Syntactic Analysis I

Masters in Informatics and Computing Engineering (MIEIC), 3rd Year

João M. P. Cardoso
Dep. de Engenharia Informática, Faculdade de Engenharia (FEUP), Universidade do Porto, Porto, Portugal
Email: jmpc@fe.up.pt

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Syntax in Programming Languages

- Syntax of programming languages cannot be handle by regular expressions (programming languages are not regular languages)
- Why?
  - (a+(b-c))*(d-(x-(y-z)))
  - if (x < y) if (y < z) a = 5 else a = 6 else a = 7
- Regular languages don’t have the required state to model nesting!
- There is none regular expression to specify expressions with balanced parenthesis!

Solution

- Context Free Grammars (CFGs)
  - Recognition done by Finite Automata with a stack (known as Push Down Automata: PDAs)
  - Or...

Grammars

- Backus Naur Form (BNF) notation to specify grammars:
  - **Terminal symbols**: Uppercase letters
  - **Non-terminal symbols**: start by an uppercase letters (or delimited by < and >)
  - In a production the left and the right side are separated by $\rightarrow$ or ::=, E.g.:
    - Expr $\rightarrow$ Term OP Term
    - Expr ::= Term OP Term
  - Alternative productions (p_1, p_2, p_3, ..., p_n) are represented by p_1 | p_2 | p_3 | ... | p_n
    - E.g.: Literal $\rightarrow$ BINARIO | OCTAL | INT | FLOAT
  - If the right hand size of a production does not contain any symbol then we write ε
    - E.g.: Palavra $\rightarrow$ ε

Grammars

- EBNF, or extended BNF
  - Includes { } to represent 0 or more occurrences
  - and [ ] to represent optional elements
  - , is used to separate gramar elements in the same production

BNF: https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_Form

EBNF: https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form

Context Free Grammars (CFGs)

- Set of terminal symbols

  { OP, INT, OPEN, CLOSE }

  Each terminal symbol defined by a regular expression
- Set of non-terminal symbols

  { Start, Expr }
- Set of productions
  - A single non-terminal symbol in the left hand side (LHS)
  - Sequence of terminal and non-terminal symbols in the right hand side (RHS)

OP = + | - | * | /

INT = [0-9] [0-9]*

OPEN = (

CLOSE =)

Start → Expr

Expr → Expr OP Expr

Expr → INT

Expr → OPEN Expr CLOSE

Production/Derivation Game

Given a string:

Repeat until there are none terminal symbols

- Select a non-terminal symbol (start by the non-terminal symbol Start)
- Select a production for that non-terminal symbol
- Substitute the non-terminal symbol with the RHS of the production

Substitute the regular expression with the correspondent strings

Generated string belongs to the language

Note: different selections produce different Strings

Production/Derivation

OP = +|-|*|/
INT = [0-9] [0-9]*
OPEN = (
CLOSE =)

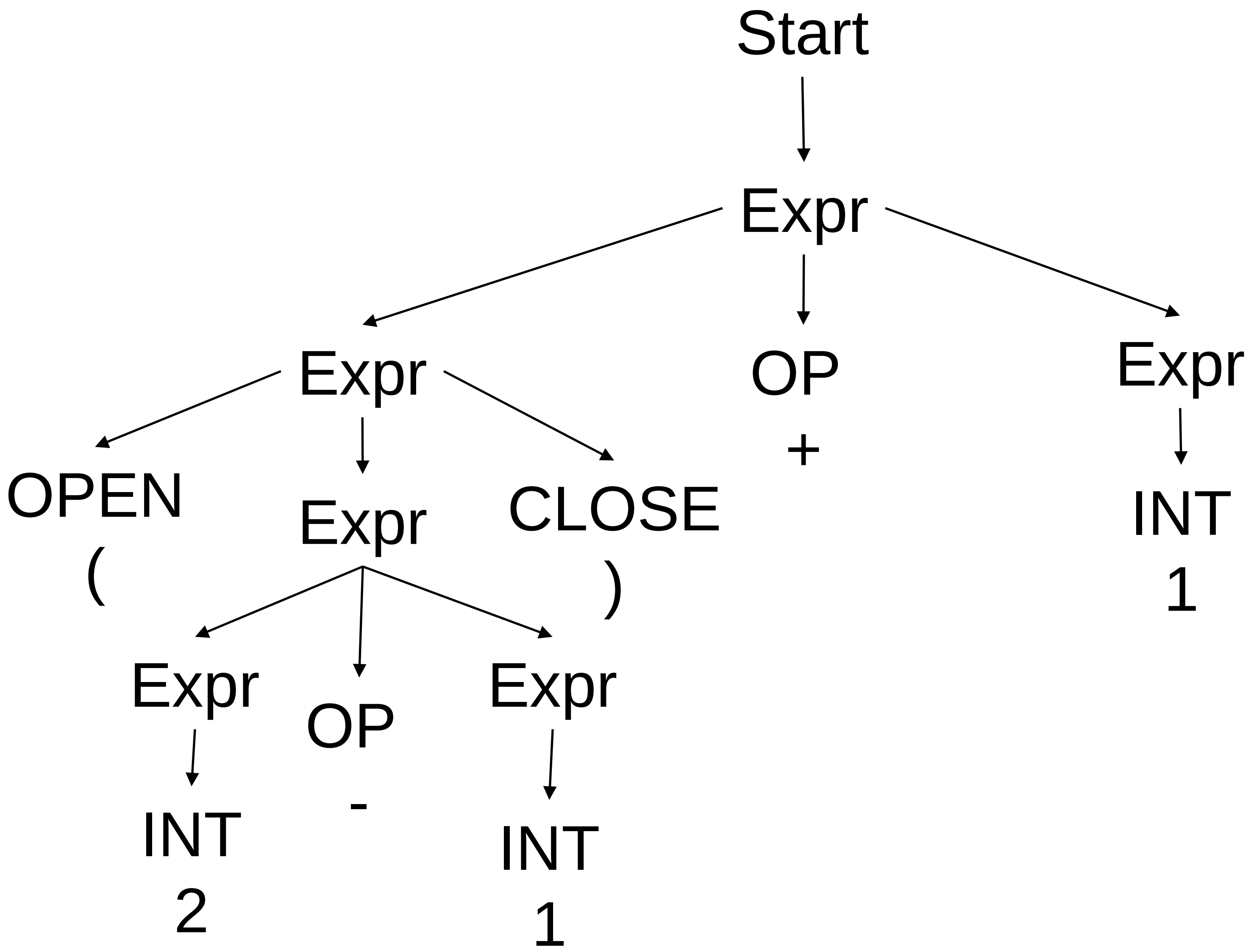
1) Start → Expr
2) Expr → Expr OP Expr
3) Expr → INT
4) Expr → OPEN Expr CLOSE

Start
Expr
Expr OP Expr
OPEN Expr CLOSE OP Expr
OPEN Expr OP Expr CLOSE OP Expr
OPEN INT OP Expr CLOSE OP Expr
OPEN INT OP Expr CLOSE OP INT
OPEN INT OP INT CLOSE OP INT
(2 - 1) + 1

Syntax Tree

- Internal nodes: non-terminal symbols
- Leaves: terminal symbols
- Edges:
  - From non-terminal symbols of the LHS of the production
  - To nodes of the RHS of the production
- Captures the derivation of a String accepted by the language

Syntax Tree for (2-1)+1



OP = +|-|*|/

INT = [0-9] [0-9]*

OPEN = (

CLOSE =)

1) Start → Expr

2) Expr → Expr OP Expr

3) Expr → INT

4) Expr → OPEN Expr CLOSE

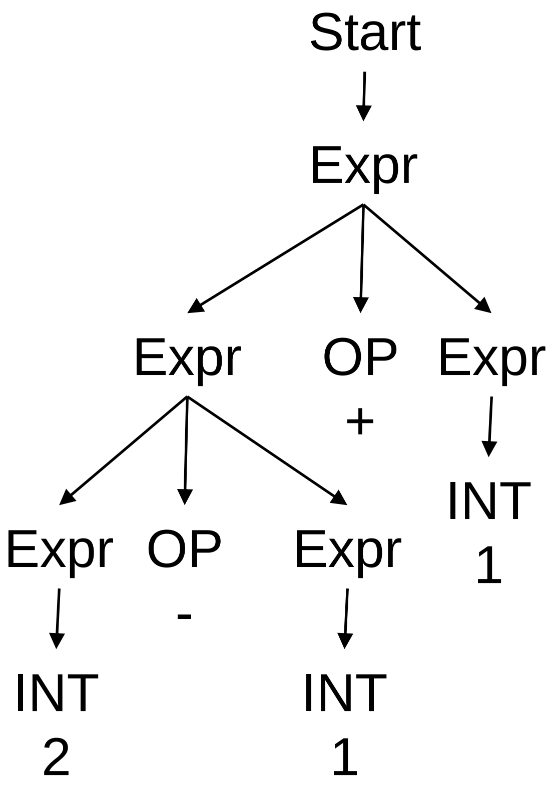
Ambiguity in a Grammar

- Multiple leftmost or rightmost derivations (implying multiple syntax trees) for the same String
- Syntax tree usually reflect semantic of the program
- Ambiguity in the grammar reflects many times ambiguity in terms of semantic (considered undesirable)

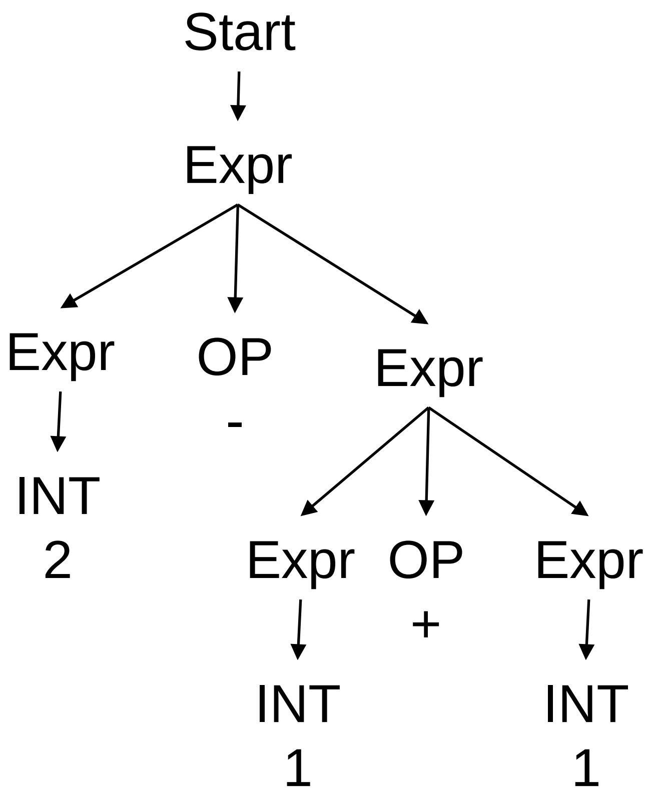
Example of Ambiguity

- Two syntax tree for 2-1+1

Tree corresponding to (2-1)+1



Tree corresponding to 2-(1+1)



Eliminating Ambiguity

- Solution: modify grammar
- All operators with left-associative

Different language!

Original Grammar

Start → Expr
Expr → Expr OP Expr
Expr → INT
Expr → OPEN Expr CLOSE

Modified Grammar

Start → Expr
Expr → Expr OP INT
Expr → INT
Expr → OPEN Expr CLOSE

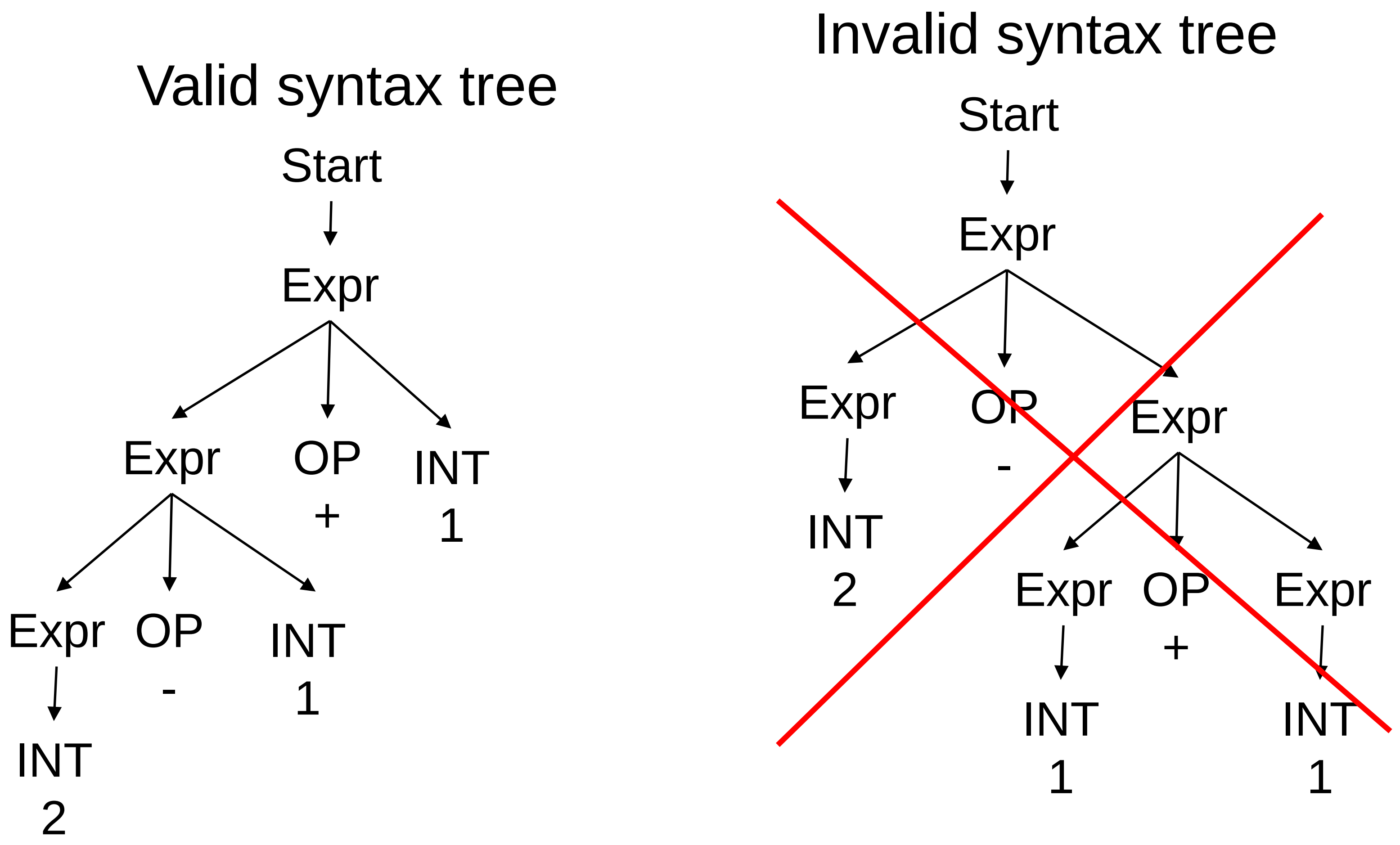
Eliminating Ambiguity

- Solution: modify grammar
- All operators with left-associative

Original Grammar	**Modified Grammar**
Start → Expr	Start → Expr
Expr → Expr OP Expr	Expr → Expr OP Expr’
Expr → INT	Expr → INT
Expr → OPEN Expr CLOSE	Expr → OPEN Expr CLOSE
	Expr’ → OPEN Expr CLOSE
	Expr’ → INT

