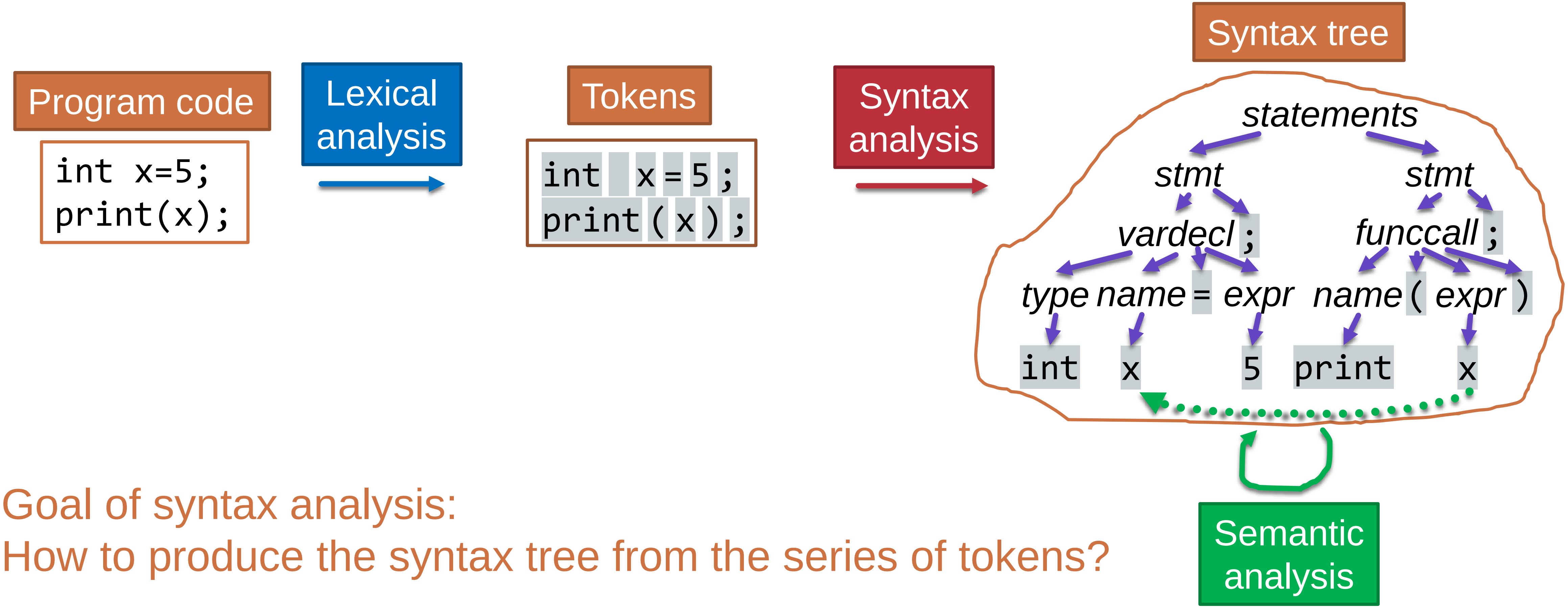
Model-based Software Development

Lecture 3

Syntax Analysis

Dr. Balázs Simon

Compiler front-end



Program code

```
int x=5;
print(x);
```

Lexical analysis

Tokens

```
int x = 5 ;
print ( x ) ;
```

Syntax analysis

Syntax tree

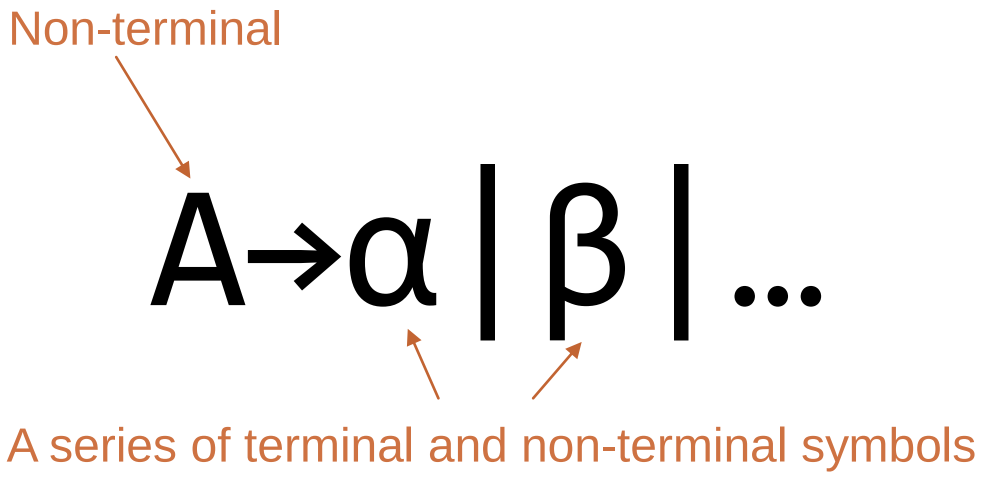
Semantic analysis

Goal of syntax analysis:
How to produce the syntax tree from the series of tokens?

This lecture: Syntax analysis

- **I. Context free (CF) grammars**
- **II. Naïve methods (BFS, DFS)**
- **III. Left analysis: LL(k), LL(*)**
- **IV. Right analysis: LR(k), LALR**
- **V. Error handling**

Context free (CF) grammars



Non-terminal

$$A \rightarrow \alpha \mid \beta \mid \ldots$$

A series of terminal and non-terminal symbols

Notation:

- Uppercase letters: non-terminal symbols
- Lowercase letters, other characters: terminals
- Greek letters: any terminal - non-terminal series

Parse tree:

- Root: start symbol
- Internal node: non-terminal symbol
- Leaf: terminal symbol (token)

CF grammar example

Production rules:

①②③ ④ ⑤
E → x|y|z|E Op E|(E)

⑥⑦⑧⑨
Op → +|-|*|/

Program code:

x+y*z

Lexical analysis (tokens/terminals):

x + y * z

One possible derivation:

