Übung 1

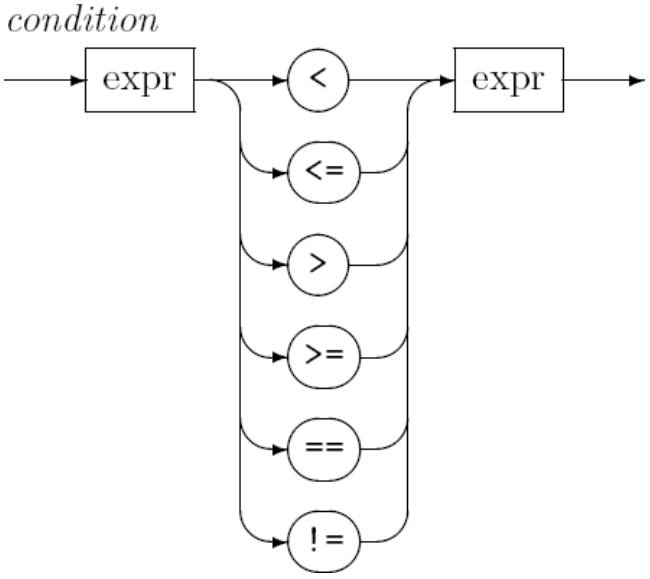
Aufgabe 1:

a) Ändern Sie das Syntaxdiagramm für *condition* so ab, dass *compOP* durch die eigentlichen Vergleichsoperatoren ersetzt wird.

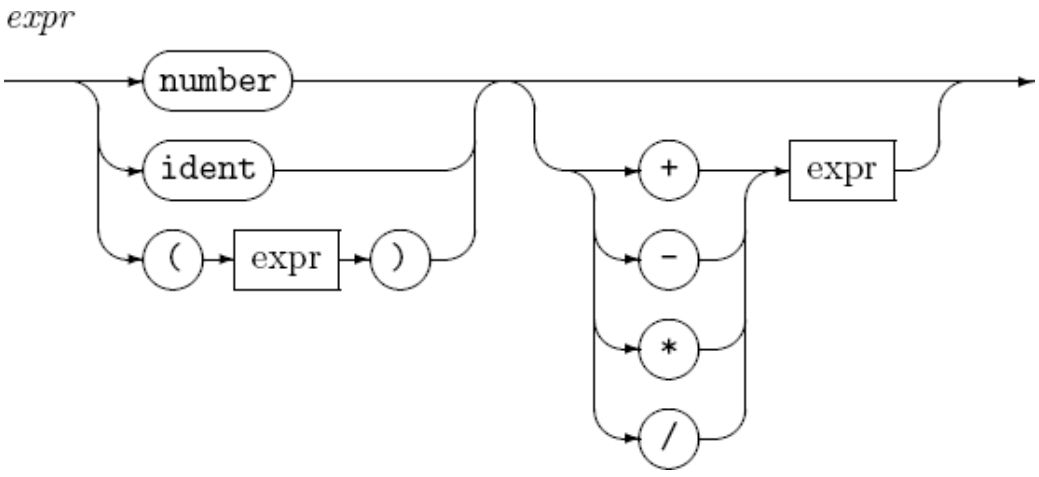
b) Fügen Sie die Syntaxdiagramme für *expression, product* und *term* so zu einem Syntaxdiagramm zusammen, dass syntaktisch genau die gleichen arithmetischen Ausdrücke erzeugt werden können

Lösung:

a) einfach compOP durch die Fallunterscheidung für die 6 Fälle unterscheiden.



b) Diagramm aus Aufgabenstellung Praktikum1 Aufg. 2 mit zusätzlichem Fall für ident.



Aufgabe 2:

Wie kann man mit javacc Kommentare behandeln?

Lösung:

```
SKIP:
     { "/*" : COMM }
     <COMM> : SKIP { "*/" : DEFAULT }
     <COMM> : SKIP { < ~[] > }
```

Aufgabe 3:

Die Fibonacci-Funktion *fib: Nat -> Nat* ist wie folgt rekursiv programmiert:

fib(0) = 0

fib(1) = 1

fib(n) = fib(n-1) + fib(n-2), falls n>1

Schreiben Sie ein M32-Assemblerprogramm zur Implementierung der Fibonacci-Funktion

Lösung:

```
            ORG   0H
FIB         EQU   8H    ; Fib(FIB) = ?

            ;Starten...
            MOV   @FIB_PRE2, 0H
            MOV   @FIB_PRE1, 1H
            MOV   @FIB_ACT, 0H

            MOV   R1, FIB ; Counter

DoLoop:     CMP   R1, 0H
            JZ    Stop
            ;PRN  @FIB_ACT
            SUB   R1, 1H
            MOV   R2, @FIB_ACT ; Aktuellen Wert speichern
            ADD   @FIB_ACT, @FIB_PRE1
            MOV     @FIB_PRE2, @FIB_PRE1
            MOV     @FIB_PRE1, R2
            JMP   DoLoop

Stop:       PRN   @FIB_ACT
            HALT

FIB_ACT           DS    1
FIB_PRE1    DS    1     ; n-1; fib(1)
FIB_PRE2    DS    1     ; n-2 ==> älter als FIB_PRE1, fib(0)
            END
```

Aufgabe 4:

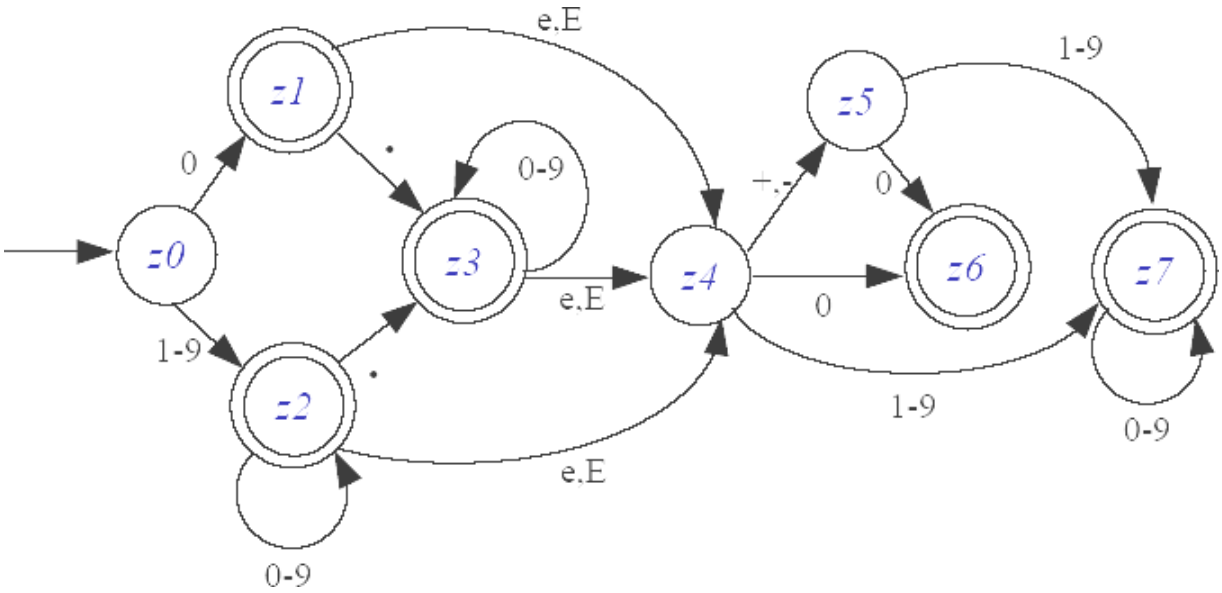
a) Geben Sie reguläre Ausdrücke zur Beschreibung der Struktur von Gleitkommazahlen aus Java an. Dabei können Sie die „f,F“-Endung der Übersichtlichkeit halber weglassen.

b) Übersetzen Sie die regulären Ausdrücke aus a) in NEAs.

Lösung:

a)
```
("0" | (["1"-"9"] ["0"-"9"]*))
((“.” ["0"-"9"]*)? (["e","E"] ["+","-"]? ("0" | (["1"-"9"] ["0"-"9"]*)))?)?
```

b) nicht mit Kleene



Aufgabe 5:

Zeigen Sie, wie man die booleschen Operationen `!`, `&&`, `||` in Mini-Java implementiert.

Lösung:

Durch if-Kontrollstrukturen:

```
!bedingung
if bedingung {
} else {
     ...
}
```

```
bedingung1 && bedingung2
if bedingung1 {
     if bedingung2 {
          ...
     }
}
bedingung1 || bedingung2
a = 0;
if bedingung1 {
     a = 1;
} else if bedingung2 {
     a = 1;
}

if a == 1 {
     ...
}
```

Aufgabe 6:

Die Fakultäts-Funktion *fak: Nat -> Nat* ist wie folgt rekursiv programmiert:

fak(0) = 1

fak(n) = n fak(n-1), falls n>0*

Schreiben Sie ein Mini-Java-Programm zur Implementierung der Fakultäts-Funktion

Lösung:

```
final int n = 5;
int i = 1, f = 1;
{
     while i <= n {
          f = f * i;
          i = i + 1;
     }
     print(f);
}
```

Übung 2

Aufgabe 7:

Welche Ausgabe liefert ein Scanner an einen Parser für das folgende Mini-Java-Programm:

```
final int c=2;
int x = 17, y = 23;

if 4*(x-11) > 10 {
  while x < 20 {
    print(x);
    x = x + c;
  }
}
else {
  while y > 5 {
    print(y);
    y = y - 2*c;
  }
}
```

Lösung:

für Schlüsselwörter Token definiert!

(FINAL,) (INT,) (IDENT, c) ("=",) (NUM, 2) (";",)

(INT,) (IDENT, x) ("=",) (NUM, 17) (",",) (IDENT, y) ("=",) (NUM, 23) (";",)

(IF,) (NUM, 4) ("*",) ("(",) (IDENT, x) ("-",) (NUM, 11) (")",) (RELOP, >) (NUM, 10) (LBRACE,)

(WHILE,) (IDENT, x) (RELOP, <) (NUM, 20) (LBRACE,)

(PRINT,) ("(",) (IDENT, x) (")",) (";",)

(IDENT, x) ("=",) (IDENT, x) ("+",) (IDENT, c) (";",)

(RBRACE)

(RBRACE)

(ELSE,) (LBRACE,)

(WHILE,) (IDENT, y) (RELOP, >) (NUM, 5) (LBRACE,)

(PRINT,) ("(",) (IDENT, y) (")",) (";",)

(IDENT, y) ("=",) (IDENT, y) ("-",) (NUM, 2) ("*",) (IDENT, c) (";",)

(RBRACE,)

(RBRACE,)

Aufgabe 8:

Implementieren Sie einen Scanner für Mini-Java direkt in Java.

Lösung: Idee

Lese jeweils das nächste Zeichen und simuliere die einzelnen Automaten.

Wenn Token gefunden, schreibe dieses ggf. mit Attribut auf Ausgabe.

Ein Java-Programm zu dieser Aufgabe hat Herr van Porten.

Aufgabe 9:

L sei die Sprache über dem Alphabet $\Sigma = \{0,1\}$, die alle Wörter gerader Länge enthält, die mit 0 beginnen und alle Wörter ungerader Länge, die mit 1 beginnen.

a) Geben Sie einen regulären Ausdruck an, der L beschreibt.

b) Geben Sie einen NEA an, der L erkennt.

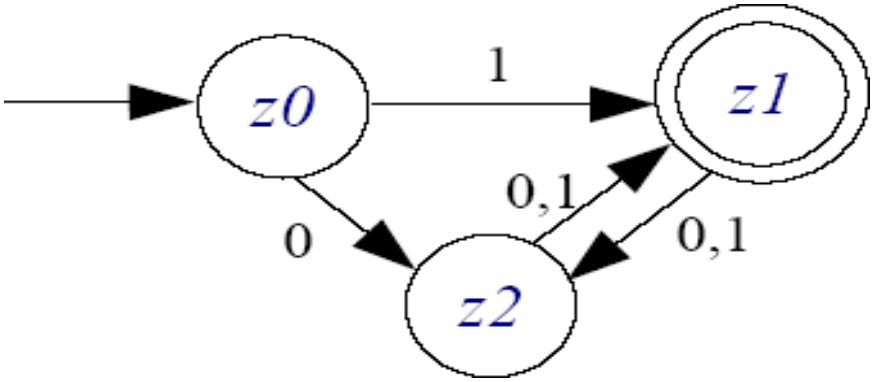
c) Transformieren Sie den NEA aus b) mit dem PMK-Algorithmus in einen äquivalenten DEA und überprüfen Sie die Korrektheit des DEAs mit
- i) einem kürzesten Wort, das mit 0 beginnt und in L ist und
- ii) einem kürzesten Wort, das mit 1 beginnt und nicht in L ist.

d) Führen Sie das NEA-Verfahren mit dem NEA aus b) und den beiden Wörtern aus c) durch.