Syntax Tree

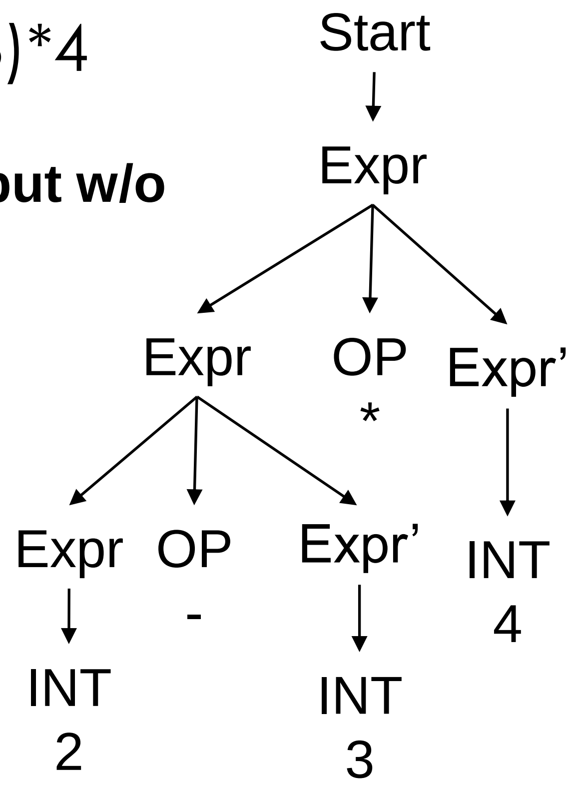
- Only one syntax tree for: 2-1+1



Precedence of Operators

- All operators with left-associative
- Without respecting the precedence of * over +
- 2-3*4 interpreted as (2-3)*4

Syntax tree for 2-3*4



Modified Grammar (w/o ambiguity but w/o respecting precedence)

Start → Expr
Expr → Expr OP Expr'
Expr → INT
Expr → OPEN Expr CLOSE
Expr' → OPEN Expr CLOSE
Expr' → INT

Grammar equivalent:

Start → Expr
Expr → Expr OP Expr'
Expr → Expr'
Expr' → OPEN Expr CLOSE
Expr' → INT

Solution to Precedence

Original Grammar

OP = + | - | * | /

INT = [0-9] [0-9]*

OPEN = (

CLOSE =)

Start → Expr

Expr → Expr OP Expr'

Expr → Expr'

Expr' → OPEN Expr CLOSE

Expr' → INT

Modified Grammar

OP1 = + | -

OP2 = * | /

INT = [0-9] [0-9]*

OPEN = (

CLOSE =)

Start → Expr

Expr → Expr OP1 Term

Expr → Term

Term → Term OP2 Final

Term → Final

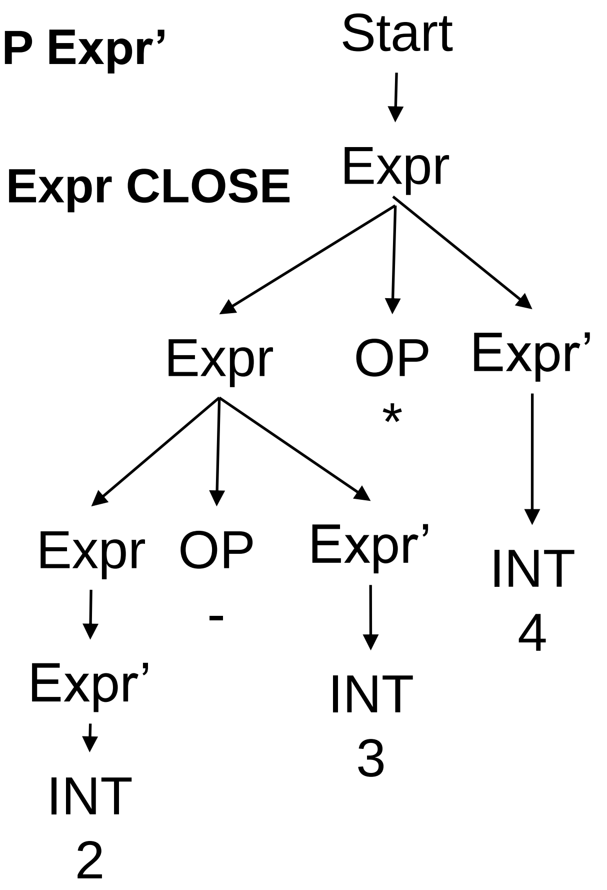
Final→ INT

Final→ OPEN Expr CLOSE

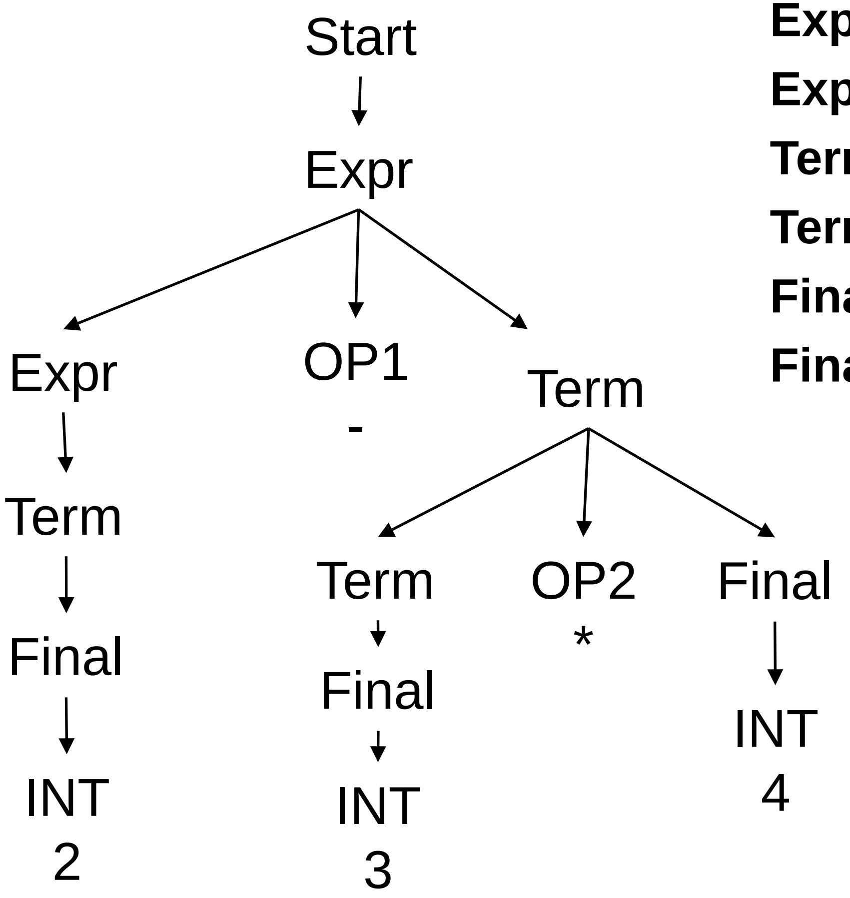
Modification in Syntax Tree

Start → Expr
Expr → Expr OP Expr'
Expr → Expr'
Expr' → OPEN Expr CLOSE
Expr' → INT

Old syntax tree for
2-3*4



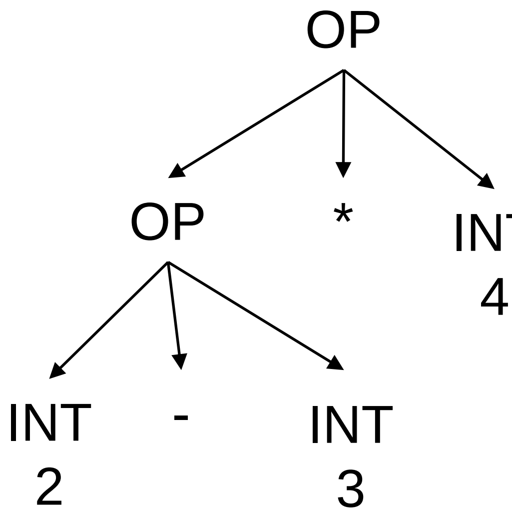
New syntax tree for
2-3*4



Start → Expr
Expr → Expr OP1 Term
Expr → Term
Term → Term OP2 Final
Term → Final
Final→ INT
Final→ OPEN Expr CLOSE

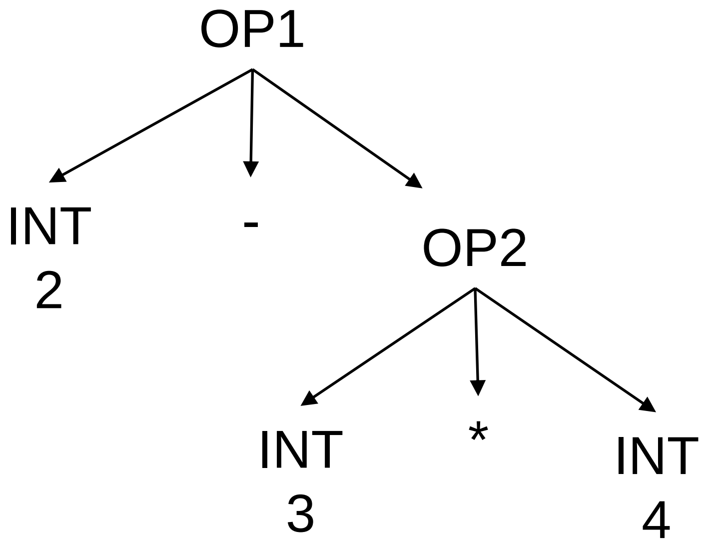
Modification in Syntax Tree

Old AST for
2-3*4



Does not respect precedence!

New AST for
2-3*4



It respects precedence!

Global Idea

- Group operators by precedence levels
  - * and / are at the top level
  - + and – are in the next level
- Non-terminal symbol for each precedence level
  - Term is non-terminal for * and /
  - Expr is non-terminal for + and -
- In each level we can do right or left associativity
- Generalize to additional levels of precedence (according to the needs)

Exercise

- Specify using BNF grammars corresponding to the regular expressions: [0-9]+ e [0-9]*
- Given the grammar:

NUM = [0-9]+

ID = [A-Za-Z][0-9A-Za-z]*

Expr → Expr "+" Term | Expr "–" Term | Term

Term → Term "*" Factor | Term "/" Factor | Factor

Factor → Primary "^" Factor | Primary

Primary → "-"Primary | Element

Element → "(" Expr ")" | NUM | ID

- Show the syntax trees for:
  - 5-2*3
  - y^3

Handling if-then-else Constructs

Start $\rightarrow$ Stat

Stat $\rightarrow$ IF Expr THEN Stat ELSE Stat

Stat $\rightarrow$ IF Expr THEN Stat

Stat $\rightarrow$ ...

Start → Stat
Stat → IF Expr THEN Stat ELSE Stat
Stat → IF Expr THEN Stat
Stat → ...

