Programming Paradigms

2nd Lecture

Prof. Janis Voigtländer

University of Duisburg-Essen

Summer Term 2017

Wadler's Law of Language Design:

In any language design, the total time spent discussing a feature in this list is proportional to two raised to the power of its position.

0. Semantics
1. Syntax
2. Lexical syntax
3. Lexical syntax of comments

Philip Wadler, 1992
(designer of Haskell, Java Generics, Links)

Describing syntax

We will discuss syntax description here:

- ... as a safeguard against ambiguities.
- ... as a basis for implementation (compilers etc., not much in this lecture) and general semantics/verification (next topic).

We will do it in a relatively pragmatic way, without foundational concepts as in the lecture "Automaten und Formale Sprachen".

Describing syntax

Part of a C program:

```
int fib(int n) {
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) { z=x; x=y; y=z+y; }
  return y; }
```

Relevant questions:

- From which "building blocks" is a program assembled?
- By which rules are these systematically combined?
- How can we formally describe (and yet intuitively understand) all that?
- How can a compiler, given such a description, determine whether a program is legal at all?

Expressed more pragmatically:

We need a way of systematically distinguishing appropriate C pieces like this:

```
int fib(int n) {
  int x=1, y=1, i, z;
  if (n>=2)
    for (i=1; i<n; i=i+1) { z=x; x=y; y=z+y; }
  return y; }
```

from “arbitrary” character sequences like this:

```
int fib(int n) {
  int x=1, y=1, i, z;
  if (n>=2; i=i+1)
    for (i=1; i<n) { z=x; x=y; y=z+y; }
  return y; }
```

Clarifying a few notions – Definitions

Alphabet: a non-empty, finite set Σ;
elements are called symbols

Word: a finite sequence of symbols from Σ

- empty word: sequence of length 0; notation: ε

Σ^*: set of all words (over alphabet Σ)

- $\varepsilon \in \Sigma^*$
- if $w \in \Sigma^*$ and $\sigma \in \Sigma$, then $w\sigma \in \Sigma^*$
- no further words in Σ^*

Concatenation: operation that puts two words together

Language: subset of Σ^*

Meta language: language for describing another language;
usually over different alphabets

Object language: language described by another language

Clarifying a few notions – Examples

Alphabet: $\Sigma = \{a, b, c\}$

Word: $abcca$

Σ^*: $\{\varepsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, \dots\}$

Concatenation: $ab \cdot cca = abcca$

Language: $\{a^n bc^n \mid n \geq 0\}$

Meta language: ???

Set notation as meta language

- It seems reasonable to employ the usual set notation from mathematics.
- Works without problems for finite languages:
  - $\emptyset$, $\{\}$
  - $\{\varepsilon\}$
  - $\{b, abc, aabcc, aaabccc\}$
- Still works well for certain infinite languages:
  - $\{a^n bc^n \mid n \geq 0\}$, where $a^n = \underbrace{a \cdots a}_{n\times}$
  - $\{(ab)^n c^m b^n c^k \mid n, m \geq 0, k \geq 1\}$,
    where $(ab)^n$ is defined by n-fold concatenation
- But already problematic for examples like the language of all "well-bracketed expressions" over $\Sigma = \{\,[\,,\,]\,\}$:
  - $\{\varepsilon, [\,], [\,][\,], [[\,]], [\,][\,][\,], [[\,]][\,], [\,][[\,]], [[[\,]]], \ldots\}$

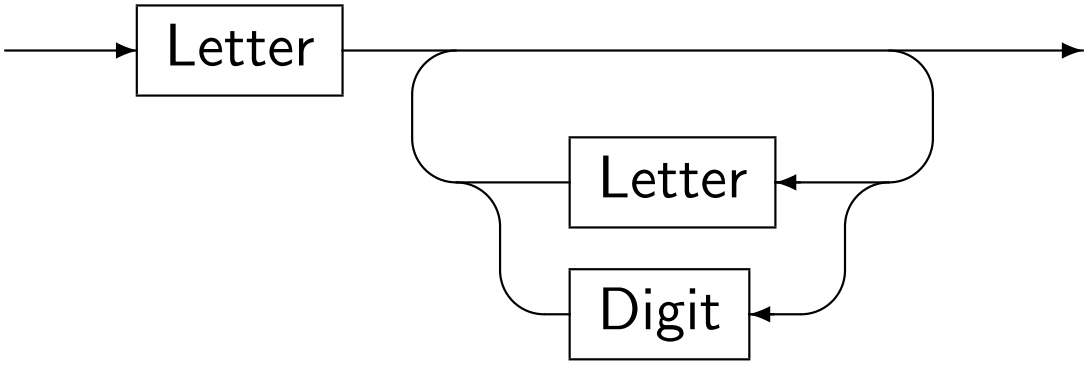
Formal description of syntax

Formal language theory provides different methods:

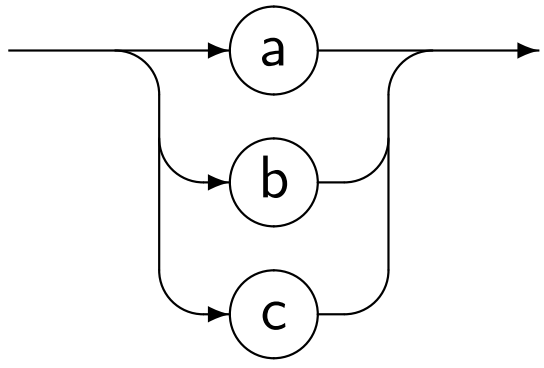
- finite automata
- formal grammars
- syntax diagrams

A simple example:

Ident



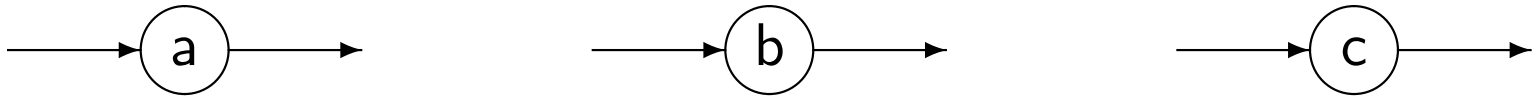
Letter



Syntax diagrams

General build-up:

- Ovals:



- Rectangles:

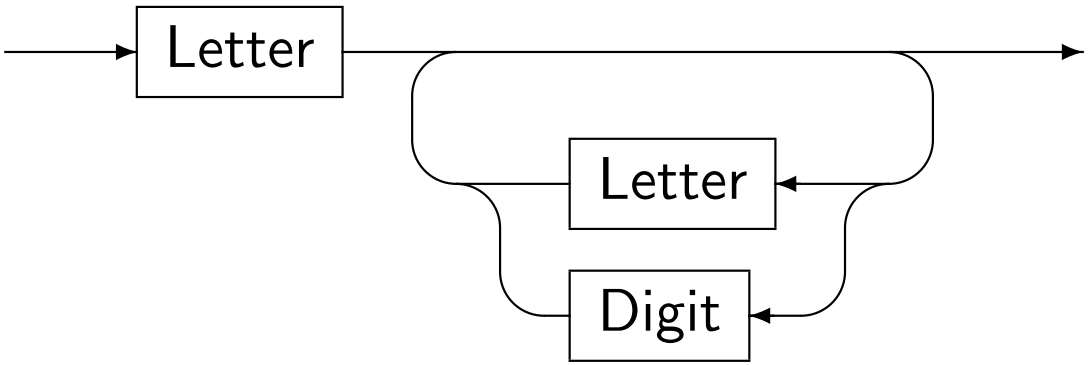
- Connections (with arrows):
  - direct lines
  - branchings
  - join points
  - no crossings

Syntax diagrams

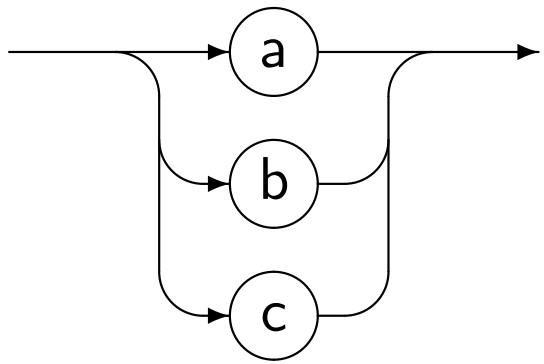
In a system of syntax diagrams:

- Every syntax diagram has a unique name, exactly one start point and exactly one end point.
- Every rectangle is labeled with the name of a syntax diagram.
- Every oval is labeled with a symbol (of the object language).
- Every rectangle/oval has exactly one entry and exactly one exit.

