PEARL90 - Programmieren mit Zeigern

von L. Frevert

1 Einleitung

Anders als in PASCAL dienen Zeiger in PEARL90 nicht zur Verwaltung von dynamisch erzeugten Variablen, sondern zur Steigerung der Effizienz bei Programmierung, Ausführungsgeschwindigkeit und Speicherausnutzung. So ermöglichen zum Beispiel Felder von Zeigern, die auf Tasks zeigen, letztere innerhalb von Schleifen zu aktivieren und kürzere und übersichtlichere Programme zu schreiben.

Mit Zeigern auf Prozeduren können die Schnittstellen zwischen Moduln in objektorientierter Weise formuliert werden, und Zeiger ermöglichen auch die Formulierung von Datentypen und Deklarationen, die den Klassen und Objekten von objektorientierten Sprachen entsprechen. Aus diesem objektorientierten Ansatz beim Schreiben von PEARL90-Programmen ergeben sich große Vorteile vor allem dann, wenn den Software-Objekten Geräte, Baugruppen oder Anlagen im technischen Prozeß entsprechen.

Wie in PASCAL sind die Zeiger in PEARL streng an die Typen der Objekte gebunden, auf die sie zeigen. Es gibt aber auch die Möglichkeit, diese Typbindnung aufzuheben, um zum Beispiel polymorphe Prozeduren zu schreiben. Dabei wird dem Programmierer die Möglichkeit geboten, durch Abfragen mit speziellen Operatoren dafür zu sorgen, daß die Programme sicher bleiben.

2 Grundlagen

2.1 Referenzstufen und Dereferenzierung

In PEARL90 gibt es normale Variable und Referenz-Variable. Die normalen Variablen werden im folgenden kurz “Variable” genannt, die Referenz-Variablen ”Zeiger”.

Eine Variable hat einen Variablennamen und einen Variableninhalt. Durch die Deklarationen `DCL (a,b) FIXED;` werden zwei Variablen mit den Variablennamen `a` und `b` geschaffen, die zunächst einen durch Zufall bestimmten Variableninhalt haben. Durch die Zuweisungen `a:=5;` und `b:=7;` bekommen die beiden Variablen die Ganzzahlwerte `5` bzw. `7` als Variableninhalte.

In den beiden Zuweisungen stehen links Variablennamen, rechts Zahlenwerte. Bekanntlich können wir Zuweisungen aber auch in der Form `a:=b;` schreiben; dann wird der Variableninhalt (nämlich der Zahlenwert `7`) derjenigen Variablen, deren Variablenname `b` ist, in die Variable

mit dem Variablennamen a kopiert, wird also deren neuer Variableninhalt. Mit der Nennung des Variablennamens b rechts vom := meinen wir in Wirklichkeit deren Inhalt; der Variablenname bildet eine "Referenz" auf den Variableninhalt, also auf den Zahlenwert. Die Tatsache, daß wir den Variablennamen nennen, aber den Inhalt meinen, nennt man "implizite Dereferenzierung".

Wir wollen festlegen, daß ein Variableninhalt die "Referenzstufe" 0 hat, ein Variablenname die Referenzstufe 1. Um die Dereferenzierung deutlich zu machen, schreiben wir die Zuweisung hier `a:=b/*1->0*/;`.

Ein Zeiger hat einen Zeigernamen und einen Zeigerinhalt. Durch die Deklaration `DCL (x,y) REF FIXED;` erhalten wir zwei Zeiger mit den Zeigernamen x und y und zunächst unbestimmtem Zeigerinhalt, die auf Variable des Typs FIXED zeigen können. Durch die Zuweisungen `x:=a;` und `y:=b;` bekommen die beiden Zeiger ihre Zeigerinhalte, nämlich die Variablennamen a bzw. b. Sie "zeigen" jetzt auf diese Variablen. Wir wollen festlegen, daß Zeigernamen die Referenzstufe 2 haben.

Wir nennen die Zahl 7 eine Konstante; eine Variable hat also immer eine Konstante als Variableninhalt, die wir per Zuweisung durch eine andere Konstante ersetzen können. Ganz analog sind Variablennamen in Bezug auf Zeiger Zeigerinhaltskonstante.

Wir können jetzt eine Zuweisung `x:=y;` schreiben; dann wird der Zeigerinhalt (nämlich der Variablenname b) desjenigen Zeigers, dessen Zeigername y ist, in den Zeiger mit dem Zeigernamen x kopiert, wird also dessen neuer Zeigerinhalt. Mit der Nennung des Zeigernamens y rechts vom := meinen wir in Wirklichkeit dessen Zeigerinhalt; der Zeigername bildet eine Referenz auf den Namen einer Variablen. Um das zu dokumentieren, steht in den Deklarationen von Zeigern das Schlüsselwort REF. Rechts vom := haben wir mit der Nennung des Zeigernamens den Zeigerinhalt gemeint; dort ist also wieder implizit dereferenziert worden. Um das deutlich zu machen, schreiben wir `x:=y/*2->1*/;`.

Statt auf der rechten Seite des := implizit dereferenzieren zu lassen, dürfen wir zur besseren Dokumentation auch explizit dereferenzieren, indem wir `x:=CONT y;` schreiben; die Floskel `CONT y` liefert uns den Zeigerinhalt des Zeigers mit dem Zeigernamen y, nämlich den Variablennamen b. Wir können also mit `CONT y` jederzeit auf denjenigen Variablennamen zugreifen, der Zeigerinhalt des Zeigers y ist.

Dieser Zeigerinhalt ist zur Zeit der Variablenname b. Diesen Sachverhalt können wir dazu benutzen, um den Variableninhalt der Variablen b zu ändern, indem wir `CONT y:=5;` zuweisen. Wenn der Variableninhalt von a die Zahl 5 ist, können wir das selbe Ergebnis erzielen, indem wir `CONT y := a;` schreiben; dann wird rechts wieder implizit dereferenziert (`CONT y :=a/*1->0*/;`). Wenn außerdem der Zeigerinhalt des Zeigers x auch noch der Variablenname a ist, dürfen wir auch CONT

`y:= CONT x;` schreiben; rechts vom `:=` wird dann einmal explizit (`CONT`) und einmal (der Variablenname `a` als Ergebnis von `CONT x`) implizit dereferenziert. Wir dürfen sogar `CONT y := x;` schreiben; dann wird rechts zweimal implizit dereferenziert (`CONT y := x/*2->1->0*/;`).

Eine Zuweisung `y := 5;` muß offensichtlich verboten sein. rechts steht ja ein Zahlenwert und links der Name eines Zeigers, der nur einen Variablennamen als Zeigerinhalt haben darf (und keinen Zahlenwert). Wenn wir festlegen, daß Zahlenwerte, Variable und Zeiger die "Referenzstufen" 0, 1 bzw. 2 haben, dann muß offenbar gelten: *Auf der rechten Seite einer Zuweisung darf keine Referenzstufe stehen, die um Zwei kleiner ist als die Referenzstufe der linken Seite.* Die Referenzstufe rechts muß um Eins kleiner sein als links; wenn das nicht der Fall ist, wird rechts implizit dereferenziert.

2.2 Vergleich von Zeigern, Leerzeiger

Wegen der impliziten Dereferenzierung muß man (auch) in PEARL beim Umgang mit Zeigern sehr sorgfältig sein. Betrachten wir zunächst die Floskel `IF a == 5 THEN`. Wir meinen damit, daß der `THEN`-Zweig ausgeführt werden soll, falls der Variablenwert der Variablen `a` gleich dem Zahlenwert `5` ist. Die Variable `a` wird also bei der Auswertung des Vergleiches implizit dereferenziert (`IF a/*1->0*/ == 5 THEN`); auf beiden Seiten vom `==` müssen Zahlenwerte genommen werden. Deshalb wäre auch `IF 5 == 6 THEN` formal richtig, obwohl das logisch natürlich Unsinn ist, weil 5 niemals 7 sein kann.

In der Floskel `IF a == b THEN` wird auf beiden Seiten implizit dereferenziert (`IF a/*1->0*/ == b/*1->0*/ THEN`); falls die Zeigerinhalte der Zeiger `x` und `y` die Variablennamen `a` bzw. `b` sind, können wir statt dessen auch schreiben `IF CONT x == CONT y THEN`; dann wird auf beiden Seiten des `==` je einmal explizit und einmal implizit dereferenziert. Wir dürfen sogar schreiben `IF x == y THEN`, weil dann zweimal implizit dereferenziert wird (`IF x/*2->1->0*/ == y/*2->1->0*/ THEN`).

Bei vielen Aufgabenstellungen in der Praxis muß untersucht werden, ob zwei Zeiger den gleichen Variablennamen als Zeigerinhalt haben. Dazu gibt es die Vergleichsoperatoren `IS` und `ISNT`. Wir dürfen also schreiben `IF x IS y THEN`, wenn wir wissen wollen, ob die Zeigerinhalte von `x` und `y` der selbe Variablenname sind. Auch hier wird wieder implizit dereferenziert(`IF x/*2->1*/ IS y/*2->1*/ THEN`), weil ja Zeigerinhalte verglichen werden; genauer dürfen wir deshalb auch `IF CONT x IS CONT y THEN` schreiben. Um zu ermitteln, ob der Zeiger `x` auf die Variable `a` zeigt, können wir `IF CONT x IS a THEN` oder `IF x IS a THEN` (das heißt `IF x/*2->1*/ IS a THEN`) programmieren.

Auf beiden Seiten von IS bzw. ISNT müssen *Zeigerinhalte*, also Variablennamen (Referenzstufe 1) stehen. Formal richtig dürfen wir daher auch schreiben `IF a IS b THEN`, obwohl das genau so unsinnig ist wie `IF 5 == 7 THEN`.

