Programmierung und Algorithmen

Kapitel 2
Algorithmische Grundkonzepte

Überblick

Intuitiver Algorithmenbegriff

Sprachen und Grammatiken

Elementare Datentypen

Terme

Intuitiver Algorithmus-Begriff

Ein Algorithmus ist eine präzise (in einer festgelegten Sprache abgefasste) endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte.

Bekannte Algorithmen

1. Addition zweier positiver Dezimalzahlen (mit Überträgen)

$$\begin{array}{rr} & 3\ 3 \\ + & 4\ 8 \\ & {\scriptstyle 1} \\ \hline & 8\ 1 \end{array}$$

2. Test, ob eine gegebene natürliche Zahl eine Primzahl ist
3. Sortieren einer unsortierten Kartei (etwa lexikographisch)
4. Berechnung der Zahl $e = 2.7182\ldots$

Eigenschaften: Terminierung

Ein Algorithmus heißt terminierend, wenn er (bei jeder erlaubten Eingabe von Parameterwerten) nach endlich vielen Schritten abbricht.

- **Frage:** Terminieren die Algorithmen von der vorigen Folie?
- Beispiel

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Eigenschaften: Determinismus

- Legt „Wahlfreiheit“ bei der Ausführung fest
- Formen
  - **deterministischer Ablauf**: eindeutige Vorgabe der Folge der auszuführenden Schritte
  - **determiniertes Ergebnis**: eindeutiges Ergebnis bei vorgegebenen Parameterwerten
- Drei Klassen bezüglich des Ablaufs
  - Ein **deterministischer Algorithmus** hat einen deterministischen Ablauf
  - Ein **randomisierter Algorithmus** hat einen deterministischen Ablauf und eine zusätzliche, **zufällige** Eingabe
  - Ein **nichtdeterministischer Algorithmus** (formales Modell!) hat einen nichtdeterministischen Ablauf
    - Bemerkung: dieses Modell wird von korrekt funktionierenden Computern üblicherweise nicht unterstützt

Randomisierte vs. Nicht-Deterministische Algorithmen

- Beispiel #1 (randomisiert)

  Eingabe: Zahl x
  zusätzliche, zufällige Eingabe: Zahl r (nicht vom Nutzer gewählt)
  1. Addiere r zu x und multipliziere Ergebnis mit 3
  2. Gib das Ergebnis aus

  - Hier ist das Ergebnis von r abhängig – das muss nicht so sein!
  - Es können auch determinierte Ergebnisse erzielt werden – ggf. sind dann andere Dinge wie Laufzeit o.ä. zufällig (siehe weitere Vorlesungen)

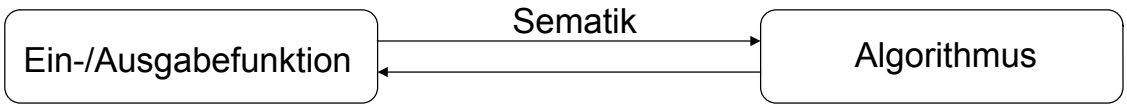
- Beispiel #2 (nicht-deterministisch)

  Eingabe: Zahl x
  1. Entweder: Addiere das Dreifache von *x* zu *x* und teile das Ergebnis durch x: $(3x + x)/x$
     Oder: Subtrahiere 4 von *x* und subtrahiere das Ergebnis von *x*: $x - (x - 4)$
  2. Gib das Ergebnis aus

Funktionen zu Algorithmen

- Wichtige Klasse: deterministische, terminierende Algorithmen
  - definieren Ein-/Ausgabefunktion

$$f : \text{Eingabewerte} \rightarrow \text{Ausgabewerte}$$



ausführbare Beschreibung

Funktionen zu Algorithmen /2

1. Addition zweier positiver Dezimalzahlen (mit Überträgen)

   $f : \mathbb{Q}^+ \times \mathbb{Q}^+ \rightarrow \mathbb{Q}^+$ mit $f(p,q) = p + q$

   $\mathbb{Q}^+$ seien die positiven Rationalzahlen

2. Test, ob eine gegebene natürliche Zahl eine Primzahl ist

$$f : \mathbb{N} \rightarrow \{\texttt{ja}, \texttt{nein}\} \text{ mit } f(n) = \begin{cases} \texttt{ja} & \text{falls } n \text{ Primzahl} \\ \texttt{nein} & \text{sonst} \end{cases}$$

