2 ST-Sprachzusammenfassung

2.1 Konventionelle Sprachelemente

Ein Programmcode ist ein **Wort** über ein **Alphabet.** Ein Alphabet ist eine Menge von **Zeichen** und ein Wort eine Folge von Alphabetzeichen. Wesentlich dabei ist, dass sich in einem Alphabet kein Wort aus verschiedenen Alphabetzeichenfolgen bilden lässt (Tarski-Kriterium). Zum Beispiel ist die Menge {I, II, III} kein Alphabet, seine Wörter wären nicht eindeutig in seine Alphabetzeichen zerlegbar. Die Menge {.I, .II, .III} hingegen ist ein Alphabet. Sie erfüllt das hinreichende Kriterium für Alphabete, das Endwortkriterium (Jedes echte Endstück eines Alphabetzeichen ist nicht selbst Alphabetzeichen, **Semiotik** - Codierungstheorie).

Nicht jedes Wort ist ein Programm. In jeder Programmiersprache wird festgelegt, welche Wörter **syntaktisch** korrekt sind und letztlich interpretiert werden können (Syntaxdiagramme, Backus-Notation sind üblich zur Sprachbeschreibung).

2.1.1 Das Alphabet

Der Zeichensatz von **ST** umfaßt **92 druckbare Zeichen**:

- 26 Großbuchstaben des englischen Alphabets **A . . Z**
- 26 Kleinbuchstaben des englischen Alphabets **a . . z**
- 10 Ziffern **0 . . 9**
- 30 Sonderzeichen **, + / \ * ~ < > = @ % | & ? !**
  [ ] { } ( ) $ # ^ ; : . ‘ ″ Leerzeichen

2.1.2 Der Kommentar *(comment)*

ST-Kommentare sind beliebige Zeichenfolgen des Alphabets und werden in Anführungsstriche geschrieben. Bei der Programmausführung werden Kommentare übergangen. Sie gehören keiner Klasse an und dienen ausschließlich der Erläuterung des Programmcodes. Sie sind oft der einzige Hinweis in den Methoden und Klassen über ihre Funktion:

″ *Kommentar* ″

Kommentare kann man in den unterschiedlichsten ST-Komponenten eintragen und dienen dem besseren Verständnis der bereits entwickelten Systems.

2.2 Objekte, Konstanten und Variablen

2.2.1 Objekte

Objekte sind die Größen, mit denen im Programm operiert wird. Sie müssen in der Maschine dargestellt und ihre **Methoden** als Programme installiert werden.
ST-Objekte werden über Zeiger angesprochen. Diese sind 32-Bit-Adressen (Segment und Offset) und verweisen auf den Speicherbereich der Objektdaten.

Für die Bezeichnung solcher Objekte, Objektklassen und ihrer Methoden in der Maschine werden **Namen** (identifier) benötigt.

2.2.2 Namen *(identifier)*

ST-Namen sind frei wählbare, beliebig lange Wörter aus den 52 Buchstaben und den 10 Ziffern des englischen Alphabets und beginnen mit einem Buchstaben. Alle Zeichen sind signifikant. Aus Gründen der leichteren Lesbarkeit ist es in OOP-Sprachen üblich, „sprechende Namen“ zu wählen, z. B. **ArrayForData**, **anInteger**, **anObject**.
Sonderzeichen und Umlaute sind grundsätzlich nicht erlaubt.

2.2.3 Konstanten *(literal)*

ST-Konstanten sind **nicht veränderliche, explizit angegebene Objekte** im Programm. Diese werden durch bloßes **Hinschreiben** definiert, gehören zu einer Klasse, die sich aus dem Wert der Konstanten ergibt.

- Klasse **Object:**
  - Klasse **Magnitude:**
    - Klasse **Character: Zeichenkonstanten** (ASCII-Zeichensatz)

      Beispiele: **$A $1 $$**

    - Klasse **Number: Zahlenkonstanten (**numerische Objekte)
      - Klasse **Integer:**
        - Klasse **SmallInteger: Ganze Zahlen** (32-Bit-Zahlen)

          Bereich: **[- 1073741824, 1073741823]** $[-2^{30}, 2^{30} -1]$

          Sind speziell abgespeichert und als einzige Daten nicht als Objekte implementiert. Sie stehen in einem speziellen Segment und werden über dem der Zahl entsprechenden 16-Bit-Offset erreicht, wobei das höherwertige Bit das Vorzeichenbit ist und eine zweites Bit zur Lokalisierung des entsprechenden Segments dient.

          Beispiele: **12 -123e3**
          8r25 (Zahlenbasis: 8)
          16r3FFFFFFF (betragsmäßig größte **SmallInteger**-Zahl)

        - Klasse **LargeInteger: Ganze Zahlen**

          Es gibt **keine** Bereichseinschränkungen, die Größe ist nur vom Hauptspeicher abhängig. Bei sehr großen Zahlen verlangsamt sich ihre Verarbeitung wesentlich oder der Rechner stürzt ab.

          Beispiel: **16r40000000** (betragsmäßig kleinste **LargeInteger**-Zahl)

      - Klasse **Float:** (8-Byte-IEEE-Format)

        Bereich: **[- 10^{-307}, - 10^{-307}]** $\cup$ **{0}** $\cup$ **[10^{-307}, 10^{307}]**

        Dichte: ca. 18 Stellen (8 Nachkommastellen werden angezeigt)

        Bei zu großen Zahlen stürzt der Rechner ab. In einigen Systemen wird zwischen den Klassen **Float** und **Double** unterschieden, hier nicht.

        Beispiele: **12.3 -10.3e2 1.0e307** (**.1 1.** nicht erlaubt!)

- Klasse **Fraction: Rationale Zahlen** (Brüche)

Es gibt **keine** Bereichseinschränkungen, die Größe ist nur vom Hauptspeicher abhängig. Bei sehr großen Zahlen verlangsamt sich ihre Verarbeitung wesentlich oder der Rechner stürzt ab.

