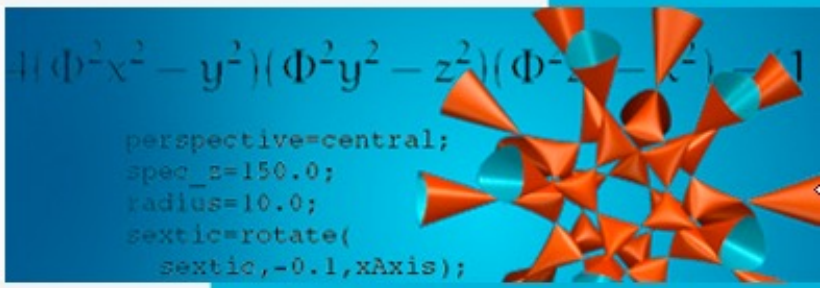
Philipps Universität Marburg

Declarative programming

Summer semester 2024

Prof. Christoph Bockisch, Steffen Dick
(Programming languages and tools)

Imke Gürtler, Daniel Hinkelmann, Aaron Schafberg, Stefan Störmer



[Script 6, 7, 8]

Meaning of BSL

- Previously: informal definition of the meaning of BSL programs
  - Abstract syntax: "literals", "function calls", etc.
  - Textual description of the reduction steps
- No definition of the meaning of struct yet

- From now on: **formal definition** of the meaning of BSL programs
  - Concrete syntax
  - Formal semantics based on syntax

Why formal definition?

- Clear definition
- For the computer
  - Enables only execution
  - Standardization
  - Analysis of programs
- For programmers
  - Correct prediction of the result of a program possible
  - Systematic learning of all programming language constructs possible

Syntax

- Abstract syntax
  - Program consists of definitions and expressions
  - Expression consists of literals and function calls
  - Etc.
- Concrete syntax
  - How exactly do you write down a number?
  - How can definitions be distinguished from expressions in a sequence?
- Syntax definition through grammar
  - Recognize whether a text corresponds to the syntax
  - Breakdown of a valid text into language elements

Semantics

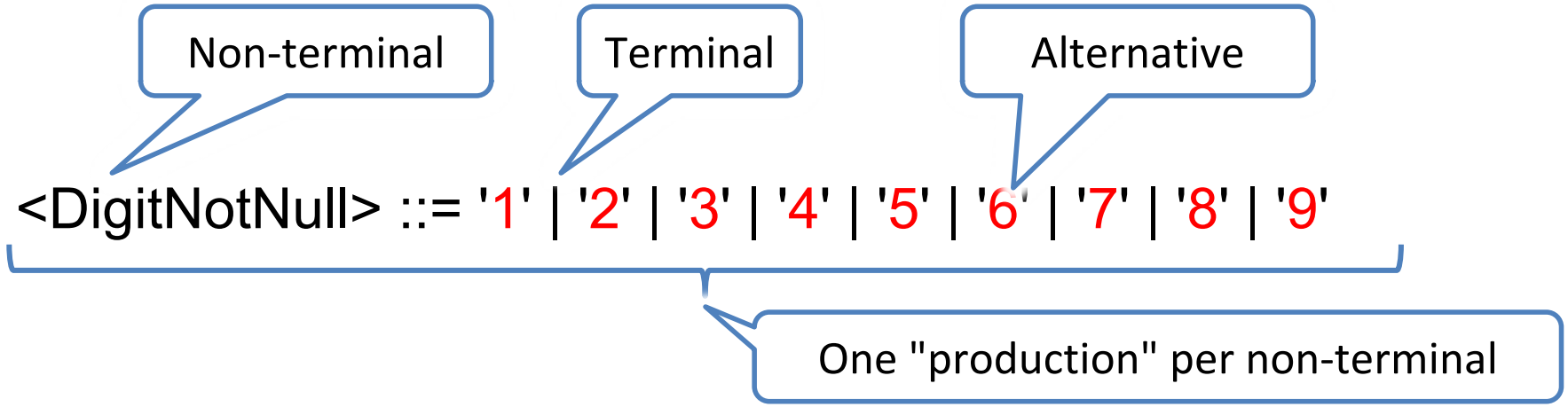
- Rules for the execution/evaluation of a program
- Through reduction steps
- Formal definition: "reduction semantics" (or "structural operational semantics" or "Plotkin semantics")
- Making logical statements from which formal proofs can be constructed
- Statements are based on a program structure defined by grammar

