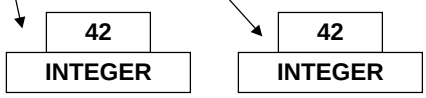
Datentypen II

- **Dynamische Datentypen**
  - Zeigertypen
  - dynamische Datenstrukturen
- **Anwendungsbeispiele**
  - lineare Liste
  - sortierte Liste
- **Prozedurtyp**

Wertsemantik

- **Die bisher betrachteten Variablen und die Zuweisung basieren auf der *Wertsemantik*:**
  - Eine Variable hat einen ***definierten*** oder ***undefinierten*** Wert.
  - Bei der ***Zuweisung*** wird der Wert des Ausdrucks der rechten Seite (rhv) an die Variable der linken Seite (lhv) zugewiesen.
  - Eine ***Identität*** von Objekten wird nicht hergestellt.
  -
```
VAR Antwort1, Antwort2: INTEGER;
…
Antwort1 := 42; (* 1 *)
Antwort2 := Antwort1; (* 2 *)
Antwort1 := 24; (* 3 *)

Antwort1 := Antwort2;
```

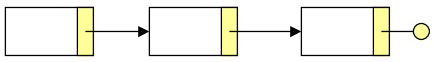


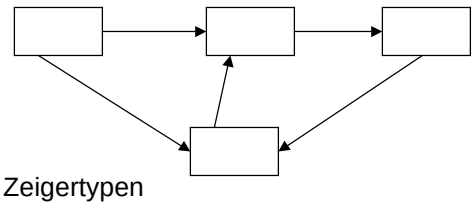
Statische Datentypen

- **Kennzeichnend für imperative Sprachen ist, daß *Wertsemantik* und *statische Datentypen* zusammenfallen:**
  - Die Zuordnung von Bezeichner und Wert geschieht über die Zuweisung.
  - Die Variablen eines statischen Typs ***behalten ihre Struktur*** während ihrer Lebensdauer bei.
  - Der ***Speicher*** einer statisch getypten Variablen wird bei der Übersetzung anhand der Deklaration ***festgelegt*** und bei der "ersten Verwendung implizit angelegt ".
- **Speicherbereiche für statisch getypte Variablen werden im sog. *Kellerspeicher* reserviert**
  - zusammenhängender Bereich pro Objekt
  - Bereich wird ***angelegt***, beim Eintritt in entsprechende Prozedur
  - Bereich wird ***freigegeben***, beim Verlassen der Prozedur

Statische Datentypen und Listen

- **Probleme mit statischen Datentypen**
  - Es werden Datenstrukturen benötigt, deren ***Größen Schwankungen*** unterworfen sind.
  - Es sollen ***komplizierte*** Datenstrukturen gebildet werden
    - Listen
    - Bäume
    - Graphen





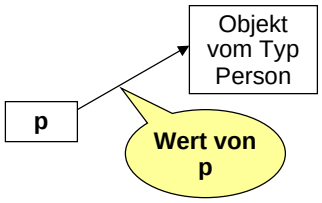
- **Lösung**
  - Dynamische Datentypen
  - oder dynamische Variable und Zeigertypen

Zeigertyp - Referenztyp

Zeigertyp

- um einen Zeigertyp zu deklarieren, bietet Modula-3 den ***Typkonstruktor***
- **`<ZeigerTyp> = REF <ReferenzierterTyp>`** an
- damit kann aus ***jedem Typ*** ein Zeigertyp abgeleitet werden

```
TYPE Person = RECORD
                anrede : Anrede;
                name   : Name;
                persnr : PersNr;
              END;

TYPE PersonRef = REF Person;

VAR p : PersonRef;
```



- p kann als Werte ***Zeiger auf Objekte*** vom Typ `Person` annehmen
- zeigt p auf kein Objekt, dann wird dies durch den Wert ***`NIL`*** angezeigt
- `NIL` ist Objekt jedes Zeigertyps

Eigenschaften dynamischer Datentypen

Eigenschaften von Objekten dynamischer Datentypen

- ***Lebensdauer*** ist nicht an die Ausführung einer Prozedur oder Moduls gebunden
- sie werden zur Laufzeit ***explizit (vom Programmierer) erzeugt*** und eventuell auch wieder beseitigt
- sie werden in einem speziell dafür vorgesehenen Speicherbereich angelegt (***Halde oder Heap***)
- können in prinzipiell ***beliebiger*** Menge geschaffen werden
- haben im Gegensatz zu den bisherigen Objekten ***keinen festen Bezeichner***
- sie werden stattdessen über einen ***Zeiger (Pointer)*** identifiziert
- Zeiger können im Keller oder auf der Halde liegen
- die referenzierten Objekte liegen ***immer*** auf der Halde