Lösung:

a) (1 | 0 (0|1)) ((0|1) (0|1))*

b)



c) NEA aus b) ist bereits ein DEA. Deshalb liefert die PMK den gleichen Automat, nur dass in den Zuständen noch Mengenklammern drum sind.
({z0}, 0) --> {z2}
({z0}, 1) --> {z1}
({z1}, 0) --> {z2}
({z1}, 1) --> {z2}
({z2}, 0) --> {z1}
({z2}, 1) --> {z1}

i) **w=01:** {z0} -0> {z2} -1> {z1} {z1} ist Endzustand des DEA

ii) **w=10:** {z0} -1> {z1} -0> {z3} {z3} kein Endzustand des DEA

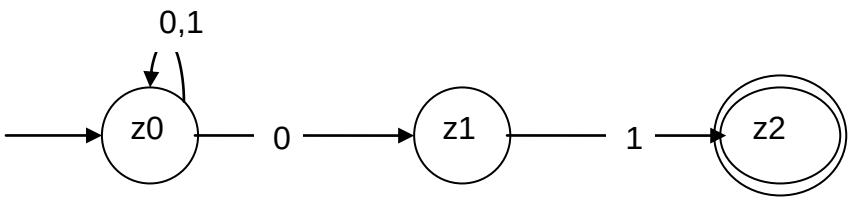
d) Gehe von {z0} aus und konstruiere jeweils Nachfolgezustand zur Laufzeit:

i) **w=01:** {z0} -0> {z2} -1> {z1} {z1} ∩ E = { z1 }

ii) **w=10:** {z0} -1> {z1} -0> {z3} {z3} ∩ E = { }

Aufgabe 10:

Gegeben sei folgender EA über dem Alphabet Σ = {0,1}:



a) Geben Sie die formale Definition für obigen EA an.

b) Ist der EA aus a) deterministisch und wenn nein, warum nicht?

c) Welche Sprache wird vom dem EA aus a) erkannt

d) Geben Sie einen regulären Ausdruck an, der L beschreibt.

Lösung:

a) M = ({z0,z1,z2}, {0,1}, dRel,z0,{z2}) mit
dRel={(z0,1,z0),(z0,0,z0),(z0,0,z1),(z1,1,z2)}.

b) M ist nicht deterministisch, da es 2 Tripel in dRel gibt mit 1. Komponente z0 und 2. Komponente 0.

c) L ist die Sprache aller Wörter über dem Alphabet {0,1}, die mit 01 enden.

d) (0|1)*01

Aufgabe 11:

a) Transformieren Sie den NEA aus Aufgabe 10 mit dem DEA-Verfahren in einen äquivalenten DEA und überprüfen Sie die Korrektheit des DEAs mit
i) einem Wort der Länge 4, das in L ist und
ii) einem Wort der Länge 4, das nicht in L ist.

b) Führen Sie das NEA-Verfahren mit dem NEA aus Aufgabe 10) und den beiden Wörtern aus a) durch.

Lösung:

a)

M = ({{z0},{z0,z1},{z0,z2}}, {0,1}, dFun, {z0}, {z0,z2} **dFun:**

({z0}, 0) --> {z0,z1}

({z0}, 1) --> {z0}

({z0,z1}, 0) --> {z0,z1}

({z0,z1}, 1) --> {z0,z2}

({z0,z2}, 0) --> {z0,z1}

({z0,z2}, 1) --> {z0}

i) **w=1101:** {z0} -1> {z0} -1> {z0}-0>{z0,z1}-1>{z0,z2}
{z0,z2} ist Endzustand des DEA

ii) **w=1100:** {z0} -1> {z0} -1> {z0}-0>{z0,z1}-0>{z0,z1}
{z0,z1} kein Endzustand des DEA

b)

Gehe von {z0} aus und konstruiere jeweils Nachfolgezustand zur Laufzeit:

i) **w=1101:** {z0} -1> {z0} -1> {z0}-0>{z0,z1}-1>{z0,z2}

{z0,z2} ∩ E = { z2 }

ii) **w=1100:** {z0} -1> {z0} -1> {z0}-0>{z0,z1}-0>{z0,z1} }

{z0,z1} ∩ E = { }

Aufgabe 12:

a) Geben Sie einen regulären Ausdruck in javaCC-Syntax an, der den strukturellen Aufbau deutscher KFZ-Kennzeichen formal spezifiziert: ein-, zwei- oder dreibuchstabige Kürzel, gefolgt von einem Bindestrich, gefolgt von einer ein- oder zweielementigen Buchstabenkombination, gefolgt von einer höchstens vierstelligen Zahl ungleich Null und ohne führende Nullen. Zwischen den vier genannten Teilen befindet sich jeweils ein Blank.

b) Überprüfen Sie die Korrektheit Ihres regulären Ausdrucks aus a) mit Hilfe von javaCC.

Lösung:

```
PARSER_BEGIN(U2_A12)
      public class U2_A12 {
            public static void main (String args []) {
      U2_A12 parser = new U2_A12(System.in);
      try {
                        parser.start();
                  } catch (ParseException e) {
                        System.err.println(e);
                  }
            }
  }
PARSER_END(U2_A12)

SKIP:
  { "\r"    | "\n" | "\t" }

TOKEN:
  {
      <LETTER : ["A"-"Z"]>
|
      <NUMBER : ["1"-"9"] (["0"-"9"]){0,3} >
|
```

```
      <KENNZEICHEN : (<LETTER>){1,3} " - " (<LETTER>){1,2} " " <NUMBER> >
  }

void start():
      {}
      {
            <KENNZEICHEN>

           <EOF> { System.out.println("Kennzeichen formal ok!"); }
      }
```

Aufgabe 13:

a) Geben Sie einen NEA M mit n Zuständen an, für den die DEA-Methode einen DEA mit 2^n Zuständen liefert.
Dabei müssen ggf. auch nicht erreichbare Zustände betrachtet werden.

b) Welchen DEA liefert die DEA-Methode für den NEA M aus a)?

Lösung:

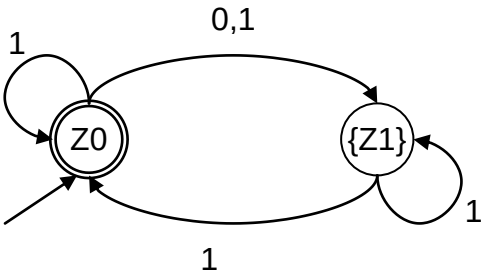
1. Beispiel

```
NEA = ({z0},{0,1},{(z0,0,z0)},z0,{z0})
DEA = ({{},{z0},{0,1},d,{z0},{{z0}}

      mit   d({z0},0) = {z0}
            d({z0},1) = {}
            d({},0) = {}
            d({},1) = {}
```

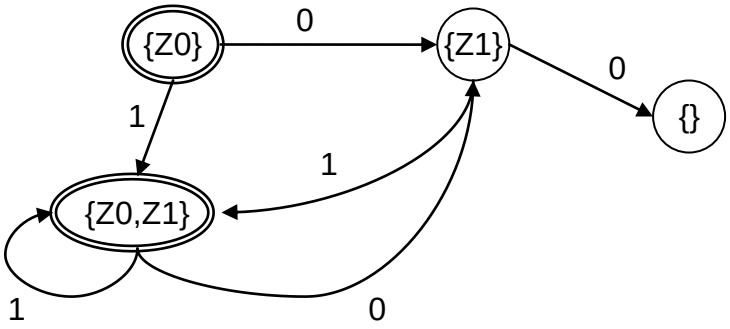
2. Beispiel

NEA



Bei der Lösung dieser Aufgabe kann man das Pferd teilweise von hinten aufzäumen, indem man zuerst einen DEA mit P(Z) Zuständen angibt und dann versucht, daraus einen NEA mit Z Zuständen zu konstruieren.

DEA



Übung 3

Aufgabe 14:

a) Geben Sie einen NEA M mit n Zuständen und ein Wort w an, so dass die NEA-Methode bei Eingabe von M und w 2^n verschiedene Zustände durchläuft.

b) Führen Sie die einzelnen Zustände in der richtigen Reihenfolge auf, die die NEA-Methode für den NEA M und das Wort w aus a) durchläuft.

Lösung:

Vorraussetzung dafür, dass alle 2^n Zustände des DEA durchlaufen werden, ist, dass $|w| = 2^n-1$. Desweiteren darf das Wort nicht in der von dem DEA bzw. NEA erkannten Sprache liegen, da sonst der Zustand {} nie erreicht werden kann.

Ich verwende den NEA aus Aufgabe 13 / Beispiel 2 für das Wort w = 100

T = {Z0}	a = 1
T = {Z0,Z1}	a = 0
T = {Z1}	a = 0
T = {}	a = EOF

$T \cap \{Z0\} == \{\} \blacktriangleright w \notin L(NEA)$

Aufgabe 15:

Überprüfen Sie, welche der folgenden Methoden in javaCC implementiert ist:

a) die DEA-Methode

b) die NEA-Methode

c) keine der beiden Methoden

Falls Sie mit c) geantwortet haben, beschreiben Sie das Vorgehen von javaCC beim Scannen.

Lösung:

Unter den FAQs zu javaCC wird das Verfahren erklärt. Es ist eine Mischung aus NEA- und DEA-Verfahren, wobei im Falle einer komplett nichtdeterministischen Entscheidung das Principle of the Longest Match verfolgt wird und ansonsten die vorderste Möglichkeit bevorzugt wird.

Somit kann man auch das Problem, dass Bezeichner keine Schlüsselwörter sein dürfen, damit lösen, dass man die Definition der Token für Schlüsselwörter vor Token für die Bezeichner aufführt.

Aufgabe 16:

a) Beschreiben Sie allgemein eine Transformation, die einen beliebigen regulären Ausdruck ra in einen NEA überführt, der die von ra beschriebene Sprache erkennt.

b) Transformieren Sie den regulären Ausdruck aus Aufgabe 10.d) gemäß Ihrer Konstruktion aus a) in einen NEA.

c) Wie unterscheidet sich der NEA aus Teil b) von dem aus Aufgabe 10?

Lösung:

a) Satz: (Kleene)

Reguläre Ausdrücke beschreiben genau die Typ3-Sprachen.

Deshalb nennt man diese auch reguläre Sprachen.

Bew.: hier nur angedeutet.

müssen bei NEA mehrere Anf. zustände zulassen: A = Menge der Anfangszustände

=> ändere PMK: Anfangszustand des DEA nicht mehr {z0}, sondern A.

ra -> Typ 3:

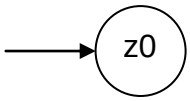
Sei ra ein regulärer Ausdruck über Σ.

Wir zeigen, dass es einen NEA gibt, der die von ra beschriebene Sprache erkennt.

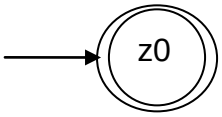
Erlaube bei NEA mehrere Anfangszustände: ersetze z0 durch Menge A der Anfangszustände

Induktion:

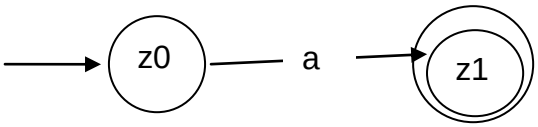
- ra = empty: Dann Sprache leer (kein Endzustand)



- ra = ew: Sprache enthält nur das leere Wort. (Anfangszustand = Endzustand)



- ra = a: Sprache enthält nur das einbuchstabige Wort a.

