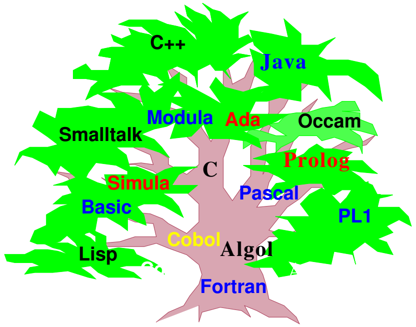
2. Überblick zu imperativen Programmiersprachen

- Historische Entwicklung
  - FORTRAN
  - ALGOL
  - COBOL
  - PL1
  - Pascal
  - C
  - Weitere Sprachen
- Daten- und Kontrollabstraktionen
- Bindung
  - Bindungszeitpunkte
  - Gültigkeitsbereiche
  - Speicherzuordnung
  - Typbindung
- Typsystem



FORTRAN (FORmula TRANslation)

- erste höhere Programmiersprache (1954-1957 bei IBM unter J. Backus entwickelt)
  - Schwerpunkt auf wissenschaftlichen und algebraischen Berechnungen
  - effiziente Compiler
- Neuerungen:
  - Maschinenunabhängigkeit
  - Arrays
  - über Laufvariable gesteuerte Schleifen (DO), IF-Anweisung
  - Kommentare, formatierte Ein/Ausgabe, ...
- Zunächst nur “arithmetische IF-Anweisung”

  IF (arithmetische Anweisung) L1, L2, L3
- Schleife mit nachträglicher Bedingungsauswertung

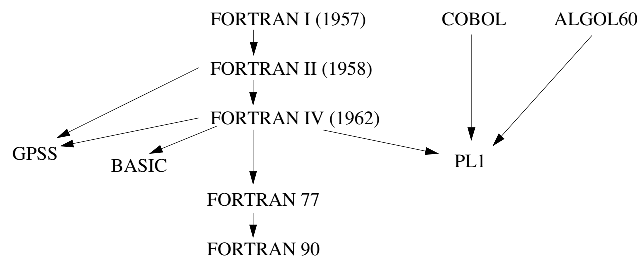
  DO L1 variable = first, last
- Allgemeine Merkmale
  - formatgebundene Programmgestaltung (Lochkartenformat)
  - teilweise implizite Typzuordnung (INTEGER falls Variablenname mit I, J, K, L oder M beginnt)

FORTRAN (2)

```
C     PROGRAMM ZUR MITTELWERTBERECHNUNG VON N ZAHLEN SOWIE DER ANZAHL
C     VON WERTEN, DIE GRÖSSER ALS DER MITTELWERT SIND
      DIMENSION A(99)
      REAL MEAN
C     EINLESEN VON ANZAHL N SOWIE N WERTEN
      READ (1,5) N
5     FORMAT (I2)
      READ (1,10) (A(I), I=1,N)
10    FORMAT (6F10.5)
C     MITTELWERTBERECHNUNG
      SUM = 0.0
      DO 15 I = 1,N
15    SUM = SUM + A(I)
      MEAN = SUM / FLOAT(N)
C     BESTIMMUNG, WIEVIELE WERTE > MEAN SIND
      NUMBER = 0
      DO 20 I = 1,N
      IF (A(I) .LE. MEAN) GOTO 20
      NUMBER = NUMBER+1
20    CONTINUE
      WRITE (2,25) MEAN, NUMBER
25    FORMAT (13H MITTELWERT: , F10.5, 5X, 21H #WERTE ÜBER MITTEL: ,I5)
      STOP
      END
```


FORTRAN (3)

- FORTRAN-Entwicklung:



- FORTRAN II: getrennte Übersetzbarkeit von Unterprogrammen (Subroutinen)
- FORTRAN IV: Typdeklarationen, logische IF-Anweisung, Prozeduren als Parameter
- FORTRAN 77: String-Behandlung, ELSE-Klausel (IF), erweiterte Schleifenkonstrukte
- FORTRAN 90: Array-Operationen, Records, Pointer, CASE, rekursive Prozeduren, ...