Um prüfen zu können, ob ein Zeiger überhaupt auf etwas zeigt, gibt es einen Zeiger `NIL`, der definiert ins Leere zeigt. Man sollte es sich zur Regel machen, Zeigern gleich bei der Deklaration (mit einer INIT-Klausel) oder sofort danach durch Zuweisung von NIL einen definierten Wert zu geben.

Es ist sehr empfehlenswert, beim Vergleich der Zeigerinhalte von Zeigern untereinander oder mit einem Variablennamen immer explizit zu dereferenzieren. Wir erreichen damit, daß dann für uns die Regel gilt: *Mindestens auf einer Seite von* `IS` *oder* `ISNT` *muß* `CONT` *stehen.* Unsinnige Vergleiche von zwei Variablennamen `IF a IS b THEN` können dadurch entdeckt werden, weil ja hier auf keiner Seite `CONT` steht. Und wenn wir aus Versehen `CONT` vor einen Variablennamen geschrieben haben, wird das vom Kompilierer als Fehler gemeldet.

2.3 Zeiger als Prozedurparameter und Funktionsergebnisse

Zeiger dürfen wie Variable auch per Wertübergabe oder per Namensübergabe in Prozeduren übergeben werden, Zeigerinhalte (also Variablennamen) dürfen Ergebnisse von Funktionsprozeduren sein wie in Beispiel 1:

```
trivial: PROC(z REF FIXED)RETURNS(REF FIXED);
  RETURN(CONT z);
END;

CONT trivial(x):=5;       ! implizite Dereferenzierung von x
CONT trivial(CONT x):=5;  ! explizite Dereferenzierung von x
CONT trivial(a):=5;       ! Aufruf mit Zeigerinhaltskonstanten
```

Beispiel 1: Funktionsprozedur mit Zeigern und mögliche Aufrufe

Die Funktion gibt den Zeigerinhalt von `z` zurück; wir können sie dazu benutzen, um der Variablen, auf die der Zeiger x gerade zeigt, einen Wert zuzuweisen. Weil das Funktionsergebnis ein Zeiger ist, (genauer gesagt dessen Zeigerwert), müssen wir explizit mit `CONT` dereferenzieren, wenn wir die Funktion links vom `:=` benutzen. Da wir im Prozedurkopf Wertübergabe vereinbart haben, wird der Parameter `x` beim ersten Aufruf implizit dereferenziert, beim zweiten explizit, und beim dritten Aufruf wird ein Zeigerinhalt (ein Variablenname) als aktueller Parameter übergeben.

Noch eine Bemerkung zur Namensübergabe: Wenn wir eine simple Prozedur schreiben wollen, die einer Variablen den Wert 5 gibt, könnte sie wie die erste Prozedur in Beispiel 2 aussehen. Wir dürfen sie aber auch wie die zweite Prozedur schreiben; der Aufruf zum Ändern der Variablen `a` ust jedoch für beide Fassungen gleich.

```
fuenfzuweisung:PROC(c FIXED IDENT);
   c:=5;
END;


fuenfzuweisung:PROC(c REF FIXED);
   CONT c:=5;
END;

fuenfzuweisung(a);
```

Beispiel 2: Zwei Versionen einer Prozedur und ihr Aufruf

3 Verkettete Listen

3.1 Einfache Verkettung

Im allgemeinen benutzt man in der Praxis kaum Zeiger auf einfache Variable, sondern man baut mit ihrer Hilfe Datenstrukturen auf, die einen bequemeren und schnelleren Zugriff auf die in ihnen enthaltenen Daten erlauben, wie z. B. verkettete Listen und Binärbäume. Nehmen wir als Beispiel eine doppelt verkettete Liste, die verschiedenen Texte von Meldungen enthält. Die zugehörigen Typ- und Variablenvereinbarungen zeigt Beispiel 3.

```
TYPE MELDUNG CHAR(20);
TYPE LISTENELEMENT STRUCT[meldung MELDUNG,
                          nAEchstes REF LISTENELEMENT];
DCL leeranker LISTENELEMENT INIT(' ',NIL),
    meldungsanker LISTENELEMENT INIT(' ',NIL),
    elementfeld (100) LISTENELEMENT;
```

Beispiel 3: Typvereinbarungen und Deklarationen für verkettete Meldungsliste

In Beispiel 3 ist der neue Datentyp `LISTENELEMENT` vereinbart, der jeweils eine Meldung und den Zeiger `nAEchstes` auf eine Variable vom Typ `LISTENELEMENT` aufnehmen kann. Bemerkenswert an dieser Typvereinbarung ist die Tatsache, daß in ihr der Typbezeichner `LISTENELEMENT` zweimal vorkommt: die Typvereinbarung von `LISTENELEMENT` enthält eine Rekursion. Die merkwürdige Schreibweise von nAEchstes soll uns dabei daran erinnern, daß wir hier einen Zeiger haben.

Mit Hilfe dieses neuen Datentyps sind zwei Variable `leeranker` und `meldungsanker` vereinbart. Sie sollen die ersten Glieder von einer Kette von Listenelementen sein. Außerdem gibt es ein ganz normales Feld von Listenelementen. Die beiden Variablen sind außerdem bei ihrer Vereinbarung so vorbesetzt worden, daß ihr Zeigerteil definiert ins Nichts zeigt.

Diese Listenelemente des Feldes sollen nun so verkettet werden, daß der `leeranker` auf eines der Listenelemente zeigt, dieses wieder auf ein anderes, letzteres wieder auf eines, usw. für alle Listenelemente des Feldes. Dazu schreiben wir eine Prozedur `einketten`, die in Beispiel 4 gezeigt ist.

```
einketten:PROC(eINzukettendes INV REF LISTENELEMENT,
               vORgaenger INV REF LISTENELEMENT);
  eINzukettendes.nAEchstes:=CONT vORgaenger.nAEchstes;
  vORgaenger.nAEchstes:=CONT eINzukettendes;
END;
```

Beispiel 4: Prozedur zum Einketten eines Listenelementes in eine Liste

Der Prozedur werden zwei Zeiger auf Listenelemente per Wertübergabe übergeben; das heißt, daß wir als Parameter bei ihrem Aufruf auch Namen von Listenelementen verwenden können. Die Prozedur trennt eine Kette `-->Vorgänger-->Nachfolger-->` hinter dem `Vorgänger` auf und schiebt das Listenelement `Einzukettendes` dazwischen: `-->Vorgänger-->Einzukettendes-->Nachfolger-->`

Um die Prozedur zu verstehen, müssen wir uns vor allem die Zuweisung

```
eINzukettendes.nAEchstes:=CONT vORgaenger.nAEchstes;
```

anschauen. Ganz links steht der Zeigername `eINzukettendes`. Dieser Zeiger hat den Namen eines Listenelementes als Zeigerinhalt, der beim Aufruf der Prozedur als Parameter übergeben worden ist. Dieser Listenelement-Name wird implizit dereferenziert, um an den Variableninhalt des Listenelementes zu gelangen. In dessen Teil `nAEchstes`, der ja einen Zeigerinhalt aufnehmen soll (Referenzstufe 2), wird ein Zeigerinhalt geschrieben, der rechts vom `:=` folgendermaßen ermittelt wird: Der Zeigerinhalt von `vORgaenger`, der ebenfalls beim Prozeduraufruf als Parameter über-

geben worden ist, wird ebenfalls implizit dereferenziert, um an den Variableninhalt zu kommen. Dessen Teil `nAEchstes` enthält einen Zeigerinhalt, auf den mit CONT zugegriffen wird, und der dann nach links kopiert wird. Wir könnten mit unserer privaten Notation also schreiben

```
eINzukettendes/*2->1*/.nAEchstes:=CONT vORgaenger/*2->1*/.nAEchstes;
```

Auf beiden Seiten der Zuweisung wird also ein Zeigername (`eINzukettendes` bzw. `vORgaenger`) implizit dereferenziert. Der Compiler erkennt jeweils am Punkt, der hinter dem Zeigernamen steht, daß er den Zeigerinhalt nehmen muß.

Wir können übrigens in der Prozedur die beiden `CONT` weglassen. Sie sind nur hingeschrieben worden, um zu dokumentieren, daß rechts vom `:=` Zeigerinhalte stehen. Desgleichen können auch die beiden `INV` im Prozedurkopf weggelassen werden. Sie stehen dort ebenfalls zur besseren Dokumentation: die beiden Zeiger werden ja in der Prozedur nicht verändert, sondern nur ihre Inhalte.

```
FOR index TO UPB elementfeld REPEAT
  einketten(elementfeld(index),leeranker);
END;
```

Beispiel 5: Benutzung der Prozedur zum Verketten der Listenelemente

Mit Hilfe dieser Prozedur können wir jetzt die Listenelemente des Elementfeldes verketten. Die dazu nötige REPEAT-Schleife zeigt Beispiel 5. In ihr werden die Listenelemente der Einfachheit halber so verkettet, daß `leeranker` auf `elementfeld(100)` zeigt. `elementfeld(1)` zeigt ins Nichts.

```
ausketten: PROC(vORgaenger INV REF LISTENELEMENT)
                RETURNS(REF LISTENELEMENT);
  DCL aUSzukettendes REF LISTENELEMENT;
  aUSzukettendes:=CONT vORgaenger.nAEchstes;
  vORgaenger.nAEchstes:=CONT aUSzukettendes.nAEchstes;
  aUSzukettendes.nAEchstes:=NIL;
  RETURN(CONT aUSzukettendes);
END;
```

Beispiel 6: Funktionsprozedur zum Ausketten eines Listenelementes

Wenn wir jetzt eine Meldung speichern wollen, müssen wir ein Listenelement aus dieser Kette entnehmen, die Meldung hineinschreiben und das Listenelement in die Kette mit dem Anker

`meldungsanker` einketten. Wir brauchen daher noch eine Prozedur zum Ausketten eines Listenelementes. Sie wird in Beispiel 6 gezeigt.

