Aufgabe 1

(a) Die Lexspezifikation ist sehr einfach. Für jedes Zeichen einer römischen Zahl wird sein numerischer Wert in der Kommunikationsvariablen *yylval* und das zugehörige Token zurückgegeben:

```
%%
[iI]   {yylval = 1; return(I);}
[vV]   {yylval = 5; return(V);}
[xX]   {yylval = 10; return(X);}
[lL]   {yylval = 50; return(L);}
[cC]   {yylval = 100; return(C);}
[dD]   {yylval = 500; return(D);}
[mM]   {yylval = 1000; return(M);}
\n     {return(yytext[0]);}
.      ;
```

Anzumerken bleibt, daß der Deklarationsteil und der Teil für Hilfsprozeduren weggelassen wurden, da sie nicht benötigt werden.

(b) Die Yacc-Spezifikation kann wie folgt aussehen:

```
%{
int fuenf = 0, fuenfzig = 0, fuenfhundert = 0;
%}

%token I V X L C D M

%%
zahlen          : rom_zahl '\n'
                | zahlen {init();} rom_zahl '\n'
                ;
rom_zahl        : tausend {if (pruef()) printf("   ==> %d\n", $1);}
                | rest {if (pruef()) printf("   ==> %d\n", $1);}
                | tausend rest {if (pruef()) printf("   ==> %d\n", $1+$2);}
                ;
tausend         : M
                | tausend M {$$=$1+$2;}
                ;
```

```
rest          : hundert
              | zehner_einer
              | hundert zehner_einer {$$=$1+$2;}
              ;
zehner_einer  : einer
              | zehner
              | zehner einer {$$=$1+$2;}
              ;
hundert       : hund_200_300
              | ein_5_10_50_100 D {fuenfhundert++;$$=$2-$1;}
              | D {fuenfhundert++;}
              | D hund_200_300 {fuenfhundert++;$$=$1+$2;}
              | ein_5_10_50_100 M {$$=$2-$1;}
              | D M {fuenfhundert++;$$=$2-$1;}
              ;
zehner        : zehn_20_30
              | ein_5_10 L {fuenfzig++;$$=$2-$1;}
              | L {fuenfzig++;}
              | L zehn_20_30 {fuenfzig++;$$=$1+$2;}
              | ein_5_10_50 C {$$=$2-$1;}
              ;
einer         : eins_2_3
              | I V {fuenf++;$$=$2-$1;}
              | V {fuenf++;}
              | V eins_2_3 {fuenf++;$$=$1+$2;}
              | I X {$$=$2-$1;}
              ;
eins_2_3      : I
              | I I {$$=$1+$2;}
              | I I I {$$=$1+$2+$3;}
              ;
zehn_20_30    : X
              | X X {$$=$1+$2;}
              | X X X {$$=$1+$2+$3;}
              ;
hund_200_300  : C
              | C C {$$=$1+$2;}
              | C C C {$$=$1+$2+$3;}
              ;
```

```
ein_5_10        : I
                | V {fuenf++;}
                | X
                ;
ein_5_10_50     : ein_5_10
                | L {fuenfzig++;}
                ;
ein_5_10_50_100     : ein_5_10_50
                    | C
                    ;

%%
#include "lex.yy.c"

void init()
{
 fuenf = fuenfzig = fuenfhundert = 0;
}

int pruef()
{
 if (fuenf > 1)
  {printf("Fehler: V nur einmal erlaubt!\n"); return 0;}
 else if (fuenfzig > 1)
  {printf("Fehler: L nur einmal erlaubt!\n"); return 0;}
 else if (fuenfhundert > 1)
  {printf("Fehler: D nur einmal erlaubt!\n"); return 0;}
 else
  return 1;
}
```