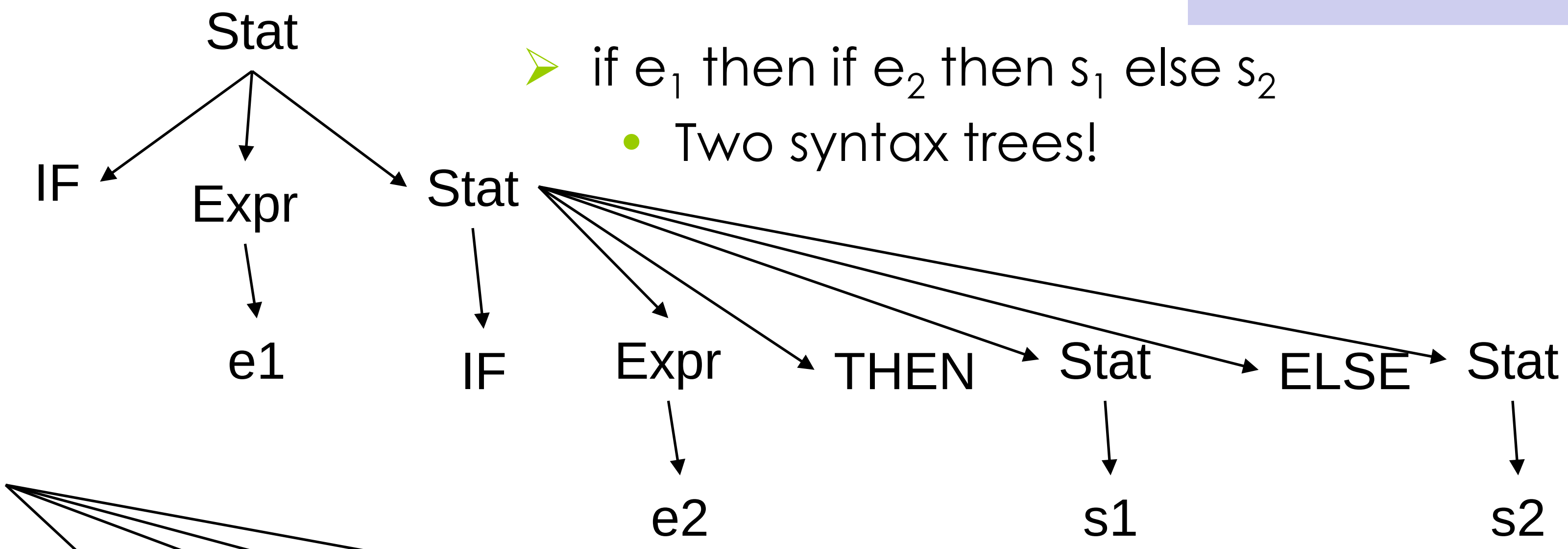
Syntax Tree

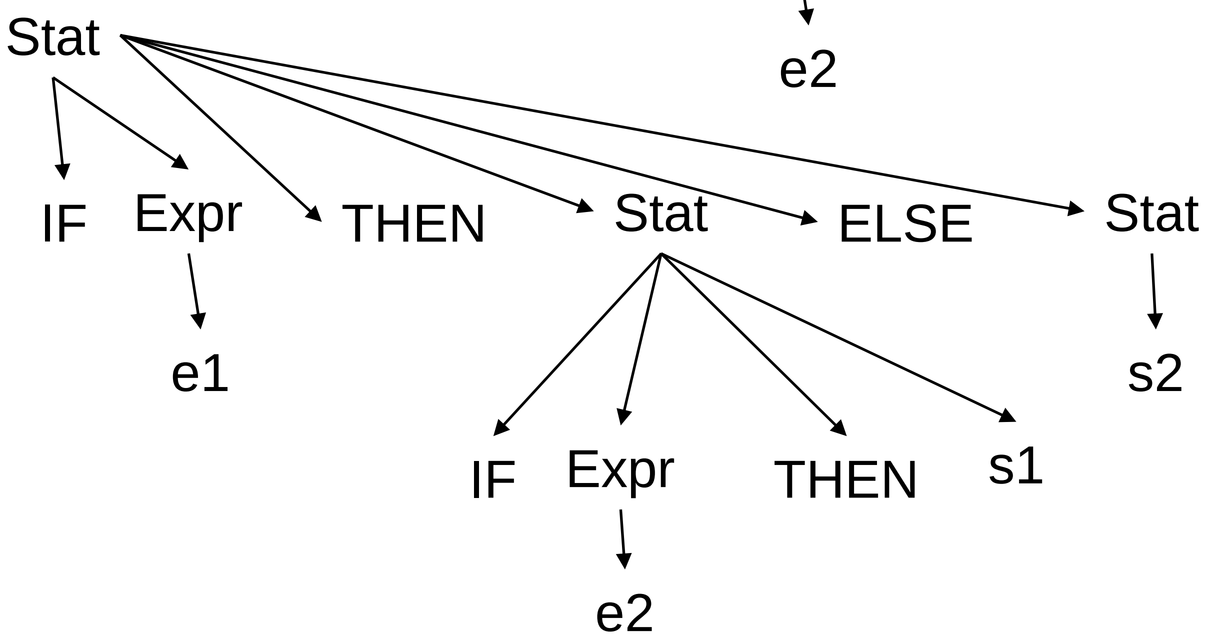
- Consider the statement:
  - if e_1 then if e_2 then s_1 else s_2

Syntax Tree

Start → Stat
Stat → IF Expr THEN Stat ELSE Stat
Stat → IF Expr THEN Stat
Stat → ...

- if e_1 then if e_2 then s_1 else s_2
  - Two syntax trees!





Which one is the correct?

Alternative Interpretations

- Ambiguous grammar
  - For the same statement:
    - if e_1 then if e_2 then s_1 else s_2
  - Syntax tree 1

if e_1

    if e_2 s_1

    else s_2

  - Syntax tree 2

if e_1

    if e_2 s_1

else s_2

Modified Grammar

Start → Stat
Stat → IF Expr THEN Stat ELSE Stat
Stat → IF Expr THEN Stat
Stat → ...

- Basic Idea: control when an IF without ELSE can occur
  - At the top level of the statements
  - Or as te last in a sequence of statements if then else if then ...

Goal → Stat
Stat → WithElse
Stat → LastElse

WithElse → IF Expr THEN WithElse ELSE WithElse
WithElse → ...

LastElse → IF Expr THEN Stat
LastElse → IF Expr THEN WithElse ELSE LastElse

Syntactic Analyzer

- Convert programs in a syntax tree
- It can be developed from the scratch!
- Or developed automatically by a parser generator
  - Accept a grammar as input
  - Produce the syntactic analyzer as output
- Parctical problem
  - The syntax tree for the modified grammar can be complex
  - We would like to start with a syntax tree

Solution

- Abstract vs Concrete Tree
  - Abstract syntax corresponds to an intuitive way to think the program structure
    - Omit detials as superfluous keywords
    - Abstract syntax can be ambigous
  - Concrete syntax corresponds to the complete grammar used to anlyze syntacticaly the language
- The syntax analyzer are many times programmed to generate Abstract Syntax Trees (ASTs)

Abstract Syntax Trees (ASTs)

- Start with an intuitive grammar but possibly ambigous
- Modify the grammar to make it non-ambiguous
  - Concrete Syntax Trees
  - Less intuitive
- Convert the concrete syntax tree (CST) in ASTs
  - They correpsonde to the intuitive agrammar for the language
  - Simpler to manipulate by the compiler

Example

Non-ambiguous grammar:

OP1 = + | -

OP2 = * | /

INT = [0-9] [0-9]*

OPEN = (

CLOSE =)

Start → Expr

Expr → Expr OP1 Term

Expr → Term

Term → OPEN Expr CLOSE

Term → Term OP2 INT

Term → INT

Intuitive grammar but ambiguous:

OP = * | / | + | -

INT = [0-9] [0-9]*

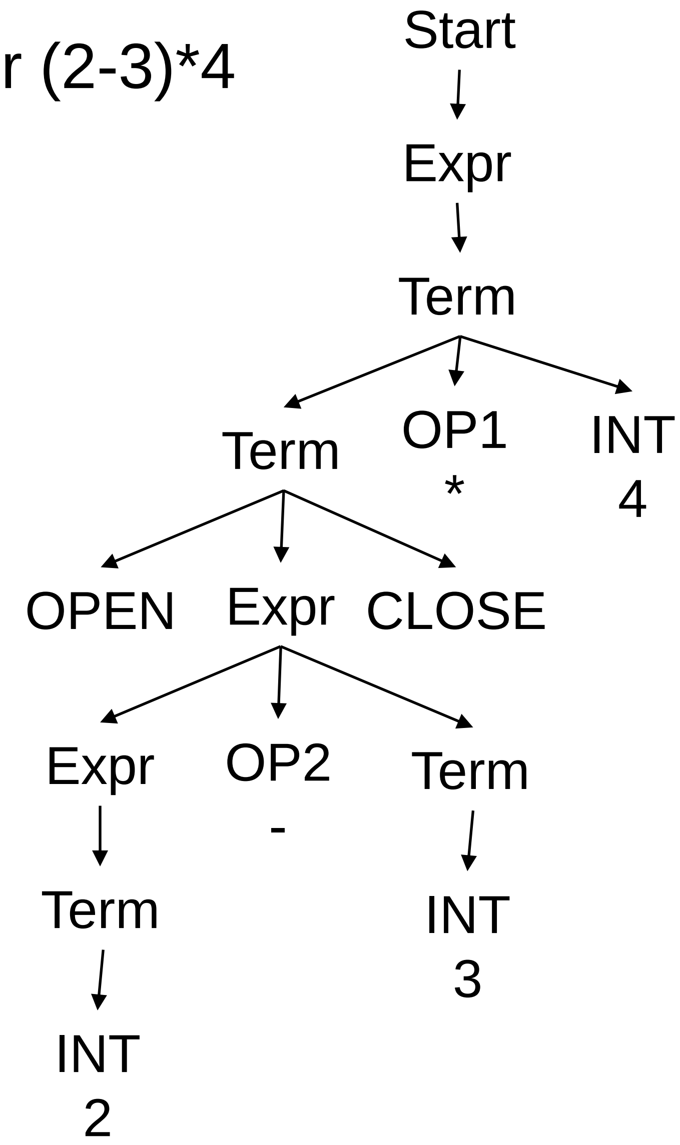
Start → Expr

Expr → Expr OP Expr

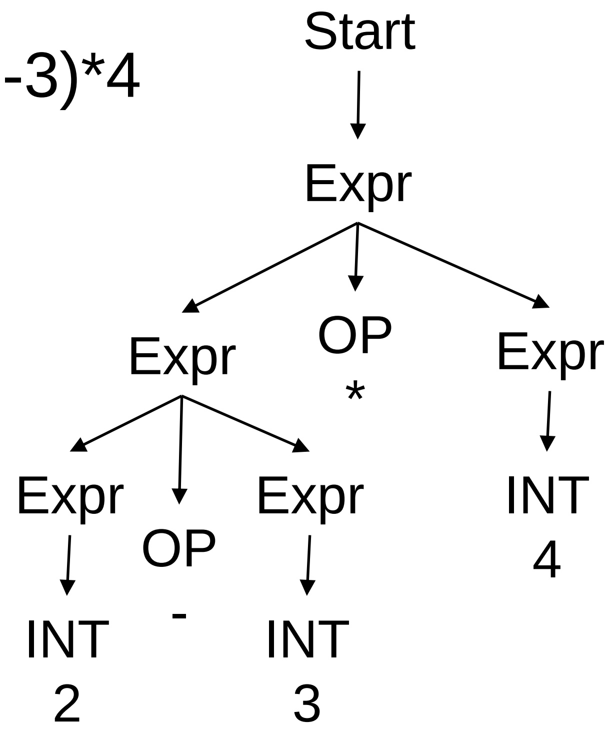
Expr → INT

Example

CST for (2-3)*4



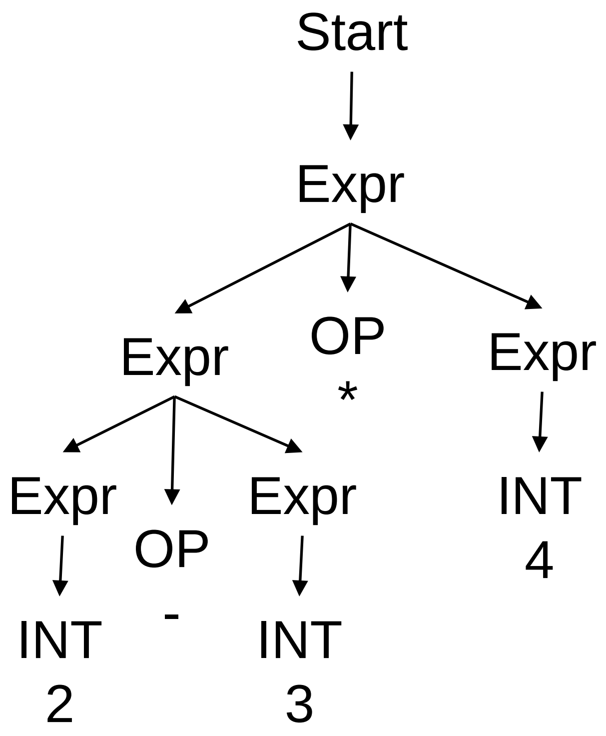
AST for (2-3)*4



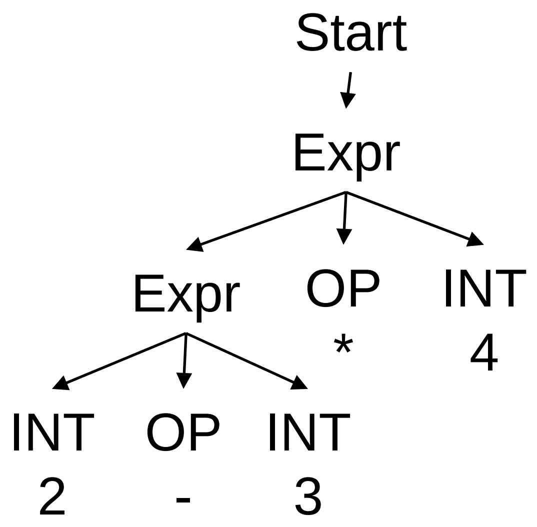
- Close to intuitive grammar
- Eliminates superfluous tokens
  - OPEN, CLOSE, etc.

Example

AST for (2-3)*4



AST for (2-3)*4
(even more simplified!)



Summary

- Levels of lexical and syntactic structures
  - Lexical – regular expressions
  - Syntactic– grammars
- Ambiguities in the grammar
  - Modified grammars
- Abstract Syntex Tree (ASTs)
- Generative vs recognizing roles
  - Generative more convenient for specifications
  - Recognizing required for implementation

Grammar Vocabulary

- Leftmost derivation
  - Always expand the non-terminal symbol that remains in the left

$$NT_1\ T_1\ T_2\ T_3\ NT_2\ NT_3$$

Apply production here first!