ALGOL (ALGOrithmic Language)

- Entwicklung von 1958-1960 als allgemeine Sprache zur Formulierung von Algorithmen
  - ALGOL58 wurde nicht realisiert
  - Compiler erst für ALGOL60
- erstmals formale Definition der Sprachsyntax (Backus-Naur-Form, BNF)
- Neuerungen:
  - Formatfreiheit für Quellprogramm
  - Typdeklarationen für Variable
  - block-strukturierte Sprache (Begin-End-Blöcke, Gültigkeits/Sichtbarkeitsbereiche für Variablen)
  - strukturierte Kontrollanweisungen (IF-THEN-ELSE-Anweisung, FOR-Schleifen)
  - Prozedurerweiterungen: Rekursion, Werteparameter (call by value), Laufzeitkeller
  - dynamische Arrays
- dennoch geringe praktische Bedeutung gegenüber FORTRAN
  - 3 Jahre später
  - geringe Unterstützung durch IBM
  - größere Komplexität für Programmierer, weniger effiziente Compiler
  - Sprachentwurf legte keine E/A-Möglichkeiten fest

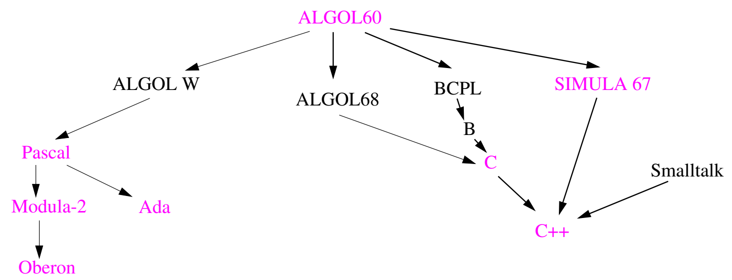
ALGOL (2)

- Programmbeispiel

```
begin
 comment       Mittelwertberechnung von N Zahlen sowie der Anzahl von
               Werten, die grösser als der Mittelwert sind;
 integer n;
 read (n);
 begin real array a[1:n];
     integer i, number;
     real sum, mean;
     sum := 0.0;
     for i := 1 step 1 until n do
          begin read (a[i]); sum := sum + a[i]; end;
     mean := sum / n;
     number := 0;
     for i := 1 step 1 until n do
               if a[i] > mean then number := number + 1;
     write ("Mittelwert: ", mean, " #Werte über Mittel: ", number);
 end;
end;
```


ALGOL (3)

- großer Einfluß auf weitere Entwicklung von Programmiersprachen



- ALGOL W (N. Wirth): Records, Referenzen (Pointer), CASE, While, ...
- Algol 68: benutzerdefinierte Typen, flexible Arrays

COBOL (COmmon Business Oriented Language)

- Entwicklung 1959/60 vom US-Verteidigungsministerium
- ANSI-Standardisierungen: 1968, 1974, 1985
- Einsatz v.a. in betriebswirtschaftlichen / kaufmännischen Anwendungen
- Neuerungen:
  - Record-Struktur (hierarchische Datenstrukturen)
  - Trennung von Datenstrukturen und Programmteil
  - umfangreiche Möglichkeiten der formatierten Ausgabe
  - lange Bezeichner, • • •
- geringer Einfluß auf weitere Sprachentwicklungen
  - Sprache weist großen Wortschatz auf und führt zu umfangreichen Programmen
  - bis 1974 keine Parameter für Prozeduren möglich
  - keine Funktionen
- weiterhin stark verbreitet