Beispiele: **(3/5) -123e-3**

((7/1) ist kein Objekt der Klasse **Fraction**. Es ist eine Nachrichtenausdruck, welcher dem Objekt **7** die Nachricht **/** mit dem Parameter **1** schickt und ein Objekt der Klasse **SmallInteger** liefert!)

- Klasse **Object:**
  - Klasse **Collection:**
    - Klasse **SequenceableCollection:**
      - Klasse **ArrayedCollection:**
        - Klasse **Array: Feldkonstante** (Folge von Objekten)

Feldelemente sind Objekte, durch einen Index gekennzeichnet. Das erste Feldelement hat grundsätzlich den Feldindex 1. Die Feldobjekte können verschiedenen Klassen angehören.

Beispiele: **size, at: ″class Object″**
 # (12.5 $A ‘ Hallo ‘ # stop # (1 2 3)) size ⇒ 5
 # (12.5 $A ‘ Hallo ‘ # stop # (1 2 3)) at: 2 ⇒ $A

- Klasse **ByteArray: Bytefeldkonstanten** (Bytefolge)

Beispiele: **# [0 11 255]**

- Klasse **EsString:**
  - Klasse **String: Zeichenkettenkonstanten** (Folge von ASCII-Zeichen)

Sie werden aus mehreren Zeichen (Objekten) zusammengesetzt.

Beispiele: **‘ Smalltalk ‘ ‘ Hallo, Welt ‘ ‘ I ‘‘ m ‘** (‘ als Entwerter)

- Klasse **Symbol: Symbolkonstanten** (Folge von ASCII-Zeichen)

Symbole sind unveränderbare Zeichen bzw. Zeichenketten und stellen ein einzelnes Objekt dar. **Namen** werden vom System als Symbole verwaltet.

Beispiele: **# start # stop # ‘ a string ‘**

Methoden ermöglichen Umwandlungen in einen anderen Klassentyp:

Beispiele:	**2.3 asInteger**	**″class Float″**	⇒	**2**
	2.3 as Fraction	**″class Float″**	⇒	**(23/10)**
	65 asCharacter	**″class Integer″**	⇒	**$A**
	$A asciiValue	**″class Character″**	⇒	**65**

Spezielle Konstanten:

- Klasse **Object:**
  - Klasse **Boolean: Wahrheitswerte**

- Klasse **True**

true verweist auf das einzige Objekt, das Basisobjekt dieser Klasse.

- Klasse **False**

false verweist auf das einzige Objekt, das Basisobjekt dieser Klasse.

Boolean-Objekte lassen sich auf **true** bzw. **false** durch Methoden der Klasse **True** bzw. **False** testen:

Beispiele:
```
| gesucht gefunden datenFeld |
datenFeld := #( 12.5 $A ' Hallo ' # stop # ( 1 2 3 ) ).
gesucht := $A.
gefunden := false.
1 to: datenFeld size do:
[ : index | ( ( datenFeld at: index ) = gesucht)
   ifTrue: [ gefunden := true ] ].
^gefunden                                    ⇒    true
```

- Klasse **Object:**
  - Klasse **UndefinedObject:**

  nil (not in list) verweist auf das einzige Objekt, das Basisobjekt dieser Klasse. Es repräsentiert ein undefiniertes Objekt oder einen undefinierten Zustand. Z.B. haben deklarierte und noch nicht definierte Variablen den Wert **nil**.

  Beispiele:
```
| wert |
wert isNil
ifTrue: [ Transcript show: ' Variable wert ist undefiniert. ' ]
                                             ⇒    an EtTranscript
```

Die Klasse **Object** ist die Wurzel der **ST-Klassenhierarchie**. Mit ihrer Methode **class** kann man die Klassenzugehörigkeit eines Objektes überprüfen. Mit andern Methoden, wie **isInteger**, **isFloat**, **isCharacter**, **isString**, **isSymbol**, erhält man ein **Boolean**-Objekt als Ergebnisobjekt. Diese Methoden werden zum einem in der Klasse **Object** (^**false**) definiert und zum anderen in der eigenen Klasse (^**true)** überlagert.

Beispiele:
```
(4/2) class          ⇒    SmallInteger
-123e-3 class        ⇒    Fraktion
12 isInteger         ⇒    true
```

2.2.4 Variablen *(variable)*

Variablen sind im klassischen Sinn Platzhalter im Speicher für Objekte der Sprache.

Ihre drei **Grundbestandteile** sind:

Name	Anfangsadresse der Bitfolge
Typ	Länge der Bitfolge, ihre Interpretation
Wert	Interpretierte Bitfolge

Eine **Variable** (**Instanzen)** in OO-Sinn ist als Erweiterung dieses Begriffes zu verstehen:

Name	Objektzeiger, Adresse auf das zugewiesene *Objekt*
Typ	Gesamtheit der Objekt-*Daten*, die zu der Objekt-*Klasse* gehören, einschließlich den ererbten *Daten*, ihren Speicherbedarf und ihre Interpretation
Wert	Interpretierte Bitfolgen aller Objekt-*Daten*

ST-Variable:

Name

ST-Name, symbolischer Bezeichner für die Adresse im Speicher.

Typ

ST ist völlig untypisiert, von Seiten des Systems finden keine Typüberprüfungen statt. Variablen müssen zwar vor dem ersten Aufruf deklariert werden. Bei der Deklaration einer Variablen muss nur ihr *Name*, nicht aber die Klasse, angegeben werden. ST stellt einen Pointer zur Verfügung, welcher zunächst auf das Objekt **nil** der Klasse **UndefinedObject** zeigt. Durch jede Wertzuweisung wird der Variablen ein neues Objekt zugeordnet und damit die zugeordnete Klasse geändert. Die Variable zeigt auf das ihr zugewiesene Objekt. Erst zur Laufzeit erfolgen Fehlermeldungen, wenn dieses eine an sich gerichtete Nachricht nicht versteht.

