Algorithmen und Programmierung

6. Kapitel
Logik und Operatoren

Prof. Matthias Werner
Professur Betriebssysteme

6.1 Boolesche Algebra

Aussagelogik

- **Aussagelogik:** Teil der Logik, in dem Eigenschaften von Aussagen, die mittels Aussagenverknüpfungen aus anderen Aussagen entstehen, untersucht werden
- Begriff der Aussage wird **nicht** inhaltlich untersucht ➡ Prädikat
- Jede Aussage hat einen Wahrheitswert

Grundsätze

- Prinzip der **Zweiwertigkeit**:
  Jede Aussage hat entweder den Wert „wahr“ oder den Wert „falsch“
- Satz vom **ausgeschlossenen Dritten** (*tertium non datur*):
  Jede Aussage ist immer entweder wahr oder falsch
- Satz vom **ausgeschlossenen Widerspruch**:
  Keine Aussage ist zugleich wahr und falsch.

Aussagelogik (Forts.)

Prinzip der **Extensionalität**

Der Wahrheitswert einer Aussageverknüpfung hängt ausschließlich von den Wahrheitswerten ihrer Bestandteile ab.

- **Zuordnung:**
  - wahre Aussagen ➡ t (*true*)
  - falsche Aussagen ➡ f (*false*)

Es existieren auch mehrwertige Logiken:

- Zustand „unbestimmt“ (z.B. *open collector* in TTL)
- Zustandsänderungen als eigener Zustand
- Fuzzy-Logik

Boolesche Algebra

- GEORG BOOLE, 1815 - 1869
- Sei $\mathcal{B} = \{0, 1\}$
  - auch $\mathcal{B} = \{f, t\}$, $\mathcal{B} = \{\bot, \top\}$
  - ensprechen Wahrheitswerten
- Funktionen $f : \mathcal{B}^n \to \mathcal{B}^m$ mit $n, m \in \mathbb{N}, n, m \geq 1$ heißen **Boolesche Funktionen**
- Wenn $m = 1$: **echte Boolesche Funktion**

Boolesche Algebra (Forts.)

- Wichtige Verknüpfungsfunktionen ➡ $n = 1$

Bezeichnung	$z = f(0)$	$z = f(1)$	Notation	alternative Bezeichnung
Identität	0	1	$z = x$	
Negation	1	0	$z = \neg x = \bar{x}$	Komplement
Einsfunktion	1	1	$z = \mathbf{1}(x)$	
Nullfunktion	0	0	$z = \mathbf{0}(x)$	

Boolesche Algebra (Forts.)

- Wichtige Verknüpfungsfunktionen ➡ $n = 2$

Bezeichnung	$z = f(x,y)$ $x{=}0$ $y{=}0$	$x{=}0$ $y{=}1$	$x{=}1$ $y{=}0$	$x{=}1$ $y{=}1$	Notation	alternative Bezeichnung
Konjunktion	0	0	0	1	$z = x \wedge y = x \cdot y = xy$	Und
Disjunktion	0	1	1	1	$z = x \vee y$	Oder
Antivalenz	0	1	1	0	$z = x \oplus y$	Exklusiv-Oder
Äquivalenz	1	0	0	1	$z = x \equiv y = x \Leftrightarrow y$	Bisubjunktion
Implikation	1	1	0	1	$z = x \Rightarrow y$	Subjunktion

Andere Funktionen können zusammengesetzt werden

Interpretation der Implikation

$$x \Rightarrow y$$

x	y	$x \Rightarrow y$
falsch	*falsch*	*wahr*
falsch	*wahr*	wahr
wahr	*falsch*	*falsch*
wahr	*wahr*	*wahr*

Aussage: „Wenn das Wetter schön ist, gehe ich baden."

- Aus „Wetter schön" folgt „Baden"
- Jedoch ist **nicht** ausgeschlossen, dass auch bei schlechten Wetter gebadet wird
- Folglich wird die Aussage **nur falsch**, wenn bei schönen Wetter **nicht** gebadet wird

Logisch vollständige Mengen

- Gibt es eine Menge von Funktionen, mit denen alle anderen Funktionen dargestellt werden können?
- Ja, z.B.:
  - $\Omega = \{\vee, \neg\}$
  - $\Omega = \{\wedge, \neg\}$
  - $\Omega = \{\mathbf{1}, \oplus, \wedge\}$
  - $\Omega = \{\text{Nicht-Und}\}$
  - $\Omega = \{\text{Nicht-Oder}\}$
- Solche Mengen nennt man **logisch vollständig**

Boolesche Algebra

De Morgansche Regeln

- $\overline{x \wedge y} = \bar{x} \vee \bar{y}$
- $\overline{x \vee y} = \bar{x} \wedge \bar{y}$
- $x \Rightarrow y = \bar{y} \Rightarrow \bar{x}$

Kommutativgesetze

- $x \vee y = y \vee x$
- $x \wedge y = y \wedge x$
- $x \oplus y = y \oplus x$

Assoziativgesetze

- $(x \wedge y) \wedge z = x \wedge (y \wedge z) = x \wedge y \wedge z$
- $(x \vee y) \vee z = x \vee (y \vee z) = x \vee y \vee z$
- $(x \oplus y) \oplus z = x \oplus (y \oplus z) = x \oplus y \oplus z$

Distributivgesetze

- $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
- $x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z)$

Idempotenz

- $x \vee x = x$
- $x \wedge x = x$

Boolesche Algebra (Forts.)

