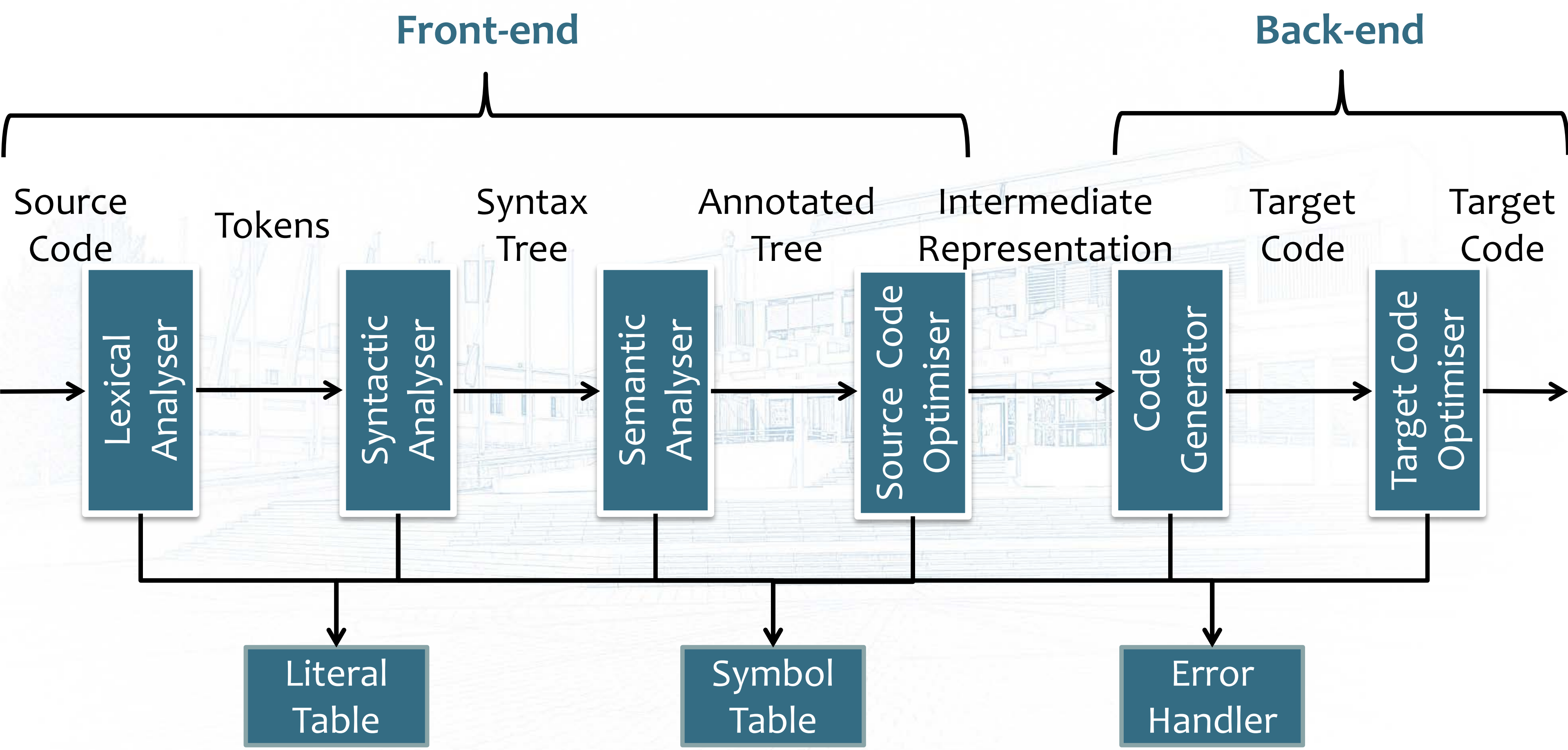
Prof. Radu Prodan

SYNTACTIC ANALYSIS CONTEXT-FREE GRAMMARS

Phases of a Compiler



Agenda

- **Introduction**
- Context-free grammars
- Ambiguous grammars
- Extended BNF
- Conclusions

Overview

- Syntactic analysis or parsing
  - Find program structure
- **Context-free grammar**
  - **Grammar rules** defines programming language syntax
  - Operates similar to scanner recognising regular expressions
- **Recursive** context-free grammars
  - E.g. nested **`for`** loops, nested **`if`** statements
- **Parse tree** or **syntax tree**
  - Increased complexity of data structure and algorithms

Parsing

- **Input:** tokens produced by lexical analyser
  - Parser calls **`getToken`** scanning procedure when needed

Sequence of tokens $\xrightarrow{\textbf{parser}}$ *Syntax Tree*

- **Output:** parse tree or syntax tree

- **Multi-pass compilers** explicitly create and save syntax tree
  - **`syntaxTree = parse();`**

- **Error handling**
  - Scanners consume incorrect characters and generate error token

- **Error recovery**
  - Infer possible correct code from incorrect code and continue parsing

Agenda

- Introduction
- **Context-free grammars**
- Ambiguous grammars
- Extended BNF
- Conclusions

Context-Free Grammar

- Syntactic structure of a programming language

- **Backus-Naur Form (BNF)** for integer arithmetic expressions
  - *exp* → *exp op exp* | (*exp*) | ***number***
  - *op* → **+** | **-** | *****

