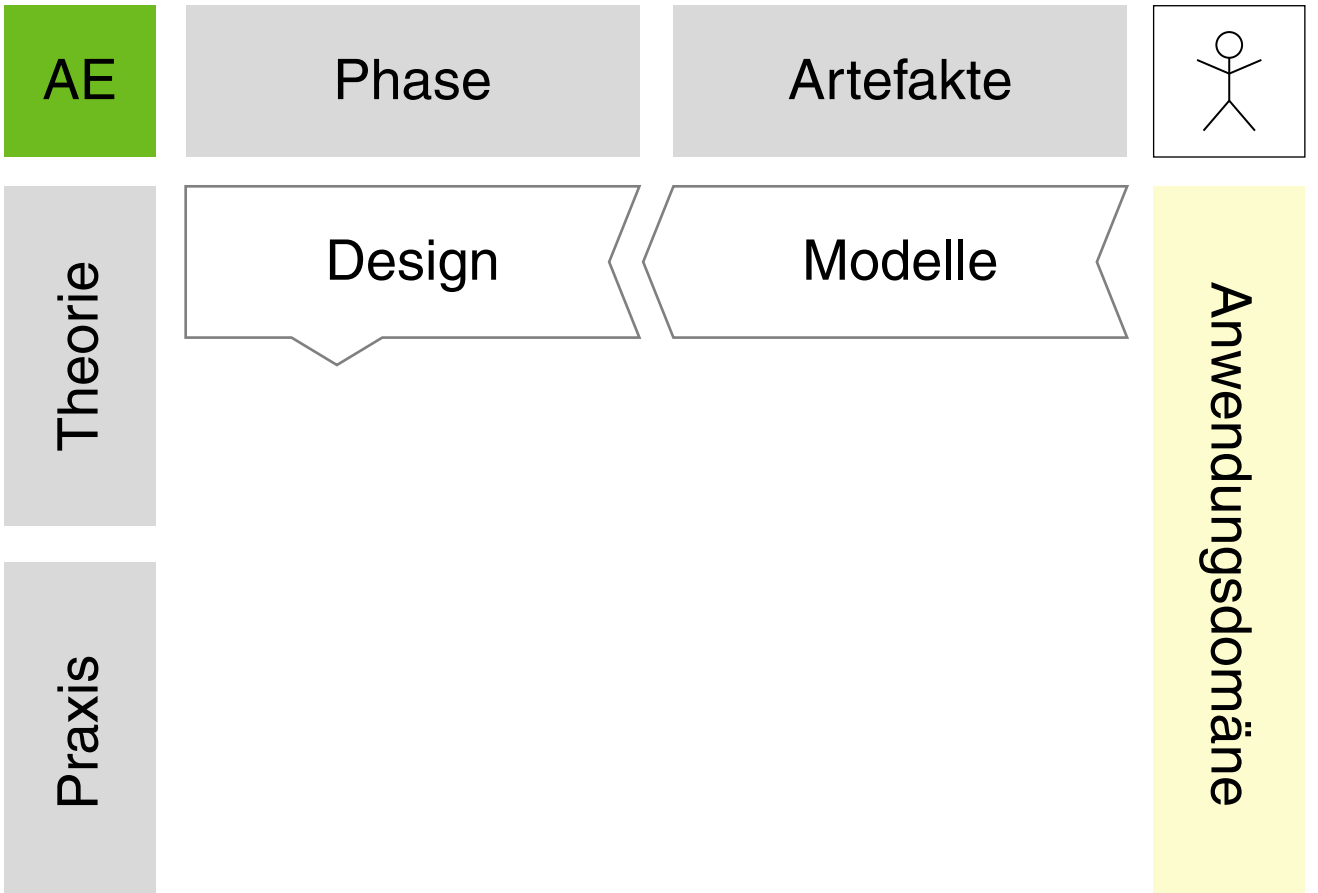
Kapitel ADS:II

II. Algorithm Engineering

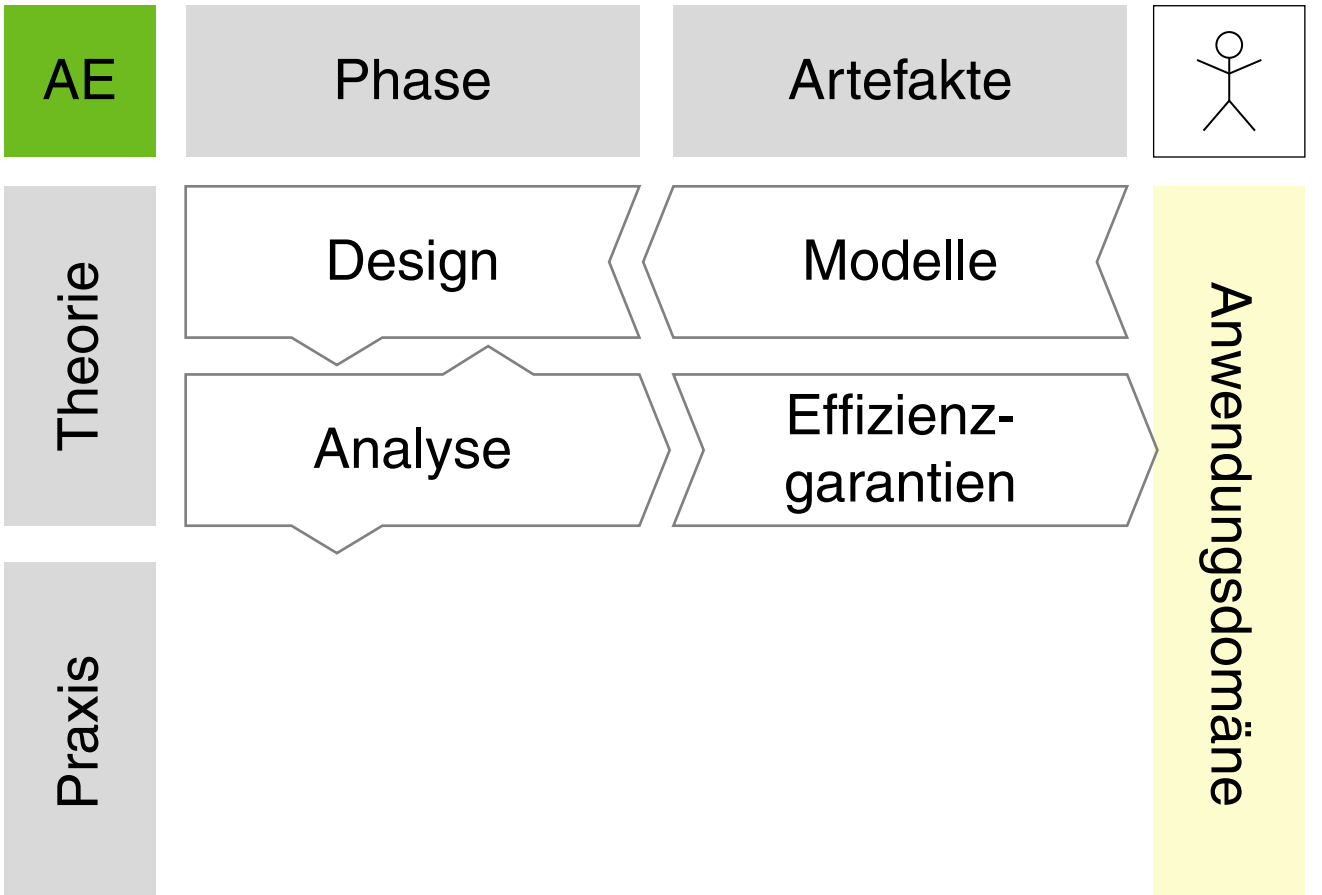
- Problemklassen und Lösungsstrategien
- Phasen des Algorithm Engineering
- Exkurs: Programmiersprachen
- Pseudocode
- Rekursive Algorithmen
- Algorithmendesign
- Algorithmenanalyse
- Algorithmenimplementierung
- Algorithmenevaluierung

