation: Wurzel oben

```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge – formaler

- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



In C++ist die anzuwendende gültige Reihenfolge nicht spezifiziert.

- "Guter Ausdruck": jede gültige Reihenfolge führt zum gleichen Ergebnis.
- Beispiel für "schlechten Ausdruck":
  `(a=b)*(b=7)`

Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	`+`	1	16	rechts
Negation	`-`	1	16	rechts
Multiplikation	`*`	2	14	links
Division	`/`	2	14	links
Modulus	`%`	2	14	links
Addition	`+`	2	13	links
Subtraktion	`-`	2	13	links

Alle Operatoren: [R-Wert $\times$] R-Wert $\rightarrow$ R-Wert

Zuweisungsausdruck – nun genauer

- Bereits bekannt: `a = b` bedeutet Zuweisung von `b` (R-Wert) an `a` (L-Wert). Rückgabe: L-Wert
- Was bedeutet `a = b = c` ?
- Antwort: Zuweisung rechtsassoziativ, also

`a = b = c` $\Longleftrightarrow$ `a = (b = c)`

Beispiel Mehrfachzuweisung:

`a = b = 0` $\Longrightarrow$ `b=0; a=0`

Division und Modulus

- Operator `/` realisiert ganzzahlige Division

`5 / 2` hat Wert 2

- In **`fahrenheit.cpp`**

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent. . . aber nicht in C++!

```
9 / 5 * celsius + 32

15 degrees Celsius are  47 degrees Fahrenheit
```

Division und Modulus

- Modulus-Operator berechnet Rest der ganzzahligen Division

`5 / 2` hat Wert 2, `5 % 2` hat Wert 1.

- Es gilt immer:

`(a / b) * b + a % b` hat den Wert von `a`.

Inkrement und Dekrement

- Erhöhen / Erniedrigen einer Zahl um 1 ist eine häufige Operation
- geht für einen L-Wert so:

```
expr = expr + 1.
```

Nachteile

- relativ lang
- expr wird zweimal ausgewertet (Effekte!)

In-/Dekrement Operatoren

Post-Inkrement

```
expr++
```

Wert von expr wird um 1 erhöht, der *alte* Wert von expr wird (als R-Wert) zurückgegeben

Prä-Inkrement

```
++expr
```

Wert von expr wird um 1 erhöht, der *neue* Wert von expr wird (als L-Wert) zurückgegeben

Post-Dekrement

```
expr--
```

Wert von expr wird um 1 verringert, der *alte* Wert von expr wird (als R-Wert) zurückgegeben

Prä-Dekrement

```
--expr
```

Wert von expr wird um 1 verringert, der *neue* Wert von expr wird (als L-Wert) zurückgegeben

In-/Dekrement Operatoren

	Gebrauch	Stelligkeit	Präz	Assoz.	L/R-Werte
Post-Inkrement	`expr++`	1	17	links	L-Wert $\rightarrow$ R-Wert
Prä-Inkrement	`++expr`	1	16	rechts	L-Wert $\rightarrow$ L-Wert
Post-Dekrement	`expr--`	1	17	links	L-Wert $\rightarrow$ R-Wert
Prä-Dekrement	`--expr`	1	16	rechts	L-Wert $\rightarrow$ L-Wert

In-/Dekrement Operatoren

Beispiel

```
int a = 7;
std::cout << ++a << "\n"; // 8
std::cout << a++ << "\n"; // 8
std::cout << a << "\n"; // 9
```

In-/Dekrement Operatoren

Ist die Anweisung

```
++expr;
```
← wir bevorzugen dies

äquivalent zu

```
expr++;
```
?

Ja, aber

- Prä-Inkrement ist effizienter (alter Wert muss nicht gespeichert werden)
- Post-In/Dekrement sind die einzigen linksassoziativen unären Operatoren (nicht sehr intuitiv)

C++ **vs.** ++C

Eigentlich sollte unsere Sprache ++C heissen, denn

- sie ist eine Weiterentwicklung der Sprache C,
- während C++ ja immer noch das alte C liefert.

Arithmetische Zuweisungen