Ident



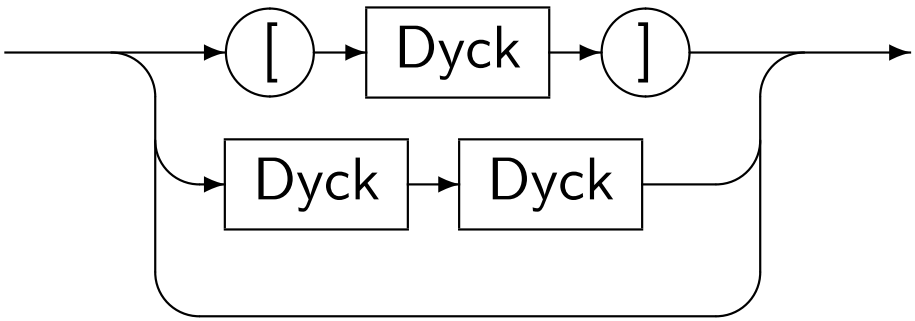
Letter



Syntax diagrams – Example

The previously problematic language of "well-bracketed expressions" $\{\varepsilon, [], [][], [[]], [][][], [[]][], [][[]], \ldots\}$ is now described as:

Dyck



Intuition: In every non-empty well-bracketed expression, for the (necessarily existing) opening bracket at the beginning, there is (exactly) one matching closing bracket:

- either at the very end; then between them there is also a well-bracketed expression;
- or somewhere in the middle; then up to that point there is a well-bracketed expression, and the same holds for the whole rest.

Syntax diagrams – Execution

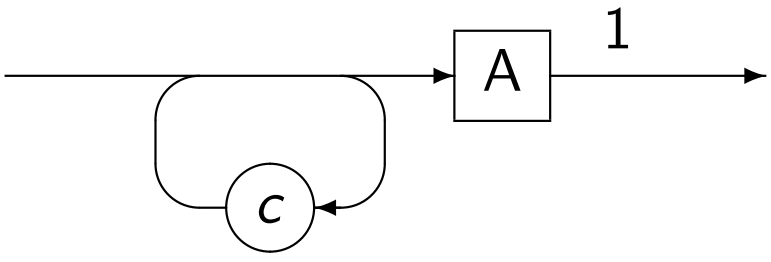
Algorithm for generating words:

1. Put a unique mark (called *return address*) on exit of each rectangle.
2. Start with an initially empty *stack*.
3. Begin at the entry of the *start diagram* (needs to be given in description).
4. Follow the connections along a legal path (not in opposite direction of arrows etc.).
   - If exit of a syntax diagram reached, continue with 5.
   - If oval reached, record inscribed symbol, and continue with 4.
   - If rectangle reached (inscribed with a syntax diagram name):
     4.1 put copy of its return address onto the stack, then
     4.2 continue with 4., at entry of the syntax diagram named.
5. - If stack not empty:
     5.1 take return address *adr* from top of stack, then
     5.2 continue with 4., at corresponding place in the system.
   - If stack empty (and necessarily have reached the exit of the start diagram), generation is finished.

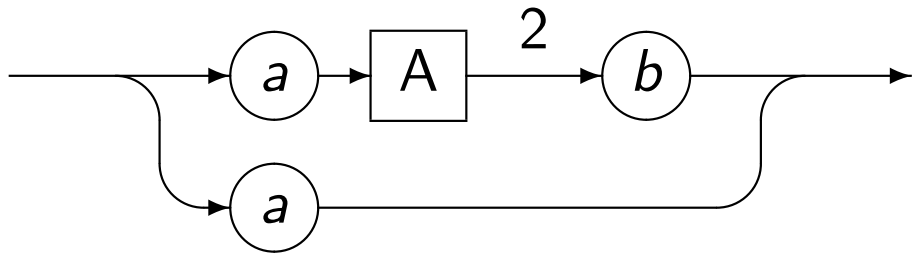
Syntax diagrams – Execution example

Start diagram: **S**

S



A



word	stack
cc	1
cca	21
ccaa	221
ccaaa	21
ccaaab	1
ccaaabb	–
ccaaabb	–

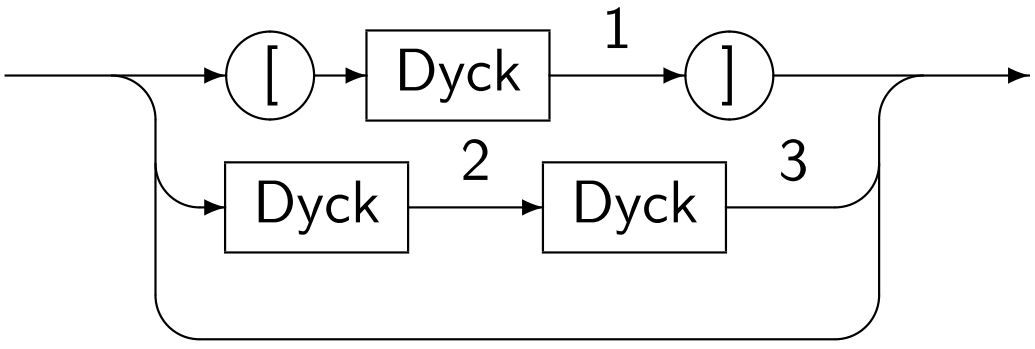
Recording a protocol:

- On stack, return addresses (recall: uniquely assigned to rectangles in the system).
- Record current word and stack content before entering a new syntax diagram.
- Record current word and stack content after exiting a syntax diagram.

Execution of syntax diagrams – Another example

Start diagram: **Dyck**

Dyck



word	stack
ε	2
[	12
[	212
[	12
[	312
[	12
[	2
[]	–
[]	3
[]	–
[]	–

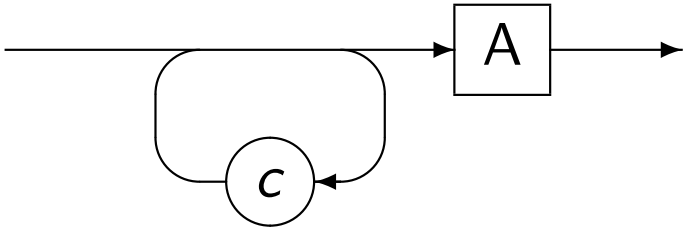
Observations:

- This is certainly not the shortest execution for [].
- Word and stack content alone do not uniquely determine the current state of execution.