Context-free grammars

- Different classes of grammars
  - Expressive power
  - Effort for the decomposition of a record
- Typically ideal trade-off for programming languages: "context-free grammars"
  - I.e. previously read text does not change the applied grammar
  - State is managed when applying the grammar rules: This makes nesting possible
- Prominent notation for context-free grammars: Extended Backus Naur Form (EBNF)

Grammar for programming languages

- Input: Character stream
- Grammar specifies which characters are permitted and in which order



- <digitNon-zero> accepts a character stream if it consists of exactly one of the digits 1-9

Grammar for programming languages

- Input: Character stream
- Grammar specifies which characters are permitted and in which order

Often an arrow instead:

<digitNotNull> ⟶ '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

- <DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
- <digitNon-zero> accepts a character stream if it consists of exactly one of the digits 1-9

Grammar for programming languages

- Use of non-terminals on the right-hand side of the production

- <digit> ::= '0' | <digitNotNull>

Any character stream that is accepted by production <digitNotNull>.

Grammar for programming languages

- Repetitions

- <integer> ::= <non-zero digit> <digit>*
  | '0'

*: Repeated as often as desired, can also be omitted completely.

- <comma number> ::= <integer> '.' <digit>+

+: Repeated as often as desired, but at least once.

Recognizing words in grammar

- Given
  - An input consisting of a sequence of characters
  - A grammar by rules in EBNF
- Procedure (informal)
  1. The current rule is the start rule
     (by convention the first rule defined)
  2. Apply an appropriate production of the current rule
  3. The characters that correspond to terminals of the production are removed from the input
  4. For each non-terminal in production, apply the procedure from 2. for the corresponding part of the input.
  5. If the input has been processed completely, the input has been accepted

Context-free grammar for numbers

<number> ::= <positiveNumber>
| '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
<commaNumber>
<integer> ::= <non-zero digit> <digit>*
| '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>

Which entries are valid?

a) -87
b) -.65
c) 0
d) -2.09900
e) 007

Context-free grammar for numbers

<number> ::= <positiveNumber>
| '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
<commaNumber>
<integer> ::= <non-zero digit> <digit>*
| '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>