Wert

Der Variablen wird ein Zeiger auf ein *Objekt* zugewiesen, in dessen Speicherbereich sich eine Menge von *Daten* befinden können. Zu dem Wert einer Variablen gehören demnach die interpretierten Bitfolgen aller Daten des zugewiesenen Objekts, einschließlich aller ererbten Daten.

Variablen können mit der Methode **isNil** der Klasse **Object** (**^false**) bzw. **UndefinedObject** (**^true**) auf **nil** gestestet werden: **anObject isNil**

Es werden je nach Lebensdauer und Sichtbarkeit globale und lokale Variablen unterschieden. Ihnen können Objekte unterschiedlicher Klassen zugewiesen werden.

Gültigkeit:	**Sichtbarkeit einer Größe**	⇒	**Zugriff**
Verfügbarkeit:	**Lebensdauer einer Größe**	⇒	**Speicher**

2.2.4.1 Globale Variablen (shared variables)

- Ihre *Namen* beginnen vereinbarungsgemäß mit einem **Großbuchstaben**.
- Sie sind fest im Systemimage installiert, d.h. ihre **Verfügbarkeit** ist uneingeschränkt.

Globale Systemvariablen:

- Namen globaler Variablen dürfen, auch als Klassennamen, noch nicht existieren (hier: Löscht Klasse aus dem System!).
- Sie sind im Image des System gespeichert,
- stehen bei der nächsten Sitzung zur Verfügung,
- vergrößern damit das Image (Zurückhaltung beim Anlegen!),
- müssen explizit dort eingetragen und gelöscht werden,

- können durch die Deklaration lokaler Variablen gleichen Namens verschattet werden.

Globale Variablen sind in der Instanz **Smalltalk** der Klasse **EsSmalltalkDictionary** erfasst. Weitere Variablen können durch Angabe ihres *Namens* und evtl. eines *Objekts* als *Wert* (sonst **nil**) direkt dort eingetragen werden.

Deklaration:
Eine Variable muss vor dem ersten Auftreten im Programmcode mittels einer Nachricht oder in das ST-System direkt eingetragen werden:

- Eintrag durch eine Nachricht: **Smalltalk at:** *#Name* **put:** *Objekt* **″class Dictionary″**
- Direkter Eintrag: **Smalltalk inspect** **″class Dictionary″**
  Mittels dem **Inspektor** kann man globale Variablen deklarieren (Variables/Add...), sich alle globalen Variablen ansehen und diese auch wieder löschen (Variables/Remove Key)

Beispiele:
Transcript ist eine vordefinierte globale Variable der Unterklasse **EtTranscript** der Klasse **EtWorkspace** und ermöglicht Bildschirmausgaben auf das **Transcript**-Fenster.

```
Transcript show: ‘ A ‘ ; show: ‘ B ‘ ; show: ‘ C ‘          ″class EtWorkspace″
```

pi ist eine Klassenmethode der Klasse **Float** zum Erzeugen des Wertes Π. Sie liest diesen Wert aus der Klassenvariablen **Pi**. Diese Methode wird zum Erzeugen einer globalen Variablen mit dem Wert Π verwendet.

```
Smalltalk at: #MyPI put: Float pi
| a | a := MyPI. ^a                    ⇒          3.14159265
```

Klassenvariablen (class variable):

- Sie werden in den Klassen deklariert,
- dienen der Aufnahme von Daten, auf die alle Objekte der Klasse und Unterklassen bzw. deren Methoden zugreifen können,
- und existieren solange, wie die entsprechende Klasse existiert. Die **Verfügbarkeit** einer Klassenvariablen entspricht demnach der Lebensdauer der sie deklarierenden Klasse.
- Ihre **Gültigkeit** ist auf die deklarierende Klasse und ihre Unterklassen eingeschränkt. Klassenvariablen werden vererbt und dürfen deshalb in der entsprechenden Teilhierarchie nur einmal vorkommen. In parallelen Hierarchien ist es möglich, Klassenvariablen gleichen Namens zu verwenden.

Deklaration: **classVariableNames: ‘** *[VariableName ...]* **‘**

Beispiel: Klassedefinition der Klasse **Float**

```
Number variableByteSubclass: #Float          ← Unterklasse der Klasse Number
   classVariableNames: ‘ Formatter Pi ‘
   poolDictionaries: ‘ ‘
```

2.2.4.2 Lokale (private) Variablen

- Ihre *Namen* beginnen vereinbarungsgemäß mit einem **Kleinbuchstaben**.
- Ihre **Verfügbarkeit** ist eingeschränkt.

Instanzvariablen (instance variable):

- Sie werden in den Klassen deklariert, bauen den Datenspeicher eines Objekts auf und sind nur dem Objekt zugänglich, d.h. ihre **Verfügbarkeit** ist auf ein Objekt eingeschränkt.
- Ihre Namen werden an Unterklassen vererbt, innerhalb einer Hierarchie dürfen sie nur einmal deklariert werden, innerhalb paralleler Hierarchien sind gleiche Instanzvariablen möglich.

Deklaration: **instanceVariableNames: ‘** *[VariableName ...]* **‘**

Beispiel: Klassendefinition der Klasse **Fraction**

```
Number subclass: #Fraction                    ← Unterklasse der Klasse Number
   instanceVariableNames:
      ‘ numerator denominator ‘               ←          Instanzvariablen
   classVariableNames: ‘ ‘
   poolDictionaries: ‘ ‘
```

Methode **+** in der Klasse **Fraction**

```
+ aNumber
"Answer a type of Number that is the sum of
    the receiver and the argument, aNumber.

    Fail if aNumber is not a type of Number."

  ^ ( ( self numerator * aNumber denominator) +
      ( self denominator * aNumber numerator ) ) /
      ( self denominator * aNumber denominator )
```

Temporäre Variablen:

- Hilfsgrößen bei der Abarbeitung eines Programmteils zum Speichern von Zwischenergebnissen.
- Die **Verfügbarkeit** einer temporären Variablen entspricht der Laufzeit des Programmteils, in dem sie deklariert wurde (innerhalb einer *Methode*, einer *Ausdrucksfolge* oder eines *Blocks*).
- Ihre Namen dürfen nicht die Namen z. Zt. verfügbarer temporärer Variablen sein.
- Temporäre Variablen verschatten globale Variablen gleichen Namens.

