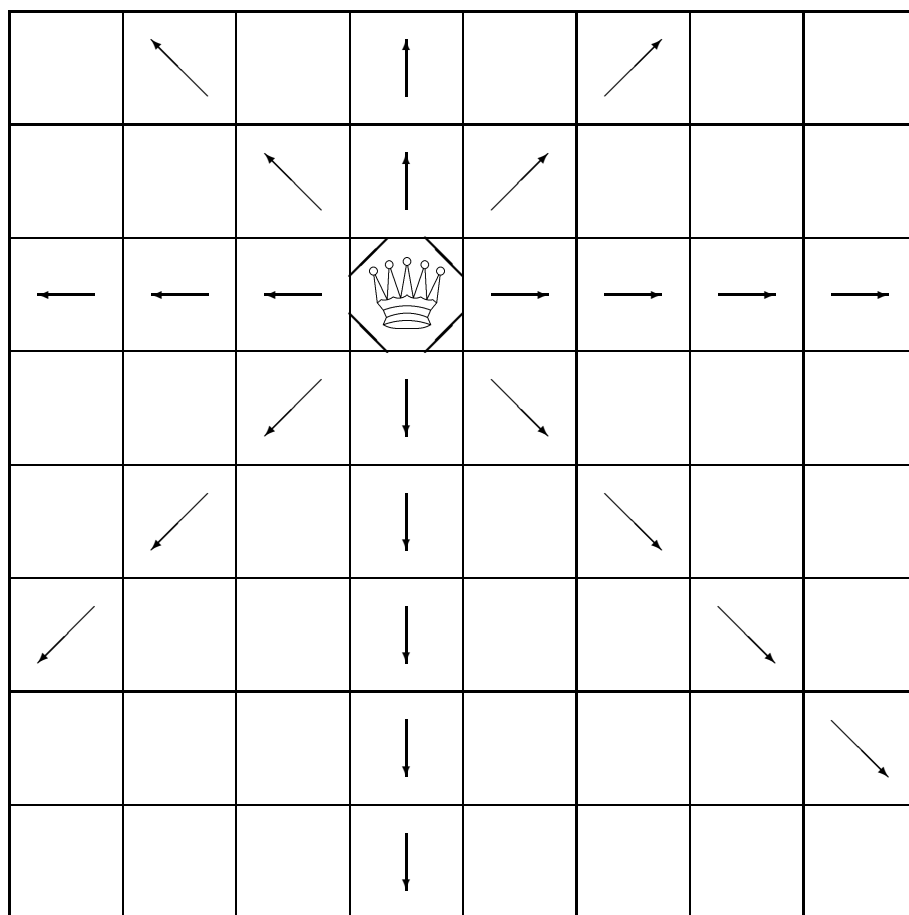


**8–Damen Problem:** Können 8 Damen so auf einem Schach–Brett aufgestellt werden, dass keine Dame eine andere Dame schlagen kann?

Eine Dame kann eine andere schlagen falls diese

- in der selben Reihe steht,
- in der selben Spalte steht, oder
- in der selben Diagonale steht.



Sultan *Suleimann Oktogamos*, 436 – 512  
Wesir *Zacharias Ben Schacharias*

# Darstellung des Schach-Bretts in C

Vorüberlegung:

- alle Damen müssen in verschiedenen Reihen stehen
- es müssen 8 Damen gesetzt werden
- also muß in jeder Reihe eine Dame stehen.
- Reihen werden sukzessive besetzt
- Zählung fängt bei 0 an

Darstellung des Bretts als C struct mit 2 Komponenten

1. Anzahl der bereits gesetzten Damen

`numberQueens`

2. Spezifikation der Spalten, in denen Damen gesetzt sind durch Feld

`column[8]`

Dame in  $i$ -ter Zeile steht in Spalte `column[i]`

```
#define BOARD_SIZE 8
```

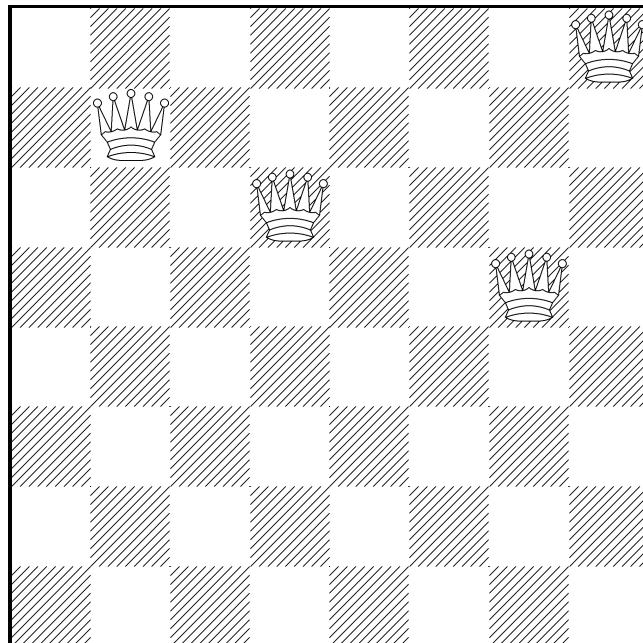
```
typedef struct {  
    unsigned numberQueens;  
    unsigned column[BOARD_SIZE];  
} Board;
```