Derivation tree

- Derivation of a non-terminal
  - Selection of an alternative production method
  - Replace all non-terminals with their derivation
- Possible derivations from <digit>
  - '0'
  - '3'

```
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5'
                                | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```

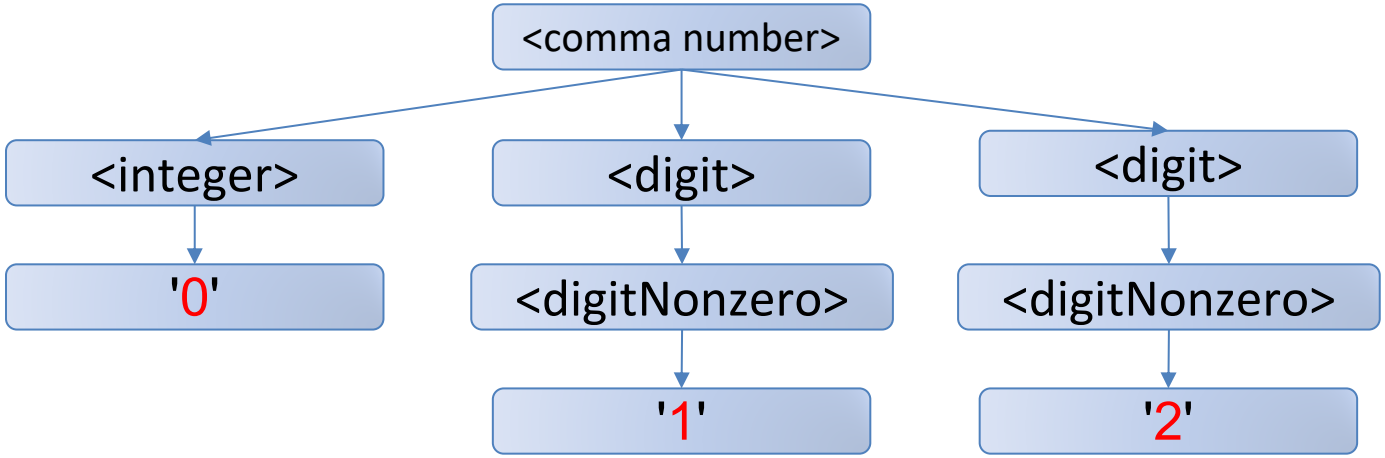
1st alternative

2nd alternative and derivation from <digitNonzero> to 3rd alternative

Derivation tree

- Log which non-terminals were derived
  → Tree

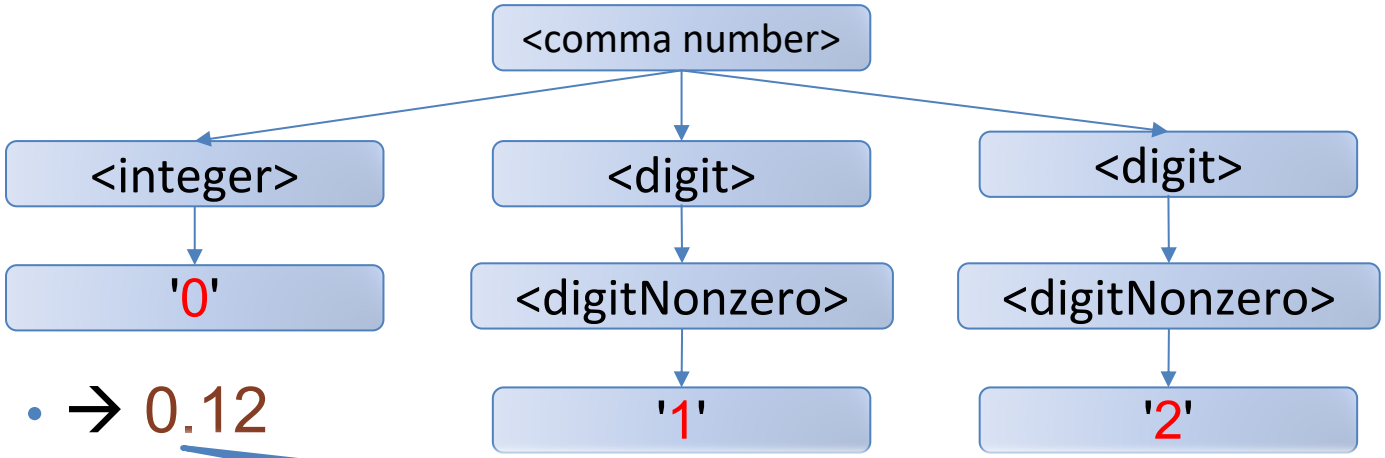
```
<integer> ::= <non-zero digit> <digit>*
                              | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```



Derivation tree

- A derivation tree corresponds to a valid sentence in grammar
- In general: any number of derivation trees
- Theorem can be read from the derivation tree: Leaves from left to right

<comma number> ::= <integer> '.' <digit>+



- → 0.12

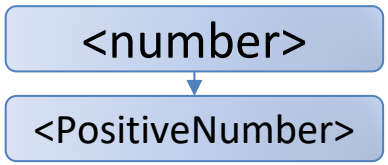
The period is prescribed in the grammar.

Derivation trees

- Generate all valid records
- Conversely: check whether a record is valid
  - Search for a derivation that corresponds to the sentence
  - Starting from start production <number>

- Example 420

No '-': only first alternative possible

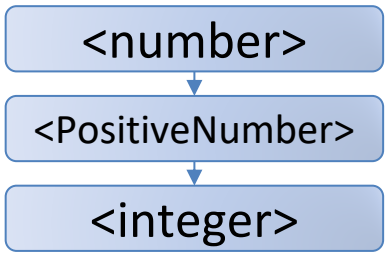



```
<number> ::= <positiveNumber>
                    | '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
                    <commaNumber>
<integer> ::= <non-zero digit> <digit>*
                    | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```

Derivation trees

- Generate all valid records
- Conversely: check whether a record is valid
  - Search for a derivation that corresponds to the sentence
  - Starting from start production <number>
- Example 420

Alternative not yet clear. -> "guess"

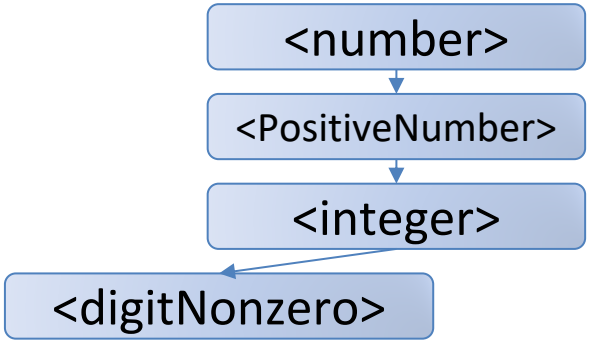