Dynamische Datentypen

Erzeugen von Haldenobjekten

- Beispiel:

```
PROCEDURE PROC …
  TYPE PersonRef = REF Person;
  VAR p : PersonRef;
BEGIN
  p := NIL;
  p := NEW (PersonRef);
END
```

- Beim Betreten der Prozedur wird Speicher für p zur Verfügung gestellt und beim Verlassen wieder freigegeben.
- Durch den Aufruf von `NEW(PersonRef)`
  - wird auf der Halde ***Speicher*** für ein Objekt von Typ Person angelegt
  - die ***Adresse*** dieses Speicherplatzes wird zurückgeliefert und der Variablen p zugewiesen
  - Das Objekt selbst erhält keinen eigenen Bezeichner – man spricht von einer ***dynamischen*** oder auch von einer ***anonymen*** Variablen.
- Dieser Speicher wird nicht freigegeben, wenn die Proz. verlassen wird

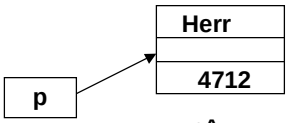
Dynamische Datentypen

Dereferenzierung - 1

- **Eine gesetzte Referenzvariable verweist auf ein Objekt.**
- **Um das Objekt selbst zu erhalten, müssen wir dem Verweis "nachgehen".**
  - Diese Operation, bei der auf das referenzierte Objekt einer Referenzvariablen zugegriffen wird, heißt ***dereferenzieren***.
- **Dereferenzieren ist, neben dem Erzeugen der referenzierten Objekte, die**
  - zweite charakteristische Operation von Referenztypen.
  - Nur eine ***gesetzte Referenzvariable*** kann dereferenziert werden.
  - Der Versuch einer Dereferenzierung der Referenz `NIL` führt in den meisten Programmiersprachen zum ***Programmabbruch***.

Dereferenzierung - 2

- **Dereferenzierung**
  - Zugriff auf ***Werte*** von Zeigerobjekten




```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

p^.persnr := 4732;

p := NEW (PersonRef);
p^.persnr := 4712;
p^.anrede := AnredeTyp.Herr;

nr := p^.persnr;
```

Laufzeitfehler: Zeigervariable nicht gesetzt

Dereferenzierungsoperator

  - In Modula-3 kann der ^-Operator entfallen, wenn ein weiterer Operator folgt (z.B."[ ]", ".")
  - Das trägt ***nicht zur Klarheit*** bei !!!
  - Darum: Verwenden Sie ***immer*** den ^-Operator !!!

Operationen auf Referenztypen

- **Zulässige Werte von Referenztypen**
  - sind Referenzen oder der Wert "***keine Referenz***" (`NIL`).
- **Gesetzte Referenzen haben keine externe Repräsentation.**
  - Entsprechend können sie ***nicht*** ausgegeben oder z.B. in ***Rechenoperationen*** verwendet werden.
- **Die einzigen zulässigen Operationen auf Referenzen sind:**
  - Test auf ***Gleichheit*** oder ***Ungleichheit***,
  - ***Zuweisung*** auf Variablen von kompatiblem Typ.
- **Beispiel in Modula-3:**

```
VAR p1, p2 : PersonRef;
…
IF p1 = NIL THEN
  p1 := NEW (PersonRef)
END;
p2 := p1;
```

Dynamische Datentypen

Zuweisung

```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;
p := NEW(PersonRef);
p^.name.nachname := "Maier";
q := NEW(PersonRef);
q^.name.nachname := "Mueller";

p := q;      (1)

p^ := q^;    (2)
```

- Der Effekt der Anweisung (1) ist streng zu unterscheiden von der Wirkung der Anweisung (2)