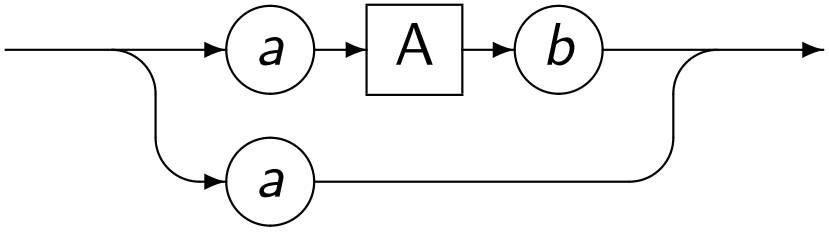
Subtlety: generation vs. recognition

Start diagram: **S**

S



A

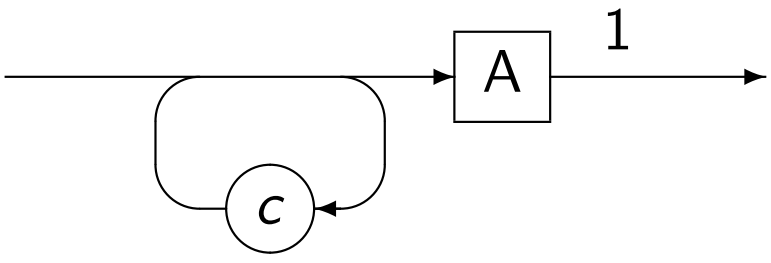


- some generated words: *a*, *ca*, *cca*, *aab*, *ccca*, *caab*, *cccca*, *ccaab*, *aaabb*, *ccccca*, *cccaab*, *caaabb*, *cccccca*, *ccccaab*, ...
- overall generated language in set notation: $\{c^n a^{m+1} b^m \mid n, m \geq 0\}$
- different, but related problem: decide/recognize whether a given word can be generated by a given SD system

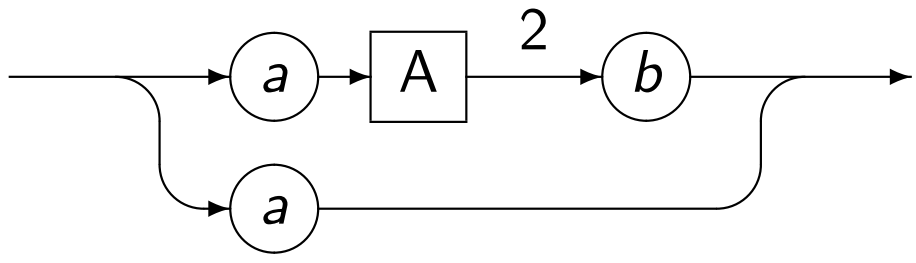
Subtlety: generation vs. recognition

Start diagram: **S**

S



A



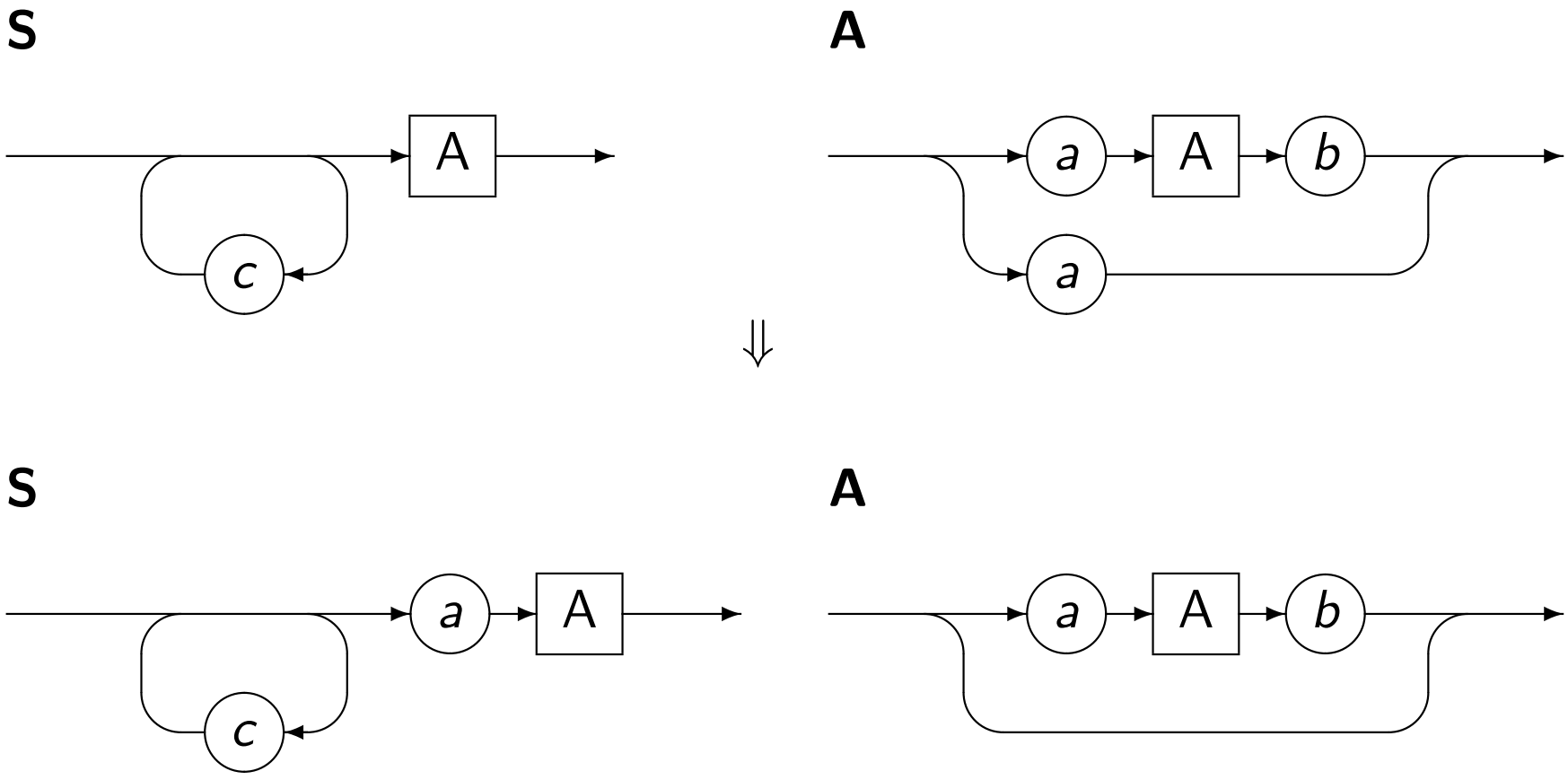
For *caab*, two failing executions, but also one successful:

word	stack
c	1
ca	–

word	stack
c	1
ca	21
caa	221

word	stack
c	1
ca	21
caa	1
caab	–
caab	–

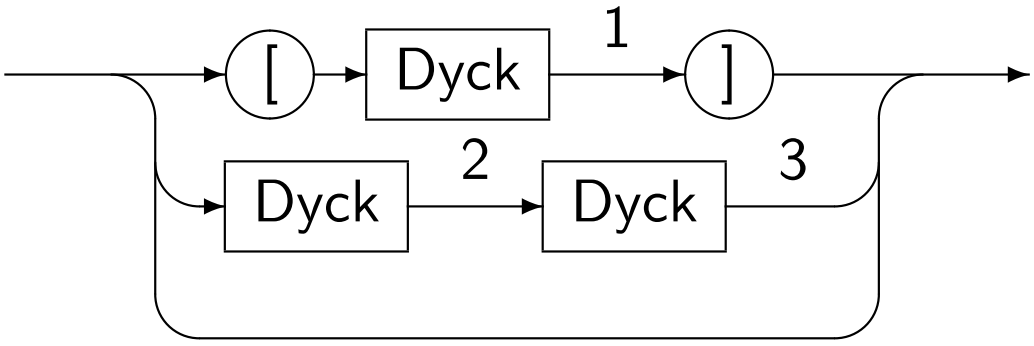
Manipulating syntax diagrams for easier recognition



- Recognize the same language $\{c^n a^{m+1} b^m \mid n, m \geq 0\}$.
- The second system enables deterministic recognition with always only one symbol "look-ahead".