And only after to the other non-terminal symbols

- Rightmost derivation
  - Always expand the non-terminal symbol that remains in the right

Initial Point

- Assume that the lexical analysis produced a **chain of tokens (terminal symbols)**
  - Each token has a type and a value
  - Types correpond to terminal symbols
  - Values correspond to the content of the token read

- Examples
  - INT(549) – token that identifies an integer with value read 549
  - IF – keyword “if” without necessaty of value
  - OP(+) – Operator with value: +

Basic Approach

- Given a **chain of tokens** as input
- Start with the start variable of the grammar
- Build a *leftmost* derivation
  - If the *leftmost* symbol is a non-terminal, select a production and apply it
  - If *leftmost* symbol is terminal, tray match with the input
  - If all the terminals were matched there was found a derivation that accepts the String!
  - Key: find the correct productions for the non-terminal symbols

Grammar Example

INT = [0-9]+

Start $\rightarrow$ Expr

Expr $\rightarrow$ Expr “+” Term

Expr $\rightarrow$ Expr “-” Term

Expr $\rightarrow$ Term

Term $\rightarrow$ Term “*” INT

Term $\rightarrow$ Term “/” INT

Term $\rightarrow$ INT

- Set of tokens (terminal symbols):

  { +, -, *, /, INT }

Syntactic Analyzer for the Grammar Example

Chain of Tokens:
INT(2) ‘-’ INT(2) ´*´INT(2)

INT = [0-9]+

Start →
  Expr

Expr →
  Expr “+”
  Term

Expr →
  Expr “-”
  Term

Expr →
  Term

Syntactic Tree

Start

Current position in the syntax tree

Input not processed
2-2*2

Sentencial form
Start

Syntax Analyzer for the Grammar Example

Syntax tree

Start

↓

Expr

↑

Current position in the syntax tree

Input not processed

2-2*2

Sentencial Form

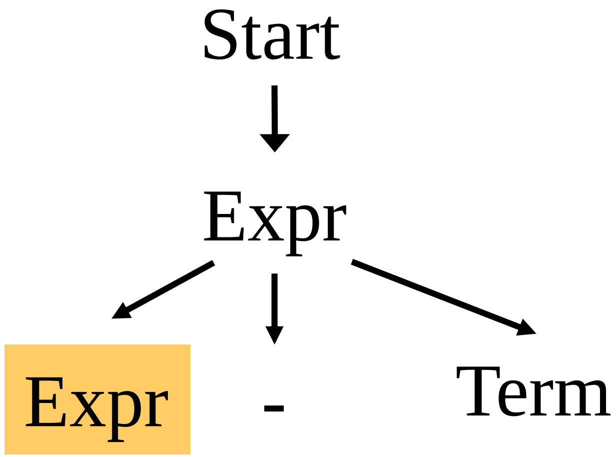
Expr

Applied prodction

Start → Expr

Syntax Analyzer for the Grammar Example

Syntax Tree



Input not processed

2-2*2

Sentencial form

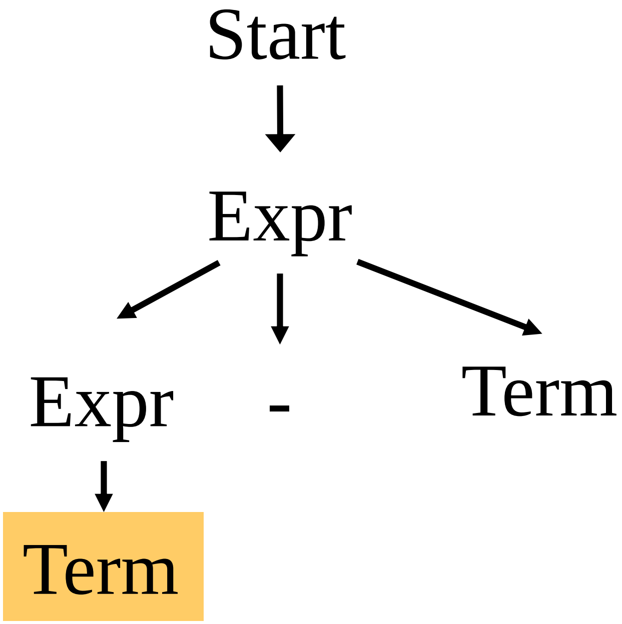
Expr - Term

Applied production

Expr $\rightarrow$ Expr - Term

Syntax Analyzer for the Grammar Example

Syntax Tree



Input not processed

2-2*2

Sentencial form

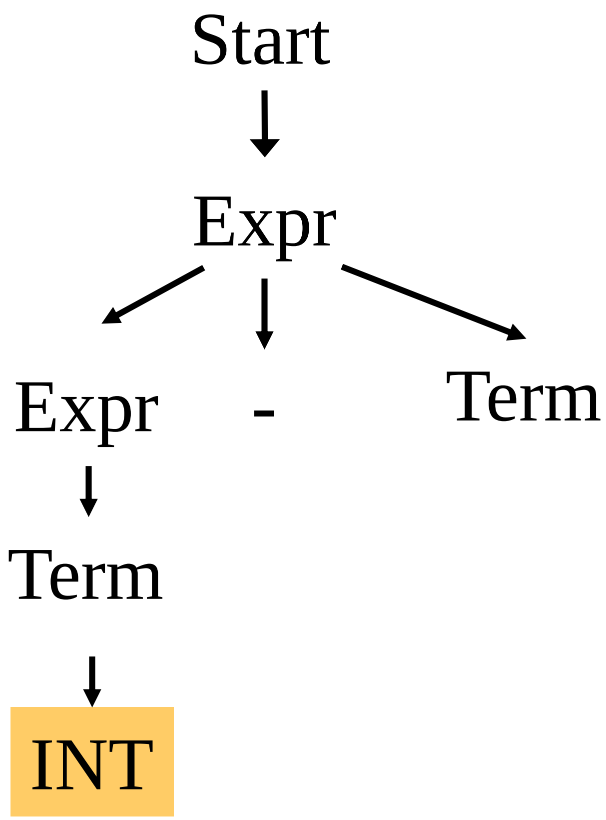
Term - Term

Applied production

Expr → Term

Syntax Analyzer for the Grammar Example

Syntax Tree



Input not processed

2-2*2

Sentencial form

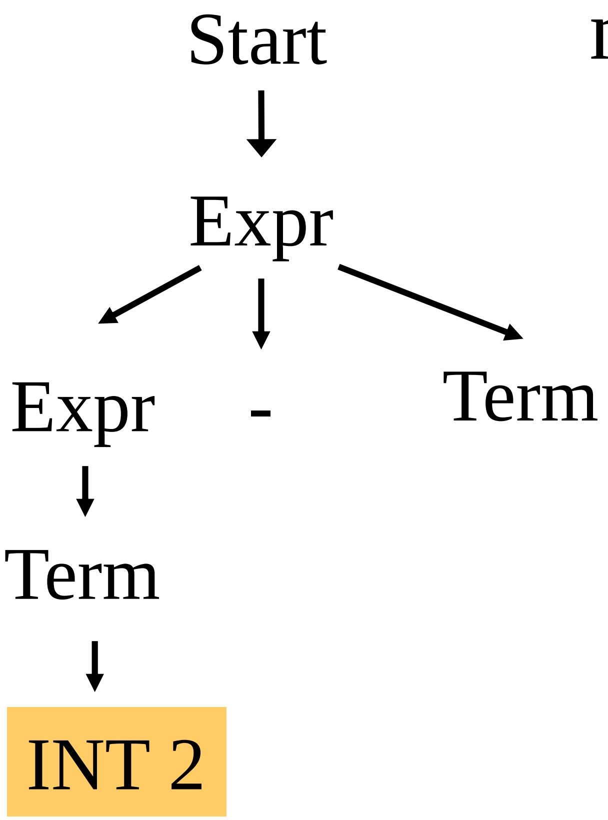
INT - Term

Applied production

Term → INT

Syntax Analyzer for the Grammar Example

Syntax Tree



Token matches!

Input not processed

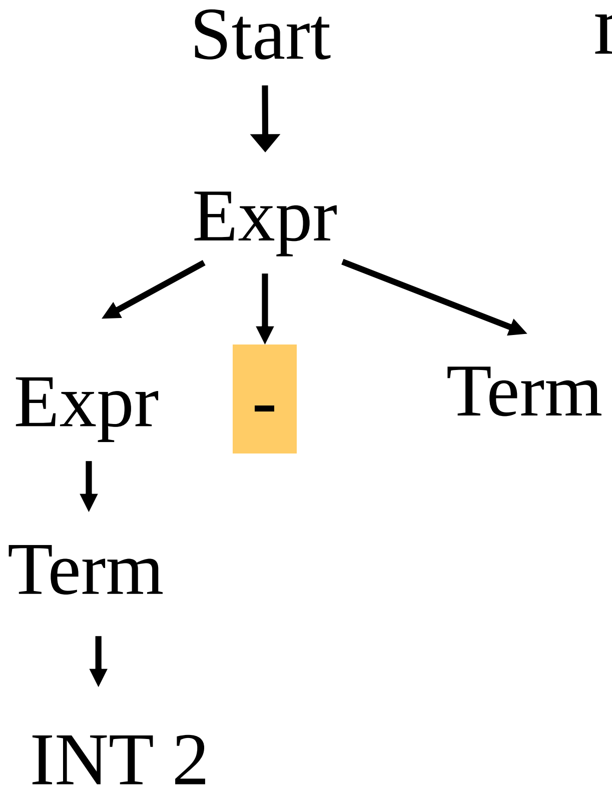
2-2*2

Sentencial form

2 - Term

Syntax Analyzer for the Grammar Example

Syntax Tree



Token matches!