Funktionen zu Algorithmen /3

3. Sortieren einer unsortierten Kartei (etwa lexikographisch)
   K Menge von Karteikarten
   S_K Menge von sortierten Karteien über K
   US_K Menge von unsortierten Karteien über K

$$f : US_K \rightarrow S_K$$

4. Berechnung der Stellen der Zahl e = 2.7182 . . .
   nicht terminierend!

Bausteine für Algorithmen

- Elementare Operationen
- Sequenzielle Ausführung (ein Prozessor!)
- Parallele Ausführung (mehrere Prozessoren!)
- Bedingte Ausführung
- Schleife
- Unter„programm“
- Rekursion: Anwendung des selben Prinzips auf kleinere Teilprobleme

Beispiel: Anweisungen in Kochbüchern

Pseudocode-Notation

- Code = (lauffähiges) Programm in einer Programmiersprache
- Pseudocode = (nicht lauffähige) Algorithmenbeschreibung in Anlehnung an Programmiersprachen
  - Schlüsselworte aus Programmiersprachen (eingedeutscht oder englisch)
  - Anweisungen in natürlicher Sprache
  - Wenn passend: mathematische Notationen, Programmfragmente

Sequenz

(1) Koche Wasser
(2) Gib Kaffeepulver in Tasse
(3) Fülle Wasser in Tasse

Sequenz und Verfeinerung

(2) Gib Kaffeepulver in Tasse

verfeinert zu

(2.1) Öffne Kaffeeglas
(2.2) Entnehme Löffel von Kaffee
(2.3) Kippe Löffel in Tasse
(2.4) Schließe Kaffeeglas

Entwurfsprinzip der schrittweisen Verfeinerung

Sequenzoperator

- Sequenzoperator: **;**

> Koche Wasser;
> Gib Kaffeepulver in Tasse;
> Fülle Wasser in Tasse

- Erspart die Durchnummerierung

Auswahl / Selektion

> **`falls`** Bedingung
> **`dann`** Schritt

bzw.

> **`falls`** Bedingung
> **`dann`** Schritt *a*
> **`sonst`** Schritt *b*

Schachtelung der Auswahl

falls Ampel ausgefallen
    dann fahre vorsichtig weiter
    sonst falls Ampel rot oder gelb
        dann stoppe
        sonst fahre weiter

Beispiel Schleife

/ nächste Primzahl */*
wiederhole
    Addiere 1;
    Teste auf Primzahleigenschaft
bis Zahl Primzahl ist;
gib Zahl aus

Schleife mit **solange**

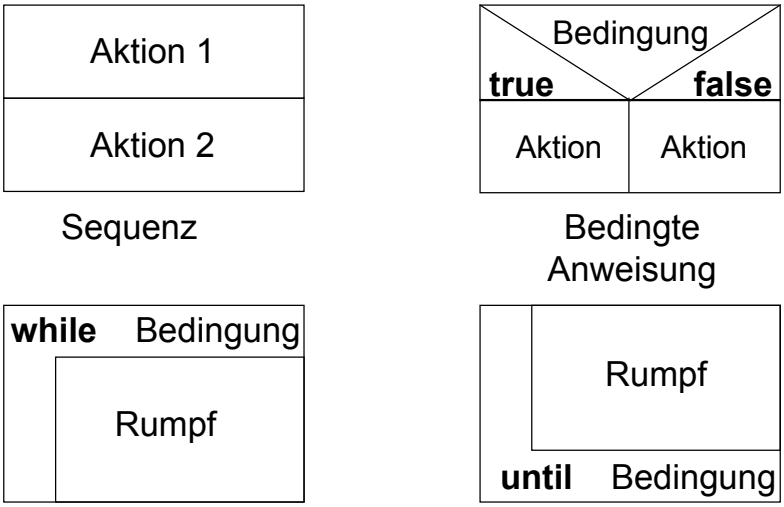
/ Bestimmung der größten Zahl einer Liste */*
Setze erste Zahl als bislang größte Zahl
`solange` Liste nicht erschöpft
`führe aus`
    Lies nächste Zahl der Liste
    `falls` diese Zahl > bislang größte Zahl
        `dann` setze diese Zahl als bislang größte Zahl;
gib bislang größte Zahl aus