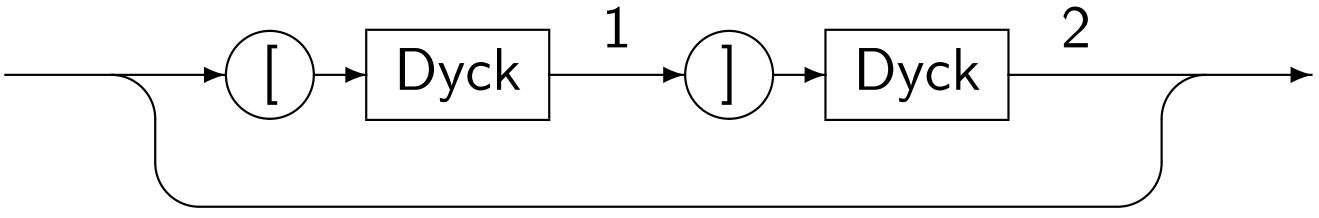
“Analogously”

Dyck



⇓

Dyck



word	stack
ε	2
[	12
[	212
[	12
[	312
[	12
[	2
[]	–
[]	3
[]	–
[]	–

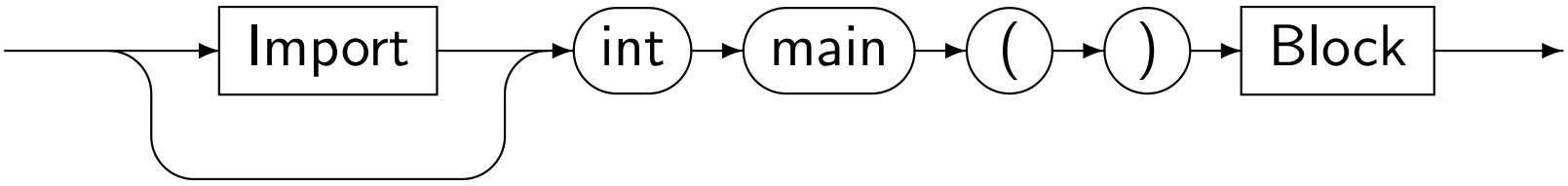
word	stack
[	1
[	–
[]	2
[]	–
[]	–

It can be proved that both diagrams specify the same language.

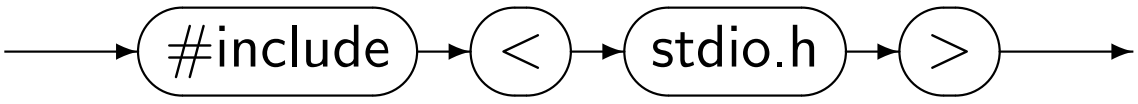
Syntax of C, top-down

Global view of simplified C programs

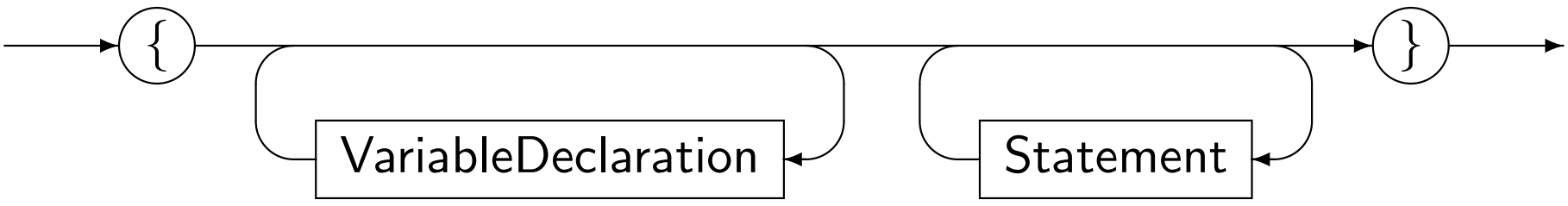
Program



Import

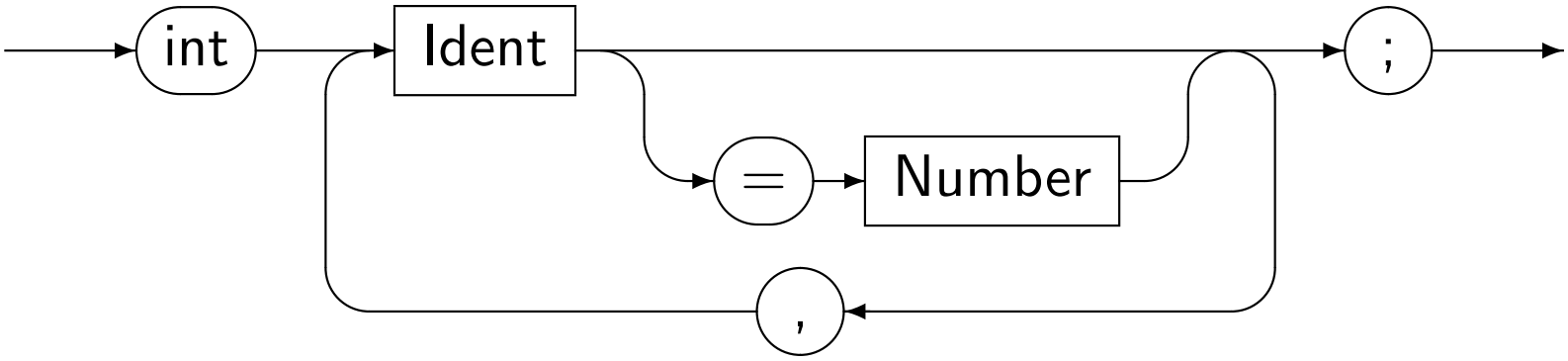


Block



Declaring variables

VariableDeclaration



Important, but not expressible via syntax diagrams:

- no double declarations of same identifier
- no use of undeclared variables in program

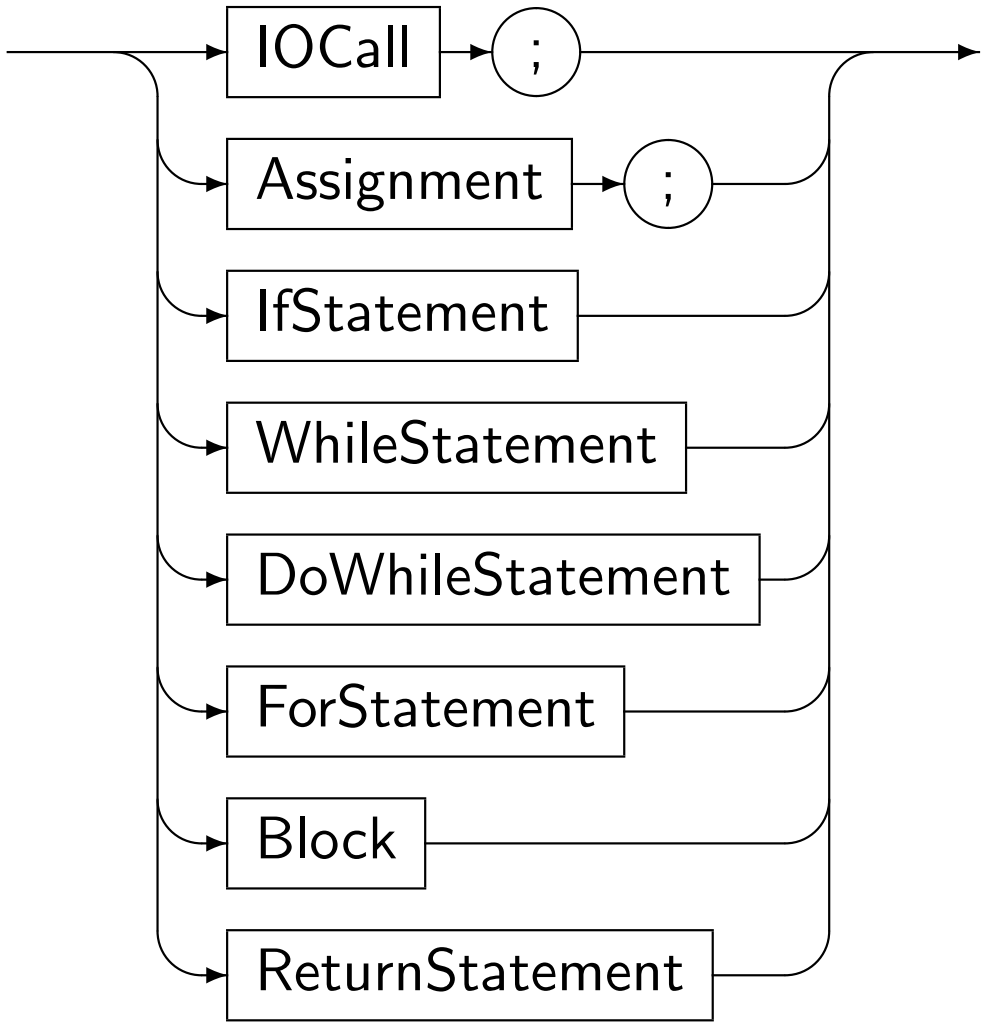
Actually doing stuff

```
#include <stdio.h>

int main()
{  int n,s,i;
   scanf("%d",&n);
   s=0;
   i=1;
   while (i<=n)
      {  s=s+i*i;
         i=i+1;
      }
   printf("%d",s);
   return 0;
}
```