- `(34 - 3) * 42`
  - Corresponds to legal string of seven tokens
  - (***number*** - ***number***) * ***number***

- `(34 - 3 * 42`
  - Not legal expression because of a missing right parenthesis

Formal Context-Free Grammar Definition

- **Context-free grammar:** $G = (T, N, P, S)$
  - **Terminal** set: T
  - **Nonterminal** set: N (disjoint from T)
  - **Productions** or **grammar rules** P: $A \rightarrow \alpha, A \in N \wedge \alpha \in (T \cup N)^*$
  - **Start symbol:** $S \in N$

- **Symbol set:** $T \cup N$

BNF Grammar for Pascal

program → *program-heading* ; *program-block* .

program-heading → . . .
program-block → . . .
. . .

- *program* is start symbol
- *program*, *program-heading*, *program-block* are nonterminals
- ; and . are terminals
- Nonterminals defined through grammar rules or productions

Comparison to Regular Expressions

- Regular expression
  - *number* = *digit digit**
  - *digit* = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- BNF form
  - *number* → *digit* | *digit number*
  - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- Repetition specified by
  - * in regular expressions
  - Recursion in BNF grammar rules

Expression Derivation

- Derivation
  - Sequence of replacements of nonterminals by one grammar rule body

exp → *exp op exp* | **(** *exp* **)** | **number**
op → **+** | **-** | *****

- Derivation for **(34 - 3) * 42**
  - Define through grammar rules: →
  - Construct by replacement: ⇒

Step	Derivation		Rule
1	*exp*	⇒ *exp op exp*	[*exp* → *exp op exp*]
2		⇒ *exp op* **number**	[*exp* → **number**]
3		⇒ *exp* * **number**	[*op* → *****]
4		⇒ **(** *exp* **)** ***** **number**	[*exp* → **(** *exp* **)**]
5		⇒ **(** *exp op exp* **)** ***** **number**	[*exp* → *exp op exp*]
6		⇒ **(** *exp op* **number** **)** ***** **number**	[*exp* → **number**]
7		⇒ **(** *exp* **-** **number** **)** ***** **number**	[*op* → **-**]
8		⇒ **(** **number** **-** **number** **)** ***** **number**	[*exp* → **number**]

Derivation

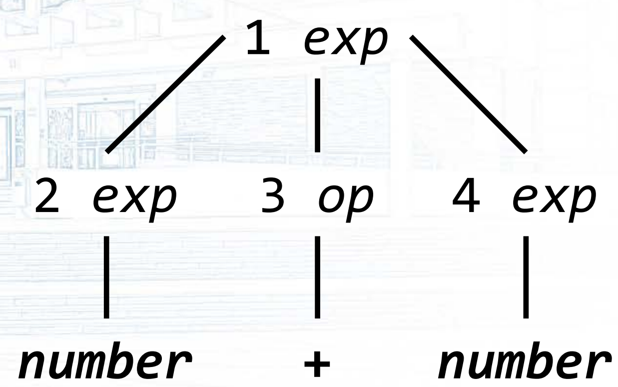
- **Derivation step** over G is of the form $\alpha A\gamma \Rightarrow \alpha\beta\gamma$
  - Where $\alpha \in (T \cup N)^*$, $\gamma \in (T \cup N)^*$, and $A \rightarrow \beta \in P$
- **Transitive closure** $\alpha_1 \Rightarrow^* \alpha_n$ of derivation step relation $\Rightarrow$
  - $\alpha_1 \Rightarrow^* \alpha_n \Leftrightarrow \exists\, \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$
- **Derivation** over grammar G is of the form $S \Rightarrow^* w$
  - $w \in T^*$ and $S \in N$ is start symbol of G
- **Language** generated by G
  - $L(G) = \{\, w \in T^* \mid \exists\, S \Rightarrow^* w \in G \,\}$
- **Leftmost derivation** $S \Rightarrow^*_{lm} w \Leftrightarrow \forall\, \alpha A\gamma \Rightarrow \alpha\beta\gamma$, then $\alpha \in T^*$
- **Rightmost derivation** $S \Rightarrow^*_{rm} w \Leftrightarrow \forall\, \alpha A\gamma \Rightarrow \alpha\beta\gamma$, then $\gamma \in T^*$

Parse Tree

- **Parse tree:** labelled tree representing a derivation
  - Interior nodes are nonterminals
  - Leaf nodes are terminals
  - Children of each internal node are body of a grammar production

- Example

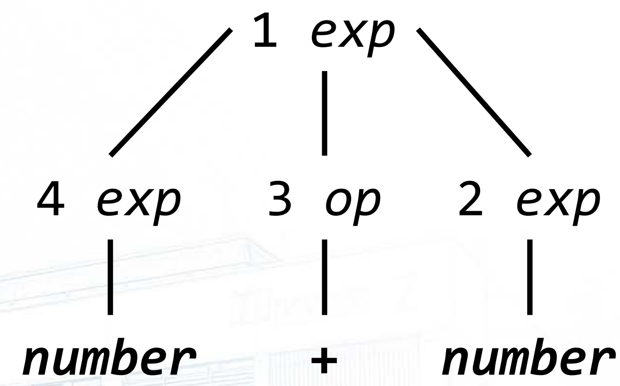
(1)	*exp*	$\Rightarrow$	*exp op exp*
(2)		$\Rightarrow$	***number*** *op exp*
(3)		$\Rightarrow$	***number*** **+** *exp*
(4)		$\Rightarrow$	***number*** **+** ***number***



