Sprachbeschreibung

EBNF = Regel , {Regel} ;
Regel = links , "=" , rechts , ";" ;
links = nonterminal ;
rechts = (terminal | nonterminal | rund | eckig | geschweift | senkrecht) , {"," , rechts} ;
rund = "(" , rechts , ")" ; // Gruppierung
eckig = "[" , rechts , "]" ; // ein oder kein mal
geschweift = "{" , rechts , "}" ; // beliebig häufige Wiederholung (auch kein mal)
senkrecht = rechts , {"|" , rechts} ; // Alternativen (entweder oder)
terminal = """ , text , """ ;
nonterminal = buchstabe,{alphanum} ;
text = zeichen,{zeichen} ;
zeichen = (buchstabe | zahl | sonderzeichen) ;
alphanum = (buchstabe | zahl) ;

Hoare-Kalkül

Hoare-Tripel	$\{P\}\ S\ \{R\}$		
Axiom der leeren Anweisung	$\{P\}$ NOP $\{P\}$		
Zuweisungsaxiom	$\{R(x=E)\}$ x:=E $\{R\}$		$\{P\}$ x:=E $\{P(x=E^{-1})\}$
Axiom der nichtterminierenden Anweisung	$\{P\}$ Abort {false}		
Regel der sequenziellen Komposition	$\{P\}\ S_1\ \{Q\}$		
	$\{Q\}\ S_2\ \{R\}$	$\Rightarrow$	$\{P\}\ S_1;S_2\ \{R\}$
Regel der Fallunterscheidung	$\{P \wedge B\}\ S_1\ \{R\}$		
	$\{P \wedge \text{-}B\}\ S_2\ \{R\}$	$\Rightarrow$	$\{P\}$ IF B THEN S_1 ELSE S_2 FI $\{R\}$
Regel der Iteration	$\{B \wedge I\}\ S\ \{I\}$	$\Rightarrow$	$\{I\}$ WHILE B DO S END $\{\text{-}B \wedge I\}$
Terminierungsregel	$\{B \wedge I \wedge \tau=k\}\ S\ \{\tau<k\}$		
	$I \wedge \tau<c \Rightarrow \text{-}B$		
	"S *terminiert*"	$\Rightarrow$	"WHILE B DO S END *terminiert*"
Regel für Prozeduren	PROC p(VAR x:T); S; ENDp;		
	$\{Q\}\ S\ \{R\}$	$\Rightarrow$	$\{Q(x=E)\}\ p(E)\ \{R(x=E)\}$
Verstärkungs- / Abschwächungsregel	$P \Rightarrow P'$		
	$R' \Rightarrow R$		
	$\{P'\}\ S\ \{R'\}$	$\Rightarrow$	$\{P\}\ S\ \{R\}$

Faustregeln zum Finden der Invarianten und der Schleifenbedingung

$I \supseteq P \cup R$	(P und R geeignet abschwächen)
$B \supseteq \text{-}R$	(Es sollte von R der Teil fehlen, der in I enthalten ist)

O-Kalkül

$f \in O(g) \quad \Leftrightarrow \quad \exists\ c \in \mathbb{R}^+, n_0 \in \mathbb{N}: \forall\ n>n_0:\ f(n) \le c \cdot g(n)$

Regeln

$f \in O(f)$
$k \cdot O(f) = O(f)$, k=const
$O(O(f)) = O(f)$
$O(f) \cdot O(g) = O(f \cdot g)$
$O(f \cdot g) = f \cdot O(g)$
$O(f) + O(g) = O(f + g)$
$f + O(g) \subseteq O(f + g)$
$O(f) + O(g) = O(f)$, falls $O(g) \subseteq O(f)$
$\sum a_i n^i \in O(n^k)$, $a_k>0$, k=max(i)

Rekurrenzen

$A(n) = A(n-1) + b\ n^k$		$O(n^{k+1})$
$A(n) = c\ A(n-1) + b\ n^k$	$c>1$	$O(c^n)$
$A(n) = c\ A(n/d) + b\ n^k$	$c>d^k$	$O(n^{\log_d(c)})$
$A(n) = c\ A(n/d) + b\ n^k$	$c<d^k$	$O(n^k)$
$A(n) = c\ A(n/d) + b\ n^k$	$c=d^k$	$O(n^k \log(n))$