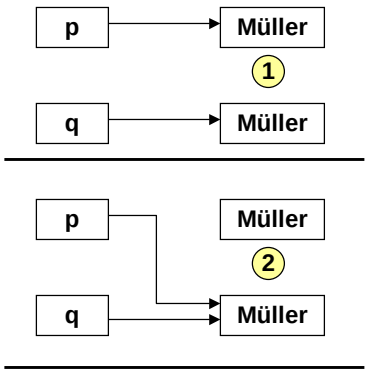
  (1) ist eine ***Referenzzuweisung***

  (2) ist eine ***Wertzuweisung***

Dynamische Datentypen

Vergleich

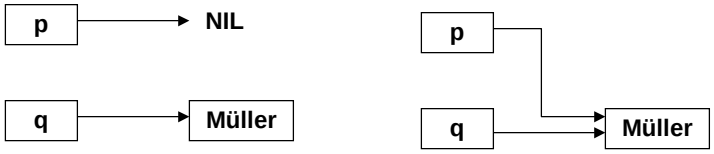
```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;

... p = q ...        (1)

... p^ = q^ ...



... p = q ...        (2)

... p^ = q^ ...
```



- Zwei Zeiger sind gleich, wenn sie auf ***dasselbe*** referenzierte Objekt (oder: auf die gleiche Adresse) zeigen!

Das Alias-Problem

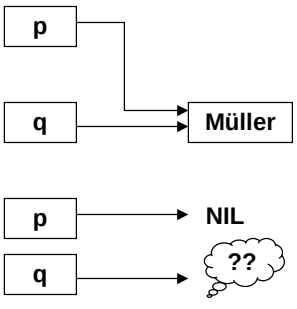
- **Die Zuweisung bei Referenztypen basiert auf der sog. *Referenzsemantik*:**
  - Der ***Verweis*** auf ein Objekt wird zugewiesen; nicht etwa der Wert des referenzierten Objekts.
  - Dies ist vielfach erwünscht, schafft aber folgendes Problem
- **Mehrere Referenzvariablen können auf dasselbe Objekt verweisen.**
  - Damit ist lokal oft nicht ***entscheidbar***, ob sich Veränderungen am Zustand eines referenzierten Objekts ergeben haben oder nicht.
  - Dies ist das ***Alias-Problem***, das bei allen Referenztypen auftritt.



Freigeben von Zeigerobjekten

- **In der Regel müssen Zeigerobjekte vom Programmierer kontrolliert werden:**
  - Er muß sie explizit ***anlegen*** und ***freigeben***
- **Beim Freigeben können folgende Fehlersituationen auftreten**
  - Nicht mehr benötigte Zeigerobjekte wurden ***nicht frei*** gegeben
    - Es wird Speicher verschwendet
  - Es werden Zeigerobjekte freigegeben, die noch ***benötigt*** werden
    - Problem der "***dangling references***"

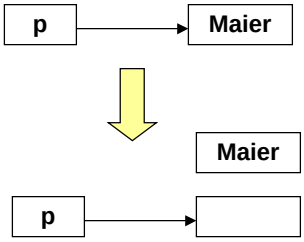
```
TYPE PersonRef = REF Person;
VAR p, q : PersonRef;
...
p := q;
DISPOSE(p);
q^.anrede := ...
```



Probleme mit Referenzvariablen

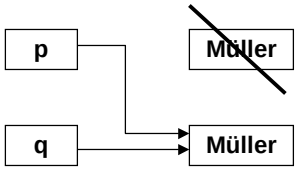
- **"Lost Object":**
  - Es kann dynamisch angelegte Objekte geben, auf die keine ***Referenzvariable*** mehr verweist.
  - Dieses Objekt kann nicht mehr erreicht werden und wird als ***Garbage*** bezeichnet.
- **Häufige Fehlersituation:**

```
TYPE PersonRef = REF Person;
VAR p : PersonRef;

p := NEW (PersonRef);
p^.name := "Maier";
...
p := NEW (PersonRef);
```



Automatische Speicherbereinigung

- **Einige Laufzeitumgebungen von Sprachen können Zeigerobjekte kontrollieren**
  - sie geben Zeigerobjekte, die ***nicht mehr erreicht*** werden können, automatisch frei
  - "***Garbage Collector***" (Müllsammler)

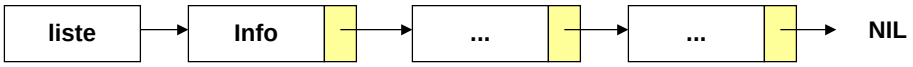


- **Diskussion Garbage Collector**
  - ***Vorteile*** (wenn keine explizite Freigabe vorhanden oder genutzt)
    - Fehlerklasse "dangling reference" ausgeschaltet
  - ***Nachteile***
    - Müllsammeln kostet Zeit