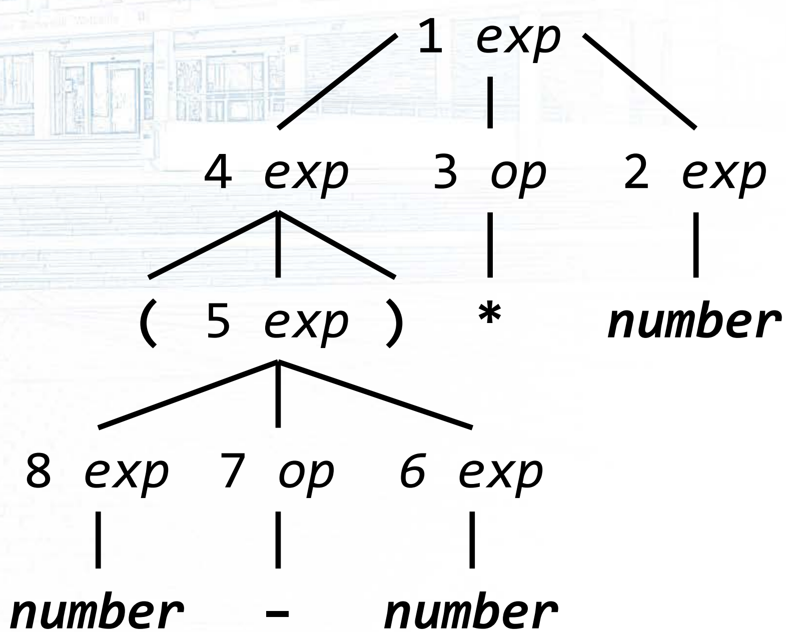
- Leftmost derivation
  - Leftmost nonterminal is replaced at each derivation step
  - Preorder numbering

Leftmost versus Rightmost Derivation

- Rightmost derivation
  - Rightmost nonterminal is replaced at each derivation step
  - Postorder numbering



$$
\begin{array}{llll}
(1) & exp & \Rightarrow & exp\ \ op\ \ exp \\
(2) & & \Rightarrow & exp\ \ op\ \ \textbf{number} \\
(3) & & \Rightarrow & exp\ \ \textbf{+}\ \ \textbf{number} \\
(4) & & \Rightarrow & \textbf{number}\ \ \textbf{+}\ \ \textbf{number}
\end{array}
$$

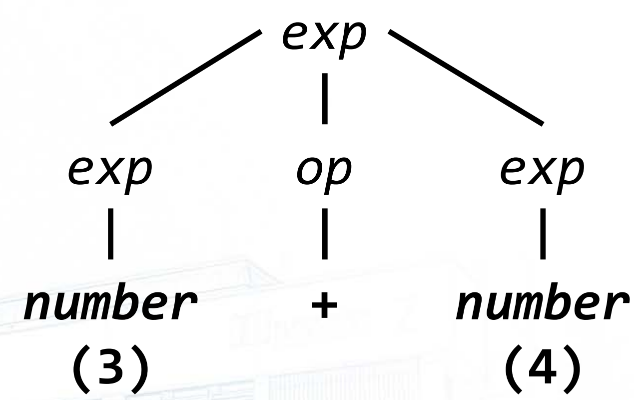


- Parse tree of rightmost derivation for **(34–3)*42**

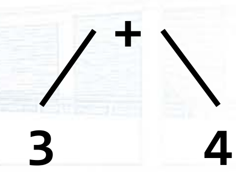
Abstract Syntax Tree (AST)

- Syntax-directed translation

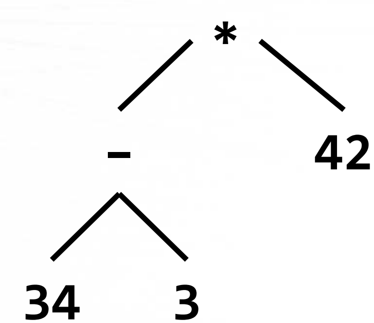
- Parse tree for **3+4**



- AST for **3+4**



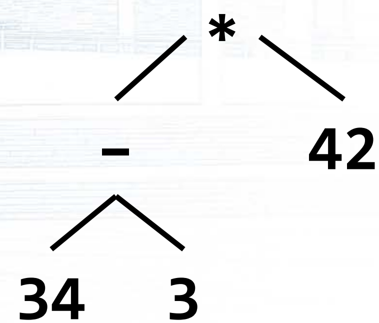
- AST for **(34–3)*42**
  - *OpExp(Times, OpExp(Minus, ConstExp(34), ConstExp(3)), ConstExp(42))*



AST for Expression Grammar

```
typedef enum { Plus, Minus, Times } OpKind;
typedef enum { OpK, ConstK } ExpKind;

typedef struct streenode
    {   ExpKind kind;
        OpKind op;
        struct streenode *lchild, *rchild;
        int val;
   }    STreeNode;
typedef STreeNode *SyntaxTree;
```