Einige Erläuterungen zur Lösung: Die Regel für *zahlen* beschreibt eine durch Returns getrennte Folge von römischen Zahlen. Zu beachten ist, daß zwischendurch die drei Variablen *fuenf*, *fuenfzig* und *fuenfhundert*, die die Häufigkeit des Auftretens ihrer entsprechenden Zeichen festhalten, neu initialisiert werden müssen (Funktion *init*()). Die Regeln für *rom_zahl*, *tausend*, *rest* und *zehner_einer* sollten leicht verständlich sein. Bezüglich der Regel *hundert*, die die Hunderter innerhalb der römischen Zahl behandelt, können verschiedene Fälle auftreten. Entweder gibt es (a) ein, zwei oder drei C nebeneinander, (b) ein Zeichen aus {I, V, X, L, C} vor einem D, (c) nur ein D, (d) ein D gefolgt von einem, zwei oder drei C oder (e) ein Zeichen aus {I, V, X, L, C, D} vor einem M. Die Regeln für *zehner* und *einer* sind analog aufgebaut. Bei den gerade beschriebenen letzten drei Regeln ist insbesondere R6 zu beachten. Die restlichen Regeln sollten leicht verständlich sein. Die Funktion

pruef() prüft die Einhaltung von R3 und gibt bei einer Verletzung eine entsprechende Meldung aus.

Aufgabe 2

Zunächst zeigt sich, daß G aufgrund der beiden Produktionen $S \rightarrow S\ ;\ S$ und $E \rightarrow E + E$ mehrdeutig ist. Wir lösen diese Mehrdeutigkeit durch Einführung zweier weiterer Nichtterminalsymbole T und F wie folgt auf:

Wir ersetzen die Produktion $S \rightarrow S\ ;\ S$ durch $S \rightarrow S\ ;\ T$ und fügen die Produktionen $T \rightarrow$ **id** := E und $T \rightarrow$ **print** (L) hinzu. Ferner ersetzen wir die Produktion $E \rightarrow E + E$ durch $E \rightarrow E + F$ und fügen die Produktionen $F \rightarrow$ **id** und $F \rightarrow$ **num** hinzu. Insgesamt erhalten wir die Grammatik $G_1 = (N_1, \Sigma_1, P_1, S)$ mit

$$
\begin{aligned}
N_1 &= \{S, E, L, T, F\},\\
\Sigma_1 &= \{\textbf{id}, \textbf{print}, \textbf{num}, ;, ,, (,), :=, +\} \text{ und}\\
P_1 &= \{\ S \rightarrow S\ ;\ T \mid \textbf{id} := E \mid \textbf{print}\ (L),\\
&\qquad T \rightarrow \textbf{id} := E \mid \textbf{print}\ (L),\\
&\qquad E \rightarrow \textbf{id} \mid \textbf{num} \mid E + F,\\
&\qquad F \rightarrow \textbf{id} \mid \textbf{num},\\
&\quad\ L \rightarrow E \mid L, E\ \}.
\end{aligned}
$$

Die Grammatik G_1 ist nicht LL(1), da linksrekursive Produktionen auftreten, die beseitigt werden müssen. Wir ersetzen dazu die Produktionen

$$S \rightarrow S\ ;\ T \mid \textbf{id} := E \mid \textbf{print}\ (L)$$

durch

$$
\begin{aligned}
S &\rightarrow \textbf{id} := E\ S' \mid \textbf{print}\ (L)\ S'\\
S' &\rightarrow ;\ T\ S' \mid \varepsilon\ ,
\end{aligned}
$$

die Produktionen

$$E \rightarrow \textbf{id} \mid \textbf{num} \mid E + F$$

durch

$$
\begin{aligned}
E &\rightarrow \textbf{id}\ E' \mid \textbf{num}\ E'\\
E' &\rightarrow +\ F\ E' \mid \varepsilon
\end{aligned}
$$

und die Produktionen

$$L \rightarrow E \mid L, E$$

durch

$$L \rightarrow E\,L'$$
$$L' \rightarrow , E\,L' \mid \varepsilon$$

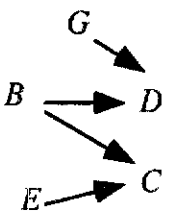
Insgesamt erhalten wir die Grammatik $G_2 = (N_2, \Sigma_2, P_2, S)$ mit