Schleifen in Programmiersprachen

`wiederhole ... bis`	**`repeat ... until ...`**
	`do ... while ...`
`solange ... führe aus`	**`while ... do ...`**
	`while (...) ...`
`wiederhole für`	**`for each ... do ...`**
	`for ... do ...`
	`for (...) ...`

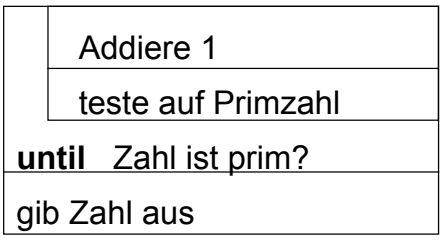
Struktogramme

- Graphische Notation für Sequenz, Bedingung, Schleife



Struktogramme: Beispiel

- Bestimmung der nächstgrößeren Primzahl



Rekursion

- Grundlegendes Prinzip der Informatik

  Die Rekursion bedeutet die Anwendung des selben Prinzips auf kleinere oder einfachere Teilprobleme, bis diese Teilprobleme so klein sind, dass sie direkt gelöst werden können.

- Definition:

  Als Rekursion (lat. recurrere „zurücklaufen“) bezeichnet man die Technik, eine Funktion durch sich selbst zu definieren (rekursive Definition). Wenn man mehrere Funktionen durch wechselseitige Verwendung voneinander definiert, spricht man von wechselseitiger Rekursion.

Rekursion

- *Um Rekursion zu verstehen, muss man erst einmal Rekursion verstehen.*

  bzw.

  Kürzeste Definition für Rekursion: siehe Rekursion.

- Bemerkungen:
  - Nicht jede rekursive Definition ist eine Definition im eigentlichen Sinn, denn die zu definierende Funktion braucht nicht wohldefiniert zu sein.
  - Jeder Aufruf der rekursiven Funktion muss sich durch Entfalten der rekursiven Definition in endlich vielen Schritten auflösen lassen. Umgangssprachlich sagt man, sie darf nicht in einen infiniten Regress („Endlosschleife“) geraten.
- Später noch ausführlicher behandelt, hier nur motivierendes Beispiel:
  - *Türme von Hanoi*

Die Aufgabe der Türme von Hanoi

- Aufgabenbeschreibung:
  - Zu jedem Zeitpunkt können Türme von Scheiben unterschiedlichen Umfangs auf drei Plätzen stehen:
    - Der ursprüngliche Standort wird als Quelle bezeichnet
    - Ein Turm der Höhe n (z.B. n=64), der zu Anfang bei der Quelle steht, soll zu einem Zielstandort (Senke) bewegt werden
    - Es steht ein dritter Standort, der sogenannte Arbeitsbereich zur Verfügung
  - Nur die jeweils oberste Scheibe eines Turms darf einzeln bewegt werden
  - Dabei darf nie eine größere auf einer kleineren Scheibe zu liegen kommen
- In welcher Reihenfolge müssen die Scheiben bewegt werden?
  - Man probiere es mit einem kleinen Beispiel, z.B. vier unterschiedlich große Geldstücke oder Bücher

Eine Strategie zur Lösung (plus Pseudocode)

- Idee: Angenommen, ich kann einen Turm der Höhe n-1 bewegen, dann kann ich auch einen Turm der Höhe n bewegen

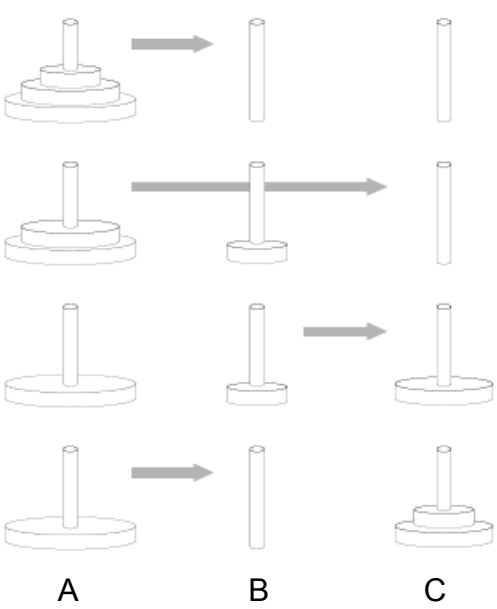
```
Modul Turmbewegung(n, Quelle, Senke, AB)
/* Bewegt einen Turm der Höhe n
   von Quelle nach Senke unter Benutzung
   des Arbeitsbereichs AB */
   falls   n = 1
   dann    bewege Scheibe von Quelle zur Senke
   sonst   Turmbewegung(n-1, Quelle, AB, Senke)
           bewege 1 Scheibe von Quelle zur Senke
           Turmbewegung(n-1, AB, Senke, Quelle)
```