COBOL-Beispiel

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     INOUT.
* Dieses Programm liest eine Datei,
* erweitert jeden Satz und schreibt
* ihn in eine neue Datei
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT INP-FIL ASSIGN TO INFILE.
        SELECT OUT-FIL ASSIGN TO OUTFILE.
DATA DIVISION.
FILE SECTION.
FD      INP-FIL
        LABEL RECORDS STANDARD
        DATA RECORD IS REC-IN.
01      REC-IN.
         05 NAME        PIC A(30).
         05 TRENNER     PIC X(5).
         05 BERTRAG     PIC 9(5).
FD      OUT-FIL
        LABEL RECORDS STANDARD
        DATA RECORD IS REC-OUT.
01      REC-OUT.
         05 NAME        PIC A(30).
         05 TRENNER     PIC X(5).
         05 BERTRAG     PIC 9(5).
         05 ZUSATZ      PIC X(20).
```

```
WORKING STORAGE SECTION.
01      EOF PIC X VALUE IS 'N'.

PROCEDURE DIVISION.

AA.     OPEN INPUT INP-FIL
        OPEN OUTPUT OUT-FIL

        PERFORM CC
        PERFORM BB THRU CC UNTIL EOF='Y'

        CLOSE INP-FIL, OUT-FIL
        DISPLAY "Programm beendet."
        STOP RUN.

BB.     MOVE REC-IN TO REC-OUT.
        MOVE "BEMERKUNG" TO ZUSATZ
        WRITE REC-OUT.

CC.     READ INP-FIL
         AT END MOVE 'Y' TO EOF.

*********** Ende *************
```


PL/I

- Entwicklung 1963-64 von IBM
- Kombination von FORTRAN, COBOL und ALGOL60
- Neuerungen: Nebenläufigkeit und Ausnahmebehandlung (ON-Bedingungen)
- Elefantensprache: schwer erlernbar, fehleranfällig, aufwendige Übersetzung
- Programmbeispiel

```
EXAMPLE: PROCEDURE OPTIONS (MAIN);
/* Berechnung des MITTELWERTs und Anzahl größerer Werte (PL1-Version)*/
 GET LIST (N);
 IF N > 0 THEN BEGIN;
     DECLARE MEAN, A(N) DECIMAL FLOAT,
               SUM DEC FLOAT INITIAL (0), NUMBER FIXED INITIAL (0);
     GET LIST (A);
     DO I = 1 TO N;          SUM = SUM + A(I);    END;
     MEAN = SUM / N;
     DO I = 1 TO N;
               IF A(I) > MEAN THEN NUMBER = NUMBER + 1;
     END;
     PUT LIST ("Mittelwert: ", MEAN, " #Werte über Mittel: ", NUMBER);
END EXAMPLE;
```


Pascal

- Entwicklung 1971 von N. Wirth, ETH Zürich
- wesentliche Ziele
  - Unterstützung der strukturierten Programmierung
  - leichte Erlernbarkeit
  - effiziente Übersetzbarkeit
- flexibles Typsystem
  - einfache Standardtypen: integer, real, boolean, char
  - einfache benutzerdefinierte Typen: Aufzählungstypen und Bereichstypen
  - strukturierte Typen: array, record, set, file
  - Pointer-Typen
  - benutzerdefinierte Typbezeichnungen
- Kontrollanweisungen
  - bedingte Anweisungen: IF (THEN ELSE) sowie CASE
  - Schleifen: REPEAT neben WHILE und FOR
- weite Verbreitung in Ausbildung und Praxis

Pascal (2)

- Programmbeispiel