Sie ist als Funktionsprozedur geschrieben worden. Wenn wir nämlich ein Listenelement aus unserer einfach verketteten Liste ausketten wollen, brauchen wir seinen Vorgänger, damit aus der Kette `-->Vorgänger-->Auszukettendes-->Nachfolger-->` die Kette `-->Vorgänger-->Nachfolger-->` wird. Deshalb haben wir der Prozedur einen Zeiger auf den Vorgänger übergeben, und der Zeiger auf dessen nächstes Element, nämlich auf das ausgekettete, wird uns zurückgegeben.

```
eLEmentzeiger:=CONT ausketten(leeranker);
eLEmentzeiger.meldung:=neue_meldung;
einketten(eLEmentzeiger,meldungsanker);
```

Beispiel 7: Anweisungsfolge zum Einketten einer neuen Meldung in die Liste

Beispiel 7 zeigt eine Anweisungsfolge, die unsere beiden Prozeduren benutzt, wenn eine neue Meldung in die Kette der vorhandenen eingekettet werden soll. Der Einfachheit halber wird sie direkt hinter dem Anker eingekettet; die Meldungskette bildet so eine first-in-first-out-Schlange.

3.2 Vermeidung von Fehlern durch Gebrauch von CONT

Jetzt wollen wir uns mit einer Prozedur beschäftigen, die alle Meldungen ausgibt, die in der Liste stehen. Sie wird in Beispiel 8 gezeigt. Wieder ist an allen Stellen, an denen `INV` oder `CONT` stehen darf (aber weggelassen werden kann), das auch hingeschrieben worden.

```
meldungsliste: PROC(aNKer INV REF INV LISTENELEMENT);
  DCL lAUfzeiger REF INV LISTENELEMENT;
  lAUfzeiger:=CONT aNKer.nAEchstes;
  WHILE CONT lAUfzeiger ISNT NIL REPEAT
     PUT lAUfzeiger.meldung TO display BY A,SKIP;
     lAUfzeiger:=CONT lAUfzeiger.nAEchstes;
  END;
END;
```

Beispiel 8: Prozedur zur Ausgabe aller Meldungen aus einer verketteten Liste

```
meldungsliste: PROC(aNKer REF LISTENELEMENT);
  DCL lAUfzeiger LISTENELEMENT;
  lAUfzeiger:=aNKer.nAEchstes;
  WHILE lAUfzeiger.nAEchstes ISNT NIL REPEAT
     PUT lAUfzeiger.meldung TO display BY A,SKIP;
     lAUfzeiger:=lAUfzeiger.nAEchstes;
  END;
END;
```

Beispiel 9: Fehlerhafte Prozedur zur Ausgabe aller Meldungen aus einer verketteten Liste

Beispiel 9 zeigt die Prozedur in nur etwas veränderter Form. Alle die `INV` und `CONT` sind weggelassen worden, und `lAUfzeiger` ist jetzt laut Vereinbarung kein Zeiger mehr, sondern eine normale Variable; das REF sei bei der Vereinbarung schlicht vergessen worden. Die Prozedur wird vom Compiler ohne Meldung von Fehlern übersetzt, aber bei ihrem Test meldet das Rechnersystem `segmentation violation Segmentation fault`. Warum?

Der Fehler liegt in der Zuweisung `lAUfzeiger:=lAUfzeiger.nAEchstes;`. Dort steht links vom `:=` wegen des vergessenen `REF` ein Variablenname, also die Referenzstufe 1. Rechts muß daher ein Variableninhalt (Referenzstufe 0) stehen, und zwar vom gleichen Typ wie links. `lAUfzeiger.nAEchstes` gleich rechts vom `:=` ist aber ein Zeigername (Referenzstufe 2). Deshalb wird dieser Zeigername zweimal implizit (`lAUfzeiger:=lAUfzeiger.nAEchstes/*2->1->0*/;`) dereferenziert. Beim letzten Listenelement der Kette steht als Inhalt von `lAUfzeiger.nAEchstes` aber der Inhalt des Nullzeigers NIL, der irgendwohin ins Leere zeigt. Beim Versuch, den Inhalt von diesem "Irgendwohin" zu holen, passiert dann der Fehler: es wird versucht, auf einen Speicherbereich zuzugreifen, auf den der Zugriff verboten ist.

In der ursprünglichen Version der Prozedur (Beispiel 8) hätte man natürlich auch in der Deklaration `DCL lAUfzeiger REF INV LISTENELEMENT;` das `REF` vergessen können; dann hätte aber der Compiler sowohl bei `WHILE CONT lAUfzeiger ISNT NIL REPEAT` als auch bei der Zuweisung einen Fehler gemeldet. Selbst wenn `REF INV` vergessen worden wären, wäre der Fehler dann bei `WHILE CONT lAUfzeiger ISNT NIL REPEAT` gemeldet worden, weil `lAUfzeiger` wegen des vergessenen `REF` ein Variablenname ist, bei dem `CONT` keinen Sinn macht.

3.3 Ringverkettung, doppelte Verkettung

In der Prozedur `ausketten` aus Beispiel 6 wird der Vorgänger desjenigen Listenelementes benötigt, das ausgekettet werden soll. Wenn wir ein Listenelement ausketten wollen, dessen Vorgänger wir zur Zeit nicht kennen, müssen wir letzteren bestimmen. Dazu könnten wir eine Funktionsprozedur `vORgaenger` mit dem Aufruf z.B. `vORgaenger(eLEmentzeiger,aNKerzeiger)` benutzen, der wir Zeiger auf das auszukettende Element und auf den Anker der Kette übergeben, in

der sich das Element befindet, und die uns den Zeiger auf den Vorgänger zurückgibt. Noch geschickter können wir dieses Ziel erreichen, wenn wir die Kette nicht beim letzten Element enden lassen, sondern dort mit einem Zeiger auf den Anker schließen. Wir müssen dann zunächst dafür sorgen, daß in die beiden Anker ein Zeiger auf sich selbst kommt: `leeranker.nAEchstes:=leeranker;` und `meldungsanker.nAEchstes:=meldungsanker;`

Dann muß die Prozedur `meldungsliste` geändert werden. Beispiel 10 zeigt sie für Ringverkettung.

```
meldungsliste: PROC(aNKer INV REF INV LISTENELEMENT);
  DCL lAUfzeiger REF INV LISTENELEMENT;
  lAUfzeiger:=CONT aNKer.nAEchstes;
  WHILE CONT lAUfzeiger ISNT CONT aNKer REPEAT
     PUT lAUfzeiger.meldung TO display BY A,SKIP;
     lAUfzeiger:=CONT lAUfzeiger.nAEchstes;
  END;
END;
```

Beispiel 10: Prozedur zur Ausgabe aller Meldungen bei Ringverkettung der Meldungsliste

Schließlich können wir eine Prozedur zur Suche nach dem Zeiger auf den Vorgänger eines Listenelementes schreiben. Sie steht in Beispiel 11. Unsere Anweisung zum Ausketten des ersten Elementes aus der Meldungsliste `(eLEmentzeiger:=CONT ausketten(leeranker);` können wir jetzt auch `eLEmentzeiger:= CONT ausketten(CONT vORgaenger(CONT leeranker.nAEchstes));` schreiben.

```
vORgaenger : PROC(eLEmentzeiger INV REF INV LISTENELEMENT )
                 RETURNS(REF LISTENELEMENT);
  DCL lAUfzeiger REF INV LISTENELEMENT;
  lAUfzeiger:=CONT eLEmentzeiger.nAEchstes;
  WHILE CONT lAUfzeiger.nAEchstes ISNT CONT eLEmentzeiger REPEAT
     lAUfzeiger:=CONT lAUfzeiger.nAEchstes;
  END;
  RETURN(CONT lAUfzeiger);
END;
```

Beispiel 11: Prozedur zur Suche nach dem Vorgänger einer Meldung

Die nächste mögliche Verbesserung unseres Programmes bestände darin, daß wir die Listen nicht mehr einfach, sondern doppelt verketten, so daß jedes Element nicht nur einen Zeiger auf den Nachfolger, sondern auch einen auf den Vorgänger enthält. Dann ist nämlich das Ausketten

eines Elementes besonders einfach, weil wir nicht erst nach dem Vorgänger suchen müssen. Beispiel 12 zeigt dieses Ausketten. Andererseits ist das Einketten dann etwas umfänglicher, weil nicht nur vorwärts, sondern auch in Gegenrichtung eingekettet werden muß. Die Prozedur wird jetzt anders aufgerufen als in Beispiel 7. Deren erste beide Zeilen müssen ersetzt werden durch die Anweisungsfolge `eLEmentzeiger:=CONT leeranker.nAEchstes; ausketten(CONT eLEmentzeiger);`

```
ausketten: PROC(eLEmentzeiger INV REF LISTENELEMENT);
  eLEmentzeiger.vORiges.nAEchstes:=CONT eLEmentzeiger.nAEchstes;
  eLEmentzeiger.nAEchstes.vORiges:=CONT eLEmentzeiger.vORiges;
  eLEmentzeiger.nAEchstes:=eLEmentzeiger.vORiges:=NIL;
END;
```

Beispiel 12: Prozedur zum Ausketten einer Meldung aus doppelter Verkettung

Um noch einmal zu erfahren, wie wichtig es ist, bei der Programmierung mit Zeigern mit Hilfe von `INV` und `CONT` genau niederzuschreiben, was gemeint ist, wollen wir in Beispiel 13 die Version von Beispiel 10 betrachten, in der diese Schlüsselwörter gestrichen sind, die aber trotzdem ohne Fehlermeldung übersetzt wird.

```
meldungsliste: PROC(aNKer  REF  LISTENELEMENT);
  DCL lAUfzeiger REF  LISTENELEMENT;
  lAUfzeiger:= aNKer.nAEchstes;
  WHILE  lAUfzeiger ISNT  aNKer REPEAT
     PUT lAUfzeiger.meldung TO display BY A,SKIP;
     lAUfzeiger:= lAUfzeiger.nAEchstes;
  END;
END;
```