$$\begin{aligned}
N_2 &= \{S, S', E, E', L, L', T, F\},\\
\Sigma_2 &= \{\textbf{id}, \textbf{print}, \textbf{num}, ;, ,, (,), :=, +\} \text{ und}\\
P_2 &= \{\ S \rightarrow \textbf{id} := E\,S' \mid \textbf{print}\ (L)\ S',\\
&\qquad S' \rightarrow ;\,T\,S' \mid \varepsilon,\\
&\qquad T \rightarrow \textbf{id} := E \mid \textbf{print}\ (L),\\
&\qquad E \rightarrow \textbf{id}\,E' \mid \textbf{num}\,E',\\
&\qquad E' \rightarrow +\,F\,E' \mid \varepsilon\\
&\qquad F \rightarrow \textbf{id} \mid \textbf{num},\\
&\quad L \rightarrow E\,L',\\
&\quad L' \rightarrow , E\,L' \mid \varepsilon\}.
\end{aligned}$$

Eine Linksfaktorisierung muß nicht durchgeführt werden, da es für kein Nichtterminal $X \in N_2$ verschiedene X-Produktionen mit gleichem Präfix gibt. Wir erhalten die LL(1)-Grammatik $G' = G_2$.

Aufgabe 3

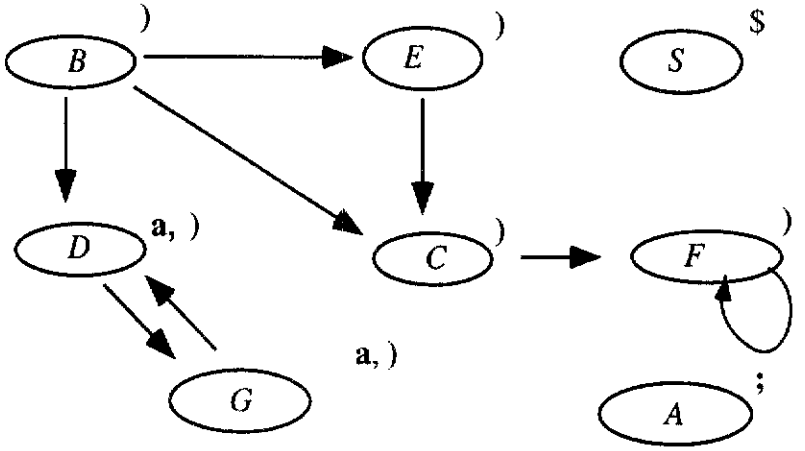
(a) Wir zeichnen den Graphen, der die Berechnungsreihenfolge festlegt. Es gilt $N_\varepsilon = \{A, F, G, E\}$



Nun berechnen wir mit Algorithmus 3.14 die FIRST- und initialen Steuermengen.

FIRST_1		Produktion		Steuermenge
{**c**}	S	→	**c a** A **;**	{**c**}
{**(**, ε}	A	→	**(** B **)**	{**(**}
		→	ε	{ε}
{**d**, **a**}	B	→	$D\,E$	{**d**}
		→	C	{**a**}
{**a**}	C	→	**a : b** F	{**a**}
{**;** , ε}	F	→	**; a : b** F	{**;**}
		→	ε	{ε}
{**d**}	D	→	**d a : b ;** G	{**d**}
{**d** , ε}	G	→	D	{**d**}
		→	ε	{ε}
{**a**, ε}	E	→	C	{**a**}
		→	ε	{ε}

(b) Für die Produktionen (3), (8), (11) und (13) enthält die initiale Steuermenge ε. Für die Nichtterminale der linken Seiten dieser Produktionen werden die FOLLOW-Mengen benötigt. Anwendung des Algorithmus 3.15 (ohne Schritt 3) liefert den folgenden Graphen.



Nun lassen sich die benötigten FOLLOW-Mengen ablesen.