```
program pasex(input, output);
 {   Berechnung des Mittelwerts von n Zahlen sowie der Anzahl von
               Werten, die größer als der Mittelwert sind }
type Atype = array [1..99] of real;
var   A: Atype;
      i, n, number: integer;
      mean, sum: real;
begin
      sum := 0.0;
      readln (n);
      for i := 1 to n do
      begin    readln (A[i]);
               sum := sum + A[i]
      end;
      mean := sum / n;
      number := 0;
      for i := 1 to n do
               if A[i] > mean then number := number + 1;
      writeln ('Mittelwert: ', mean, '#Werte über Mittel: ', number);
end.
```

C

- 1972 bei Bell Labs entwickelt (D. Ritchie)
- enge Verbindung zu Unix-Betriebssystem
- Merkmale
  - Unterstützung systemorientierter Programmierung
  - zahlreiche Operatoren
  - hohe Effizienz
  - keine vollständige Typprüfung
  - schwer lesbare Programme
- zahlreiche Änderungen in ANSI C (1989)
- Programmbeispiel

```
#include <stdio.h>
main () /* Berechnung von MITTELWERT und Anzahl größerer Werte (C-Version)*/
{ float a[100], mean, sum = 0.0;
 int n, i, number = 0;
 scanf (”%d”, &n);
 for (i=0; i<n; i++) { scanf (“%f”, &a[i]); sum = sum + a[i]; }
 mean = sum / n;
 for (i=0; i<n; i++) { if (a[i] > mean) number++; }
 printf (“Mittelwert: %d\n #Werte über Mittel: %d\n“, mean, number);
}
```


Weitere imperative Sprachen

- APL (A Programming Language)
  - Entwicklung um 1960 (Harvard, IBM)
  - Programmierung mathematischer Berechnungen, v.a. auf Arrays und Matrizen
  - großer Umfang an Operatoren (automatischer Ablauf von Iterationen)
  - schwer lesbare Programme (keine Strukturierung, griechischer Symbolvorrat)
- BASIC (Beginners All-purpose Symbolic Instruction Code)
  - 1964 am Dartmouth College entwickelt (J. Kemeny, T. Kurtz)
  - Ziel: leichte Erlernbarkeit
  - trotz Mangel an modernen Sprachkonstrukten v.a. auf Heim-PCs weit verbreitet
- Ada
  - Entwurf 1980 beendet (J. Ichbiah) im Auftrag des US-Verteidigungsministeriums
  - wesentliche Neuerungen: abstrakte Datentypen (Packages), Nebenläufigkeit/Parallelität (Tasks), Ausnahmebehandlung (exception handling), generische Programmteile, Überladen (Overloading)
  - sehr großer Sprachumfang, hohe Komplexität
- Modula-2
  - Erstentwurf 1982 durch N. Wirth, Revisionen 1985 und 1988
  - Weiterentwicklung von Pascal um Modulkonzept (abstrakte Datentypen), Nebenläufigkeit (Koroutinen), Unterstützung von Systemprogrammierung

Datenabstraktionen

- Abstraktion von spezifischen Eigenschaften von Daten

1. *Fundamentale Datenabstraktion:* Abstraktion von interner Datenrepräsentation und Speicherung

- Konzept der (elementaren) Datentypen, z.B. Integer, Real, etc.

FORTRAN77	Pascal	C	Ada	Modula
INTEGER	integer	int	integer	INTEGER
REAL	real	float	float	REAL
LOGICAL	boolean	-	boolean	BOOLEAN
CHARACTER	char	char	character	CHARACTER
DOUBLE PRECISION	-	double	long_float	LONGREAL

- Konzept der Variablen

Datenabstraktionen (2)

2. *Strukturierte Datenabstraktion:* Bildung von Datenstrukturen durch Anwendung von Typkonstruktoren
   - Aggregation von Werten (kartesisches Produkt): Felder (Arrays) und Strukturen (Records)
   - Vereinigung: Union (Algol68, C) bzw. variante Records (Pascal, Modula, Ada)
   - Untermengenbildung: Bereichstypen (Pascal, Modula, Ada)
   - Potenzmengenbildung: Sets (Pascal, Modula)
   - Pointer-Typen

