

# 3 Syntax von Programmiersprachen

## Syntax („Lehre vom Satzbau“)

- ▶ formale Beschreibung des Aufbaus der „Worte“ und „Sätze“, die zu einer Sprache gehören;
- ▶ im Falle einer **Programmiersprache** Festlegung, wie Programme aussehen müssen.

## Hilfsmittel bei natürlicher Sprache

- ▶ Wörterbücher;
- ▶ Rechtschreibregeln, Trennungsregeln, Grammatikregeln;
- ▶ Ausnahmelisten;
- ▶ Sprachgefühl.

# Hilfsmittel

## Hilfsmittel bei Programmiersprachen

- ▶ Listen von **Schlüsselwörtern** wie `if`, `int`, `else`, `while`...
- ▶ Regeln, wie einzelne Worte (**Tokens**) z.B. **Namen** gebildet werden.  
Frage: Ist `x10` ein zulässiger Name für eine Variable (oder `_ab` oder `A#B` oder `0A?B`)?...
- ▶ Grammatikregeln, die angeben, wie größere Komponenten aus kleineren aufgebaut werden.  
Frage: Ist ein `while`-Statement im `else`-Teil erlaubt?
- ▶ Kontextbedingungen.  
**Beispiel:** Eine Variable muss erst deklariert sein, bevor sie verwendet wird.

Programmiersprachen sind

- ▶ formalisierter als natürliche Sprache
- ▶ besser für maschinelle Verarbeitung geeignet.

## Semantik („Lehre von der Bedeutung“)

- ▶ Ein Satz einer (natürlichen) Sprache verfügt zusätzlich über eine **Bedeutung**, d.h. teilt einem Hörer/Leser einen Sachverhalt mit (↑**Information**)
- ▶ Ein Satz einer Programmiersprache, d.h. ein Programm verfügt ebenfalls über eine **Bedeutung**...

# Syntax vs. Semantik

Die Bedeutung eines Programms ist

- ▶ alle möglichen **Ausführungen** der beschriebenen Berechnung (↑**operationelle Semantik**); oder
- ▶ die definierte **Abbildung** der Eingaben auf die Ausgaben (↑**denotationelle Semantik**).

## Achtung!

Ist ein Programm **syntaktisch korrekt**, heißt das noch lange nicht, dass es auch das „richtige“ tut, d.h. **semantisch korrekt** ist!!!

## 3.1 Reservierte Wörter

- ▶ `int`  
⇒ Bezeichner für Basistypen;
- ▶ `if, else, then, while...`  
⇒ Schlüsselwörter für Programmkonstrukte;
- ▶ `(, ), ", ', {, }, ,, ;`  
⇒ Sonderzeichen;

## 3.2 Was ist ein erlaubter Name?

### Schritt 1:

Festlegung erlaubter Zeichen:

```
letter ::= $ | _ | a | ... | z | A | ... | Z
digit  ::= 0 | ... | 9
```

- ▶ **letter** und **digit** bezeichnen **Zeichenklassen**, d.h. Mengen von Zeichen, die gleich behandelt werden.
- ▶ Das Symbol „|“ trennt zulässige Alternativen.
- ▶ Das Symbol „...“ repräsentiert die Faulheit, alle Alternativen wirklich aufzuzählen.



## 3.2 Was ist ein erlaubter Name?

### Schritt 2:

Festlegung der Zeichenanordnung:

$$\text{name} \quad ::= \quad \text{letter ( letter | digit )}^*$$

- ▶ Erst kommt ein Zeichen der Klasse **letter**, dann eine (eventuell auch leere) Folge von Zeichen entweder aus **letter** oder aus **digit**.
- ▶ Der Operator „\*“ bedeutet „beliebig oft wiederholen“ („weglassen“ ist 0-malige Wiederholung).
- ▶ Der Operator „\*“ ist ein **Postfix**-Operator. Das heißt, er steht hinter seinem Argument.

# Beispiele

\_178

Das\_ist\_kein\_Name

x

\_

\$Password\$

...sind legale Namen.

5ABC

!Hallo!

x'

a=b

-178

...sind keine legalen Namen.

## Achtung

Reservierte Wörter sind als Namen verboten.

## 3.3 Ganze Zahlen

Werte, die direkt im Programm stehen, heißen **Konstanten**.  
Ganze nichtnegative Zahlen bestehen aus einer nichtleeren Folge von Ziffern:

`number ::= digit digit*`

- ▶ Wie sähe die Regel aus, wenn wir führende Nullen verbieten wollen?

# Beispiele

17

12490

42

0

00070

...sind `int`-Konstanten

"Hello World!"