Unterteilung:

Methoden: *methodenName f_Parameter Ausdrucksfolge*

Verlangt eine Methode weitere Parameter, so werden diese im Methodenkopf formal deklariert und mit der Übergabe von Argumenten durch eine Nachricht definiert, diese können innerhalb einer Methode nicht verändert werden und sind nur in der Methode verfügbar.

Ausdrucksfolge: **|** *Variable ...* **|** *Ausdruck [* **.** *Ausdruck] ...*

Variablen einer Ausdrucksfolge werden am Anfang der Folge deklariert, dienen der Aufnahme von Zwischenergebnissen und sind nur innerhalb dieser verfügbar.

Block: **[:** *blockVariable* **|** *blockDefinition* **]**

Blockvariablen werden am Anfang eines Blockes deklariert, sind nur innerhalb des Blockes verfügbar und dienen als eine Art Laufvariable.

Beispiel: Methode **collect:** für die Klasse **Collection.**
collect: erzeugt eine neue Datenstruktur aus der Empfängerklasse, bei der jedes Element nach den Anweisungen des Blocks verändert wurde.

```
collect: aBlock                                    ←  formaler Parameter
"Answer a Collection that is created by iteratively evaluating the one
argument block, aBlock using each element of the receiver as an
argument.

Fail if aBlock is not a one-argument Block."

| newCollection |                                     ←  Ergebnisvariable
newCollection := self growEmptyBy: 0.
self do: [ : element |                             ←  Blockvariable
   newCollection add: ( aBlock value : element ) ].
^ newCollection
```

2.2.4.3 Vordefinierte Variablen

Zwei vordefinierte Variablen sind systemweit bekannt und verwendbar. Ihre Namen sind reserviert.

self

Spricht das aktuelle Objekt selbst an und findet in eigenen Methoden Anwendung. Damit kann ein Objekt sich selbst eine Nachricht schicken. Es sind rekursive Methoden möglich.

Beispiele: Methode **squared** für die Klasse **Number**

```
squared
"Answer the receiver, a type of Number,
    multipled by itself."

^self * self
```

Methode **sign** für die Klasse **Number**

```
sign
"Answer 1 if the receiver is greater than 0,
   answer -1 if the receiver is less than 0,
   otherwise answer 0."

( self > 0 ) ifTrue: [ ^1 ].
( self < 0 ) ifTrue: [ ^-1 ].
^0
```

Methode **factorial** für die Klasse **Integer** ← rekursive Definition der Fakultät

```
factorial
"Answer an Integer which is  the
   factorial of the receiver. For example,
   6 factorial == 6*5*4*3*2*1.

   Fail if the receiver is less than 0."

self > 1 ifTrue: [ ^( self - 1 ) factorial * self ].
self < 0 ifTrue: [ ^self error: self errorNegativeFactorial ].
^1
```

super
Spricht Methodendefinitionen der direkten Oberklasse an und dient im wesentlichen der Initialisierung der aus der Oberklasse ererbten Instanzvariablen mittels der dort definierten Initialisierungsmethode. Damit ist jede Klasse nur für die Initialisierung ihrer eigenen Instanzvariablen zuständig und braucht sich nicht um in der Hierarchie vorher deklarierte zu kümmern.

Beispiel:
- Klasse **Object:**
  - Klasse **CwExtendedPrompter:**
    - Klasse **CwTwoButtonPrompter:**

```
CwExtendedPrompter subclass: #CwTwoButtonPrompter
   instanceVariableNames: ' button1Name button2Name button1
button2 '
   classVariableNames: ' '
   poolDictionaries: ' CwConstants '
```

Methode **initialize** ← Redefinition einer Methode

```
initialize
   "Initialize the receiver."

super initialize.                          ← Methodenaufruf der Oberklasse
   self button1Name: ' OK ' ; button2Name: ' Cancel '
```

2.3 Ausdrücke

Methoden werden im wesentlichen durch **Folgen von Ausdrücke** aufgebaut. Eine Ausdrucksfolge wird durch Hintereinanderschreiben von Ausdrücken gebildet, wobei der Punkt als Trennzeichen Verwendung findet. Temporäre Variablen werden, falls benötigt, am Anfang der Ausdrucksfolge in Senkrechtstrichen deklariert. Ausdrücke liefern bei ihrer Auswertung stets ein Ergebnisobjekt.

Ausdrucksfolge *(expressionSeries)*

Syntax: | *Variable* ... | *Ausdruck* [**.** *Ausdruck*] ...

Die Ausdrucksauswertung erfolgt grundsätzlich von links nach rechts. Ergebnisobjekt ist das zuletzt erzeugte Objekt der Ausdrucksfolge.

Beispiele: **| a | a := # (1 2 3) . a at: 1** ⇒ **1**
| a b c | a := 1 . b := 2 . c := a + b ⇒ **3**

Der Operator Caret
Jede Ausdruckfolge kann nur ein Ergebnisobjekt zurückgeben. In der Regel ist es das als letztes erzeugte Objekt.
Mit dem Caret-Operator ^ (lies: return) kann man das Rückgabeobjekt direkt festlegen. Nach dem Caret-Operator wird das Programm abgebrochen. Es darf kein weiterer Ausdruck abgearbeitet werden, sonst erfolgt eine Fehlermeldung.