   Benutzerdefinierte Datentypen:
   - Definition problembezogener Datentypen mit eigener Typbezeichnung
   - Verbesserung der Lesbarkeit, Änderbarkeit und Prüfbarkeit der Typverträglichkeit

3. *Modulabstraktionen: Abstrakte Datentypen* (Bsp.: Ada, Modula)
   - problembezogene Datentypen mit spezifischen Zugriffsfunktionen
   - Trennung von Schnittstelle und Implementierung
   - Datenkapselung

Kontrollabstraktionen

- Abstraktion von Eigenschaften des Kontrollflusses, z.B. Abarbeitungsreihenfolge

1. Fundamentale Kontrollanweisungen: Abstraktion von Maschinenbefehlen
   - Zuweisung (abstrahiert von Speicherzugriffen)
   - GOTO-Befehl (abstrahiert von Sprunganweisungen)
2. Strukturierte Kontrollabstraktionen: Gruppierung von Anweisungen
   - bedingte Anweisungen (IF bzw. Fallunterscheidung)
   - strukturierte Schleifenmechanismen (While, Repeat, For, Loop)
   - Unterprogramme (Prozeduren, Funktionen)
3. Modulabstraktionen: Gruppierung von zusammengehörigen Prozeduren innnerhalb von Moduln
   => abstrakte Datentypen

Bindung (binding)

- wesentliches Unterscheidungskriterium zwischen Programmiersprachen
- Bindung = Zuordnung von Attributen (Eigenschaften) zu einem Namen (Bezeichner) bzw. von Operationen zu einem Symbol
- Beispiele (Pascal):

`const n = 5;`	Name `n`; Attribute: Konstante, Wert `5`
`var x: integer;`	Name `x`; Attribute: Variable, Datentyp `integer`
`function f (n: integer): boolean;` `...`	Name `f` Attribute: Funktion, Parameteranzahl, Parametername, Parametertyp, Ergebnistyp, Programmrumpf
`x := 2`	Name `x`, neues Attribut: Wert `2`
`new (y)`	Name `y`, Attribut: neuer Wert Name `y^`, Attribut: Speicherplatz

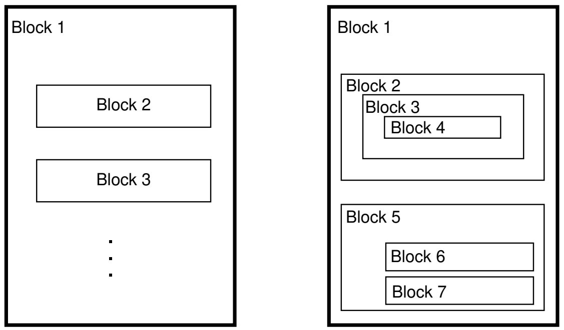
- je nach Bindungszeit (vor oder während der Ausführung) unterscheidet man zwischen statischer und dynamischer Bindung bzw. statischen und dynamischen Attributen

Bindung (2)

- genauere Unterscheidung von Bindungszeitpunkten
  - Sprachdefinitionszeit (Bedeutung vordefinierter Operatoren wie *, Attribute von vordefinierten Namen wie `boolean` etc.)
  - Sprachimplementierungszeit (realisierungsspezifische Attribute für vordefinierte Namen wie Konstante `maxint` etc.)
  - Übersetzungszeit (z.B. Datentyp einer Variablen, Wert einer Konstanten)
  - Link-Zeit (z.B. Rumpf einer externen Prozedur)
  - Ladezeit (z.B. Speicherplatz für globale Variablen)
  - Ausführungszeit (z.B. Wert einer Variablen, Speicherplatz für dynamisch erzeugte Variablen)
- Compiler verwaltet Attribute innerhalb einer Symboltabelle
- wichtige Attribute von Variablen
  - Name
  - Adresse
  - Wert
  - Typ
  - Gültigkeitsbereich (scope) -> lokale vs. globale Variablen
  - Speicherzuteilung (statisch, semidynamisch/automatisch, dynamisch)

Gültigkeitsbereiche

- Gültigkeitsbereich (scope) einer Variable legt ihre Sichtbarkeit fest
- meist statische (lexikographische) Gültigkeitsbereiche
  - Verwendung bei Programmiersprachen mit Blockstruktur (Algol, Pascal, Modula, Ada, C, etc.)
  - Variable v ist in dem Block B sichtbar, in dem sie deklariert wird, sowie in allen inneren Blöcken
  - in äußeren Blöcken von B ist v nicht gültig
  - Verwendung gleicher Namen in unterschiedlichen Blöcken kann zu Überdeckung führen



Gültigkeitsbereiche (2)

- dynamische Gültigkeitsbereiche: Zuordnung einer Variablen v erfolgt zur letzten durchlaufenen Deklaration für v (Bsp. LISP)