Phasen des Algorithm Engineering

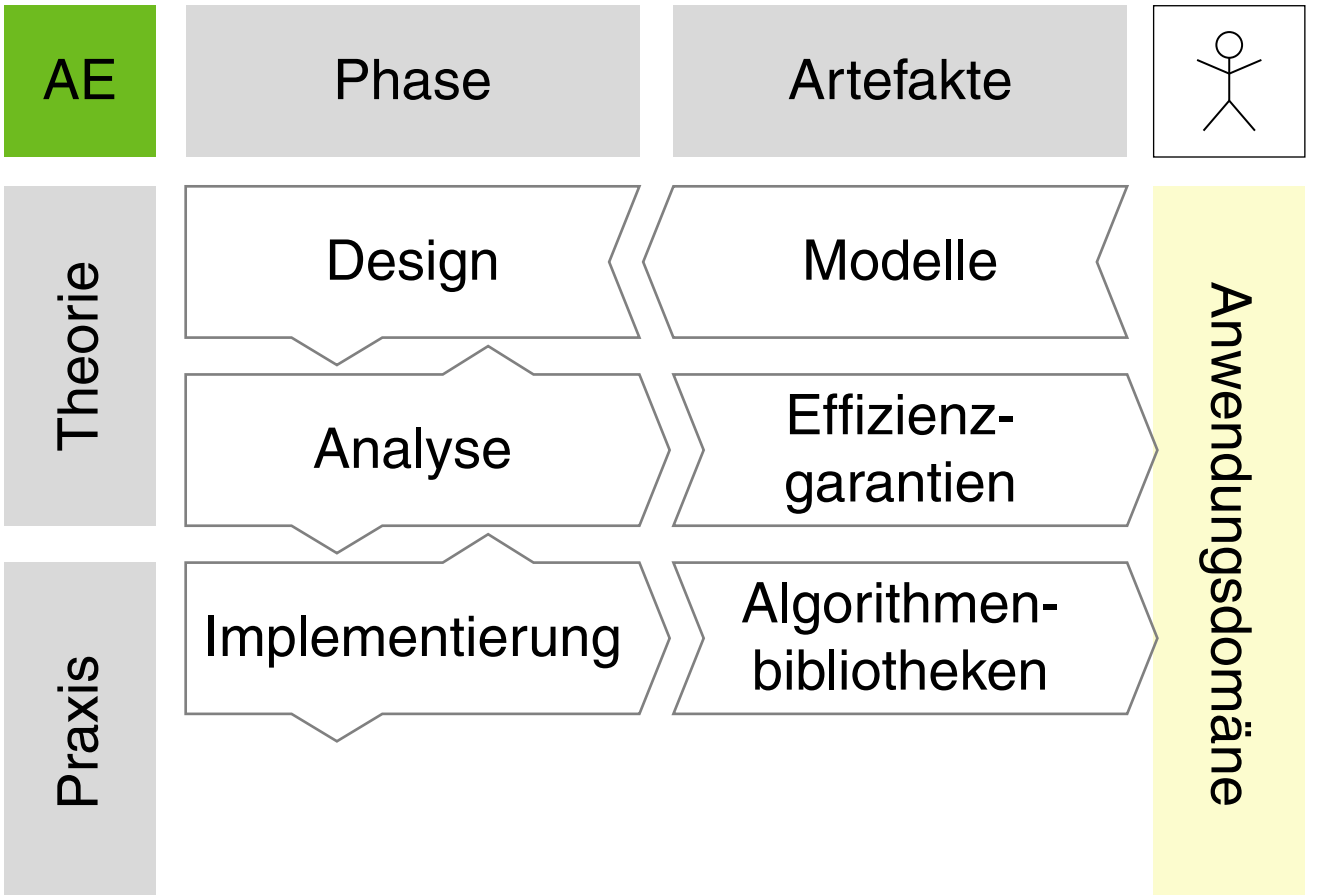
Phasen des Algorithm Engineering



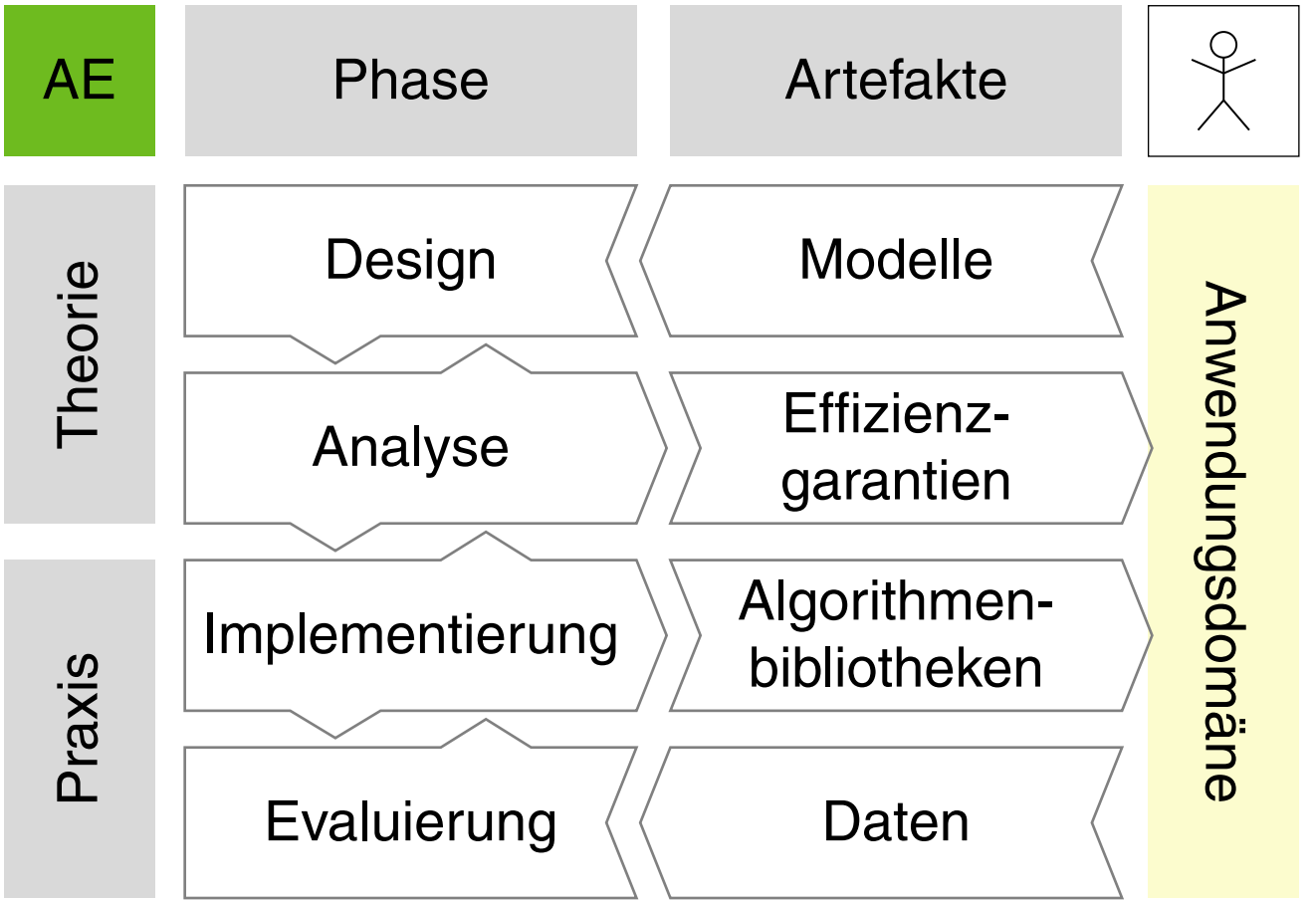
Phasen des Algorithm Engineering



Phasen des Algorithm Engineering



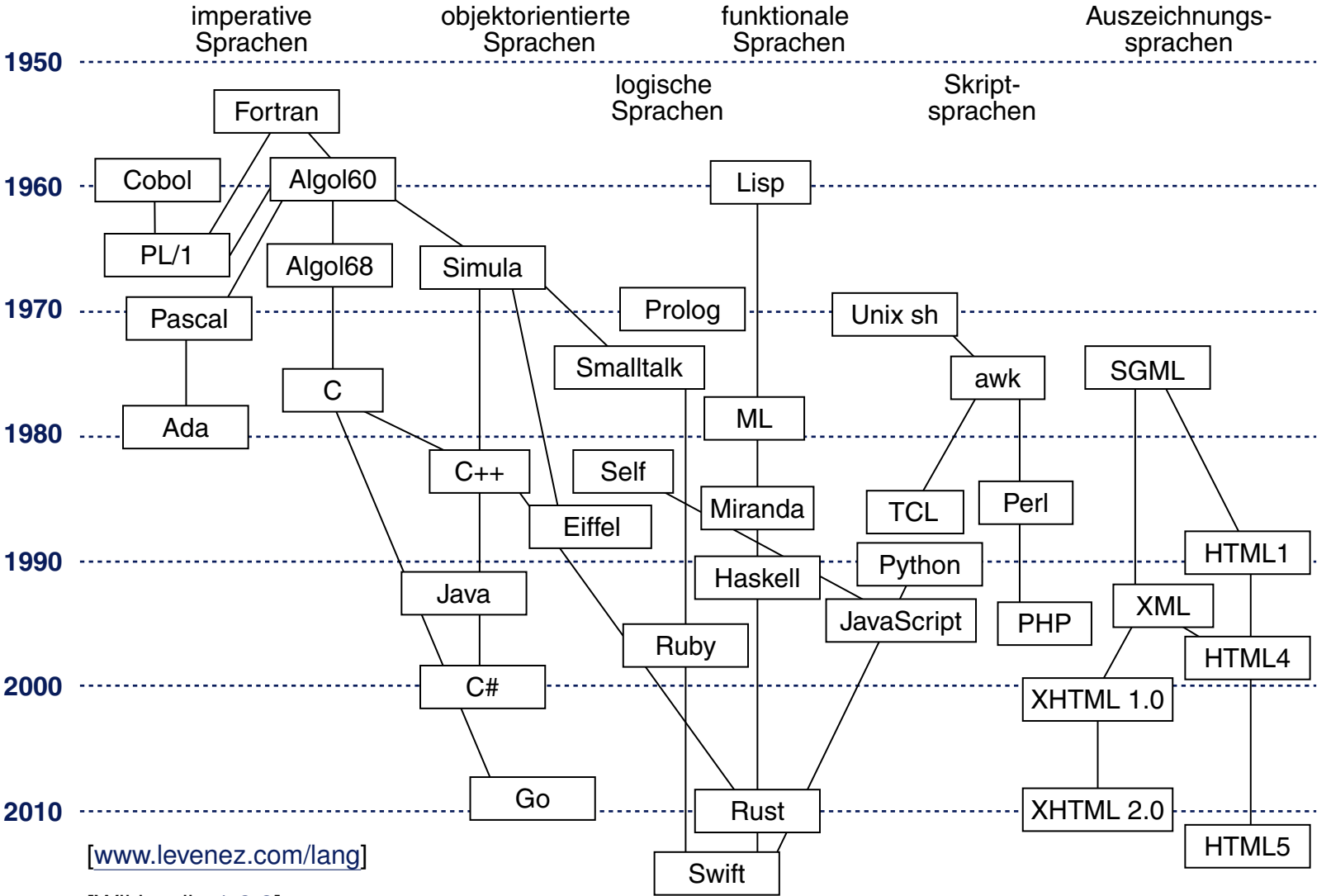
Phasen des Algorithm Engineering



Bemerkungen: (frei nach [Sanders 2009])

- Phasen:
  - Design: Entwurf eines Algorithmus gemäß harten und weichen Randbedinungen wie zum Beispiel Terminiertheit, Korrektheit, Effizienz sowie Einfachheit, Implementierbarkeit und Modularität.
  - Analyse: Theoretische Betrachtung von Algorithmen bezüglich ihrer Komplexität sowie Beweis ihre Korrektheit und Terminiertheit.
  - Implementierung: Robuste Umsetzung von Algorithmen in einer konkreten Programmiersprache, gegebenenfalls unter Ausnutzung von Hardwarespezifika.
  - Evaluierung: Experimentelle Auswertung von Algorithmenimplementierungen zur Prüfung theoretischer Annahmen, zum Vergleich alternativer Implementierungen und zum Beleg der Praxistauglichkeit.
- Artefakte:
  - Modelle: Ein Modell ist ein (vereinfachtes) Abbild eines Systems, das seinen Nutzer dazu ermächtigt, Fragen bezüglich des Systems zu beantworten.
  - Effizienzgarantien: Schranken bezüglich Zeit- und Platzverbrauch.
  - Algorithmenbibliotheken: Auf Wiederverwendbarkeit, Erweiterbarkeit und leichte Nutzbarkeit ausgelegte Sammlungen von Algorithmen für bestimmte Problemstellungen.
  - Daten: Reale oder möglichst realistische Probleminstanzen um die Valididät und Vergleichbarkeit experimenteller Ergebnisse sicherzustellen.