Syntax: **^** *Ausdruck*

Beispiel:
```
| beta |
beta := 90 degreesToRadians.
^ 2 * beta sin                          ⇒ 2

| a b |
a := 1. b := 2. a > b ifTrue: [ ^a ]. ^b   ⇒ 2
```

Ausdruck ***(expression)***:
In ST werden Ausdrücke in **einfache Ausdrücke**, **Wertzuweisungen** und **Nachrichtenausdrücke** unterteilt.

2.3.1 Einfacher Ausdruck *(primary)*

Zu den einfachen Ausdrücken gehören die schon betrachteten **Variablen** und **Konstante**, des weiteren **Ausdrucksgruppierungen** und **Blöcke.**

Konstanten und **Variablen** haben als Wert das ihnen zugewiesene Objekt.

2.3.1.1 Ausdrucksgruppierung

Syntax: **(** *Ausdruck* **)**

Durch Klammern lassen sich Ausdrücke gruppieren. Der Wert einer *Ausdrucksgruppierung* entspricht dem Wert des geklammerten *Ausdrucks*.

Beispiele:
```
( 2 * 3 ) + ( 3 * 4 )      ⇒  18
1 @ 1 class                ⇒  1 @ SmallInteger
( 1 @ 1 ) class            ⇒  Point
( 3 < 4 ) + 5              ⇒  Fehlermeldung
```

2.3.1.2 Block (block)

Syntax: **[** *[* **:** *blockVariable* **...** | *]* *Ausdrucksfolge* **]**

Blöcke sind in ST natürlich auch Objekte und zwar der Klasse **BlockContextTemplate**, Unterklasse der Klasse **Block**. Sie werden aus einer Folge von Ausdrücken, durch eckige Klammern begrenzt, gebildet und dienen der Strukturierung. Der Wert einer *Blocks* entspricht dem Wert der geklammerten *Ausdrucksfolge.*

Beispiele:
```
| sum |
sum := 0.
[ sum := sum + 1. ^sum ] class          ⇒  BlockContextTemplate

| sum |
sum := 0.
[ | a | a := 2. sum := sum + a ].
^sum                                    ⇒  0
```

Die Anweisungen innerhalb eines Blockes werden nicht sofort ausgeführt, sondern erst, wenn an den Block die Nachricht **value** gesendet wurde.

Beispiel:
```
| sum |
sum := 0.
[ | a | a := 2. sum := sum + a ] value.
^sum                                    ⇒  2

| indexUp index|
index := 0.
indexUp := [ index := index + 1 ].
index := indexUp value.
index := indexUp value.
^index                                  ⇒  2
```

Blöcke mit Blockvariablen benötigen zur Auswertung einen oder mehrere Variablenwerte, die als Argument der Nachrichten **value: value:value: value:value:value:** bzw. **valueWithArguments:** (**class BlockContextTemplate**) mitgegeben werden können.

Beispiel:
```
| sum |
sum := 0.
[ : a | sum := sum + a ] value: 2.
^sum                                    ⇒  2

| sum |
sum := 0.
[ : a : b | sum := sum + a + b ] value: 2 value: 3.
^sum                                    ⇒  5
```

```
| sum |
sum := 0.
[ : a : b | sum := sum + a + b ] valueWithArguments: #( 2 3 ).
^sum                                                    ⇒   5
```

7	**Ausdrucksgruppierung** **Block**	**(** *Ausdruck* **)** **[** *[* **:** *blockVariable* **...** **|** *]* *Ausdrucksfolge* **]**	→
1	**Ausdrucksfolge**	**|** *Variable ...* **|** *Ausdruck [* **.** *Ausdruck] ...*	→

2.3.2 Wertzuweisung

Syntax: *Variable* **:=** *Ausdruck*

ST-80: ← statt **:=**.
Wertzuweisungen sind keine Nachrichten!
Der *Variablen* wird eine Kopie des Zeigers auf das **Objekt** zugewiesen (nicht eine Kopie des Objektes) welches Ergebnis des *Ausdrucks* ist. Wertzuweisungen sind somit Verweise auf ein Objekt. Der Wert einer Wertzuweisung entspricht dem Ergebnisobjekt.

Beispiel:
```
| a b |
a := Array new: 3.
a at: 1 put: 1. a at: 2 put: 2. a at: 3 put: 3.
b := a.                                  "a und b zeigen auf dasselbe Objekt"
b at: 1 put: ' Eins '.
a includes: ' Eins '                     ⇒   true
```

Mehrfachwertzuweisungen sind möglich, ihre Auswertung erfolgt von rechts nach links.

Beispiel: `| a b c | a := b := c := #( 1 2 3 )`

7	**Ausdrucksgruppierung** **Block**	**(** *Ausdruck* **)** **[** *[* **:** *blockVariable* **...** **|** *]* *Ausdrucksfolge* **]**	→
2	**Wertzuweisung**	*Variable* **:=** *Ausdruck*	←
1	**Ausdrucksfolge**	**|** *Variable ...* **|** *Ausdruck [* **.** *Ausdruck] ...*	→

2.3.3 Nachrichtenausdruck *(messageExpression)*

Nachrichten dienen der Kommunikation zwischen den Objekten. Die Instanzvariablen eines Objekts sind nach außen abgekapselt und können von anderen Objekten nicht verändert werden. Eine *Nachricht*, die an das *Objekt (receiver)* gerichtet wurde, stößt eine namensgleiche Objektmethode an. Folglich kann ein Objekt eine Nachricht nur verarbeiten, falls die passende Methode in der Objektklasse definiert wurde.

Syntax: *Objekt Nachricht*

Ein Nachrichtenausdruck liefert stets als Ergebnis seiner Verarbeitung ein Objekt zurück. Wird durch die angestoßene Methode kein Ergebnisobjekt erzeugt, so wird das Empfängerobjekt der Nachricht zurückgegeben.

Nachrichtenausdrücke können einfach oder strukturiert sein.