Input not processed
-2*2

Sentencial form
2 - Term

Syntax Analyzer for the Grammar Example

Syntax Tree

Start

Expr

Expr - Term

Term

INT 2

Input not processed

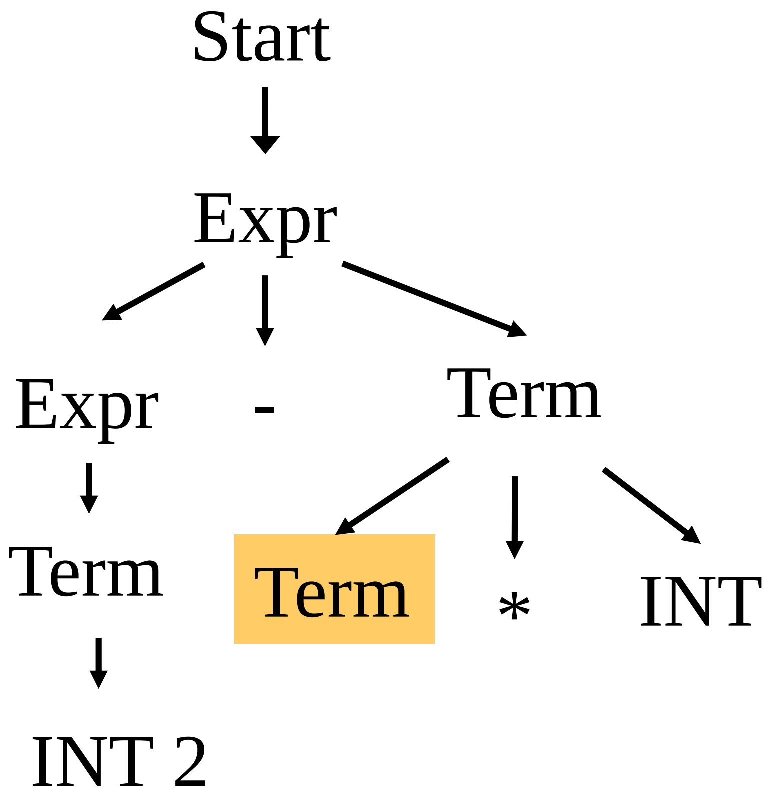
2*2

Sentencial form

2 - Term

Syntax Analyzer for the Grammar Example

Syntax Tree



Input not processed

2*2

Sentencial form

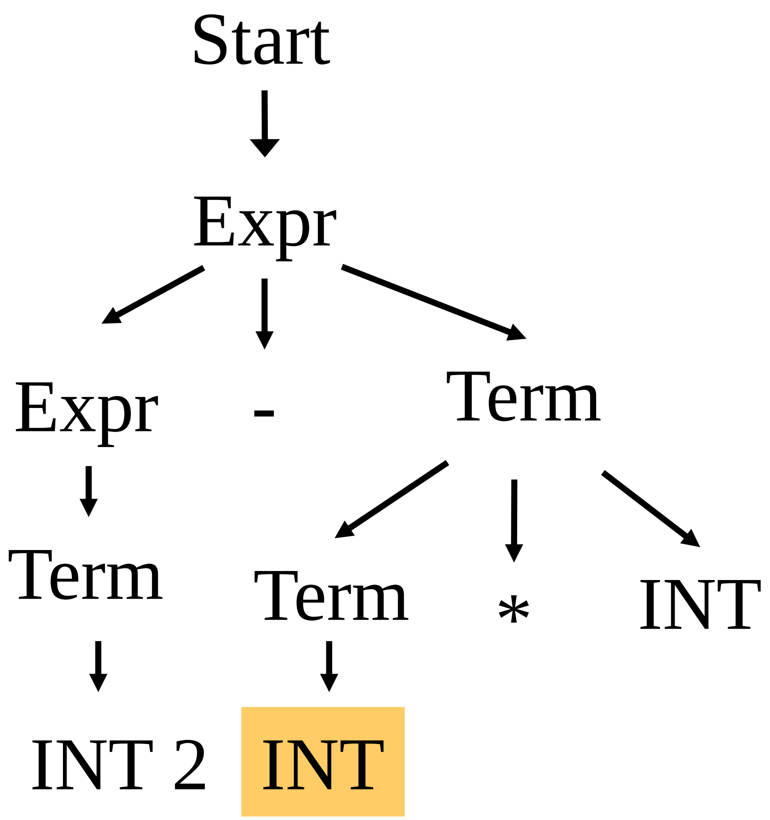
2 – Term*INT

Applied production

Term → Term * INT

Syntax Analyzer for the Grammar Example

Syntax Tree



Input not processed

2*2

Sentencial form

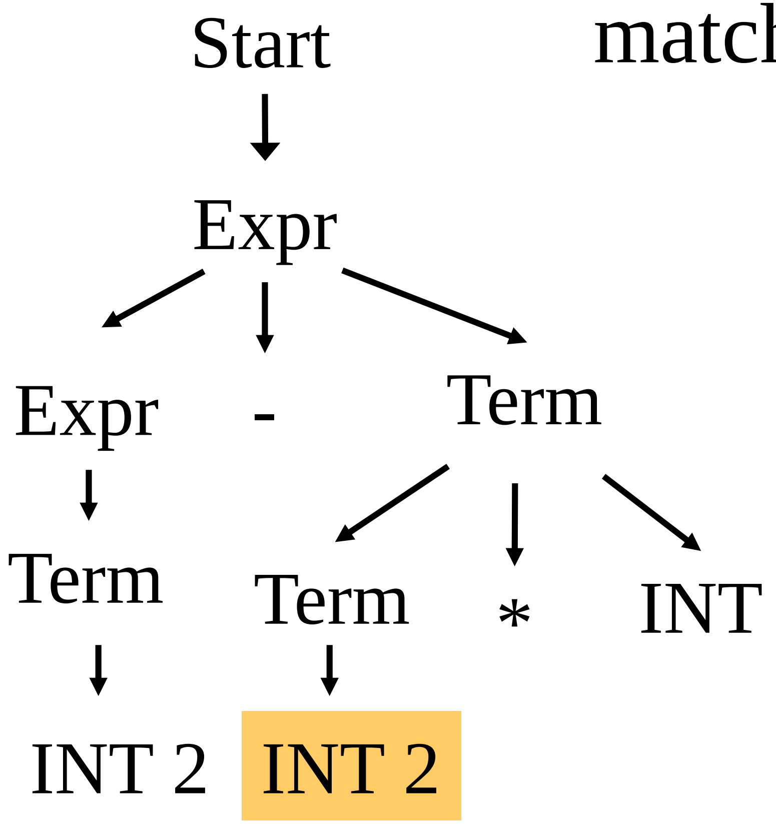
2 – INT*INT

Applied production

Term → INT

Syntax Analyzer for the Grammar Example

Syntax Tree



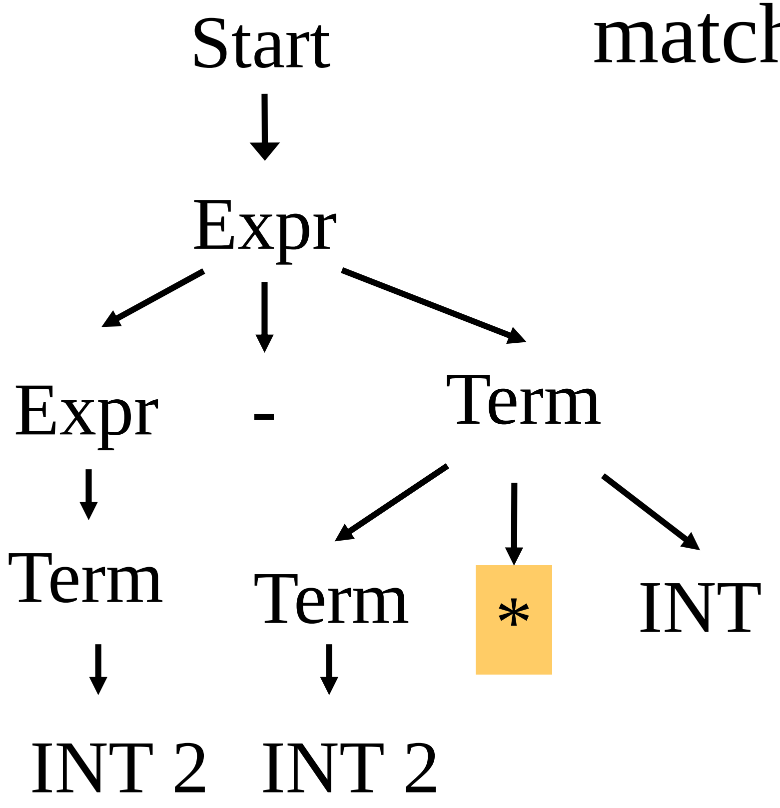
Token matches!

Input not processed
2*2

Sentencial form
2 – 2*INT

Syntax Analyzer for the Grammar Example

Syntax Tree



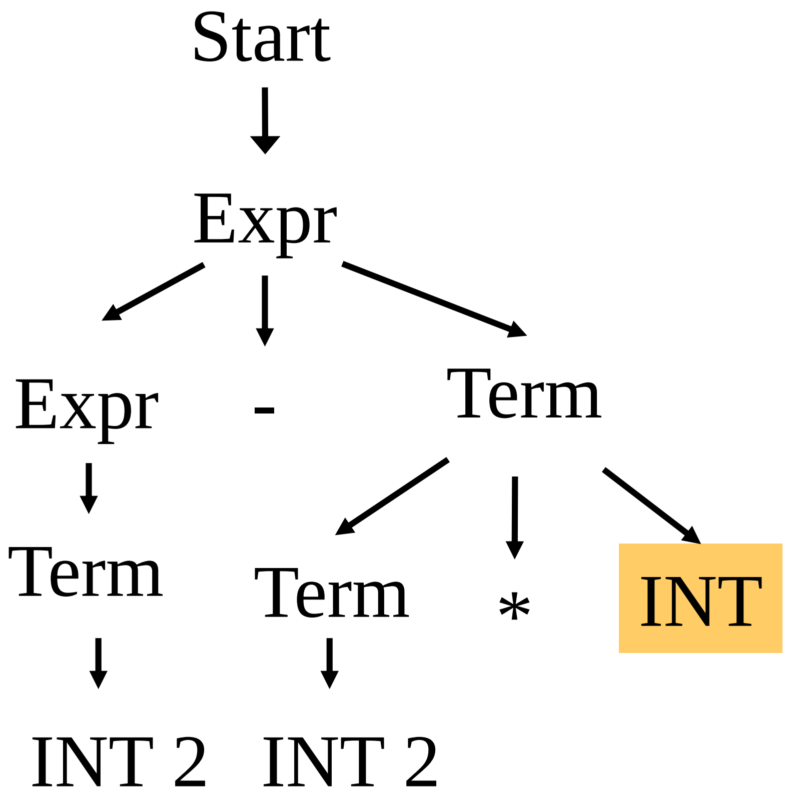
Token matches!

Input not processed
*2

Sentencial form
2 – 2*INT

Syntax Analyzer for the Grammar Example

Syntax Tree



Input not processed

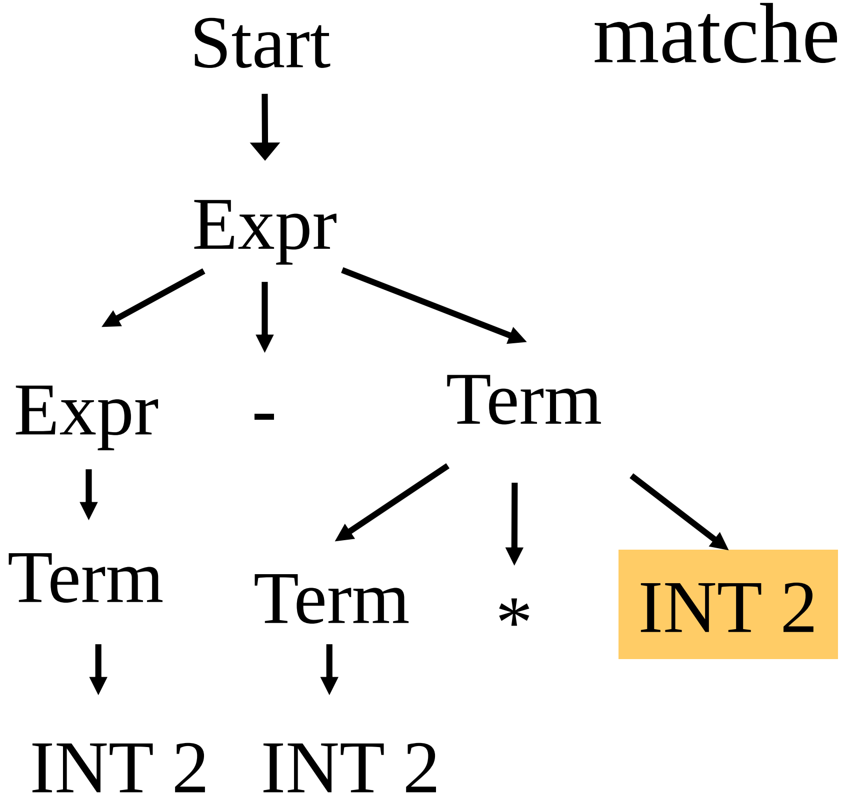
2

Sentencial form

2 – 2*INT

Syntax Analyzer for the Grammar Example

Syntax Tree



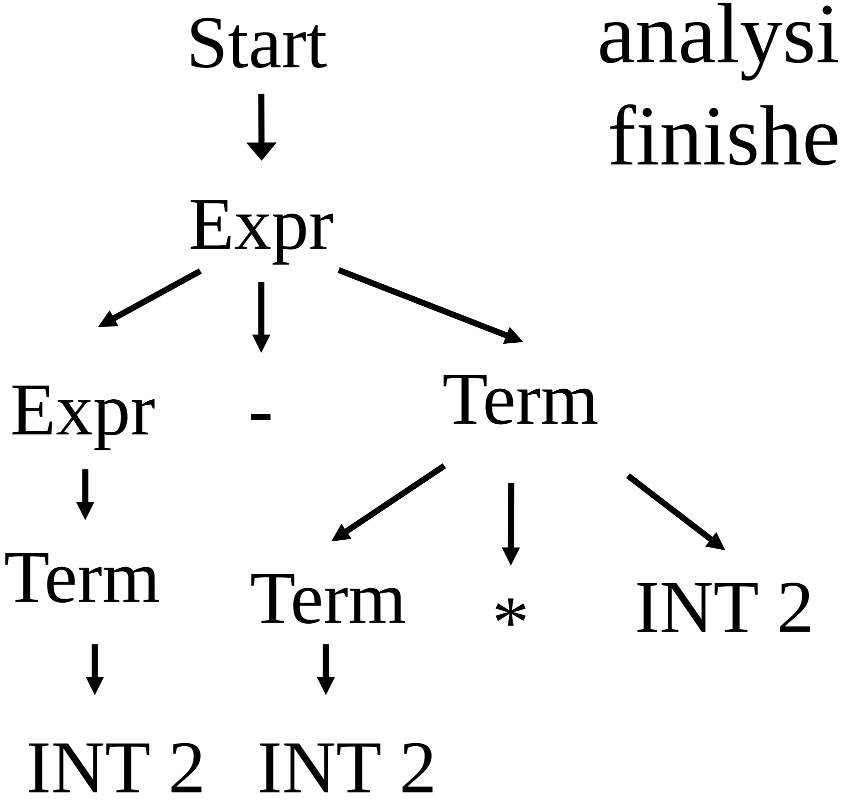
Token matches!

Input not processed
2

Sentencial form
2 – 2*2

Syntax Analyzer for the Grammar Example

Syntax Tree



Syntactic analysis finished

Input not processed

Sentencial form

2 – 2*2

Summary

- Three actions (mechanisms)
  - Apply production to expand the current non-terminal symbol at the syntax tree
  - Match the current terminal symbol
  - Accept the syntax analysis as correct
- What is the production to be used for each non-terminal symbols?
- An approach: *Backtracking*
  - Try an alternative
  - When it is clear the alternative failed then try another alternative

Predictive Syntactic Analyzer

- Alternative to *backtracking*
- Very useful for programming languages that can be designed to make easier the analysis
- Basic idea:
  - Lookahead in the sequence of tokens
  - Value k of lookahead corresponds to the next k tokens in the chain of tokens being seen
  - Decision about the production to apply is based on the following tokens
  - We use the token level in the lookahead mechanism

Grammar Example

Start $\rightarrow$ Expr

Expr $\rightarrow$ Term Expr’

Expr’ $\rightarrow$ “+” Term Expr’

Expr’ $\rightarrow$ “-” Term Expr’

Expr’ $\rightarrow$ ε

Term $\rightarrow$ INT Term’

Term’ $\rightarrow$ “*” INT Term’

Term’ $\rightarrow$ “/” INT Term’

Term’ $\rightarrow$ ε

INT = [0-9]+

- Set of tokens (terminal symbols):

{ +, -, *, /, INT }

Points of Selection

- Assume that Term’ is the current position in the syntax tree
- 3 different productions to apply

  Term’ $\rightarrow$ “*” INT Term’

  Term’ $\rightarrow$ “/” INT Term’

  Term’ $\rightarrow \varepsilon$
- Use the next Token to decide (i.e., lookahead =of 1)
  - If next token is *, apply Term’ $\rightarrow$ * Int Term’
  - If next token is /, apply Term’ $\rightarrow$ / Int Term’
  - Otherwise, apply Term’ $\rightarrow \varepsilon$

Multiple Productions with the Same RHS Prefix

- Grammar example:

  Nt $\rightarrow$ IF THEN

  Nt $\rightarrow$ IF THEN ELSE

- Assume that Nt *is the* sintax *tree* and IF is the next token

- Which is the production to apply?
- Value of lookahead needed?

Multiple Productions with the Same RHS Prefix (cont.)

- Solution: Factoring the Grammar
- New grammar include the common prefix in a single production:

  Nt → IF THEN Nt'
  Nt' → ELSE
  Nt' → ε

  Nt → IF THEN
  Nt → IF THEN ELSE

- None selection when next token is an IF
- Alternatives were unified in a single production

- Use a lookahead greater than 1?

Multiple Productions with the Same RHS Prefix (cont.)

- What about the productions with non-terminal symbols?

  $Nt \rightarrow Nt_1\ \alpha_1$

  $Nt \rightarrow Nt_2\ \alpha_2$

- We have to select based on the first possible terminals Nt_1 and Nt_2 can derive
- And if Nt_1 or Nt_2 can derive ε?
  - We have to select based on α_1 and α_2

FIRST AND FOLLOW SETS

Definitions: First() and Follow() Sets

Notation

T os a terminal,

NT is a non-terminal,

S is terminal or non-terminal,

and α and β represent sequences with terminals and/or non-terminals

- First(β) Set
  - Set of leftmost terminal symbols in all possible derivation trees of β
  - T$\in$ First(β) if T can appear as a first symbol of a derivation strating in β
  - Start with the concept of *NT* deriving ε:
    - NT $\rightarrow$ ε implies that NT derives ε
    - NT $\rightarrow$ NT_1 ... NT_n and if all NT_i ($1 \leq i \leq n$) derive ε implies that NT derives ε

Rules for First()

Notation

- T is a terminal
- NT is a non-terminal
- S is terminal or non-terminal
- α and β represent sequences with terminals and/or non-terminals

1) $T \in First(T)$
2) $First(S) \subseteq First(S\ \beta)$
3) NT derives ε implies:

   $First(\beta) \subseteq First(NT\ \beta)$
4) $NT \rightarrow S\ \beta$ implies:

   $First(S\ \beta) \subseteq First(NT)$

First() Example

- First(Term')?

