Imperative Sprachen

- Blöcke
  - Bereich im Programmtext (in-line-Block)
  - oder Rumpf einer Funktion/Prozedur
  - mit Anfangs- und Ende-Markierung
  - lokale (auf Block beschränkte) Deklarationen möglich

äußerer Block

```
{int x = 2;
        {int y = 3;
         x = y + 4;
        }
…
}
```

innerer Block

- im inneren Block: y lokale Variable, x globale Variable
- im äußeren Block: nur x lokale Variable

Imperative Sprachen

Eigenschaften Block-strukturierter Programmiersprachen

- neue Variablendeklarationen an verschiedenen Stellen in Programm
- Deklaration innerhalb des umgebenden Blocks sichtbar
- Schachtelung von Blöcken erlaubt, Überlappung nicht
- zur Laufzeit: bei Eintritt in Block Speicherbereitstellung für hier deklarierte Variablen
- zur Laufzeit: bei Verlassen des Blocks Speicherfreigabe des Speichers lokaler Variablen
- nicht lokal deklarierter Bezeichner: global, Deklaration aus nächst-umgebendem Block wird gewählt

Imperative Sprachen

```
{….
int x = 1;
int y = 3;
float z = 2.5;
…
    {int z = 12;
     int x = 5;
     …
          {float w = x + z;
           int x = 0;
           int v = y *2;
           …
           }
    }
}
```

Wert von

x	y	z	w	v
1	-	-	-	-
1	3	-	-	-
1	3	2.5	-	-
1	3	12	-	-
5	3	12	-	-
5	3	12	17	-
0	3	12	17	-
0	3	12	17	6
5	3	12	-	-
1	3	2.5	-	-

Imperative Sprachen

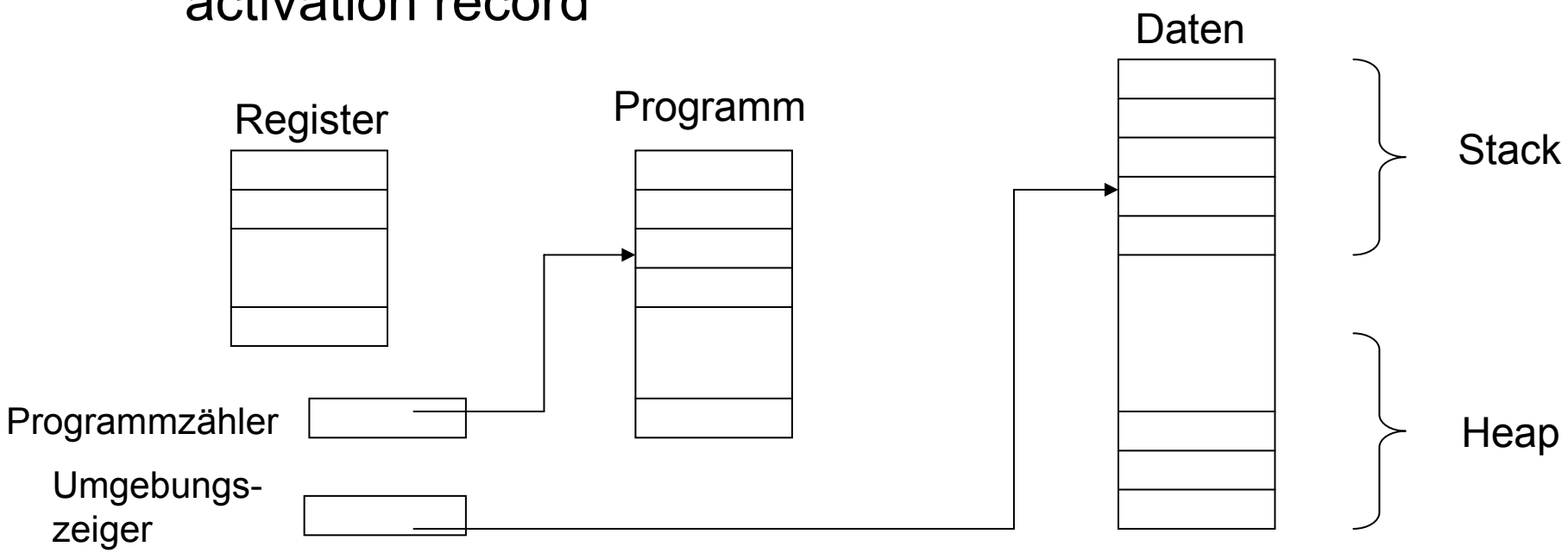
Variablenklassen im Zusammenhang mit Blöcken