Absorptionsregeln

- $x \vee \bar{x} = 1$
- $x \vee (x \wedge y) = x$
- $x \oplus x = 0$
- $x \wedge \bar{x} = 0$
- $x \wedge (x \vee y) = x$
- $x \oplus \bar{x} = 1$

Substitution von Konstanten

- $x \vee 0 = x$
- $x \wedge 0 = 0$
- $x \oplus 0 = x$
- $x \vee 1 = 1$
- $x \wedge 1 = x$
- $x \oplus 1 = \bar{x}$

Normalformen

- Boolesche Ausdrücke können sehr schnell unübersichtlich werden
- Abhilfe: Vereinheitlichung in Normalformen

Definition 6.1 (Disjunktive Normalform)

*Eine boolsche Funktion ist in **disjunktiver Normalform** (DNF), wenn sie eine Disjunktion (Oder) von Konjunktionsstermen (Und) ist, wobei die Konjunktionsterme nur (ggf. negierten) Funktionsparameter enthält.*

$$y = \bigvee_i \left(\bigwedge_j [\neg] x_{i,j} \right)$$

Definition 6.2 (Konjunktive Normalform)

*Eine boolsche Funktion ist in **konjunktiver Normalform** (KNF), wenn sie eine Konjunktion (Und) von Disjunktionstermen (Oder) ist, wobei die Disjunktionsterme nur (ggf. negierten) Funktionsparameter enthält.*

$$y = \bigwedge_i \left(\bigvee_j [\neg] x_{i,j} \right)$$

Normalformen (Forts.)

- Normalformen können auf verschiedene Arten erstellt werden
  - Umformen mit Hilfe der booleschen Algebra
  - Nutzung von Wahrheitstabellen

x_1	x_2	x_3	$f(x_1, x_2, x_3)$
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	1
1	0	1	1
0	1	1	0
1	1	1	1

DNF:

$$y = x_1\bar{x}_2\bar{x}_3 \vee x_1x_2\bar{x}_3 \vee \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2x_3 \vee x_1x_2x_3$$

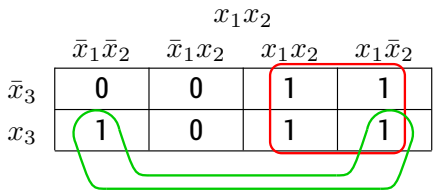
KNF:

$$y = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

Optimierung

- Mehrere Möglichkeiten, Ausdrücke zu redunzieren
- Beispiel **KV-Diagramm** (auch: Karnaugh-Veitch-Diagramm oder Karnaugh-Diagramm)

$$y = f(x_1, x_2, x_3) = x_1\bar{x}_2\bar{x}_3 \vee x_1x_2\bar{x}_3 \vee \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2x_3 \vee x_1x_2x_3$$



- $y = x_1 \vee \bar{x}_2x_3$
- Mehr in VL „Grundlagen der Technische Informatik“ oder „Einführung in die Funktion von Computern“

6.2 Logik in C

Typen

- C benutzt logische (boolsche) Ausdrücke und kennt auch einen entsprechenden Typ
- Da dieser Typ eng an Integer angelegt ist, wurde er vor C99 dem Programmierer nicht explizit zur Verfügung gestellt
- In C99 gibt es `_Bool`
- Durch Einbinden von `stdbool.h` steht Typ `bool` zur Verfügung
  - Werte: `false` und `true`
  - Bei C99 ist `bool` ein Synonym für `_Bool`

Merke:

Überall, wo ein Boolescher Typ verlangt wird, kann ein Integer stehen.

Dabei wird der Wert 0 als `false` interpretiert, jeder andere Wert als `true`.

Operatoren

- Wir haben boolesche Ausdrücke schon verwendet...
  - `if ( logischer Ausdruck ) ...`
  - `while ( logischer Ausdruck ) ...`
  - `do { ... } while ( logischer Ausdruck ) ...`
  - `for ( init, logischer Ausdruck, next ) ...`
- ... und Operatoren kennengelernt, die boolesche Werte liefern, z.B.
  - „`==`“ liefert `true`, wenn zwei Werte gleich sind
  - „`!=`“ liefert `true`, wenn zwei Werte ungleich sind
- Die folgende Tabelle listet alle in C eingebaute Operatoren, die einen booleschen Wert liefern:

Operatoren (Forts.)