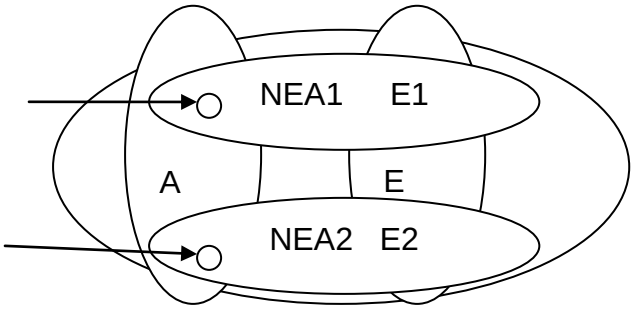


Schlüsse:

Nach Induktionsvoraussetzung gibt es einen NEA1=(Z1,Σ,δ1,A1,E1), der die Sprache, die von a1 beschrieben wird, erkennt und analog einen NEA2=(Z2,Σ,δ2,A2,E2), der die Sprache, die von a2 beschrieben wird, erkennt. Bem.: A1 und E1 können gleiche Zustände enthalten.

- ra = (a1|a2): Vereinigung der Sprachen, die von a1 oder a2 beschrieben werden.

Der Vereinigungsautomat (Z1\cup Z2,Σ,δ1\cup δ2,A1\cup A2,E1\cup E2) erkennt dann die gesuchte Sprache. (Parallelschaltung)



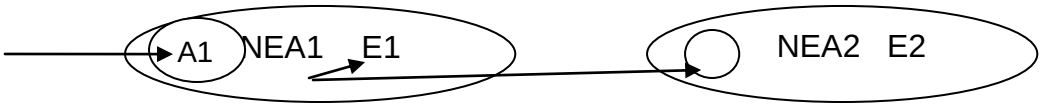
- ra = (a1xa2): Konkatenation der Sprachen, die von a1 oder a2 beschrieben werden.

Schalten NEA1 und NEA2 in Serie:

Startzustände von NEA1 und Endzustände von NEA2,

d.h.: E1 keine Endzustände mehr und A2 keine Anfangszustände

alle Zustände von NEA1, die einen Pfeil zu Endzustand von NEA1 haben, erhalten zusätzlich einen Pfeil mit gleicher Beschriftung zu Startzustand von NEA2.

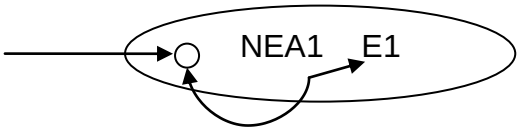


- ra = (a1)*: n-malige Konkatenation der Sprache, die von a1 beschrieben wird, mit sich selbst n = 0,1,2...

ε ist in (a1)*

Wenn ε nicht in Sprache, die von a1 beschrieben wird, dann muss man neuen Zustand hinzufügen, der Start und Endzustand ist und keine Verbindung zu Rest hat, um ε zu erkennen.

Neuer Automat hat die gleichen Start- und Endzustände wie NEA1. Um mehrmalige Konkatenation zu erreichen, muss man diese aber rückkoppeln. Dazu fügt man von allen Zuständen von NEA1, die einen Pfeil zu Endzustand von NEA1 haben, zusätzlich einen Pfeil mit gleicher Beschriftung zu allen Startzuständen von NEA1.



Somit haben wir für jeden regulären Ausdruck einen NEA konstruiert, der die von dem regulären Ausdruck beschriebene Sprache erkennt.

Müsste noch bewiesen werden -> Literatur Induktion

b)

Basisfälle	
RA0 = 0	NEA0 = ({Z0,Z1},Σ,d0,{Z0},{Z1}) mit d0 = {(Z0,0,Z1)}
RA1 = 1	NEA1 = ({Z2,Z3},Σ,d1,{Z2},{Z3}) mit d1 = {(Z2,1,Z3)}
Vereinigung	
RA2 = RA0|RA1	NEA2 = ({Z0,Z1,Z2,Z3},Σ,d2,{Z0,Z2},{Z1,Z3}) mit d2 = {(Z0,0,Z1),(Z2,1,Z3)}
Kleene-Stern	
RA3 = RA2*	NEA3 = ({Z0,z1,Z2,Z3},Σ,d3,{Z0,Z2},{Z0,Z1,Z2,z3}) mit d3 = {(Z0,0,Z1),(Z2,1,Z3), (Z0,0,Z0),(Z2,1,Z2), (Z0,0,Z2),(Z2,1,Z0)}
Hilfskonstrukt	
RA4 = 01	NEA4 = ({Z4,Z5,Z6},Σ,d4,{Z4},{Z6}) mit d4 = {(Z4,0,Z5),(Z5,1,Z6)}
Konkatenation	
RA5 = RA3xRA4	NEA5 = ({Z0,Z1,Z2,Z3,Z4,Z5,Z6},Σ,d5,{Z0,Z2,Z5},{Z6}) mit d5 = {(Z0,0,Z1),(Z2,1,Z3), (Z0,0,Z0),(Z2,1,Z2), (Z0,0,Z2),(Z2,1,Z0), (Z4,0,Z5),(Z5,1,Z6), (Z0,0,Z4),(Z2,1,Z4)}

Ich habe das Hilfskonstrukt eingeführt, da ich mir nicht sicher, ob, wenn ich RA5 = RA3xRA0xRA1 setze, es nicht zu Namenskonflikten bei den Zuständen aus NEA3 und NEA0 bzw. NEA1 kommt.

c) Ist viel komplexer -> darum führt man meist noch eine Minimalisierung des NEAs durch

Aufgabe 17:

a) Beschreiben Sie den Begriff "Nichtdeterministisches Verfahren".

b) Kann man jedes nichtdeterministische Verfahren in ein deterministisches transformieren und wenn ja, wie?

Lösung:

a.)

Ein nichtdeterministisches Verfahren erlaubt im Gegensatz zu einem deterministischem Verfahren, das eindeutig definiert, an welchem Punkt des Algorithmus welcher Folgeschritt zu unternehmen ist, dass es zu einem Punkt mehrere (verschiedene) Folgeschritte geben kann, von denen keiner, einer oder mehrere zum Ziel führen können. Welchen Folgeschritt man unternimmt, ist nicht eindeutig festgelegt und bleibt somit offen.

b.)

Der Nichtdeterminismus lässt sich auflösen, indem man eine eindeutige Reihenfolge bei der Auswahl der Folgeschritte festlegt. Endet der gewählte Schritt erfolglos, führt man Backtracking durch, d.h. man kehrt (sukzessiv) zum Ausgangspunkt zurück und unternimmt dann den nächsten Folgeschritt in der festgelegten Reihenfolge.

Aufgabe 18:

Betrachten Sie die CFG GSE auf Folie "Syntax-Analyse 8" und das Wort
w = IDENT "=" NUMBER "*" IDENT "*" NUMBER ";"

a) Geben Sie Syntaxdiagramme an, die die durch GSE erzeugte Sprache beschreiben.

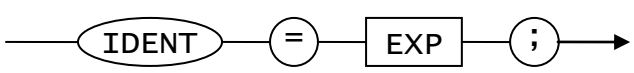
b) Geben Sie eine Linksableitung von w an.

c) Geben Sie eine Rechtsableitung von w an.

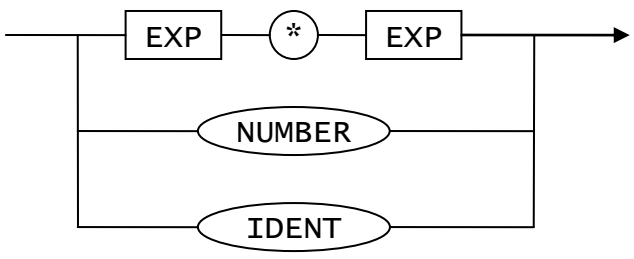
d) Ist die in b) angegebene Linksableitung die einzige Linksableitung von w? Wenn nein, geben Sie noch eine weitere Linksableitung an.

Lösung:

STMT



EXP



Linksableitung	**Alternative Linksableitung**
STMT	STMT
$\Rightarrow_{(1)}$ IDENT=EXP;	$\Rightarrow_{(1)}$ IDENT=EXP;
$\Rightarrow_{(2)}$ IDENT=EXP*EXP;	$\Rightarrow_{(2)}$ IDENT=EXP*EXP;
$\Rightarrow_{(2)}$ IDENT=EXP*EXP*EXP;	$\Rightarrow_{(3)}$ IDENT=NUMBER*EXP;
$\Rightarrow_{(3)}$ IDENT=NUMBER*EXP*EXP;	$\Rightarrow_{(2)}$ IDENT=NUMBER*EXP*EXP;
$\Rightarrow_{(4)}$ IDENT=NUMBER*IDENT*EXP;	$\Rightarrow_{(4)}$ IDENT=NUMBER*IDENT*EXP;
$\Rightarrow_{(3)}$ IDENT=NUMBER*IDENT*NUMBER;	$\Rightarrow_{(3)}$ IDENT=NUMBER*IDENT*NUMBER;
Rechtsableitung	
STMT	
$\Rightarrow_{(1)}$ IDENT=EXP;	
$\Rightarrow_{(2)}$ IDENT=EXP*EXP;	
$\Rightarrow_{(2)}$ IDENT=EXP*EXP*EXP;	
$\Rightarrow_{(3)}$ IDENT=EXP*EXP*NUMBER;	
$\Rightarrow_{(4)}$ IDENT=EXP*IDENT*NUMBER;	
$\Rightarrow_{(3)}$ IDENT=NUMBER*IDENT*NUMBER;	

Aufgabe 19:

Geben Sie eine CFG und ein Wort w an, so dass es für das Wort w zwei verschiedene Linksableitungen zu w gibt und beweisen Sie Ihre Aussage durch Angabe der beiden Linksableitungen zu w.

Lösung:

siehe Aufgabe 18 und …

G = ({S},{0},S,P)

mit P = {

S -> 0S0S0S (1)

S -> 0 (2)

S -> ε (3)

}

Zwei verschiedene Linksableitungen für w = 00000 …

$S \Rightarrow_{(1)} 0S0S0S \Rightarrow_{(1)} 00S0S0S0S0S$ 5x$\Rightarrow_{(3)}$ 000000

$S \Rightarrow_{(1)} 0S0S0S \Rightarrow_{(2)} 000S0S \Rightarrow_{(2)} 00000S \Rightarrow_{(2)}$ 000000

Aufgabe 20:

a) Geben Sie eine CFG an, die die Menge der nichtleeren Palindrome (Wort gleich dem gespiegelten Wort) über dem Alphabet Σ = {a,b,c} erzeugt.
Bsp.: w=abccba ist in der zu erzeugenden Menge.

b) Geben Sie eine Top-Down Analyserechnung des Top-Down Analyseautomaten zur CFG aus Teil a) für das Wort w=abccba an.

c) Geben Sie eine Top-Down Analyserechnung des Top-Down Analyseautomaten zur CFG aus Teil a) für das Wort w=abccba an, die erfolglos abbricht.

Lösung:

G = ({S},{a,b,c},S,P)

mit P = {

```
      S -> aSa,          (1)
      S -> bSb,          (2)
      S -> cSc,          (3)
      S -> aa,           (4)
      S -> bb,           (5)
      S -> cc,           (6)
      S -> a,            (7)
      S -> b,            (8)
      S -> c             (9)
```

}

Erfolgreiche Rechnung

=> (z,abccba,S,)
=> (z,abccba,aSa,1)
=> (z,bccba,Sa,1)
=> (z,bccba,bSba,12)
=> (z,ccba,Sba,12)
=> (z,ccba,ccba,126)
=> (z,cba,cba,126)
=> (z,ba,ba,126)
=> (z,a,a,126)
=> (z,ε,ε,126)

Erfolglose Rechnung

=> (z,abccba,S,)
=> (z,abccba,a,7)
=> (z,bccba,ε,7)

Übung 4

Aufgabe 21:

a) Berechnen Sie zu GSE und der Satzform NUMBER <Exp> ";" die $first_1$-, $first_2$-, $first_3$- und $first_4$-Mengen.

b) Ist die Satzform aus a) interessant zur Überprüfung der LL(k)-Eigenschaft von GSE?

c) Falls Sie b) mit "Nein" beantwortet haben, geben Sie eine Satzform an, die interessant zur Überprüfung der LL(k)-Eigenschaft von GSE?