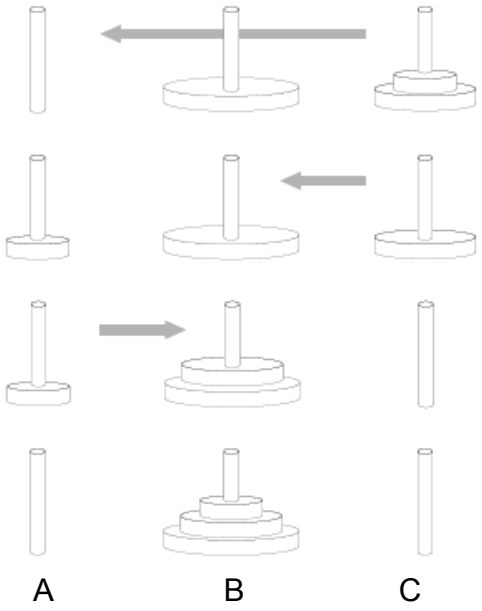
Türme von Hanoi: Beispielablauf (für n=3)

```
Turm(3,A,B,C)
    Turm(2,A,C,B)
        Turm(1,A,B,C)
            bewege A → B
        bewege A → C
        Turm(1,B,C,A)
            bewege B → C
    bewege A → B
    Turm(2,C,B,A)
        Turm(1,C,A,B)
            bewege C → A
        bewege C → B
        Turm(1,A,B,C)
            bewege A → B
```

Türme von Hanoi: Beispielablauf grafisch dargestellt

Ablaufbeispiel:





Sprachen und Grammatiken

- **Syntax**: Formale Regeln, welche Sätze gebildet werden können:
  - „Der Elefant aß die Erdnuss.“ (syntaktisch korrekt)
  - „Der Elefant aß Erdnuss die.“ (syntaktisch falsch)

- **Semantik**: (Formale) Regeln, welche Sätze eine *Bedeutung* haben:
  - „Der Elefant aß die Erdnuss.“ (semantisch korrekt, „sinnhaft“)
  - „Die Erdnuss aß den Elefant.“ (semantisch falsch, „sinnlos“)

Begriffsbildung

- Grammatik: Regelwerk zur Beschreibung der Syntax
- Produktionsregel: Regel einer Grammatik zum Bilden von Sätzen, z. B.

  Satz ↦ Subjekt Prädikat Objekt.

- Generierte Sprache:
  - Alle durch Anwendungen der Regeln einer Sprache erzeugbaren Sätze

Reguläre Ausdrücke

- Einfache Konstrukte, um Konstruktionsregeln für Zeichenketten festzulegen:
  - „Worte“ *a*, *b*, etc.
  - Sequenz: *pq*
  - Auswahl: *p* + *q*
  - Iteration: *p**

Reguläre Ausdrücke: Beispiele

- Mit *L* beginnende Binärzahlen über *L* und *O*:

$$L(L + O)^*$$

- Bezeichner einer Programmiersprache, die mit einem Buchstaben anfangen müssen:

$$(a + b + ... + z)(a + b + ... + z + 0 + 1 + ... + 9)^*$$

Backus-Naur-Form

- Festlegung der Syntax von Kunstsprachen:
  - Ersetzungsregeln der Form

```
linkeSeite ::= rechteSeite
```

- `linkeSeite` ist Name des zu definierenden Konzepts
- `rechteSeite` gibt Definition in Form einer Liste von Sequenzen aus Konstanten und anderen Konzepte (evtl. einschließlich dem zu definierenden!).
  Listenelemente sind dabei durch | getrennt.
- Sprechweise:
  - Definierte Konzepte: Nichtterminalsymbole
  - Konstanten: Terminalsymbole