Operator	Bezeichnung	Beispiel	Ergebnis
`!`	Negation	`!x`	$\bar{x}$
`<`	Kleiner-als	`x < y`	Wahr wenn x kleiner als y ist
`>`	Größer-als	`x > y`	Wahr wenn x größer als y ist
`<=`	Kleiner-gleich	`x <= y`	Wahr wenn x kleiner oder gleich y ist
`>=`	Größer-gleich	`x >= y`	Wahr wenn x größer oder gleich y ist
`==`	Gleich	`x == y`	Wahr wenn x und y den gleichen Wert haben
`!=`	Ungleich	`x != y`	Wahr wenn x und y nicht den gleichen Wert haben
`&&`	Und	`x && y`	Wahr wenn sowohl x als auch y wahr sind
`||`	Oder	`x || y`	Wahr wenn entweder x oder/und y wahr ist

Kurzschlussoperatoren

- Im Allgemeinen ist in C die Auswertungsreihenfolge in einem C-Ausdruck **nicht** festgelegt
- Die Logik-Operatoren `&&` und `||` bilden da eine Ausnahme

Kurzschlussoperatoren

Die Operatoren `&&` und `||` werden stets von links nach rechts ausgewertet und **zwar nur, bis das Ergebnis feststeht** ➡ **Kurzschlussoperatoren**

- Es ist eine gängige Programmiertechnik, links im Kurzschlussoperator eine Bedingung abzusichern, die rechts zu einem Laufzeitfehler führen würde

```c
/* shortcut.c -- partial evaluation */
#include<stdio.h>

enum { arraysize=4 };
int z[arraysize] = { 2, 42, 0, 23 };
int r[arraysize];

int main()
{
  for (int i=0; i<arraysize; ++i) {
     (z[i] != 0) && (r[i] = 1000/z[i]);
     printf("%d. value: %d\n",i,r[i]);
  }
  return 0;
}
```

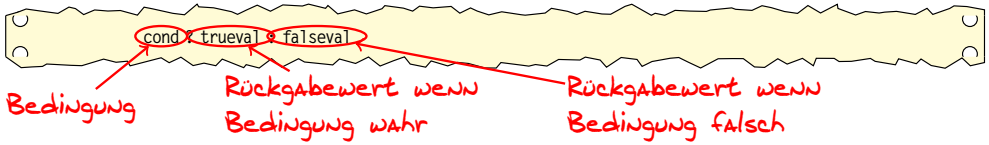

Bedingungsoperator

- Wir haben im Kapitel 2 bereits das `if`-Statement kennengelernt, das die Ausführung eines Programms in Abhängigkeit von einer Bedingung steuert
- Wenn dagegen ein **Wert** von einer Bedingung abhängig sein soll, kann der **Bedingungsoperator** genutzt werden
  - Entspricht dem funktionalen Programmiermodell

Bedingungsoperator

Der Bedingungsoperator „`?:`“ ist der einzige ternäre (dreistellige) Operator in C.

Er gibt in Abhängigkeit von einem logischen Ausdruck unterschiedliche Werte zurück.



Bedingungsoperator (Forts.)

- Die Funktion von „`shortcut.c`“ kann dann in etwa so erreicht werden:

```c
/* cond-op.c -- condition operator */
#include<stdio.h>
#include<limits.h>

enum { arraysize=4 };
int z[arraysize] = { 2, 42, 0, 23 };
float r[arraysize];

int main()
{
  for (int i=0; i<arraysize; ++i) {
    r[i] = (z[i] != 0) ?  1000/z[i] : INT_MAX;
    printf("%d. value: %d\n",i,r[i]);
  }
  return 0;
}
```

- Gegenüber der Lösung von Folie 17 kann hier im für den Fall der Division durch 0 ein Wert zugewiesen werden (z.B. `INT_MAX`, der in `limits.h` definiert ist)

Mehrfachfallunterscheidung

- Häufig wird bei der Fallunterscheidung (Schlüsselwort „`if`“) durch den Gleichheitsoperator („`==`“) eine Ganzahlvariable mit einem Literal verglichen
- Falls die Variable mehrere sinnvolle Werte annehmen kann, kommt es zum „Ketten-If“:

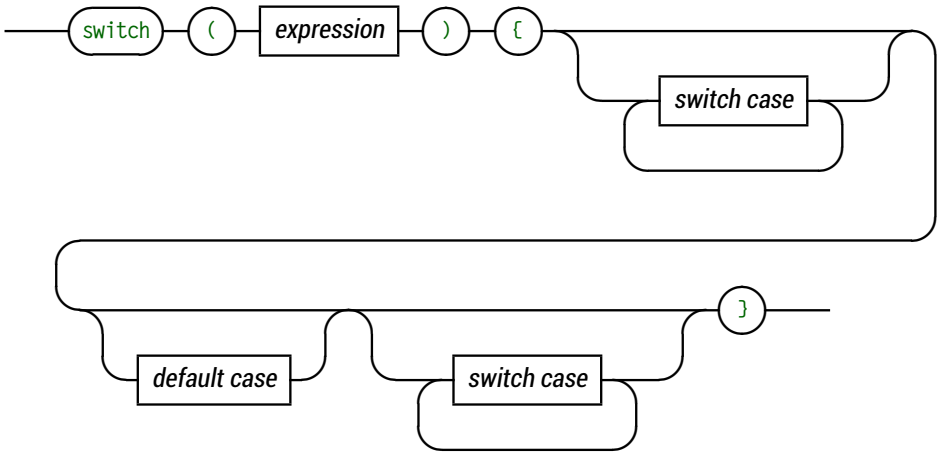
```c
  ...
if (x == 23 ) {
  ...
 } else if (x == 42 ) {
  ...
 } else if (x == 111 ) {
  ...
 } else {
  ...
 }
```

- Dies ist nicht gut lesbar
- Es ist nicht gut zu erkennen, zu welchen `if` ein `else` gehört

Mehrfachfallunterscheidung (Forts.)

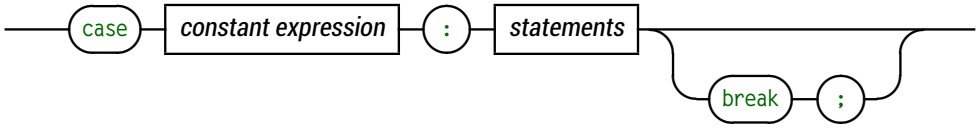
- Für derartige Fälle hat C ein eigenes Konstrukt: die **Mehrfachfallunterscheidung**

selection

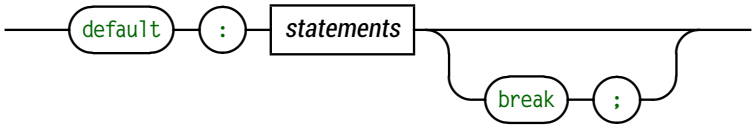


Mehrfachfallunterscheidung (Forts.)

switch case



default case



Mehrfachfallunterscheidung (Forts.)

```c
/* switch.c -- switch selection */
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char *argv[])
{
  if (argc !=2) return -1;
  switch (atoi(argv[1])) {
    default:
      printf("Keine besondere Zahl\n");
      break;
    case 42:
      printf("Antwort auf die Grosse Frage\n");
      break;
    case 23:
      printf("Zahl der Illuminaten\n");
      break;
  }
 return 0;
}
```

```
> ./switch 42
Antwort auf die Grosse Frage
> ./switch 5
Keine besondere Zahl.
```


Mehrfachfallunterscheidung (Forts.)

Anmerkungen

- Vom `switch`-Schlüsselwort wird zum jeweilig zutreffenden Fall gesprungen
- Von dort wird solange weitergegangen, bis
  - zum Ende eines Blocks – oder
  - zu einem `break` – oder
  - zu einem `return`
- Bei Blockende wird „normal“ weitergearbeitet
- Bei `break` wird hinter das Blockende gesprungen
- Bei `return` wird die Funktion (und damit auch der `switch`-Block) beendet

Achtung!

Ohne ein `break` oder `return` wird der nächste Fall abgearbeitet!

Falls dies tatsächlich beabsichtigt ist, sollte unbedingt ein Kommentar darauf hinweisen. In MISRA-C[1] ist das Auslassen eines `break` ganz verboten.

[1] Regelwerk der Motor Industry Software Reliability Association

Mehrfachfallunterscheidung (Forts.)

- Wenn kein Fall zutrifft, wird (so vorhanden) der `default`-Fall angesprungen
  - Der `default`-Fall kann weggelassen werden; dies wird aber **nicht empfohlen**
  - Es darf maximal **einen** `default`-Fall geben
  - Der `default`-Fall darf an beliebiger Stelle stehen; es ist jedoch günstig, ihn an den Anfang oder das Ende des `switch`-Blocks zu setzen

```c
// punctuation.c -- switch fall through
#include<stdio.h>
#include<stdbool.h>

bool is_punctuation_mark(char c){
  switch (c){
    case '.':   /* falls through */
    case '!':   /* falls through */
    case '?': return true;
    default : return false;
  }
}

int main(){
  char x;
  printf("Input character: ");
  scanf("%c",&x);
  if (is_punctuation_mark(x))
    printf("Punctation mark\n");
  else
    printf("Not a punctation mark\n");
  return 0;
}
```