```
a += b
```

$\Leftrightarrow$

```
a = a + b
```

Analog für `-`, `*`, `/` und `%`

Arithmetische Zuweisungen

	Gebrauch	Bedeutung
`+=`	`expr1 += expr2`	`expr1 = expr1 + expr2`
`-=`	`expr1 -= expr2`	`expr1 = expr1 - expr2`
`*=`	`expr1 *= expr2`	`expr1 = expr1 * expr2`
`/=`	`expr1 /= expr2`	`expr1 = expr1 / expr2`
`%=`	`expr1 %= expr2`	`expr1 = expr1 % expr2`

Arithmetische Zuweisungen werten expr1 nur einmal aus.

Zuweisungen haben Präzedenz 4 und sind rechtsassoziativ

Binäre Zahlendarstellungen

Binäre Darstellung ("Bits" aus $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

entspricht der Zahl $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

Beispiel: **`101011`** entspricht **`43`**.

Niedrigstes Bit, Least Significant Bit (LSB)

Höchstes Bit, Most Significant Bit (MSB)

Binäre Zahlen: Zahlen der Computer?

Wahrheit: Computer rechnen mit Binärzahlen.

NEUE ZÜRCHER ZEITUNG

TECHNIK

Mittwoch, 30. August 1950 Blatt 15
Mittagausgabe Nr. 1796 (50)

Das programmgesteuerte Rechengerät an der Eidgenössischen Technischen Hochschule in Zürich

Die Entwicklung programmgesteuerter Rechenmaschinen in den Vereinigten Staaten von Amerika wurde in den Artikeln „Elektronische Rechenmaschinen" (vgl. Nr. 2140 der „N. Z. Z." vom 13. Oktober 1948) und „Die neueste elektronische Rechenmaschine" (vgl. Nr. 871 der „N. Z. Z." vom 26. April 1950) behandelt. Nachstehend soll von einem Gerät deutscher Herkunft — Zuse K.-G., Neukirchen — die Rede sein, welches im Juli dieses Jahres am Institut für angewandte Mathematik der Eidgenössischen Technischen Hochschule in Zürich, das unter der Leitung von Prof. Dr. E. Stiefel steht, in Betrieb genommen wurde. Damit ist dieses Institut in der Lage, dem in der Schweiz immer stärker werdenden Bedürfnis nach einer leistungsfähigen Zentralstelle für numerische Rechnungen wenigstens teilweise gerecht zu werden. Bereits sind einige mathematische Probleme behandelt worden, und die Erledigung vieler anderer Aufgaben ist vorbereitet.

Merkmale des Gerätes

Das Gerät ist ein Glied in dem längeren Entwicklungsprogramm des Ingenieurs *Konrad Zuse;* es wurde im Auftrag des *Institutes für angewandte Mathematik* der E. T. H. unter Berücksichtigung von dessen Wünschen und Ideen von Zuse als *„Modell Z 4"* konstruiert. Die ursprüngliche Entwicklung in Deutschland erfolgte in den Kriegsjahren und verlief völlig unabhängig von den Untersuchungen in den Vereinigten Staaten. Es ist überaus interessant festzustellen, wie für die meisten wichtigen funktionellen Probleme beiderorts genau dieselbe Lösung gefunden wurde, wie aber anderseits gewissen Fragen sekundärer Wichtigkeit eine ganz unterschiedliche Bedeutung beigemessen wurde.

Eine kurze technische Charakterisierung lautet wie folgt: Elektromechanisch arbeitendes Gerät mit 2200 Relais, 21 Schrittschaltern und einem Speicher für 64 Zahlen, welcher mit neuartigen, mechanischen Schaltgliedern arbeitet; Verwendung des Dualsystems und der halblogarithmischen Darstellung; Multiplikationszeit 2,5 Sekunden; Programmsteuerung mit Hilfe zweier Lochstreifen, auf die wahlweise umgeschaltet werden kann; Eingabe von Zahlen durch eine Tastatur oder durch einen Lochstreifen; Abgabe der Resultate durch Lampenfeld, Lochstreifen oder Druckwerk.