Exkurs: Programmiersprachen



[www.levenez.com/lang]

[Wikipedia 1 2 3]

Exkurs: Programmiersprachen [Kastens 2005]

Ebenen von Spracheigenschaften

Ein Satz einer Sprache ist eine Folge von Zeichen eines gegebenen Alphabets.
Zum Beispiel ist ein PHP-Programm ein Satz der Sprache PHP:

```
$line = fgets ($fp, 64);
```

Exkurs: Programmiersprachen [Kastens 2005]

Ebenen von Spracheigenschaften

Ein Satz einer Sprache ist eine Folge von Zeichen eines gegebenen Alphabets. Zum Beispiel ist ein PHP-Programm ein Satz der Sprache PHP:

```
$line = fgets ($fp, 64);
```

Die Struktur eines Satzes wird auf zwei Ebenen definiert:

1. Notation von Symbolen (Lexemen, Token).
2. Syntaktische Struktur.

Die Bedeutung eines Satzes wird auf zwei weiteren Ebenen an Hand der Struktur für jedes Sprachkonstrukt definiert:

3. Statische Semantik.
   Eigenschaften, die *vor* der Ausführung bestimmbar sind.
4. Dynamische Semantik.
   Eigenschaften, die *erst während* der Ausführung bestimmbar sind.

Exkurs: Programmiersprachen [Kastens 2005]

Ebene 1: Notation von Symbolen

Ein Symbol wird aus einer Folge von Zeichen des Alphabets gebildet. Die Regeln zur Notation von Symbolen werden durch reguläre Ausdrücke definiert.

```
$line = fgets ($fp, 64);
```

Exkurs: Programmiersprachen [Kastens 2005]

Ebene 1: Notation von Symbolen

Ein Symbol wird aus einer Folge von Zeichen des Alphabets gebildet. Die Regeln zur Notation von Symbolen werden durch reguläre Ausdrücke definiert.

```
$line = fgets ($fp, 64);
```

Wichtige Symbolklassen in Programmiersprachen:

Symbolklasse	**Beispiel in PHP**
Bezeichner *(Identifier)* Verwendung: Namen für Variablen, Funktionen, etc.	`$line, fgets`
Literale *(Literals)* Verwendung: Zahlkonstanten, Zeichenkettenkonstanten	`64, "telefonbuch.txt"`
Wortsymbole *(Keywords)* Verwendung: kennzeichnen Sprachkonstrukte	`while, if`
Spezialzeichen Verwendung: Operatoren, Separatoren	`<= = ; { }`

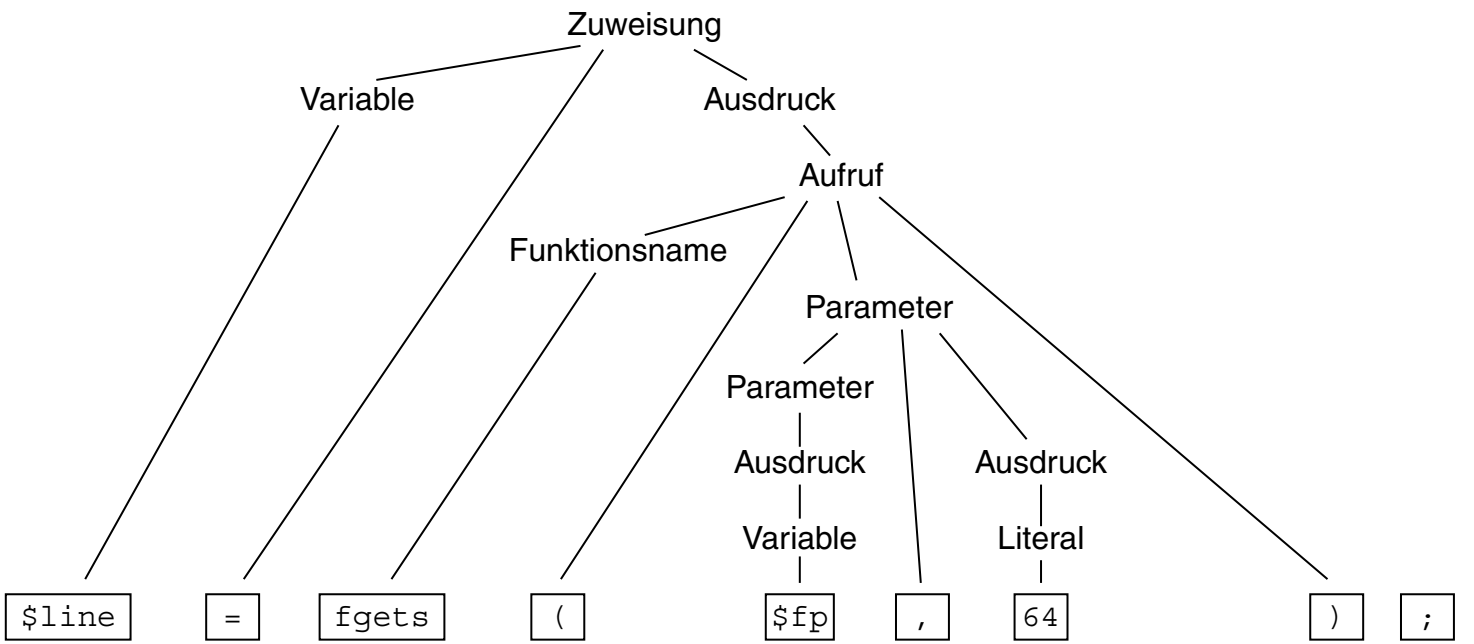
Bemerkungen:

- Zwischenräume, Tabulatoren, Zeilenwechsel und Kommentare zwischen den Symbolen dienen der Lesbarkeit und sind sonst bedeutungslos.
- In Programmiersprachen bezeichnet der Begriff „Literal“ Zeichenfolgen, die zur Darstellung der Werte von Basistypen zulässig sind. Sie sind nicht benannt, werden aber über die jeweilige Umgebung ebenfalls in die Programmressourcen eingebunden. Literale können nur in rechtsseitigen Ausdrücken auftreten. Meist werden die Literale zu den Konstanten gerechnet und dann als literale Konstanten bezeichnet, da beide – im Gegensatz zu Variablen – zur Laufzeit unveränderlich sind.
  Das Wort „Konstante“ im engeren Sinn bezieht sich allerdings mehr auf in ihrem Wert unveränderliche Bezeichner, d.h., eindeutig benannte Objekte, die im Quelltext beliebig oft verwendet werden können, statt immer das gleiche Literal anzugeben. [Wikipedia]

Exkurs: Programmiersprachen [Kastens 2005]

Ebene 2: Syntaktische Struktur

Ein Satz einer Sprache wird in seine Sprachkonstrukte gegliedert; sie sind meist ineinander geschachtelt. Diese syntaktische Struktur wird durch einen Strukturbaum dargestellt, wobei die Symbole durch Blätter repräsentiert sind:



Die Syntax einer Sprache wird durch eine kontextfreie Grammatik definiert. Die Symbole sind die Terminalsymbole der Grammatik.

Exkurs: Programmiersprachen [Kastens 2005]

Ebene 3: Statische Semantik

Eigenschaften von Sprachkonstrukten, die ihre Bedeutung (Semantik) beschreiben, soweit sie anhand der Programmstruktur festgestellt werden können, ohne das Programm auszuführen (= statisch).

Elemente der statischen Semantik für übersetzte Sprachen:

- Bindung von Namen.
  Regeln, die einer Anwendung eines Namens seine Definition zuordnen.
  Beispiel: zu dem Funktionsnamen in einem Aufruf muss es eine Funktionsdefinition mit gleichem Namen geben.

Exkurs: Programmiersprachen [Kastens 2005]

Ebene 3: Statische Semantik

Eigenschaften von Sprachkonstrukten, die ihre Bedeutung (Semantik) beschreiben, soweit sie anhand der Programmstruktur festgestellt werden können, ohne das Programm auszuführen (= statisch).

Elemente der statischen Semantik für übersetzte Sprachen:

- Bindung von Namen.
  Regeln, die einer Anwendung eines Namens seine Definition zuordnen.
  Beispiel: zu dem Funktionsnamen in einem Aufruf muss es eine Funktionsdefinition mit gleichem Namen geben.

- Typregeln.
  Sprachkonstrukte wie Ausdrücke und Variablen liefern bei ihrer Auswertung einen Wert eines bestimmten Typs. Er muss im Kontext zulässig sein und kann die Bedeutung von Operationen näher bestimmen.
  Beispiel: die Operanden des „$*$“-Operators müssen Zahlwerte sein.

Exkurs: Programmiersprachen [Kastens 2005]

Ebene 4: Dynamische Semantik

Eigenschaften von Sprachkonstrukten, die ihre Wirkung beschreiben und erst bei der Ausführung bestimmt oder geprüft werden können (= dynamisch).