Bit-Operatoren

- Eng verwandt mit booleschen Operatoren sind die sogenannten **Bit-Operatoren**
- Dabei wird eine logische Operation **auf alle Bits** einer Ganzzahl angewendet

Operator	Entspricht
`&`	bit-weises Und
`^`	bit-weise Antivalenz
`|`	bit-weises Oder
$\sim$	bit-weise Negation

Beispiel

Sei `x=42` und `y=23`. Dann gilt:

`x & y = 2`	`x ^ y = 61`	`x | y = 63`	$\sim$ `y = -24`
00101010	00101010	00101010	
$\wedge$ 00010111	$\oplus$ 00010111	$\vee$ 00010111	$\neg$ 00010111
00000010	00111101	00111111	11101000

Bit-Operatoren (Forts.)

- Es gibt noch andere Bit-Operatoren, die aber eher mit arithmetischen als mit logischen Operationen verwandt sind

Operator	Entspricht
`<<`	bit-weises Linksschieben
`>>`	bit-weises Rechtsschieben

- Beide Operatoren haben zwei Argumente:
  - links steht der Ausgangswert
  - rechts die Anzahl von zu verschiebenden Bits
- Der Wert der „Auffüllbits“ ist unterschiedlich
  - Bei „`<<`“ wird mit 0 aufgefüllt
  - Bei „`>>`“ wird mit dem Wert des linkesten Bits aufgefüllt

Beispiel

1	1	0	0	1	1	0	1	`>> 1 =`	1	1	1	0	0	1	1	0

Bit-Operatoren (Forts.)

Anmerkung

Linksschieben um n Bit entspricht (ohne Überlauf) einer Multiplikation mit 2^n.

Rechtsschieben um n Bit entspricht einer Division durch 2^n (ggf. mit Abrunden).

```c
/* shift.c -- demonstrates bit shift */
#include<stdio.h>

int main()
{
  signed short int x = -42;
  for (int i=0; i< (unsigned short)sizeof(short)*8; i=i+1)
    printf("%3hd << %2hd = %6hd\t%3hd >> %3hd = %6hd\n",
      x, i, x << i, x, i, x >> i);

  return 0;
}
```

Bit-Operatoren (Forts.)

```
> ./shift
-42 <<  0 =    -42        -42 >>  0 =    -42
-42 <<  1 =    -84        -42 >>  1 =    -21
-42 <<  2 =   -168        -42 >>  2 =    -11
-42 <<  3 =   -336        -42 >>  3 =     -6
-42 <<  4 =   -672        -42 >>  4 =     -3
-42 <<  5 =  -1344        -42 >>  5 =     -2
-42 <<  6 =  -2688        -42 >>  6 =     -1
-42 <<  7 =  -5376        -42 >>  7 =     -1
-42 <<  8 = -10752        -42 >>  8 =     -1
-42 <<  9 = -21504        -42 >>  9 =     -1
-42 << 10 =  22528        -42 >> 10 =     -1
-42 << 11 = -20480        -42 >> 11 =     -1
-42 << 12 =  24576        -42 >> 12 =     -1
-42 << 13 = -16384        -42 >> 13 =     -1
-42 << 14 = -32768        -42 >> 14 =     -1
-42 << 15 =      0        -42 >> 15 =     -1
```


Bit-Operatoren (Forts.)

Achtung!

Bit-Operatoren (Argumente + Ergebnis: Ganzahlen) dürfen **nicht** mit den logischen Operatoren (Argumente + Ergebnis: Wahrheitswerte) verwechselt werden!

```c
/* bits.c -- logical vs. bit opaerators  */
#include<stdio.h>

int main()
{
  int x=42,y=85;
  printf("First evaluation: ");
  if (x && y) printf("true.\n");
  else printf("false.\n");

  printf("Second evaluation: ");
  if (x & y) printf("true.\n");
  else printf("false.\n");

  return 0;
}
```

```
> ./bits
First evaluation: true.
Second evaluation: false.
```


Faulheitsoperatoren

- Da alle Ganzahlausdrücke als logische Ausdrücke aufgefasst werden können, können praktisch alle Operatoren in logischen Ausdrücken auftreten
- Mit den Bit-Operatoren haben wir alle grundsätzlichen Operatoren kennengelernt
- Jedoch gibt es noch drei Operatorenklassen, die Kurzschreibweisen für andere Konstrukte bieten (➡ **Faulheitsoperatoren** 😊)

- **Kommaoperator**: verbindet zwei Teilausdrücke
- **In-/Dekrementoperatoren**: Wertvergrößerung/-verkleinerung als Seiteneffekt
- **Selbstzuweisungsoperatoren**: Kurzschreibweisen für
  `var = var` *op* `arg`