- lokale Variable: Speicherung im activation record des Blocks (auf Stack)
- Parameter: ebenfalls in activation record des Blocks
- globale Variable: gespeichert im activation record eines umgebendes Blocks

Imperative Sprachen

Speicher-Management in blockorientierten Sprachen:

- Trennung Programmspeicher und Datenspeicher
- Programmzähler mit Adresse der akt. Anweisung
- Register
- Umgebungszeiger (in Datenspeicher) zeigt auf akt. activation record



Imperative Sprachen

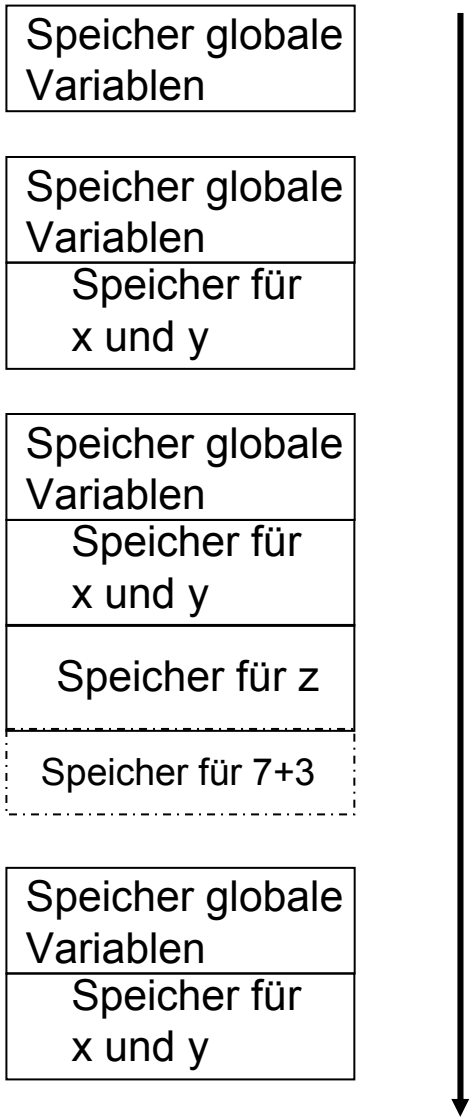
In-line-Blocks

1. lokale Variablen:

```
. . .
{int x = 0;
 int y = x+1;

        {int z = (7+3)*8;}

. . .
}
```