Das duale Zahlsystem

Allgemein wird programmgesteuerten Rechengeräten häufig das duale Zahlsystem zugrunde gelegt, welches nur die zwei Zahlsymbole 0 und 1 verwendet, während das bekannte Dezimalsystem

Lesen wir eine Dezimalzahl von rechts nach links, so erhöht sich das Gewicht von Stelle zu Stelle um den Faktor 10. Im *Dualsystem* ist nun einfach dieser Faktor 10 durch 2 zu ersetzen. Also bedeutet die (nunmehr duale) Zahl abcd,efg den Ausdruck:

$$a \cdot 2^3 + b \cdot 2^2 + c \cdot 2^1 + d \cdot 2^0 + e \cdot 2^{-1} + f \cdot 2^{-2} + g \cdot 2^{-3}$$

Die Zahl 1 wird in beiden Systemen gleich dargestellt. Um jedoch duale von dezimalen Zahlen deutlich zu trennen, schreiben wir die duale 1 als L. — Dagegen weicht schon die 2 ab, indem sie dual L0 lautet; denn dies bedeutet ja $1 \cdot 2^1 + 0 \cdot 2^0 = 2$. Wenn einer Zahl (ohne Stellen nach dem Komma) rechts eine Null zugefügt wird, so vergrößert sie sich um den Faktor 2 (und nicht, wie im Dezimalsystem, um den Faktor 10). Auf diese Weise kann aus L0 = 2 auf einfachste Weise gebildet werden: L00 = 4, L000 = 8, L0000 = 16, usw.

Die Dualzahl L0L0L bedeutet nun also:

$$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21$$

Ganz analog sind etwaige Stellen nach dem Komma zu interpretieren; so wird L, 0LL wie folgt übersetzt:

$$1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + 1/4 + 1/8 = 1{,}375$$

Der große Vorteil, der das Dualsystem für Rechenautomaten so geeignet macht, nämlich die Reduktion der Anzahl der verwendeten Symbole auf nur zwei, wird allerdings durch einen Nachteil erkauft: Es braucht mehr Stellen, um eine bestimmte Zahl darzustellen. Die einstellige Zahl 8

Aenderung des Maßstabes durchgerechnet werden können.

Die beschriebene Darstellung bringt eine gewisse Komplikation der Rechenoperationen mit sich. So müssen vor einer Addition die beiden Summanden zunächst so verschoben werden, daß ihre Kommata untereinander zu liegen kommen, was an Hand eines Beispiels erläutert werden soll. Damit der Leser nicht durch das ungewohnte duale Zahlsystem verwirrt wird, ist das Beispiel im Dezimalsystem durchgeführt; doch wird daran erinnert, daß das Gerät in Wirklichkeit mit dualen Zahlen rechnet.

Es soll also etwa addiert werden: $2{,}345678 \times 10^2 + 9{,}876543 \times 10^{-1}$ (Man beachte, daß die eigentliche Zahl stets zwischen 1 und 10 liegt, also das Komma nach der ersten Stelle hat.) Nun müssen die beiden Summanden „ausgerichtet" werden, d. h. die beiden Exponenten sind einander gleich zu machen, und zwar erhält der kleinere Exponent den Wert des größeren, also 2. Die Zahlen lauten nun, richtig untereinander geschrieben und addiert, wie folgt:

$$\begin{array}{r} 2{,}345678 \times 10^2 \\ 0{,}009876 \times 10^2 \\ \hline 2{,}355554 \times 10^2 \end{array}$$