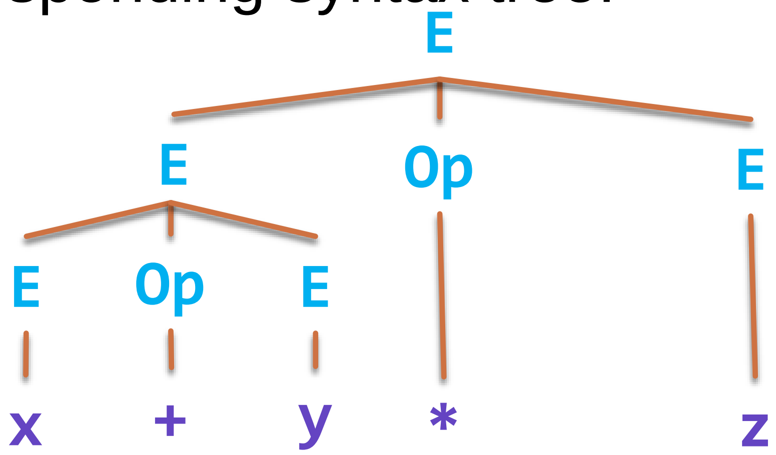
$\boxed{E} \xrightarrow{\text{④}} \boxed{E}\ Op\ E \xrightarrow{\text{④}} E\ Op\ \boxed{E}\ Op\ E \rightarrow$

$\xrightarrow{\text{②}} E\ \boxed{Op}\ y\ Op\ E \xrightarrow{\text{⑥}} \boxed{E} + y\ Op\ E \rightarrow$

$\xrightarrow{\text{①}} x + y\ \boxed{Op}\ E \xrightarrow{\text{⑧}} x + y * \boxed{E} \rightarrow$

$\xrightarrow{\text{③}} x + y * z$

Corresponding syntax tree:



meaning:
(x+y)*z

Conclusions

- There are many possible derivations
  - ambiguous grammar
- The meaning is determined by the structure of the syntax tree
  - depends on the order of the applied rules
  - precedence of operators
- Unambiguous grammar:
  - syntax tree is always unambiguous

Unambiguous grammar

Production rules:

E → ①E+T | ②E-T | ③T

T → ④T*F | ⑤T/F | ⑥F

F → ⑦x | ⑧y | ⑨z | ⑩(E)

One possible derivation:

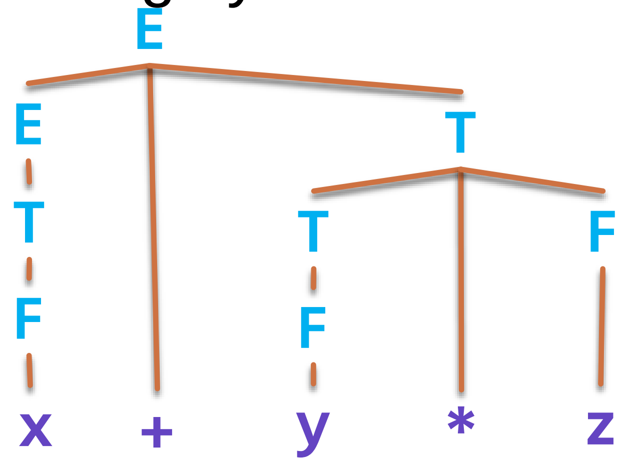
$\boxed{E} \xrightarrow{\textcircled{1}} E + \boxed{T} \xrightarrow{\textcircled{4}} E + \boxed{T} * F \rightarrow$

$\xrightarrow{\textcircled{6}} E + \boxed{F} * F \xrightarrow{\textcircled{8}} \boxed{E} + y * F \rightarrow$

$\xrightarrow{\textcircled{3}} \boxed{T} + y * F \xrightarrow{\textcircled{6}} \boxed{F} + y * F \rightarrow$

$\xrightarrow{\textcircled{7}} x + y * \boxed{F} \xrightarrow{\textcircled{9}} x + y * z$

Corresponding syntax tree:



meaning:
x+(y*z)

Program code:

x+y*z

Lexical analysis (tokens/terminals):

x + y * z

Questions

- How can we make the analysis automated?
  - there are many different algorithms
- How can we detect if a grammar is unambiguous?
  - depends on the analyzer: either when we construct the analyzer or when we run the analyzer
- Can we make a grammar unambiguous?
  - usually yes: by rewriting the production rules
  - caution: the structure may change

This lecture: Syntax analysis

- I. Context free (CF) grammars
- II. Naïve methods (BFS, DFS)
- III. Left analysis: LL(k), LL(*)
- IV. Right analysis: LR(k), LALR
- V. Error handling

Naïve methods

- Idea:
  - graph search algorithms (BFS, DFS)
  - backtracking
- Start with the start symbol
- Nodes:
  - symbol series derivable from the start symbol in one or more steps
- Edges:
  - between α and β if β can be directly (in a single step) derived from α

Example

Production rules:

E → ①E+T | ②E-T | ③T

T → ④T*F | ⑤T/F | ⑥F

F → ⑦x | ⑧y | ⑨z | ⑩(E)