Elemente der dynamischen Semantik:

- Regeln zur Analyse von Voraussetzungen, die für eine korrekte Ausführung eines Sprachkonstruktes erfüllt sein müssen.
  Beispiel: ein numerischer Index einer Array-Indizierung, wie in `$var[$i]`, darf nicht kleiner als 0 sein.

Exkurs: Programmiersprachen [Kastens 2005]

Ebene 4: Dynamische Semantik

Eigenschaften von Sprachkonstrukten, die ihre Wirkung beschreiben und erst bei der Ausführung bestimmt oder geprüft werden können (= dynamisch).

Elemente der dynamischen Semantik:

- Regeln zur Analyse von Voraussetzungen, die für eine korrekte Ausführung eines Sprachkonstruktes erfüllt sein müssen.
  Beispiel: ein numerischer Index einer Array-Indizierung, wie in `$var[$i]`, darf nicht kleiner als 0 sein.

- Regeln zur Umsetzung bestimmter Sprachkonstrukte.
  Beispiel: Auswertung einer Zuweisung der Form

  Variable = Ausdruck

  Die Speicherstelle der Variablen auf der linken Seite wird bestimmt. Der Ausdruck auf der rechten Seite wird ausgewertet. Das Ergebnis ersetzt dann den Wert an der Stelle der Variablen. [SELFHTML]

Bemerkungen:

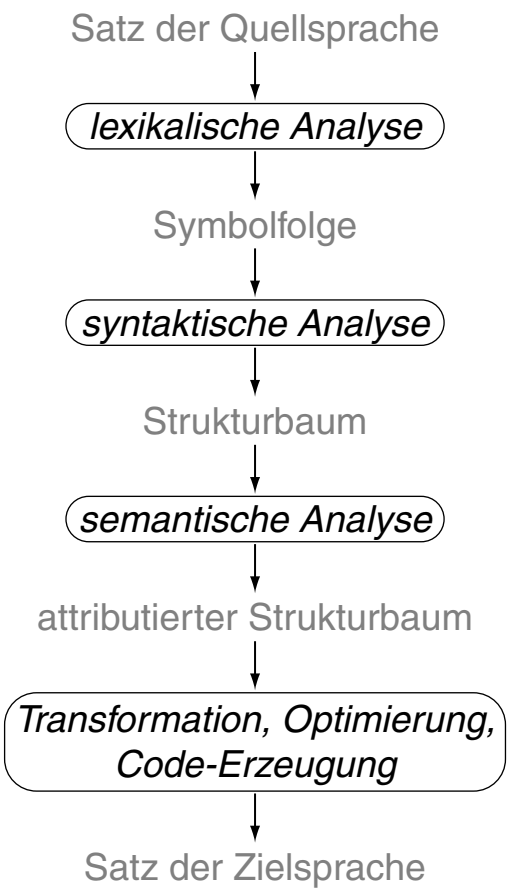
- Auf jeder der vier Ebenen gibt es also Regeln, die korrekte Sätze erfüllen müssen.
- In der Sprache PHP gehören die Typregeln zur dynamischen Semantik, da sie erst bei der Ausführung des Programms anwendbar sind.
- In der Sprache JavaScript gehören die Bindungsregeln zur statischen Semantik und die Typregeln zur dynamischen Semantik.

Exkurs: Programmiersprachen [Kastens 2005]

Übersetzung von Sprachen

Ein Übersetzer transformiert jeden korrekten Satz (Programm) der Quellsprache in einen gleichbedeutenden Satz (Programm) der Zielsprache.

- Die meisten Programmiersprachen zur Software-Entwicklung werden übersetzt. Beispiele: C, C++, Java, Ada, Modula.
- Zielsprache ist dabei meist eine Maschinensprache eines realen Prozessors oder einer abstrakten Maschine.
- Übersetzte Sprachen haben eine stark ausgeprägte statische Semantik.
- Der Übersetzer prüft die Regeln der statischen Semantik; viele Arten von Fehlern lassen sich vor der Ausführung finden.



Exkurs: Programmiersprachen [Kastens 2005]

Interpretation von Sprachen

Ein Interpretierer liest einen Satz (Programm) einer Sprache und führt ihn aus.

Für Sprachen, die strikt interpretiert werden, gilt:

- sie haben eine einfache Struktur und keine statische Semantik
- Bindungs- und Typregeln werden erst bei der Ausführung geprüft
- nicht ausgeführte Programmteile bleiben ungeprüft

Beispiele: Prolog, interpretiertes Lisp

Moderne Interpretierer erzeugen vor der Ausführung eine interne Repräsentation des Satzes; dann können auch Struktur und Regeln der statischen Semantik vor der Ausführung geprüft werden.

Beispiele: die Skriptsprachen JavaScript, PHP, Perl

Bemerkungen:

- Es gibt auch Übersetzer für Sprachen, die keine einschlägigen Programmiersprachen sind: Sprachen zur Textformatierung (LaTeX$\rightarrow$ PDF), Spezifikationssprachen (UML $\rightarrow$ Java).
- Interpretierer können auf jedem Rechner verfügbar gemacht werden und lassen sich in andere Software integrieren.
- Ein Interpretierer schafft die Möglichkeit einer weiteren Kapselung der Programmausführung gegenüber dem Betriebssystem.
- Interpretation kann 10-100 mal zeitaufwändiger sein, als die Ausführung von übersetztem Maschinencode.

Pseudocode

Einführung

Charakteristika:

- imperativ, strukturiert
- keine standardisierte Syntax; nur Konventionen
- syntaktische Elemente entlehnt aus gängigen Programmiersprachen
- Verwendung natürlicher Sprache sowie mathematischer Notation
- unabhängig von zugrunde liegender Technologie
- Förderung der Interpretation durch Menschen
- hinreichend formal, um Mehrdeutigkeiten zu vermeiden und die korrekte manuelle Übersetzung in eine Programmiersprache zu ermöglichen

Pseudocode

Einführung

Charakteristika:

- imperativ, strukturiert
- keine standardisierte Syntax; nur Konventionen
- syntaktische Elemente entlehnt aus gängigen Programmiersprachen
- Verwendung natürlicher Sprache sowie mathematischer Notation
- unabhängig von zugrunde liegender Technologie
- Förderung der Interpretation durch Menschen
- hinreichend formal, um Mehrdeutigkeiten zu vermeiden und die korrekte manuelle Übersetzung in eine Programmiersprache zu ermöglichen

Anwendung:

- Analyse von Algorithmen in Forschung und Lehre
- Hilfsmittel im Softwareentwurf zur Dokumentation und Refaktorierung

Pseudocode

Einführung

Algorithmus: Insertion Sort.

Eingabe: A. Array von n Zahlen.

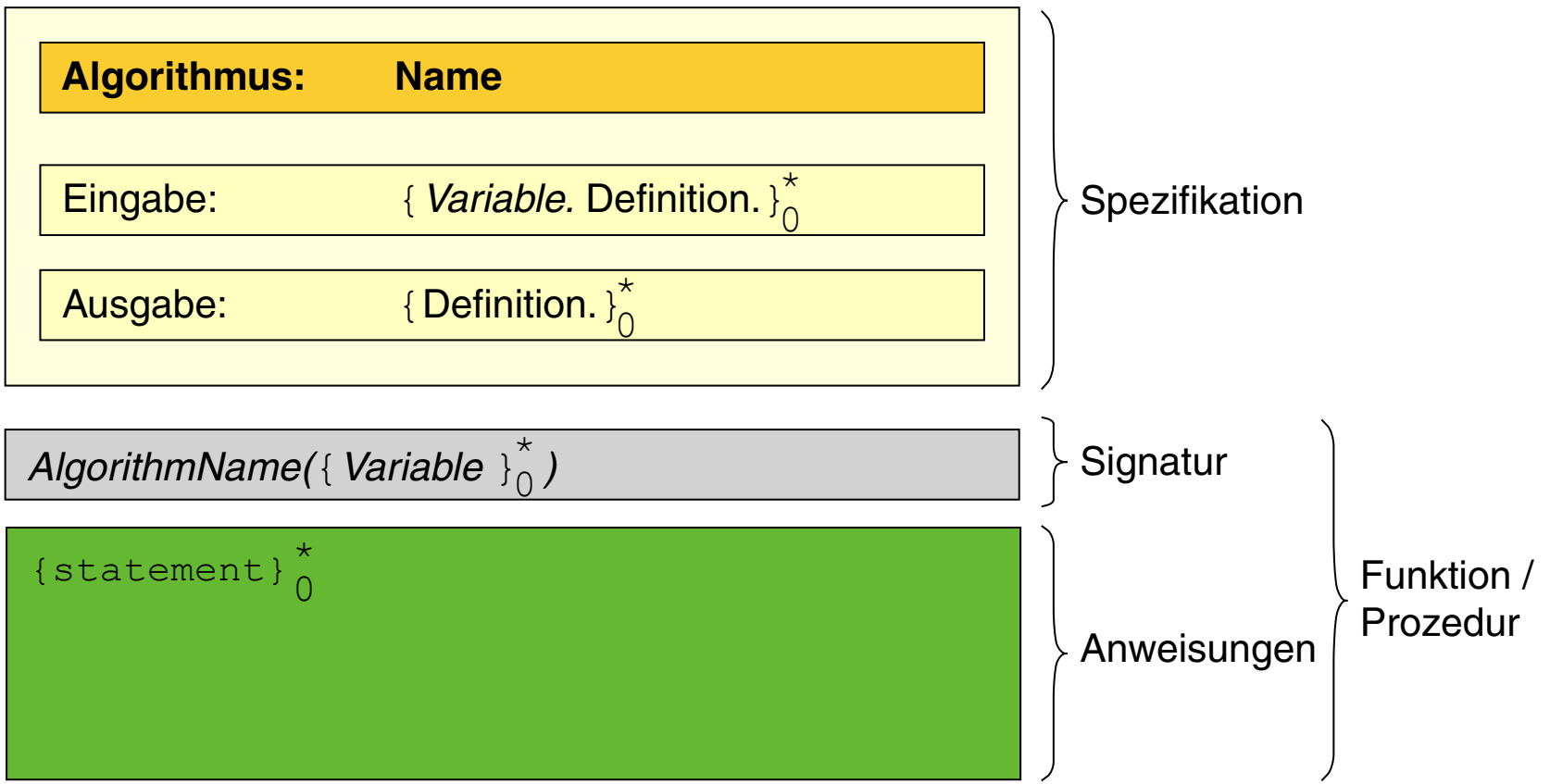
Ausgabe: Eine aufsteigend sortierte Permutation von A.