Es ist ersichtlich, daß bei der kleineren der beiden Zahlen rechts einige Stellen abgeschnitten werden mußten; denn wenn die Summanden siebenstellig gegeben waren, so soll auch das Resultat nicht mehr als sieben Stellen enthalten.

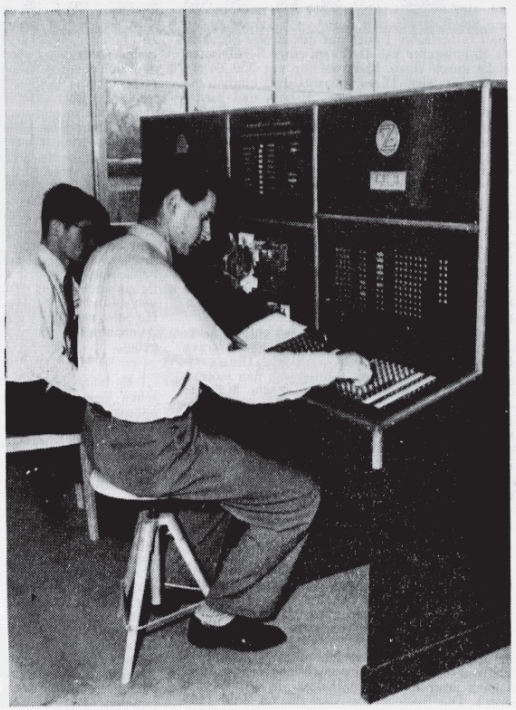


Abb. 2. Der Schaltpult bei der Fertigung eines Rechenplanes. Die Abtaster für den Lochstreifen sind deutlich sichtbar.

Befehle können „bedingt" gegeben werden, d. h. ihre Ausführung wird von der Natur eines errechneten Resultates abhängig gemacht. Erst dadurch werden die außerordentlich weiten Perspek-

Binäre Zahlen: Zahlen der Computer?

Klischee: Computer reden 0/1-Kauderwelsch.

01001110 01011010 01011010

Freitag, 8. Juni 2012 · Nr. 131 · 233. Jhg. 01001010 01010110 01001101 www.nzz.ch · Fr. 4.00 · €3.50

01000110 01101100 11111100

01000010 01100101
01110010 01101001

01100011 01101000 01110100 01100101

00100000 11111100 01100010
01100101 01110010 00100000
01101110 01100101 01110101 011-
00101 01110011 00100000 010-
01101 01100001 01110011

01110011 01100001

00100000 01110110

01100110 01100101

Hexadezimale Zahlen

Zahlen zur Basis 16. Darstellung

$$h_n h_{n-1} \ldots h_1 h_0$$

entspricht der Zahl

$$h_n \cdot 16^n + \cdots + h_1 \cdot 16 + h_0.$$

Schreibweise in C++: vorangestelltes **`0x`**

Beispiel: **`0xff`** entspricht **`255`**.

Hex Nibbles

hex	bin	dec
0	0000	0
1	**0001**	**1**
2	**0010**	**2**
3	0011	3
4	**0100**	**4**
5	0101	5
6	0110	6
7	0111	7
8	**1000**	**8**
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits. Die Zahlen 1, 2, 4 und 8 repräsentieren Bits 0, 1, 2 und 3.
- „Kompakte Darstellung von Binärzahlen”.

32-bit Zahlen bestehen aus acht Hex-Nibbles:
`0x00000000 -- 0xffffffff` .
`0x400` $= 1Ki$.
`0x100000` $= 1Mi$.
`0x40000000` $= 1Gi$.
`0x80000000`: höchstes Bit einer 32-bit Zahl gesetzt.
`0xffffffff`: alle Bits einer 32-bit Zahl gesetzt.
„`0x8a20aaf0` ist eine Addresse in den oberen 2G des 32-bit Addressraumes”

Beispiel: Hex-Farben

#00FF00

r g b

Wozu Hexadezimalzahlen?

"Für Programmierer und Techniker" (Auszug aus der Bedienungsanleitung des Schachcomputers *Mephisto II*, 1981)

Beispiele:

8200

a) Anzeige ***8200***
MEPHISTO ist mit genau 2 Bauern-Einheiten im Vorteil.

7F00

b) Anzeige ***7F00***
MEPHISTO ist mit genau 1 Bauern-Einheit im Nachteil.

Die Anzeige erfolgt in ***hexadezimaler Schreibweise.*** Im Gegensatz zum gewohnten Dezimalsystem gehen die Ziffern an jeder Stelle von 0 bis F (A = 10, B = 11, ... F = 15).
Für mathematisch Vorgebildete nachstehend die Umrechnungsformel in das dezimale Punktsystem:

$$ABCD = (Ax16^3) + (Bx16^2) + (Cx16^1) + (Dx16^0)$$

Für A gilt: 7 = -1; 8 = 0; 9 = +1 usw.
Eine Bauerneinheit (B) wird ausgedrückt in $16^2 = 256$ Punkten.
Dieses auf den ersten Blick vielleicht etwas komplizierte System dient der Service-Freundlichkeit von MEPHISTO, sowie insbesondere der Entwicklungsarbeit an zukünftigen, noch stärkeren Programmen, ist also mehr für unsere Programmierer und Techniker vorgesehen.

Beispiele:

805E

c) Anzeige ***805E***
(E=14) Umrechnung nach folgendem Verfahren:
$(14x16^0) + (5x16^1) + (0x16^2) + (0x16^3) = 14+80+0+0 =$