Beispiel: **(1 @ 1) class** ⇒ **Point**
Methode **@** der Klasse **Number** erzeugt ein Objekt der Klasse **Point**

(1 @ 1) + 1 ⇒ **2 @ 2**

Methode **+** der Klasse **Point**:

```
+ delta

    "Answer a new point that is the sum of the receiver and delta."

    ^delta isPoint
        ifTrue: [self class x: x + delta x y: (y + delta y)]    ″Klassenmethode x:y:″
        ifFalse: [self class x: x + delta y: (y + delta)]
```

Es wird in den Blöcken nicht untersucht, ob **delta** eine Zahl ist. Bei Fehlern erscheint die Meldung, dass die Methode + vom Empfängerobjekt nicht verstanden wird.

Nachricht
In Abhängigkeit der zugrundegelegten Nachricht wird in **einwertige**, **zweiwertige** und **Schlüsselwortnachrichten** unterschieden.

2.3.3.1 Einwertige Nachricht (unaryMessage)

Syntax: *selector*

selector
Selektoren dienen der Identifikation einer Nachricht und sind *ST-Namen*, die vereinbarungsgemäß mit einem Kleinbuchstaben beginnen. Sie stoßen Objektmethoden mit diesem Namen an, welche keine Argumente benötigen.

Beispiel:

class, **size**	″**class Object**″
asInteger	″**class Charater**″
isInteger, **asCharacter**	″**class Integer**″
sqrt, **degreesToRadians**, **radiansToDegrees**	″**class Number**″
asUppercase	″**class EsString**″

12 class ⇒ **SmallInteger**

# (1 2 3) size	⇒	**3**
$5 asInteger	⇒	**53**
5 asCharacter	⇒	**$♣**
12 isInteger	⇒	**true**
16 sqrt	⇒	**4.0**
180 degreesToRadians	⇒	**3.14159265**
Float pi radiansToDegrees	⇒	**180.0**
‘ Smalltalk ‘ asUppercase	⇒	**SMALLTALK**

2.3.3.2 Zweiwertige Nachricht (binaryMessage)

Syntax: *selector argument*

selector
Selektoren sind Operatoren, die aus ein oder zwei Sonderzeichen bestehen.

argument
Argumente sind *Ausdrücke*, deren Ergebnisobjekt als Operand übergeben wird.

Zwischen dem Zielobjekt, dem Selektor und dem Argument sind Leerzeichen überflüssig. Eine Ausnahme bildet das Minuszeichen.

Beispiel:

3 >= 5	⇒	**false**
5 -3	⇒	**Fehler: - 3**
5 - 3	⇒	**2**
‘ Hallo ‘ , ‘ ! ‘	⇒	**Hallo!**
(3 / 7) + (6 / 8)	⇒	**(33 / 28)**
2 @ 3	⇒	**2 @ 3**

```
| a b |
a := Array new: 3. a at: 1 put: $a. a at: 2 put: $b. a at: 2 put:
$c.
b := # ( $a $b $c ).
```

a = b	⇒	**true**
a == b	⇒	**false**

Arithmetische zweiwertige Nachrichten:
+ *(addieren)*, - *(subtrahieren)*, * *(multiplizieren)*, / *(dividieren)*,
// *(ganzzahligdividieren)*, \\ *(modulo)*

Vergleiche mit zweiwertigen Nachrichten:
< *(kleiner)*, <= *(kleiner gleich)*, > *(größer)*, >= *(größer gleich)*,
= *(gleich)*, == *(identisch)*

Logische zweiwertige Nachrichten:
& *(UND)*, | *(ODER)*

Andere zweiwertige Nachrichten:
, ***(verketten)*****, @** ***(Punkt erzeugen)***

Weitere Beispiele:

3/2+5/2	⇒	**13/4**
3/2+(5/2)	⇒	**4**
3/4	⇒	**3/4**
3/4.0	⇒	**0.75**
3=3.0	⇒	**true**
3==3.0	⇒	**false**
3==(3/1)	⇒	**true**

2.3.3.3 Schlüsselwortnachricht (keywordMessage)

Syntax: *selector***:** *argument [selector***:** *argument . . .]*

selector
Selektoren sind *ST-Namen*, die vereinbarungsgemäß mit einem Kleinbuchstaben beginnen. Sie stoßen Methoden mit diesem Namen an, welche Argumente benötigen.

argument
Ausdruck, dessen Ergebnisobjekt als Argument der Objektmethode übergeben wird. Es gibt Schlüsselwortnachrichten mit einem oder mehreren Argumenten.

Beispiel:	**gcd:**		**″class Integer″**
	between:and:		**″class Magnitude″**
	to:by:do:		**″class Integer″**
	21 gcd: 6	⇒	**3**
	15 between: 10 and: 20	⇒	**true**
	1 to: 10 by: 2 do: [: i | Transcript show: i]	⇒	**13579**

2.3.3.4 Nachrichtenkette

Syntax: *Nachricht Nachricht* **. . .**

Ein Nachrichtenausdruck liefert stets ein Objekt zurück, so dass das Ergebnisobjekt eine weitere Nachricht verarbeiten kann. Die Nachrichtenauswertung erfolgt grundsätzlich von links nach rechts unter Berücksichtigung von Prioritäten. Das Ergebnisobjekt ist das zuletzt erhaltene Objekt.