Beispiel 13: Schlampig geschriebene Prozedur zur Ausgabe aller Meldungen

Wenn wir in dieser Prozedur in der Deklaration REF vergessen und schreiben `DCL lAUfzeiger LISTENELEMENT;`, wird die Prozedur ebenfalls ohne Fehlermeldung übersetzt. Beim Programmtest werden wir jedoch feststellen, daß die Liste der Meldungen immer wieder ausgegeben wird, daß wir also eine Endlosschleife programmiert haben. Der Grund: in der Zeile `WHILE lAUfzeiger ISNT aNKer REPEAT` wird jetzt untersucht, ob der Variablenname `lAUfzeiger` gleich dem Zeigerinhalt von `aNKer` (gleich dem Variablennamen des Kettenankers) ist, was natürlich nie der Fall ist. Ein Programmierer, dem das nicht ganz klar ist, könnte jetzt die Prozedur so verbessern, wie es Beispiel 14 zeigt. Dann läuft sie im Test fehlerlos. Bei genauem Hinblicken unterscheidet sie sich aber dahingehend von Beispiel 13, daß jetzt in der Zeile `lAUfzeiger:= lAUfzeiger.nAEchstes;` nicht mehr

ein Zeiger, sondern ein ganzes Listenelement zugewiesen wird, was natürlich viel mehr Rechenzeit erfordert.

```
meldungsliste: PROC(aNKer  REF  LISTENELEMENT);
  DCL lAUfzeiger   LISTENELEMENT;
  lAUfzeiger:= aNKer.nAEchstes;
  WHILE  lAUfzeiger.nAEchstes ISNT  aNKer.nAEchstes REPEAT
     PUT lAUfzeiger.meldung TO display BY A,SKIP;
     lAUfzeiger:= lAUfzeiger.nAEchstes;
  END;
END;
```

Beispiel 14: Falsche Verbesserung der Prozedur bei falscher Deklaration von `lAUfzeiger`

Falls wir die Prozedur so ordentlich schreiben wie in Beispiel 10, wäre uns bei der falschen Deklaration `DCL lAUfzeiger LISTENELEMENT;` der Fehler in der Zeile `WHILE CONT lAUfzeiger ISNT CONT aNKer REPEAT` gemeldet worden, weil `CONT` nicht vor einer normalen Variablen stehen darf.

Das Beispiel 7 enthält übrigens auch noch einen Fehler: bei Ausführung der Anweisung `eLEmentzeiger:=CONT ausketten(leeranker);` könnte es sein, daß die Liste, die mit `leeranker` beginnt, außer dem Anker keine Elemente mehr enthält, so daß `leeranker.nAEchstes` den Wert `NIL` hat. Die Folge würde sein, daß in der Prozedur `ausketten` (Beispiel 6) die Variable `aUSzukettendes` den Wert `NIL` bekäme und deshalb in der Anweisung `vORgaenger.nAEchstes := CONT aUSzukettendes.nAEchstes;` ein Laufzeitfehler `segmentation violation Segmentation fault` auftreten würde.

4 Sonderfälle von Zeigern

4.1 Umgehung der Typbindung

Es kann ohne weitere Maßregeln ziemlich mühsam sein, in einem längeren Programm die Anweisung zu finden, in der ein Laufzeitfehler infolge falsch gesetzten Zeigerinhalts ausgelöst wird, und die eigentliche Ursache zu erkennen. Glücklicherweise gibt es in PEARL eine Möglichkeit, sich die Inhalte von Zeigervariablen (das heißt, die Adressen von normalen Variablen) als Zahlenwerte ausgeben zu lassen; `NIL` hat bei einer derartigen Ausgabe dann z. B. den Zahlenwert Null.

Normalerweise sind in PEARL Zeigervariable fest an einen ganz bestimmten Datentyp gebunden, und der Kompilierer meldet einen Fehler, wenn man z. B. versuchen würde, einer `REF FLOAT`-Zeigervariablen den Namen einer `FIXED`-Variablen zuzuweisen. Für Sonderzwecke ist es jedoch möglich, Zeigervariable zu vereinbaren, die formal auf Strukturen beliebigen Inhaltes, also auf alles zeigen können. Das geschieht z. B. durch eine Vereinbarung `aLLeszeiger REF STRUCT[];`.

Die leeren eckigen Klammern hinter `STRUCT` deuten dabei an, daß es auf den Inhalt der Struktur nicht ankommen soll, und daß man diesem Zeiger den Namen eines Objektes von beliebigem Typ zuweisen darf, ohne daß der Kompilierer bei der Übersetzung des Programmes dagegen protestiert. Deshalb kann man in den Speicherplatz für eine Variable, auf die ein Zeiger `REF STRUCT[];` zeigt, auch Variablen ganz anderen Typs, also mit ganz anderem Bedarf an Speicherplatz zuweisen. Das kann natürlich zu sehr schweren Fehlern führen. Deshalb gibt es den Operator `SIZEOF`, mit dem der Platzbedarf für eine Variable fesstellbar ist.

Beispiel 14 zeigt, wie man einen derartigen Zeiger zur Ausgabe von Variablenadressen benutzen kann.

```
zeigerwertausgabe: PROC(text CHAR(30),eLEmentzeiger INV REF INV LISTENELEMENT);
   TYPE REFSTRUKTUR STRUCT[zEIger REF INV LISTENELEMENT];
   TYPE FIXEDSTRUKTUR STRUCT[zahl INV FIXED(31)];
   DCL fixedstruktur FIXEDSTRUKTUR;
   DCL refstruktur REFSTRUKTUR;
   DCL aLLeszeiger REF STRUCT[] INIT(refstruktur);
   DCL fIXedzeiger REF FIXEDSTRUKTUR;
   IF SIZEOF fixedstruktur /= SIZEOF refstruktur THEN
      PUT 'refstruktur und fixedstruktur sind ungleich lang'
               TO display BY A,SKIP;
   FIN;
   refstruktur.zEIger:=eLEmentzeiger;
   fIXedzeiger:=aLLeszeiger;
   PUT text,fIXedzeiger.zahl TO display BY A,F(10),SKIP;
END;
```

Beispiel 15: Prozedur zur Ausgabe von Zeigerwerten (Variablenadressen)

In der Prozedur sind zwei Datentypen `REFSTRUKTUR` bzw. `FIXEDSTRUKTUR` vereinbart und entsprechende Variable `refstruktur` bzw. `fixedstruktur` deklariert. Diese beiden Variablen sind exakt gleich lang, wie in der Verzweigung `IF SIZEOF fixedstruktur...` geprüft wird. Sie unterscheiden sich nur dadurch, daß ihre erste und einzige Komponente in `refstruktur` einen Zeigerinhalt und in `fixedstruktur` eine `FIXED(31)`-Zahl aufnehmen kann. Außerdem gibt es zwei Zeiger `aLLeszeiger` und `fIXedzeiger`, wobei `aLLeszeiger` auf `refstruktur` zeigt.

Die Prozedur übernimmt als zweiten Parameter den Zeiger `eLEmentzeiger` und weißt ihn `refstruktur.zEIger:=eLEmentzeiger;` zu. Dann bekommen (`fIXedzeiger := aLLeszeiger;`) beide Zeiger den selben Zeigerninhalt. Deshalb zeigen sie letzendlich auch auf die selben Daten. Diese Daten haben aber eine doppelte Bedeutung: sie stellen beide Male einen Verbund dar; nur wird dessen erste und einzige Komponente einmal (beim Zugriff durch `refstruktur.zEIger`) als Zeigervariable und zum anderen (beim Zugriff durch `fIXedzeiger.zahl`) als `FIXED(31)`-Variable interpretiert.

Deshalb ist es der Prozedur jetzt möglich, den Wert der Zeigervariablen (die Adresse einer normalen Variablen) als `FIXED(31)`-Wert auszugeben. Außerdem gibt sie als Erläuterung den Text aus, den sie als ersten Parameter übernommen hat.

Bei der Suche nach einem falschen Zeigerinhalt würde man im Test diese Prozedur an allen Stellen aufrufen, an denen man den Fehler vermutet, und bekäme dadurch die Information geliefert, die man zu seiner Beseitigung braucht.

Auf ganz ähnliche Art und Weise kann man übrigens auch eine `FLOAT`-Variable mit einem Feld von Bitketten überlagern, um zu studieren, wie der Computer `FLOAT`-Werte intern darstellt.

4.2 Zeiger auf Zeichenketten verschiedener Länge

```
   beispiel: PROC;
      DCL zeile CHAR(40) INIT('----------------------------------------');
      DCL zEIlenzeiger REF  CHAR();
      DCL (maxlaenge,aktlaenge) FIXED(31);
      zEIlenzeiger:=zeile;
      CONT zEIlenzeiger:='FH Bielefeld';
      maxlaenge:=SIZEOF zEIlenzeiger MAX;
      aktlaenge:=SIZEOF zEIlenzeiger LENGTH;
      PUT 'Zeilendaten: maxlaenge=',maxlaenge,',aktlaenge= ',aktlaenge
         TO display BY A,F(5),A,F(5),SKIP;
      PUT CONT zEIlenzeiger,'!' TO display BY 2 A,SKIP;
      PUT zeile,'!' TO display BY 2 A,SKIP;
   END;
------------------------------------------------------------------------
Zeilendaten: maxlaenge=   40,aktlaenge=    12
FH Bielefeld!
FH Bielefeld----------------------------!
```

Beispiel 16: Prozedur zur Benutzung von REF CHAR() und Ergebnis

Zeiger, die mit `REF STRUCT[]` vereinbart worden sind, bilden eine Ausnahme von der Regel, daß man bei PEARL90 mit Zeigern nur auf einen einzigen Datentyp zeigen kann; z. B. kann nach `DCL zEIlenzeiger REF CHAR(80);` der `zEIlenzeiger` nur auf Zeichenketten genau dieser Länge zeigen, und wenn man der Zeile, auf die er zeigt, eine Zeichenkette zuweist (`CONT zEIlenzeiger:='FH Bielefeld';`), dann wird die Zeile rechts mit Leerzeichen aufgefüllt.