$= +94$ ***Punkte.***

7F80

d) Anzeige ***7F80***
(7=-1; F=15) Umrechnung wie folgt:
$(0x16^0) + (8x16^1) + (15x16^2) - (1x16^3) = 0+128+3840-4096 =$

Wozu Hexadezimalzahlen?

Die NZZ hätte viel Platz sparen können...

01001110 01011010 01011010

Freitag, 8. Juni 2012 · Nr. 131 · 233. Jhg. — *01001010 01010110 01001101* — www.nzz.ch · Fr. 4.00 · €3.50

01000110 01101100 11111100

01100011 01101000 01110100 01101100 01101001 01101110 01100111 01110011 01100101 01101100 01100101 01101110 01100100 0010-0000 01101001 01101110 00100000 01010000 01100001 01110100 01110010 01100001 01110011 00001101 00001010 01000100 01101001 01100101 00100000 01100111 01110010 01101001 01100101 01100011 01101000 01101001 01110011 01100011

01101000

01000010 01100101 01110010 01101001

01100011 01101000 01110100 01100101

00100000 11111100 01100010 01100101 01110010 00100000 01101110 01100101 01110101 011-00101 01110011 00100000 010-01101 01100001 01110011

01110011 01100001

01101011 01100101 01110010 00100000 011-01001 01101110 00100000 01010011 01111-001 01110010 01101001 01100101 01101110 00001101 00001010 00001101 00001010 01010101 01101110 01101111 00101101 010-00010 01100101 01101111 01100010 01100001 01100011 01101000 01110100 01100101 011-10010 00100000 01110110 01101111 01101101 00100000 01010011 01100011 01101000 011-00001 01110101 01110000 01101100 01100-001 01110100 01111010 00100000

01100110 01100101

01110010 01101110 01100111 01100101 011-01000 01100001 01101100 01110100 0110-0101 01101110 00001101 00001010 00001101 00001010 01001100 01100001 01110101 011-10100 00100000 01000010 01100101 0111-00110 01101001 01100011 01101000 01110100

01100101 01110011 00100000 01001101 011-00001 01110011 01110011 01100001 01101-011 01100101 01110010 00100000 01110011 01110100 01100001 01110100 01110100 011-00111 01100101 01100110 01101101 01101110 01100100 01100101 01101110 00101110 001-00000 01000100 01101001 01100101 001-00000 01010010 01100101 01100111 01101001 01100101 01110010 01110101 01101110 01100111 00100000 01101101 01100001 011-00011 01101000 01110100 01100101 00100-000 01110101 01101110 01100111 01100101 01101110 01100001 01101110 01101110 0111-0100 01100101 00100000 10101011 01010100 01100101 01110010 01110010 01101111 011-10010 01101001 01110011 01110100 01100101 01101110 10111011 00100000 01100100 011-00001 01100110 11111100 01110010

00100000 01110110

01100101 01110010 01100001 01101110 0111-0100 01110111 01101111 01110010 01101100 01101100 01101001 01100011 01101000 001-01110 00001101 00001010 00001101 000-01010 01001010 11111100 01110010 011001-11 00100000 01000010 01101001 01110011 01100011 01101000 01101111 01100110 011-00110 00101100 00100000 01000010 01100-

Wertebereich des Typs `int`

- Repräsentation mit B Bits. Wertebereich umfasst die 2^B ganzen Zahlen:

$$\{-2^{B-1}, -2^{B-1}+1, \ldots, -1, 0, 1, \ldots, 2^{B-1}-2, 2^{B-1}-1\}$$

- Auf den meisten Plattformen $B = 32$
- Für den Typ `int` garantiert C++ $B \geq 16$
- Hintergrund: Abschnitt 2.2.8 (Binary Representation) im Skript

Wertebereich des Typs `int`