Grammar

Term' $\rightarrow$ * INT Term'

Term' $\rightarrow$ / INT Term'

Term' $\rightarrow$ ε

Solution

First(Term') = {*,/, ε}

First(* INT Term') = {*}

First(/ INT Term') = {/}

First(*) = {*}

First(/) = {/}

First() Set

- If two or more different productions for the same non-terminal symbol have First sets with common terminal symbols then:
  - The grammar cannot be analysed with a predictive LL(1) parser without backtracking
    - Example:
    - $S \rightarrow X \$$
    - $X \rightarrow a$
    - $X \rightarrow a\ b$

    - First($X \rightarrow a$) = { a }
    - First($X \rightarrow a\ b$) = { a }
    - **Which production to choose when the currrent symbol is a?**

Follow() Set

- For the non-terminal A, Follow(A) is the set of the first terminals that can follow after A in a derivation
- Rules for Follow()
  - $\$ \in$ Follow(S), where S is the start symbol
  - If $A \rightarrow \alpha B \beta$ is a production then

    First(β) $\subseteq$ Follow(B)
  - If $A \rightarrow \alpha B$ is a production then

    Follow(A) $\subseteq$ Follow(B)
  - If $A \rightarrow \alpha B \beta$ is a production and β derives ε then

    Follow(A) $\subseteq$ Follow(B)

Algorithm for Follow()

for all nonterminals NT
    Follow(NT) = {}
Follow(S) = {$}
while Follow sets keep changing
    for all productions $A \rightarrow \alpha B \beta$
        $\text{Follow}(B) = \text{Follow}(B) \cup \text{First}(\beta)$
        if (β derives ε) $\text{Follow}(B) = \text{Follow}(B) \cup \text{Follow}(A)$
    for all productions $A \rightarrow \alpha B$
        $\text{Follow}(B) = \text{Follow}(B) \cup \text{Follow}(A)$

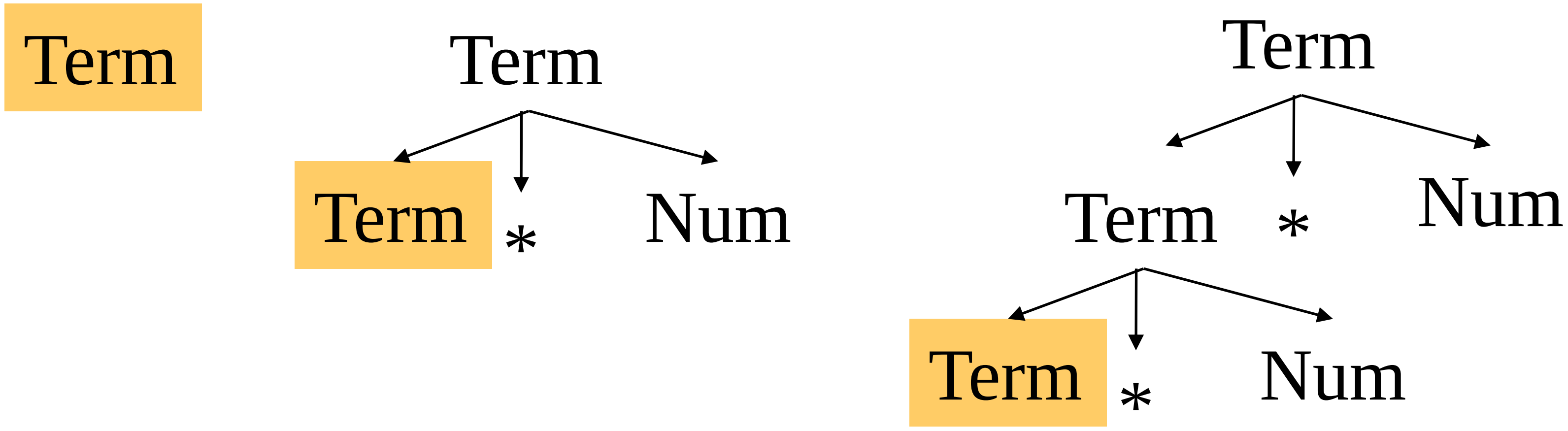
Example Follow()

- Grammar examples:
  - $S \rightarrow X\ \$$
    $X \rightarrow a$
    $X \rightarrow a\ b$
    - Follow(S) = { \$ }
    - Follow(X) = { \$ }
  - $S \rightarrow X\ \$$
    $X \rightarrow$ "(" X ")"
    $X \rightarrow \varepsilon$
    - Follow(S) = { \$ }
    - Follow(X) = { ")", \$ }

DESCENDENT (TOP-DOWN) SYNTACTIC ANALYSIS

Descendent Syntactic Analyzer

- Left recursion may lead to infinite loops!
- Example of production:
  - Term → Term*Num
- Potential steps of the analysis:

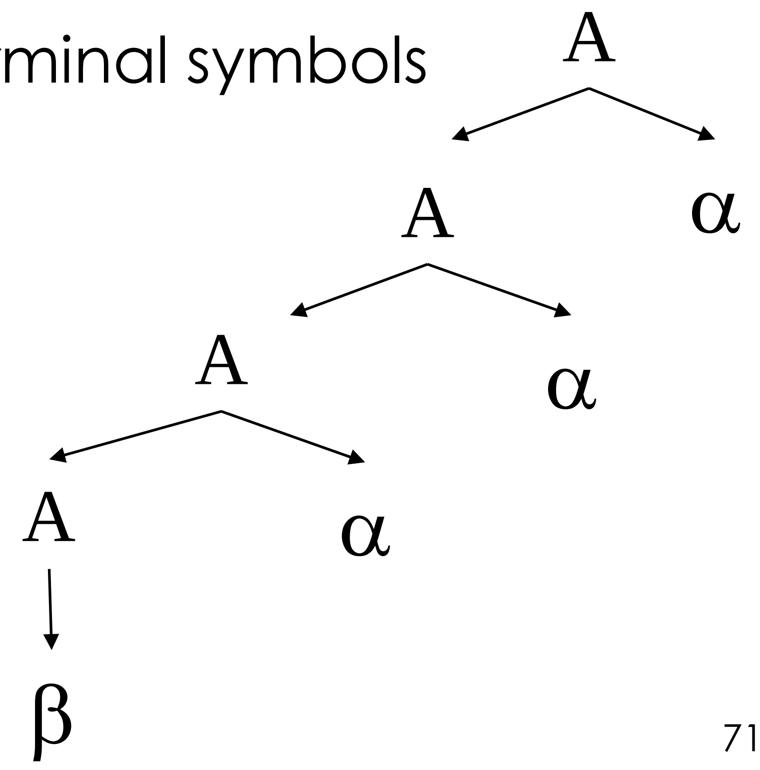


Descendent Syntactic Analyzer

- Left recursion may lead to infinite loops!
- Solution: modify grammar to eliminate left recursion

Eliminate Left Recursion

- Given the productions of the form:
  - $A \to A\ \alpha$
  - $A \to \beta$
  - Sequences α, β of terminal and non-terminal symbols which do not start with A
- Repetition of the derivation: $A \to A\ \alpha$ forms the syntax tree:



Eliminate Left Recursion

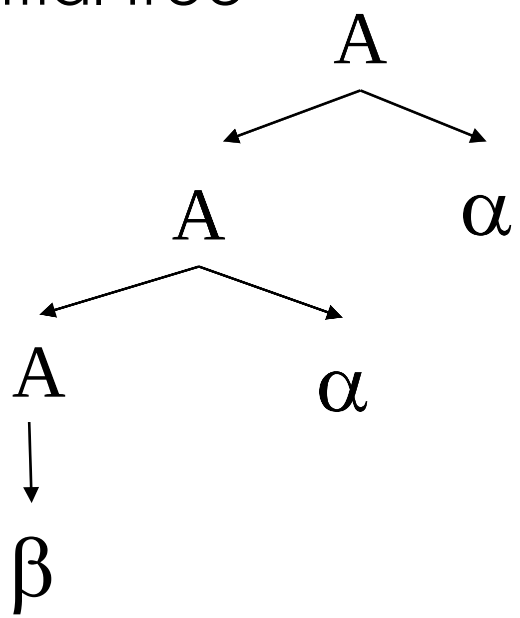
- Productions for substitutions
  - $A \rightarrow A\ \alpha$
  - $A \rightarrow \beta$

$A \rightarrow \beta\ R$ R is a new non-termnal symbol

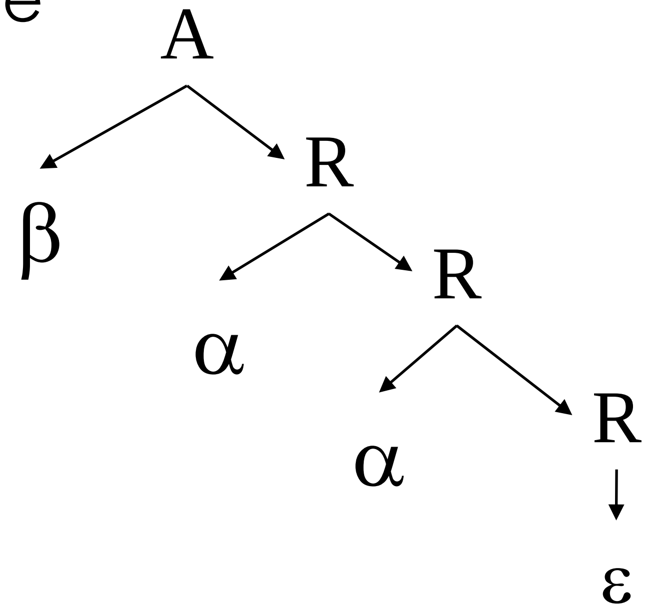
$R \rightarrow \alpha\ R$

$R \rightarrow \varepsilon$

Initial tree



New tree



Grammar Example

INT = [0-9]+

Start $\rightarrow$ Expr

Expr $\rightarrow$ Expr “+” Term

Expr $\rightarrow$ Expr “-” Term

Expr $\rightarrow$ Term

Term $\rightarrow$ Term “*” INT

Term $\rightarrow$ Term “/” INT

Term $\rightarrow$ INT

- Set of tokens (terminal symbols):

{ +, -, *, /, INT }

Modified Grammar

Part of the original gramar:

Term → Term “*” INT

Term → Term “/” INT

Term → INT

Part of the modified gramar:

Term → INT Term’

Term’ → “*” INT Term’

Term’ → “/” INT Term’

Term’ → ε

Modified Grammar

INT = [0-9]+

Start $\rightarrow$ Expr

Expr $\rightarrow$ Expr "+" Term

Expr $\rightarrow$ Expr "-" Term

Expr $\rightarrow$ Term

Term $\rightarrow$ Term "*" INT

Term $\rightarrow$ Term "/" INT

Term $\rightarrow$ INT

INT = [0-9]+

Start $\rightarrow$ Expr

Expr $\rightarrow$ Term Expr'

Expr' $\rightarrow$ "+" Term Expr'

Expr' $\rightarrow$ "-" Term Expr'

Expr' $\rightarrow$ ε

Term $\rightarrow$ INT Term'

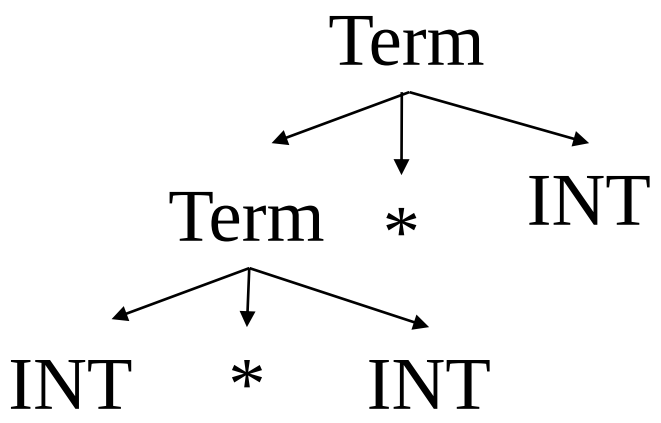
Term' $\rightarrow$ "*" INT Term'

Term' $\rightarrow$ "/" INT Term'