InsertionSort(A)

```
1.  FOR j = 2 TO n DO
2.    a_j = A[j]
3.    i = j − 1
4.    WHILE i > 0 AND A[i] > a_j DO
5.      A[i + 1] = A[i]
6.      i = i − 1
7.    ENDDO
8.    A[i + 1] = a_j
9.  ENDDO
10. return(A)
```

Pseudocode

Einführung

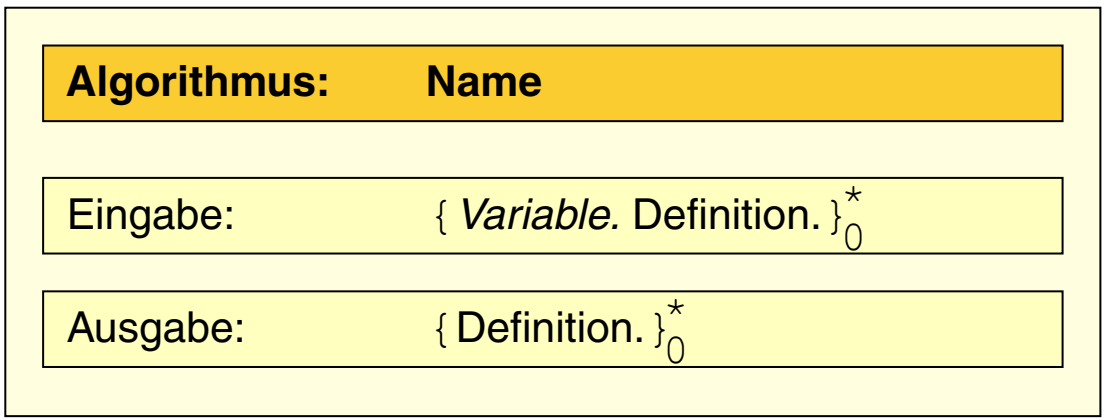


Bemerkungen:

- Anders als vollwertige Pogrammiersprachen erlaubt Pseudocode die Darstellung von Algorithmen in der für Menschen anschaulichsten Weise. Es ist explizit erlaubt, die nachfolgend genannten Konventionen abzuändern, solange es der Verständlichkeit dient.
- Die große Anzahl verfügbarer Programmiersprachen macht einen Konsens einer für die Darstellung von Algorithmen geeigneten, vollwertigen Sprache nahezu unmöglich.
- Auch Pseudocode ändert sich mit der Zeit, jedoch langsamer als Programmiersprachen. Es werden heute syntaktische Elemente moderner Sprachen übernommen.
- Pseudocode kompakt:
  1. Spezifikation und Signatur
  2. Grundlagen der Syntax
  3. Variablen
  4. Operatoren
  5. Datentypen
  6. Kontrollstrukturen

Pseudocode

Spezifikation und Signatur

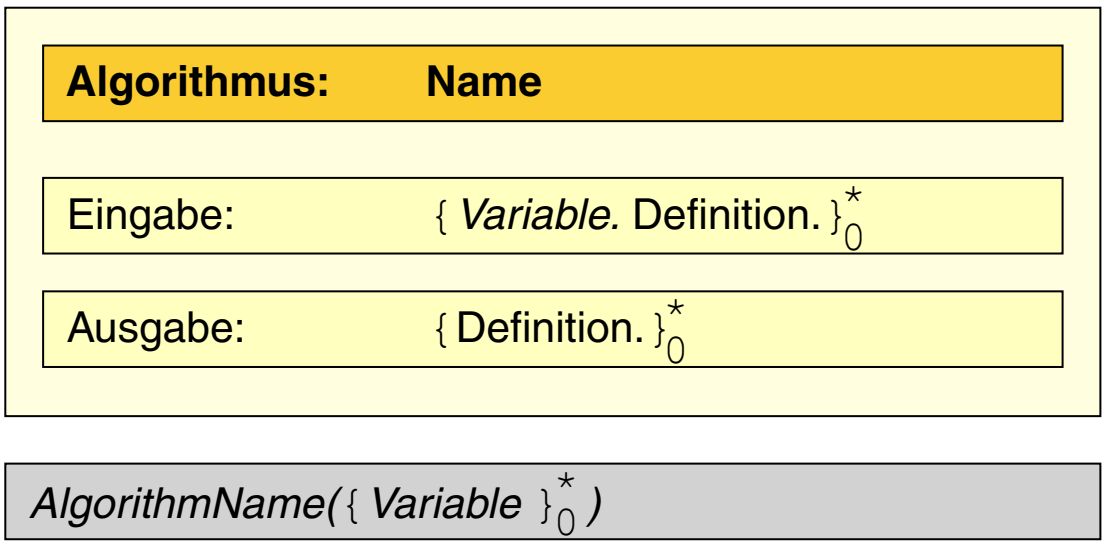


Spezifikation:

- Name des Algorithmus
  Offizieller oder am weitesten verbreiteter Name; gegebenenfalls auf Englisch.
- Eingabe
  Variable-Definition-Paare mit Nennung der jeweils erwarteten Datenstruktur.
- Ausgabe
  Definition der Ausgabe mit Nennung der Datenstruktur.

Pseudocode

Spezifikation und Signatur



Signatur:

- Bezeichner
  Name des Algorithmus in *CamelCase*-Notation.
- Parametertupel
  Eingabeparameter als Tupel; kommaseparierte Liste der Variablen.

Pseudocode

Grundlagen der Syntax

Bezeichner:

- Namen sind Zeichenketten ohne Leerzeichen in *CamelCase*-Notation.
- Namen von Variablen und Hilfsfunktionen beginnen mit Kleinbuchstaben.
- Mathematische Objekte werden als einzelne Buchstaben verschiedener Alphabete gemäß der Konventionen der Mathematik bezeichnet. Optional sind hoch- und tiefgestellte Variablen erlaubt.

Anweisungen:

- Ein Semikolon am Zeilenende ist möglich, kann aber entfallen.
- Zwischen Anweisungen in derselben Zeile muss ein Semikolon stehen.
- `//` kommentiert bis Zeilenende aus.

Bemerkungen:

- CamelCase (zu deutsch Binnenmajuskel) ist eine Namenskonvention für Bezeichner in Programmiersprachen, bei der Großbuchstaben im Wortinneren verwendet werden, um die Lesbarkeit zusammengefügter Wörter zu erhöhen.

Pseudocode

Variablen

- Variablen werden durch Initialisierung gleichzeitig definiert und deklariert.
- Eine Variable kann Werte beliebigen Typs annehmen.
- Unterscheidung von lokalen und globalen Variablen.

- Eine Variable ist lokal für eine Funktion, wenn sie innerhalb des Bindungsbereichs der Funktion deklariert wird.
- Ihre Gültigkeit beginnt ab der Zeile ihrer Deklaration bis zum Ende einer umgebenden Schleife bzw. der Funktion.
- Globale Variablen gelten im ganzen Programm, dass die Funktion eines Algorithmus ausführt und müssen nicht explizit als Eingabeparameter übergeben, jedoch als Teil der Spezifikation definiert werden.

Pseudocode

Variablen

- Variablen werden durch Initialisierung gleichzeitig definiert und deklariert.
- Eine Variable kann Werte beliebigen Typs annehmen.
- Unterscheidung von lokalen und globalen Variablen.

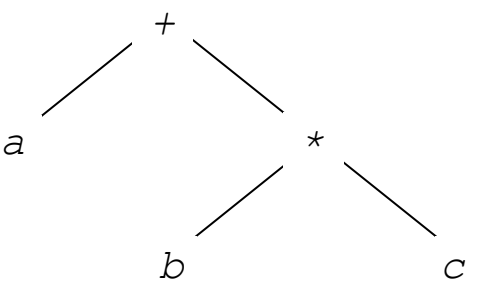
- Eine Variable ist lokal für eine Funktion, wenn sie innerhalb des Bindungsbereichs der Funktion deklariert wird.
- Ihre Gültigkeit beginnt ab der Zeile ihrer Deklaration bis zum Ende einer umgebenden Schleife bzw. der Funktion.
- Globale Variablen gelten im ganzen Programm, dass die Funktion eines Algorithmus ausführt und müssen nicht explizit als Eingabeparameter übergeben, jedoch als Teil der Spezifikation definiert werden.

Pseudocode [Kastens 2005]

Operatoren: Präzedenz, Assoziativität

Ein Operator mit höherer Präzedenz bindet seine Operanden stärker als ein Operator mit niedrigerer Präzedenz. Durch Klammerung lässt sich die Präzedenz in Termen vorschreiben. Beispiel:

```
a + b * c
```

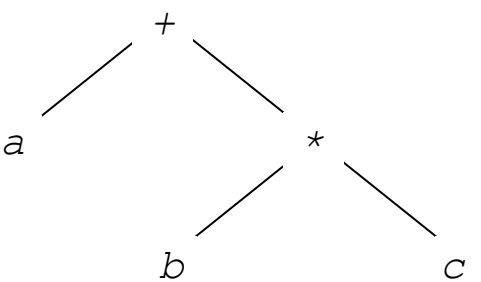


Pseudocode [Kastens 2005]

Operatoren: Präzedenz, Assoziativität

Ein Operator mit höherer Präzedenz bindet seine Operanden stärker als ein Operator mit niedrigerer Präzedenz. Durch Klammerung lässt sich die Präzedenz in Termen vorschreiben. Beispiel:

```
a + b * c
```



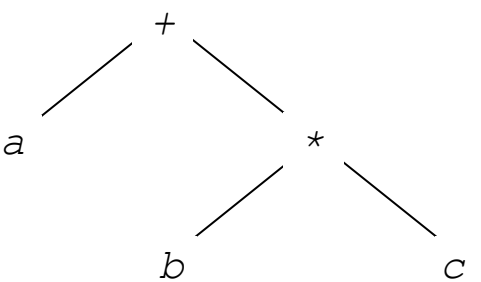
Ein Operator ist linksassoziativ (rechtsassoziativ), wenn beim Zusammentreffen von Operatoren gleicher Präzedenz der linke (rechte) Operator seine Operanden stärker bindet als der rechte (linke).