Anwendungs-beispiele

Dynamische Datenstrukturen

- **Ziel:**
  - Definition dynamischer Datenstrukturen, die ***einfach*** vom Programmierer erstellt, verwaltet und kontrolliert werden können
- **Referenzvariablen**
  - können auf zusammengesetzte Datenstrukturen verweisen, die selbst wieder ***Referenzobjekte*** enthalten. Wenn diese Strukturen rekursiv sind, sprechen wir von ***Verweisketten***, die den Aufbau dynamischer Datenstrukturen ermöglichen.
- **Beispiel: Einfach verkettete lineare Liste**
  - Elemente der Liste sind ***Zeigerobjekte***
  - Jedes Listenobjekt ist so konstruiert, das es ***selbst*** wieder auf ein Listenobjekt ***verweisen*** kann
  - zusätzlich enthält es die ***eigentlichen*** Informationen



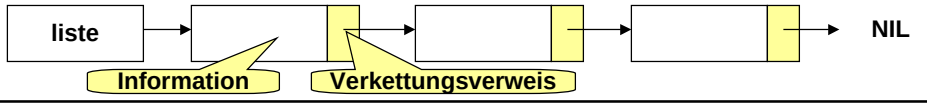
Anwendungs-beispiele

Lineare Liste

```
TYPE Jahr = [1890 .. 1998];

TYPE Person = RECORD
  name, vorname : TEXT;
  geburtsjahr : Jahr;
END;

TYPE ListenElement =
  RECORD
    person : Person;
    nachfolger : RListenElement;
  END;
TYPE RListenElement = REF ListenElement;
VAR liste : RListenElement;
```

Typ, für die in der Liste verwaltete Information

"Anker", um auf die Liste zugreifen zu können



Anwendungs-beispiele

Suchen in linearen Listen - 1

```
PROCEDURE Suche (liste : RListenElement;
                 was : Person): RListenElement =
VAR position : RListenElement;
    gefunden : BOOLEAN := FALSE;

BEGIN
  position := liste;
  WHILE (position # NIL AND NOT gefunden) DO
    IF was = position^.person THEN
      gefunden := TRUE;
    ELSE
      position := position^.nachfolger;
    END;
  END;
  RETURN position;
END Suche;
```

Sequentielles Suchen

Ergebnis : Zeiger auf das gefundene Element sonst NIL

Anwendungs-beispiele

Suchen in linearen Listen - 2

- **Listen sind rekursive Datenstrukturen**
  - ***Rekursive Datenstrukturen*** können *elegant* mit ***rekursiven Prozeduren*** bearbeitet werden!

```
PROCEDURE SucheRek (liste : RListenElement;
                    was : Person): RListenElement =
BEGIN
  IF liste # NIL THEN
    IF was = liste^.person THEN
       RETURN liste;
    ELSE
      RETURN SucheRek (liste^.nachfolger, was);
    END;
  ELSE
    RETURN NIL;
  END;
END SucheRek;
```

rekursiver Aufruf

Anwendungs-beispiele

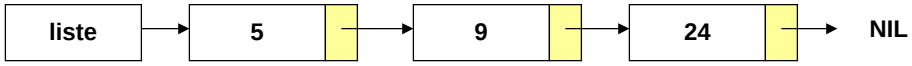
Sortierte lineare Liste

```
TYPE ListenElement =
  RECORD
    wert : INTEGER;
    nachfolger : RListenElement;
  END;
TYPE RListenElement = REF ListenElement;
VAR liste : ListenElement;
```

Für alle Elemente x der Liste gilt:

`x^.wert <= x^.nachfolger^.wert`

Liste soll immer im Zustand "aufsteigend" sortiert sein

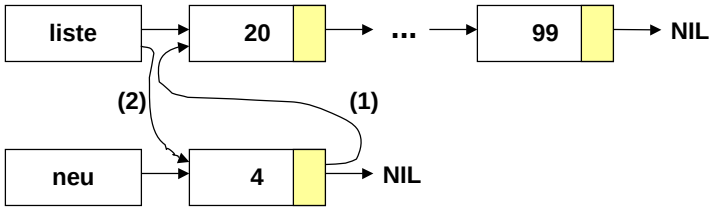


Anwendungs-beispiele

Einfügen in eine sortierte Liste - 1

- **Fall 1: Die Liste ist leer.**

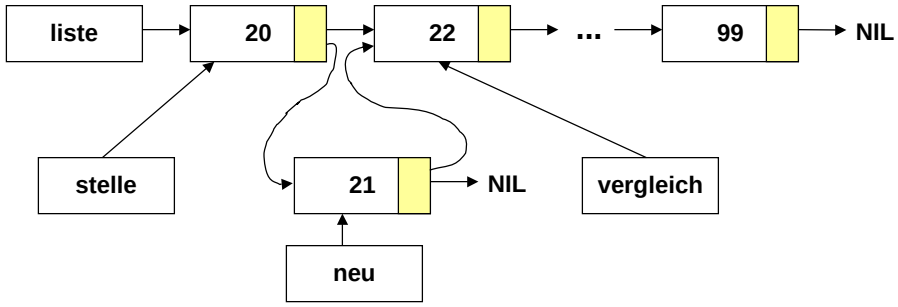
- **Fall 2: Element muß vorne eingefügt werden**



Anwendungs-beispiele

Einfügen in eine sortierte Liste - 2

- **Fall 3: Element muß sonst irgendwo eingefügt werden.**
  - Es muß nach der ***Einfügestelle*** gesucht werden
  - Damit man auf die Elemente vor und hinter der Einfügestelle zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das Element hinten angefügt werden muß !!

Anwendungs-beispiele

Einfügen: Lösung A -1

```
PROCEDURE Einfuegen (VAR liste : RListenElement; w : INTEGER) =
VAR neu, einfuegestelle : RListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                  (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN         (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);        (* neu vorne einhaengen *)
  ELSE
    einfuegestelle := SucheEinfuegestelle(liste, w);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```

Anwendungs-beispiele

Einfügen: Lösung A -2

```
PROCEDURE ErzeugeNeuesListenelement(w : INTEGER): RListenElement =
VAR elem : RListenElement;
BEGIN
  elem := NEW(RListenElement);
  elem^.wert := w;
  elem^.nachfolger := NIL;
  RETURN elem
END ErzeugeNeuesListenelement;
```

```
PROCEDURE EinfuegenVorne(VAR liste : RListenElement;
                         VAR neu   : RListenElement)=
BEGIN
  neu^.nachfolger := liste;
  liste := neu;
END EinfuegenVorne;
```


Anwendungs-beispiele

Einfügen: Lösung A -3

```
PROCEDURE SucheEinfuegestelle(liste : RListenElement;
                              w : INTEGER): RListenElement =
VAR stelle, vergleich: ListenElement;
BEGIN
  vergleich := liste;
  stelle := liste;
  WHILE (vergleich # NIL) AND (vergleich^.wert <= w) DO
    stelle := vergleich;
    vergleich := vergleich^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```

```
PROCEDURE FuegeAnEinfuegestelleEin(VAR neu   : RListenElement;
                                   VAR stelle: RListenElement)=
BEGIN
  neu^.nachfolger := stelle^.nachfolger;
  stelle^.nachfolger := neu;
END FuegeAnEinfuegestelleEin;
```

Anwendungs-beispiele

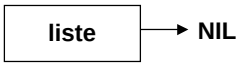
Einfügen: Lösung B

```
PROCEDURE Einfuegen (VAR liste : RListenElement; w : INTEGER) =
VAR neu, position, vorgaenger : RListenElement;
BEGIN
  neu := NEW(RListenElement);
  neu^.wert := w;
  neu^.nachfolger := NIL;

  IF liste = NIL THEN             (* liste ist leer *)
    liste := neu;
  ELSIF (w <= liste^.wert) THEN   (* w ist kleinstes Element*)
    neu^.nachfolger := liste;     (* w vorne einhaengen *)
    liste := neu;
  ELSE                            (* Einfuegestelle suchen *)
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert <= w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                          (* vorgaenger = Einfuegestelle *)
    neu^.nachfolger := position;  (* w einhaengen *)
    vorgaenger^.nachfolger := neu;
  END;
END Einfuegen;
```