$\text{FOLLOW}_1(A) = \{\mathbf{;}\}$

$\text{FOLLOW}_1(F) = \{\mathbf{)}\}$

$\text{FOLLOW}_1(G) = \{\mathbf{a}\ ,\ \mathbf{)}\}$

$\text{FOLLOW}_1(E) = \{\mathbf{)}\}$

Wir erhalten

	Produktion	Steuermenge
(1)	$S \rightarrow$ **c a** A **;**	{**c**}
(2)	$A \rightarrow$ **(** B **)**	{(}
(3)	$\rightarrow \varepsilon$	{;}
(4)	$B \rightarrow D\,E$	{**d**}
(5)	$\rightarrow C$	{**a**}
(6)	$C \rightarrow$ **a : b** F	{**a**}
(7)	$F \rightarrow$ **; a : b** F	{;}
(8)	$\rightarrow \varepsilon$	{)}
(9)	$D \rightarrow$ **d a : b ;** G	{**d**}
(10)	$G \rightarrow D$	{**d**}
(11)	$\rightarrow \varepsilon$	{**a** ,)}
(12)	$E \rightarrow C$	{**a**}
(13)	$\rightarrow \varepsilon$	{)}

(c) Die Analysetafel hat folgende Gestalt:

$n \in N_1 \setminus t \in \Sigma$	**c**	**d**	**a**	**b**	**(**	**)**	**:**	**;**
S	1							
A					2			3
B		4	5					
C			6					
F						8		7
D		9						
G		10	11			11		
E			12			13		

Aufgabe 4

(a)

(i) f hat den Typ $int \times \alpha \rightarrow int$. Da auf x die Addition angewandt wird, muß x vom Typ *int* sein. Da x+1 das Ergebnis darstellt, muß auch das Ergebnis vom Typ *int* sein. Da y keiner Einschränkung unterliegt, kann man die Typvariable α annehmen.

(ii) g hat den Typ $\alpha \times \alpha \rightarrow int$. Aufgrund des Gleichheitstests müssen x und y den gleichen Typ haben, sie unterliegen sonst aber keiner Einschränkung. Man kann daher die

Typvariable α für beide Parameter annehmen. Das Ergebnis muß vom Typ *int* sein, da 0 oder 1 als Ergebnis geliefert wird.

(iii) h hat den Typ $int \times int \rightarrow \alpha$. Da auf x das > Prädikat angewandt wird, muß x vom Typ *int* sein. Da in der ersten Alternative y als erstes Argument verwandt wird, muß auch y vom Typ *int* sein. (Man kann auch argumentieren, daß x als zweites Argument verwandt wird.) Das Ergebnis der Funktion ist nicht eingeschränkt; daher ergibt sich die Typvariable als Ergebnistyp.

(b) Um den Ausdruck auszuwerten, müssen wir die Regel für die Applikation anwenden. Darin müssen wir zunächst die Funktion zu einer Closure und das Argument zu einem Wert auswerten:

$I(\textbf{fn}\ i \Rightarrow i*2, U) = (\textbf{fn}\ i \Rightarrow i*2, U)$
$I(z, U) = lookup(z, U) = 7$

Damit ergibt sich für die Applikation:

$I((\textbf{fn}\ i \Rightarrow i*2)\ z, U) = I(i*2, (i, 7)\cdot U)$

Als nächstes muß die binäre vordefinierte Funktion * angewandt werden. Es sei im folgenden $U' = (i, 7)\cdot U$. Man erhält dann:

$I(i*2, U') = apply(*, I(i, U'), I(2, U'))$

Um *apply* auswerten zu können, muß man zuvor die Variable und die Konstante auswerten:

$I(i, U') = lookup(i, (i, 7)\cdot U) = 7$
$I(2, U') = 2$

Damit erhält man schließlich:

$apply(*, 7, 2) = 14$

(c) Die Übersetzung liefert den folgenden SECD-Code:

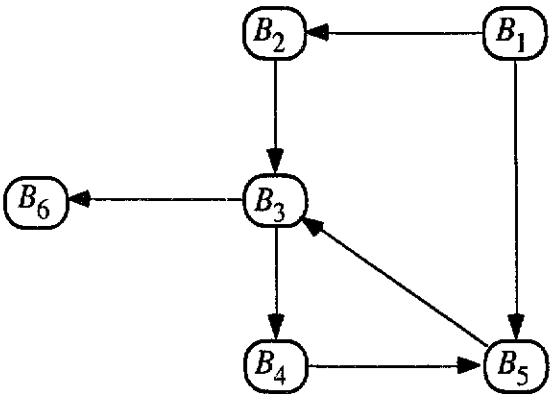
[*LDC* [*LDC* [*LDV* 2, *LDV* 1, *GT*, *COND* ([*LDV* 2, *RET*], [*LDV* 1, *RET*])]]]