Pseudocode [Kastens 2005]

Operatoren: Präzedenz, Assoziativität

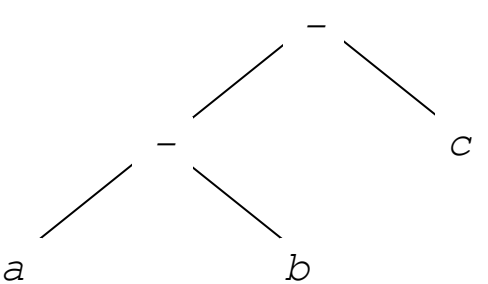
Ein Operator mit höherer Präzedenz bindet seine Operanden stärker als ein Operator mit niedrigerer Präzedenz. Durch Klammerung lässt sich die Präzedenz in Termen vorschreiben. Beispiel:

```
a + b * c
```



Ein Operator ist linksassoziativ (rechtsassoziativ), wenn beim Zusammentreffen von Operatoren gleicher Präzedenz der linke (rechte) Operator seine Operanden stärker bindet als der rechte (linke). Beispiel:

```
a - b - c
```



Pseudocode

Operatoren: Übersicht

Ausgewählte Operatoren für die Nutzung in Pseudocode:

Präzedenz	Stelligkeit	Assoziativität	Operatoren	Erklärung
2	2	rechts	$=$	Zuweisungsoperatoren
4	2	links	**OR**	logische Disjunktion
5	2	links	**AND**	logische Konjunktion
6	2	links	$\mid$	Bitoperator
7	2	links	$\hat{}$	Bitoperator
8	2	links	&	Bitoperator
9	2	links	$== \neq$	Gleichheit
10	2	links	$< \leq > \geq$	Ordnungsvergleich
11	2	links	<< >> >>>	shift-Operatoren
12	2	links	$+ -$	Konkatenation, Add., Subtr.
13	2	links	$* / \div$	Arithmetik
14	1		**NOT** $-$	Negation (logisch, arithm.)
15	1		() [] .	Aufruf, Index, Objektzugriff

Bemerkungen:

- Es handelt sich um einen angepassten Auszug der Operatoren, die in vielen Programmiersprachen vorhanden sind (siehe [Wikipedia]). Es gibt keine Beschränkungen, welche Operatoren in Pseudocode verwendet werden, solange ihre Semantik klar definiert ist.

Pseudocode

Datentypen: Primitive

`number`

- Keine Unterscheidung zwischen Ganzzahlen und Gleitpunktzahlen.
- Das Symbol ∞ hat einen Wert, der größer als die größte darstellbare Zahl ist.

`string`

- Zeichenkettenliterale mit einfachen oder doppelten Anführungszeichen.
- Konkatenation wie in Java: $s =$ `"Hello"` $+$ `"world!"`
- Zeichenkettenfunktionen werden in objektorientierter Notation verwendet. Beispiele: `s.length`, `s.indexOf(substr)`, `s.charAt(i)`.

`boolean`

- Literale: *True* und *False*

`nil`

- Der Wert *NIL* steht dafür, dass eine Variable keinen gültigen Wert hat.

Bemerkungen:

- „nil“ (eigentlich „nīl“) ist die kontrahierte Form von „nihil“, lateinisch für „nichts“.

Pseudocode

Datentypen: Objekte

Objekte bestehen aus Komponenten, die jeweils einen Bezeichner und einen Wert haben.

Objektkomponenten können Funktionen sein und heißen dann Methoden. Komponenten, die keine Methoden sind, heißen Eigenschaften oder Attribute.

Zugriff auf Objektkomponenten mittels Punktnotation:

- Attribut: *Objektbezeichner* . *Attributbezeichner*

  array.*length*; *car*.*brand*; *entry*.*key*

- Methode: *Objektbezeichner* . *Methodenbezeichner*(...)

  stack.*push*(24); *queue*.*first*(); *list*.*add*(42)

Objekte sind vornehmlich mathematische Objekte oder Datenstrukturen.

Pseudocode

Datentypen: Arrays

Ein Array ist eine Abbildung von Indizes auf Werte. Jedes Element eines Arrays bildet ein Paar bestehend aus numerischem Index und zugeordnetem Wert.

Deklaration und Typisierung:

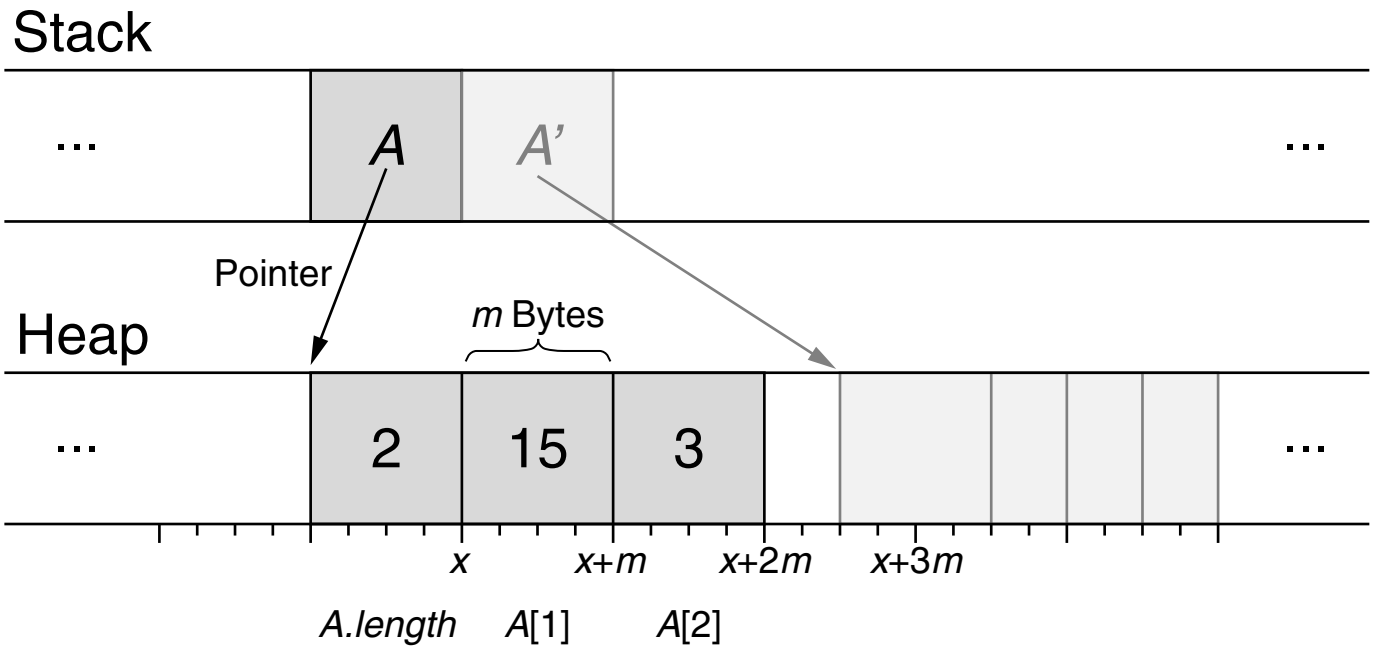
- Die Länge eines Arrays wird zur Initialisierungszeit definiert:
  $A = \mathit{Array}(n)$ oder `initialize array` A `of length` n
- Alle Werte eines gegebenen Arrays sind vom gleichen Typ.
- Nach der Initialisierung sind alle Werte 0, *False*, oder *NIL*.

Zugriffsoperatoren:

- Auslesen von Array-Elementen:
  $a_i = A[i]$ weist den Wert des i-ten Elements von A der Variable a_i zu.
- Zuweisen und Verändern von Array-Elementen:
  $A[i] = a$ weist dem i-ten Element von A den Wert a zu.

Pseudocode

Datentypen: Arrays



Speicherlayout:

- Ein Array belegt ein konsekutives Stück Speicher im Computer.
- Der Zugriff auf das i-te Element erfolgt über einfache Arithmetik:
  $A[i] \rightarrow \mathit{Heap}[x..x + i \cdot m]$, wobei x der Anfang der Speicheradresse des ersten Elements ist.
- Der Zugriff außerhalb von $1 \leq i \leq A.\mathit{length}$ führt zu einer Zugriffsverletzung.

Bemerkungen:

- Stack und Heap sind Speicherbereiche eines laufenden Prozesses im RAM eines Computers. Auf dem Stack werden Aufrufe von Funktionen, ihre Variablen primitiven Datentyps, sowie Pointer zu Objekten auf dem Heap in der Reihenfolge aufgeführt, in der sie benötigt wurden. Ist eine Funktion beendet, wird ihr Stack-Speicher wieder freigegeben. [Wikipedia]

  Auf dem Heap werden Objekte, deren Größe dynamisch zur Laufzeit bestimmt wird, abgelegt. Hier kann es mit der Zeit zu Fragmentierung und ungenutzten Lücken kommen. Objekte, die auf dem Heap abgelegt sind, müssen vom Programmierer explizit vor Funktionsende oder in einem anderen Teil des Programms wieder freigegeben werden, da sie nicht automatisch gelöscht werden. Einige Laufzeitumgebungen kontrollieren regelmäßig, ob es belegte Bereiche gibt, die nicht mehr benötigt werden, und geben den Speicher gegebenenfalls wieder frei (Garbage Collection). [Wikipedia]

  Während der Stack-Speicher analog zur Datenstruktur Stack funktioniert, hat der Heap-Speicher nichts mit der gleichnamigen Datenstruktur gemein.