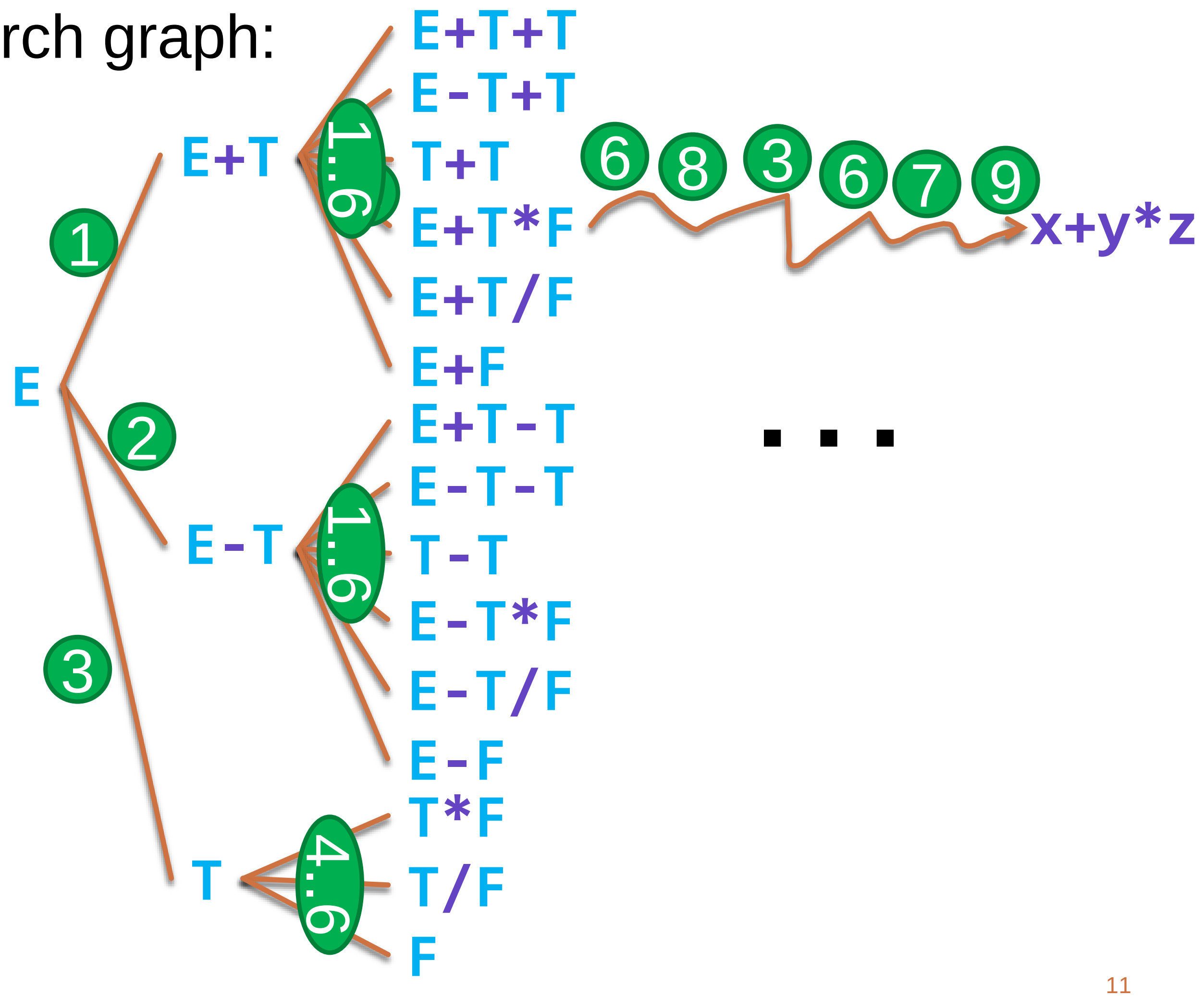
Program code:

x+y*z

Lexical analysis (tokens/terminals):

x + y * z

Search graph:



Example: Breadth-First Search (BFS)

Production rules:

E → ①E+T | ②E-T | ③T
T → ④T*F | ⑤T/F | ⑥F
F → ⑦x | ⑧y | ⑨z | ⑩(E)

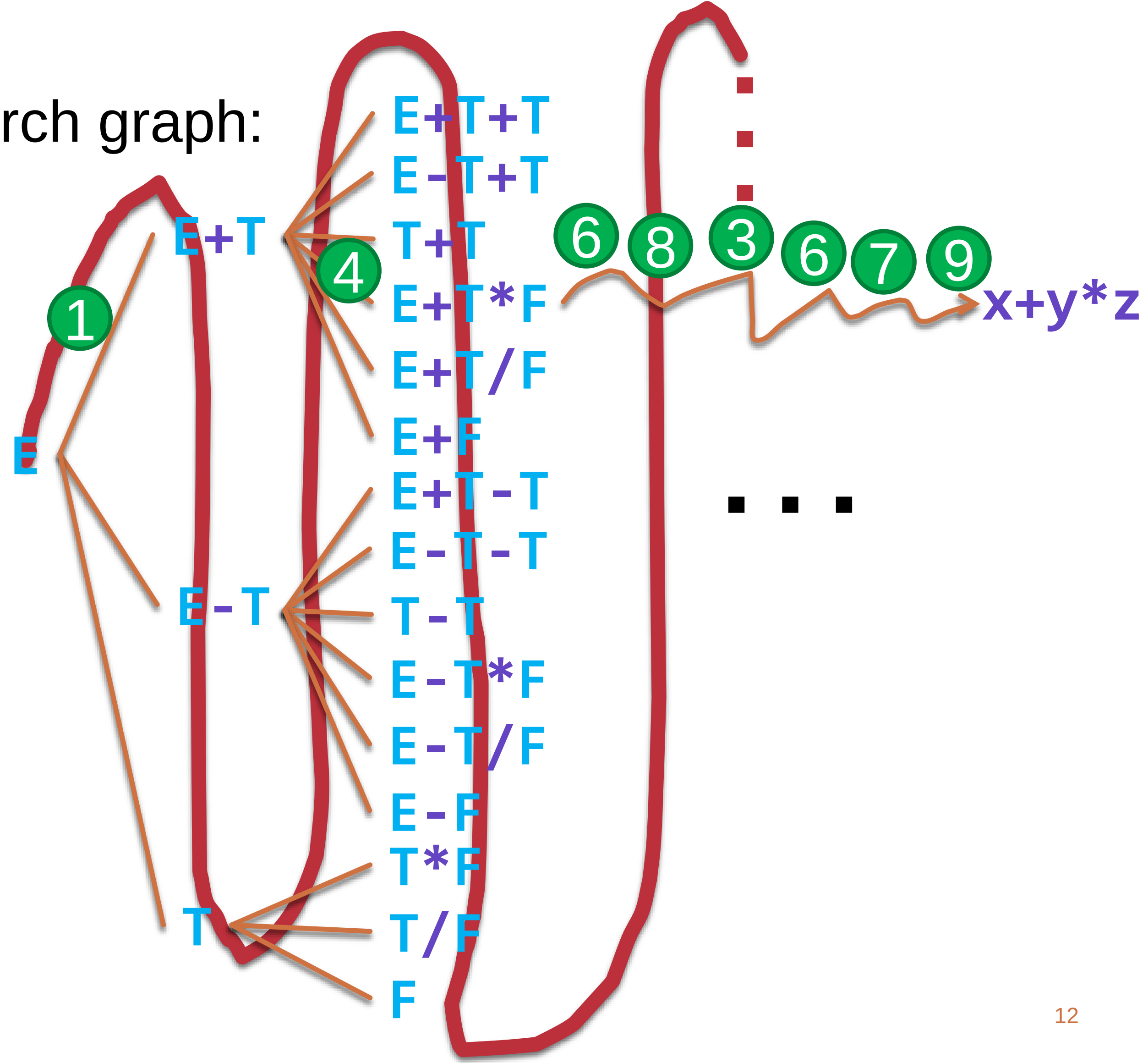
Program code:

x+y*z

Lexical analysis (tokens/terminals):

x + y * z

Search graph:



E
E+T
E-T
T
E+T+T
E-T+T
T+T
E+T*F
E+T/F
E+F
E+T-T
E-T-T
T-T
E-T*F
E-T/F
E-F
T*F
T/F
F
. . .
x+y*z

Example: Depth-First Search (DFS)

Production rules:

E → ① E+T | ② E-T | ③ T

T → ④ T*F | ⑤ T/F | ⑥ F

F → ⑦ x | ⑧ y | ⑨ z | ⑩ (E)

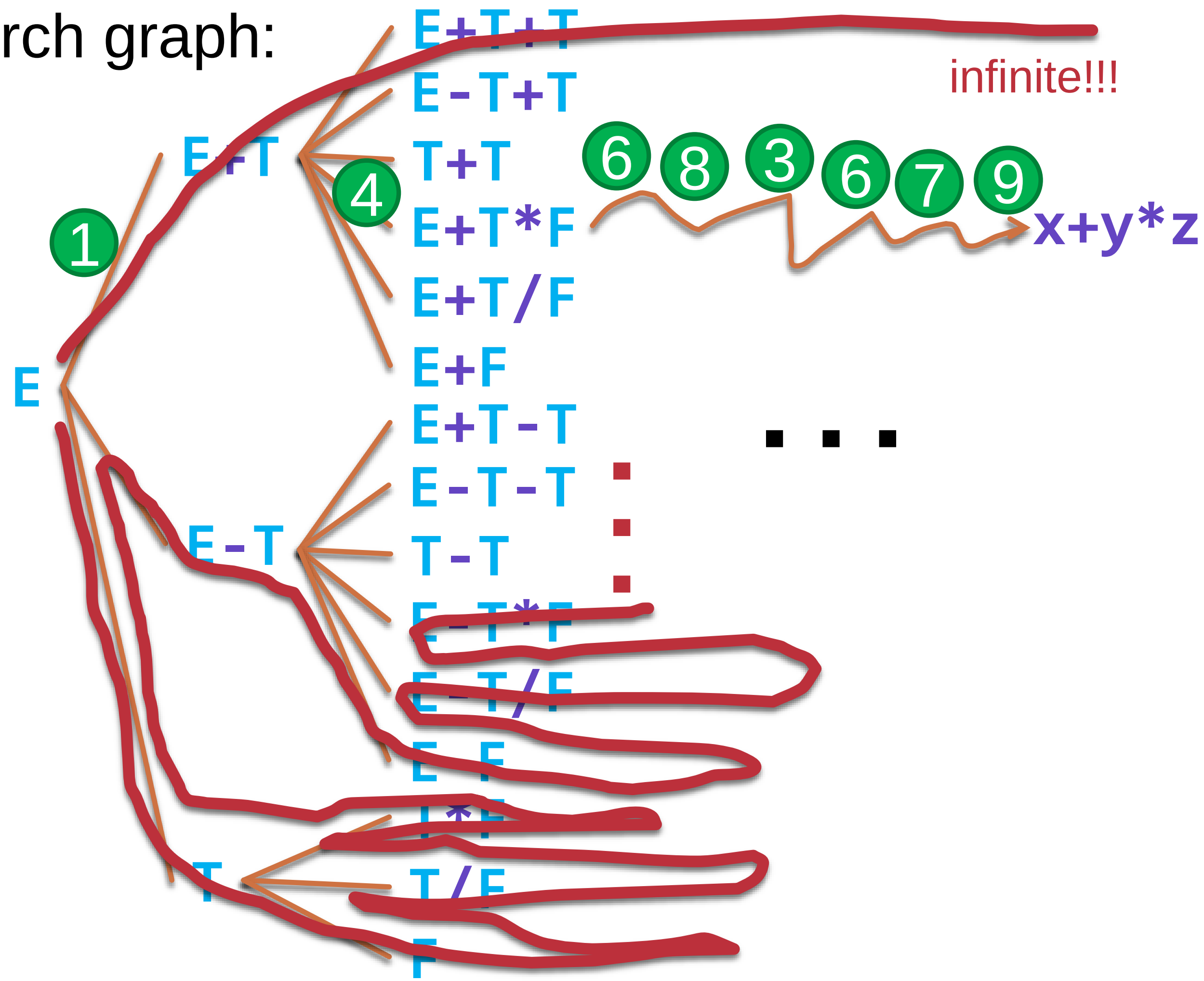
Program code:

x+y*z

Lexical analysis (tokens/terminals):

x + y * z

Search graph:



Conclusions

- Search graph ≠ syntax tree
  - syntax tree can be constructed based on the edges of the search graph
- Problem: large state space, many branches
  - BFS: exponential memory, exponential time
  - DFS: smaller memory, but can be infinite in time
    - problem: left recursion (`E` → `E+T` → `E+E+T` → `E+E+E+T` → `...`)
    - solution: later…
- State space must somehow be reduced
- Rarely used in practice

This lecture: Syntax analysis

- I. Context free (CF) grammars
- II. Naïve methods (BFS, DFS)
- **III. Left analysis: LL(k), LL(*)**
- IV. Right analysis: LR(k), LALR
- V. Error handling

Left analysis: LL(k)

- Prediction instead of backtracking
- Idea:
  - L: read from left to right
  - L: always rewrite the leftmost non-terminal (leftmost derivation)
  - decision between alternatives: based on the lookahead of **k** terminals (token)
- Errors recognized automatically: comparing the produced and predicted terminals
  - extraneous and missing tokens
- Problem: left recursion

Left analysis example

Production rules:

$$S \rightarrow \overset{①}{Sa} \mid \overset{②}{b}$$

Program code:

baaaa

Lexical analysis (tokens/terminals):

b	a	a	a	a

LL(1) analysis:

Derivation

Read position

Look-ahead

$$S,\ |baaaa \xrightarrow{①} Sa,\ |baaaa \rightarrow$$

$$\xrightarrow{①} Saa,\ |baaaa \xrightarrow{①} Saaa,\ |baaaa \rightarrow$$

$$\xrightarrow{①} Saaaa,\ |baaaa \rightarrow ?$$

We should stop here using rule 2, however, we can't possibly know this by the look-ahead!