```
<number> ::= <positiveNumber>
                    | '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
                    <commaNumber>
<integer> ::= <non-zero digit> <digit>*
                    | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```

Derivation trees

- Generate all valid records
- Conversely: check whether a record is valid
  - Search for a derivation that corresponds to the sentence
  - Starting from start production <number>
- Example 420

No '0': only first alternative possible

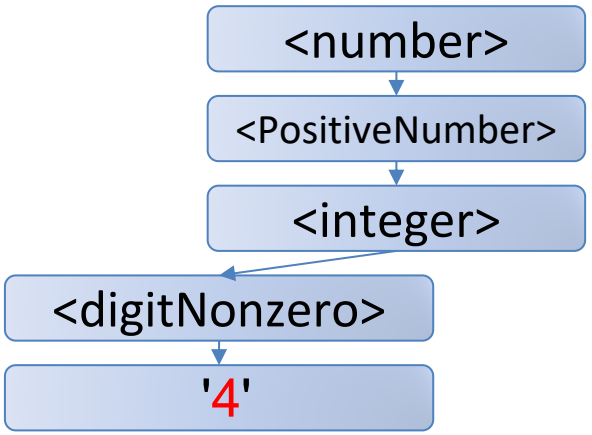



```
<number> ::= <positiveNumber>
                    | '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
                    <commaNumber>
<integer> ::= <non-zero digit> <digit>*
                    | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```

Derivation trees

- Generate all valid records
- Conversely: check whether a record is valid
  - Search for a derivation that corresponds to the sentence
  - Starting from start production <number>
- Example 420

'4': only fourth alternative possible




```
<number> ::= <positiveNumber>
                    | '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
                    <commaNumber>
<integer> ::= <non-zero digit> <digit>*
                    | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```

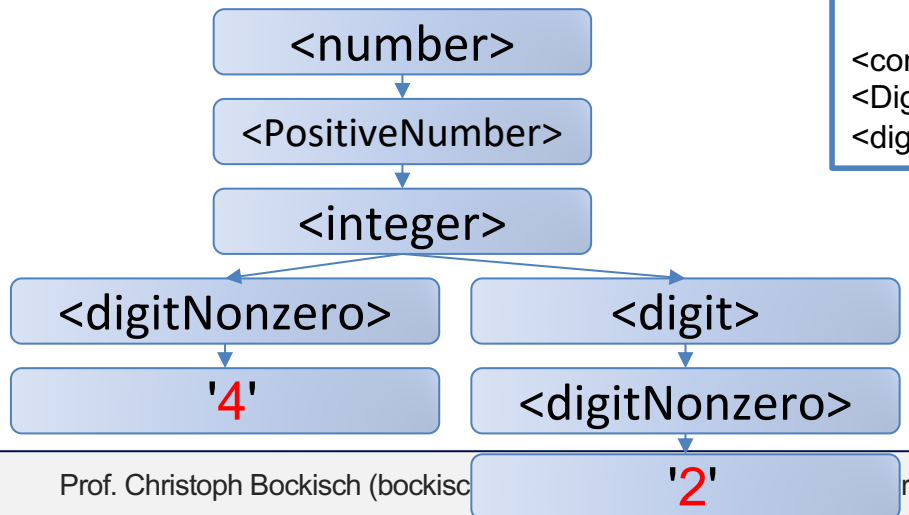
Derivation trees

- Generate all valid records
- Conversely: check whether a record is valid
  - Search for a derivation that corresponds to the sentence
  - Starting from start production <number>

