3.1 Reservierte Wörter

- `int`
  → Bezeichner für Basis-Typen;
- `if`, `else`, `while`
  → Schlüsselwörter aus Programm-Konstrukten;
- `(`,`)`, `"`,`'`, `{`,`}`, `,`,`;`
  → Sonderzeichen.

3.2 Was ist ein erlaubter Name?

Schritt 1: Angabe der erlaubten Zeichen:

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

digit ::= `0` | ... | `9`

3.2 Was ist ein erlaubter Name?

Schritt 1: Angabe der erlaubten 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 :-)

Schritt 2: Angabe der Anordnung der Zeichen:

name ::= 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 ofte Wiederholung" ("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'
  -178
  ```

... sind keine legalen Namen :-)
  ```

- ```
  5ABC
  !Hallo!
  x'
  -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:

$$\text{number} \quad ::= \quad \text{digit digit}^*$$

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 alles legale `int`-Konstanten.
- `"Hello World!"`
  `0.5e+128`

  ... sind keine `int`-Konstanten.

Ausdrücke, die aus Zeichen (-klassen) mithilfe von

| (Alternative)

* (Iteration)

(Konkatenation) sowie

? (Option)

... aufgebaut sind, heißen reguläre Ausdrücke[a] (↑Automatentheorie).

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

[a] Gelegentlich sind auch ε, d.h. das "leere Wort" sowie ∅, d.h. die leere Menge zugelassen.

Reguläre Ausdrücke reichen zur Beschreibung einfacher Mengen von Worten aus.

- (letter letter)*
  – alle Wörter gerader Länge (über `a,...,z,A,...,Z`);
- letter* `test` letter*
  – alle Wörter, die das Teilwort `test` enthalten;
- _ digit* `17`
  – alle Wörter, die mit _ anfangen, dann eine beliebige Folge von Ziffern aufweisen, die mit `17` aufhört;
- exp ::= (`e`|`E`)(`+`|`-`)? digit digit*
  float ::= digit digit* exp |
  digit* (digit . | . digit) digit* exp?
  – alle Gleitkomma-Zahlen ...

Identifizierung von

- reservierten Wörtern,
- Namen,
- Konstanten

Ignorierung von

- White Space,
- Kommentaren

... erfolgt in einer ersten Phase (↑Scanner)

$\Longrightarrow$ Input wird mit regulären Ausdrücken verglichen und dabei in Wörter (“Tokens”) aufgeteilt.

In einer zweiten Phase wird die Struktur des Programms analysiert (↑Parser).

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`

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 Variablen-Namen.