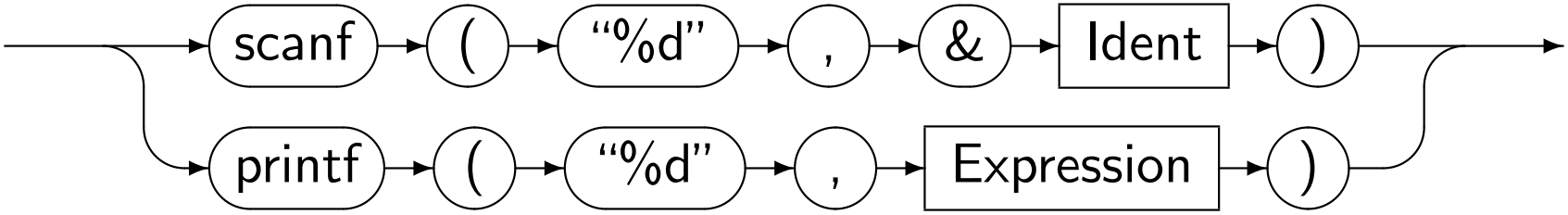
Subset of all possible statements

Statement

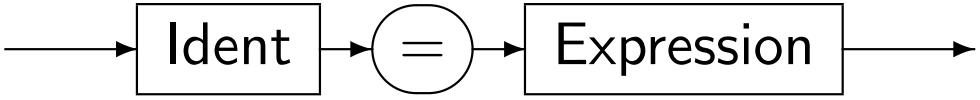


Input and output, variable assignment, conditionals

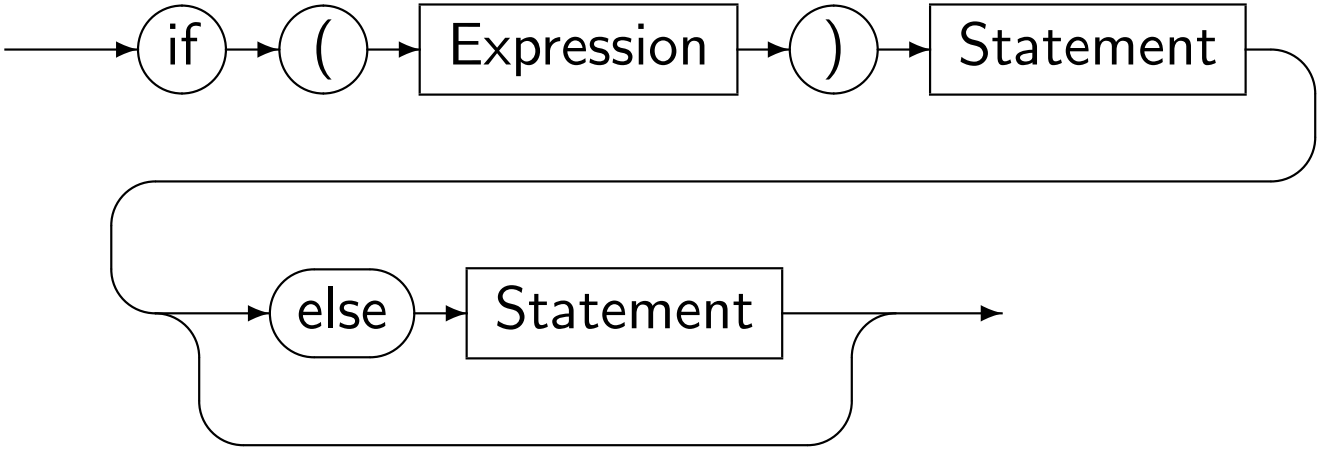
IOCall



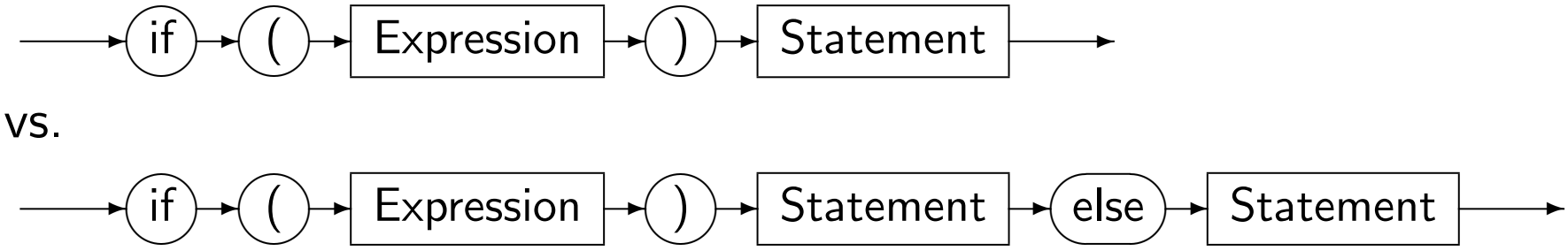
Assignment

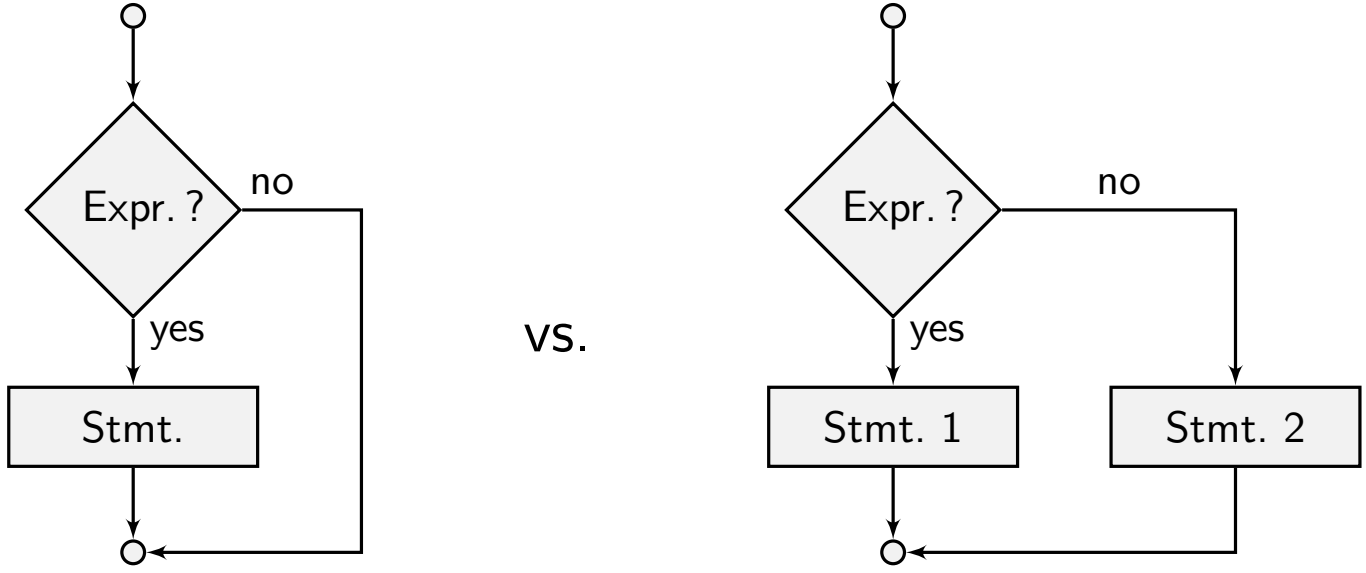


IfStatement



Optional “else” (remember when we talk Haskell)



