1)

A = a {B | de } c | C.
B = b [c] | [b] f.
C = {c} d.

a)

First(A):= {a, c, d}
First(B):= {b, f}
First(C):= {c, d}
Follow(A):= {}
Follow(B):= {c, d, b, f}
Follow(C):= {}

b)

LL(1) konflikt in B, da
B = B1 | B2.
B1 = b [c].
B2 = [b] f.
First(B1):= {b}
First(B2):= {b, f}
First(B1) $\cap$ First(B2) = {b} $\neq$ {} $\Rightarrow$ LL(1)-Konflikt
Also muss der Konflikt behoben werden. Am einfachsten geht das, indem man alle Optionen von B einzeln anschreibt, also:
B = b | bc | bf | f.
Dann kann man die ersten drei Optionen zusammenfassen und kommt schließlich auf:
B = b [c | f] | f.
Dieses NTS ist dann ohne Linksrekursionen und somit LL(1) konform.

2)

Factor = number | ident [Params] | „new“ ident (Params | Dims).
Params = “(“ [ident {“,” ident}] “)”
Dims = “[“ number {“,” number} “]”.

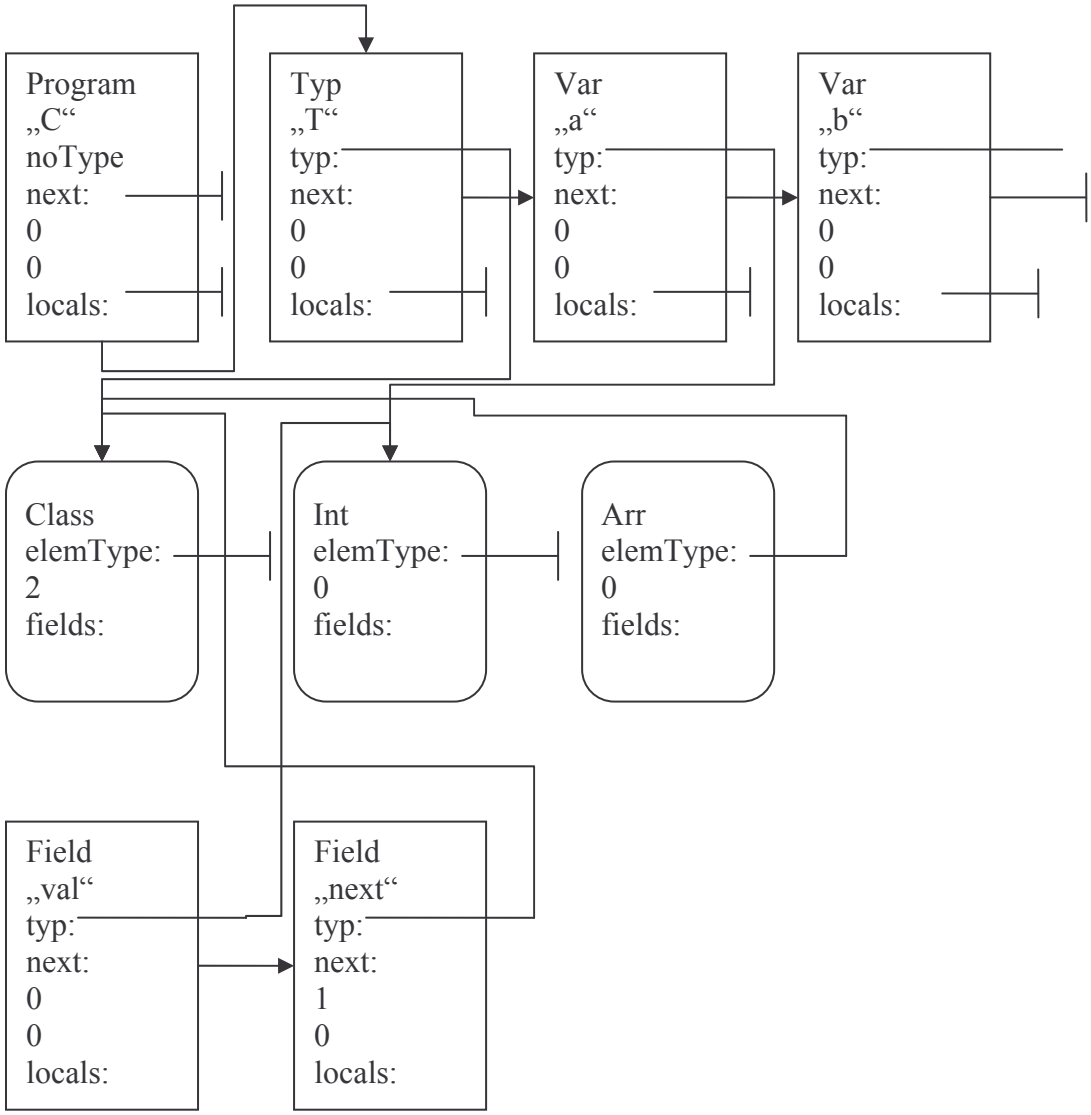
```
void parse() {
     scan();
     Factor();
     check(eof);
}

void Factor() {
     switch (sym) {
          case number:
               scan(); break;
          case ident:
               scan();
```

```
                if (sym == lpar) Params();
                break;
           case newKW:
                scan();
                check (ident);
                if (sym == lpar) Params();
                else if (sym == lbrack) Dims();
                else error("'(' or '[' expected);
                break;
           default:
                error("number, ident or new expected");
                break;
     }
}

void Params() {
     check (lpar);
     if (sym == ident) {
          scan();
          while (sym == comma) {
               scan ();
               check (ident);
          }
     }
     check (rpar);
}

void Dims() {
     check (lbrack);
     check (number);
     while (sym == comma) {
          scan();
          check (number);
     }
     check (rbrack);
}
```