Problemlösung

Deklarationen

`ps`: Zustand
`psL`: Zustandsliste

`optimal?(ps)`: Ist `ps` Lösung?
`successors(ps)`: Liefert alle Nachfolger von `ps`
`feasible(psL)`: Liefert alle zulässigen Zustände aus `psL`
`insert(psL1,psL2)`: sortiert `psL1` in `psL2` entsprechend des Algorithmus'

```
PROCEDURE Solve(Anfang:State):State
  VAR
    psL:LIST OF State
  BEGIN
    psL:=Anfang
    WHILE (NOT optimal?(ft(psL))) AND (psL≠Ø)
      insert(feasible(successors(ft(psL))),rt(psL))
    END WHILE
    RETURN(ft(psL))
  END PROC
```

Algorithmus	**Voraussetzung**	**psL**	**insert**	**feasible**
Tiefensuche	endlicher Suchraum	Stack (LIFO) 1 Pfad	vorn anhängen	erlaubt `AND NOT in(psL)`
Breitensuche	Suchraum	Queue (FIFO) 1 Ebene	hinten anhängen	erlaubt [`AND` unbekannt]
B'n' B	" + lokales Krit: bisherige Kosten `g`	nach `g` sortierte Queue	nach `g` sortiert einfügen	"
A*	" + untere Schranke für Restkosten h*	nach `f*=g+h*` sortierte Queue	nach `f*` sortiert einfügen, doppelte durch kürzere ersetzen	"
greedy	lokales Krit zum globalen Optimum	Variable (`ps`)	zuweisen	best

Tiefensuche rekursiv

```
PROCEDURE Tiefensuche(Anfang:State):State
  VAR
    best:State
  PROCEDURE BT(psL:LIST OF State)
    VAR
      p:State
    BEGIN
      IF optimal?(ft(psL)) THEN IF best>ft(psL) THEN best:=ft(psL)
                           ELSE FOR p:=feasible(successors(ft(psL)))
                                  BT(p++psL)
                                END FOR
      END IF
    END PROC

  BEGIN
    best:=Ø
    BT(Anfang)
    RETURN(best)
  END PROC
```

Relaxation

$\delta_{min}(x,y) = D_n(x,y) \leq ... \leq D_{i+1}(x,y) \leq D_i(x,y) \leq ... \leq D_0(x,y)$

$D_{i+1}(x,y) = \min(D_i(x,y), D_i(x,a) + \text{edge}(a,y))$

Dijkstra-Algorithmus (single-source shortest-paths problem)

```
PROCEDURE Dijkstra(Graph:Nodes,Start:Node,VAR D:ARRAY OF cost,VAR P:ARRAY OF Node)
  VAR
    x,u:Node
  BEGIN
    FOR x:=Graph                      //Noch keine Wege gefunden
      D[x]:=∞
      P[x]:=Ø
    END FOR
    D[Start]:=0                       //Hier sind wir schon

    WHILE Graph≠Ø                     //Alle Knoten abarbeiten
      u:=min(Graph)                   //Knoten mit kleinster Entfernung entfernen
      FOR x:=successors(u)            //Gibt es eine Abkürzung über u?
        IF D[x]>D[u]+edge(u,x) THEN   //Ja!
          D[x]:=D[u]+edge(u,x)        //Relaxation
          P[x]:=u                     //Vorgänger merken für Wegrekonstruktion
        END IF
      END FOR
    END WHILE
  END PROC

PROCEDURE Pfad(P:ARRAY OF Node,Start:Node,Ziel:Node):LIST OF Node
  BEGIN
    IF Start=Ziel THEN RETURN(Start)
                  ELSE RETURN(Pfad(P,Start,P[Ziel])++Ziel)
    END IF
  END PROC
```