Resolving left recursion: by transforming the grammar

Production rules:

$$S \rightarrow \overset{\textcircled{1}}{b\hat{S}}$$

$$\hat{S} \rightarrow \overset{\textcircled{2}}{a\hat{S}} \mid \overset{\textcircled{3}}{\varepsilon}$$

Program code:

baaaa

Lexical analysis (tokens/terminals):

b a a a a

LL(1) analysis:

Derivation

Read position

Look-ahead

$S,\ |baaaa \xrightarrow{\textcircled{1}} b\hat{S},\ b|aaaa \rightarrow$

$\xrightarrow{\textcircled{2}} ba\hat{S},\ ba|aaa \xrightarrow{\textcircled{2}} baa\hat{S},\ baa|aa \rightarrow$

$\xrightarrow{\textcircled{2}} baaa\hat{S},\ baaa|a \rightarrow$

$\xrightarrow{\textcircled{2}} baaaa\hat{S},\ baaaa|\varepsilon \rightarrow$

$\xrightarrow{\textcircled{3}} baaaa,\ baaaa|\varepsilon$

Left recursion

- Solution 1: transforming the grammar
  - disadvantage: the structure of the syntax tree changes
- Solution 2: priority between different rules
  - ANTLR4: direct left recursion is allowed (but indirect is not!)

Solution 1: transforming the grammar

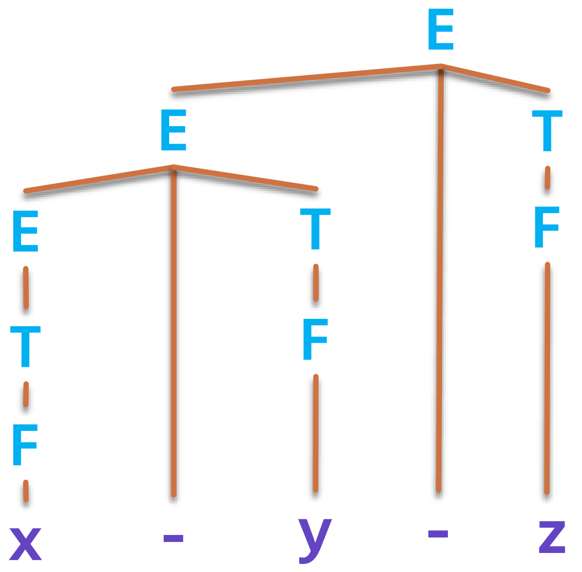
E → E+T | E-T | T
T → T*F | T/F | F
F → x | y | z | (E)

Question: why is this a bad solution?

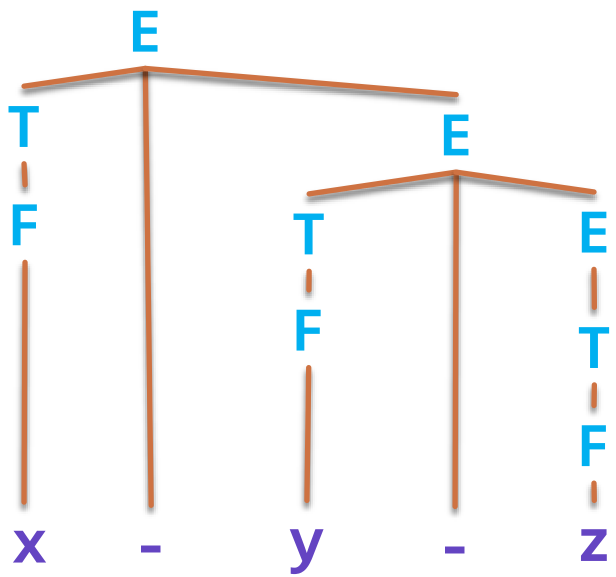
E → T+E | T-E | T
T → F*T | F/T | F
F → x | y | z | (E)

Answer: associativity of the operators is important!

meaning:
(x-y)-z



vs.