```
...
var x: integer;
procedure A;
 begin ... write (x); ... end;
procedure B;
 var x: real;
 begin ... x:= 2.0; ... A; ... end;
begin (* Hauptprogramm *)
 ... x := 1; ... B; ... A; ...
end.
```

- Vorteile statischer Gültigkeitsbereiche: hohe Sicherheit, Effizienz, Verständlichkeit

Speicherzuordnung

- bestimmt Lebensdauer von Variablen (generell dynamisches Attribut <-> Gültigkeitsbereich)
- Speicherzuordnung erfolgt bei Blockeintritt (statisch bzw. semidynamisch) oder dynamisch durch spezielle Anweisungen (Zeigervariablen)
- statische Variablen (z.B. globale Variablen):
  - Speicherallokation zu Beginn der Programmausführung (Start des Hauptprogramms)
  - feste Speicherzuordnung bis zum Programmende
- semidynamische Variablen
  - automatische Allokation bei Blockeintritt (z.B. Prozeduraufruf)
  - Freigabe bei Blockaustritt
  - Verwaltung im Kellerspeicher
- statische, lokale Variablen (Bsp.: ALGOL, PL1, C)
  - Allokation zu Beginn der Programmausführung
  - Problem der Initialisierung
- dynamische Variablen
  - explizite Allokation und Deallokation
  - Speicherung auf Halde

Typbindung

- vor Zugriff auf eine Variable muß Typzuordnung erfolgt sein
- Typdeklaration impliziert statische Typzuordnung (-> statisch typisierte Sprachen)
  - meist explizite Deklaration

```
VAR i, j: INTEGER;
```

  - Alternative: implizite Deklaration über Default-Konventionen (Bsp.: FORTRAN, BASIC, PL1)
    erstes Auftreten einer Variablen führt zu impliziter Typzuordnung
- dynamische Typbindung: Datentyp einer Variablen wird erst bei Wertzuweisung ermittelt

```
LIST <- 10.2 5.8 1.9
• • •
LIST <- 54
```

  - Beispiele von dynamisch typisierten Sprachen: APL, SNOBOL4, LISP
  - Nachteile:
- bei Konstanten erfolgt häufig (statische) Typzuordnung über den Wert

Typsystem

- Typsystem einer Sprache besteht aus
  - Verfahren zur Konstruktion von Typen
  - Regeln zur Festlegung von Typäquivalenz und Typkompatibiltät (-> Typüberprüfungen)
  - Regeln zur Ableitung von Typinformationen (Typinferenz)
- Typprüfungen (type checking)
  - Operanden von Operatoren müssen kompatible Typen aufweisen, d.h. der Typ entspricht der Operatorspezifikation oder kann automatisch in einen zulässigen Typ konvertiert werden
  - dynamische Typbindung impliziert dynamische Typprüfungen
  - statische Typbindung garantiert nicht immer, daß alle Typprüfungen zur Übersetzungszeit erfolgen (Probleme v.a. bei Vereinigungstypen)
- Streng typisierte Sprachen (strongly typed languages)
  - Datentypen sämtlicher Namen/Objekte stehen zur Übersetzungszeit fest
  - alle Typfehler werden zur Übersetzungszeit erkannt
  - Beispiele streng typisierter Sprachen: Algol68, Ada
  - Pascal und Modula weisen “Schlupflöcher” auf (-> statisch jedoch nicht streng typisiert)

Typäquivalenz (Typgleichheit)

- strukturelle Äquivalenz: Typen weisen dieselbe Struktur (denselben Wertebereich) auf
  - Verwendung in Algol60, Algol68, FORTRAN, COBOL
  - unterschiedliche Interpretationen üblich bei Abweichungen im Indexbereich von Arrays und Komponentennamen von Records