Backus-Naur-Form: Beispiele /1

- `Bezeichner`

```
⟨Ziffer⟩       ::=  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
⟨Buchstabe⟩    ::=  a | b | c | . . . | z
⟨Zeichenkette⟩ ::=  ⟨Buchstabe⟩ |
                    ⟨Ziffer⟩ |
                    ⟨Buchstabe⟩ ⟨Zeichenkette⟩ |
                    ⟨Ziffer⟩ ⟨Zeichenkette⟩
⟨Bezeichner⟩   ::=  ⟨Buchstabe⟩ |
                    ⟨Buchstabe⟩ ⟨ Zeichenkette⟩
```

Backus-Naur-Form: Beispiele /2

- Syntax für Pseudo-Code-Algorithmen

```
⟨atom⟩       ::= 'addiere 1 zu x' | ...
⟨bedingung⟩  ::= 'x=0' | ...
⟨sequenz⟩    ::= ⟨block⟩  ;  ⟨block⟩
⟨auswahl⟩    ::= 'falls' ⟨bedingung⟩ 'dann' ⟨block⟩ |
                 'falls' ⟨bedingung⟩ 'dann' ⟨block⟩
                                    'sonst' ⟨block⟩
⟨schleife⟩   ::= 'wiederhole' ⟨block⟩
                              'bis' ⟨bedingung⟩ |
                 'solange' ⟨bedingung⟩
                           'führe aus' ⟨block⟩
⟨block⟩      ::= ⟨atom⟩     | ⟨sequenz⟩  |
                 ⟨auswahl⟩ | ⟨schleife⟩
```

Backus-Naur-Form: Beispiele /3

- Achtung!
  - Die vorige Folie zeigt ein Beispiel einer BNF zur Beschreibung von Pseudo-Code-Algorithmen
  - D.h. „falls“, „solange“, etc. sind darin Terminale (daher auch in '' gesetzt) und gehören nicht zu den erlaubten Mitteln einer BNF
  - "..." ist „Meta-Zeichen“ und keine gültige BNF-Notation! Es soll damit im Beispiel ausgedrückt werden, dass weitere Definitionen alternativer Anweisungen etc. definiert werden können.

Backus-Naur-Form: Beispiele /4

- BNF einer BNF – „Meta-Beschreibung“! :o)
  - Inhalte von Anführungszeichen (z.B. ‘ | ‘) sind als Terminale zu verstehen

```
⟨BNF⟩           ::= ⟨Regel⟩ | ⟨BNF⟩ ⟨NeueZeile⟩ ⟨Regel⟩
⟨Regel⟩         ::= ⟨NichtTerminal⟩  ' ::= ' ⟨Liste⟩
⟨Liste⟩         ::= ⟨NichtTerminal⟩ | ⟨Terminal⟩ |
                    ⟨Liste⟩⟨Liste⟩ |
                    ⟨Liste⟩ ' | ' ⟨Liste⟩
⟨NichtTerminal⟩ ::= '⟨' ⟨Zeichenkette⟩ '⟩'
⟨Terminal⟩      ::= ⟨Zeichenkette⟩    (Oder nach Bedarf)
⟨NeueZeile⟩     ::= '\n'                 (Zeilenumbruch)
⟨Zeichenkette⟩  Wie im ersten Beispiel
```


Datentypen als Algebren

- Daten: zu verarbeitende Informationseinheiten
- Datentypen: Zusammenfassung „gleichartiger“ Daten und auf ihnen erlaubter Operationen
- Algebra (mathematisch): (Werte-)Menge plus Operationen
- Daher: Algebren als Abstraktion von Datentypen

Elementare Datentypen

- Algebra = Wertemenge plus Operationen (mathematisches Konzept)
  - Beispiel: Natürliche Zahlen $\mathbb{N}$ mit +, - , * , ÷ , etc.
- Wertemengen werden in der Informatik als Sorten bezeichnet
- Operationen entsprechen Funktionen, werden durch *Algorithmen* realisiert
- *mehrsortige Algebra* = Algebra mit mehreren Sorten
  - Beispiel: Natürliche Zahlen plus Wahrheitswerte mit +, - , * , ÷ , auf Zahlen, ¬ , Λ , V , ... auf Wahrheitswerten, =, <, >, ≤ , ... als Verbindung
- Datentyp (im Gegensatz zum mathematischen Konzept der Algebra): Interpretierbare Werte mit ausführbaren Operationen.