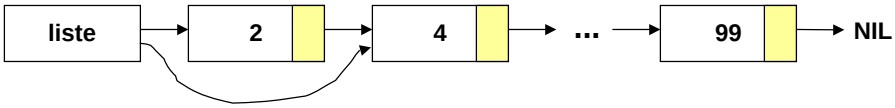

Anwendungs-beispiele

Löschen in einer sortierten Liste - 1

- **Fall 1: Die Liste ist leer.**



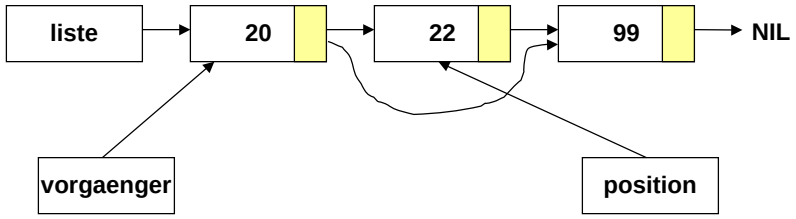
- **Fall 2: Das erste Element muß gelöscht werden**



Anwendungs-beispiele

Löschen in einer sortierten Liste - 2

- **Fall 3: Element muß sonst irgendwo gelöscht werden.**
  - Damit man auf die Elemente vor und hinter dem zu löschenden Element zugreifen kann, benötigt man zwei Hilfszeiger



Verhält sich auch korrekt, wenn das letzte Element gelöscht werden muß !!

Anwendungs-beispiele

Löschen: Lösung A

```
PROCEDURE Loeschen (VAR liste    : RListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : RListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    position := liste;
    vorgaenger := liste;
    WHILE (position # NIL) AND (position^.wert # w) DO
      vorgaenger := position;
      position := position^.nachfolger;
    END;                          (* position = NIL v position^.wert = w *)
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        liste := position^.nachfolger;
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```

Anwendungs-beispiele

Löschen: Lösung B -1

```
PROCEDURE Loeschen (VAR liste    : RListenElement;
                                w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : RListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position);
    gefunden := (position # NIL);  (* position= NIL v position^.wert = w *)
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```


Anwendungs-beispiele

Löschen: Lösung B -2

```
PROCEDURE Suche (liste : RListenElement;
                 w : INTEGER;
                 VAR vorgaenger, stelle : RListenElement) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE LoescheVorne(VAR liste : RListenElement) =
BEGIN
  IF liste # NIL THEN
    liste := liste^.nachfolger;
  END;
END LoescheVorne;
```

Anwendungs-beispiele

Beobachtung !

- **Einfügen und Loeschen benutzen ähnliche Such-Prozeduren**

```
PROCEDURE Suche (liste : RListenElement; w : INTEGER;
                 VAR vorgaenger, stelle : RListenElement) =
BEGIN
  stelle := liste; vorgaenger := liste;
  WHILE (stelle # NIL) AND (stelle^.wert # w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

```
PROCEDURE SucheEinfuegestelle(liste : RListenElement;
                                  w : INTEGER): RListenElement =
VAR stelle, vergleich: RListenElement;
BEGIN
  vergleich := liste; stelle := liste;
  WHILE (vergleich # NIL) AND (vergleich^.wert <= w) DO
    stelle := vergleich;
    vergleich := vergleich^.nachfolger;
  END;
  RETURN stelle;
END SucheEinfuegestelle;
```


Prozedurtyp

Prozedurtyp und Prozeduren dieses Typs

- **Frage:**
  - Gibt es eine Möglichkeit, Prozeduren so zu ***parametrisieren***, daß als Parameter ein Algorithmus (Prozedur) übergeben werden kann?
- **Prozedurtyp**
  - Ein Prozedurtyp definiert eine ***Signatur.***
  - Die Werte eines Prozedurtyps sind ***Prozeduren***, die der vorgegebenen Signatur entsprechen.
  - Entsprechend können Variablen als ***Prozedurvariablen*** deklariert werden.
  - Prozedurvariablen können ***passende*** Prozeduren zugewiesen werden.
  - Prozedurvariablen können als ***Parameter*** übergeben werden.
  - Gesetzte Prozedurvariablen können in Anweisungen mit ***aktuellen Parametern*** aufgerufen werden.

Prozedurtyp

Beispiel: ProzedurTyp - 1

```
TYPE  Bereich      = [-10 .. 10]; Index   = [1 .. 10];
      Zahlenmenge  = SET OF Bereich;
      Zahlenfeld   = ARRAY Index OF Bereich;
      TestProzedur = PROCEDURE (a : INTEGER) : BOOLEAN;

PROCEDURE IstPositiv (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i > 0);
END IstPositiv;