$exp \rightarrow exp\ op\ exp \mid (\ exp\ ) \mid \textbf{number}$

$op \rightarrow + \mid - \mid *$

Grammar of Paired Braces

- $E \rightarrow (\; E \;) \mid \mathbf{a}$
  - Nonterminals: E
  - Terminals: **(**, **)**, and **a**
  - $L(G) = \{$ **a**, **(a)**, **((a))**, **(((a)))**, … $\} = \{ (^n\mathbf{a})^n \mid n \in \mathbb{N} \}$
  - Derivation for **((a))**
    - $E \Rightarrow (\; E \;) \Rightarrow ((\; E \;)) \Rightarrow ((\; \mathbf{a} \;))$
- $E \rightarrow (\; E \;)$
  - Nonterminals: E
  - Terminals: **(** and **)**
  - $L(G) = \Phi$
    - Missing non-recursive case (or **base case**)

Left and Right Recursion

- Left recursive grammar G_l
  - $A \rightarrow A\mathbf{a} \mid \mathbf{a}$
  - $A \Rightarrow A\mathbf{a} \Rightarrow A\mathbf{aa} \Rightarrow A\mathbf{aaa} \Rightarrow \mathbf{aaaa}$

- Right recursive grammar G_r
  - $A \rightarrow \mathbf{a}A \mid \mathbf{a}$
  - $A \Rightarrow \mathbf{a}A \Rightarrow \mathbf{aa}A \Rightarrow \mathbf{aaa}A \Rightarrow \mathbf{aaaa}$

- $L(G_l) = L(G_r) = \{ a^n \mid n \in \mathbb{N}^* \} = L(\mathbf{a+})$
  - Same language as generated by regular expression **a+**

ε-Production

- Grammar for same language as regular expression **a***
  - Needs rule notation that generates empty string

- **ε-production**
  - *empty* →
  - *empty* → ε

- Left recursive grammar G_l
  - $A \rightarrow A\ \mathbf{a} \mid \varepsilon$

- Right recursive grammar G_r
  - $A \rightarrow \mathbf{a}\ A \mid \varepsilon$

- $L(G_l) = L(G_r) = \{\ \mathbf{a}^n \mid n \in \mathbb{N}\ \} = L(\mathbf{a}^*)$

Simplified Statement Grammar

- Without ε-productions

```
statement → if-stmt | other
if-stmt → if ( exp ) statement
        | if ( exp ) statement else statement
exp → 0 | 1
```

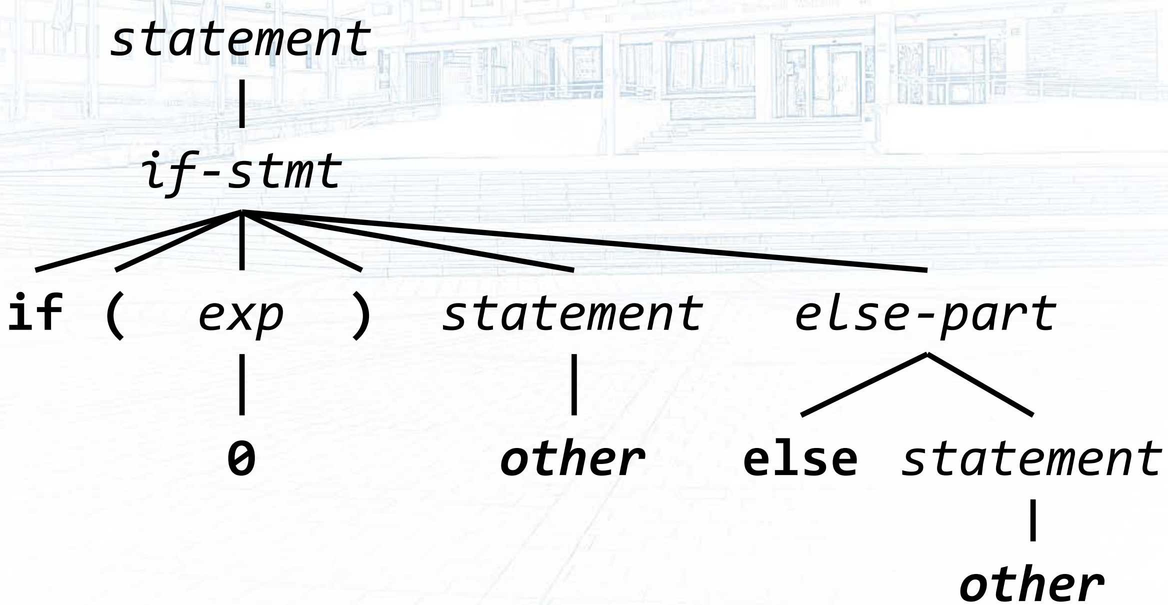
- With ε-productions

```
statement → if-stmt | other
if-stmt → if ( exp ) statement else-part
else-part → else statement | ε
exp → 0 | 1
```

Statement Grammar Parse Tree

statement → *if-stmt* | ***other***
if-stmt → **if** **(** *exp* **)** *statement* *else-part*
else-part → **else** *statement* | ε
exp → **0** | **1**