3)

```
class C
     class T {
          int val;
          T next;
     }
     int a;
     int b;
{...}
```

Symboltabelle: (so in etwa)



4)

Anm.: Hier gibt es sehr viele verschiedene Ansätze, die richtig sind. Wichtig ist, dass man darauf achtet, dass es durchaus eine Rolle spielt, ob man die (. und .) schließt oder offen lässt. Hier ist eine möglich Variante:

```
FieldList =                                  (. int x, ctr = 0; .)
  {
    FieldDecl <⇑x, ⇓ctr>                     (. ctr+=x; .)
  }                                          (. print("Gesamt: "+ctr); .)

FieldDecl<⇓x, ⇑x> =                          (. int x, size; String name;
                                             .)
  Type <⇑size>
  Id <⇑name>                                 (. x+=size;
                                             print (name + ": "+x); .)
  {
    "," Id<⇑name>                            (. x+=size;
                                             print(name+": "+x); .)
  }
  ";"

Type<⇑size> =                                (. int size; .)
  "boolean"                                  (. size = 1; .)
  | "char"                                   (. size = 2; .)
  | "int"                                    (. size = 4; .)

Id<⇑name> =                                  (. String name; .)
  ident .                                    (. name = t.string; .)
```

5)

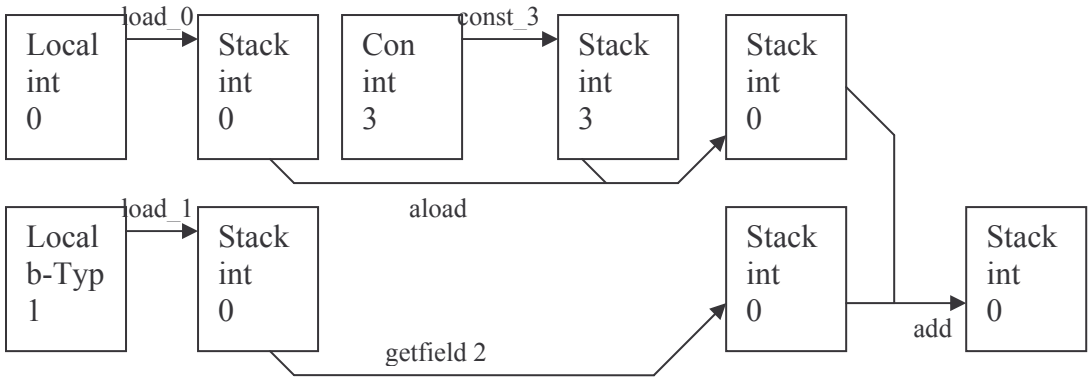
a[3] + b.f

Erklärung:

a ist ein Pointer auf ein Array am Heap. Zuerst muss man also load_0 ausführen, um die Adresse von a auf den Stack zu laden. Danach kann man mit const_3 den Wert 3 auf den Stack laden und mit aload schließlich den Wert des indizierten Arrayelements.

b ist ein Pointer auf ein Objekt am Heap. Zuerst muss man also mit load_1 die Adresse auf den Stack laden. Danach kann man mit getfield 2 das 2. Feld (f) auslesen. In diesem Fall ist natürlich 2 schon zur Compilezeit bekannt, deshalb kann man es hier gleich einsetzen.

Mit add addiert man die 2 Werte am Stack und legt das Ergebnis auf den Stack.



Erzeugter Code:

```
load_0
const_3
aload
load_1
getfield 2
add
```

6)

(von Andreas Kothmeier)

```
LALR(1)-Tabellenerzeugung

Gegebene Grammatik:
~~~~~~~~~~~~~~~~~~~

E=aEb.
E=x.

Das heißt wir brauchen noch eine Pseudoproduktion E'=E#, mit der wir dann
die Tabellen-
erzeugung starten.

Wir haben jetzt also 3 Regeln:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

0 E'=E#.
1 E=x.      //um die Wegweiser zu finden, müssen wir die nichtrekursiven
2 E=aEb.    //Regeln vor die Rekursiven ordnen

0 E'=.E#          shift x -> 1               <- 1. Wegweiser ist x
  E=.x      /#b   shift a -> 2
  E=.aEb    /#b   shift E -> 3
-------------------------------------------------------
1 E=x.      /#b   red   {#,b} (nach Regel 1)  <- b ist hier Wegweiser
-------------------------------------------------------
2 E=a.Eb    /#b   shift x -> 1               <- x ist Wegweiser
  E=.x      /#b   shift a -> 2
  E=.aEb    /#b   shift E -> 4
-------------------------------------------------------
3 E'=E.#          acc   #                   <- # ist hier Wegweiser
-------------------------------------------------------
4 E=aE.b    /#b   shift b -> 5               <- b ist hier Wegweiser
```

```
-----------------------------------------------------------
5 E=aEb.    /#b   red   {#,b} (nach Regel 2)  <- b ist hier Wegweiser
-----------------------------------------------------------
```

Daher erhalten wir folgende LALR(1)-Tabelle:
~~~~~~

	a	b	x	#	E	guide
0	s2		s1		s3	x
1		r1		r1		b
2	s2		s1		s4	x
3				acc		#
4		s5				b
5		r2		r2		b

Übrigens: Wegweiser ist immer das Terminalsysmbol, das bei der 1.
~~~~~~~~~ Terminalsymbol-Aktion des jeweiligen Zustandes verwendet wird.