Lösung:

a)

$first_1$ = {Number}

$first_2$ = {NumberNumber,
NumberIdent}

$first_3$ = {NumberNumber;,
NumberIdent;,
NumberNumber*,
NumberIdent*}

$first_4$ = {NumberNumber;,
NumberIdent;,
NumberNumber*Number,
NumberNumber*Ident,
NumberIdent*Number,
NumberIdent*Ident}

b) Die Satzform ist nicht interessant, da nur Satzformen interessant sind, die vom Startsymbol aus abgeleitet werden können.

c) Interessant wäre z.B.: IDENT "=" <Exp> "*" <Exp> ";"

Aufgabe 22:

Gegeben sei folgende Grammatik: G = ({S,A},{a,b},S,R) mit

R = { S -> ε,
S -> abA,

A -> Saa,

A -> b }.

a) Geben Sie drei der kürzesten Wörter an, die von G erzeugt werden.

b) Geben Sie jeweils eine Links-Ableitung für die drei Wörter aus a) an.

c) Geben Sie das minimale k an, so dass G LL(k) ist und begründen Sie Ihre Behauptung.

Lösung:

w = ϵ	S => ϵ
w = abb	S => abA => abb
w = abaa	S => abA => abSaa => abaa

Die Grammatik ist nicht LL(1), da a sowohl la(1), als auch la(2).

Die Grammatik ist LL(2), was ich anhand eines Beispiels verdeutlichen werde …

```
=> (z,ababaaaa,S,)
=> (z,ababaaaa,abA,2)
=> (z,babaaaa,bA,2)
=> (z,abaaaa,A,2)
```

Für die A-Regeln würde bereits ein LookAhead von eins ausreichen, da man von A aus nur Satzformen erzeugen kann, die mit einem b anfangen (zweite A-Regel) oder mit einem a (indirekt über erste A-Regel und dann eine S-Regel).

```
=> (z,abaaaa,Saa,23)
```

Sowohl mit der ersten (indirekt, Ausnahme w = ε) als auch mit der zweiten S-Regel kann man nur Satzformen erzeugen, die mit einem a anfangen. Aber nur mit der zweiten S-Regel kann man Satzformen erzeugen, deren zweites Zeichen ein b ist, also muss hier die zweite S-Regel angewendet werden ..

```
=> (z,abaaaa,abAaa,232)
=> (z,baaaa,bAaa,232)
=> (z,aaaa,Aaa,2323)
=> (z,aaaa,Saaaa,2323)        // s.o.
=> (z,aaaa,aaaa,23231)
…
```

Aufgabe 23:

Welche Sprachen werden von LL(0)-Grammatiken erzeugt?

Lösung:

Da man bei einer LL(0)-Grammatik immer ohne Lookahed entscheiden können muss, welche Regel angewendet werden soll, darf es bei der Regelauswahl erst gar keine Alternativen geben, d.h. es muss immer genau eine Regel anwendbar sein oder keine. Wenn zwei (oder mehr) Regeln anwendbar wären, dann wäre die Grammatik nicht LL(0), da man nicht ohne einen Lookahead von mindestens eins deterministisch entscheiden könnte, welche der zwei Regeln angewendet werden soll. **Daher können von LL(0)-Grammatiken nur Sprachen erzeugt werden, die höchstens ein Wort enthalten.** Bei Sprachen, die mehr als ein Wort enthalten, muss es mindestens zwei Regeln mit gleicher linker Regelseite geben, dies darf es bei LL(0)-Grammatiken jedoch nicht geben.

Beispiel

```
G = {{S},{a,b},S,P}
mit P = {
      S -> a,
      S -> b
}
```

Selbst bei dieser einfachen Typ-3-Sprache braucht man einen Lookahead von eins um deterministisch entscheiden zu können, welche Regel angewendet werden soll.

Aufgabe 24:

Zeigen Sie, dass GSE die LL(k)-Eigenschaft für jedes k>=0 verletzt.

Hinweis: Gehen Sie dabei von einem beliebigem k aus und zeigen Sie, dass die Bedingung aus der Definition von LL(k) verletzt ist.

Lösung:

Sie k beliebig aber fest.

Dann wird die Bedingung in der Definition der LL(k)-Grammatik durch die Ableitung der Wörter mit k/2+1 IDEs und k/2 "*" verletzt.

Siehe Beispiel für k == 3 auf Folien

Aufgabe 25:

Geben Sie eine CFG an, die linksrekursiv ist, aber bei der keine direkte Linksrekursion vorliegt und zeigen Sie, dass die Grammatik wirklich linksrekursiv ist.

Lösung:

G = ({S,A},{a,b},S,R)
mit R = {
S -> Aa
A -> Sb | ϵ
}

S => Aa => **S**ba => Aaba => **S**baba => …

Aufgabe 26:

Eliminieren Sie die direkte Linksrekursion in der Grammatik GSE.

Lösung:

<STMT> -> IDENT "=" <EXP> ”;“
<EXP> -> (”NUMBER“|”IDENT”)<PRC>
<PRC> -> “*”<EXP>
<PRC> -> ϵ

Übung 5

Aufgabe 27:

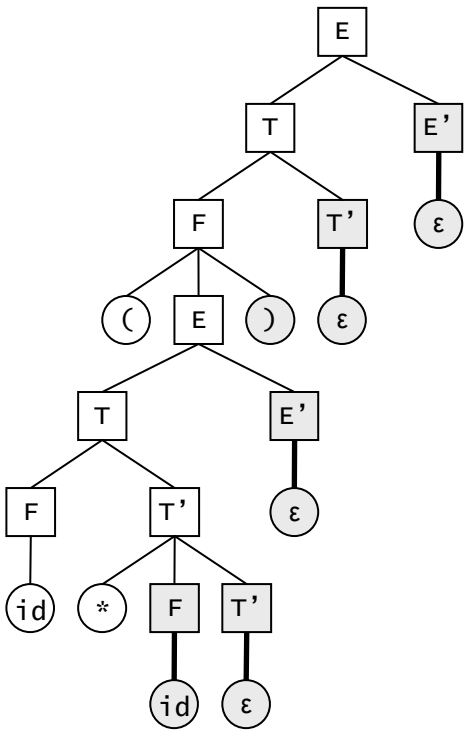
a) Führen Sie die Beispielanalyse (deterministisch) auf Folie "Syntax-Analyse 33" zu Ende.

b) Geben Sie den Syntaxbaum an, der der Ableitung in a) entspricht.

c) Geben Sie eine Rechtsableitung für den Syntaxbaum aus b) an.

Linksableitung:

```
=> ( z , id ) ,  F T' E' ) T' E'   , 1471485      )
=> ( z , id ) ,  id T' E' ) T' E'  , 14714858     )
=> ( z , )    ,  T' E' ) T' E'     , 14714858     )
=> ( z , )    ,  E' ) T' E'        , 147148586    )
=> ( z , )    ,  ) T' E'           , 1471485863   )
=> ( z ,      ,  T' E'             , 1471485863   )
=> ( z ,      ,  E'                , 14714858636  )
=> ( z ,      ,                    , 147148586363 )
```



Rechtsableitung:

```
      E
=>    TE'
=>    T
=>    FT'
=>    F
=>    (E)
=>    (TE')
=>    (T)
=>    (FT')
=>    (F*FT')
=>    (F*F)
=>    (F*id)
=>    (id*id)
```

Aufgabe 28:

a) Gibt es ein k, so dass die Grammatik aus Aufgabe 20 a) eine LL(k)-Grammatik ist? Wenn ja, geben Sie das minimale an.

b) Geben Sie eine CFG an, die die Menge der nichtleeren Palindrome (Wort gleich dem gespiegelten Wort) über dem Alphabet Σ = {a,b,c} mit Mittelsymbol "-" erzeugt.
Bsp.: w=abc-cba ist in der zu erzeugenden Menge.

c) Gibt es ein k, so dass die Grammatik aus Aufgabe b) eine LL(k)-Grammatik ist? Wenn ja, geben Sie das minimale an.

Lösung:

Die Grammatik aus Aufgabe 20 ist nicht LL(k). Es kann keinen Lookahead bestimmter Länge geben, mit der man eindeutig bestimmen kann, welche Regel angewendet werden soll. Ein einfaches Beispiel sind die Palindrome, die ausschließlich aus einem einzigen Zeichen bestehen. Bei der Satzform …

Saaa …

kann man nicht entscheiden, ob S nach aa oder nach aSa abgeleitet werden soll, da man nicht vorhersehen kann, wieviele a's noch folgen werden bzw. wann die Mitte erreicht ist.

Grammatik für Palindrome mit ausgezeichnetem Mittesymbol m:

G = ({S},{a,b,c,-},S,P)

mit P = {

S -> aSa, (1)
S -> bSb, (2)
S -> cSc, (3)
S -> -, (4)

}

Diese Grammatik ist LL(1), da man durch Vorauslesen von einem Zeichen entweder ein a, b oder c liest oder ein -. Beim Vorauslesen von (a,b,c) müssen die Regeln (1,2,3) angewendet werden. Beim Vorauslesen des Mittesymbols - muss die Regel 4 angewendet werden.

la_1 = fi(aSafo(S)) = {a}
la_2 = fi(bSbfo(S)) = {b}
la_3 = fi(cScfo(S)) = {c}
la_4 = fi(-fo(S)) = {-}

$la_i \cap la_j = \varnothing$ für $i \neq j$

Aufgabe 29:

a) Geben Sie eine erfolgreiche Bottom-Up Analyserechnung des Bottom-Up Analyseautomaten zur CFG G'_{AE} von Folie "Syntax-Analyse 28" für das Wort w="(" id "*" id ")" an.

b) Geben Sie eine nicht erfolgreiche Bottom-Up Analyserechnung des Bottom-Up Analyseautomaten zur CFG G'_{AE} für das Wort w="(" id "*" id ")" an.

Ich verwende die Notation aus Indermark, da ich diese im Zusammenhang mit der Bottom-Up Analyse-Rechnung wesentlich leichter zu verstehen fand.

(Stapel, Eingabe, Ausgabe) // Stackspitze liegt rechts

Erfolgreiche Rechnung

```
=> (z , “(“ id * id “)“ , $              ,             )
=> (z , id * id “)“     , “(“ $          ,             )
=> (z , * id “)”        , id “(“ $       ,             )
=> (z , * id “)”        , F “(“ $        , 8           )
=> (z , id “)”          , * F “(“ $      , 8           )
=> (z , “)”             , id * F “(“ $   , 8           )
=> (z , “)”             , F * F “(“ $    , 88          )
=> (z , “)”             , T’ F * F “(“ $ , 886         )
=> (z , “)”             , T’ F “(“ $     , 8865        )
=> (z , “)”             , T “(“          , 88654       )
=> (z , “)”             , E’ T “(“       , 886543      )
=> (z , “)”             , E “(“          , 8865431     )
=> (z ,                 , “)“ E “(“      , 8865431     )
=> (z ,                 , F              , 88654317    )
=> (z ,                 , T’ F           , 886543176   )
=> (z ,                 , T              , 8865431764  )
=> (z ,                 , E’ T           , 88654317643 )
=> (z ,                 , E              , 886543176431 )
```

Erfolglose Rechnung

```
=> (z , “(“ id * id “)“ ,                  , )
=> (z , id * id “)“     , “(“              , )
=> (z , * id “)”        , id “(“           , )
=> (z , id “)”          , * id “(“         , )
=> (z , “)”             , id * id “(“      , )
=> (z ,                 , “)” id * id “(“  , )
```

Kein Shiften möglich; außer epsilon-Regeln (die auch nicht zum Ziel führen) keine Regeln anwendbar.