```
TYPE A1 = array [1..10]of real;
     A2 = array [0..9]of real;
     R1 = record x: boolean;
                 y: A1; end;
```

```
R2 = record x: boolean;
        y: array [1..10]of real;
     end;

R3 = record  y: A1;
             x: boolean; end;
```

- Namensäquivalenz (Ada): Typen müssen denselben Typnamen aufweisen

```
TYPE A1 = array [1..10]of real;
     A3 = A1;
     Alter = integer;
```

```
VAR s1, s2: array [1..10]of real;
```

- Deklarationsäquivalenz: Zwischenform von Struktur- und Namensäquivalenz (Pascal, Modula)
  - Typnamen sind äquivalent, wenn sie auf denselben Ursprungstyp(namen) zurückgeführt werden können

```
TYPE A1 = array [1..10]of real;
     A4 = array [1..10]of real;
```

- C: Deklarationsäquivalenz für Strukturen und Vereinigungen, sonst Strukturäquivalenz

Typäquivalenz (2)

	Ja	Nein	?
Strukturäquivalenz			
Namensäquivalenz			
Deklarationsäquivalenz			

Typkompatibilität

- Lockerung der Korrektheitsregeln gegenüber Typäquivalenz
- Beispiele
  - Pascal, Modula: Unterbereichstypen sind kompatibel zum Basistyp
  - gemischte Real/Integer-Ausdrücke teilweise zulässig in Pascal, z.B. `r2 := r1 + i;`
  - große Freiheitsgrade bezüglich gemischten Ausdrücken in C
    zulässiger Ausdruck: `i1 = i2 + 4.867`
- Zuweisungskompatibilität (Zuweisung: x := e;)
  - zulässig, falls x und e vom selben Typ sind (obligatorisch für Ada)
  - Ausnahmen bei unterschiedlichen Typen sind sprachspezifisch
    - Pascal: auch zulässig, falls x vom Typ `real` und e vom Typ `integer`
    - Modula: CARDINAL und INTEGER sind zuweisungsverträglich
- Ausdruckskompatibilität
  - Operandentypen müssen übereinstimmen oder verträglich sein
  - Ada und Modula verlangen strenge Ausdruckskompatibilität, d.h. Ausdrücke mit gemischten Typen sind nicht zulässig (unzulässig in Modula: `i1 := c1 + i2;`)
  - geringe Restriktionen in PL1 oder C (Addition von Integer und Character, etc.)

Typumwandlung

- Arten der Typumwandlung
  - implizit durch das Typsystem (Typanpassung)
  - explizit durch Konversionsfunktionen oder Typtransfers (casts)
- Automatische Typanpassungen beeinträchtigen strenge Typisierung und Zuverlässigkeit
- Typkonversion durch expliziten Aufruf von Funktion zur Umwandlung von Werten
  - Pascal: trunc, round
  - Modula: TRUNC, FLOAT (Bsp.: `i := TRUNC (r);`)
- Typtransfers (casts): i.a. keine echte Wertänderung, sondern neue Interpretation der internen Repräsentation
  - Modula: Hilfsfunktionen CARDINAL, INTEGER, ... (Bsp.: `i1 := INTEGER (c1) + i2;`)
  - C: i = (int) r
    ipointer = (int *) malloc (sizeof (int))