Aufgabe 5

(a) Basisblöcke:

B_1	`(1) if x<3 then goto 6`	Erste Anweisung
B_2	`(2) x := x-1`	nach Verzweigung
B_3	`(3) y := y-1` `(4) if y>7 then goto 8`	Ziel von goto
B_4	`(5) y := x`	nach Verzweigung
B_5	`(6) x := y+2` `(7) goto 3`	Ziel von goto
B_6	`(8) x := y-1`	nach Verzweigung

(b) Flußgraph:



(c) Die Mengen *gen* und *kill* kann man direkt aus den Basisblöcken ablesen:

Block	*gen*	*kill*
B_1	{}	{}
B_2	{2}	{6, 8}
B_3	{3}	{5}
B_4	{5}	{3}
B_5	{6}	{2, 8}
B_6	{8}	{2, 6}

Zu Beginn gilt für alle Mengen $in(B_i) = \{\}$ und $out(B_i) = gen(B_i) \cup (in(B_i) - kill(B_i)) = gen(B_i)$:

Block	*in*	*out*
B_1	{ }	{ }
B_2	{ }	{2}
B_3	{ }	{3}
B_4	{ }	{5}
B_5	{ }	{6}
B_6	{ }	{8}

Nun werden wiederholt $in(B_i)$ und $out(B_i)$ neu berechnet, bis sich keine Änderungen mehr in den Mengen $out(B_i)$ ergeben. Aus der Vorgängerinformation des Flußgraphen und der allgemeinen Gleichung $out(B_i) = gen(B_i) \cup (in(B_i) - kill(B_i))$ ergeben sich die folgenden iteriert zu berechnenden Gleichungen. (Die Gleichungen für $in(B_2)$ und $in(B_5)$ sind bereits vereinfacht, da $out(B_1) = \{\}$. Ebenso kann man $out(B_2)$ vereinfachen, da $in(B_2) = \{\}$.)

$in(B_1) = \{\}$ $\quad$ $out(B_1) = \{\} \cup (\{\} - \{\}) = \{\}$

$in(B_2) = \{\}$ $\quad$ $out(B_2) = \{2\}$

$in(B_3) = out(B_2) \cup out(B_5)$ $\quad$ $out(B_3) = \{3\} \cup (in(B_3) - \{5\})$

$in(B_4) = out(B_3)$ $\quad$ $out(B_4) = \{5\} \cup (in(B_4) - \{3\})$

$in(B_5) = out(B_4)$ $\quad$ $out(B_5) = \{6\} \cup (in(B_5) - \{2, 8\})$

$in(B_6) = out(B_3)$ $\quad$ $out(B_6) = \{8\} \cup (in(B_6) - \{2, 6\})$

Nach der ersten Iteration erhält man:

Block	*in*	*out*
B_1	{ }	{ }
B_2	{ }	{2}
B_3	{2, 6}	{2, 3, 6}
B_4	{2, 3, 6}	{2, 5, 6}
B_5	{2, 5, 6}	{5, 6}
B_6	{2, 3, 6}	{3, 8}

Die zweite Iteration ergibt:

Block	*in*	*out*
B_1	{}	{}
B_2	{}	{2}
B_3	{2, 5, 6}	{2, 3, 6}
B_4	{2, 3, 6}	{2, 5, 6}
B_5	{2, 5, 6}	{5, 6}
B_6	{2, 3, 6}	{3, 8}

Da sich keine Änderungen ergeben haben, kann man das Verfahren beenden.

Aufgabe 5

Wir wählen das Register R für die Berechnung von T3 aus, um eine möglichst kurze Befehlsfolge zu erhalten:

```
LOAD  R,a
SUB   R,b
LOAD  S,c
MULT  S,d
STORE R,T1      (T1 und T2 haben noch weitere Anwendungen)
LOAD  R,S
ADD   R,x
SUB   R,T1      (T3 kann überschrieben werden)
STORE R,T4      (T4 hat noch eine weitere Anwendung)
ADD   R,y
MULT  S,T4      (T2 kann nun überschrieben werden)
ADD   S,R       (T6 kann überschrieben werden)
STORE S,z
```