```
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 Schreib-Operation;
- eine (einseitige oder zweiseitige) bedingte Verzweigung;
- oder eine Schleife.

```
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, angewand auf zwei Argument-Ausdrücke.
- Einziger unärer Operator ist (bisher :-) die Negation.
- Mögliche binäre Operatoren sind Addition, Subtraktion, Multiplikation, (ganz-zahlige) Division und Modulo.

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.

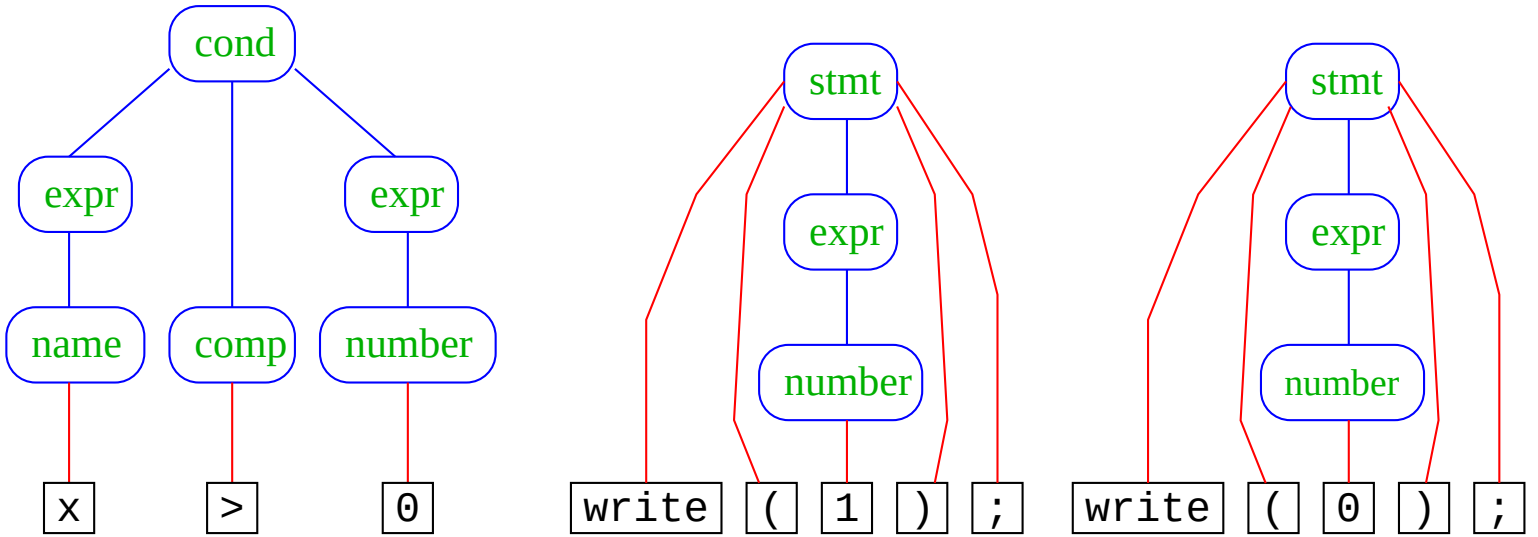
Puh!!! Geschafft ...

Beispiel:

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

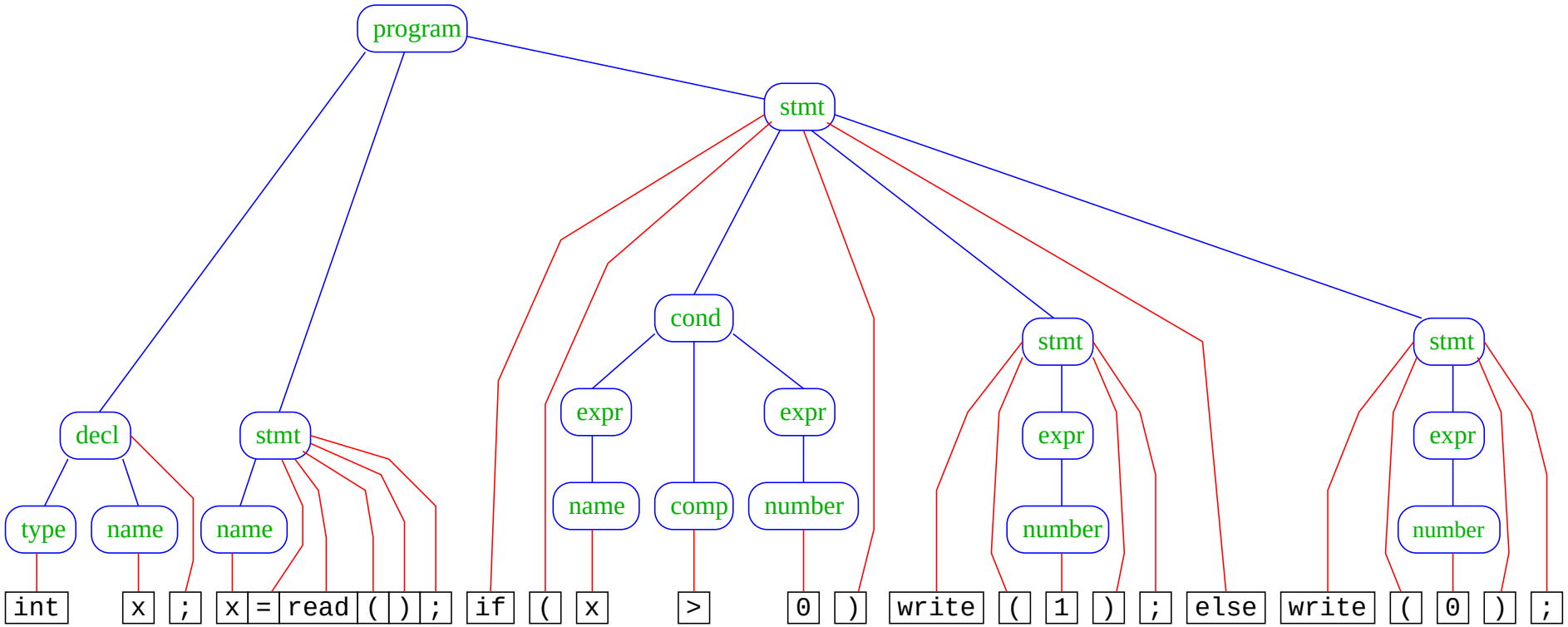
Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch Syntax-Bäume:

Syntax-Bäume für `x > 0` sowie `write(0);` und `write(1);`



Blätter: Wörter/Tokens

innere Knoten: Namen von Programm-Bestandteilen


program
stmt
decl
stmt
cond
stmt
stmt
type
name
name