- `if (0) other else other`

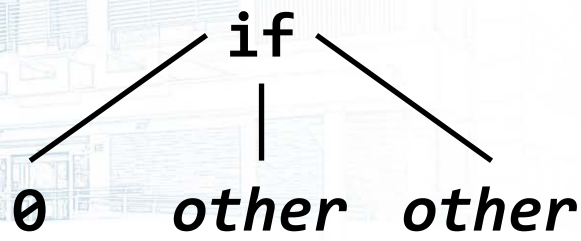


Statement Grammar AST

```
typedef enum { ExpK, StmtK } NodeKind;
typedef enum { Zero, One } ExpKind;
typedef enum { IfK, OtherK } StmtKind;

typedef struct streenode
  {   NodeKind kind;
      ExpKind ekind;
      StmtKind skind;
      struct streenode *test, *thenpart, *elsepart;
  }   STreeNode;

typedef STreeNode *SyntaxTree;
```

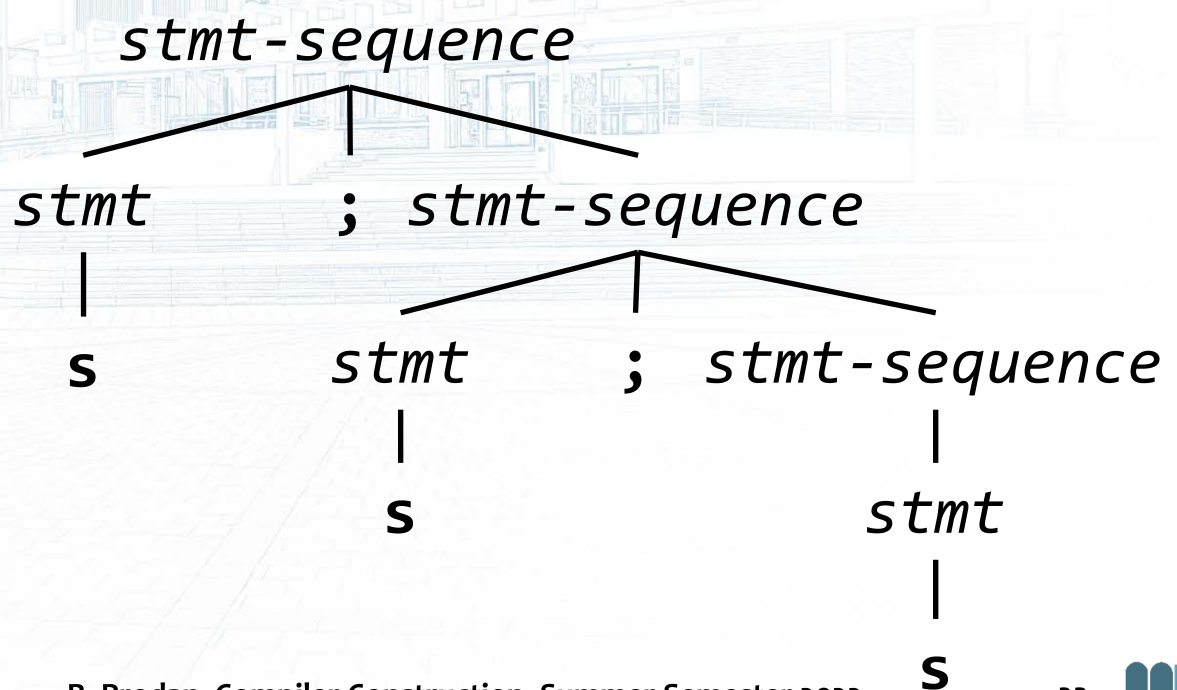


Statement Sequence Grammar

stmt-sequence → *stmt* ; *stmt-sequence* | *stmt*

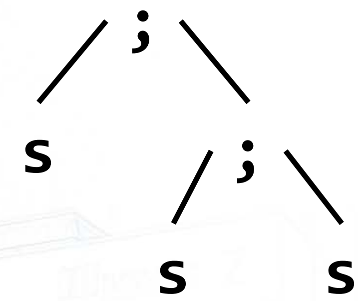
stmt → **s**

- Input string: **s;s;s**

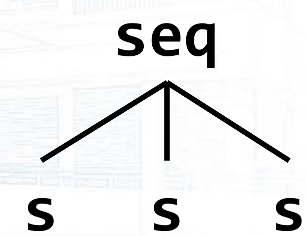


Statement Sequence AST

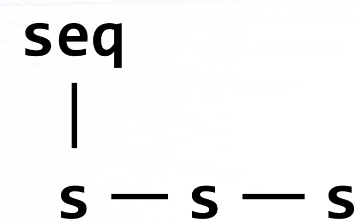
- Right derivation



- Variable number of children



- Leftmost-child right-sibling



Agenda

- Introduction
- Context-free grammars
- **Ambiguous grammars**
- Extended BNF
- Conclusions

Ambiguity

- Ambiguous grammar
  - Generates string with two distinct parse trees
  - Similar to nondeterministic automaton
  - Considered as incomplete specification

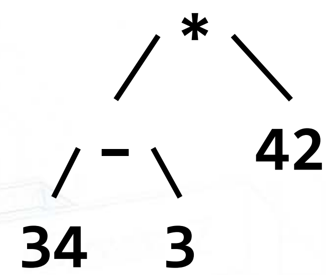
exp → *exp* *op* *exp* | (*exp*) | ***number***

op → **+** | **-** | *****

- Correct syntax tree for **34 - 3 * 42**
  - **34 - 3 = 31, 31 * 42 = 1302**
  - **3 * 42 = 126, 34 - 126 = -92**