Beispiel:	**rounded, sqrt**		**″class Number″**
	2 + 4 / 2	⇒	**3**

2 sqrt rounded ⇒ **1**
16 sqrt * 5 ⇒ **20.0**
5 * 16 sqrt ⇒ **20.0**

7	**Ausdrucksgruppierung** **Block**	**(** *Ausdruck* **)** **[** *[* **:** *blockVariable* **. . . |** *]* *Ausdrucksfolge* **]**	→
6	**unäre Nachricht**	*selector*	→
5	**binäre Nachricht**	*selector argument*	→
4	**Schlüsselwortnachricht**	*selector* **:** *argument [selector* **:** *argument . . .]*	→
2	**Wertzuweisung**	*Variable* **:=** *Ausdruck*	←
1	**Ausdrucksfolge**	**|** *Variable . . .* **|** *Ausdruck [* **.** *Ausdruck] . . .*	→

Abschlußbeispiel:

5 factorial between: 3 * 6 and: ‘ Smalltalk ‘ size * 20 ⇒ **true**

```
----------------              -----------------------
      120                                9
                   ----------   ------------------------------------
                       18                   180
```

2.3.4 Nachrichtenkaskade *(cascadeMessage)*

Syntax: *Nachrichtenausdruck* **;** *Nachricht* **. . .**

Das Empfängerobjekt einer Nachrichtenkaskade ist stets dasselbe. Die Nachrichtenauswertung erfolgt grundsätzlich von links nach rechts. Ergebnisobjekt ist das zuletzt erhaltene Objekt.

Beispiel: **2 * 3 ; * 4 ; * 5** ⇒ **10**

| a |
a := Array new: 5 ; at: 1 put: ‘ Eins ‘ ; at: 2 put: ‘ Zwei ‘ ;
at: 3 put: ‘ Drei ‘.
^a ⇒ **(‘ Eins ‘ ‘ Zwei ‘ ‘ Drei ‘)**

7	**Ausdrucksgruppierung** **Block**	**(** *Ausdruck* **)** **[** *[* **:** *blockVariable* **. . . |** *]* *Ausdrucksfolge* **]**	→
6	**unäre Nachricht**	*selector*	→
5	**binäre Nachricht**	*selector argument*	→
4	**Schlüsselwortnachricht**	*selector* **:** *argument [selector* **:** *argument . . .]*	→
3	**Nachrichtenkaskade**	*Nachrichtenausdruck* **;** *Nachricht* **. . .**	→
2	**Wertzuweisung**	*Variable* **:=** *Ausdruck*	←
1	**Ausdrucksfolge**	**|** *Variable . . .* **|** *Ausdruck [* **.** *Ausdruck] . . .*	→

2.3.5 Strukturierte Nachrichtenausdrücke

Analog herkömmlicher Programmiersprachen gibt es in **ST** auch die Möglichkeiten der strukturierten Programmierung in Form von **Auswahl- und Schleifennachrichtenausdrücken**. In Abhängigkeit von *Bedingungen*, das sind Ausdrücke, die ein **Boolean**-Objekt liefern, werden Nachrichten nicht oder auch mehrmals abgesendet.

2.3.5.1 Logische Nachrichtenausdrucksverknüpfung

Mittels der beiden Nachrichten **and:** bzw. **or:** kann man komplexere logische Abfragen aufbauen.

Syntax: *Bedingung* **and:** *booleanBlock*

Bedingung
Ausdruck, dessen Auswertung ein **Boolean**-Objekt liefert.

and: or:
Methoden der Unterklassen der Klasse **Boolean**, die als Argument einen *Block* benötigen. Dieser erhält die Nachricht **value** in Abhängigkeit von der Auswertung der *Bedingung*. Das Ergebnisobjekt des Blockes ist ein **Boolean**-Objekt und wird in der üblichen Weise weiterverarbeitet. Die Methoden arbeiten optimierend, d. h. die Auswertung des *booleanBlockes* erfolgt nur, falls erforderlich.

booleanBlock
Argument ist ein *Block*, dessen Auswertung ein **Boolean**-Objekt liefert.

Beispiel: **| c |**
c := $3. (c >= $0) and: [c <= $9] ⇒ **true**

2.3.5.2 Auswahlnachrichtenausdrücke

Syntax: *Bedingung* **ifTrue:** *trueBlock*
ifFalse: *falseBlock*

Bedingung
Ausdruck, dessen Auswertung ein **Boolean**-Objekt liefert, welches Empfänger der darauffolgenden Nachricht ist.

ifTrue:ifFalse: ifFalse:ifTrue: ifTrue: ifFalse:
Diese Methoden der Unterklassen der Klasse **Boolean** bestimmen entsprechend dem Wert einer *Bedingung* den als Argument mitgelieferten auszuführenden *Block*. Dieser bekommt die Nachricht **value** zugesendet.

trueBlock falseBlock
Argumente sind *Blöcke*.

Beispiel:
```
| prompter zahl1 zahl2 max |
prompter := CwTextPrompter new title: ' '.
                "CwTextPrompter ist eine Klasse für ein einfaches Eingabefenster,
                                        Unterklasse der Klasse CwPrompter."

zahl1 := ( prompter answerString: ' ' ;
   messageString: 'Geben Sie bitte die erste Zahl ein : ' )
   prompt asNumber.        "Eingabezeichen werden in Ziffern umgewandelt,
                            Zeichen, die keine Ziffern sind, werden ignoriert,
                                    bei nur Text wird das Objekt 0 geliefert."
zahl2 := ( prompter answerString: ' ' ;
   messageString: 'Geben Sie bitte die zweite Zahl ein : ' )
   prompt asNumber.

zahl1 > zahl2
   ifTrue: [ max := zahl1 ]
   ifFalse: [ max := zahl2 ].

prompter := CwMessagePrompter new title: ' Ergebnis ' ;
        "CwMessagePrompter ist eine Klasse für ein einfaches Ausgabefenster,
                                        Unterklasse der Klasse CwPrompter."
   messageString: ' Die groesste Zahl ist: ' , max printString ;
                                "Ausgaben sind nur als Text möglich."
   buttonType: XmOK ;
   iconType: XmNOICON ;
   prompt
```

■ **1. Aufgabe**

2.3.5.3 Schleifennachrichtenausdrücke

timesRepeat: - Nachricht

Syntax: *anInteger* **timesRepeat:** *repeatBlock*

anInteger
Nachricht an ein Objekt der Klasse **Integer**, welches die Anzahl der Wiederholungen bestimmt.

timesRepeat:
Methode der Klasse **Number**.