expr
expr
expr
expr
name
comp
number
number
number
int
x
;
x
=
read
(
)
;
if
(
x
>
0
)
write
(
1
)
;
else
write
(
0
)
;

Bemerkungen:

- Die vorgestellte Methode der Beschreibung von Syntax heißt EBNF-Notation (Extended Backus Naur Form Notation).
- Ein anderer Name dafür ist erweiterte kontextfreie Grammatik (↑Linguistik, Automatentheorie).
- Linke Seiten von Regeln heißen auch Nicht-Terminale.
- Tokens heißen auch Terminale.

Noam Chomsky, MIT

John Backus, IBM (Erfinder von Fortran)

Achtung:

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

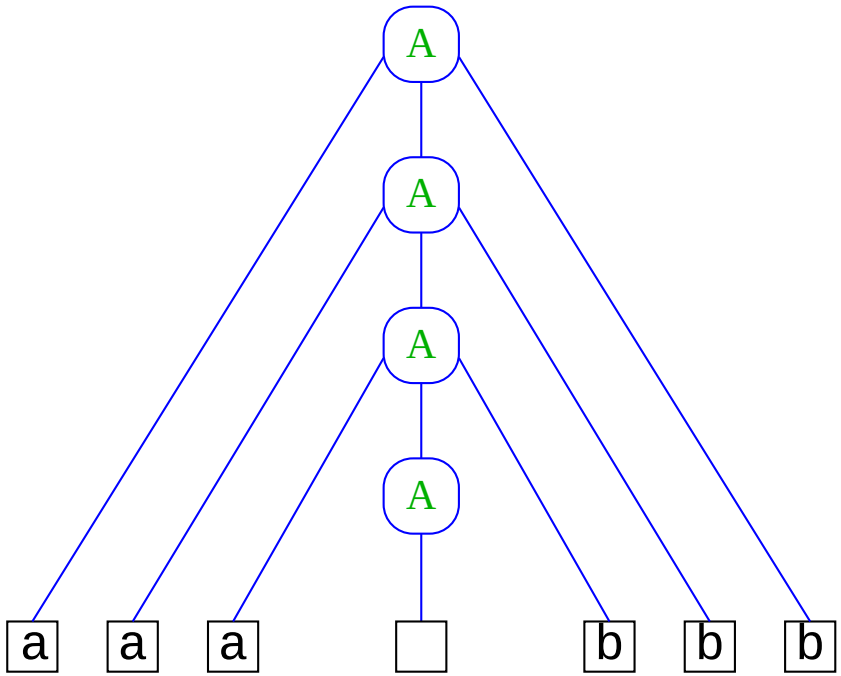
Beispiel:

$$\mathcal{L} = \{\varepsilon, \mathtt{ab}, \mathtt{aabb}, \mathtt{aaabbb}, \ldots\}$$

lässt sich mithilfe einer Grammatik beschreiben:

$$\mathtt{A} \quad ::= \quad (\ \mathtt{a}\ \mathtt{A}\ \mathtt{b}\ )?$$

Syntax-Baum für das Wort `aaabbb` :



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