Aufgabe 30:

a) Geben Sie eine Grammatik für Mini-Java an, die bis auf folgende Ausnahme LL(1) ist: Die LL(1)-Eigenschaft darf lediglich für die Regeln zum "else" verletzt werden.

b) Zeigen Sie, dass Ihre Grammatik aus Teil a) bis auf die eine Ausnahme LL(1) ist.

siehe Datei "CB06A30 Lösung"

$la_{30} \cap la_{31}$ = {"else"} => G ist nicht LL(1)

Aufgabe 31:

a) Gegeben sei folgende Grammatik G = ({S,T},{a, nil, (,), #},S,R) mit

R = { S -> a (1)
S -> nil (2)
S -> (T) (3)
T -> S#T (4)
T -> S (5) }

Führen Sie die Links-Faktorisierung für G durch.

b) Bestimmen Sie die look-ahead-Mengen der Regeln für die in a) erhaltene Grammatik. Ist sie aus LL(1)?

Der besseren Lesbarkeit halber habe ich das , $\in \Sigma$ durch # ersetzt.

S -> a (1)
S -> nil (2)
S -> (T) (3)
T -> SA (4)
A -> # T (5)
A -> ϵ (6)

R	fi	fo
S	a , nil , (	ϵ,), #
T	a , nil , (	)
A	# , ϵ	)

la_1 = la(S,a) = fi(a fo(S)) = {a}
la_2 = la(S,nil) = fi(nil fo(S)) = {nil}

la_3 = la(S,(T)) = fi((T) fo(S)) = {(}
la_4 = la(T,SA) = fi(SA fo(T)) = {a,nil,(}
la_5 = la(A,#T) = fi(#T fo(A)) = {#}
la_6 = la(A, ϵ) = fi(ϵ, fo(A)) = {)}

$la_1 \cap la_2 = \varnothing$

$la_1 \cap la_3 = \varnothing$

$la_2 \cap la_3 = \varnothing$

$la_5 \cap la_6 = \varnothing$

Die Grammatik ist vom Typ LL(1).

Übung 6

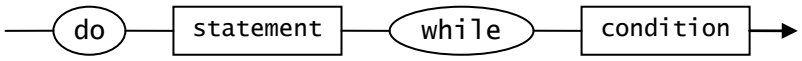
Aufgabe 32:

a) Erweitern Sie die Syntax von Mini-Java um die folgenden beiden Java-Konstrukte: do-while-Schleife & case-Statement

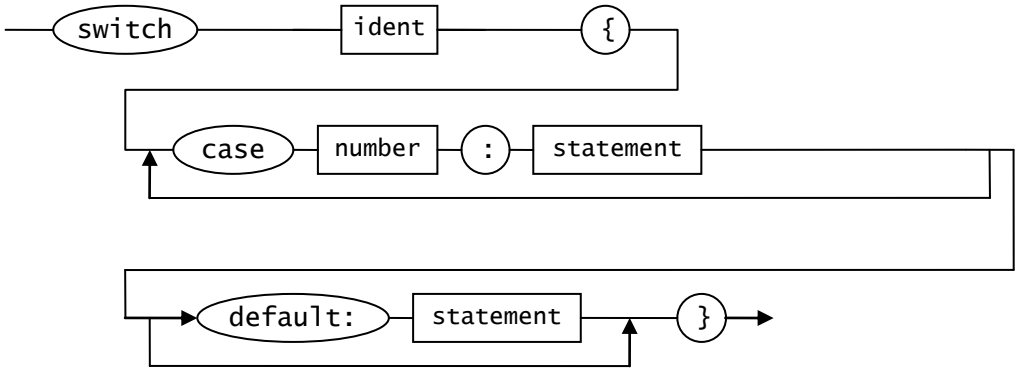
b) Geben Sie ein möglichst einfaches Beispielprogramm an, das die beiden in a) beschriebenen Java-Konstrukte enthält.

Lösung:

Endgesteuerte Schleife



Switch-Statement



<u>Programmbeispiel</u>:

```
int i=1,out;
{
  do {
      switch i {
            case 1: out=1;
            case 2: out=2;
      }
      i=i+1;
  } while i < 2
  print(out);
}
```

Aufgabe 33:

Programmieren Sie einen Parser für Mini-Java-Exp mittels rekursivem Abstieg ohne la-Mengen.

Lösung:

```
void programm() throws ParseException {
	constDecl();
	varDecl();
	statement();
}

void constDecl() throws ParseException {
	if(sym == “final“) { nextSym();
		if(sym == “int“) { nextSym();
			constZuw();
			constList();
			if(sym == “;“) {
				nextSym();
				return;
			}
		}
	} else return;
	throw new ParseException();
}

void constZuw() throws ParseException {
	if(sym == “ident“) { nextSym();
		if(sym == “=“) { nextSym();
			if(sym == “number“) { nextSym();
				return;
			}
		}
	}
	throw new ParseException();
}

void constList() throws ParseException {
	if(sym == “,“) {
		nextSym();
		constZuw();
		constList();
	}
}
void varDecl() throws ParseException {
	if(sym == “int“) { nextSym();
		varZuw();
```

```
                varList();
                if(sym == ";") {
                        nextSym();
                        return;
                }
        } else return;
        throw new ParseException();
}

void varZuw() throws ParseException {
        if(sym == "ident") { nextSym();
                if(sym == "=") { nextSym();
                        if(sym == "number") { nextSym();
                                return;
                        }
                } else return;
        }
        throw new ParseException();
}

void varList() throws ParseException {
        if(sym == ",") {
                nextSym();
                varZuw();
                varList();
        }
}

void statement() throws ParseException {
        if(sym == "print") { nextSym();
                if(sym == "(") { nextSym();
                        E();
                        if(sym == ")") {
                                nextSym();
                                return;
                        }
                }
        }
        throw new ParseException();
}
```

Methoden für die Grammatik für Ausdrücke analog zu Syntax-Analyse-Folie-Nr. 36

Aufgabe 34:

a) Geben Sie ein Mini-Java-Exp-Programm an mit dem gleichen Umfang wie das Programm auf Folie "Zwischencodeerz. 6".

b) Berechnen Sie schrittweise die Semantik Ihres Programms aus a).

Lösung:

```
final int a = 5, b = 10;
int x = 5, y = 10;
print((a*x+b*y));
```

```
M(P) = E((a*x+b*y),DV(int x=5, y=10;,DC(final int a=5, b=10;, rho0),sig0))
     = E((a*x+b*y),DV(int x=5, y=10;,[a/5,b/10],sig0)
     = E((a*x+b*y),[a/5,b/10,x/add1,y/add2],[add1/5,add2/10])
     = E(a*x+b*y,[a/5,b/10,x/add1,y/add2],[add1/5,add2/10])
     = E(a*x,[a/5,b/10,x/add1,y/add2],[add1/5,add2/10])
       + E(b*y,[a/5,b/10,x/add1,y/add2],[add1/5,add2/10])
     = E(a,[a/5,b/10,x/add1,y/add2],[add1/5,add2/10])
       * E(x,[a/5,b/10,x/add1,y/add2],[add1/5,add2/10])
       + E(b,[a/5,b/10,x/add1,y/add2],[add1/5,add2/10])
       * E(y,[a/5,b/10,x/add1,y/add2],[add1/5,add2/10])
     = 5 * 5 + 10 * 10
     = 125
```

Aufgabe 35:

a) Geben Sie die Übersetzung des Mini-Java-Exp-Programms aus Aufgabe 34 in ZE-Code an.

b) Führen Sie die Rechnung des ZE-Codes aus a) schrittweise durch und vergleichen Sie das Ergebnis mit dem Ergebnis aus Teil b) der vorherigen Aufgabe.

Lösung:

```
update(final int a=5, b=10; int x=5,y=10; st0, h0)
= ([a/(const,5),b/(const,10),x/(var,1),y/(var,2)],[1/5,2/10])
```

mit `st = [a/(const,5),b/(const,10),x/(var,1),y/(var,2)]` folgt …

```
  exptrans((a*x+b*y),st)
= exptrans(a*x+b*y,st)
= exptrans(a*x,st)
```

```
  exptrans(b*y,st)
  ADD
= exptrans(a,st)
  exptrans(x,st)
  MULT
  exptrans(b,st)
  exptrans(y,st)
  MULT
  ADD
= LIT 5     (1)
  LOD 1     (2)
  MULT      (3)
  LIT 10    (4)
  LOD 2     (5)
  MULT      (6)
  ADD       (7)

=> (1 , []          , [1/5,2/10])
=> (2 , [5]         , [1/5,2/10])
=> (3 , [5,5]       , [1/5,2/10])
=> (4 , [25]        , [1/5,2/10])
=> (5 , [10,25]     , [1/5,2/10])
=> (6 , [10,10,25]  , [1/5,2/10])
=> (7 , [100,25]    , [1/5,2/10])
=> (8 , [125]       , [1/5,2/10])
```

Aufgabe 36:

Im Gegensatz zum ZE-Code stehen Operationen, wie ADD und MULT beim M32-Assembler nicht unmittelbar auf dem Stack zur Verfügung.

Wie kann man diese Operationen analog zum ZE-Code im M32-Assembler simulieren?

Lösung:

- obersten beiden Operanden auf Stack nach R0 und R1.
- Operation auf R0 und R1 ausführen
- Ergebnis nach R0
- Ergebnis von R0 auf Stack

Übung 7

Aufgabe 37:

Berechnen Sie die Semantik M(P) Ihres Mini-Java-Programms aus Praktikum Versuch 1 Aufgabe 1.

Lösung:

```
final int n = 1;
int fib0 = 0, fib1 = 1, fib2 = 0, i = 1;
{
        while i<=n {
                fib2 = fib1;
                fib1 = fib0;
                fib0 = fib1 + fib2;
                i = i + 1;
        } // block2
        print(fib0);
} // block1
```

Bei der Berechnung der Semantik des Programms habe ich manchmal mehrere Schritte zu einem einzigen zusammengefügt, so dass die Rechnung übersichtlicher bleibt. Desweiteren verwende ich für den Zustandsraum ausschließlich die Variable sig, die ich ggf. aktualisiere.

M(P)

= **S**(block1,**DV**(int fib0 = 0, fib1 = 1, fib2 = 0, i = 1;, **DC**(final int n = 2;,*rho0*),*sig0*))
= **S**(block1,**DV**(int fib0 = 0, fib1 = 1, fib2 = 0, i = 1;,[n/2],*sig0*))
= **S**(block1,[n/2,fib0/add0,fib1/add1,fib2/add2,i/add3],[add0/0,add1/1,add2/0,add3/1])

Abkürzungen:

rho = [n/2,fib0/add0,fib1/add1,fib2/add2,i/add3]
sig = [add0/0,add1/1,add2/0,add3/1]

= **S**(print(fib0),*rho*,**S**(while (i<=n) block2, *rho*, *sig*))

= **S**(print(fib0),*rho*,IF **C**(i<=n,*rho*,*sig*) == true
THEN **S**(while (i<=n) block2,*rho*,**S**(block2,*rho*,*sig*))
ELSE *sig*)

= **S**(print(fib0),*rho*,IF **E**(i,*rho*,*sig*) <= **E**(n,*rho*,*sig*) == TRUE THEN … ELSE …)

= **S**(print(fib0),*rho*,IF 1 **<=** 1 == TRUE THEN … ELSE …)

= **S**(print(fib0),*rho*, **S**(while (i<=n) block2,*rho*, **S**(block2,*rho*,*sig*)))

= **S**(print(fib0),*rho*, **S**(while (i<=n) block2,*rho*, **S**(i=i+1;,*rho*,
S(fib0=fib1+fib2;,*rho*,
S(fib1=fib0;,*rho*,
S(fib2=fib1;,*rho*,*sig*))))))

= **S**(print(fib0),*rho*, **S**(while (i<=n) block2,*rho*, **S**(i=i+1;,*rho*,
S(fib0=fib1+fib2;,*rho*,
S(fib1=fib0;,*rho*,
sig[add2/**E**(fib1,*rho*,*sig*)])))))

Aktualisierung: *sig* = [add0/0,add/1,add2/1,add3/1]

= **S**(print(fib0),*rho*, **S**(while (i<=n) block2,*rho*, **S**(i=i+1;,*rho*,
S(fib0=fib1+fib2;,*rho*,
sig[add1/**E**(fib0;,*rho*,*sig*)])))))

Aktualisierung: *sig* = [add0/0,add1/0,add2/1,add3/1]