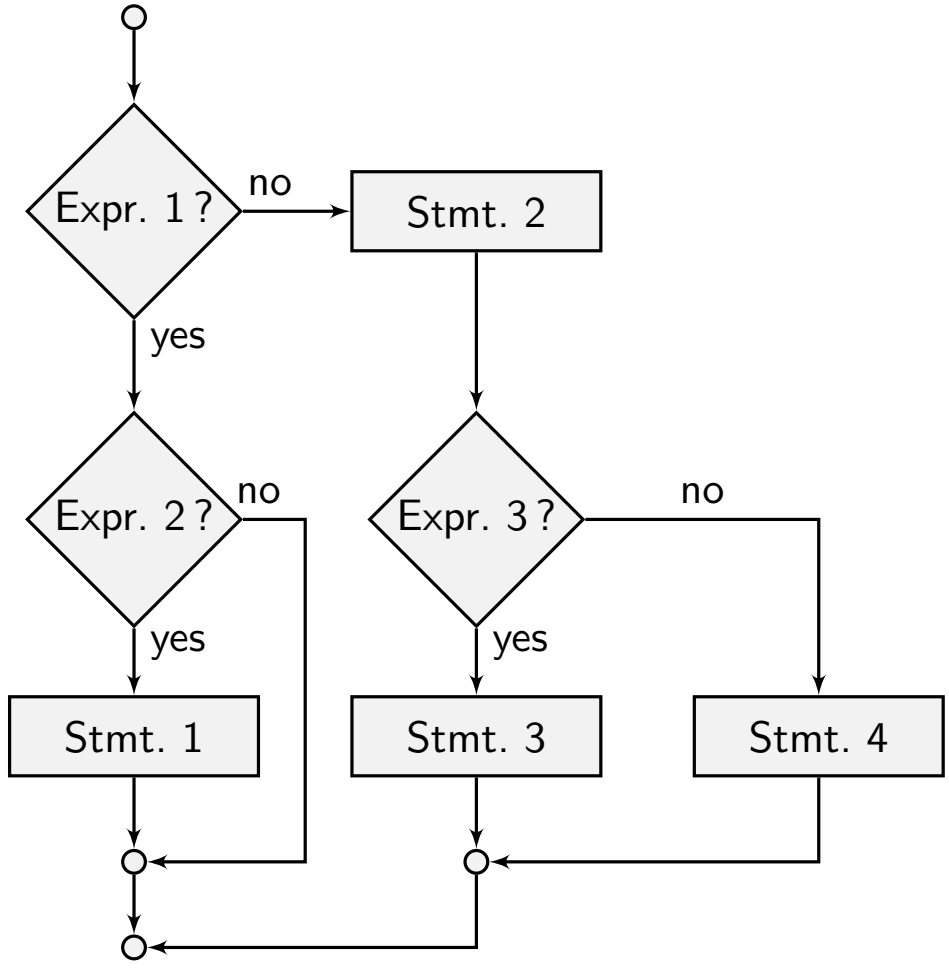


- Of course, statements could be blocks, hence contain several further statements (including another **if**).

Nested conditionals

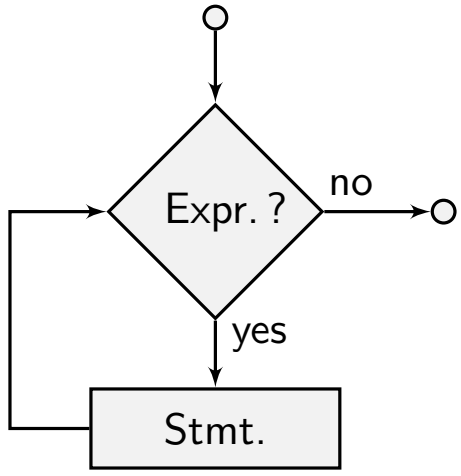
```
if (Expr. 1) {if (Expr. 2) Stmt. 1}
 else {
    Stmt. 2
    if (Expr. 3) Stmt. 3
    else Stmt. 4
 }
```

Without the first {...}-pair, would have been difficult to interpret it this way!



Loops

WhileStatement



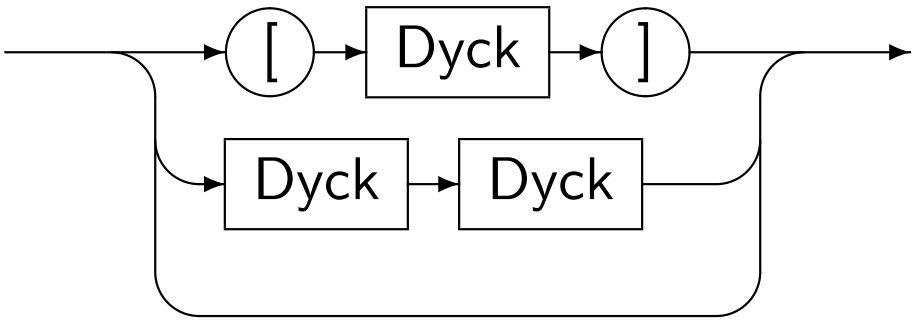
- For simplicity, no consideration of other loops here.

Backus-Naur Form (BNF)

An alternative to syntax diagrams

Instead of a graphical notation . . .

Dyck



. . . a textual notation:

$\langle Dyck \rangle$ ::= ‘[’ $\langle Dyck \rangle$ ‘]’ | $\langle Dyck \rangle$ $\langle Dyck \rangle$ |

Ingredients of such a production:

- a non-terminal, like $\langle Dyck \rangle$, on the left-hand side;
- several alternatives, separated by |, on the right-hand side,
- made up of non-terminals and/or terminals, like ‘[’.

An alternative to syntax diagrams

Interpretation by repeatedly applying productions from left to right, each time choosing any of the available alternatives.

For the example,

$\langle Dyck \rangle$ ::= ‘[’ $\langle Dyck \rangle$ ‘]’ | $\langle Dyck \rangle$ $\langle Dyck \rangle$ |

one possible derivation:

$$\begin{aligned}
\langle Dyck \rangle &\rightarrow \text{‘[’ } \langle Dyck \rangle \text{ ‘]’} \\
&\rightarrow \text{‘[’ } \langle Dyck \rangle \ \langle Dyck \rangle \text{ ‘]’} \\
&\rightarrow \text{‘[’ ‘[’ } \langle Dyck \rangle \text{ ‘]’ } \langle Dyck \rangle \text{ ‘]’} \\
&\rightarrow \text{‘[’ ‘[’ ‘]’ } \langle Dyck \rangle \text{ ‘]’} \\
&\rightarrow \text{‘[’ ‘[’ ‘]’ ‘]’} = \texttt{[[]]}
\end{aligned}$$

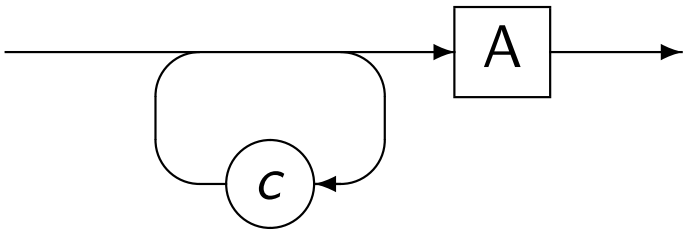
or another (of many):

$$\begin{aligned}
\langle Dyck \rangle &\rightarrow \langle Dyck \rangle \ \langle Dyck \rangle \\
&\rightarrow \langle Dyck \rangle \\
&\rightarrow \text{‘[’ } \langle Dyck \rangle \text{ ‘]’} \\
&\rightarrow \text{‘[’ ‘]’} = \texttt{[]}
\end{aligned}$$

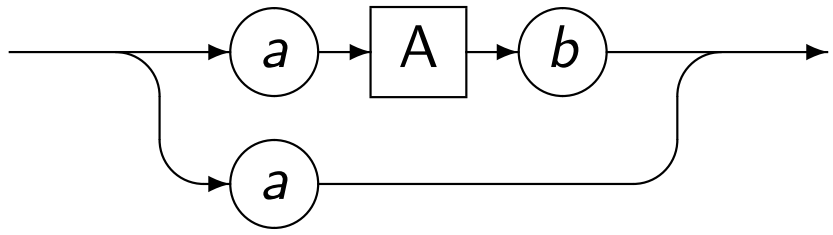
The two mechanisms (SDs and BNF) are equally powerful!