Um bequemer mit Zeichenketten arbeiten zu können, gibt es deshalb auch Zeichenkettenzeiger, die auf Ketten beliebiger Länge zeigen dürfen: `DCL zEIlenzeiger REF CHAR();`. Wenn einem derartigen Zeiger der Name einer Zeichenkette zugewiesen wird, dann wird zusätzlich über die Gesamtlänge der Zeichenkette und ihren aktuellen Füllungsstand Buch gefüllt. Beispiel 16 soll das verdeutlichen. In ihm sind die maximale und die aktuelle Länge der Zeichenkette `zeile`, auf die der Zeiger `zEIlenzeiger` zeigt, durch den Operator `SIZEOF` mit dem Zusatz `MAX` bzw. `LENGTH` bestimmt worden. Bei der Ausgabe des Zeileninhaltes mittels zEIlenzeiger wird nur der aktuelle Inhalt ausgegeben, bei der Ausgabe von Zeile zusätzlich auch der Rest.

```
   einlesebeispiel: PROC;
      DCL zeile CHAR(40) INIT('----------------------------------------');
      DCL zEIlenzeiger REF  CHAR();
      DCL (maxlaenge,aktlaenge,laenge) FIXED(31);
      zEIlenzeiger:=zeile;
      PUT 'Gib Zeile fuer einlesebeispiel: ' TO display BY A;
      GET CONT zEIlenzeiger FROM tasten BY S(laenge);
      GET FROM tasten BY SKIP;
      maxlaenge:=SIZEOF zEIlenzeiger MAX;
      aktlaenge:=SIZEOF zEIlenzeiger LENGTH;
      PUT 'Zeilendaten: maxlaenge=',maxlaenge,',aktlaenge= ',aktlaenge
         TO display BY A,F(5),A,F(5),SKIP;
      PUT CONT zEIlenzeiger,'!' TO display BY 2 A,SKIP;
      CONT zEIlenzeiger:=zEIlenzeiger.CHAR(1:laenge);
      maxlaenge:=SIZEOF zEIlenzeiger MAX;
      aktlaenge:=SIZEOF zEIlenzeiger LENGTH;
      PUT 'Zeilendaten: maxlaenge=',maxlaenge,',aktlaenge= ',aktlaenge
         TO display BY A,F(5),A,F(5),SKIP;
      PUT CONT zEIlenzeiger,'!' TO display BY 2 A,SKIP;
   END;
------------------------------------------------------------------------
Gib Zeile fuer einlesebeispiel: FH Bielefeld
Zeilendaten: maxlaenge=   40,aktlaenge=    40
FH Bielefeld                           !
Zeilendaten: maxlaenge=   40,aktlaenge=    12
FH Bielefeld!
```

Beispiel 17: Prozedur zum Einlesen mit REF CHAR() und Ergebnis

In der Zuweisung `CONT zEIlenzeiger:='FH Bielefeld';` wird die Zeile, auf die `zEIlenzeiger` zeigt, von links mit dem Text `FH Bielefeld` gefüllt. Wichtig ist bei dem kurzen Beispiel vor allen Dingen, daß die Zuweisung mit Hilfe des Zeigers erfolgt; dadurch wird das System veranlaßt, die Anzahl der Zeichen zu bestimmen und beim Zeiger `zEIlenzeiger` einzutragen. Hätte die Anweisung `zeile:='FH Bielefeld';` gelautet, dann wäre die Zeile rechts mit Leerzeichen aufgefüllt worden, und die aktuelle Länge des Zeileninhaltes wäre deshalb 40 gewesen.

Auch die Ausgabe des aktuellen Inhaltes (und eines darauf folgenden Ausrufezeichens) in der vorletzten Anweisung der Prozedur klappt nur so, wenn sie mit Hilfe von zEIlenzeiger erfolgt; in der letzten Anweisung wird der Inhalt von `zeile` direkt ausgegeben; wir sehen, daß nur der Anfang mit `FH Bielefeld` überschrieben worden ist. Die Zeile hat übrigens bei ihrer Deklaration als Inhalt lauter Bindestriche bekommen, um diesen Sachverhalt zu verdeutlichen; wenn sie ohne INIT-Klausel vereinbart worden wäre, hätte irgendein zufälliger Inhalt hinter `FH Bielefeld` gestanden, der möglicherweise Steuersequenzen entahlten und die Ausgabe völlig verstümmelt hätte.

Beim Einlesen einer Zeichenkette in eine `CHAR`-Variable wird bekanntlich die Variable rechts mit Leerzeichen aufgefüllt, wenn die eingelesene Zeichenkette kürzer ist als die Variable; man kann jedoch die Anzahl Zeichen in der Zeichenkette durch Benutzung des S-Formates beim Einlesen bestimmen. Beispiel 17 zeigt, wie das benutzt werden kann, um in einem CHAR()-Zeiger die wirkliche Länge einer Zeichenkette zu setzen: nach dem Einlesen sind zunächst aktuelle und maximale Länge identisch 40; durch die Zuweisung `CONT zEIlenzeiger:=zEIlenzeiger.CHAR(1:laenge);` wird die aktuelle Länge entsprechend gesetzt.

4.3 Zeiger auf Zeichenketten-Konstanten

Eine weitere Besonderheit stellen Zeiger dar, die auf Zeichenketten-Konstanten zeigen sollen. Beispiel 18 zeigt eine kurze Prozedur, die die Eigenschaften eines derartigen Zeigers veranschaulicht. Er wird mit `DCL kONstzeiger REF INV CHAR();` vereinbart. Diesem Zeiger wird der Name der Zeichenketten-Konstanten `zeile` zugewiesen, die mit dem Inhalt `FH Bielefeld` vereinbart worden ist, so daß er auf diese Zeichenketten-Konstante zeigt. Dann werden die maximale und aktuelle Länge bestimmt und deren Werte und die Zeichenkette, auf die `kONstzeiger` zeigt, in den ersten beiden Zeilen des Ergebnisses ausgegeben.

Danach folgt die Zuweisung `kONstzeiger := 'Dies ist der neue Inhalt';`. Sie bewirkt, daß `kONstzeiger` jetzt auf diese andere Zeichenketten-Konstante zeigt, und daß die bei ihm mitgeführten Zähler die entsprechenden Zahlenwerte bekommen, wie die Zeilen drei und vier des Ergebnisses veranschaulichen.

```
   invchar: PROC;
      DCL (maxlaenge,aktlaenge) FIXED(31);
      DCL zeile INV CHAR(12) INIT('FH Bielefeld');
      DCL kONstzeiger REF INV CHAR();
      kONstzeiger:=zeile;
      maxlaenge:=SIZEOF kONstzeiger MAX;
      aktlaenge:=SIZEOF kONstzeiger LENGTH;
      PUT 'Zeilendaten: maxlaenge=',maxlaenge,',aktlaenge= ',aktlaenge
         TO display BY A,F(5),A,F(5),SKIP;
      PUT CONT kONstzeiger,'!' TO display BY 2 A,SKIP;
      kONstzeiger:='Dies ist der neue Inhalt';
      maxlaenge:=SIZEOF kONstzeiger MAX;
      aktlaenge:=SIZEOF kONstzeiger LENGTH;
      PUT 'Zeilendaten: maxlaenge=',maxlaenge,',aktlaenge= ',aktlaenge
         TO display BY A,F(5),A,F(5),SKIP;
      PUT CONT kONstzeiger,'!' TO display BY 2 A,SKIP;
      PUT 'zeile: ', zeile TO display BY 2 A,SKIP;
   END;
-----------------------------------------------------
Zeilendaten: maxlaenge=   12,aktlaenge=    12
FH Bielefeld!
Zeilendaten: maxlaenge=   24,aktlaenge=    24
Dies ist der neue Inhalt!
zeile: FH Bielefeld
```

Beispiel 18: Prozedur zur Benutzung von REF INV CHAR() und Ergebnis

Die Zuweisung `kONstzeiger := 'Dies ist der neue Inhalt';` enthält offenbar eine Besonderheit: auf ihrer linken Seite haben wir die Referenzstufe Zwei, rechts die Referenzstufe Null, was ja in anderem Zusammenhang verboten ist.

Ganz wichtig ist, daß durch die letztgenannte Zuweisung der Zeiger umgesetzt wird und nicht etwa (verbotenerweise) der Inhalt des Objektes `zeile` geändert wird, auf das er vor der Zuweisung gezeigt hatte. In den ausgegebenen Ergebnissen der Prozedur wird das durch die letzte Zeile deutlich, die den Inhalt von `zeile` enthält.

Wir hätten auch die Deklaration `DCL zeile INV CHAR(12) INIT('FH Bielefeld');` weglassen können und statt `kONstzeiger:=zeile;` auch sofort `kONstzeiger:='FH Bielefeld';` schreiben dürfen.