- Es gibt drei Arten, die Elemente eines Arrays zu indizieren:
  - Zero-based Indexing: Das erste Element eines Arrays erhält den Index 0.
  - One-based Indexing: Das erste Element eines Arrays erhält den Index 1.
  - n-based Indexing: Der Index des ersten Elements kann frei gewählt werden.

  Zero-based Indexing ist weit verbreitet in Programmiersprachen. One-based indexing wird oft in der Literatur verwendet. Es gibt Situationen in denen die jeweils eine Art der Indizierung „natürlicher“ erscheint als die andere. Letztlich muss man sich immer darüber im Klaren sein, welche der beiden Arten im aktuellen Kontext angewendet wird, da es sonst leicht zu sogenannten „off-by-one“-Fehlern kommt. [Wikipedia]

Pseudocode

Datentypen: Arrays

Die Größe eines Arrays kann nach seiner Initialisierung nicht verändert werden.
Das Hinzufügen oder Löschen von Elementen erfordert das Kopieren des Arrays.

Pseudocode

Datentypen: Arrays

Die Größe eines Arrays kann nach seiner Initialisierung nicht verändert werden.
Das Hinzufügen oder Löschen von Elementen erfordert das Kopieren des Arrays.

Algorithmus: Array Insert.

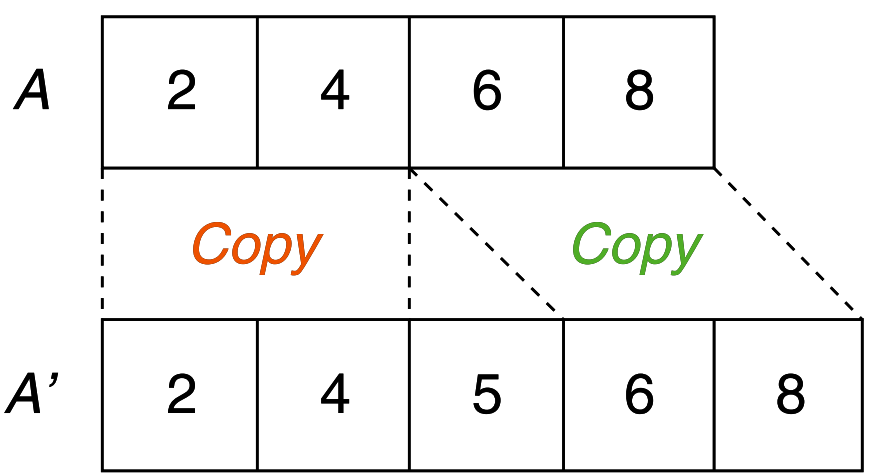
Eingabe: A. Array von n Werten.
v. Einzufügender Wert.
i. Index in dem v einzufügen ist.

Ausgabe: Kopie von A mit $A[i] = v$.

ArrayInsert(A, v, i)

```
1. IF i < 1 THEN error()
2. A' = Array(max(n + 1, i))
3. IF i > 1 THEN
     i' = min(n, i − 1)
     Copy(A, 1, i', A', 1, i')
   ENDIF
4. A'[i] = v
5. IF i ≤ n THEN
     Copy(A, i, n, A', i + 1, n + 1)
   ENDIF
6. return(A')
```

ArrayInsert$(A, 5, 3)$



Pseudocode

Datentypen: Arrays

Die Größe eines Arrays kann nach seiner Initialisierung nicht verändert werden.
Das Hinzufügen oder Löschen von Elementen erfordert das Kopieren des Arrays.

Algorithmus: Array Insert.

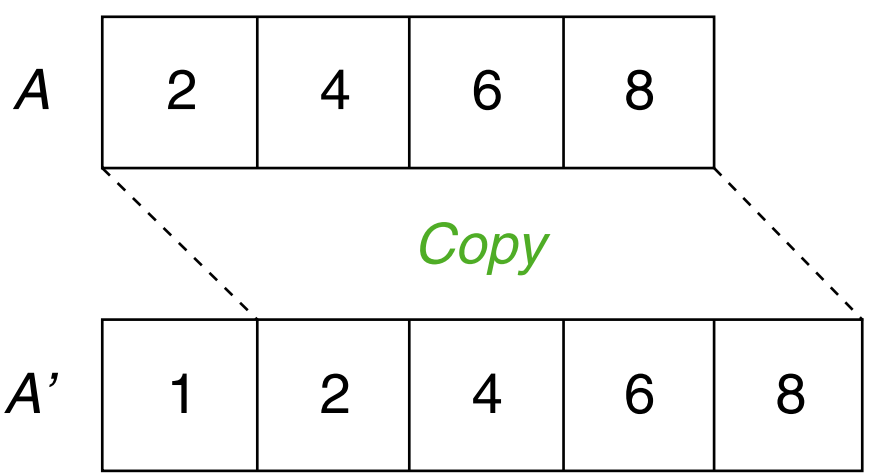
Eingabe: A. Array von n Werten.
v. Einzufügender Wert.
i. Index in dem v einzufügen ist.

Ausgabe: Kopie von A mit $A[i] = v$.

ArrayInsert(A, v, i)

```
1. IF i < 1 THEN error()
2. A' = Array(max(n + 1, i))
3. IF i > 1 THEN
     i' = min(n, i − 1)
     Copy(A, 1, i', A', 1, i')
   ENDIF
4. A'[i] = v
5. IF i ≤ n THEN
     Copy(A, i, n, A', i + 1, n + 1)
   ENDIF
6. return(A')
```

ArrayInsert$(A, 1, 1)$



Pseudocode

Datentypen: Arrays

Die Größe eines Arrays kann nach seiner Initialisierung nicht verändert werden.
Das Hinzufügen oder Löschen von Elementen erfordert das Kopieren des Arrays.

Algorithmus: Array Insert.

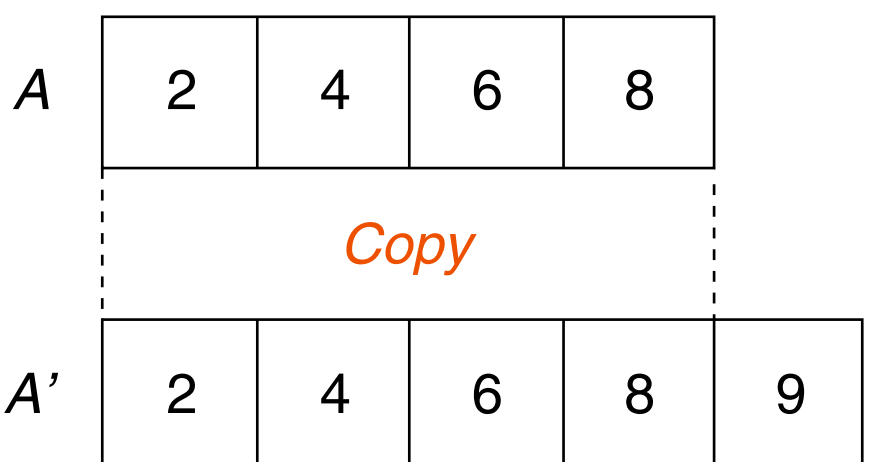
Eingabe: A. Array von n Werten.
v. Einzufügender Wert.
i. Index in dem v einzufügen ist.

Ausgabe: Kopie von A mit $A[i] = v$.

ArrayInsert(A, v, i)

```
1. IF i < 1 THEN error()
2. A' = Array(max(n + 1, i))
3. IF i > 1 THEN
     i' = min(n, i − 1)
     Copy(A, 1, i', A', 1, i')
   ENDIF
4. A'[i] = v
5. IF i ≤ n THEN
     Copy(A, i, n, A', i + 1, n + 1)
   ENDIF
6. return(A')
```

ArrayInsert$(A, 9, 5)$



Pseudocode

Datentypen: Arrays

Die Größe eines Arrays kann nach seiner Initialisierung nicht verändert werden.
Das Hinzufügen oder Löschen von Elementen erfordert das Kopieren des Arrays.

Algorithmus: Array Delete.

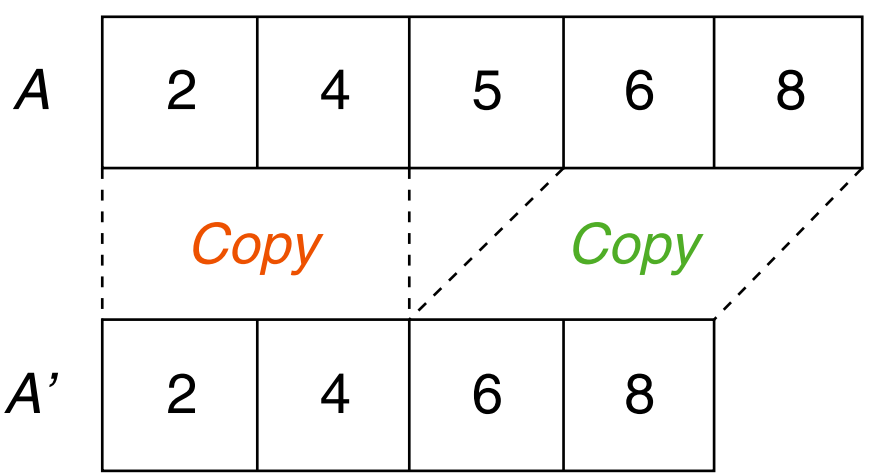
Eingabe: A. Array von n Werten.
i. Index des zu löschenden Werts.

Ausgabe: Kopie von A ohne den Wert in $A[i]$.

ArrayDelete(A, i)

```
1. IF i < 1 OR i > n THEN error()
2. A' = Array(n − 1)
3. IF i > 1 THEN
     Copy(A, 1, i − 1, A', 1, i − 1)
   ENDIF
4. IF i < n THEN
     Copy(A, i + 1, n, A', i, n − 1)
   ENDIF
5. return(A')
```

ArrayDelete$(A, 3)$



Pseudocode

Datentypen: Arrays

Die Größe eines Arrays kann nach seiner Initialisierung nicht verändert werden.
Das Hinzufügen oder Löschen von Elementen erfordert das Kopieren des Arrays.