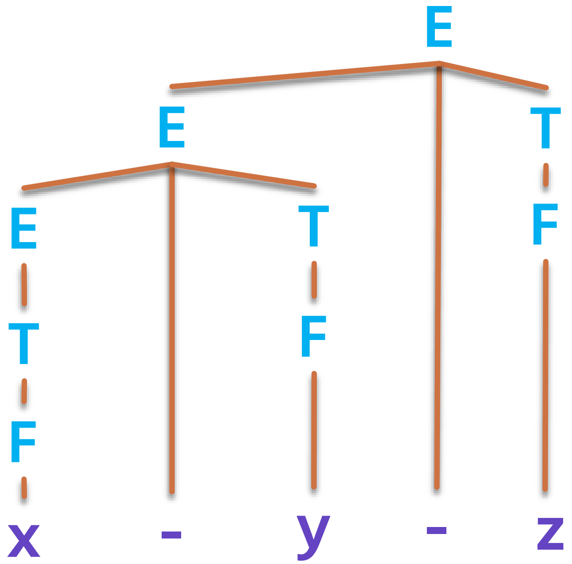


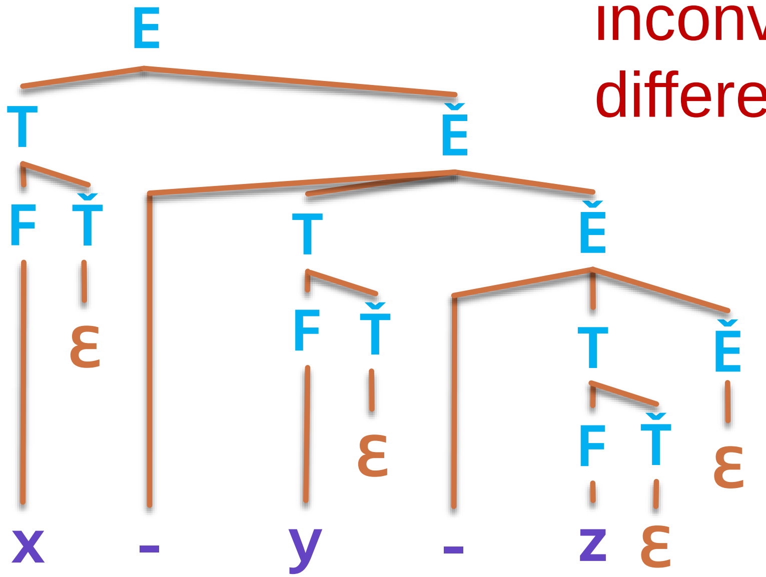
meaning:
x-(y-z)

Solution 1: correct transformation of the grammar

E → E+T | E-T | T
T → T*F | T/F | F
F → x | y | z | (E)

E → TĚ
Ě → +TĚ | -TĚ | ε
T → FŤ
Ť → *FŤ | /FŤ | ε
F → x | y | z | (E)





inconvenience:
different structure

LL(1) analysis

Production rules:

$$E \rightarrow \overset{①}{T\check{E}}$$

$$\check{E} \rightarrow \overset{②}{+T\check{E}} \mid \overset{③}{-T\check{E}} \mid \overset{④}{\varepsilon}$$

$$T \rightarrow \overset{⑤}{F\check{T}}$$

$$\check{T} \rightarrow \overset{⑥}{*F\check{T}} \mid \overset{⑦}{/F\check{T}} \mid \overset{⑧}{\varepsilon}$$

$$F \rightarrow \overset{⑨}{x} \mid \overset{⑩}{y} \mid \overset{⑪}{z} \mid \overset{⑫}{(E)}$$

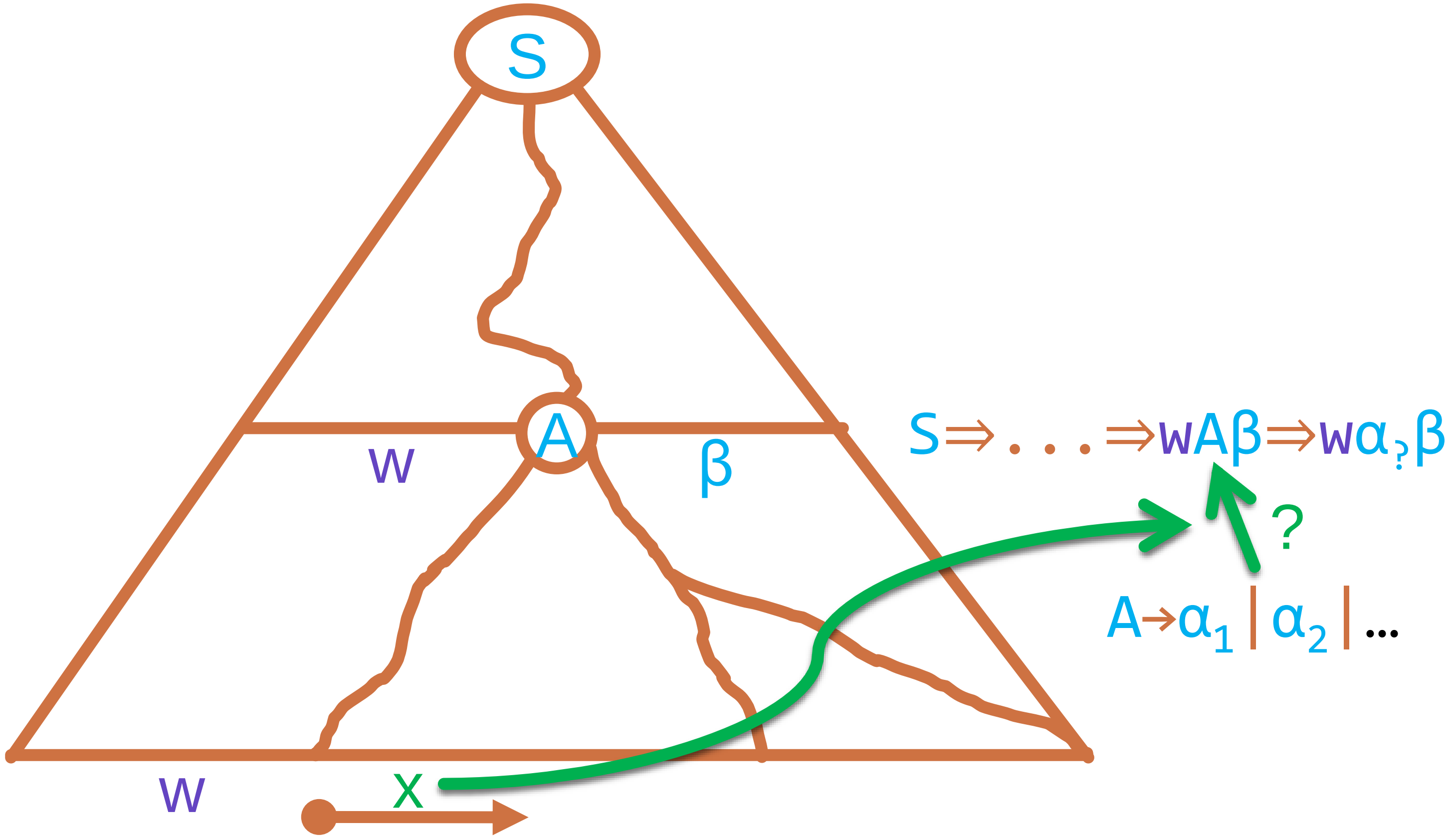
LL(1) analysis:

E, |x+y*z $\xrightarrow{①}$ TĚ, |x+y*z →
$\xrightarrow{⑤}$ FŤĚ, |x+y*z $\xrightarrow{⑨}$ xŤĚ, x|+y*z →
$\xrightarrow{⑧}$ xĚ, x|+y*z $\xrightarrow{②}$ x+TĚ, x+|y*z →
$\xrightarrow{⑤}$ x+FŤĚ, x+|y*z $\xrightarrow{⑩}$ x+yŤĚ, x+y|*z →
$\xrightarrow{⑥}$ x+y*FŤĚ, x+y*|z →
$\xrightarrow{⑪}$ x+y*zŤĚ, x+y*z|ε →
$\xrightarrow{⑧}$ x+y*zĚ, x+y*z|ε →
$\xrightarrow{④}$ x+y*z, x+y*z|ε

Program code: **x+y*z**

Lexical analysis (tokens): x + y * z

LL(k) overview



We have to take into account the characters generated by A and β, when we compute x of length **k**.