Bei vielen praktischen Anwendungen sollen durch eine Prozedur Texte ausgegeben werden, deren Länge von Fall zu Fall unterschiedlich sind, z. B. durch eine Prozedur `meldungsausgabe: PROC( meldung CHAR(40));`. Durch diesen Prozedurkopf wird festgelegt, daß die Meldungen höchstens 40

Zeichen haben dürfen; wenn im Aufruf z. B. `meldungsausgabe('Ganz kurz');` steht, wird dieser Text bei der Paramterübergabe in die Prozedur mit Leerzeichen zu insgesamt 40 Zeichen aufgefüllt. Um nur die 9 Zeichen auszugeben, müßten wir feststellen, wieviele Leerzeichen hinter dem z stehen und die `PUT`-Anweisung entsprechend schreiben. Um diesen Umstand zu sparen, können wir wieder den Mechanismus des `REF INV CHAR()` benutzen. Beispiel 19 zeigt die vollständige Prozedur, wieder mit der Bestimmung der aktuellen Zahlen für die Zeichenanzahlen.

```
    meldungsausgabe: PROC(mELdungszeiger REF INV CHAR());
      DCL (maxlaenge,aktlaenge) FIXED(31);
      maxlaenge:=SIZEOF mELdungszeiger MAX;
      aktlaenge:=SIZEOF mELdungszeiger LENGTH;
      PUT 'Meldungsdaten: maxlaenge=',maxlaenge,',aktlaenge= ',aktlaenge
         TO display BY A,F(5),A,F(5),SKIP;
      PUT CONT mELdungszeiger,'!' TO display BY 2 A,SKIP;
   END;
------------------------------------------------------------------------
Aufruf: meldungsausgabe('Ganz kurz');
Ergebnis:
Meldungsdaten: maxlaenge=    9,aktlaenge=     9
Ganz kurz!
```

Beispiel 19: Prozedur zur Parameterübergabe mit REF INV CHAR() und Ergebnis

Beim Aufruf der Prozedur wird `mELdungszeiger` auf den aktuellen Parameter, nämlich die Zeichenketten-Konstante `'Ganz kurz'` gesetzt und gleichzeitig die zu `mELdungszeiger` gehörenden Zähler auf den aktuellen Stand gebracht. In der PUT-Anweisung werden dehalb nur die Zeichen `'Ganz kurz'` und das Ausrufezeichen ausgegeben; letzteres dient wieder dazu, um anzuzeigen, daß wirklich keine Leerzeichen auf `'Ganz kurz'` folgen.

4.4 Zeiger auf Tasks und Prozeduren

Bisher haben wir uns mit Zeigern beschäftigt, die auf Daten zeigen. In PEARL90 dürfen Zeiger jedoch auch auf alle übrigen Objekte außer auf Zeiger selbst zeigen, also auch auf Semaphore, Bolts, Datenstationen, Prozeduren und Tasks. Das gibt uns die Möglichkeit, mit z. B. Hilfe von Zeigerfeldern viele Datenstionen in einer Schleife zu eröffnen. Da bei diesen Objekten keine Zuweisungen an den jeweiligen Namen möglich sind, brauchen wir beim Umgang mit derartigen Zeigern nicht so sorgfältig zu überlegen, ob explizit, implizit oder überhaupt nicht dereferenziert werden muß; nach der Vereinbarung `DCL tASkpointer INV REF TASK INIT(start);` ist es klar, daß durch `ACTIVATE tASkpointer;` die Task mit Namen `start` aktiviert werden soll. Der einzige Fehler, den wir beim

Umgang mit derartigen Zeigern leicht machen können, besteht darin, daß wir vergessen, dem Zeiger bei der Deklaration oder durch eine Zuweisung einen definierten Inhalt zu geben.

Die Adresse einer bestimmten Task können wir in PEARL90 mit Hilfe der Einbaufunktion TASK ermitteln. Sie läßt z. B. nach dem Aufruf `tASkzeiger:=TASK(start); tASkzeiger` auf die Task `start` zeigen.

Wir können damit auch innerhalb einer Prozedur feststellen, von welcher Task sie aufgerufen worden ist: wir brauchen nur den Zeiger `DCL tASkpointer REF TASK;` zu vereinbaren und diesem mit `tASkpointer:=TASK;` das Ergebnis der Einbaufunktion `TASK` zuzuweisen; ohne Angabe des Tasknamens liefert `TASK` nämlich die Adresse der Task, in welcher die Zuweisung steht.

Nach einer Vereinbarung `DCL pROcpointer REF PROC(REF INV CHAR());` und der Zuweisung `pROcpointer:=meldungsausgabe;` können wir die Prozedur `meldungsausgabe` auch benutzen, indem wir sie mit `pROcpointer('Ganz kurz');` aufrufen. Interessant bei dieser Verwendung von Zeigern auf Prozeduren ist, daß wir damit ein Mittel haben, auf einfache Weise während des Programmlaufes einmal die eine und dann eine andere Prozedur mit dem selben Zeigeraufruf zu benutzen, indem wir einfach durch Zuweisung den Zeiger auf eine andere Prozedur zeigen lassen. Selbstverständlich müssen dabei beide Prozeduren die selben formalen Parameter bezüglich Reihenfolge und Typ haben, damit sie mit den gleichen Parametern aufgerufen werden können. Man nennt diese Eigenschaft Polymorphie.

4.5 Modulglobale Zeiger auf lokale Variable

Normalerweise ist es verboten, mit Zeigern, die außerhalb einer Prozedur vereinbart worden sind, auf lokale Daten der Prozedur zu zeigen. Daten innerhalb einer Prozedur sollen ja von außen unsichtbar sein und können deshalb gelöscht werden, sobald die Prozedur verlassen wird. Der Speicherplatz, der für sie benötigt wurde, kann dann vom System für andere Zwecke benutzt werden. Die erwähnten Zeiger könnten deshalb auf Speicherplatz zeigen, der schon längst anderen Zwecken dient.

In PEARL können sich jedoch Tasks sehr lange in einer Prozedur aufhalten, wenn sie zum Beispiel suspendiert worden sind. Deshalb darf man in Sonderfällen auch von außen mit Zeigern auf lokale Variable einer Task oder Prozedur zeigen, solange man sicher ist, daß sich die zugehörige Task in einem Wartezustand befindet.

Nehmen wir zum Beispiel eine Prozedur zur Abgabe von Fehlermeldungen über Defekte in Externgeräten (z. B.: Drucker hat kein Papier mehr): Die Tasks sollen ihr einen derartigen Meldungs-

text übergeben. Der Text wird von der Prozedur ausgegeben und die Task suspendiert sich dann selbst, um auf die Beseitigung des Fehlers zu warten. Vorher werden jedoch Meldung und Adresse der Task (die ja mit einer Anweisung wie `tASKzeiger := TASK;` ermittelt werden kann) von der Prozedur zusammen in eine Liste eingekettet, die alle anstehenden Fehlermeldungen enthält. Wenn dann einer dieser Fehler behoben ist, wird diese Liste nach dem zugehörigen Meldungstext durchsucht, das betreffende Listenelement ausgekettet und die zugehörige Task fortgesetzt.

Für die Elemente dieser Liste braucht man nun nicht Teile eines Feldes zu nehmen, etwa wie im Beispiel 3, sondern man kann sie als lokale Variable innerhalb der Prozedur vereinbaren. Da sich bei PEARL90 in allen Prozeduren, die auf Modulebene vereinbart worden sind, beliebig viele Tasks gemeinsam aufhalten können (die Prozeduren sind reentrant-fähig), werden die lokalen Variablen einer solchen Prozedur nämlich so oft im Speicher des Rechners (im Stack der aufrufenden Tasks) angelegt, wie sich Tasks in der Prozedur befinden, weshalb man sie selbstverständlich auch verketten darf. Der Anker einer solchen verketteten Liste muß natürlich auf Modullevel vereinbart werden, und der Zugriff mehrerer Tasks auf die Liste (z. B. beim Einketten oder Ausketten einer Meldung) muß mit Semaphoren oder Bolts koordiniert werden.

5 Objektorientierte Programmierung

5.1 Objektorientierte Modulschnittstellen

Zeiger auf Prozeduren können wir auch dazu benutzen, die Exportschnittstellen von Moduln einfacher darzustellen. Beispiel 20 zeigt einen Modul, der in einem Multitasking-Programm eine Variable verwaltet. Da mit mehreren Tasks eventuell gleichzeitig auf die Variable `druck` zugegriffen werden kann, müssen die Zugriffe durch Bolt-Operationen koordiniert werden. Deshalb ist die Variable in dem Modul versteckt worden, und der Zugriff kann nur über globale Prozeduren erfolgen.

In Beispiel 20 besteht die Exportschnittstelle des Moduls aus den zwei Prozeduren `setzen` und `lesen`, deren Deklaration über den Modul verstreut ist; wenn der Modul länger wäre, könnte es Schwierigkeiten machen, bei Durchlesen diese Attribute GLOBAL überhaupt zu finden. In Beispiel 21 ist deshalb die Exportschnittstelle des Moduls mit Hilfe von Zeigern auf diese beiden Prozeduren anders definiert worden. Sie steht jetzt ganz am Ende und ist leicht auffindbar, und ein Modul, der eine der Prozeduren benutzen will, braucht nur `SPC Druck INV STRUCT[setzen REF PROC(FLOAT), lesen REF PROC RETURNS(FLOAT)] GLOBAL;` zu enthalten, um z. B. den Druck durch den Aufruf `Druck.setzen(5.2);` auf diesen Wert zu setzen.

```
MODULE (Druck_als_Objekt); PROBLEM;
 DCL druck FLOAT; ! aendern mit z.B. Druck.setzen(5.);
 DCL zugriff BOLT;
 setzen:PROC (neuerdruck INV FLOAT) GLOBAL;
    RESERVE zugriff;
     druck:=neuerdruck;
    FREE zugriff;
 END;
 lesen:PROC RETURNS(FLOAT) GLOBAL;
    DCL lokalerdruck FLOAT;
    ENTER zugriff;
     lokalerdruck:=druck;
    LEAVE zugriff;
    RETURN(lokalerdruck);
 END;
MODEND;
```

Beispiel 20: Modul zur Verwaltung einer Variablen

```
MODULE (Druck_als_Objekt); PROBLEM;
 DCL druck FLOAT; ! aendern mit z.B. Druck.setzen(5.);
 DCL zugriff BOLT;
 setzen:PROC (neuerdruck INV FLOAT);
    RESERVE zugriff;
     druck:=neuerdruck;
    FREE zugriff;
 END;
 lesen:PROC RETURNS(FLOAT);
    DCL lokalerdruck FLOAT;
    ENTER zugriff;
     lokalerdruck:=druck;
    LEAVE zugriff;
    RETURN(lokalerdruck);
 END;
 DCL Druck INV STRUCT[setzen REF PROC(FLOAT)
                  lesen REF PROC RETURNS(FLOAT)] GLOBAL
                      INIT(setzen,lesen);
MODEND;
```

Beispiel 21: Modul aus Beispiel 20 mit objektorientierter Schnittstelle

5.2 Klassen, Methoden und Objekte

Auch in der Prozeßdatenverarbeitung bietet ein Programmierstil große Vorteile, bei dem die Daten und die Prozeduren, die mit ihnen arbeiten, zu Objekten zusammengefaßt werden. Wir

haben eben ein Beispiel kennengelernt, in dem ein ganzer Modul als Objekt aufgefaßt worden ist. Nun ist dieser Modul relativ klein; wenn man es in einer Anwendung mit vielen Drücken zu tun hätte, wären so viele Moduln wie Drücke erforderlich, und jeder dieser Moduln müßte je eine Prozedur `setzen` und `lesen` enthalten.