Floyd-Warshall-Algorithmus (all-pairs shortest-paths problem)

```
PROCEDURE FW(Graph:Nodes,VAR D:ARRAY OF cost,VAR P:ARRAY OF Node)
  VAR
    x,y,u:Node
  BEGIN
    FOR x:=Graph                          //Noch keine Wege gefunden
      FOR y:=Graph
        D[x,y]:=edge(x,y)
        P[x,y]:=Ø
      END FOR
    END FOR

    FOR u:=Graph                          //Alle Knoten abarbeiten
      FOR x:=Graph
        FOR y:=Graph
          IF D[x,y]>D[x,u]+D[u,y] THEN    //Gibt es eine Abkürzung über u?
            D[x,y]:=D[x,u]+D[u,y]         //Relaxation
            P[x,y]:=u                     //Vorgänger merken für Wegrekonstruktion
          END IF
        END FOR
      END FOR
    END FOR
  END PROC

PROCEDURE Pfad(P:ARRAY OF Node,x:Node,y:Node):LIST OF Node
  BEGIN
    IF P[x,y]=Ø THEN RETURN(Ø)
                ELSE RETURN(Pfad(P,Start,P[x,y])++P[x,y]++Pfad(P,P[x,y],Ziel))
    END IF
  END PROC
```

Semaphoren

```
class semaphore{
private: ProcessList queue;
         int         counter;
public:  semaphore(int startwert) {counter=startwert;};
         void P(void)             {if(counter--<=0) BlockProcess(queue);};
         void V(void)             {if(counter++< 0) ContinueProcess(queue);};
};
class semacount:public semaphore,public int{
public:  semacount(int i,int j=0):semaphore(i),int(j) {};
};
```

Synchronisation

Keine Priorisierung

```
semacount Zaehler1(1,0),Zaehler2(1,0);
semaphore Ausschluss1(m),Ausschluss2(n),Ausschluss(1);
```

```
Process1(){
  Ausschluss1.P();
  Zaehler1.P();
    if(!Zaehler1++) Ausschluss.P();
  Zaehler1.V();
      //Kritischer Abschnitt
  Zaehler1.P();
    if(!--Zaehler1) Ausschluss.V();
  Zaehler1.V();
  Ausschluss1.V();
}
```

```
Process2(){
  Ausschluss2.P();
  Zaehler2.P();
    if(!Zaehler2++) Ausschluss.P();
  Zaehler2.V();
      //Kritischer Abschnitt
  Zaehler2.P();
    if(!--Zaehler2) Ausschluss.V();
  Zaehler2.V();
  Ausschluss2.V();
}
```

Priorisierung von Prozess 1

```
semaphore Halt(1),Vorhalt(1);
```

```
Process1(){
  Zaehler.P();
    if(!Zaehler1++){Halt.P();
                    Ausschluss.P();}
  Zaehler1.V();
  Ausschluss1.P();
      //Kritischer Abschnitt
  Ausschluss1.V();
  Zaehler1.P();
    if(!--Zaehler1) {Ausschluss.V();
                     Halt.V();}
  Zaehler1.V();
}
```

```
Process2(){
  Ausschluss2.P();
  Vorhalt.P();Halt.P();
  Zaehler2.P();
    if(!Zaehler2++) Ausschluss.P();
  Zaehler2.V();
  Halt.V();Vorhalt.V();
      //Kritischer Abschnitt
  Zaehler2.P();
    if(!--Zaehler2) Ausschluss.V();
  Zaehler2.V();
  Ausschluss2.V();
}
```

1. Leser-Schreiber-Problem

```
Schreiber(){
  Ausschluss.P();
    //Kritischer Abschnitt
  Ausschluss.V();
}
```

```
Leser(){
  Zaehler1.P();
    if(!Zaehler1++) Ausschluss.P();
  Zaehler1.V();
      //Kritischer Abschnitt
  Zaehler1.P();
    if(!--Zaehler1) Ausschluss.V();
  Zaehler1.V();
}
```

2. Leser-Schreiber-Problem

```
Schreiber(){
  Zaehler1.P();
    if(!Zaehler1++) Halt.P();
  Zaehler1.V();
  Ausschluss.P();
      //Kritischer Abschnitt
  Ausschluss.V();
  Zaehler1.P();
    if(!--Zaehler1) Halt.V();
  Zaehler1.V();
}
```

```
Leser(){
  Vorhalt.P();Halt.P();
  Zaehler2.P();
    if(!Zaehler2++) Ausschluss.P();
  Zaehler2.V();
  Halt.V();Vorhalt.V();
      //Kritischer Abschnitt
  Zaehler2.P();
    if(!--Zaehler2) Ausschluss.V();
  Zaehler2.V();
}
```