Advantages of LL(k)

- Simple
  - can be programmed manually (e.g., Roslyn – C# compiler): Recursive Descent Parser
  - analyzer can be easily generated: based on a table
- Fast
- Small memory footprint
- Easy to detect and signal errors
  - smaller errors can be fixed, and the analysis can be continued
  - e.g., skipping an unexpected token, inserting a missing token

Example (1/3): LL(k) analyzer written manually

Production rules:

E → TĚ
Ě → +TĚ | -TĚ | ε
T → FŤ
Ť → *FŤ | /FŤ | ε
F → x | y | z | (E)

C# code:

```
E ParseE()
{
    var t = ParseT();
    var ehat = ParseEHat();
    return new E(t, ehat);
}
```

Example (2/3): LL(k) analyzer written manually

Production rules:

E → TĚ
Ě → +TĚ | -TĚ | ε
T → FŤ
Ť → *FŤ | /FŤ | ε
F → x | y | z | (E)

C# code:

```
EHat? ParseEHat() {
    var la1 = LA(1);
    switch (la1)
    {
        case "+":
            Match("+");
            var t1 = ParseT();
            var ehat1 = ParseEHat();
            return new EHat(t1, ehat1);
        case "-":
            Match("-");
            var t2 = ParseT();
            var ehat2 = ParseEHat();
            return new EHat(t2, ehat2);
        default:
            return null;
    }
}
```

Look-ahead

Step forward

Example (3/3): LL(k) analyzer written manually

C# code:

```
F? ParseF()
{
    var la1 = LA(1);
    switch (la1)
    {
        case "x": Match("x"); return new F("x");
        case "y": Match("y"); return new F("y");
        case "z": Match("z"); return new F("z");
        case "(":
            Match("(");
            var e = ParseE();
            Match(")");
            return new F(e);
        default:
            Unexpected(la1);
            return null;
    }
}
```

Error, if something else comes!

Error if an unexpected token comes!

Production rules:

E → TĚ
Ě → +TĚ | -TĚ | ε
T → FŤ
Ť → *FŤ | /FŤ | ε
F → x | y | z | (E)

LL(*) analysis

- Idea:
  - instead of a fixed **k** lookahead, predict the winning alternative using a state machine
  - the state machine can read ahead any number of tokens
- Advantage: stronger than LL(k)
- In practice:
  - ANTLR3 - LL(*): http://www.antlr.org/papers/LL-star-PLDI11.pdf
  - ANTLR4 - ALL(*) (Adaptive LL): http://www.antlr.org/papers/allstar-techreport.pdf
    - even allows direct left recursion with precedence between the alternatives

This lecture: Syntax analysis

- I. Context free (CF) grammars
- II. Naïve methods (BFS, DFS)
- III. Left analysis: LL(k), LL(*)
- **IV. Right analysis: LR(k), LALR**
- V. Error handling

Right analysis: LR(k)

- Prediction instead of backtracking
- Idea:
  - L: read from left to right
  - R: always the rightmost part of the derivation is changed (top of the stack)
  - decision between shift-reduce: based on the lookahead of **k** terminals (token)
    - shift: put the next terminal onto the top of the stack
    - reduce: replace some symbols on the top of the stack, a potential right side of a rule, with the left side of the rule
- Automatic error recognition: the first error can be recognized precisely
  - backtracking is necessary to continue the analysis
- Left recursion is not a problem

LR(1) analysis

Production rules:

$E \rightarrow \overset{\textcircled{1}}{E+T} \mid \overset{\textcircled{2}}{E-T} \mid \overset{\textcircled{3}}{T}$

$T \rightarrow \overset{\textcircled{4}}{T*F} \mid \overset{\textcircled{5}}{T/F} \mid \overset{\textcircled{6}}{F}$

$F \rightarrow \overset{\textcircled{7}}{x} \mid \overset{\textcircled{8}}{y} \mid \overset{\textcircled{9}}{z} \mid \overset{\textcircled{10}}{(E)}$

Program code:

x+y*z

Lexical analysis (tokens):

x + y * z

LR(1) analysis:

Stack

Read position

Look-ahead

ε, |x+y*z $\xrightarrow{S}$ x, x|+y*z $\xrightarrow{\textcircled{7}}$ F, x|+y*z →

$\xrightarrow{\textcircled{6}}$ T, x|+y*z $\xrightarrow{\textcircled{3}}$ E, x|+y*z $\xrightarrow{S}$ E+, x+|y*z →