Leftmost Derivation

exp → *exp op exp* | (*exp*) | ***number***
op → **+** | **-** | *****



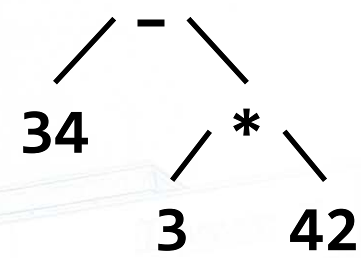
- Input string: **34–3*42**

Step	Leftmost Derivation	Rule
1	*exp* ⇒ *exp op exp*	[*exp* → *exp op exp*]
2	⇒ *exp op exp op exp*	[*exp* → *exp op exp*]
3	⇒ ***number*** *op exp op exp*	[*exp* → ***number***]
4	⇒ ***number*** **-** *exp op exp*	[*op* → **-**]
5	⇒ ***number*** **-** ***number*** *op exp*	[*exp* → ***number***]
6	⇒ ***number*** **-** ***number*** ***** *exp*	[*op* → *****]
7	⇒ ***number*** **-** ***number*** ***** ***number***	[*exp* → ***number***]

Another Leftmost Derivation

exp → *exp op exp* | (*exp*) | **number**

op → + | - | *



- Input string: **34–3*42**

Step	Leftmost Derivation	Rule
1	*exp* ⇒ *exp op exp*	[*exp* → *exp op exp*]
2	⇒ **number** *op exp*	[*exp* → **number**]
3	⇒ **number** - *exp*	[*op* → -]
4	⇒ **number** - *exp op exp*	[*exp* → *exp op exp*]
5	⇒ **number** - **number** *op exp*	[*exp* → **number**]
6	⇒ **number** - **number** * *exp*	[*op* → *]
7	⇒ **number** - **number** * **number**	[*exp* → **number**]

Disambiguating Rules

- Precedence relation of mathematical operators
- Left associative subtraction
  - **`34 - 3 - 42 = (34 - 3) - 42 = -11`**
  - **`34 - (3 - 42) = 73`**
- Non-associative operation
  - A sequence of more than one operator is not allowed
    - **`34 - 3 - 42`** or **`34 - 3 * 42`** are illegal
  - Only fully parenthesized expressions are legal
    - **`(34 - 3) - 42`**, **`34 - (3 * 42)`**

Ambiguity Removal

- Replace one recursion with base case
- Separate operators with different precedence
- Consider associativity when writing recursion

exp → *exp addop term* | *term*
addop → **+** | **-**
term → *term mulop factor* | *factor*
mulop → *****
factor → **(** *exp* **)** | ***number***

Ambiguous Rule	Nonambiguous left associative	Nonambiguous right associative
exp → *exp addop exp* | *term* *term* → *term mulop term* | *factor*	*exp* → *exp addop term* | *term* *term* → *term mulop factor* | *factor*	*exp* → *term addop exp* | *term* *term* → *factor mulop term* | *factor*

Dangling Else Problem

statement → *if-stmt* | ***other***

if-stmt → **if** **(** *exp* **)** *statement*

  | **if** **(** *exp* **)** *statement* **else** *statement*

exp → **0** | **1**

- **`if (0) if (1) other else other`**
  - Two distinct parse trees

- Ambiguous grammar because of optional else

- Most closely nested (disambiguating) rule

Other Dangling Else Solutions

- Mandatory **`else`** part
  - LISP and other functional languages

- Bracketing keyword

```
if-stmt → if condition then statement-sequence end if
        | if condition then statement-sequence
              else statement-sequence end if
```

Most Closely Nested Rule	Bracketing	Bracketing keyword	Required else

`if (x != 0)`
`  if (y == 1/x)`
`    ok = true`
`  else z = 1/x`
	`if (x == 0) {`
	`  if (y == 1/x) {`
	`    ok = true`
	`  } else z = 1/x`
	`}`
		`if (x != 0)`
		`  if (y == 1/x)`
		`    ok = true`
		`  else z = 1/x`
		`  end if`
		`end if`
			`if (x != 0)`
			`  if (y == 1/x)`
			`    ok = true`
			`  else z = 1/x`
			`else`

Inessential Ambiguity

seq

|

s — s — s

- Statement sequence grammar

stmt-sequence → *stmt* ; *stmt-sequence* | *stmt*

stmt → **s**

- Left or a right recursive grammars produce same abstract syntax tree structure

- Inessential ambiguity
  - Semantic does not depend on disambiguating rule

- Associative operations generate inessential ambiguity
  - Addition, multiplication, concatenation

Agenda

- Introduction
- Context-free grammars
- Ambiguous grammars
- **Extended BNF**
- Conclusions