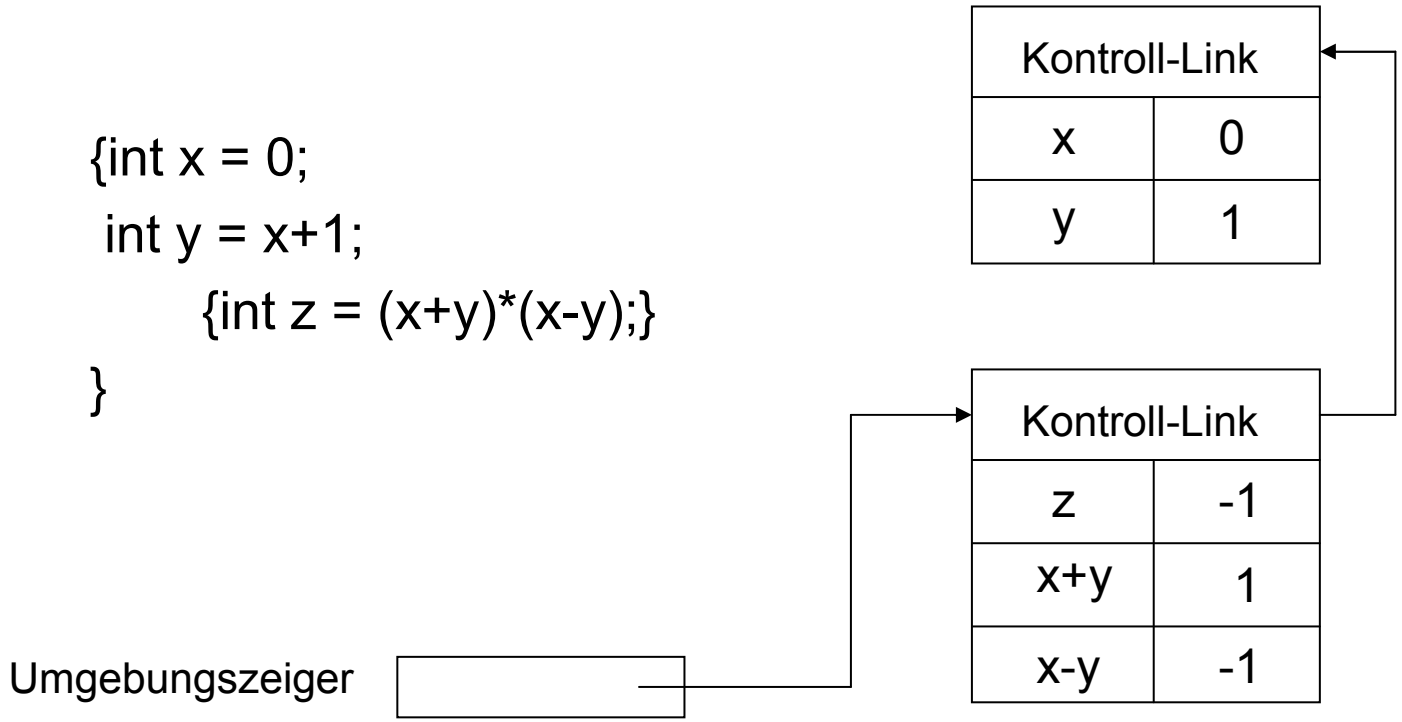
Veränderung des Stacks zur Ausführungs-zeit

Imperative Sprachen

In-line Blocks

2. globale Variablen:

```
{int x = 0;
 int y = x+1;
        {int z = (x+y)*(x-y);}
}
```