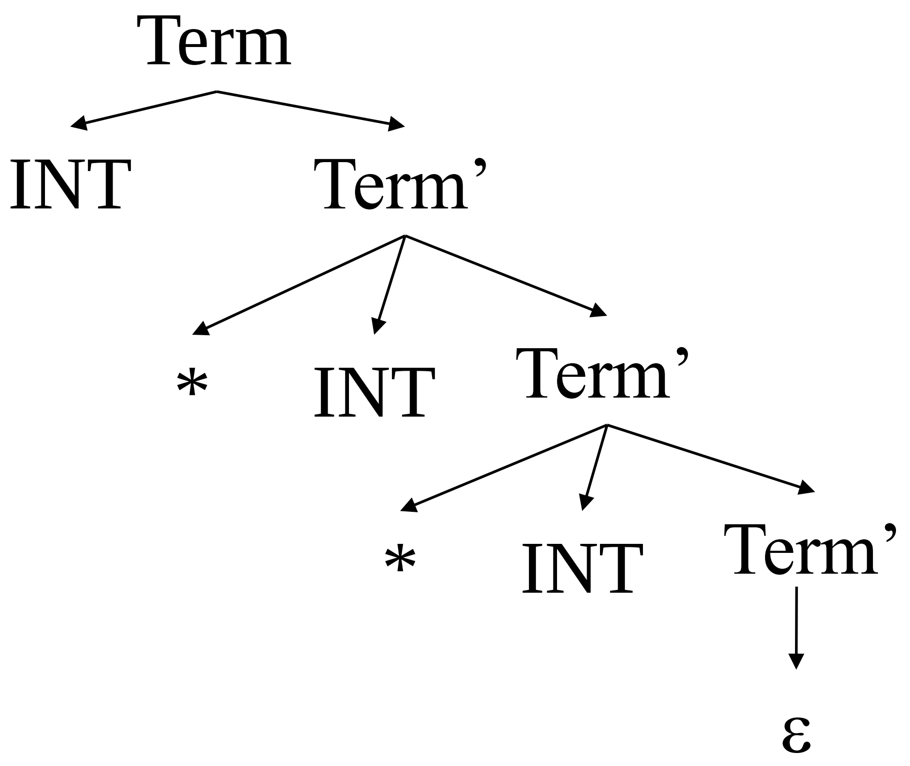
Term' $\rightarrow$ ε

Comparing the Syntax Trees

Original grammar



Modified grammar

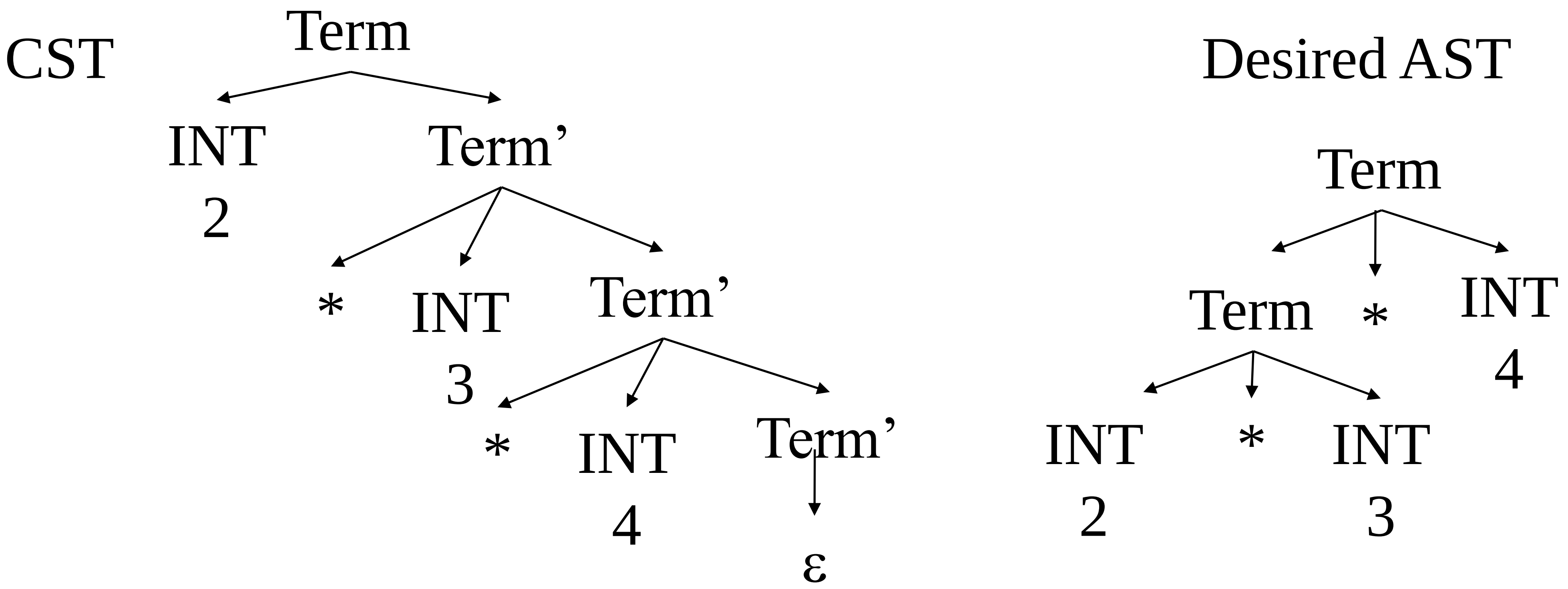


Eliminate Left Recursion

- Necessary in the predictive syntactic analysis
- Modify the search algorithm in the productions space
  - Eliminate direct infinite recursion
  - However: modified grammar is less intuitive (and so is the concrete syntax tree!)
  - Requires more transformations to achieve the desired AST

Requires more transformations to achieve the desired AST

- Syntax tree for: 2*3*4



- Solution: Build AST during derivation!

Summary

- Descendent Syntactic Analyzer (**top-down parser**)
- Use *Lookahead* to avoid *Backtracking* – known as ***predictive descendent syntactic analyzer***
- Modify grammar to avoid the necessity to inspect many subsequent tokens (*lookahead*): factorization
- Modify grammar to avoid infinite loops
- How to implement a descendent syntactic analyser?

Syntactic Analyser Manually Developed

- One procedure per non-terminal symbol
  - Analyses the current input symbol
  - Calls recursively procedures for the RHS of the selected production
- Simple case: the procedures return true if the syntactic analysis was well succeeded and false, otherwise
- Other cases:
  - In case of syntactic errors procedures can return more information (e.g., line number of token, column number of token, value of token, expected tokens…)
  - In case of of building a syntax tree, procedures can return associated node in the tree

Example

Term → INT Term'

Term' → "*" INT Term'

Term' → "/" INT Term'

Term' → ε

Productions for the symbol non-terminal Term:

Term → INT Term'

Function NextToken() steps one token position in the chain of tokens generated by the lexical analysis and returns the token in the new position

- Procedure for the non-terminal symbol Term:

```
Term() {
    if (token == INT) {
        token = NextToken();
        return TermPrime();
    } else return false;
}
```

Example (cont.)

Term → INT Term'

Term' → "*" INT Term'

Term' → "/" INT Term'

Term' → ε

- Procedure for the non-terminal symbol Term':

```
TermPrime() {
        if(token == '*') {
                token = NextToken();
                if (token == INT) {
                        token = NextToken();
                        return TermPrime();
                } else return false;
        elsif(token == '/') {
                token = NextToken();
                if (token == INT) {
                        token = NextToken();
                        return TermPrime();
                } else return false;

        } else return true;
}
```

```
TermPrime() {
        if((token == '*') || (token == '/')) {
                token = NextToken();
                if (token == INT) {
                        token = NextToken();
                        return TermPrime();
                } else return false;
        } else return true;
}
```

Example (cont.)

Term $\rightarrow$ INT Term'

Term' $\rightarrow$ "*" INT Term'

Term' $\rightarrow$ "/" INT Term'

Term' $\rightarrow$ ε

- Pseudo-code for the part of the program responsible to call the syntactic analyzer:

```
...
token = NextToken();
boolean result = Term();

if(result && token==null) write "Accept!" // null, EOF,...
else write "Reject!"
...
```

Construction of the Syntax Tree

- Each procedure returns the part of the tree for the part of the String (chain of tokens) analysed do far
- We can use exceptions to make clear the code structure (other option is to use an error function)
- Generally, we can use the syntactic analyser algorithm for different goals (besides the recognition or not of the input String)
  - Typically, it produces an AST instead of the CST

Construction of the Syntax Tree

- With generation of exceptions:

Term → INT Term'
Term' → "*" INT Term'
Term' → "/" INT Term'
Term' → ε

```
Term() {
    if (token == INT) {
            oldToken = token;
            token = NextToken();
            node = TermPrime();
            if (node == NULL) return oldToken;
            else return new TermNode(oldToken, node);
    } else throw SyntaxError;
}
```

Construction of the Syntax Tree

- With generation of exceptions:

```
TermPrime() {
    if ((token == '*') || (token == '/')) {
        first = token;
        next = NextToken();
        if (next == INT) {
            token = NextToken();
            return new TermPrimeNode(first, next, TermPrime());
        } else throw SyntaxError;
    } else return NULL;
}
```

Term $\rightarrow$ INT Term’

Term’ $\rightarrow$ “*” INT Term’

Term’ $\rightarrow$ “/” INT Term’

Term’ $\rightarrow$ ε

Construction of the Syntax Tree

- Without generation of exceptions

```
Term() {
    if (token == INT) {
        oldToken = token;
        token = NextToken();
        node = TermPrime();
        if (node == NULL) return oldToken;
        else return new TermNode(oldToken, node);
    } else error();
}
TermPrime() {
    if ((token == '*') || (token == '/')) {
        first = token; next = NextToken();
        if (next == INT) {
            token = NextToken();
            return new TermPrimeNode(first, next, TermPrime());
        } else error();
    } else return NULL;
}
```

Term $\rightarrow$ INT Term’

Term’ $\rightarrow$ “*” INT Term’

Term’ $\rightarrow$ “/” INT Term’

Term’ $\rightarrow$ ε

Syntax Tree for 2*3*4

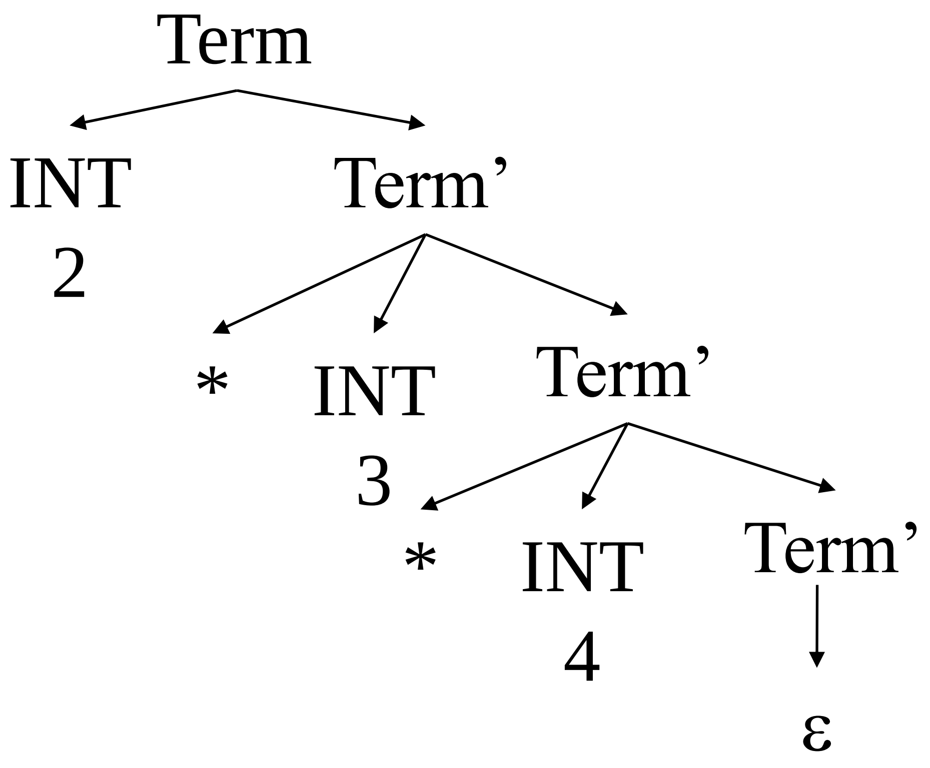
Term $\rightarrow$ INT Term'

Term' $\rightarrow$ "*" INT Term'

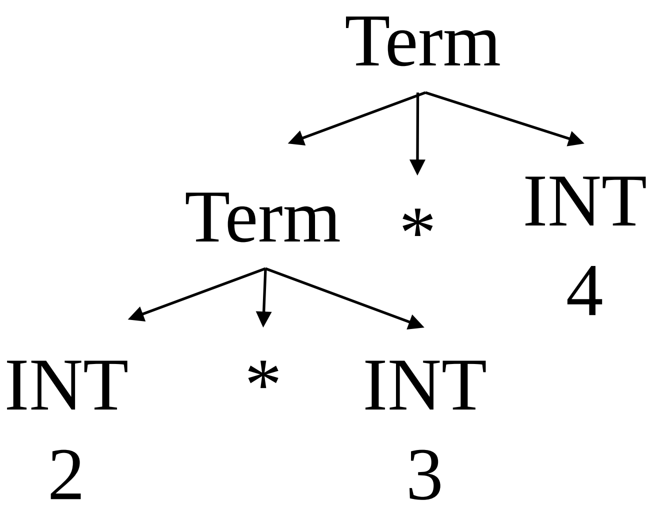
Term' $\rightarrow$ "/" INT Term'

Term' $\rightarrow$ ε

CST

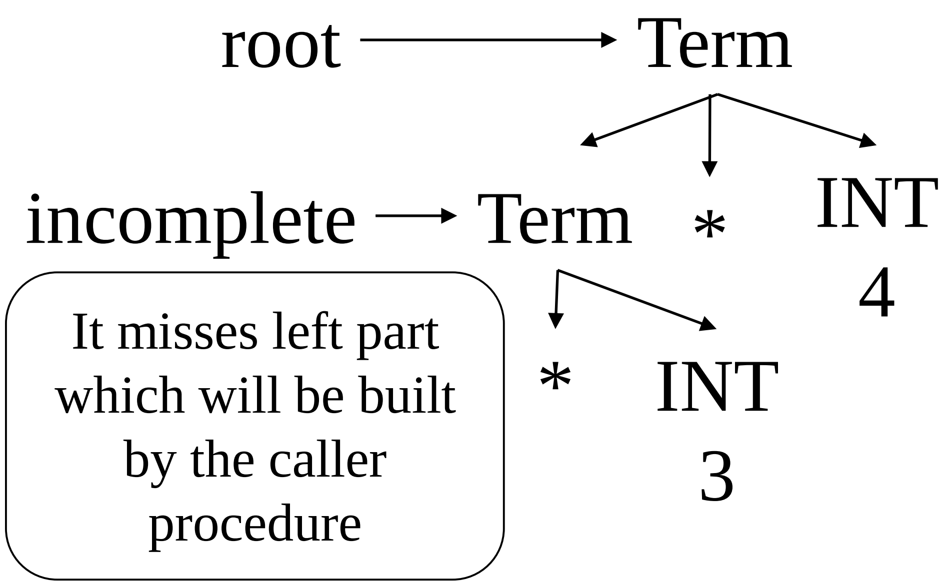


AST desired



Direct Generation of the AST

- TermPrime builds an incomplete tree
  - It lacks *leftmost child*
  - Returns the root and the incomplete node
- (root, incomplete) = TermPrime()
  - Call with token: *
  - Tokens missing: 3 * 4