repeatBlock
Argument ist der zu wiederholende *Block.*

Beispiele:
```
| sum index |
sum := 0. index := 1.
100 timesRepeat [ sum := sum + index. index := index + 1 ].
```

```
^sum                                    ⇒          5050
```

```
|prompter oldString newString oldIndex newIndex ch|

prompter := CwTextPrompter new title: ' Eingabe '.
newIndex := oldIndex := 1.
oldString :=
   ( prompter answerString: ' '; messageString: ' String ' )
prompt.
newString := String new: oldString size.
oldString size timesRepeat:
   [ch := oldString at: oldIndex.
      ch isVowel
         ifFalse:
            [ newString at: newIndex put: ch.
            "isVowel ist eine Methode der Klasse Character,
            die auf Vokale testet."
            newIndex := newIndex + 1].
   oldIndex := oldIndex + 1].
newString trimBlanks.   "trimBlanks ist eine Methode der Klasse EsString,
                                        die vorhandene Leerzeichen löscht."
prompter := CwMessagePrompter new title: ' Ergebnis ';
   messageString: newString;
   buttonType: XmOK;
   iconType: XmNOICON;
   prompt
```

Beide Strings (old und new) haben die gleiche Länge!

whileTrue: -, whileFalse: - Nachricht

Syntax: *whileBlock* **whileTrue:** *repeatBlock*

Der *repeatBlock* wird solange wiederholt, bis der *whileBlock* das Objekt **false** liefert.

whileBlock
Block, dessen Auswertung ein Objekt der Klasse **Boolean** liefert, welches Empfänger der darauffolgenden Nachricht ist.

whileTrue: whileFalse:
Diese Methoden der Klasse **Block** wiederholen solange den als Argument mitgelieferten *repeatBlock*, bis der *whileBlock* das Objekt **false** bzw. **true** liefert.

Beispiel:
```
|number | number := 0.
[ number < 100 ] whileTrue: [ number := number + 1 ].
^number                                 ⇒          100
```

to:by:do: - Nachricht

Syntax:
startWert **to:** *endWert* *[* **by:** *schrittweite* *]* **do: [:** *laufVariable* **|** *Ausdrucksfolge* **]**

startWert endWert schrittweite
Ausdrücke, die ein Objekt der Klasse **Number** liefern.
Es wird über den Empfänger iteriert. Jedes Objekt wird der Blockvariablen übergeben.

Die **to:do: - Nachricht** hat die *Schrittweite* 1.

laufVariable
Temporäre Blockvariable.

Beispiel:
```
| sum |
sum := 0.
1/10 to: 1 by: 1/10 do: [ : i | sum := sum + i ].
^sum                                    ⇒        11/2

| anArray |
anArray := Array new: 5.
1 to: anArray size do:
   [ : index | anArray at: index put: 0 ]
^anArray                                ⇒    ( 0 0 0 0 0 )

″ Ermittelt die Anzahl der Zahlen in einem String ″
| sum string |
sum := 0.
string := ‘ asdfr1fdf3443478 ‘.
1 to: string size do:
[ : i | ( string at: i ) isDigit ifTrue: [ sum := sum + 1 ] ].
^sum                                    ⇒        8
```

do: - Nachricht

Syntax: *Object* **do: [:** *laufVariable* **|** *Ausdrucksfolge* **]**

Es wird über den Empfänger iteriert. Jedes Objekt wird der Blockvariablen übergeben. Der Empfänger kann jede Unterklasse von **Collection** und **Stream** sein.

Beispiel:
```
″ Ermittelt die Anzahl der Zahlen in einem String ″
| sum |
sum := 0.
‘ asdfr1fdf3443478 ‘ do: [ : i | i isDigit ifTrue: [ sum := sum +
1 ] ].
^sum                                    ⇒        8
```

```
″Berechne die Anzahl der Vokale einer Datei″
| input sumOfVowels |
sumOfVowels := 0.
"CfsDirectoryDescriptor chdir: ' c:\user '."
(input := CfsReadFileStream open: 'c:\user\max.st')
   isCfsError ifTrue: [^self error: input message].
input do:
   [ :ch | ch isVowel ifTrue: [ sumOfVowels := sumOfVowels +
1]].
input close.
^sumOfVowels                                    ⇒          95
```

collect:, select:, reject:

Bei diesen weiteren leistungsfähigen Nachrichten ist der Empfänger ein Objekt von **Collection**.

collect: ist eine Methode von **Collection** und deren Unterklassen und erzeugt eine neue Datenstruktur aus der Empfängerklasse, bei der jedes Element nach den Anweisungen des Blocks verändert wurde. Gegenüber einer einfachen **to:by:do:**-Nachricht ist die **collect:** -Nachricht wesentlich langsamer.

Beispiel:
```
″ Berechne die Quadratwurzeln eines Feldes ″
| a b |
a := #( 1 2 3 4 5 6 7 8 9 10 ).
b := a collect: [ : i | i squared ].
^b                                  ⇒   ( 1 4 9 16 25 36 49 64 81 100 )
```

select: ist eine Methode von **Collection** und deren Unterklassen und erzeugt eine neue Datenstruktur, in der alle Elemente des Empfänger enthalten sind, für die ein gegebener Block zutrifft.

Beispiel:
```
″ Berechne die Anzahl der geraden Zahlen in einem Feld ″
| a b |
a := Array new: 10001.
1 to: a size do: [ : i | a at: i put: i ].
b := a select: [ : n | n even ].
^b size                                         ⇒          5000
```

reject: ist eine Methode von **Collection** und deren Unterklassen und erzeugt eine neue Datenstruktur, in der alle Elemente des Empfängers enthalten sind, für die der Block **nicht** zutrifft.

Beispiel:
```
″ Berechne die Anzahl der ungeraden Zahlen in einem Feld ″
| a b |
a := Array new: 10001.
1 to: a size do: [ : i | a at: i put: i ].
b := a reject: [ : n | n even ].
^b size                                         ⇒          5001
```