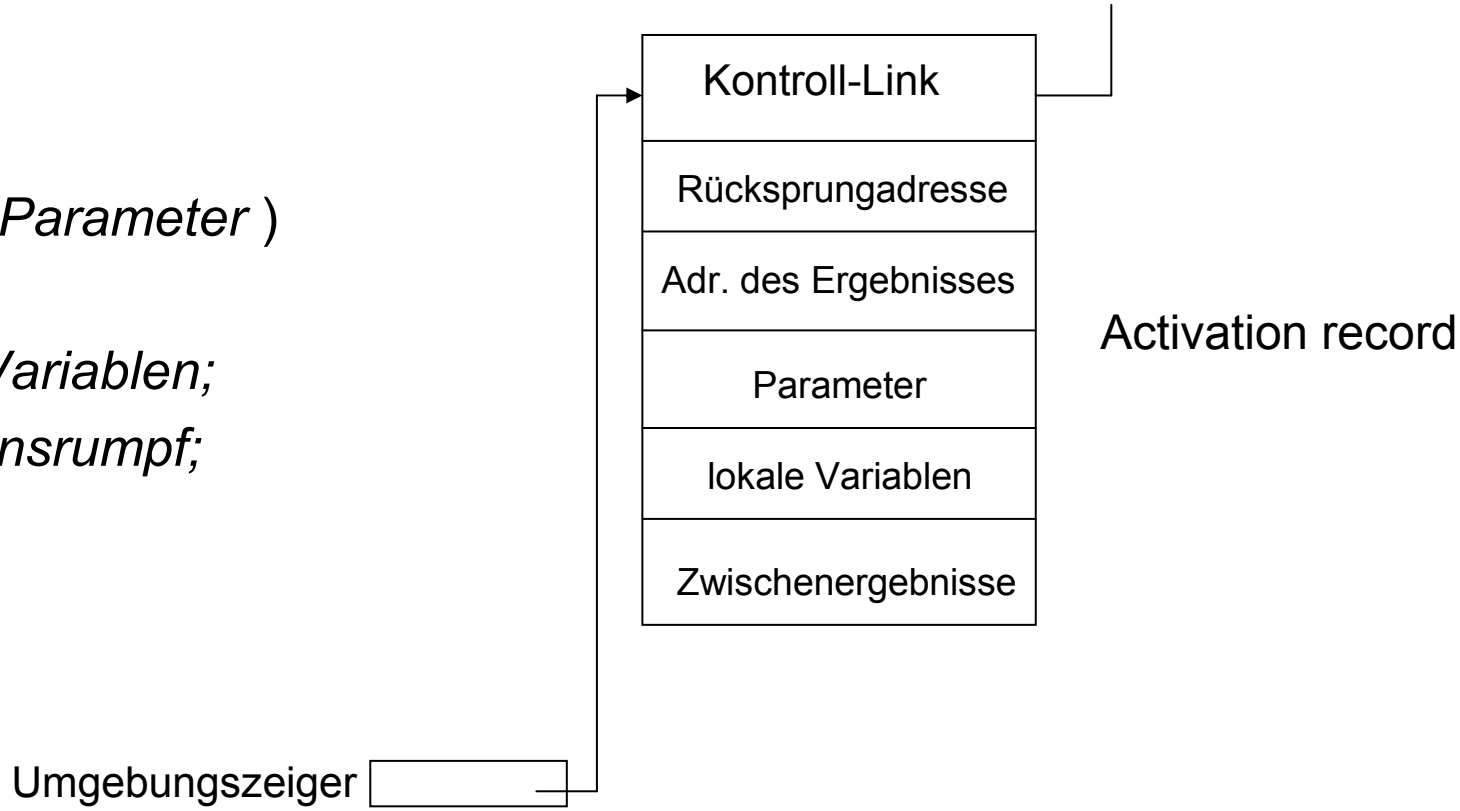


Imperative Sprachen

Block als Funktions-/Prozedurrumpf:

1. Parameter und lokale Variablen:

type f (*Parameter*)
{
lokale Variablen;
Funktionsrumpf;
}

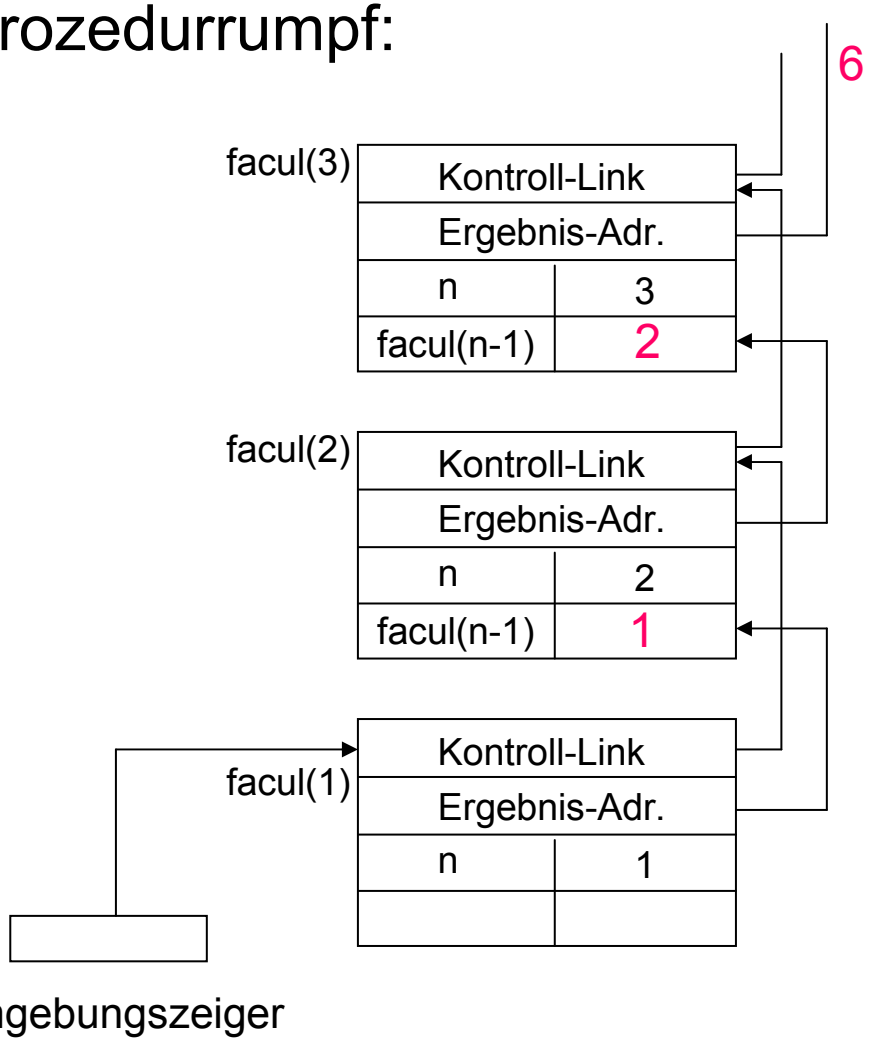


Imperative Sprachen

Block als Funktions-/Prozedurrumpf:

```
int facul ( int n )
{
if (n<=1)
    return 1;
else
    return n*facul(n-1);
}

…
x=facul(3);
```



Imperative Sprachen

Block als Funktions-/Prozedurrumpf:

2. Parameterübergabe:

– in Deklaration: *formale* Parameter
– bei Aufruf: *aktuelle* Parameter

– Unterschiede bei der Parameterübergabe
  - bzgl. Berechnungszeit des akt. Parameters (vor vs. während Ausführung des Rumpfs)
  - bzgl. des Speicherplatzes für aktuellen Parameter (neuer vs. Zeiger auf existierenden)

– Zwei Möglichkeiten, aktuellen Parameter zu übergeben:
  - Call-by-Value (Pass-by-Value, Wert-Parameter)
  - Call-by-Reference (Pass-by-Reference, Referenz-Parameter)

Imperative Sprachen

Parameter in Pascal:

```
PROGRAM myprog;
VAR i: INTEGER;
PROCEDURE erhoehe (zahl: INTEGER);
        BEGIN
                zahl := zahl+1;
        END;

BEGIN
        i := 4;
        erhoehe (i);
        write(`i hat den Wert `);
        writeln(i);
END;
```

Was gibt das Programm aus?

Imperative Sprachen

Referenz-Parameter in Pascal: mittels VAR kennzeichnen

```
PROGRAM myprog;
VAR i: INTEGER;
PROCEDURE erhoehe (VAR zahl: INTEGER);
        BEGIN
                zahl := zahl+1;
        END;

BEGIN
        i := 4;
        erhoehe (i);
        writeln(`i hat den Wert `,i);
END;
```

Ohne VAR: Wert-Parameter

Imperative Sprachen

Regeln zum Umgang mit Parametern:

- in Prozeduren keine globalen Variablen, sondern nur Parameter benutzen
- wo Wert-Parameter genügt, auch nur Wertparameter nehmen
- soll Prozedur Ergebnis liefern, ist Referenz-Parameter nötig

Ausnahmen:

- große Datenstrukturen als Parameter: Referenzparameter spart Speicherplatz
- FILE-Datentyp (sequenzielle Folge gleichen Typs) muss immer als Referenzparameter übergeben werden
- Standardeingabe(input)/Standardausgabe(output) können überall als globale Variable verwendet werden

Imperative Sprachen

Block als Funktions-/Prozedurrumpf:

3. globale Variablen:

```
int x=1;                    /*  Block 0  */
int g(int z)
        {return x+z;}       /*  Block 1  */
int f(int y)
        {int x = y+1;       /*  Block 2  */
         return g(y*x);}

int erg = f(3);             /*  Welchen Wert hat erg?  */
```

Imperative Sprachen

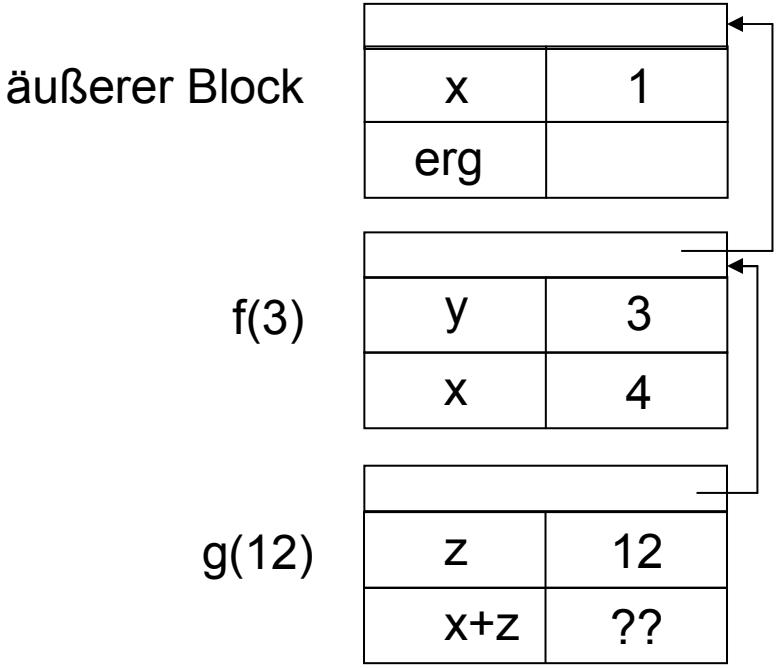
zwei Vorgehensweisen zum Auffinden der Deklaration globaler Variablen

- Statische Reichweite: Bezeichner bezieht sich auf gleichnamigen Bezeichner in der nächsten Umgebung des Programmtextes

- Dynamische Reichweite: Bezeichner bezieht sich auf gleichnamigen Bezeichner des vorherigen Activation Records

Imperative Sprachen

Runtime-Stack des Beispiels:



dynamische Reichweite: x (in x+z) entspricht x aus zuletzt erzeugtem Activation Record – also 4