PROCEDURE IstNegativ (i : INTEGER) : BOOLEAN =
BEGIN
  RETURN (i < 0);
END IstNegativ;

PROCEDURE Filtern (        feld : FeldTyp;
                   VAR resultat : Zahlenmenge;  p : TestProzedur) =
BEGIN
  FOR i := FIRST(Index) TO LAST(Index) DO
    IF p(feld[i]) THEN
      resultat := resultat + Zahlenmenge{feld[i]};
    END;
  END;
END Filtern;
```

Gegeben ist ein Feld mit Zahlen. Bestimme die darin enthaltenen positiven und negativen Zahlen

Signatur-konforme Prozeduren

Prozedur-parameter

Prozedurtyp

Beispiel: ProzedurTyp - 2

```
VAR proc : TestProzedur;

    werte := Zahlenfeld{3, -5, 9, 8, -6, 9, -6, -1, -1, 5};
    positiv, negativ := Zahlenmenge{};

BEGIN
  proc := IstPositiv;
  Filtern (werte, positiv, proc);
  proc := IstNegativ;
  Filtern (werte, negativ, proc);

  SIO.PutLine ("Positive Zahlen:");
  FOR e := FIRST(Bereich) TO LAST(Bereich) DO
    IF e IN positiv THEN SIO.PutInt(e); SIO.PutText(", "); END;
  END;
  SIO.Nl();
```

Prozedurvariable

Zuweisung einer Prozedur an die Prozedurvariable

Zuweisung an Prozedurvariable

- **Zuweisung**
  - TYPE PType = PROCEDURE …
    VAR proc : Ptype

    proc := PT; (* Ausdruck der vom Typ PT ist *)
  - Diese Zuweisung ist korrekt, wenn
    - PT = ***NIL*** ist (NIL ist mit jedem Wert eines Prozedurtyps kompatibel)
    - ***Anzahl*** der Parameter und deren ***Typen*** sind gleich für PT und PType
    - Beide haben den gleichen ***Ergebnistyp*** oder keinen

```
TYPE TestProzedur = PROCEDURE (a : INTEGER)  : BOOLEAN;
     PT1          = PROCEDURE (in : INTEGER) : BOOLEAN;

VAR test : TestProzedur;
      p1 : PT1;
…
p1 := test;
test := p1;
```

Typen sind signatur-konform

Prozedurtyp

Verwendung Prozedurtyp - 1

```
TYPE Vergleichsoperation = PROCEDURE (a, b : INTEGER) : BOOLEAN;

PROCEDURE Suche (liste : ListenElement;
                     w : INTEGER;
                 VAR vorgaenger, stelle : ListenElement;
                    op : Vergleichsoperation) =
BEGIN
  stelle := liste;
  vorgaenger := liste;
  WHILE (stelle # NIL) AND  NOT op(stelle^.wert, w) DO
    vorgaenger := stelle;
    stelle := stelle^.nachfolger;
  END;
END Suche;
```

signaturkonform

```
PROCEDURE IstGleich
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a = b;
END IstGleich;
```

```
PROCEDURE Groesser
          (a,b: INTEGER): BOOLEAN =
BEGIN
  RETURN a > b;
END Groesser;
```

Prozedurtyp

Verwendung Prozedurtyp - 2