- Example 420

Input not yet finished.

```
<number> ::= <positiveNumber>
                    | '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
                    <commaNumber>
<integer> ::= <non-zero digit> <digit>*
                    | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```



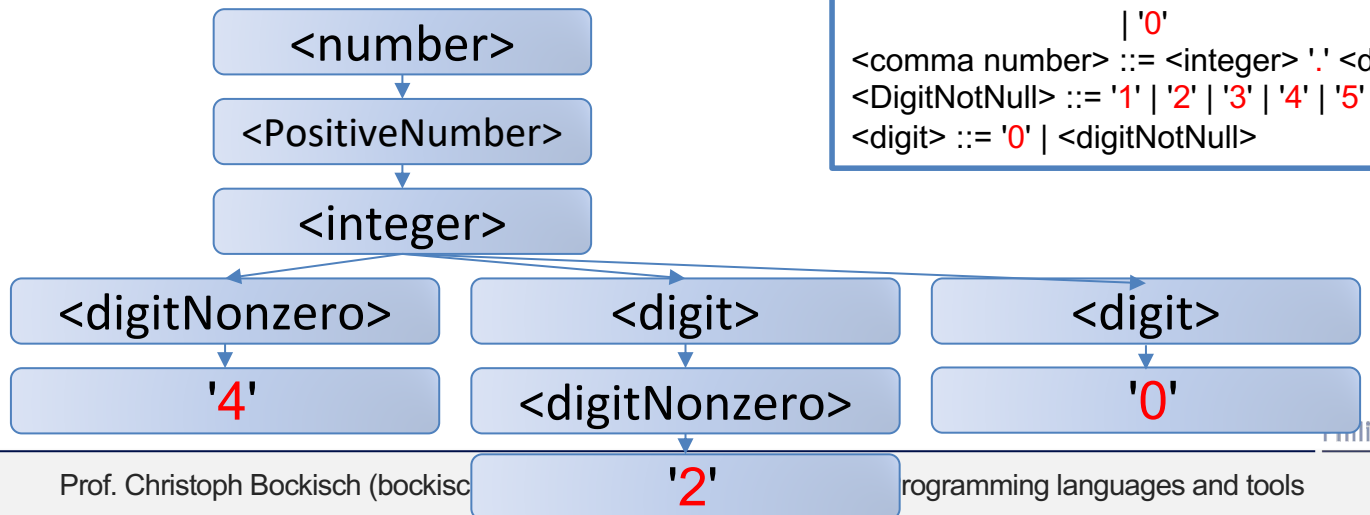
Derivation trees

- Generate all valid records
- Conversely: check whether a record is valid
  - Search for a derivation that corresponds to the sentence
  - Starting from start production <number>

- Example 420

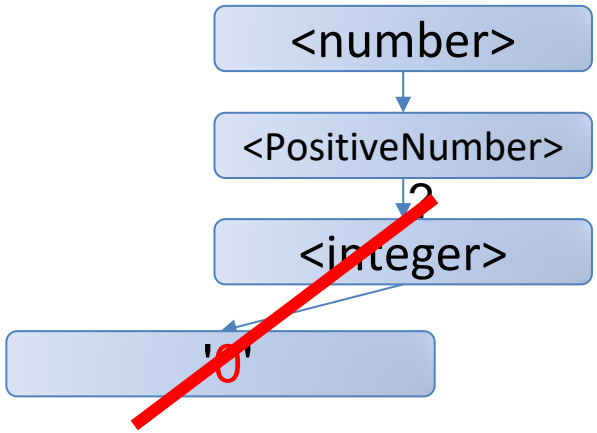
Input at the end.

```
<number> ::= <positiveNumber>
                    | '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
                    <commaNumber>
<integer> ::= <non-zero digit> <digit>*
                    | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```