→ erg = 16

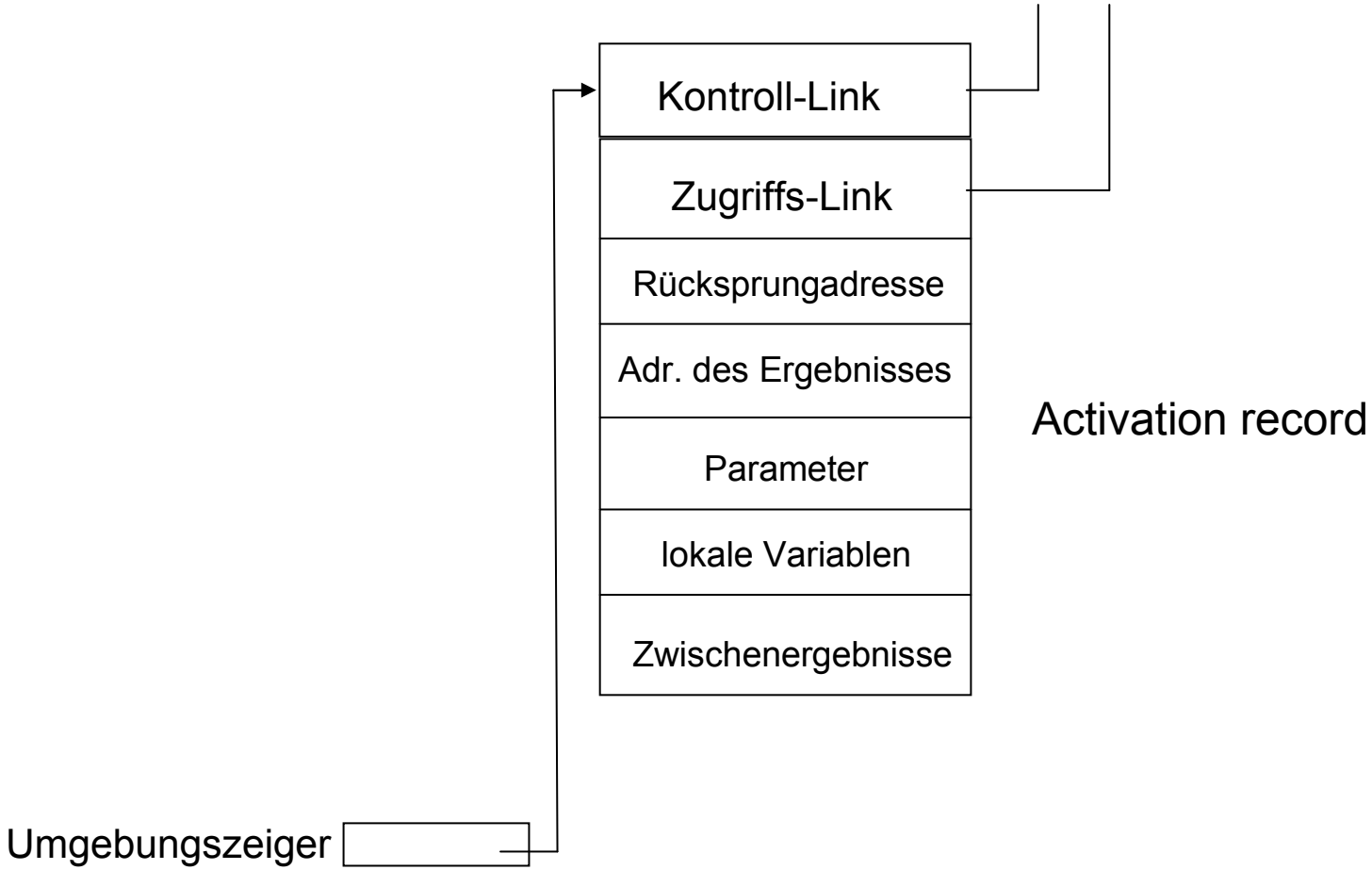
statische Reichweite: x entspricht x aus nächst-umgebendem Block im Quelltext – also x = 1

→ erg = 13

Was unterstützt C, Pascal, Java,…?