Algorithmus: Array Delete.

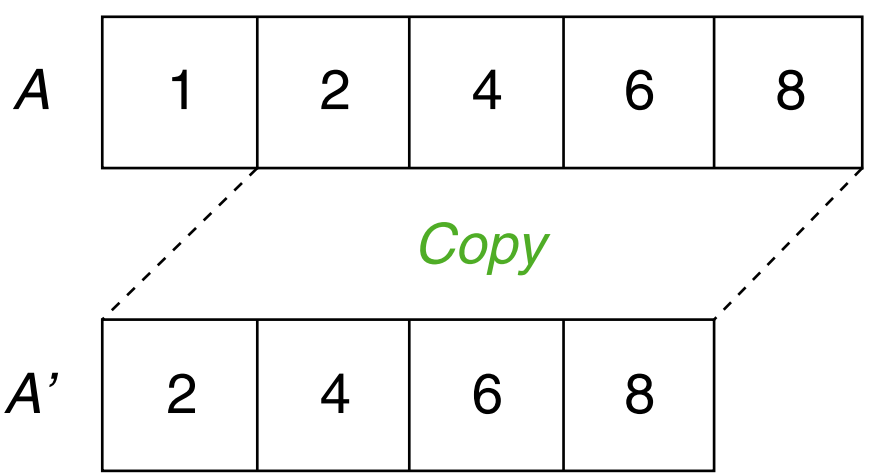
Eingabe: A. Array von n Werten.
i. Index des zu löschenden Werts.

Ausgabe: Kopie von A ohne den Wert in $A[i]$.

ArrayDelete(A, i)

```
1. IF i < 1 OR i > n THEN error()
2. A' = Array(n − 1)
3. IF i > 1 THEN
     Copy(A, 1, i − 1, A', 1, i − 1)
   ENDIF
4. IF i < n THEN
     Copy(A, i + 1, n, A', i, n − 1)
   ENDIF
5. return(A')
```

ArrayInsert$(A, 1)$



Pseudocode

Datentypen: Arrays

Die Größe eines Arrays kann nach seiner Initialisierung nicht verändert werden.
Das Hinzufügen oder Löschen von Elementen erfordert das Kopieren des Arrays.

Algorithmus: Array Delete.

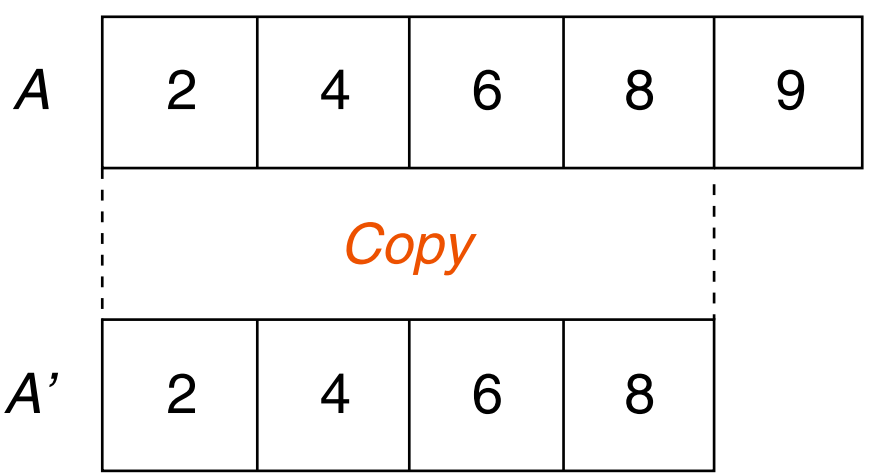
Eingabe: A. Array von n Werten.
i. Index des zu löschenden Werts.

Ausgabe: Kopie von A ohne den Wert in $A[i]$.

ArrayDelete(A, i)

```
1. IF i < 1 OR i > n THEN error()
2. A' = Array(n − 1)
3. IF i > 1 THEN
     Copy(A, 1, i − 1, A', 1, i − 1)
   ENDIF
4. IF i < n THEN
     Copy(A, i + 1, n, A', i, n − 1)
   ENDIF
5. return(A')
```

ArrayInsert$(A, 5)$



Pseudocode

Kontrollstrukturen

- Anweisungsfolge:
- Bedingte Anweisung:
- Return- und Error-Anweisung:
- **`WHILE`**-Schleife:
- **`FOR`**-Schleife:

Pseudocode

Kontrollstrukturen

- Anweisungsfolge:

  $i = 1;\ \ n = 0;\ \ a = i$

  Eine Anweisungsfolge innerhalb einer bedingten Anweisung oder einer Schleife definiert einen *Scope*: Variablen sind nur bis zum ihrem Ende gültig.

- Bedingte Anweisung:

  IF $i < 1$ **THEN** $a = 1$ **ELSEIF** $i > 1$ **THEN** $a = -1$ **ELSE** $a = 0$ **ENDIF**

- Return- und Error-Anweisung:

- **WHILE**-Schleife:

- **FOR**-Schleife:

Pseudocode

Kontrollstrukturen

- Anweisungsfolge:

  $i = 1$`;` $n = 0$`;` $a = i$

  Eine Anweisungsfolge innerhalb einer bedingten Anweisung oder einer Schleife definiert einen *Scope*: Variablen sind nur bis zum ihrem Ende gültig.

- Bedingte Anweisung:

  `IF` $i < 1$ **`THEN`** $a = 1$ **`ELSEIF`** $i > 1$ **`THEN`** $a = -1$ **`ELSE`** $a = 0$ **`ENDIF`**

- Return- und Error-Anweisung:

  return$()$ *return*(n) *return*(n, A) *error*`("message")` *error*$()$

- **`WHILE`**-Schleife:

- **`FOR`**-Schleife:

Pseudocode

Kontrollstrukturen

- Anweisungsfolge:

  $i = 1$; $n = 0$; $a = i$

  Eine Anweisungsfolge innerhalb einer bedingten Anweisung oder einer Schleife definiert einen *Scope*: Variablen sind nur bis zum ihrem Ende gültig.

- Bedingte Anweisung:

  IF $i < 1$ **THEN** $a = 1$ **ELSEIF** $i > 1$ **THEN** $a = -1$ **ELSE** $a = 0$ **ENDIF**

- Return- und Error-Anweisung:

  return() *return*(n) *return*(n, A) *error*("message") *error*()

- **WHILE**-Schleife:

  $i = 24$; **WHILE** $i > 0$ **DO** $i = i - 1$ **ENDDO**

  $i = 0$; **DO** $i = i + 1$ **WHILE** $i < 42$;

- **FOR**-Schleife: Gegeben Array A mit $n = A.\mathit{length}$

  FOR $i = 1$ **TO** n **DO** $A[i] = i * n$ **ENDDO**

  FOR $i = n$ **DOWNTO** 1 **BY** 2 **DO** $A[i] = A[i] \cdot A[i]$ **ENDDO**

  FOREACH $i \in [1, n]$ **DO** $A[i] = i * n$ **ENDDO**

Pseudocode

Parameterübergabe beim Funktionsaufruf

- Wertparameter (*call by value*):

  Der formale Parameter (in der Funktionssignatur) ist eine Variable, die mit dem Wert des aktuellen Parameters (in einem Funktionsaufruf) initialisiert wird.

  Änderungen der Variable innerhalb der Funktion beeinflussen den Wert des aktuellen Parameters nicht.

  Primitive Datentypen werden als Wertparameter übergeben.

- Referenzparameter (*call by reference*):

  Der formale Parameter (in der Funktionssignatur) ist eine Referenz auf das Objekt, auf das der aktuelle Parameter (in einem Funktionsaufruf) verweist.

  Änderungen am Objekt innerhalb der Funktion bleiben nach Ende der Funktion erhalten und verändern den aktuellen Parameter, da beide auf dasselbe Objekte im Speicher verweisen.

  Objekte, Arrays und Datenstrukturen werden als Referenzparameter übergeben.

Pseudocode

Parameterübergabe beim Funktionsaufruf

- Wertparameter (*call by value*):

  Der formale Parameter (in der Funktionssignatur) ist eine Variable, die mit dem Wert des aktuellen Parameters (in einem Funktionsaufruf) initialisiert wird.

  Änderungen der Variable innerhalb der Funktion beeinflussen den Wert des aktuellen Parameters nicht.

  Primitive Datentypen werden als Wertparameter übergeben.

- Referenzparameter (*call by reference*):

  Der formale Parameter (in der Funktionssignatur) ist eine Referenz auf das Objekt, auf das der aktuelle Parameter (in einem Funktionsaufruf) verweist.

  Änderungen am Objekt innerhalb der Funktion bleiben nach Ende der Funktion erhalten und verändern den aktuellen Parameter, da beide auf dasselbe Objekte im Speicher verweisen.

  Objekte, Arrays und Datenstrukturen werden als Referenzparameter übergeben.

Pseudocode

Parameterübergabe beim Funktionsaufruf

- Wertparameter (*call by value*):

  Der formale Parameter (in der Funktionssignatur) ist eine Variable, die mit dem Wert des aktuellen Parameters (in einem Funktionsaufruf) initialisiert wird.

  Änderungen der Variable innerhalb der Funktion beeinflussen den Wert des aktuellen Parameters nicht.

  Primitive Datentypen werden als Wertparameter übergeben.

- Referenzparameter (*call by reference*):

  Der formale Parameter (in der Funktionssignatur) ist eine Referenz auf das Objekt, auf das der aktuelle Parameter (in einem Funktionsaufruf) verweist.

  Änderungen am Objekt innerhalb der Funktion bleiben nach Ende der Funktion erhalten und verändern den aktuellen Parameter, da beide auf dasselbe Objekte im Speicher verweisen.

  Objekte, Arrays und Datenstrukturen werden als Referenzparameter übergeben.