Kommaoperator

- Der **Kommaoperator** „`,`“ lässt zwei Ausdrücke, die keinen Bezug zueinander haben (müssen), in **einem** Statement hintereinander stehen
- Der Wert eines Kommaoperatorausdrucks ist der Wert des zweiten Teilausdrucks
  - Der Wert des vorderen Teilausdrucks wird verworfen, nur evtl. Seiteneffekte sind relevant
- Der Kommaoperator garantiert (wie die Kurzschlussoperatoren) eine **Auswertungsreihenfolge** von links nach rechts
- Einsatz meist nur im Kopf von `for`-Schleifen
  - ➡ mehrere Initialisierungen oder Schleifenänderungen
- Der Kommaoperator hat eine **sehr** niedrige Priorität

Kommaoperator (Forts.)

```c
/* interval.c -- use of comma operator in for loops */
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
  if (argc != 3) return -1;

  for(int lower=atoi(argv[1]), upper=atoi(argv[2]);
      lower<=upper; lower=lower+1, upper=upper-1) {
     printf("[%d,%d] includes %d integer.\n",
            lower,upper,upper-lower+1);
  }
  return 0;
}
```


Inkrement- und Dekrementoperatoren

- Die Operatoren „++“ bzw. „--“ lassen sich auf alle **Skalartypen** (alle Ganzahltypen, Gleikommatypen, Zeiger) anwenden
- Operand **muss** ein L-Wert sein
- Operatoren **vergrößert** (++) bzw. **verkleinert** (--) den Operandenwert
  - um 1 bei Ganz- oder Gleitkommazahlen
  - um die Größe des Grundtyps bei Zeigern
- Die Operatoren gibt es als **Präfix-** und als **Postfixoperatoren**
  - Bei der Präfixvariante wird **erst** die Veränderung (Seiteneffekt) vorgenommen und **dann** der Wert des Gesamtausdrucks berechnet
  - Bei der Postfixvariante wird **erst** der Wert des Gesamtausdrucks berechnet und **dann** die Veränderung (Seiteneffekt) vorgenommen
- Operatoren werden häufig in Schleifen genutzt

Inkrement- und Dekrementoperatoren (Forts.)

- Unter Nutzung des Inkrement- und Dekrementoperators sieht das Intervallprogramm so aus:

```c
// interval2.c -- use of increment/decrement
#include<stdio.h>
#include<stdlib.h>

int main(int argc, char *argv[]){
  if (argc != 3) return -1;

  for(int lower=atoi(argv[1]),
          upper=atoi(argv[2]);
          lower<=upper; ++lower, --upper){
    printf("[%d,%d] includes %d integer.\n",
           lower,upper,upper-lower+1);
  }
  return 0;
}
```

Achtung!

Da i.d.R. die Auswertungsreihenfolge nicht festgelegt ist, müssen Seiteneffekte bei mehrmaligen Variablengebrauch in einem Ausdruck vermieden werden.

Inkrement- und Dekrementoperatoren (Forts.)

- Folgender Code demonstriert den Unterschied zwischen Prä- und Postfixoperator

```c
#include<stdio.h>

int main()
{
  int x=42;
  printf("A: %d\n",x);
  printf("B: %d\n",++x);
  printf("C: %d\n",x++);
  printf("D: %d\n",x);

  return 0;
}
```

```
> ./prevspost
A: 42
B: 43
C: 43
D: 44
```

- Kommt es **nur** auf den Seiteneffekt an, ist in C egal, was genutzt wird

Selbstzuweisungsoperatoren

- Häufig gibt es Konstrukte wie: `x = x * 2`
- In diesem Fall gibt es einen Zuweisungsoperator, der den gleichen Effekt hat:

```
x *= 2;
```

- Analog gibt es:

`+=`	`%=`	`|=`
`-=`	`&=`	`>>=`
`/=`	`^=`	`<<=`

- **Achtung:** Mitunter wird die Nutzung dieser Operatoren als unübersichtlich empfunden

Priorität und Assozivität

- Wenn mehrere Operatoren in einem Ausdruck benutzt werden, hängt die Auswertungsreihenfolge von der Priorität und der Assoziativität ab
  - Priorität: Operatoren mit höherer Priorität werden zuerst ausgewertet
  - Assoziativität: Bestimmt die Richtung der Auswertung

Prio.	Operator	Assoz.
hoch	`() [] -> .`	⇒
	`++ - ! ~ (type) sizeof +`[1] `-`[1] `*`[2] `&`[2]	⇐
	`* / %`	⇒
	`+ -`	⇒
	`« »`	⇒
	`< <= => >`	⇒
	`== !=`	⇒
	`&`	⇒
	`^`	⇒
	`|`	⇒
	`&&`	⇒
	`||`	⇒
	`?:`	⇐
	`= += -= *= /= %= &= ^= |= «= »=`	⇐
niedrig	`,`	⇒

[1] Vorzeichen [2] Adressoperatoren

Priorität und Assozivität (Forts.)