Signaturen von Datentypen

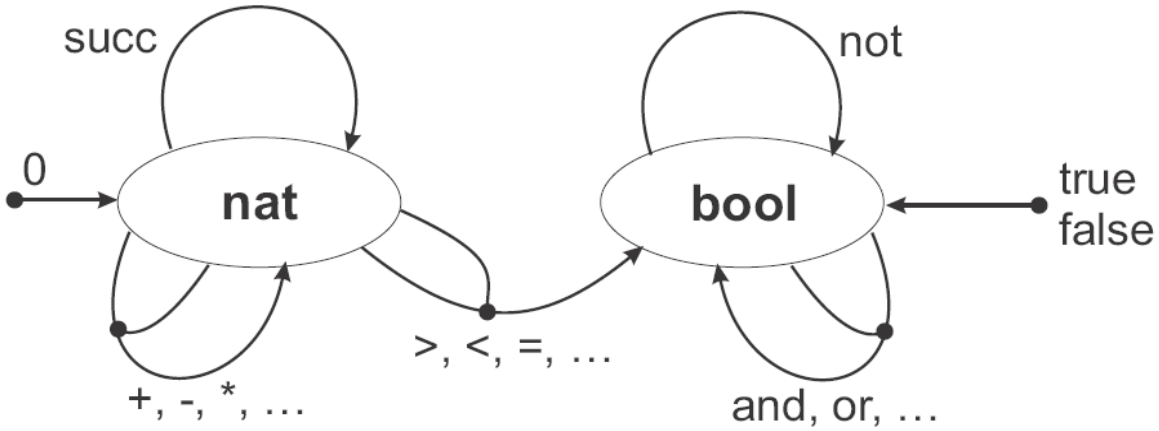
- Begriff der Signatur:
  - Formalisierung der Schnittstellenbeschreibung eines Datentyps
  - Angabe der (Bezeichner / Namen der) Sorten
  - Angabe der (Bezeichner / Namen der) Operationen
  - Stelligkeit der Operationen
  - Sorten der einzelnen Parameter
  - *Konstanten* als nullstellige Operationen

Signaturen von Datentypen: Beispiel

```
type nat
sorts nat, bool
functions
      0     :      → nat
      succ  : nat → nat
      +     : nat × nat → nat
      ≤     : nat × nat → bool
      ...
```

- Die Funktion 0 hat keinen Eingabetyp (ist also eine nullstellige Funktion), daher steht vor dem Pfeil nichts!

Signaturen von Datentypen: Beispiel /2



Der Datentyp **bool**

- **`boolean`**, auch **`bool`**: Datentyp der Wahrheitwerte
- Werte: { **`true`**, **`false`** }
- Operationen:
  - ¬ , auch **`not`**: logische Negation
  - Λ , auch **`and`**: logisches Und
  - V , auch **`or`**: logisches Oder
  - ⇒ : Implikation

Der Datentyp **bool** /2

- Operationen definiert durch *Wahrheitstafeln:*

	¬
`false`	`true`
`true`	`false`

Λ	`false`	`true`
`false`	`false`	`false`
`true`	`false`	`true`

V	`false`	`true`
`false`	`false`	`true`
`true`	`true`	`true`

- $p \Rightarrow q$ ist definiert als $\neg p \vee q$

Der Datentyp **integer**

- **`integer`**, auch **`int`**: Datentyp der ganzen Zahlen
- Werte: { . . . , - 2, - 1, 0, 1, 2, 3, 4, . . . }
- Operationen +, - , · , ÷ , **`mod`**, **`sign`**, >, <, ...
  - +: **`int`** × **`int`** → **`int`**
    Addition, 4 + 3 ↦ 7
  - **`mod`**: **`int`** × **`int`** → **`int`**
    Rest bei Division, 19 **`mod`** 7 ↦ 5
  - **`sign`**: **`int`** → {- 1, 0, 1 }
    Vorzeichen, **`sign`**(7) ↦ 1, **`sign`**(0) ↦ 0, **`sign`**(- 7) ↦ - 1,
  - >: **`int`** × **`int`** → **`bool`**
    Größerrelation: 4 > 3 ↦ **`true`**
  - Weiter übliche Operationen, insbesondere in folgenden Beispielen **`abs`** zur Berechnung des Absolutbetrags, **`odd`** und **`even`**