= **S**(print(fib0),rho, **S**(while (i<=n) block2,*rho*, **S**(i=i+1;,*rho*,
sig[fib0/**E**(fib1+fib2;,*rho*,*sig*)])))

mit **E**(fib1+fib2,rho,sig) = **E**(fib1,rho,sig) + **E**(fib2,rho,sig) = 0+1 = 1 folgt …
Aktualisierung: *sig* = [add0/1,add1/0,add2/1,add3/1]

= **S**(print(fib0),*rho*, **S**(while (i<=n) block2, *rho*, **S**(i=i+1;,*rho*,*sig*)))

= **S**(print(fib0),*rho*, **S**(while (i<=n) block2, *rho*, *sig*[add3/**E**(i+1,*rho*,*sig*)]))

mit **E**(i+1,*rho*,*sig*) = **E**(i,*rho*,*sig*) + **E**(1,*rho*,*sig*) = 1+1 = 2 folgt …
Aktualisierung: *sig* = [add0/1,add1/0,add2/1,add3/2]

= **S**(print(fib0,*rho*, **S**(while (i<=n) block2, *rho*, *sig*))

= **S**(print(fib0),*rho*,IF **C**(i<=n,*rho*,*sig*) == true
THEN **S**(while (i<=n) block2,*rho*,**S**(block2,*rho*,*sig*))
ELSE *sig*)

= **S**(print(fib0),*rho*,IF **E**(i,*rho*,*sig*) <= **E**(n,*rho*,*sig*) == TRUE THEN … ELSE …)

= **S**(print(fib0),*rho*,IF 2 <= 1 == TRUE THEN … ELSE …)

= **S**(print(fib0),*rho*,*sig*)

= [add0/1,add1/0,add2/1,add3/2]

Aufgabe 38:

a) Erweitern Sie die Definition der Semantik (Statement-Semantik S) von Mini-Java um die beiden in Aufgabe 32 beschriebenen Java-Konstrukte.

b) Berechnen Sie die Semantik Ihres Programms aus Aufgabe 32 Teil b)

Lösung:

a)

S(do stmt while cond,*rho*,*sig*) = S(while cond stmt, *rho*, S(stmt,*rho*,*sig*))

Für das switch mit default-Zweig benötigt man zur Berechnung noch ein semantisches ELSE, da man sonst den default-Zweig nicht vernünftig berechnen kann …

S(switch ident { case $number_1$: $stmt_1$ break;… case $number_n$: $stmt_n$ break; default: $stmt_d$ },*rho*,*sig*) =

IF **C**(ident == $number_1$, rho, sig) == true THEN **S**(stmt1,rho,sig)
ELSE IF **C**(ident == $number_n$, rho, sig) == true THEN **S**($stmt_n$,rho,sig)
ELSE S($stmt_d$,rho,sig)

Für switch ohne default-Zweig genügen die bisherigen Semantik-Regeln …

S(switch ident { case $number_1$: $stmt_1$ break; case $number_n$: $stmt_n$ },*rho*,*sig*) =

IF **C**(ident == $number_1$, *rho*, *sig*) == true THEN **S**($stmt_1$,*rho*,*sig*)
IF **C**(ident == $number_n$, *rho*, *sig*) == true THEN **S**($stmt_n$,*rho*,*sig*)

b)

```
int i=1,out;
{ // block1
  do { // block2
       switch i { // block3
              case 1: out=1; break;
              case 2: out=2; break;
              default: out=0;
       }
       i=i+1;
  } while i < 2
  print(out);
}
```

M(P)

= **S**(block1,**DV**(int i=1, out;, *rho0*,*sig0*))

= **S**(block1,[i/add1,out/add2],[add1/1,add2/0])

Abkürzungen:

rho = [i/add1,out/add2]
sig = [add1/1,add2/0]

= **S**(print(out),*rho*,**S**(do block2 while i<2,*rho*,*sig*))
= **S**(print(out),*rho*,**S**(while i<2 block2,*rho*,**S**(block2,*rho*,*sig*)))
= **S**(print(out),*rho*,**S**(while i<2 block2,*rho*,**S**(i=i+1;,*rho*,S(switch block3,*rho*,*sig*))))

Nebenrechnung:

IF **C**(ident == 1 , *rho*, *sig*) == true THEN **S**(out=1;,*rho*,*sig*)
ELSE IF **C**(ident == 2 , *rho*, *sig*) == true THEN **S**(out=2;,*rho*,*sig*)
ELSE S(out=0 ,*rho*,*sig*)

=> **S**(out=1;,*rho*,*sig*) = *sig*[add2/**E**(1,*rho*,*sig*)] = *sig*[add2/1]

Aktualisierung: *sig* = [add1/1,add2/1]

= **S**(print(out),rho,**S**(while i<2 block2,*rho*,**S**(i=i+1;,*rho*,*sig*)))
= **S**(print(out),rho,**S**(while i<2 block2,*rho*,*sig*[add1/**E**(i+1,rho,*sig*)]))

mit **E**(i+1,*rho*,*sig*) = **E**(i,*rho*,*sig*) + **E**(1,*rho*,*sig*) = 1+1 = 2 folgt …
Aktualisierung: *sig* = [add1/2,add1/1]

= **S**(print(out),*rho*,**S**(while i<2 block2, *rho*, *sig*))
= **S**(print(out),*rho*,IF C(i<2,*rho*,*sig*) THEN **S**(while i<2 block2,*rho*,**S**(block2,*rho*,*sig*))
ELSE *sig*)

mit **C**(i<2,*rho*,*sig*) = **E**(i,*rho*,*sig*) < **E**(2,*rho*,*sig*) = 2 < 2 = false folgt …

= **S**(print(out),*rho*,*sig*)
= [add1/2,add2/1]

Übung 8

Aufgabe 39:

a) Geben Sie die Übersetzung des Mini-Java-Programms aus Aufgabe 37 in Z-Mini-Java-Code an.

b) Führen Sie die Rechnung des Z-Mini-Java-Codes aus a) schrittweise durch und vergleichen Sie das Ergebnis mit dem Ergebnis aus Aufgabe 37.

```
update(final int n=1; int fib0=0,fib1=1,fib2=0,i=1;st0,h0) = (st,h)  mit …
       st = [n/(const,1),fib0/(var,0),fib1/(var,1),fib2/(var,2),i/(var,3)]
       h = [0/0,1/1,2/0,3/1]

       cmdtrans(block1,st)

=      cmdtrans(while i<=n block2,st)
       cmdtrans(print(fib0);,st)

= a0:  condtrans(i<=n,st)
       JMC a1
       cmdtrans(block2,st)
       JMP a0
  a1:  exptrans(fib0,st)
       POP

= a0:  exptrans(i,st)
       exptrans(n,st)
       LE
       JMC a1
       cmdtrans(fib2 = fib1;, st)
       cmdtrans(fib1 = fib0;, st)
       cmdtrans(fib0 = fib1+fib2;, st)
       cmdtrans(i = i+1, st)
       JMP a0
  a1:  LOD 0
       POP

= a0:  LOD 3
       LIT 1
       LE
       JMC a1
       exptrans(fib1, st)
       STO 2
       exptrans(fib0, st)
       STO 1
       exptrans(fib1+fib2, st)
```

```
        STO 0
        exptrans(i+1, st)
        STO 3
        JMP a0
  a1:   LOD 0
        POP

= a0:   LOD 3
        LIT 1
        LE
        JMC a1
        LOD 1
        STO 2
        LOD 0
        STO 1
        exptrans(fib1, st)
        exptrans(fib2, st)
        ADD
        STO 0
        exptrans(i, st)
        exptrans(1, st)
        ADD
        STO 3
        JMP a0
  a1:   LOD 0
        POP

= a0:   LOD 3
        LIT 1
        LE
        JMC a1
        LOD 1
        STO 2
        LOD 0
        STO 1
        LOD 1
        LOD 2
        ADD
        STO 0
        LOD 3
        LIT 1
        ADD
        STO 3
        JMP a0
  a1:   LOD 0
        POP
```

```
1:   LOD 3
2:   LIT 1
3:   LE
4:   JMC 18
5:   LOD 1
6:   STO 2
7:   LOD 0
8:   STO 1
9:   LOD 1
10:  LOD 2
11:  ADD
12:  STO 0
13:  LOD 3
14:  LIT 1
15:  ADD
16:  STO 3
17:  JMP 1
18:  LOD 0
19:  POP
20:
```

```
(1,  []    , [0/0,1/1,2/0,3/1])
(2,  [1]   , [0/0,1/1,2/0,3/1])
(3,  [1,1] , [0/0,1/1,2/0,3/1])
(4,  [1]   , [0/0,1/1,2/0,3/1])
(5,  []    , [0/0,1/1,2/0,3/1])
(6,  [1]   , [0/0,1/1,2/0,3/1])
(7,  []    , [0/0,1/1,2/1,3/1])
(8,  [0]   , [0/0,1/1,2/1,3/1])
(9,  []    , [0/0,1/0,2/1,3/1])
(10, [0]   , [0/0,1/0,2/1,3/1])
(11, [0,1] , [0/0,1/0,2/1,3/1])
(12, [1]   , [0/0,1/0,2/1,3/1])
(13, []    , [0/1,1/0,2/1,3/1])
(14, [1]   , [0/1,1/0,2/1,3/1])
(15, [1,1] , [0/1,1/0,2/1,3/1])
(16, [2]   , [0/1,1/0,2/1,3/1])
(17, []    , [0/1,1/0,2/1,3/2])
(1,  []    , [0/1,1/0,2/1,3/2])
(2,  [2]   , [0/1,1/0,2/1,3/2])
(3,  [1,2] , [0/1,1/0,2/1,3/2])
(4,  [0]   , [0/1,1/0,2/1,3/2])
(18, []    , [0/1,1/0,2/1,3/2])
(19, [1]   , [0/1,1/0,2/1,3/2])
(20,       , [0/1,1/0,2/1,3/2])
```

Aufgabe 40:

a) Erweitern Sie die Definition der Übersetzung von Mini-Java in Z-Mini-Java-Code um die beiden in Aufgabe 32 beschriebenen Java-Konstrukte.

b) Führen Sie die Übersetzung des Mini-Java-Programms aus Aufgabe 32 in Z-Mini-Java-Code durch.

c) Führen Sie die Rechnung des Z-Mini-Java-Codes aus b) schrittweise durch und vergleichen Sie das Ergebnis mit dem Ergebnis aus Aufgabe 38 b).

Lösung:

Übersetzung für switch-Statement

```
cmdtrans(switch ident { case number1 : stmt1 case numbern : stmtn default: stmtd }) =

            exptrans(ident,st)
            exptrans(number1 ,st)
            NE
            JMC a(next+1)
            exptrans(ident,st)
            exptrans(numbern ,st)
            NE
            JMC a(next+n)
            cmdtrans(stmtd ,st)          // Fehlt der default-Zweig, entfällt diese Zeile
            JMP a(next+n+1)
a(next+1):  cmdtrans(stmt1 ,st)
            JMP a(next+n+1)
a(next+n):  cmdtrans(stmtn ,st)
            JMP a(next+n+1)
a(next+n+1):
```