$\xrightarrow{S}$ E+y, x+y|*z $\xrightarrow{\textcircled{8}}$ E+F, x+y|*z →

$\xrightarrow{\textcircled{6}}$ E+T, x+y|*z $\xrightarrow{S}$ E+T*, x+y*|z →

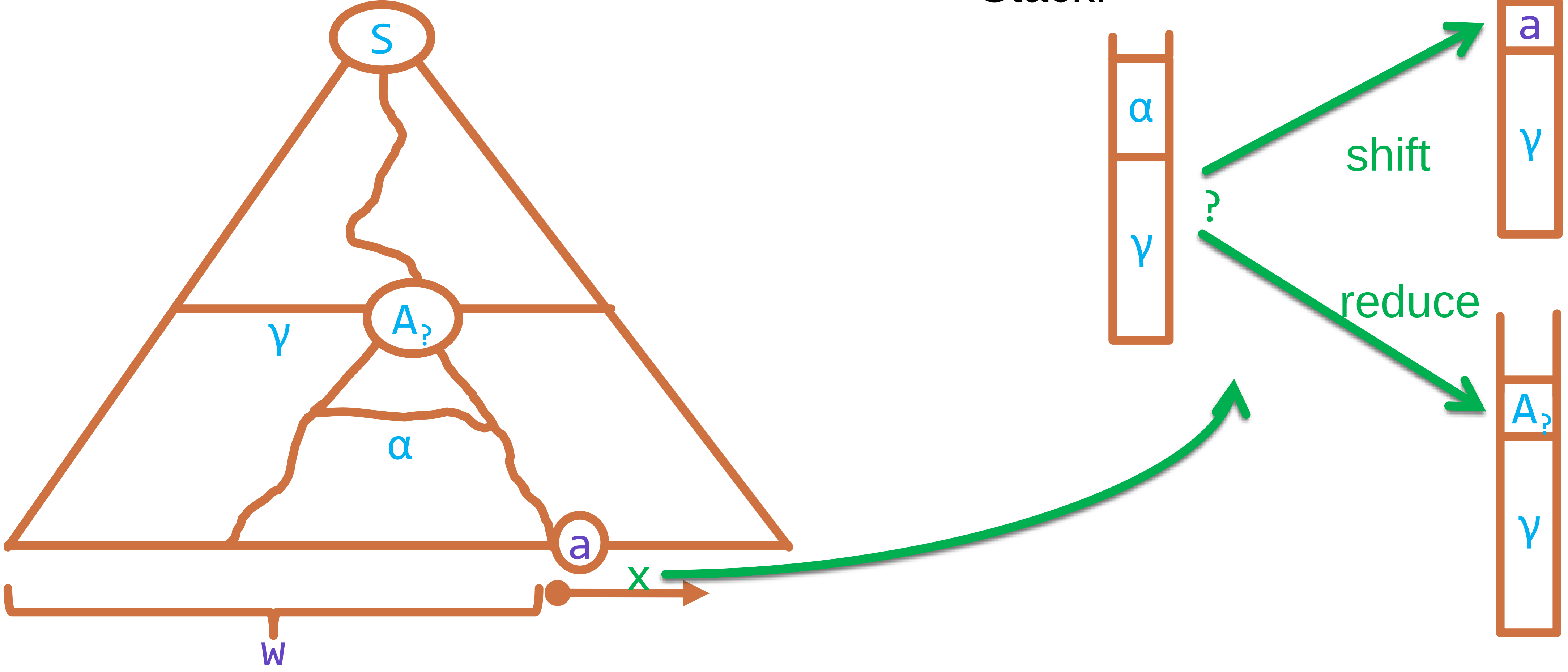
$\xrightarrow{S}$ E+T*z, x+y*z|ε $\xrightarrow{\textcircled{9}}$ E+T*F, x+y*z|ε →

$\xrightarrow{\textcircled{4}}$ E+T, x+y*z|ε $\xrightarrow{\textcircled{1}}$ E, x+y*z|ε

LR(k) overview

$S \Rightarrow ... \Rightarrow \gamma A_? x \Rightarrow \gamma \alpha x \Rightarrow ... \Rightarrow wx$

Stack:



Shift-Reduce conflict: Dangling Else

What is the problem with the following grammar?

```
Statement  → IfStatement | Expression
Expression  → IDENTIFIER
IfStatement  →
  ①IF Expression THEN Statement |
  ②IF Expression THEN Statement ELSE Statement
```

Possible solutions:

1. Changing the grammar
2. Prefer either shift or reduce
3. Precedence between rules

How does it analyze the following?

```
IF Cond1 THEN
    IF Cond2 THEN DoSomething1
  ② ELSE DoSomething2
```

Shift?

How does it analyze the following?

```
IF Cond1 THEN
    IF Cond2 THEN DoSomething1 ①
ELSE DoSomething2
```

Reduce?

(Problems like „dangling else” are an issue in general, not just for LR analysis.)

Resolution of the Shift-Reduce conflict: changing the grammar

```
Statement → IfStatement | Expression
Expression → IDENTIFIER
IfStatement →
  IF Expression THEN Statement |
  IF Expression THEN Statement ELSE Statement
```

⬇

```
Statement → IfStatement | Expression
Expression → IDENTIFIER
IfStatement →
  IF Expression THEN Statement |
  IF Expression THEN IfThenElseStatement ELSE Statement
IfThenElseStatement →
  IF Expression THEN IfThenElseStatement ELSE IfThenElseStatement |
  Expression
```

ELSE is always bound to the nearest IF

LR(k) analysis

- Advantages:
  - Stronger than LL(k)
  - Deterministic
  - Fast: linear in time
  - Left recursion is not a problem
  - Can be generated automatically: based on a table
- Disadvantages:
  - The table is very large, even for LR(1)
  - Shift-reduce conflicts must be resolved

LALR analysis

- LALR = Look-Ahead LR parser
- Simplified right analyzer
  - LR(1) analyzer: stronger than LR(0)
  - LR(0) state space: smaller state space than LR(1)
- Can be generated automatically
  - E.g., Yacc, Bison

This lecture: Syntax analysis

- **I. Context free (CF) grammars**
- **II. Naïve methods (BFS, DFS)**
- **III. Left analysis: LL(k), LL(*)**
- **IV. Right analysis: LR(k), LALR**
- **V. Error handling**

Error handling

- Goal of syntax analysis:
  - production of the syntax tree or finding as many errors as possible
- On error:
  - error message, return to consistent state, continue analysis
- Report:
  - position (file name, line, character)
  - severity (error, warning, info, etc.)
  - error message

Error position vs. error symptom

Error symptom:
`)` is missing before `;`

```
x = (a+b*c;
```

Error (probably):
`)` missing before `*`

Position of the symptom is not necessary the same as the position of the error!

Errors in various phases of compilation

- Lexical error:
  - e.g., invalid character, missing end of a string or comment, premature end of file
- Syntax error:
  - code is not valid according to the grammar
  - correction: with the fewest operations (inserting, deleting or replacing a token)
- Semantic error:
  - error in the static semantics (e.g., type error)
  - some of these can only be detected during optimization (pl. invalid indexing)
- Lack of resources:
  - e.g., insufficient memory
  - can occur in any phase

Thank you!