# Graphische Darstellung des Schach-Bretts

```
Board = {  
    4,  
    { 7, 1, 3, 6 }  
}
```

- 4 Damen sind gesetzt
- Dame in der 0-ten Reihe steht in Spalte 7
- Dame in der 1-ten Reihe steht in Spalte 1
- Dame in der 2-ten Reihe steht in Spalte 3
- Dame in der 3-ten Reihe steht in Spalte 6

Zählung beginnt bei 0, wegen C Indizierung der Felder



$\langle 0, 0 \rangle$						$\langle x + d, y - d \rangle$	
					$\ddots$		
		$\ddots$		$\langle x + 1, y - 1 \rangle$			
			$\langle x, y \rangle$				
		$\ddots$		$\langle x + 1, y + 1 \rangle$			
	$\ddots$				$\ddots$		
						$\langle x + d, y + d \rangle$	
							

1. Steigende Diagonale:

$$x + y = (x + 1) + (y - 1) = \dots = (x + d) + (y - d)$$

Damen in Zeile  $i$  und Zeile  $j$  in selber steigender Diag.

$$i + \text{column}[i] = j + \text{column}[j]$$

2. Fallende Diagonale:

$$x - y = (x + 1) - (y + 1) = \dots = (x + d) - (y + d)$$

Damen in Zeile  $i$  und Zeile  $j$  in selber fallender Diag.

$$i - \text{column}[i] = j - \text{column}[j]$$

## Test, ob Dame gesetzt werden kann

**Gegeben:**     Brett     board  
                 Zeile      $x = \text{board} \rightarrow \text{numberQueens}$   
                 Spalte     $y = \text{nextColumn}$

**Frage:**        Kann Dame in  $\langle x, y \rangle$  gesetzt werden?

**Vorgehen:**    1. Überprüfe Spalte nextColumn  
                 2. Überprüfe fallende Diagonale  
                 3. Überprüfe steigende Diagonale

```
bool check(Board* board, unsigned nextColumn)
{
    unsigned row = board->numberQueens;
    for (unsigned i = 0; i < board->numberQueens; ++i)
    {
        if (board->column[i] == nextColumn)
        {
            return false;
        }
        if (i - board->column[i] == row - nextColumn)
        {
            return false;
        }
        if (i + board->column[i] == row + nextColumn)
        {
            return false;
        }
    }
    return true;
}
```

# Hilfs-Funktionen

Erzeugung eines leeren Bretts

1. Speicherplatz allozieren
2. numberQueens mit 0 initialisieren

```
Board* createEmpty()
{
    Board* newBoard = malloc( sizeof(Board) );
    newBoard->numberQueens = 0;
    return newBoard;
}
```

Hinzufügen einer Dame (erzeugt neues Brett)

1. Speicherplatz für neues Brett allozieren
2. numberQueens initialisieren
3. vorhandene Damen kopieren
4. neue Dame hinzufügen

```
Board* addQueen(Board* board, unsigned nextColumn)
{
    Board* newBoard = malloc( sizeof(Board) );
    newBoard->numberQueens = board->numberQueens + 1;
    for (unsigned i = 0; i < board->numberQueens; ++i) {
        newBoard->column[i] = board->column[i];
    }
    newBoard->column[board->numberQueens] = nextColumn;
    return newBoard;
}
```

# Backtracking

## Algorithmus

1. Wenn alle Damen auf dem Brett sind: fertig!
2. Finde nächste Spalte, in die Dame gesetzt werden kann und setze Dame: neues Brett.
3. Rekursiv: Versuche, neues Brett zu vervollständigen.
4. Funktioniert: fertig.
5. Sonst: versuche nächste Spalte

```
Board* complete(Board* board)
{
    if (board->numberQueens == BOARD_SIZE) {
        return board;
    }
    for (unsigned i = 0; i < BOARD_SIZE; ++i) {
        if (check(board, i)) {
            Board* nextBoard = addQueen(board, i);
            printBoard(nextBoard);
            Board* result = complete(nextBoard);
            if (result != 0) {
                return result;
            } else {
                // !!! backtrack !!!
                printBoard(board);
                free(nextBoard);
                continue;
            }
        }
    }
    return 0;
}
```