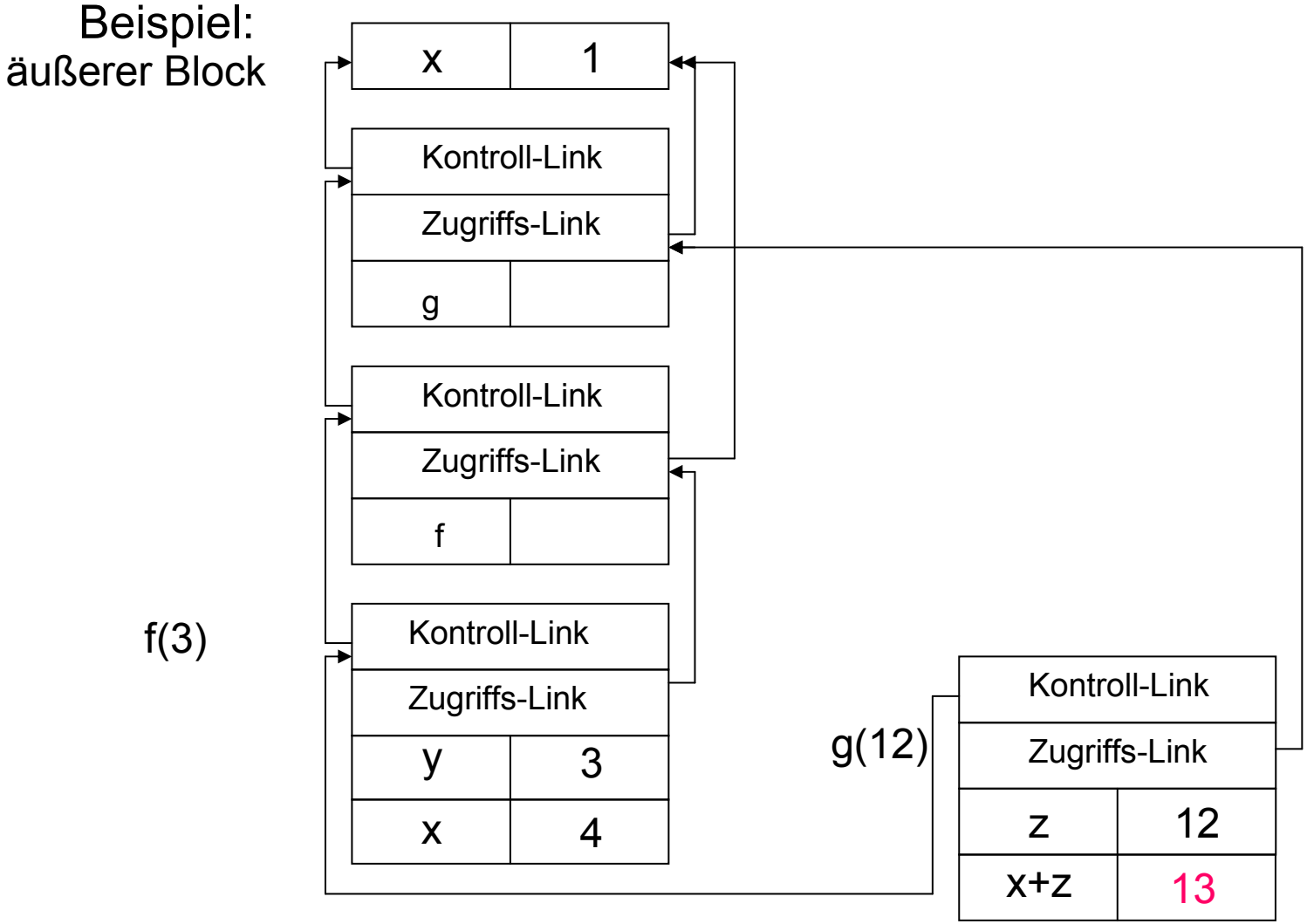
Imperative Sprachen

Activation Record mit sog. Zugriffs-Link bei statischer Reichweite



Imperative Sprachen

Runtime-Stack mit Zugriffs-Link (statische Reichweite) zu obigem Beispiel:



Imperative Sprachen

Funktion/Prozedur als Parameter:

- für statische Reichweite Erweiterung d. Activation Records nötig
- Hilfsmittel: Closures
- Closure: Zeigerpaar (Zeiger auf Activation Record, Zeiger auf Funktionscode)
- in Activation Record: formaler Parameter ist Funktion → Zeiger auf zugeh. Closure eintragen
- in Activation Record: Funktion als aktueller Parameter → Zugriffslink entspr. Zeiger auf Activation Record im zugeh. Closure

Imperative Sprachen

Beispiel Closure:

```
int x = 4;
int f (int y) { return x*y;}
int g (int → int h) {
    int x = 7;
    return h(3)+x;
}
g(f);
```