A more involved example

S



A



Need to express the loop indirectly:

$\langle S \rangle$ ::= $\langle Cs \rangle$ $\langle A \rangle$

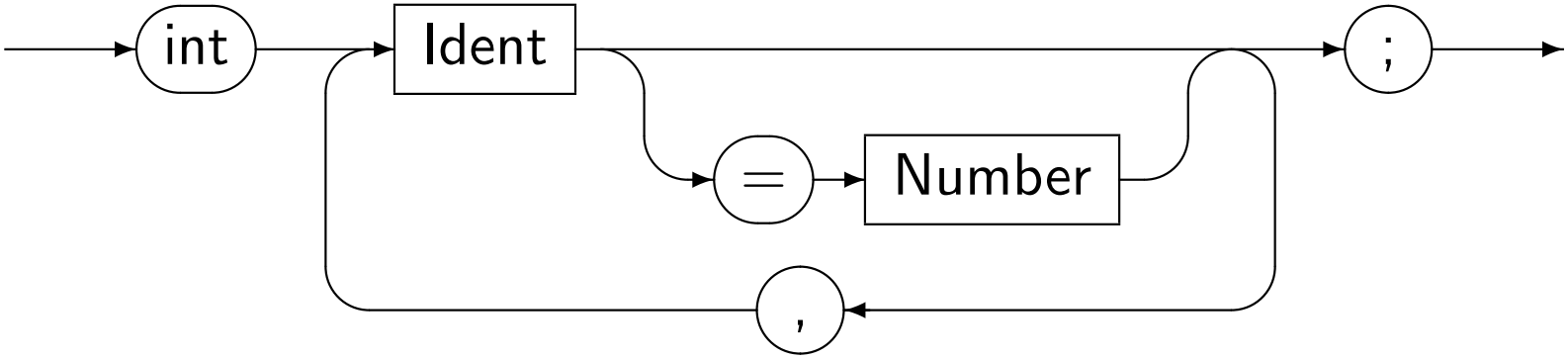
$\langle Cs \rangle$ ::= 'c' $\langle Cs \rangle$ |

While translation from SD to BNF is again very direct for $\langle A \rangle$:

$\langle A \rangle$ ::= 'a' $\langle A \rangle$ 'b' | 'a'

Yet another example

VariableDeclaration



$\Downarrow$

$\langle \mathit{VariableDeclaration} \rangle ::= \text{`}\texttt{int}\text{'}\ \langle \mathit{Items} \rangle\ \text{`;'}$

$\langle \mathit{Items} \rangle ::= \langle \mathit{Item} \rangle\ \text{`,'}\ \langle \mathit{Items} \rangle \mid \langle \mathit{Item} \rangle$

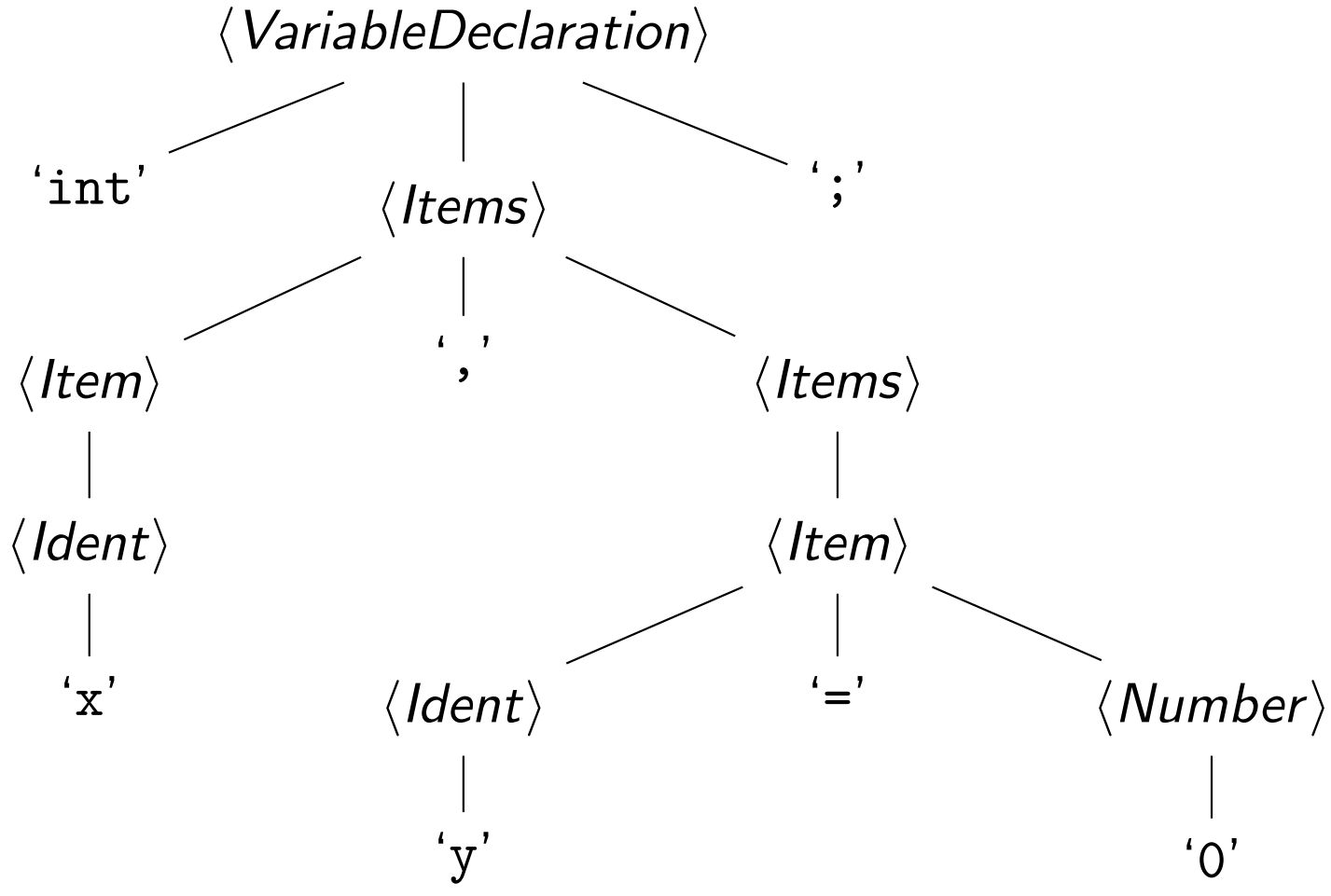
$\langle \mathit{Item} \rangle ::= \langle \mathit{Ident} \rangle \mid \langle \mathit{Ident} \rangle\ \text{`='}\ \langle \mathit{Number} \rangle$

$\langle \mathit{Ident} \rangle ::= \ldots$

$\langle \mathit{Number} \rangle ::= \ldots$

Derivation trees (also: parse trees)

Recording derivations in nested rather than linearized fashion:



. . . provides structure for compilers etc., and for verification . . .

The importance of grammar design

Consider the following two BNFs:

$\langle Expression \rangle ::= \langle Term \rangle$ '+' $\langle Expression \rangle \mid \langle Term \rangle$

$\langle Term \rangle ::= \langle Factor \rangle$ '*' $\langle Term \rangle \mid \langle Factor \rangle$

$\langle Factor \rangle ::= \langle Number \rangle \mid \langle Ident \rangle \mid$ '(' $\langle Expression \rangle$ ')'

vs.