Übersetzung für do-while-Statement

```
cmdtrans(do stmt while cond, st) =  a(next):   cmdtrans(stmt,st)
                                               condtrans(cond,st)
                                               JMC a(next+1)
                                               JMP a(next)
                                    a(next+1):
```

```
update(int i=1,out,st0,h0) = (st,h)  mit ...
       st = [i/(var,0),out/(var,1)]
       h = [0/1,1/0]

=      cmdtrans(block1,st)

=      cmdtrans(do block2 while i<2),st)
       cmdtrans(print(out),st)

=  a0: cmdtrans(block2,st)
       condtrans(i<2,st)
       JMC a1
       JMP a0
   a1: exptrans(out,st)
       POP

=  a0: cmdtrans(switch block3,st)
       cmdtrans(i=i+1,st)
       exptrans(i,st)
       exptrans(2,st)
       LT
       JMC a1
       JMP a0
   a1: LOD 1
       POP

=  a0: exptrans(i,st)
       exptrans(1,st)
       NE
       JMC a2
       exptrans(i,st)
       exptrans(2,st)
       NE
       JMC a3
       cmdtrans(out=0,st)
       JMP a4
   a2: cmdtrans(out=1 ,st)
       JMP a4
   a3: cmdtrans(out=2 ,st)
       JMP a4
   a4: cmdtrans(i=i+1,st)
       exptrans(i,st)
       exptrans(2,st)
       LT
       JMC a1
       JMP a0
   a1: LOD 1
       POP
```

```
=  a0: LOD 0
       LIT 1
       NE
       JMC a2
       LOD 0
       LIT 2
       NE
       JMC a3
       LIT 0
       STO 1
       JMP a4
   a2: LIT 1
       STO 1
       JMP a4
   a3: LIT 2
       STO 2
       JMP a4
   a4: LOD 0
       LIT 1
       ADD
       STO 0
       LOD 0
       LIT 2
       LT
       JMC a1
       JMP a0
   a1: LOD 1
       POP
```

```
1:   LOD 0
2:   LIT 1
3:   NE
4:   JMC 12
5:   LOD 0
6:   LIT 2
7:   NE
8:   JMC 14
9:   LIT 0
10:  STO 1
11:  JMP 18
12:  LIT 1
13:  STO 1
14:  JMP 18
15:  LIT 2
16:  STO 2
17:  JMP 18
```

```
18:  LOD 0
19:  LIT 1
20:  ADD
21:  STO 0
22:  LOD 0
23:  LIT 2
24:  LT
25:  JMC 27
26:  JMP 1
27:  LOD 1
28:  POP


(1,  []    , [0/1,1/0])
(2,  [1]   , [0/1,1/0])
(3,  [1,1] , [0/1,1/0])
(4,  [0]   , [0/1,1/0])
(12, []    , [0/1,1/0])
(13, [1]   , [0/1,1/0])
(14, []    , [0/1,1/1])
(18, []    , [0/1,1/1])
(19, [1]   , [0/1,1/1])
(20, [1,1] , [0/1,1/1])
(21, [2]   , [0/1,1/1])
(22, []    , [0/2,1/1])
(23, [2]   , [0/2,1/1])
(24, [2,2] , [0/2,1/1])
(25, [0]   , [0/2,1/1])
(27, []    , [0/2,1/1])
(28, [1]   , [0/2,1/1])
(28, []    , [0/2,1/1])
```

Aufgabe 41:

Die große Ackermannfunktion ist wie auf Folie "Mini-Java-FunProc 2" definiert:

ack: nat x nat -> nat

ack (a,0) = 1, für alle nat. Zahlen a >= 0

ack (0,1) = 2

ack (0, b+2) = b + 4, für alle nat. Zahlen b >= 0

ack (a+1,b+1) = ack (a,ack(a+1,b)), a,b >= 0

Berechnen Sie die Funktionswerte:

ack(0,0), ack(0,1), ..., ack(0,14)

ack(1,0), ack(1,1), ..., ack(1,8)

ack(2,0), ack(2,1), ..., ack(2,4)

ack(3,0), ack(3,1), ..., ack(3,3)

und tragen Sie sie in folgendes Schema ein:

A / B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0															
1															
2															
3															

Lösung: **Info I A28.**

A / B	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	1	2	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	4	6	8	10	12	14	16						
2	1	2	4	8	16										
3	1	2	4	16											

```
ack(a,0) = 1                        für a >= 0
ack(0,1) = 2
ack(0,b) = b+2                      für b >= 2
ack(a,b) = ack(a-1,ack(a,b-1))      für a,b >= 1
```

Aufgabe 42:

Geben Sie ein M32-Assembler-Programm zur Implementierung der großen Ackermannfunktion an, wobei Sie die Eingabewerte als Konstanten im Programm definieren.

Lösung:

Programm, das sich aus Übersetzung der rekursiven Implementierung ergibt, oder Programm, dass sich aus Übersetzung der iterativen Implementierung ergibt.

```
MOV  SSP,0FFFFFFFFH
JMP  main0
HALT
```

```
main0: MOV  TOS,0
       MOV  R0,SSP
       MOV  TOS,R0
       MOV  TOS,P0
       MOV  TOS,03H
       MOV  TOS,02H
       JMP  ack2
P0:    PRN  TOS
       HALT


ack2:  MOV  R0,SSP
       MOV  TOS,01H(R0)
       MOV  TOS,0H
       MOV  R2,TOS
       MOV  R1,TOS
       CMP  R1,R2
       JNZ  P1
       MOV  TOS,01H
       MOV  R1,TOS
       MOV  SSP,R0
       ADD  SSP,02H
       MOV  R2,TOS
       MOV  R0,TOS
       MOV  TOS,R1
       JMP  R2
P1:    MOV  TOS,02H(R0)
       MOV  TOS,0H
       MOV  R2,TOS
       MOV  R1,TOS
       CMP  R1,R2
       JNZ  P3
       MOV  TOS,01H(R0)
       MOV  TOS,01H
       MOV  R2,TOS
       MOV  R1,TOS
       CMP  R1,R2
       JNZ  P4
       MOV  TOS,02H
       MOV  R1,TOS
```

```
        MOV  SSP,R0
        ADD  SSP,02H
        MOV  R2,TOS
        MOV  R0,TOS
        MOV  TOS,R1
        JMP  R2
P4:     MOV  TOS,01H(R0)
        MOV  TOS,02H
        MOV  R1,TOS
        MOV  R2,TOS
        ADD  R2,R1
        MOV  TOS,R2
        MOV  R1,TOS
        MOV  SSP,R0
        ADD  SSP,02H
        MOV  R2,TOS
        MOV  R0,TOS
        MOV  TOS,R1
        JMP  R2
P5:
P3:     MOV  TOS,R0
        MOV  TOS,P7
        MOV  TOS,02H(R0)
        MOV  TOS,01H
        MOV  R1,TOS
        MOV  R2,TOS
        SUB  R2,R1
        MOV  TOS,R2
        MOV  TOS,R0
        MOV  TOS,P8
        MOV  TOS,02H(R0)
        MOV  TOS,01H(R0)
        MOV  TOS,01H
        MOV  R1,TOS
        MOV  R2,TOS
        SUB  R2,R1
        MOV  TOS,R2
        JMP  ack2
P8:     JMP  ack2
P7:     MOV  R1,TOS
        MOV  SSP,R0
        ADD  SSP,02H
```

```
        MOV  R2,TOS
        MOV  R0,TOS
        MOV  TOS,R1
        JMP  R2
P6:
P2:     HALT
```

Aufgabe 43:

Geben Sie ein Mini-Java-Programm zur Implementierung der großen Ackermannfunktion an, wobei Sie die Eingabewerte als Konstanten im Programm definieren.

Lösung:

Für die Berechnung der Ackermann-Funktion benötigt man entweder einen Stack oder ein Array zur Speicherung von berechneten Werten (beides für iterative Lösung notwendig) oder Funktionen (für eine rekursive Lösung), beides steht im reinen MINI-JAVA nicht zur Verfügung, weshalb meiner Meinung nach eine Lösung dieser Aufgabe nicht möglich ist.

Übung 9

Aufgaben 44 - 48: keine Lösungen vorhanden, da Probeklausur

Übung 10

Aufgabe 49:

Geben Sie ein Mini-Java-FunProc-Programm zur Implementierung der großen Ackermannfunktion an, wobei Sie die Eingabewerte als Konstanten im Programm definieren.

Lösung:

```
final int a = ..., b = ...;
func ack(int a, int b) {
      int ret = 0;

      if b == 0 {
            ret = 1;
      } else if a == 0 {
            if b == 1 ret = 2;
                  else ret = b + 2;
      } else {
            ret = ack(a-1,ack(a,b-1));
      }
      return ret;
}
print(ack(a,b));
```

Aufgabe 50:

a) Geben Sie ein Mini-Java-FunProc-Programm an, das die Fibonacci-Funktion aus Aufgabe 3 rekursiv implementiert (für den Aufruf fib(2)).

b) Geben Sie die Übersetzung Ihres Programms aus a) in ZFP-Code an.

```
func fib(int f) {
  int ret;
       if(f == 0) ret = 0;
  else if(f == 1) ret = 1;
  else ret = fib(f-1) + fib(f-2);
  return ret;
}
print(fib(2));
```

```
update(func fib(int f) { int ret; … },st0,h0) = ([fib/(f,1,0,adr(fib)],h0)
```

update(func fib(int f) { int ret; … },st_{fib}0,h0) = (st_{fib}0[f/(var,1),ret/(var,2)],h0)

Abkürzung: sts = Sammlung aller Symboltabellen = st_{fib}st

trans(P) = (cmdtrans(print(fib(2)),sts) HALT; profunctrans(func fib(int f) { … }),h)

Da h gleich bleibt, betrachte ich nur noch die erste Komponente.

```
=             exptrans(fib(2),sts)
              POP;
              HALT;
  adr(fib):   cmdtransfunproc(stmt)
              exptrans(ret,stfibst)          //  … = LODLOCAL 2;
              RET;
```

mit exptrans(fib(2), sts) =

```
            LIT 2;
            CALL(adr(fib),1,0);
```

und cmdtransfunproc(stmt) =

```
adr(fib):   condtrans(f == 0, stfibst)
            JMC a(next);
            cmdtrans(ret = 0;, stfibst)
            JMP a(next+1);
a(next):    cmdtrans(if(f==1) ret=1; else ret=fib(f-1)+fib(f-2);, stfibst)
a(next+1):
```

=

```
adr(fib):   LODLOCAL 1;
            LIT 0;
            EQ;
            JMC a(next);
            LIT 0;
            STOLOCAL 2;
            JMP a(next+1);
a(next):    condtrans(f==1, stfibst)
            JMC a(next+2);
            cmdtrans(ret=1;, stfibst)
            JMP a(next+3);
a(next+2):  cmdtrans(ret=fib(f-1)+fib(f-2);, stfibst)
a(next+1):
a(next+3):
```

=

```
adr(fib):   LODLOCAL 1;
            LIT 0;
            EQ;
            JMC a(next);
            LIT 0;
            STOLOCAL 2;
            JMP a(next+1);
a(next):    LODLOCAL 1;
            LIT 1;
            EQ;
            JMC a(next+2);
            LIT 1;
            STOLOCAL 2;
            JMP a(next+3);
a(next+2):  exptrans(fib(f-1), stfibst)
            exptrans(fib(f-2), stfibst)
            ADD;
            STOLOCAL 2;
a(next+1):
a(next+3):
```

=

```
adr(fib):   LODLOCAL 1;
            LIT 0;
            EQ;
            JMC a(next);
            LIT 0;
            STOLOCAL 2;
            JMP a(next+1);
a(next):    LODLOCAL 1;
            LIT 1;
            EQ;
            JMC a(next+2);
            LIT 1;
            STOLOCAL 2;
            JMP a(next+3);
a(next+2):  LODLOCAL 1;
            LIT 1;
            SUB;
            CALL(adr(fib),1,0);
```

```
            LODLOCAL 1;
            LIT 2;
            SUB;
            CALL(adr(fib),1,0);
            ADD;
            STOLOCAL 2;
a(next+1):
a(next+3):
```

folgt ...

```
1:   LIT 2;
2:   CALL(5,1,0);
3:   POP;
4:   HALT;
5:   LODLOCAL 1;
6:   LIT 0;
7:   EQ;
8:   JMC 12;
9:   LIT 0;
10:  STOLOCAL 2;
11:  JMP 29;
12:  LODLOCAL 1;
13:  LIT 1;
14:  EQ;
15:  JMC 19;
16:  LIT 1;
17:  STOLOCAL 2;
18:  JMP 29;
19:  LODLOCAL 1;
20:  LIT 1;
21:  SUB;
22:  CALL(5,1,0);
23:  LODLOCAL 1;
24:  LIT 2;
25:  SUB;
26:  CALL(5,1,0);
27:  ADD;
28:  STOLOCAL 2;
29:  LODLOCAL 2;
30:  RET;
```

Aufgabe 51:

Berechnen Sie die Semantik ihres ZFP-Programms aus Aufgabe 50 b).