Achtung!

Der Operatorenvorrang kann zu unerwarteten Ergebnissen führen.

Im Zweifelsfall sollte immer geklammert werden.

```c
// rank.c --  operators' rank
#include<stdio.h>
#include<stdbool.h>

int main()
{
  int i;
  int j=0xFF42;
  bool b;
  b = 0x42 == j & 0x00FF;
  i= 23,42;
  printf("i=%d, b=%d\n",i,(int)b);
  return 0;
}
```

```
./rank
i=23, b=0
```


6.3 Automaten

- Betrachte folgende Aufgabenstellung:

> Ein Programm soll die Wörter eines Satzes in umgekehrter Reihenfolge ausgeben.
>
> Satzzeichen sollen in der Ausgabe nicht erscheinen.

- Satz liegt als Zeichenkette vor
- Erste Aufgabe: Erkennung von Wörtern
- Ansatz:
  - Wenn ein Buchstabe einem Leer- oder Satzzeichen **folgt**, beginnt hier ein Wort
  - ➡ es muss also bekannt sein, was **vorher** geschah

Satzinvertierung

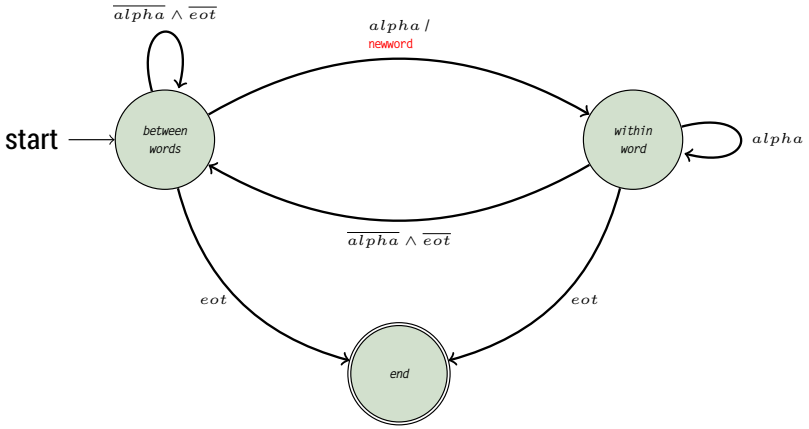
```c
/* reverse.c -- boolean states */
#include<stdio.h>
#include<stdbool.h> // true, false
#include<ctype.h>   // for: isalpha

enum{ maxwords=20};
char* words[maxwords];

int main(int argc, char* argv[]) {
  int j=0;
  _Bool skip_space;    /* continue as long as there are spaces */
  if (argc != 2) return -1; /* None or more than one argument */

  for(int i=0, skip_space=true; argv[1][i]!='\0'; ++i){
    if ((skip_space==true) && (isalpha(argv[1][i])==true)) {
       words[j++] = &argv[1][i];
       skip_space = false;
    } else if ((skip_space==false) && (isalpha(argv[1][i])==false)) {
       argv[1][i]='\0';
       skip_space = true;
    }
  }
                                        ⋮
```


Satzinvertierung (Forts.)

```c
                                        ⋮
  for (int i=j-1; i>=0; --i)
    printf("%s ",words[i]);
  printf("\n");
  return 0;
}
```

```
> ./reverse "all you need is: love."
love is need you all
```


Zustand



Definition

Definition 6.3 (Endlicher deterministischer Automat)

*Ein **endlicher (deterministischer) Automat** (EA, Zustandsmaschine, finite state machine ➡ FSM, auch: finite state automata) ist ein Modell zur Beschreibung von Abläufen (z.B. in Computern).*

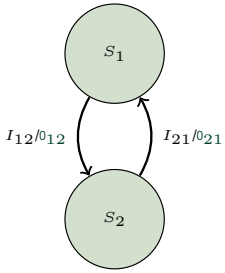
*Ein EA besteht aus einer Menge von **Zuständen** $\mathbf{S}$ (states) und **Zustandsübergängen** $\mathbf{T} : \mathbf{S} \times \Gamma \to \mathbf{S}$ (**Transitionen**, transitions).*

*Ein EA startet in einem **Startzustand**. Er „verarbeitet" eine Sequenz von **Zeichen** oder **Ereignissen** Γ. Dabei bestimmt das nächste Zeichen/Ereignis, in welchen Zustand der EA wechselt.*

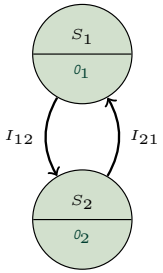
*Ein EA kann einen oder mehrere **Endzustände** (accepting states) besitzen. Wird ein solcher Zustand erreicht, ist die Abarbeitung beendet.*

