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 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.

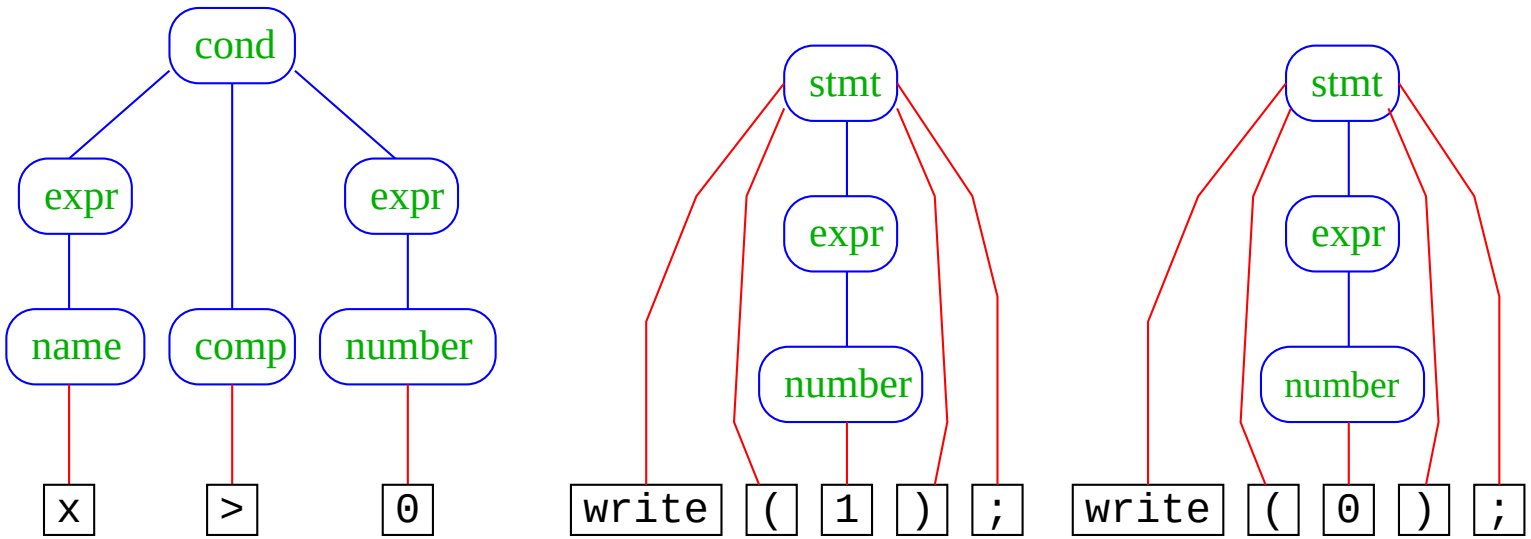
Puh!!! Geschafft ...

Beispiel:

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

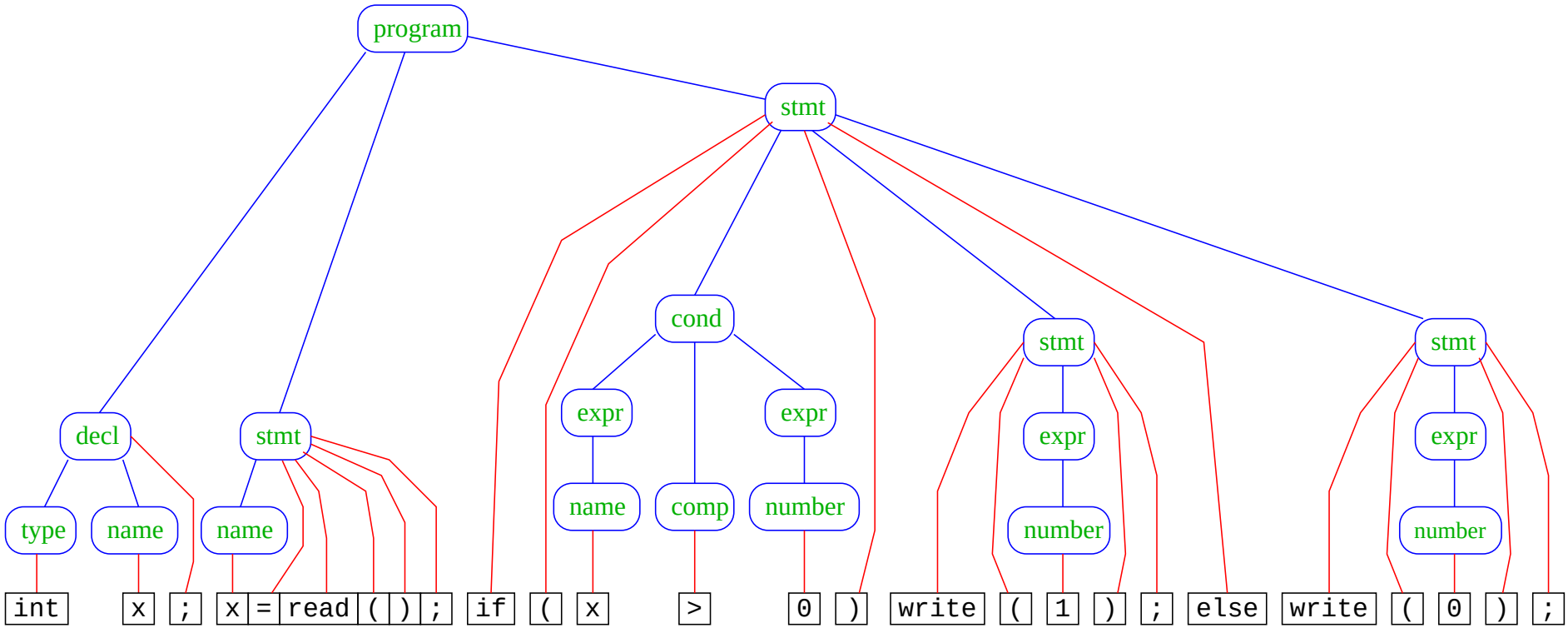
Die hierarchische Untergliederung von Programm-Bestandteilen veranschaulichen wir durch Syntax-Bäume:

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
number
name
comp
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.

Achtung:

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

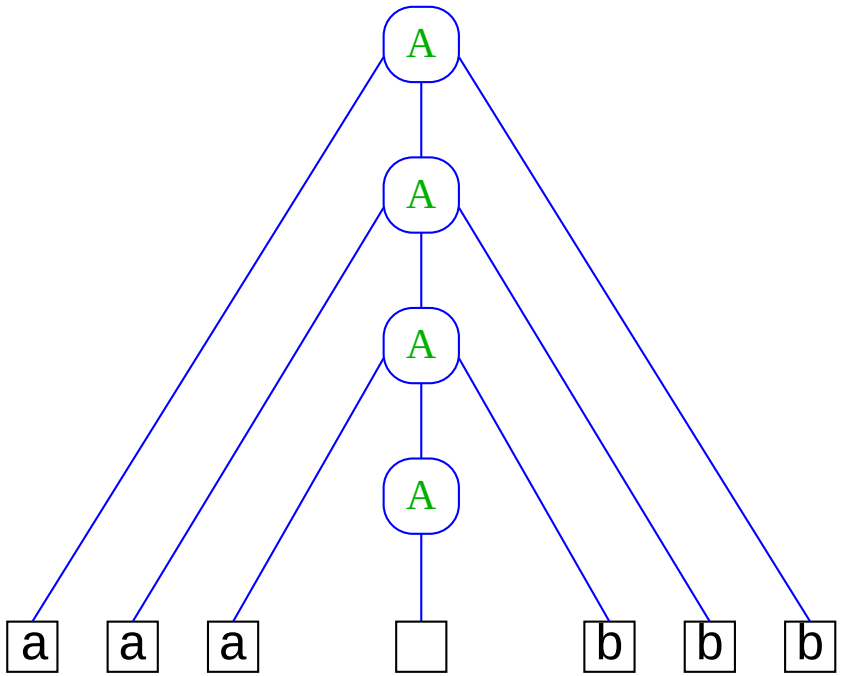
Beispiel:

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

lässt sich mithilfe einer Grammatik beschreiben:

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

Syntax-Baum für das Wort `aaabbb` :



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

Weiteres Beispiel:

$\mathcal{L}$ = alle Worte mit gleich vielen a's und b's

Zugehörige Grammatik:

$$
\begin{array}{lcl}
\mathrm{S} & ::= & (\ \mathtt{b}\ \mathrm{A} \mid \mathtt{a}\ \mathrm{B}\ )^* \\
\mathrm{A} & ::= & (\ \mathtt{b}\ \mathrm{A}\ \mathrm{A} \mid \mathtt{a}\ ) \\
\mathrm{B} & ::= & (\ \mathtt{a}\ \mathrm{B}\ \mathrm{B} \mid \mathtt{b}\ )
\end{array}
$$

Syntax-Baum für das Wort `aababbba` :