Zeichenketten

- **`char`**: Zeichen in Texten { *A*, .., *Z*, *a*, .. } mit Operation =
- **`string`**: Zeichenketten über **`char`**
  - Gleichheit: =
  - Konkatenation („Aneinanderhängen“): +
  - Selektion des *i* -ten Zeichens: s[*i*]
  - Länge: **`length`**
  - jeder Wert aus **`char`** wird als ein **`string`** der Länge 1 aufgefasst, wenn er wie folgt notiert wird: ’A’
  - **`empty`** als leerer **`string`**
  - weitere sinnvolle Operatoren, etwa **`substring`**

Felder

- **`array`**: „Felder“ mit Werten eines Datentyps als Eintrag, Ausdehnung fest vorgegeben
- Definition:

```
array 1..3 of int;
array 1..3, 1..3 of int;
```

- Operationen:
  - Gleichheit =
  - Selektion eines Elements: *A*[*n*] oder *A*[1, 2]
  - Konstruktion eines Feldes: (1, 2, 3) oder ((1, 2, 3), (3, 4, 5), (6, 7, 8))

int-Terme

1. Die **`int`**-Werte .., - 2, - 1, 0, 1, ... sind **`int`**-Terme.
2. Die **`bool`**-Werte **`true`** und **`false`** sind **`bool`**-Terme
3. Sind *t* , *u* **`int`**-Terme, so sind auch *(t* + *u)*, *(t* - *u)*, *(t* * *u)*, *(t* ÷ *u)*, **`sign`**(*t*), **`abs`**(*t*) **`int`**-Terme.
4. Sind *t* , *u* **`bool`**-Terme so sind auch (*t* ∧ *u*), (*t* ∨ *u*), (¬*t*) **`bool`**-Terme.
5. Sind t, u **`int`**-Terme, so sind (t = u), (t < u), (t > u), … **`bool`**-Terme.
6. Ist *b* ein **`bool`**-Term, und sind *t* , *u* **`int`**-Terme, so ist auch **`if`** *b* **`then`** *t* **`else`** *u* **`fi`** ein **`int`**-Term.
7. Nur die durch diese Regeln gebildeten Zeichenketten sind **`int`**-Terme.

Algemeiner: Bedingte Terme

- Bedingte Terme:

  if *b* **then** *t* **else** *u* **fi**

  - *b* boolscher Term, *t* und *u* zwei Terme gleicher Sorte

- Auswertung bedingter Terme (am Beispiel):

  if true then *t* **else** *u* **fi** ↦ *t*
  if false then *t* **else** *u* **fi** ↦ *u*
  if true then 3 **else** ⊥ **fi** ↦ 3
  if false then 3 **else** ⊥ **fi** ↦ ⊥

  - Im Gegensatz zu Operationen erzwingt ein Teilausdruck, der undefiniert ist, nicht automatisch die Undefiniertheit des Gesamtterms!

Algorithmus zur Termauswertung

- Auswertung von „innen nach außen“ (unter Beachtung von Klammern und Vorrangregeln), bei bedingten Termen wird zuerst die

  Bedingung ausgewertet.

  1+ **if true** ∨ ¬ **false then** 7 · 9 + 7 - 6 **else abs**(3 - 8) **fi**
  ↦ 1+ **if true** ∨ **true then** 7 · 9 + 7 - 6 **else abs** (3 - 8) **fi**
  ↦ 1+ **if true then** 7 · 9 + 7 - 6 **else abs**(3 - 8) **fi**
  ↦ 1 + 7 · 9 + 7 - 6
  ↦ 1 + 63 + 7 - 6
  ↦ 64 + 7 - 6
  ↦ 71 - 6
  ↦ 65

Zusammenfassung

- Algorithmenbegriff
- Terminierung, Determinismus, nichtdeterminierte vs. determinierte Algorithmen
- Algorithmenbausteine
- Algorithmenbeschreibung: Sprachen und Grammatiken
- Elementare Datentypen: **`bool`**, **`integer`**, Zeichenketten und Felder
- Terme: Bildung und Auswertung
- Literatur: Saake/Sattler: *Algorithmen und Datenstrukture*n, Kapitel 2