$\langle Expression \rangle ::= \langle Factor \rangle$ '*' $\langle Expression \rangle \mid \langle Factor \rangle$

$\langle Factor \rangle ::= \langle Term \rangle$ '+' $\langle Factor \rangle \mid \langle Term \rangle$

$\langle Term \rangle ::= \langle Number \rangle \mid \langle Ident \rangle \mid$ '(' $\langle Expression \rangle$ ')'

The importance of grammar design

Both allow derivation of the concrete expression `2 + 3 * 5`:

$$
\begin{aligned}
\langle \textit{Expression} \rangle &\rightarrow \quad \langle \textit{Term} \rangle \text{ `+' } \langle \textit{Expression} \rangle \\
&\rightarrow^{2} \langle \textit{Factor} \rangle \text{ `+' } \langle \textit{Term} \rangle \\
&\rightarrow^{2} \langle \textit{Number} \rangle \text{ `+' } \langle \textit{Factor} \rangle \text{ `*' } \langle \textit{Term} \rangle \\
&\rightarrow^{6} \text{ `2' `+' `3' `*' `5' } = \texttt{2 + 3 * 5}
\end{aligned}
$$

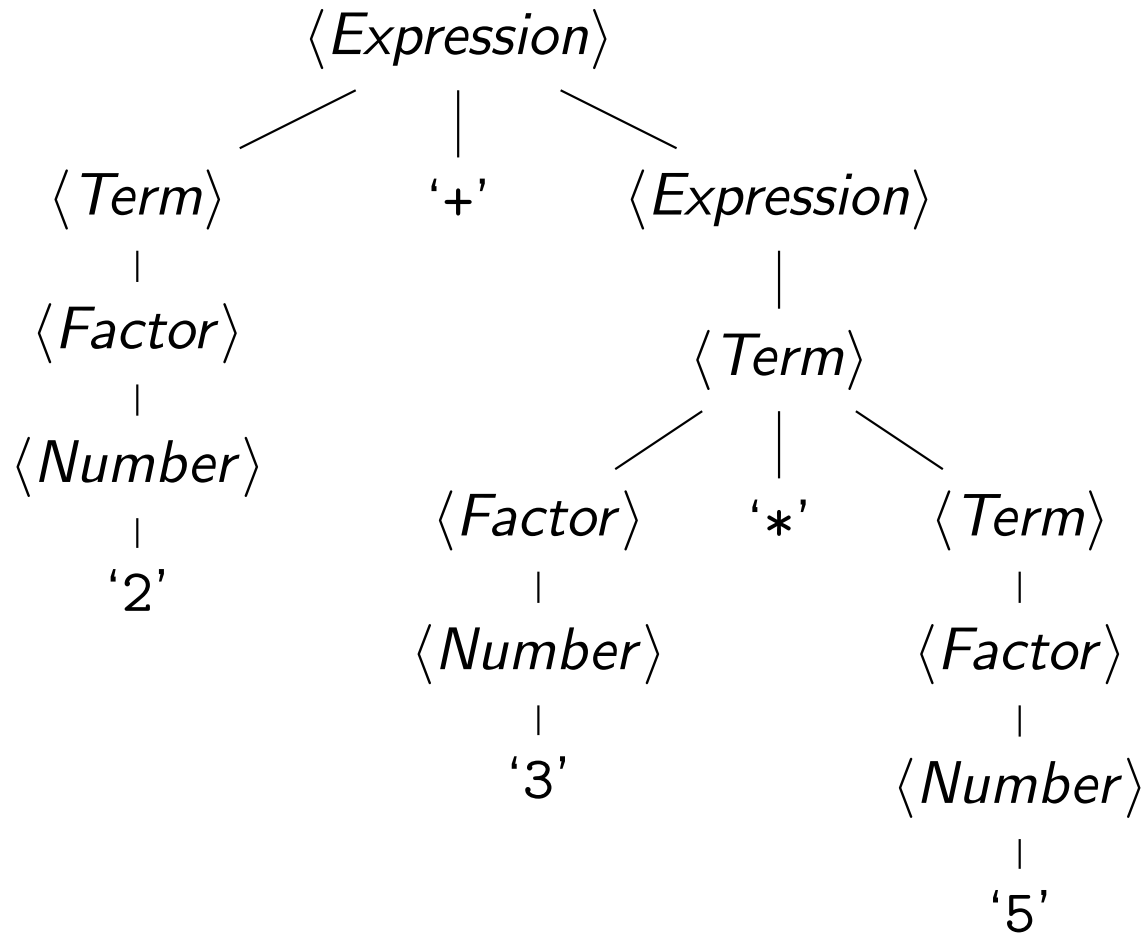
and

$$
\begin{aligned}
\langle \textit{Expression} \rangle &\rightarrow \quad \langle \textit{Factor} \rangle \text{ `*' } \langle \textit{Expression} \rangle \\
&\rightarrow^{2} \langle \textit{Term} \rangle \text{ `+' } \langle \textit{Factor} \rangle \text{ `*' } \langle \textit{Factor} \rangle \\
&\rightarrow^{3} \langle \textit{Number} \rangle \text{ `+' } \langle \textit{Term} \rangle \text{ `*' } \langle \textit{Term} \rangle \\
&\rightarrow^{5} \text{ `2' `+' `3' `*' `5' } = \texttt{2 + 3 * 5}
\end{aligned}
$$

What could there be to not like about either of these?

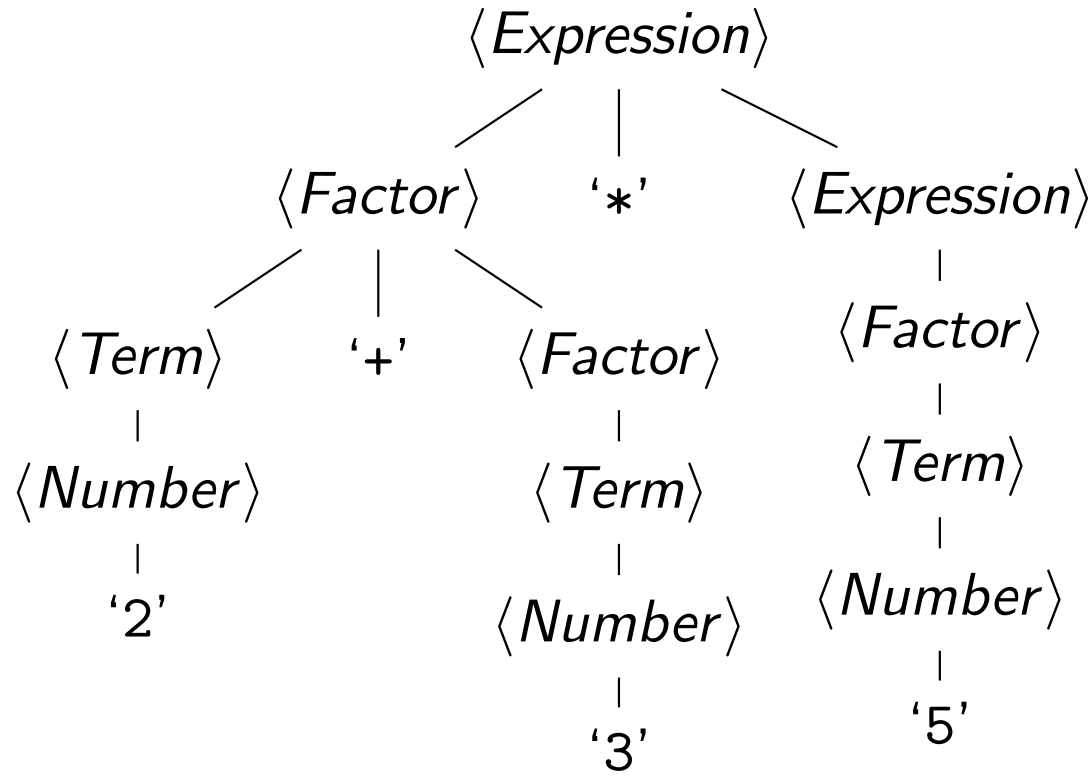
The importance of grammar design

Let's take a look at the derivation trees, on the one hand:



The importance of grammar design

. . . and on the other:



Which one should we prefer, semantically?