```
// Program: limits.cpp
// Output the smallest and the largest value of type int.

#include <iostream>
#include <limits>

int main()
{
  std::cout << "Minimum int value is "
            << std::numeric_limits<int>::min() << ".\n"
            << "Maximum int value is "
            << std::numeric_limits<int>::max() << ".\n";
  return 0;
}
```

Zum Beispiel

```
Minimum int value is -2147483648.
Maximum int value is 2147483647.
```

Überlauf und Unterlauf

- Arithmetische Operationen (`+`, `-`, `*`) können aus dem Wertebereich herausführen.
- Ergebnisse können inkorrekt sein.

`power8.cpp`: $15^8 = -1732076671$

`power20.cpp`: $3^{20} = -808182895$

- Es gibt *keine Fehlermeldung!*

Der Typ `unsigned int`

- Wertebereich

$$\{0, 1, \ldots, 2^B - 1\}$$

- Alle arithmetischen Operationen gibt es auch für **`unsigned int`**.
- Literale: **`1u`**, **`17u`** ...

Gemischte Ausdrücke

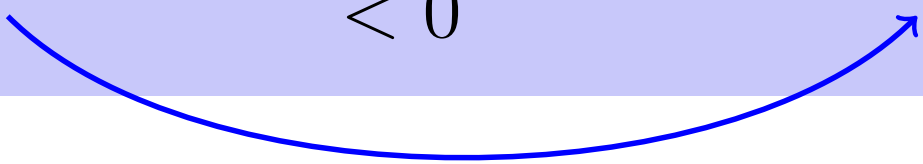
- Operatoren können Operanden verschiedener Typen haben (z.B. **`int`** und **`unsigned int`**).

```
17 + 17u
```

- Solche gemischten Ausdrücke sind vom „allgemeineren” Typ **`unsigned int`**.
- **`int`**-Operanden werden *konvertiert* nach **`unsigned int`**.

Konversion

`int` Wert	Vorzeichen	`unsigned int` Wert
x	≥ 0	x
x	< 0	$x + 2^B$



Bei Zweierkomplementdarstellung passiert dabei intern gar nichts

Konversion “andersherum”

Die Deklaration

```
int a = 3u;
```

konvertiert `3u` nach `int`.

Der Wert bleibt erhalten, weil er im Wertebereich von `int` liegt; andernfalls ist das Ergebnis implementierungsabhängig.