Derivation trees

- Backtracking when creating the derivation tree
- Example 0.1

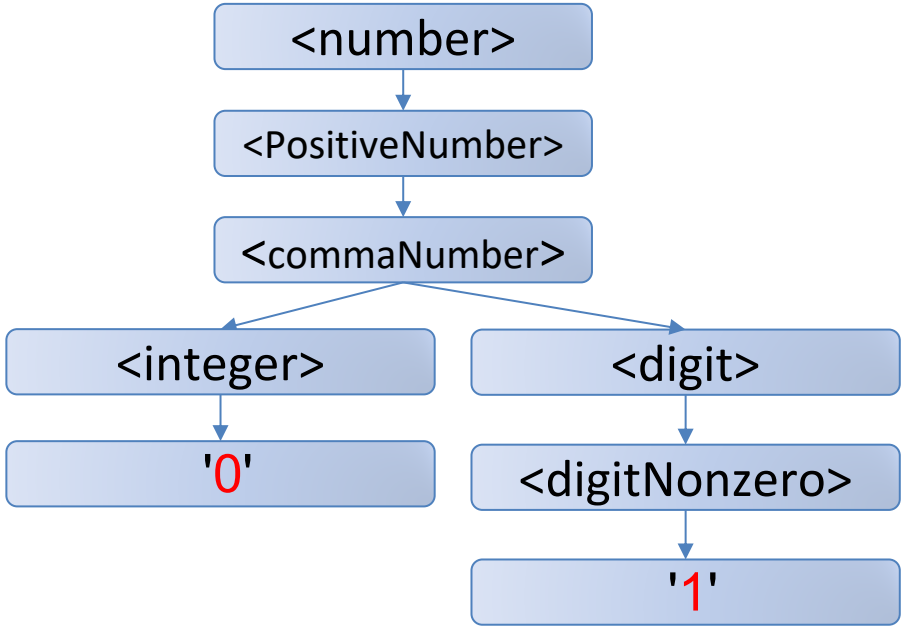



```
<number> ::= <positiveNumber>
                        | '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
                        <commaNumber>
<integer> ::= <non-zero digit> <digit>*
                        | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```

Input not yet empty.
Incorrect derivation selected. Backtracking.

Derivation trees

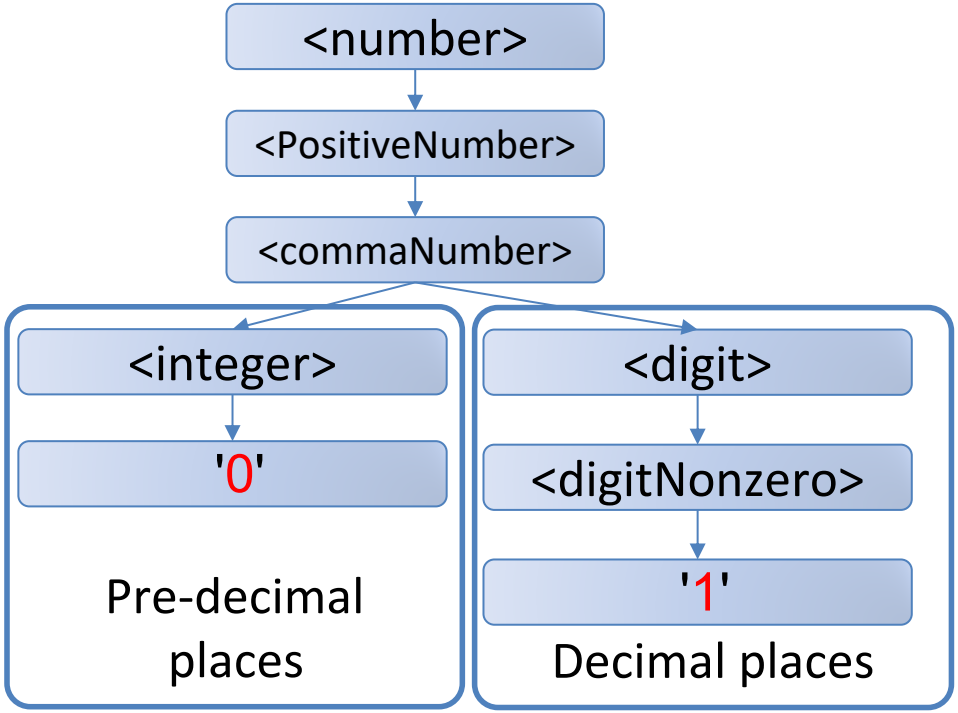
- Backtracking when creating the derivation tree
- Example 0.1




```
<number> ::= <positiveNumber>
                    | '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
                    <commaNumber>
<integer> ::= <non-zero digit> <digit>*
                    | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```

Derivation trees and program structure

- Program structure recognizable in derivation tree
- Example 0.1




```
<number> ::= <positiveNumber>
                    | '-' <PositiveNumber>
<PositiveNumber> ::= <Integer>
                    <commaNumber>
<integer> ::= <non-zero digit> <digit>*
                    | '0'
<comma number> ::= <integer> '.' <digit>+
<DigitNotNull> ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<digit> ::= '0' | <digitNotNull>
```

Types of syntax

- Abstract syntax
  - Structure
  - Independent of coding
  - Token separators are ignored (whitespace, comments)
  - Keywords are not specified
- Concrete syntax
  - Coding
  - Whitespace, Comments
  - Identifier for keywords

Distinction between concrete and abstract syntax not relevant to the exam.