Lösung:

```
(1  , []      , []                    , h0)  //  LIT 2;
(2  , [2]     , []                    , h0)  //  CALL(5,1,0);

(5  , []      , [4,3,0,2]             , h0)  //  LODLOCAL 1;
(6  , [2]     , [4,3,0,2]             , h0)  //  LIT 0;
(7  , [0,2]   , [4,3,0,2]             , h0)  //  EQ;
(8  , [0]     , [4,3,0,2]             , h0)  //  JMC 12;
(12 , []      , [4,3,0,2]             , h0)  //  LODLOCAL 1;
(13 , [2]     , [4,3,0,2]             , h0)  //  LIT 1;
(14 , [1,2]   , [4,3,0,2]             , h0)  //  EQ;
(15 , [0]     , [4,3,0,2]             , h0)  //  JMC 19;
(19 , []      , [4,3,0,2]             , h0)  //  LODLOCAL 1;
(20 , [2]     , [4,3,0,2]             , h0)  //  LIT 1;
(21 , [1,2]   , [4,3,0,2]             , h0)  // SUB;
(22 , [1]     , [4,3,0,2]             , h0)  // CALL(5,1,0);

(5  , []      , [4,23,0,1][4,3,0,2]   , h0)  //  LODLOCAL 1;
(6  , [1]     , [4,23,0,1][4,3,0,2]   , h0)  //  LIT 0;
(7  , [0,1]   , [4,23,0,1][4,3,0,2]   , h0)  //  EQ;
(8  , [0]     , [4,23,0,1][4,3,0,2]   , h0)  //  JMC 12;
(12 , []      , [4,23,0,1][4,3,0,2]   , h0)  //  LODLOCAL 1;
(13 , [1]     , [4,23,0,1][4,3,0,2]   , h0)  //  LIT 1;
(14 , [1,1]   , [4,23,0,1][4,3,0,2]   , h0)  //  EQ;
(15 , [1]     , [4,23,0,1][4,3,0,2]   , h0)  //  JMC 19;
(16 , []      , [4,23,0,1][4,3,0,2]   , h0)  //  LIT 1;
(17 , [1]     , [4,23,0,1][4,3,0,2]   , h0)  //  STOLOCAL 2;
(18 , []      , [4,23,1,1][4,3,0,2]   , h0)  //  JMP 29;
(29 , []      , [4,23,1,1][4,3,0,2]   , h0)  //  LODLOCAL 2;
(30 , [1]     , [4,23,1,1][4,3,0,2]   , h0)  //  RET;

(23 , [1]     , [4,3,0,2]             , h0)  // LODLOCAL 1;
(24 , [2,1]   , [4,3,0,2]             , h0)  // LIT 2;
(25 , [2,2,1] , [4,3,0,2]             , h0)  // SUB;
(26 , [0,1]   , [4,3,0,2]             , h0)  // CALL(5,1,0);

(5  , [1]     , [4,27,0,0][4,3,0,2]   , h0)  // LODLOCAL 1;
(6  , [0,1]   , [4,27,0,0][4,3,0,2]   , h0)  // LIT 0;
```

```
(7  , [0,0,1] , [4,27,0,0][4,3,0,2]  , h0)  // EQ;
(8  , [1,1]   , [4,27,0,0][4,3,0,2]  , h0)  // JMC 12;
(9  , [1]     , [4,27,0,0][4,3,0,2]  , h0)  // LIT 0;
(10 , [0,1]   , [4,27,0,0][4,3,0,2]  , h0)  // STOLOCAL 2;
(11 , [1]     , [4,27,0,0][4,3,0,2]  , h0)  // JMP 29;
(29 , [1]     , [4,27,0,0][4,3,0,2]  , h0)  // LODLOCAL 2;
(30 , [0,1]   , [4,27,0,0][4,3,0,2]  , h0)  // RET;

(27 , [0,1]   , [4,3,0,2]            , h0)  // ADD;
(28 , [1]     , [4,3,0,2]            , h0)  // STOLOCAL 2;
(29 , []      , [4,3,1,2]            , h0)  // LODLOCAL 2;
(30 , [1]     , [4,3,1,2]            , h0)  // RET;

(3  , [1]     , []                   , h0)  //  POP;
(4  , []      , []                   , h0)  //  HALT;
```

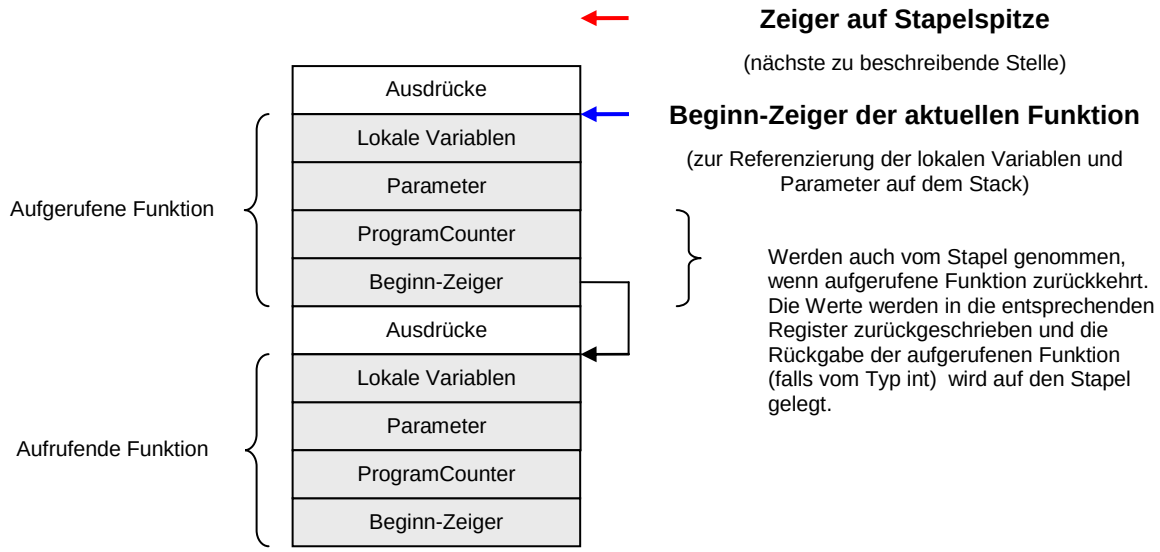
Aufgabe 52:

Ändern Sie die abstrakte Maschine ZFP so ab, dass sie mit einem Keller auskommt. D.h. Sie müssen den Prozedurkeller und Datenkeller auf einem Stack realisieren.

Überlegen Sie sich hierzu, wann Ausdrücke ausgewertet werden und wie Sie sich merken, wo der eigentliche Prozedurkeller wieder beginnt.

Lösung:

Zwischen und oberhalb der Blöcke auf dem Prozedurkeller können jetzt Ausdruckswerte vom Datenkeller liegen. Man muss sich deshalb merken, wo der darunter liegende Block auf dem Prozedurkeller beginnt. Das macht man durch den Beginnzeiger (absolute Position des Tops). Außerdem verwendet man ein Register, das auf den Top des obersten Blocks auf dem Prozedurkeller zeigt.



Aufgabe 53:

Übersetzen Sie das Programm auf Folie „Datenstrukturen 4“ in Zwischen-Code.

```
type arrT  = int[3];
type strT  = struct { int anfang, ende };
int i, arrT a, strT s, int* p;
{
   i = 0;
   s.anfang = i;
   p = new int[1];
   *p = 42;
   while i < 3 {
      a[i] = *p * i;
      i = i + 1;
   }
   s.ende = i;
}
```

Lösung:

1: LIT 0;	**11: LOD 0;**	**21: JMC 36;**	**31: LOD 1;**
2: STO 1;	**12: LIT 1;**	**22: LIT 2;**	**32: LIT 1;**
3: LIT 5;	**13: ADD;**	**23: LOD 1;**	**33: ADD;**
4: LIT 0;	**14: STO 0;**	**24: AC 3;**	**34: STO 1;**
5: ADD;	**15: LOD 7;**	**25: ADD;**	**35: JMP 18;**
6: LOD 1;	**16: LIT 42;**	**26: LOD 7;**	**36: LIT 5;**
7: STOI;	**17: STOI;**	**27: LODI;**	**37: LIT 1;**
8: LIT 7;	**18: LOD 1;**	**28: LOD 1;**	**38: ADD;**
9: LOD 0;	**19: LIT 3;**	**29: MULT;**	**39: LOD 1;**
10: STOI;	**20: LT;**	**30: STOI;**	**40: STOI;**

Aufgabe 54:

Führen Sie die Rechnung des Zwischen-Code Programms aus Aufgabe 53 durch.

Lösung:

```
(1,    ,        [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //LIT 0
(2,   0,        [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //STO 1
(3,    ,        [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //LIT 5
(4,   5,        [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //LIT 0
(5, 0 5,        [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //ADD
(6,   5,        [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //LOD 1
(7, 0 5,        [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //STOI
```

```
(8,     ,       [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //LIT 7
(9,    7,       [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //LOD 0
(10, 8 7,       [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/0])        //STOI
(11,    ,       [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/0])    //LOD 0
(12,   8,       [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/0])    //LIT 1
(13, 1 8,       [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/0])    //ADD
(14,   9,       [0/8,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/0])    //STO 0
(15,    ,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/0])    //LOD 7
(16,   8,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/0])    //LIT 42
(17, 42 8 ,     [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/0])    //STOI
(18,    ,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LOD 1
(19,   0,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LIT 3
(20, 3 0,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LT
(21,   1,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //JMC 36
(22,    ,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LIT 2
(23,   2,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LOD 1
(24, 0 2,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //AC 3
(25, 0 2,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //ADD
(26,   2,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LOD 7
(27, 8 2,       [0/9,1/0,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LODI
(28, 42 2,      [0/9,1/0,2/8,3/0,4/0,5/0,6/0,7/8,8/42])   //LOD 1
(25, 1 2,       [0/9,1/1,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //ADD
(26,   3,       [0/9,1/1,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LOD 7
(27, 8 3,       [0/9,1/1,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LODI
(28, 42 3,      [0/9,1/1,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //LOD 1
(29, 1 42 3,    [0/9,1/1,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //MULT
(30, 42 3,      [0/9,1/1,2/0,3/0,4/0,5/0,6/0,7/8,8/42])   //STOI
(31,    ,       [0/9,1/1,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LOD 1
(32,   1,       [0/9,1/1,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LIT 1
(33, 1 1,       [0/9,1/1,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //ADD
(34,   2,       [0/9,1/1,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //STO 1
(35,    ,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //JMP 18
(18,    ,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LOD 1
(19,   2,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LIT 3
(20, 3 2,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LT
(21,   1,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //JMC 36
(22,    ,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LIT 2
(23,   2,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LOD 1
(24, 2 2,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //AC 3
(25, 2 2,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //ADD
(26,   4,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LOD 7
(27, 8 4,       [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LODI
(28, 42 4,      [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //LOD 1
(29, 2 42 4,    [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //MULT
(30, 84 4,      [0/9,1/2,2/0,3/42,4/0,5/0,6/0,7/8,8/42])  //STOI
(31,    ,       [0/9,1/2,2/0,3/42,4/84,5/0,6/0,7/8,8/42]) //LOD 1
(32,   2,       [0/9,1/2,2/0,3/42,4/84,5/0,6/0,7/8,8/42]) //LIT 1
```

```
(33, 1 2,       [0/9,1/2,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //ADD
(34,   3,       [0/9,1/2,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //STO 1
(35,    ,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //JMP 18
(18,    ,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //LOD 1
(19,   3,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //LIT 3
(20, 3 3,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //LT
(21,   0,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //JMC 36
(36,    ,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //LIT 5
(37,   5,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //LIT 1
(38, 1 5,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //ADD
(39,   6,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //LOD 1
(40, 3 6,       [0/9,1/3,2/0,3/42,4/84,5/0,6/0,7/8,8/42])  //STOI
(41,    ,       [0/9,1/3,2/0,3/42,4/84,5/0,6/3,7/8,8/42])  //STOPP
```

Weihnachtsaufgabe:

Darstellung der römischen Zahlen von 1 bis 399 durch einen regulären Ausdruck.

Lösung:

Beispiel: 399 ≡ CCCXCIX

349 ≡ CCCXLIX

Regulärer Ausdruck: `C{0,3}  ( L? X{0,3} | X[LC] )  ( V? I{0,3} | I[VX])`

Wobei das „?“ für Vielfachheit 0 bis 1 (also optional) und „[…]“ für genau ein Element aus den Klammern steht.