Alle diese Druck-Moduln würden gleich aufgebaut sein und aus einer modul-internen Variablen, einem Bolt und zwei Prozeduren `setzen` und `lesen` bestehen. In der Umgangssprache würden wir sagen, daß es sich um Objekte gleichen Typs handelt. Da der Begriff Typ jedoch in PEARL90 schon mit der Bedeutung des Datentyps besetzt ist, wollen wir besser sagen, daß diese Modul-Objekte der selben Klasse angehören. Die Prozeduren `setzen` und `lesen` sind in diesem Konzept Tätigkeiten, die ein derartiges Objekt für uns ausführen kann; in der Sprechweise der objektorientierten Programmierung sind sie Methoden, und ihr Aufruf ist eine Botschaft an das jeweilige Objekt.

Die Zeiger in PEARL90 bieten mit ihrer Eigenschaft, auf alles außer anderen Zeigern zeigen zu können, die Möglichkeit, unsere Drücke anders zu realisieren als durch viele einzelne Moduln. Wir brauchen nur wie in Beispiel 22 einen neuen Typ zu vereinbaren, der eine Druckvariable und Zeiger auf die übrigen Bestandteile eines Druck-Objektes enthält. Selbstverständlich müssen die Prozeduren, auf die die Zeiger `setzen` und `lesen` zeigen, vorhanden sein. Sie sind ebenfalls in Beispiel 22 enthalten.

```
TYPE DRUCK STRUCT[
          druck FLOAT,
          zugriff INV REF BOLT,
          setzen INV REF PROC(INV REF DRUCK,neuerdruck INV FLOAT),
          lesen INV REF PROC(INV REF DRUCK)RETURNS(FLOAT)];

DRUCK_setzen: PROC(unserdruck DRUCK,neuerdruck FLOAT);
   RESERVE unserdruck.zugriff;
    unserdruck.druck:=neuerdruck;
   FREE unserdruck.zugriff;
END;

DRUCK_lesen:PROC(unserdruck REF DRUCK) RETURNS(FLOAT);
   DCL lokalerdruck FLOAT;
   ENTER unserdruck.zugriff;
    lokalerdruck:=unserdruck.druck;
   LEAVE unserdruck.zugriff;
   RETURN(lokalerdruck);
END;
```

Beispiel 22: Vereinbarung der Klasse DRUCK

Nach diesen Vereinbarungen können wir jetzt in Beispiel 23 ein Exemplar vom Datentyp `DRUCK` vereinbaren. Dabei werden die Variablen `ersterdruck` und `zweiterdruck` sofort initialisiert. Dazu müssen die Bolts `ersterdruck_bolt` und `zweiterdruck_bolt` vorher deklariert worden sein.

```
DCL ersterdruck_bolt BOLT,
    ersterdruck DRUCK INIT(
       0,
       ersterdruck_bolt,
       DRUCK_setzen,
       DRUCK_lesen);


DCL zweiterdruck_bolt BOLT,
    zweiterdruck DRUCK INIT(
       0,
       zweiterdruck_bolt,
       DRUCK_setzen,
       DRUCK_lesen);
```

Beispiel 23: Deklaration von Objekten der Klasse DRUCK

Die Prozeduren `DRUCK_setzen` und `DRUCK_lesen` haben einen Zeiger auf eine Variable vom Datentyp `DRUCK` als ersten formalen Parameter, damit sie auf eine entsprechend deklarierte Variable zugreifen können. Derartige Deklarationen zeigt Beispiel 23. Die in ihm enthaltenen Deklarationen zweier Objekte unterscheiden sich nur in `ersterdruck` und `zweiterdruck`; weitere Objekte der Klasse `DRUCK` würden sich anhand dieser Muster leicht erzeugen lassen.

Die Anweisung zum Setzen des ersten Druckes auf den Wert 5.0 lautet dann offensichtlich:

```
ersterdruck.setzen(ersterdruck,5.0);
```

5.3 Beispiel Verkehrsampel

In der Praxis würde man viele Drücke sicher nicht einzeln im Programm deklarieren, sondern sie in einem Feld oder sogar als Bestandteil einer Datenbank zusammenfassen. Ein Beispiel, das der Praxis schon eher entspricht, ist die Steuerung des Verkehrs an Ampelkreuzungen.

In einem vereinfachten Ansatz besteht jede Kreuzung aus zwei Straßen und der Fläche, auf der sich die Straßen kreuzen. Jede Straße hat zwei Ampeln, jede Ampel besteht aus der Hardware und einem Bild, falls man sie auf dem Bildschirm darstellen will. Wir benötigen also die Klassen `KREUZUNG`, `FLAECHE`, `STRASSE`, `AMPEL`, `AMPELHARDWARE` und `AMPELBILD`.

Die Ampeln einer Straße sollen in einer bestimmten zeitlichen Abfolge der farbigen Lichtsignale gesteuert werden. Dazu brauchen wir für jede Straße eine Ampelsteuerung. Wenn die Ampeln außerdem in verkehrsarmen Zeiten gemeinsam gelb blinken sollen, müssen sie auch dafür eine Steuerung vom Typ Blinker besitzen.

Zum Stellen der Ampeln würde man normalerweise eine Prozedur benutzen, der der Name der betreffenden Ampel übergeben würde und ein Bitmuster, das anzeigt, welche der drei farbigen Leuchten eingeschaltet werden soll. Beim objektorientierten Ansatz bedeutet das, daß die Klasse AMPEL eine Methode `stellen` enthalten muß. Zur Initialisierung der Hardware (Eröffnung der Prozeßdatenstationen) muß es außerdem eine Methode geben, die wir `init` nennen wollen. Auch die anderen Klassen werden in der Regel eine derartige Methode enthalten müssen.

Beispiel 24 zeigt die Typvereinbarung für die Klasse `STRASSE`. Jede Straße enthält zunächst eine Strukturkomponente `ampelstatus`, in der notiert wird, welches Farbmuster die Ampeln zeigen sollen. Weitere Komponenten zeigen, ob gerade ein Schaltvorgang, z. B. Rot–>Rot-Gelb–>Grün läuft. Weitere Komponenten legen fest, wie lange die Ampelphasen dauern sollen.

```
TYPE STRASSE STRUCT[
     ampelstatus BIT(3),
     (schaltstatus,
     blinkstatus,
     steuerungsstatus) BIT(1),
     (gelbzeit,rotgelbzeit,blinkzeit,
                     rotueberlapp,gruenzeit,maxgruenzeit) DUR,
     (hin,her) INV REF AMPEL,
     flaeche REF FLAECHE,
     induktschleife INV REF EREIGNIS,
     init INV REF PROC(INV REF STRASSE, INV REF FLAECHE),
     (rotschalten,
     gruenschalten,
     gelbschalten,
     ausschalten) INV REF PROC(INV REF STRASSE),
     blinker INV REF BLINKER,
     steuerung INV REF STEUERUNG];
```

Beispiel 24: Typvereinbarung für die Klasse `STRASSE`

Jede Straße enthält außerdem zwei Ampeln, für jede Richtung `hin` und `her` eine, die Kreuzungsfläche, zwei Induktionsschleifen, die zusammengeschaltet sind und ein Ereignis auslösen.

Die Methoden der Klasse `STRASSE` sind `rotschalten`, `gruenschalten`, `gelbschalten` und `ausschalten`. Sie dauern nur ein paar Sekunden; Beispiel 25 zeigt die PEARL-Prozedur für die Methode `rotschalten`.

```
STRASSE_rotschalten:PROC(strasse INV REF STRASSE);
  strasse.schaltstatus:='1'B;
  strasse.gelbschalten(strasse);
  AFTER strasse.gelbzeit RESUME;
  strasse.ampelstatus:=ROTMUSTER;
  strasse.hin.stellen(strasse.hin,strasse.ampelstatus);
  strasse.her.stellen(strasse.her,strasse.ampelstatus);
  strasse.schaltstatus:='0'B;
END;
```

Beispiel 25: Methode `rotschalten` der Klasse `STRASSE`

Die Steuerung der aufeinander folgenden Ampelphasen wird durch ein Objekt der Klasse `STEUERUNG` bewirkt, das zu jeder Straße gehört; ein Objekt der Klasse `BLINKER` bewirkt, daß die Ampeln der Straße gelb blinken.