Extended BNF

- Repetitive constructs

- Optional constructs

EBNF Repetitive Constructs

- Left recursive: $A \rightarrow A\ \alpha \mid \beta$
  - Kleene closure in regular expressions: $A \rightarrow \beta\ \alpha^*$
  - EBNF: $A \rightarrow \beta\ \{\ \alpha\ \}$
- Right recursive: $A \rightarrow \alpha\ A \mid \beta$
  - Kleene closure in regular expressions: $A \rightarrow \alpha^*\ \beta$
  - EBNF: $A \rightarrow \{\ \alpha\ \}\ \beta$

Expression Grammar in EBNF

$$exp \rightarrow exp\ addop\ term \mid term$$

- ENBF left associative form

$$exp \rightarrow term\ \{\ addop\ term\ \}$$

- EBNF right associative form

$$exp \rightarrow \{\ term\ addop\ \}\ term$$

EBNF Optional Constructs

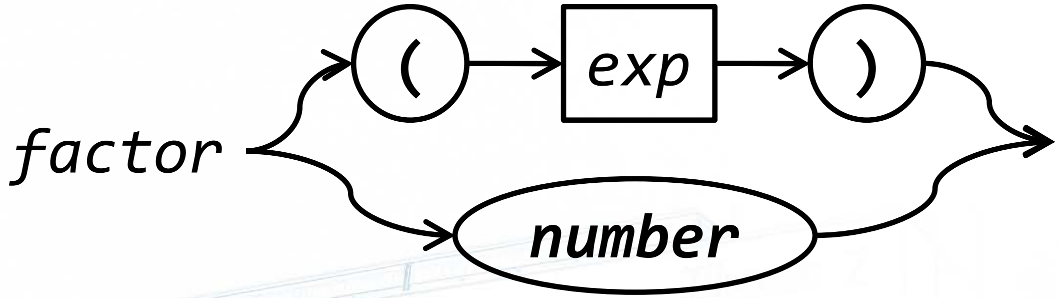
- Grammar rules for if-statements
  - BNF: *if-stmt* → **if** (*exp*) *statement*
    | **if** (*exp*) *statement* **else** *statement*
  - EBNF: *if-stmt* → **if** (*exp*) *statement* [**else** *statement*]

- Statement sequence grammar
  - BNF: *stmt-sequence* → *stmt* **;** *stmt-sequence* | *stmt*
  - EBNF: *stmt-sequence* → *stmt* [**;** *stmt-sequence*]

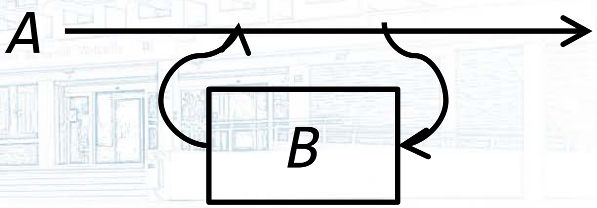
- Addition operation in right associative form
  - BNF: *exp* → *term addop exp* | *term*
  - EBNF: *exp* → *term* [*addop exp*]

EBNF Syntax Diagrams

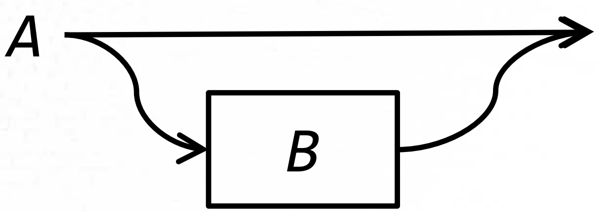
$$factor \rightarrow \texttt{(}\ exp\ \texttt{)}\ \mid\ \textbf{number}$$



$$A \rightarrow \{\ B\ \}$$



$$A \rightarrow [\ B\ ]$$



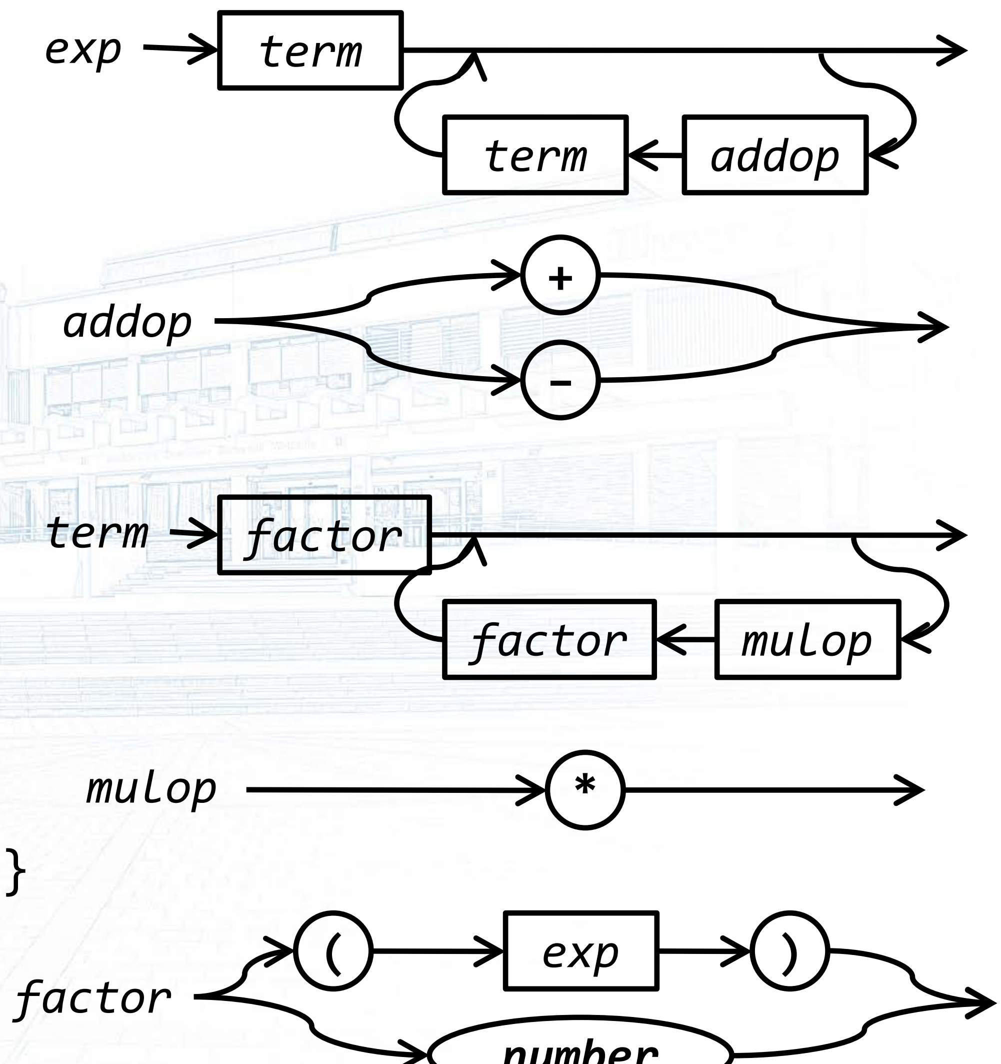
Syntax Diagrams for Simple Arithmetic Expression Grammar