# Backtracking: Anwendung

Grammatik für arithmetische Ausdrücke:

$$G_{\text{calc}} = \langle T, N, R, S \rangle$$

1.  $T = \{\text{number}\}$
2.  $N = \{Sum, Prod, Factor\}$
3. Die Menge der Regeln ist wie folgt gegeben:

$$\begin{array}{lcl} Sum & \rightarrow & Sum \text{ "+" } Prod \\ & | & Sum \text{ "-" } Prod \\ & | & Prod \\ \\ Prod & \rightarrow & Prod \text{ "*" } Factor \\ & | & Prod \text{ "/" } Factor \\ & | & Factor \\ \\ Factor & \rightarrow & "(" Sum ")" \\ & | & number \end{array}$$

4.  $S = Sum$

Vorteile dieser Grammatik

1. Grammatik ist *eindeutig*: Jeder Ausdruck aus  $\mathcal{L}(G_{\text{calc}})$  hat genau einen Parse-Baum.
2. Grammatik berücksichtigt
  - (a) *Punkt vor Strich*:

$$x + y * z = x + (y * z)$$

- (b) Links-Assoziativität von "-" und "/"

$$x - y - z = (x - y) - z$$



# Projekt: Taschenrechner

**Ziel:** Sprache  $\mathcal{L}(G_{\text{calc}})$  der arithmetische Ausdrücke mit *recursive descent* Parser auswerten

**Problem:** Grammatik  $G_{\text{calc}}$  ist links-rekursiv

**Lösung:** Recursive Descent Parser muß backtracken!

**Nachteil:** resultierender Parser nicht sehr effizient

**Vorteil:** einfach zu implementieren

## Vorgehen:

### 1. Festlegung der Daten-Strukturen

Jedem Nicht-Terminal  $X \in N$  entspricht eine struct

```
typedef struct X* XPtr;
```

(a) *Sum* entspricht struct Sum

(b) *Prod* entspricht struct Prod

(c) *Factor* entspricht struct Factor

### 2. Festlegung der Funktionen

Jedem Nicht-Terminal  $X \in N$  entspricht Funktion

```
XPtr parseX(char* begin, char* end)
```

die die Sprache  $\mathcal{L}(X)$  erkennt.

begin: Pointer auf erstes Zeichen

end: Pointer **hinter** letztes Zeichen

Funktion erfolgreich, wenn

```
*begin *(begin+1) ... *(end-1)  $\in \mathcal{L}(X)$ 
```

## Festlegung der Daten-Strukturen

Genaue Analogie zwischen Grammatik und Daten-Struktur:

$$\begin{array}{lcl} \text{Sum} & \rightarrow & \text{Sum "+" Prod} \\ & | & \text{Sum "-" Prod} \\ & | & \text{Prod} \end{array}$$

```
struct Sum {  
    char      operation;  
    SumPtr    arg1;  
    ProdPtr   arg2;  
    unsigned  ctr;  
};
```

Bedeutung der Komponenten

operation: entweder "+" oder "-"  
arg1: erstes Argument oder 0  
arg2: zweites Argument oder einziges Argument  
ctr: nur zur graphischen Ausgabe

Semantik:

1. Fall:  $\text{arg1} \neq 0$ :

Fall liegt vor, wenn eine der ersten beiden Grammatik Regeln verwendet wurde. Semantik:

$\text{arg1 operation arg2}$

2. Fall:  $\text{arg1} == 0$ :

Fall liegt vor, wenn die letzte Grammatik Regeln verwendet wurde. Semantik:

$\text{arg2}$

# Festlegung der Daten-Strukturen

## Grammatik-Regeln für *Prod*

$$\begin{array}{lcl} \textit{Prod} & \rightarrow & \textit{Prod} \text{ "*" } \textit{Factor} \\ & | & \textit{Prod} \text{ "/" } \textit{Factor} \\ & | & \textit{Factor} \end{array}$$

```
struct Prod {  
    char      operation;  
    ProdPtr   arg1;  
    FactorPtr arg2;  
    unsigned  ctr;  
};
```

## Bedeutung der Komponenten

**operation:** entweder "\*" oder "/"

**arg1:** erstes Argument oder 0

**arg2:** zweites Argument oder einziges Argument

**ctr:** nur zur graphischen Ausgabe

## Semantik:

### 1. Fall: $\text{arg1} \neq 0$ :

Fall liegt vor, wenn eine der ersten beiden Grammatik Regeln verwendet wurde. Semantik:

$\text{arg1 operation arg2}$

### 2. Fall: $\text{arg1} == 0$ :

Fall liegt vor, wenn die letzte Grammatik Regeln verwendet wurde. Semantik:

$\text{arg2}$

# Festlegung der Daten-Strukturen

## Grammatik-Regeln für *Factor*

$$\begin{array}{lcl} \textit{Factor} & \rightarrow & \text{"(" Sum ")} \\ & | & \text{number} \end{array}$$

```
struct Factor {  
    int      number;  
    SumPtr   sumPtr;  
    unsigned ctr;  
};
```

## Bedeutung der Komponenten

**number:** Zahl, falls zweite Regel verwendet wurde

**sumPtr:** geklammerter Ausdruck, falls erste Regel verwendet wurde

**ctr:** nur zur graphischen Ausgabe

## Semantik:

### 1. Fall: `sumPtr != 0`:

Fall liegt vor, wenn die erste Grammatik Regeln verwendet wurde. Semantik:

`sumPtr`

### 2. Fall: `sumPtr == 0`:

Fall liegt vor, wenn die letzte Grammatik Regeln verwendet wurde. Semantik:

`number`

# Parsen von *Sum*

## Grammatik–Regeln

$$\begin{array}{lcl} \textit{Sum} & \rightarrow & \textit{Sum} \text{ "+" } \textit{Prod} \\ & | & \textit{Sum} \text{ "-" } \textit{Prod} \\ & | & \textit{Prod} \end{array}$$

## Algorithmus

1. Versuche, letzte Regel anzuwenden  
Falls erfolgreich: fertig!
2. Suche Zeichen "+" oder "-" in Intervall  
[begin, end-1]  
Sei ptr gefunden mit
  - (a)  $\text{begin} < \text{ptr} < \text{end}$
  - (b)  $\text{*ptr} \in \{ \text{"+"}, \text{"-"} \}$Falls erfolgreich:
  - (a) Parse rekursiv [begin,ptr] als *Sum*
  - (b) Parse rekursiv [ptr+1,end] als *Prod*Falls erfolgreich: fertig  
Sonst backtracken: Gehe zu 2. und suche nächstes Zeichen "+" oder "-".
3. Falls Suche nach "+" oder "-" scheitert:  
Parsen von *Sum* nicht möglich  
return 0;

## Parsen von *Sum*

```
SumPtr parseSum(char* begin, char* end)
{
    ProdPtr prod = parseProd(begin, end);
    if (prod != 0) {
        SumPtr sum = malloc( sizeof(struct Sum) );
        sum->arg1 = 0;
        sum->arg2 = prod;
        sum->ctr  = nodeCounter++;
        return sum;
    }
    for (char* ptr = begin + 1; ptr < end; ++ptr)
    {
        if (*ptr == '+' || *ptr == '-') {
            SumPtr firstSum = parseSum(begin, ptr);
            if (firstSum != 0) {
                ProdPtr prod = parseProd(ptr + 1, end);
                if (prod != 0) {
                    SumPtr sum =
                        malloc( sizeof(struct Sum) );
                    sum->operation = *ptr;
                    sum->arg1 = firstSum;
                    sum->arg2 = prod;
                    sum->ctr  = nodeCounter++;
                    return sum;
                } else {
                    free(firstSum);
                }
            }
        }
    }
    return 0;
}
```

## Parsen von *Fact*

### Grammatik–Regeln

$$\begin{array}{lcl} \textit{Factor} & \rightarrow & \text{"(" Sum ")} \\ & | & \text{number} \end{array}$$

### Algorithmus:

1. Falls erstes Zeichen "(" ist, wende erste Regel an.
  - (a) Parse rekursiv *Sum*:  
`parseSum(begin + 1, end - 1);`
  - (b) Überprüfe, ob letztes Zeichen ")" ist.
2. Falls erstes Zeichen Ziffer ist, wende zweite Regel an.  
Werte der Ziffern im Ascii–Alphabet

'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
48	49	50	51	52	53	54	55	56	57

**Beobachtung:** Werte sind kontinuierlich verteilt.

```
unsigned char2unsigned(char c) {  
    return c - '0';  
}
```

## Parsen von *Fact*

```
FactorPtr parseFactor(char* begin, char* end) {
    if (*begin == '(') {
        SumPtr sum = parseSum(begin + 1, end - 1);
        if (sum != 0 && *(end-1) == ')') {
            FactorPtr factor =
                malloc( sizeof(struct Factor) );
            factor->sumPtr = sum;
            factor->ctr     = nodeCounter++;
            return factor;
        } else {
            free(sum);
            return 0;
        }
    }
    if (isdigit(*begin)) {
        int num = 0;
        while (isdigit(*begin) && begin < end) {
            num = 10 * num + *begin - '0';
            ++begin;
        }
        if (begin == end) {
            FactorPtr factor =
                malloc( sizeof(struct Factor) );
            factor->number = num;
            factor->sumPtr = 0;
            factor->ctr     = nodeCounter++;
            return factor;
        }
    }
    return 0;
}
```