Dieses Blinken muß sich einschalten und ausschalten lassen. Deshalb besitzt die Klasse `BLINKER` (Beispiel 26) unter anderem die Methoden `start` und `stop`. Weil beim Ausschalten gewartet werden muß, bis die Ampeln nicht mehr Gelb zeigen, gibt es die Methode `beendet`. Damit ein Blinker weiß, für welche Straße er arbeitet, wird ihm deren Namen als Argument übergeben, wenn die betreffende Straße ihren Blinker initialisiert.

```
TYPE BLINKER STRUCT[
     init INV REF PROC(INV REF BLINKER,INV REF STRASSE),
     (start,
     stop,
     beendet) INV REF PROC(INV REF BLINKER),
     strasse REF STRASSE,
     zyklus INV REF ZYKLUS,
     methode REF PROC(INV REF BLINKER,INV REF BIT(1)),
     status REF BIT(1)];
```

Beispiel 26: Klasse `BLINKER` als Bestandteil von `STRASSE`

Bei normaler Programmierung würde man bei PEARL90 für das Blinken eine Task programmieren, die in einer REPEAT-Schleife die Ampeln einer Straße eine vorgegebene Zeit auf Gelb und dann eine Zeitlang auf Aus schaltet. Das muß man selbstverständlich auch bei einem objektorientierten

Ansatz. Nur wird hier das Blinken mit einem Objekt der Klasse ZYKLUS (Beipiel 27) bewirkt. Die zu dieser Klasse gehörenden Methoden werden wieder als PEARL-Prozeduren geschrieben (Beispiel 28).

```
TYPE ZYKLUS STRUCT[
            init INV REF PROC(INV REF ZYKLUS,
                              REF STRUCT[],
                              REF PROC(INV REF STRUCT[],INV REF BIT(1)),
                              REF BIT(1)),
            start INV REF PROC(INV REF ZYKLUS),
            stop INV REF PROC(INV REF ZYKLUS),
            beendet INV REF PROC(INV REF ZYKLUS),
            auftraggeber REF STRUCT[],
            auftrag REF PROC(INV REF STRUCT[],INV REF BIT(1)),
            auftragsteuerbit REF BIT(1),
            task INV REF TASK,
            endesema INV REF SEMA];
```

Beispiel 27: Typvereinbarung für die Klasse ZYKLUS

```
ZYKLUS_init:PROC(zyklus INV REF ZYKLUS,
                 auftraggeber INV REF STRUCT[],
                 auftrag INV REF PROC(INV REF INV STRUCT[],INV REF BIT(1)),
                 auftragsteuerbit INV REF BIT(1));
   zyklus.auftraggeber:=auftraggeber;
   zyklus.auftrag:=auftrag;
   zyklus.auftragsteuerbit:=auftragsteuerbit;
   WHILE TRY zyklus.endesema REPEAT END;      ! Fuer Wiederanlauf
END;

ZYKLUS_start : PROC(zyklus INV REF ZYKLUS );
  CONT zyklus.auftragsteuerbit:='1'B;
  ACTIVATE zyklus.task;
END;

ZYKLUS_stop : PROC(zyklus INV REF ZYKLUS);
  CONT zyklus.auftragsteuerbit:='0'B;
END;

ZYKLUS_beendet : PROC(zyklus INV REF ZYKLUS);
  REQUEST zyklus.endesema;
END;
```

Beispiel 28: Die Methoden der Klasse ZYKLUS

Jedes Objekt der Klasse `ZYKLUS` enthält eine Task und einen Semaphor, der zum Signalisieren dient, daß sich die Task beendet hat (für die Methode `beendet`). Bei der Methode `start` werden ein Steuerbit gesetzt und die Task aktiviert. Diese Task führt einen Auftrag aus, die dem Objekt der Klasse `ZYKLUS` zusammen mit dem Auftraggeber und dem Zeiger auf ein Steuerbit als Argumente der Methode `init` übergeben werden (Beispiel 28).

```
BLINKER_init:PROC(blinker INV REF BLINKER,
                  strasse INV REF ADAPTSTRASSE);
      blinker.strasse:=strasse;
      blinker.status:=strasse.blinkstatus;
      blinker.zyklus.init(blinker.zyklus,
                          blinker,
                          blinker.methode,
                          blinker.status);
END;
```

Beispiel 29: Methode `init` der Klasse `BLINKER`

Im Falle des Blinkens fungiert ein Objekt der Klasse `BLINKER` als Auftraggeber, das Blinken ist der Auftrag. Deshalb muß die Methode `init` des Blinkers so lauten wie in Beispiel 29: zunächst wird ihr der Zeiger auf die Straße übergeben, zu der der Blinker gehört. Dann wird dafür gesorgt, daß `blinker.status` auf das Statusbit `strasse.blinkstatus` zeigt; schließlich werden der Auftraggeber `blinker`, der Auftrag `blinker.methode` und der Zeiger auf das Steuerbit `blinker.status` als Argumente der Methode `zyklus.init` an das Objekt `blinker.zyklus` der Klasse `ZYKLUS` übergeben.

```
DCL kreuzung_westost_blinker BLINKER INIT(
     BLINKER_init,
     BLINKER_start,
     BLINKER_stop,
     BLINKER_beendet,
     NIL,           ! Strasse, uebergeben bei BLINKER_init
     kreuzung_westost_blinker_zyklus,
     BLINKER_methode,
     NIL);
```

Beispiel 30: Deklaration eines Objektes der Klasse `BLINKER`

Die Komponente `zyklus.auftragsteuerbit` enthält durch dieses Weiterreichen eines Parameters letzendlich den Zeiger auf das Bit `blinkstatus` der Straße, deren Teil der jeweilige Blinker

ist. Es ist gesetzt, solange die Ampeln der Straße blinken, und wird durch die Methode `stop` des Zyklus-Objektes zurückgesetzt, damit das Blinken beendet wird.

Beispiel 30 und Beispiel 31 enthalten die Objektvereinbarung eines Blinkers und des zugehörigen Zyklus-Objektes.

```
kreuzung_westost_blinker_zyklus_task:TASK;
  kreuzung_westost_blinker_zykluslus.auftrag
           (kreuzung_westost_blinker_zyklus.auftraggeber,
           kreuzung_westost_blinker_zyklus.auftragsteuerbit);
  RELEASE kreuzung_westost_blinker_zyklus.endesema;
END;

DCL  kreuzung_westost_blinker_zyklus_endesema SEMA;

DCL kreuzung_westost_blinker_zyklus ZYKLUS INIT(
    ZYKLUS_init,
    ZYKLUS_start,
    ZYKLUS_stop,
    ZYKLUS_beendet,
    NIL,            ! Auftraggeber, uebergeben vom Blinker bei ZYKLUS_init
    NIL,            ! Auftrag, uebergeben vom Blinker
    NIL,            ! Steuerbit, uebergeben vom Blinker
    kreuzung_westost_blinker_zyklus_task,
    kreuzung_westost_blinker_zyklus_endesema);
```

Beispiel 31: Objekt der Klasse `ZYKLUS` für den Blinker aus Beispiel 30

Der Auftrag für das Zyklus-Objekt aus Beispiel 31 ist die Methode `BLINKER_methode` aus Beispiel 32. Sie wird dem Zyklus-Objekt als Argument übergeben (Beispiel 28), wenn ein Blinker den ihm gehörenden Zyklus mit der Methode `ZYKLUS_init` initialisiert.

```
BLINKER_methode:PROC(blinker INV REF BLINKER,steuerbit INV REF BIT(1));
   WHILE CONT steuerbit REPEAT
     AFTER blinker.strasse.blinkzeit RESUME;
     blinker.strasse.gelbschalten(blinker.strasse);
     AFTER blinker.strasse.blinkzeit RESUME;
     blinker.strasse.ausschalten(blinker.strasse);
   END;
END;
```

Beispiel 32: Blinken als Prozedur mit Blinkschleife

Objekte der Klasse `ZYKLUS` sind polymorph; sie können für alle Klassen arbeiten, die bitgesteuerte Schleifen als nebenläufige Methoden ausführen lassen wollen.

5.4 Rezept für objektorientierte Programmierung in PEARL90

Die bisher gegebenen Beispiele zeigen, wie man bei der objektorientierten Programmierung vorgehen sollte:

1. Festlegung einer Objekt-Klasse als PEARL90-Typvereinbarung einer Struktur. Diese sollte aus folgenden Teilen bestehen:
   - Dem Klassenbezeichner (Typbezeichner) in Großbuchstaben,
   - den Variablen des Objektes,
   - Zeigern auf die Methoden des Objektes (PEARL-Prozeduren, deren erster Parameter ein Zeiger auf den Objekt-Typ ist),
   - Zeigern auf die Objekte, die im Objekt enthalten sind.
2. Schreiben der Methoden als PEARL-Prozeduren nach folgenden Regeln:
   - Der Prozedurbezeichner besteht aus dem Klassenbezeichner,angehängtem Unterstrich und angehängtem Methodenbezeichner (Kleinbuchstaben),
   - Der erste Parameter der Prozedur besteht aus einem Zeiger auf den (Klassen-)Typ,
   - Weitere Parameter (Argumente der Methode) können folgen,
   - Darauf folgt eine ganz normale PEARL-Prozedur, die die Komponenten des ersten Prozedurparameters und die Argumente benutzt.
3. Schreiben des Musters für die Deklaration eines Objektes. Diese sollte bestehen aus
   - Vereinbarungen von anderen Objekten, Tasks, Semaphoren oder Bolts, die im Objekt enthalten sind,
   - der Deklaration eines Objektes des Objekttyps mit einer INIT-Liste,
   - in der die Variablen des Objektes Anfangswerte bekommen,
   - in die die Bezeichner der Methoden-Prozeduren an richtiger Stelle eingetragen sind,
   - in die Bezeichner von Objekten, Tasks usw. eingetragen sind, die im jetzt deklarierten Objekt enthalten sein sollen.

Diese Rezepte, um auch in PEARL90 objektorientiert zu programmieren, sind eigentlich recht einfach. Deshalb läge es nahe, eine Spracherweiterung für einen Vorübersetzer vorzusehen, in der Klassen- und Objektvereinbarungen enthalten sind; der Vorübersetzer könnte dann daraus die notwendigen PEARL-Vereinbarungen nach obigen Rezepten automatisch erzeugen.

Inhaltsverzeichnis