Abstract syntax tree

- Derivation tree for abstract grammar
- → Abstract syntax tree (AST)

- Reduction rules:
  - Step-by-step rewriting of AST

Syntax of BSL

Entire program

<program> ::= <def-or-expr>*

Definition or expression

<def-or-expr> ::= <definition> | <e>

Definition of constants, functions and structures

<definition> ::= ...

Expression

<e> ::= ...

Value

<v> ::= ...

Syntax of BSL

<definition>::= '(' 'define' '(' <name> <name>+ ')' <e> ')'
| '(' 'define' <name> <e> ')'
| '(' 'define-struct' <name> '(' <name>+ ')' ')'

Syntax of BSL

```
<e> ::= '(' <name> <e>+ ')'
        | '(' 'cond' {'[' <e> <e> ']'}+ ')'
        | '(' 'cond' {'[' <e> <e> ']'}* '[' 'else' <e> ']' ')'
        | '(' 'if' <e> <e> <e> ')'
        | '(' 'and' <e> <e>+ ')'
        | '(' 'or' <e> <e>+ ')'
        <name>
        | <v>
```

Curly brackets to indicate repetitions of sequences.

Syntax of BSL

```
<e> ::= '(' <name> <e>+ ')'
        | '(' 'cond' {'[' <e> <e> ']'}+ ')'
        | '(' 'cond' {'[' <e> <e> ']'}* '[' 'else' <e> ']' ')'
        | '(' 'if' <e> <e> <e> ')'
        | '(' 'and' <e> <e>+ ')'
        | '(' 'or' <e> <e>+ ')'
        <name>
        | <v>
```

Why is a distinction made between and/or and function call?

Syntax of BSL

```
<e> ::= '(' <name> <e>+ ')'
        | '(' 'cond' {'[' <e> <e> ']'}+ ')'
        | '(' 'cond' {'[' <e> <e> ']'}* '[' 'else' <e> ']' ')'
        | '(' 'if' <e> <e> <e> ')'
        | '(' 'and' <e> <e>+ ')'
        | '(' 'or' <e> <e>+ ')'
        <name>
        | <v>
```

Why is a distinction made between and/or and function call?

Evaluation of the arguments differently: Evaluation only until the result has been determined.

Syntax of BSL

<v> ::= '<' 'make-'<name> <v>* '>'
<number>
<boolean>
<string>
<image>

Must not actually occur in BSL program. Use for AST rewriting by reduction rules.
Not to be confused with a function call to create a structure instance.

Syntax of BSL

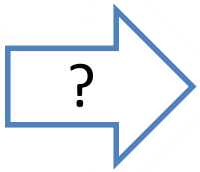
- Other non-terminals (without indication of production):
- <name>
  - Valid identifiers for functions, constants, structures and parameters
- <number>
  - Numbers, see previous lecture
- <boolean>
  - Truth values true or false
- <string>
  - String in quotation marks
- <image>
  - Image literals

BSL core language

- Simplification of syntax by omitting syntactic sugar
- Already known:
  - if
  - else
- In addition
  - (or e_1 ... e_n) ⇨ (not (and (not e_1) ... (not e_n)))
- BSL core language is a subset of the BSL language
  - BSL programs can be mapped to equivalent BSL core language programs by transformation

Is and syntactic sugar?

- Evaluation of and
- Evaluation of the sub-expressions up to
  - An expression false results in → Total expression is false
  - All expressions evaluated to true are → Total expression is true
- Restriction of evaluation positions known from cond expressions
- Can and be realized by cond?



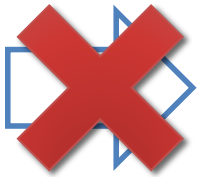
>(and true 42) >(cond [true 42] [else false])

Is and syntactic sugar?

- Evaluation of and
- Evaluation of the partial expressions up to
  - An expression false results in → Total expression is false
  - All expressions evaluated to true are → Total expression is true
- Restriction of evaluation positions known from cond expressions
- Can and be realized by cond?
  - → No!

>(and true 42)

and: question
is not true or false: 42



>(cond [true 42] [else false])
result 42

Syntax of Kern BSL

- Only grammar for expressions is changed

```
<e> ::= '(' <name> <e>+ ')'
        | '(' 'cond' {'[' <e> <e> ']'}+ ')'
        | '(' 'and' <e> <e>+ ')'
        <name>
        | <v>
```