```
PROCEDURE Loeschen (VAR liste : ListenElement; w : INTEGER;
                    VAR gefunden : BOOLEAN) =
VAR position, vorgaenger : ListenElement;
BEGIN
  IF liste = NIL THEN gefunden := FALSE;
  ELSE
    Suche(liste, w, vorgaenger, position, IstGleich);
    gefunden := (position # NIL);
    IF gefunden THEN
      IF position = liste THEN (*gef. Element ist vorne *)
        LoescheVorne(liste);
      ELSE
        vorgaenger^.nachfolger := position^.nachfolger;
      END;
    END;
  END;
END Loeschen;
```


Prozedurtyp

Verwendung Prozedurtyp - 3

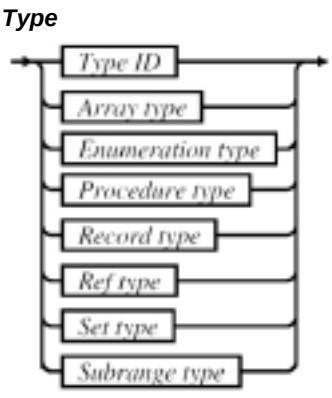
```
PROCEDURE Einfuegen (VAR liste : ListenElement; w : INTEGER) =
VAR neu, einfuegestelle, groessererWert : ListenElement;
BEGIN
  neu := ErzeugeNeuesListenelement(w);
  IF liste = NIL THEN                (* liste ist leer *)
    liste := neu;
  ELSIF (w < liste^.wert) THEN       (* w ist kleinstes Element*)
    EinfuegenVorne(liste, neu);      (* neu vorne einhaengen *)
  ELSE
    Suche(liste, w, einfuegestelle, groessererWert , Groesser);
    FuegeAnEinfuegestelleEin(neu, einfuegestelle);
  END;
END Einfuegen;
```

Diskussion: Dynamische Datentypen

- **Je deutlicher die Realisierung dynamischer Datenstrukturen durch Zeiger in der Sprache sichtbar ist,**
  - desto leichter fällt die softwaretechnisch ***unsaubere*** Verwendung.
- **Die explizite Speicherverwaltung führt zu einigen Problemen.**
  - Moderne imperative Sprachen sollten daher auf explizites Löschen verzichten und hierfür einen ***Garbage Collector*** besitzen.
- **Erst in objektorientierten Sprachen,**
  - die vorrangig mit ***Referenzsemantik*** arbeiten,
  - können grundlegende Probleme der Referenzierung gelöst werden.
- **Prozedurtypen sind nicht dynamisch, aber**
  - Prozedurvariablen können dynamisch unterschiedlichen Prozedurobjekten zugewiesen werden

Zusammenfassung: Datentypen

- **Datentypen**
  - definieren ***Werte*** und zulässige ***Operationen***
  - helfen, ***Abstraktionen*** zu formulieren
  - repräsentieren das ***Vokabular*** eines Systems
    - Person, Liste etc.
  - dienen der ***Sicherheit*** der Programme
    - jedes Objekt hat einen Typ
    - Typ-Prüfung bei Parameterübergabe und Zuweisung
  - müssen wohlüberlegt ***entworfen*** werden
    - manifestieren sich im Programm
    - sind dann nur ***schwer veränderbar***



Was haben wir gelernt?

- Charakterisierung statischer Datentypen (Wiederholung) und Defizit für Listenverarbeitung
- Typen für Listenelemente, Zeigertypen, dynamische Objekte (Haldenobjekte)
- dynamische Datenobjekte: Explizite Erzeugung, implizite Bezeichnung, Lebensdauer, Halde (Heap), Freigabe
- Derefenzieren, Lesen und Setzen von Haldenobjekten, Setzen von Zeigerobjekten, Vergleich
- Aliasing, nicht mehr greifbare Haldenobjekte, hängende Zeiger, Garbage Collection
- Lineare Liste: Einfügen/Löschen (vorn und in der Mitte), Suchen, geordnete lineare Liste
- Arten von Zeigern (Zweck ihrer Verwendung)
- Prozedurobjekt, Prozedurtyp, Prozedurvariable, Einsatz für Flexibilität

Glossar

- Wertsemantik, Verweis (Zeiger) semantik
- statische Datentypen, dynamische Datentypen, rekursive Datentypen
- Zeigertyp, Zeigerobjekt (-konstante oder -variable)
- Haldenobjekttyp, dynamische (anonyme) Datenobjekte: Haldenobjekte, Erzeugung mit Einrichtung eines Zeigerwerts, Löschen mit Löschen des Zeigerwerts
- statische Datenobjekte, Datenobjekte auf dem Kellerspeicher, Datenobjekte auf der Halde
- Zeiger: Dereferenzieren, Setzen, Lesen, Vergleichen, Setzen/Lesen von Komponenten von Haldenobjekten
- Aliasing, inaccessible objects, dangling references
- Ankerzeiger, Verkettungszeiger, Durchlaufzeiger, „semantische“ Zeiger, Abkürzung von Zugriffswegen
- einfach verkettete lineare Liste, sortierte Liste, doppelt verkettete Liste
- Listenoperationen: Einfügen vorn, in der Mitte, Suchen, Löschen vorn, in der Mitte, Suche als allgemeine Prozedur für Listenoperationen
- Prozedurtyp und Parameterprofil, signaturkonforme Prozedurobjekte, Prozedurvariable und Prozedurobjektzuweisung, Lebensdauer von Zeigerobjekten, von über Zeiger zugegriffenen Haldenobjekten