- BNF expression grammar

```
exp → exp addop term
       | term
addop → + | -
term → term mulop factor
        | factor
mulop → *
factor → ( exp ) | number
```

- EBNF expression grammar

```
exp → term { addop term }
addop → + | -
term → factor { mulop factor }
mulop → *
factor → ( exp ) | number
```



Syntax Diagrams for Simplified Grammar of If-Statements

- BNF grammar

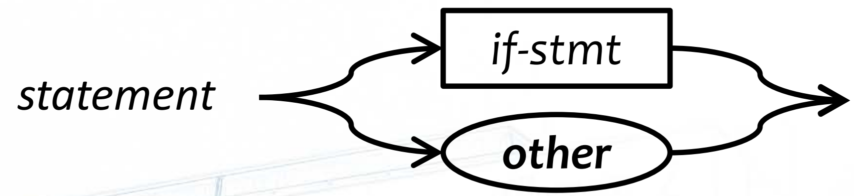
statement→ *if-stmt* | ***other***

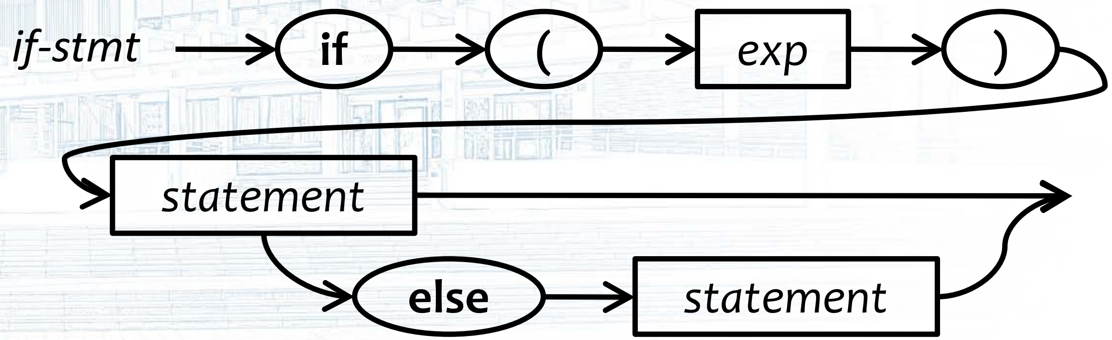
if-stmt → **if** (*exp*) *statement*

  | **if** (*exp*) *statement*

    else *statement*

exp → **0** | **1**





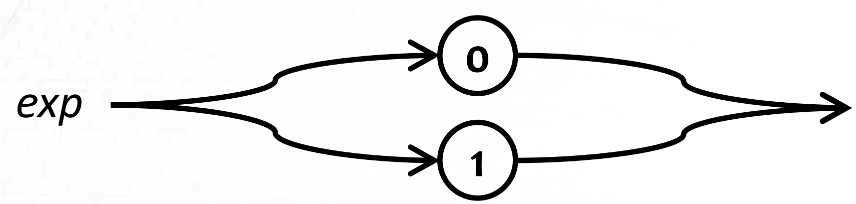
- EBNF grammar

statement → *if-stmt* | ***other***

if-stmt → **if** (*exp*) *statement*

  [**else** *statement*]

exp → **0** | **1**



Agenda

- Introduction
- Context-free grammars
- Ambiguous grammars
- Extended BNF
- **Conclusions**

Conclusions

- Syntactic analysis or parsing
- Specified through context-free grammars
- Represented as Abstract Syntax Trees
- Ambiguous grammars
- BNF and EBNF representation