Aktionen

- Zustandsänderungen in einem endlichen Automaten können zu **Aktionen** oder **Ausgaben** führen
- Zwei Varianten:
  - Aktion/Ausgabe ist an einen bestimmten Übergang gebunden (➡ **Mealy-Automat**)
  - Aktion/Ausgabe ist an die Ankunft in einem bestimmten Zustand gebunden (➡ **Moore-Automat**)



Mealy

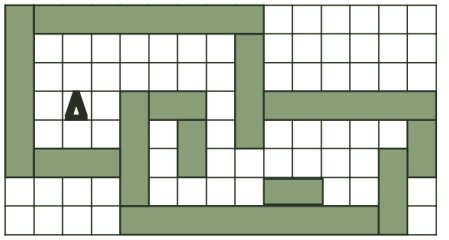


Moore

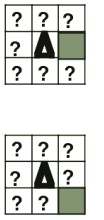
Programmentwurf mit Automaten

- Es ist häufig hilfreich, einen Ablauf zunächst als Automaten zu entwerfen
- Die Programmierung kann dann nach einem einheitlichen Muster erfolgen

Ein Roboter hat zwei Sensoren, die feststellen, ob unmittelbar vor ihm ($h = true$) oder unmittelbar rechts von ihm ($r = true$) ein Hindernis (Wand) ist. Er kann zwei Aktionen durchführen: Vorwärts gehen (F) und sich 90°nach rechts drehen (R).
Er hat die Aufgabe, in einem Labyrinth eine Wand zu finden und dann sich immer mit der rechten Seite an der Wand entlang bewegen.



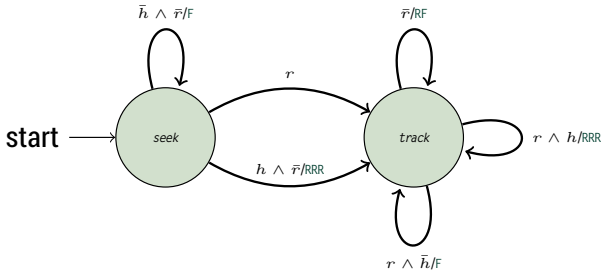
Aufgabe



$r = true$

Programmentwurf mit Automaten (Forts.)

- Automat:



Programmentwurf mit Automaten (Forts.)

```
    switch (state){
    case seek:
      if ((h==false) && (r=false))
        robot(forward);
      else if (r==true)
        state=track;
      else if ((h==true) && (r==false)) {
        robot(turn); robot(turn);  robot(turn);
      }
      break;
    case track:
      if (r==false){
        robot(turn);
        robot(forward);
      }
      else if ((r==true) && (h==true)){
        robot(turn); robot(turn); robot(turn);
      }
      else if ((r==true) && (h==false))
        robot(forward);
    }
```

Automaten und Grammatik

- Endliche Automaten sind eng verwand mit regulären Grammatiken (siehe Kapitel 5):

Äquivalenz von EA und regulären Grammatiken

Jede Folge von zulässigen Eingabesymbolen bzw. -ereignissen, die in einen Endzustand führen, entspricht einer formalen Sprache, die durch eine reguläre Grammatik beschrieben werden kann.

Oder anders formuliert:

Zu jeder regulären Grammatik existiert (mindestens) ein endlicher Automat, der genau alle Ausdrücke dieser Sprache akzeptiert.

Automaten und Grammatik (Forts.)

- Für kontextfreie Sprachen ist ein endlicher Automat **nicht** hinreichend

Beispiel

Allgemeine Klammerausdrücke (Regel: Klammern müssen in Paaren auftreten) sind nicht regulär und lassen sich nicht durch einen endlichen Automaten beschreiben.

- **Mögliche Abhilfen**:
  - Automat mit Kellerspeicher
  - Rekursiver Aufruf des Automaten

Aufgaben

Aufgabe 6.1

Beweisen(!) Sie[2], dass der Algorithmus von Euklid tatsächlich den größten gemeinsamen Teiler berechnet.

Aufgabe 6.2

Entwickeln Sie einen Automaten, der ausgibt (terminiert), ob eine römische Zahl korrekt ist.

Beispiele:

- XIX ➡ korrekt
- $XLIX$ ➡ korrekt
- $XVIX$ ➡ nicht korrekt
- IL ➡ nicht korrekt

[2] ... falls Sie dies nicht sowieso schon getan haben...

Aufgaben (Forts.)

Aufgabe 6.3

Elf große Schachteln stehen auf einem Tisch. Eine unbekannte Anzahl von Schachteln davon wird ausgewählt und in jede acht kleinere Schachtel platziert. Eine unbekannte Anzahl dieser kleineren Schachteln wird ausgewählt und in jede acht winzige Schachtel platziert.

Zum Schluss gibt es 102 leere Schachteln. Wie viele Schachteln gibt es insgesamt?