0.5e+128

...sind keine `int`-Konstanten

# Reguläre Ausdrücke

Die Alternative hat eine geringere Bindungsstärke als die Konkatenation. D.h.  $ab|c$  steht für die Wörter  $ab$  oder  $c$  und nicht für  $ab$  oder  $ac$ .

Ausdrücke, die aus Zeichen(-klassen) mithilfe von

- | (Alternative)
- \* (Iteration)
- (Konkatenation) sowie
- ? (Option)

...aufgebaut sind, heißen **reguläre Ausdrücke** (↑Automatentheorie).

Der Postfix-Operator „?“ besagt, dass das Argument eventuell auch fehlen darf, d.h. einmal oder keinmal vorkommt.

Gelegentlich sind auch  $\epsilon$ , d.h. das „leere Wort“ sowie  $\emptyset$ , d.h. die leere Menge zugelassen.

## Beispiele

Reguläre Ausdrücke reichen zur Beschreibung **einfacher** Mengen von Worten aus.

- ▶  $(\text{letter letter})^*$   
⇒ alle Wörter gerader Länge (über  $\$, \_, a, \dots, z, A, \dots, Z$ );
- ▶  $\text{letter}^* \text{test letter}^*$   
⇒ alle Wörter, die das Teilwort **test** enthalten;
- ▶  $\_ \text{digit}^* 17$   
⇒ alle Wörter, die mit  $\_$  anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit 17 aufhört;
- ▶  $\text{exp} ::= (\text{e|E})(+|-)? \text{digit digit}^*$   
 $\text{float} ::= \text{digit digit}^* \text{exp} \mid$   
 $\text{digit}^* (\text{digit} \cdot \mid \cdot \text{digit}) \text{digit}^* \text{exp}?$   
⇒ alle Gleitkommazahlen...

# Programmverarbeitung

## 1. Phase (↑Scanner)

Identifizierung von

- ▶ reservierten Wörtern,
- ▶ Namen,
- ▶ Konstanten

Ignorierung von

- ▶ Whitespace,
- ▶ Kommentaren

⇒ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter („Tokens“) zerlegt.

## 2. Phase (↑Parser)

Analyse der **Struktur** des Programms.

## 3.4 Struktur von Programmen

Programme sind **hierarchisch** aus Komponenten aufgebaut. Für jede Komponente geben wir Regeln an, wie sie aus anderen Komponenten zusammengesetzt sein können.

```
program ::= decl* stmt*
decl    ::= type name ( , name )* ;
type    ::= int
```

- ▶ Ein Programm besteht aus einer Folge von Deklarationen, gefolgt von einer Folge von Statements.
- ▶ Eine Deklaration gibt den Typ an, hier: **int**, gefolgt von einer Komma-separierten Liste von Variablennamen.



# Anweisungen

```
stmt ::= ; | { stmt* } |  
      name = expr; | name = read(); |  
      write( expr ); |  
      if ( cond ) stmt |  
      if ( cond ) stmt else stmt |  
      while ( cond ) stmt
```

- ▶ Ein Statement ist entweder „leer“ (d.h. gleich ; ) oder eine geklammerte Folge von Statements;
- ▶ oder eine Zuweisung, eine Lese- oder Schreiboperation;
- ▶ eine (einseitige oder zweiseitige) bedingte Verzweigung;
- ▶ oder eine Schleife.

# Ausdrücke

`expr` ::= `number` | `name` | `( expr )` |  
`unop expr` | `expr binop expr`

`unop` ::= `-`

`binop` ::= `-` | `+` | `*` | `/` | `%`

- ▶ Ein Ausdruck ist eine Konstante, eine Variable oder ein geklammerter Ausdruck
- ▶ oder ein unärer Operator, angewandt auf einen Ausdruck,
- ▶ oder ein binärer Operator, angewandt auf zwei Argumentausdrücke.
- ▶ Einziger unärer Operator ist (bisher) die Negation.
- ▶ Mögliche binäre Operatoren sind Addition, Subtraktion, Multiplikation, (ganzzahlige) Division und Modulo.

# Bedingungen

```
cond ::= true | false | ( cond ) |  
      expr comp expr |  
      bunop cond | cond bbinop cond  
comp ::= == | != | <= | < | >= | >  
bunop ::= !  
bbinop ::= && | ||
```

- ▶ Bedingungen unterscheiden sich dadurch von Ausdrücken, dass ihr Wert nicht vom Typ `int` ist sondern `true` oder `false` (ein **Wahrheitswert** – vom Typ `boolean`).
- ▶ Bedingungen sind darum Konstanten, Vergleiche
- ▶ oder logische Verknüpfungen anderer Bedingungen.

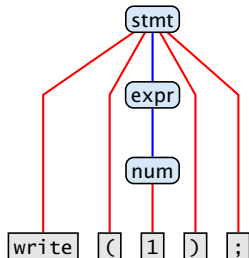
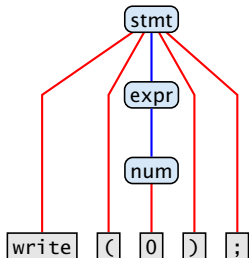
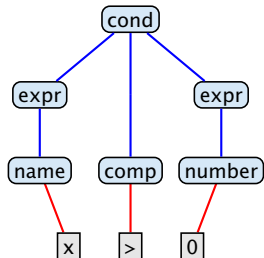
# Beispiel

```
int x;  
x = read();  
if (x > 0)  
    write(1);  
else  
    write(0);
```

Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch **Syntax-Bäume**.

# Syntaxbäume

Syntaxbäume für  $x > 0$  sowie `write(0);` und `write(1);`



**Blätter:**

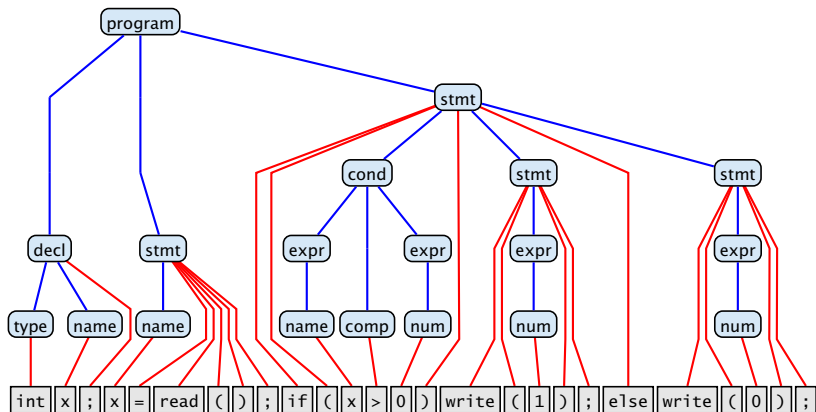
Wörter/Tokens

**innere Knoten:**

Namen von Programmbestandteilen

# Beispiel

Der komplette Syntaxbaum unseres Beispiels:

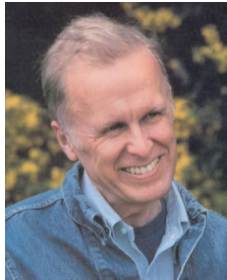


# Bemerkungen

- ▶ Die vorgestellte Methode der Beschreibung von Syntax heißt **EBNF-Notation** (**E**xtended **B**ackus **N**aur **F**orm Notation).
- ▶ Ein anderer Name dafür ist **erweiterte kontextfreie Grammatik** (↑**Linguistik**, ↑**Automatentheorie**).
- ▶ Linke Seiten von Regeln heißen auch **Nichtterminale**.
- ▶ Tokens heißen auch **Terminale**.



Noam Chomsky,  
MIT



John Backus, IBM  
Turing Award  
(Erfinder von Fortran)



Peter Naur,  
Turing Award  
(Erfinder von Algol60)



# Kontextfreie Grammatiken

## Achtung:

- ▶ Die regulären Ausdrücke auf den rechten Regelseiten können sowohl Terminale wie Nichtterminale enthalten.
- ▶ Deshalb sind kontextfreie Grammatiken **mächtiger** als reguläre Ausdrücke.

## Beispiel:

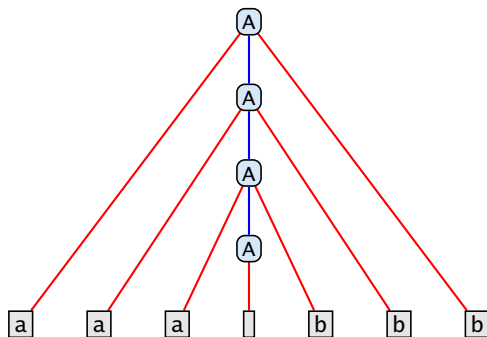
$$\mathcal{L} = \{\epsilon, ab, aabb, aaabbb, \dots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

$$A ::= (aAb)?$$

# Kontextfreie Grammatiken

Syntaxbaum für das Wort **aaabbb**:



Für  $\mathcal{L}$  gibt es aber keinen regulären Ausdruck  
(↑Automatentheorie).