Code for Term

Input

2*3*4

```
Term() {
    if (token == INT) {
        leftmostInt = token;
        token = NextToken();
        (root, incomplete) = TermPrime();
        if (root == NULL) return leftmostInt;
        incomplete.leftChild = leftmostInt;
        return root;
    } else throw SyntaxError;
}
```

token $\longrightarrow$ INT 2

Code for Term

Input

2*3*4

```
Term() {
    if (token == INT) {
        leftmostInt = token;
        token = NextToken();
        (root, incomplete) = TermPrime();
        if (root == NULL) return leftmostInt;
        incomplete.leftChild = leftmostInt;
        return root;
    } else throw SyntaxError;
}
```

token → INT
2

Code for Term

Input

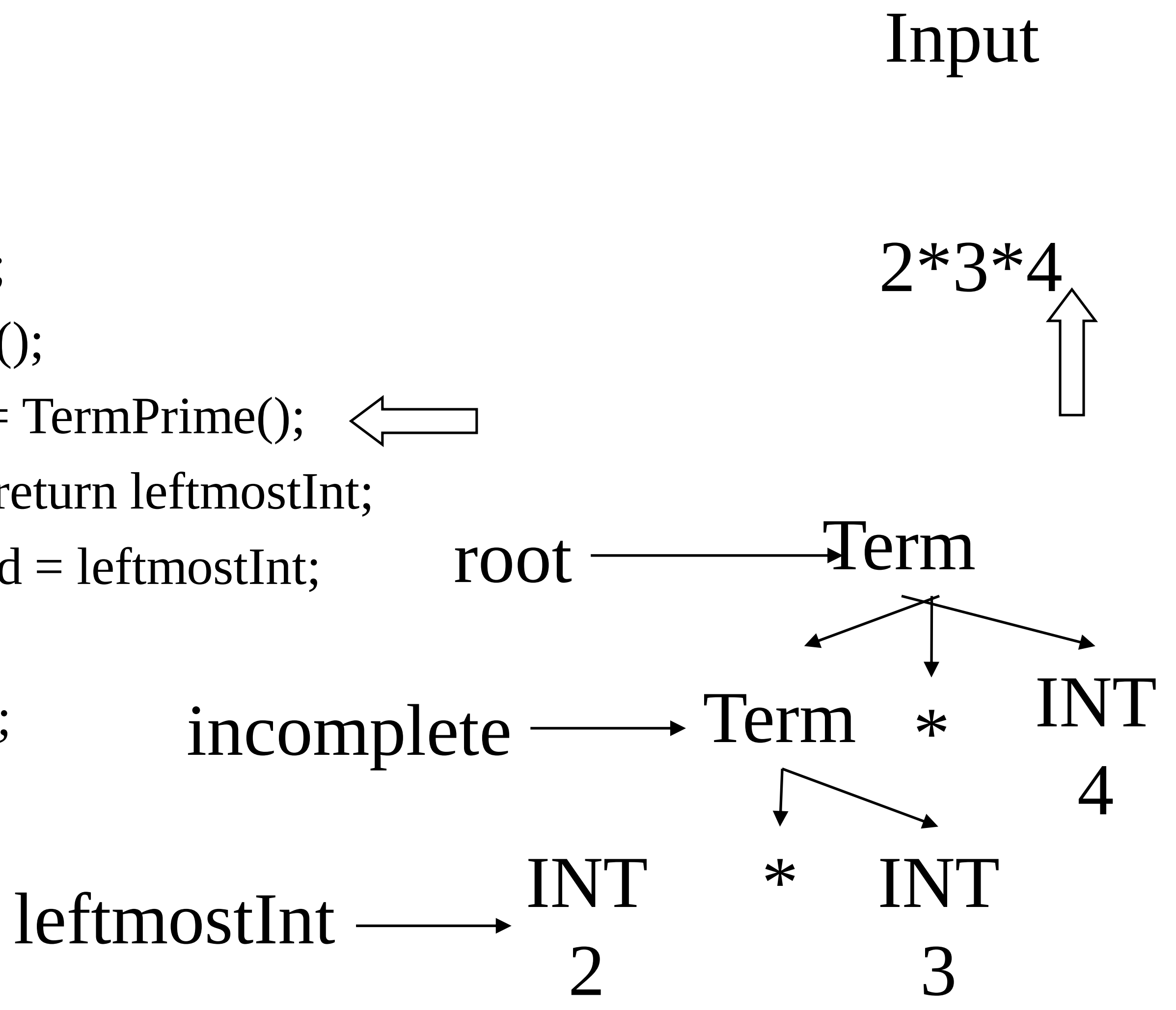
2*3*4

```
Term() {
    if (token == INT) {
        leftmostInt = token;
        token = NextToken();
        (root, incomplete) = TermPrime();
        if (root == NULL) return leftmostInt;
        incomplete.leftChild = leftmostInt;
        return root;
    } else throw SyntaxError;
}
```

leftmostInt $\longrightarrow$ INT 2

Code for Term

```
Term() {
    if (token == INT) {
        leftmostInt = token;
        token = NextToken();
        (root, incomplete) = TermPrime();
        if (root == NULL) return leftmostInt;
        incomplete.leftChild = leftmostInt;
        return root;
    } else throw SyntaxError;
}
```

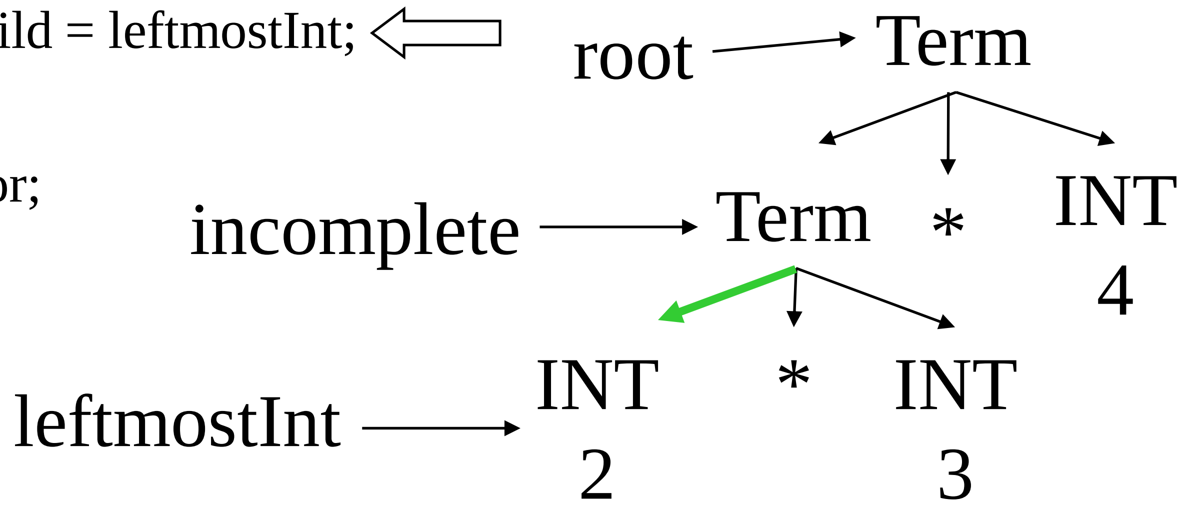


Code for Term

Input

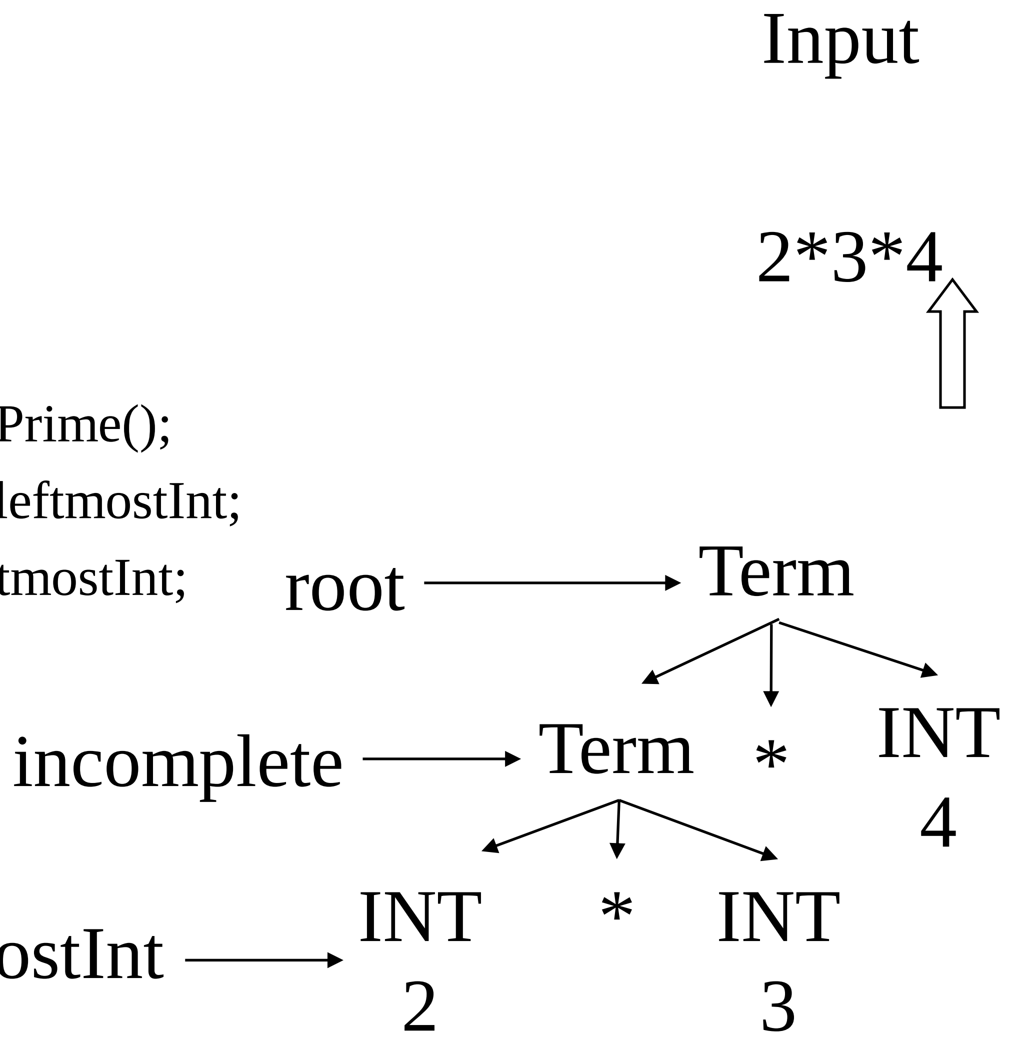
2*3*4

```
Term() {
    if (token == INT) {
        leftmostInt = token;
        token = NextToken();
        (root, incomplete) = TermPrime();
        if (root == NULL) return leftmostInt;
        incomplete.leftChild = leftmostInt;
        return root;
    } else throw SyntaxError;
}
```



Code for Term

```
Term() {
    if (token == INT) {
        leftmostInt = token;
        token = NextToken();
        (root, incomplete) = TermPrime();
        if (root == NULL) return leftmostInt;
        incomplete.leftChild = leftmostInt;
        return root;
    } else throw SyntaxError;
}
```



Code for TermPrime

```
TermPrime() {
    if((token == '*') || (token == '/')) {
        op = token;
        next = NextToken();
        if (next == INT) {
            token = NextToken();
            (root, incomplete) = TermPrime();
            if (root == NULL) {
                root = new ExprNode(NULL, op, next);
                return(root, root);
            } else {
                newChild = new ExprNode(NULL, op, next);
                incomplete.leftChild = newChild;
                return(root, newChild);
            }
        } else throw SyntaxError;
    } else return(NULL,NULL);
}
```

Left child to be placed by the caller procedure

Summary

- Descendent syntactic analyser (*Top-Down Parser*)
- Use *Lookahead* to avoid *Backtracking*
- The *parser* consists of a set of procedures mutually recursive

GRAMMAR TERMINOLOGY

Terminology

- Many techniques for syntactic analysis
  - Each one can handle a set of CFGs (context free grammars)
  - Categorization of the techniques
- Examples: LL(1), LL(2)
- LL(k)
  - Descendent (*top-down*), predictive
  - Construct derivation *leftmost* and from top to bottom

$\boxed{L}\,\boxed{L}\,(\boxed{k})$

L – Analysis from left to right.

L – Leftmost derivation

K – Lookahead value

Classify a Grammar as LL(1)

- How to verify if a grammar is LL(1)?
  - If the syntactic table does not have more than one production in each cell
- Syntactic table of the Predictive Analyzer
  - One row per non-terminal
  - One column per terminal
  - Put $X \rightarrow \gamma$ in row X, column T, for each $T \in First(\gamma)$
  - If γ can derive ε then put production $X \rightarrow \gamma$ in row X, column T, for each $T \in Follow(X)$

Classify a Grammar as LL(1)

Grammar:

$Z \rightarrow$ "d"

$Z \rightarrow X\ Y\ Z$

$Y \rightarrow \varepsilon$

$Y \rightarrow$ "c"

$X \rightarrow Y$

$X \rightarrow$ "a"

- Put production $X \rightarrow \gamma$ in row X, column T, for each $T \in First(\gamma)$
- If γ can derive ε then put production $X \rightarrow \gamma$ in row X, column T, for each $T \in Follow(X)$

Non-terminals	Terminals		
	"d"	"c"	"a"
Z			
Y			
X			

Classify a Grammar as LL(1)

Grammar:

$Z \rightarrow$ "d"

$Z \rightarrow X\ Y\ Z$

$Y \rightarrow \varepsilon$

$Y \rightarrow$ "c"

$X \rightarrow Y$

$X \rightarrow$ "a"

- Put production $X \rightarrow \gamma$ in row X, column T, for each $T \in First(\gamma)$ ①
- If γ can derive ε then put production $X \rightarrow \gamma$ in row X, column T, for each $T \in Follow(X)$ ②

Non-terminals	Terminals		
	"d"	"c"	"a"
Z	$Z \rightarrow X\ Y\ Z$ ① $Z \rightarrow$ "d"	$Z \rightarrow X\ Y\ Z$ ①	$Z \rightarrow X\ Y\ Z$ ①
Y	$Y \rightarrow \varepsilon$ ②	$Y \rightarrow \varepsilon$ ② $Y \rightarrow$ "c" ①	$Y \rightarrow \varepsilon$ ②
X	$X \rightarrow Y$ ①②	$X \rightarrow Y$ ①②	$X \rightarrow Y$ ② $X \rightarrow$ "a" ①

Classify a Grammar as LL(1)

Grammar:

Z → "d"

Z → X Y Z

Y → ε

Y → "c"

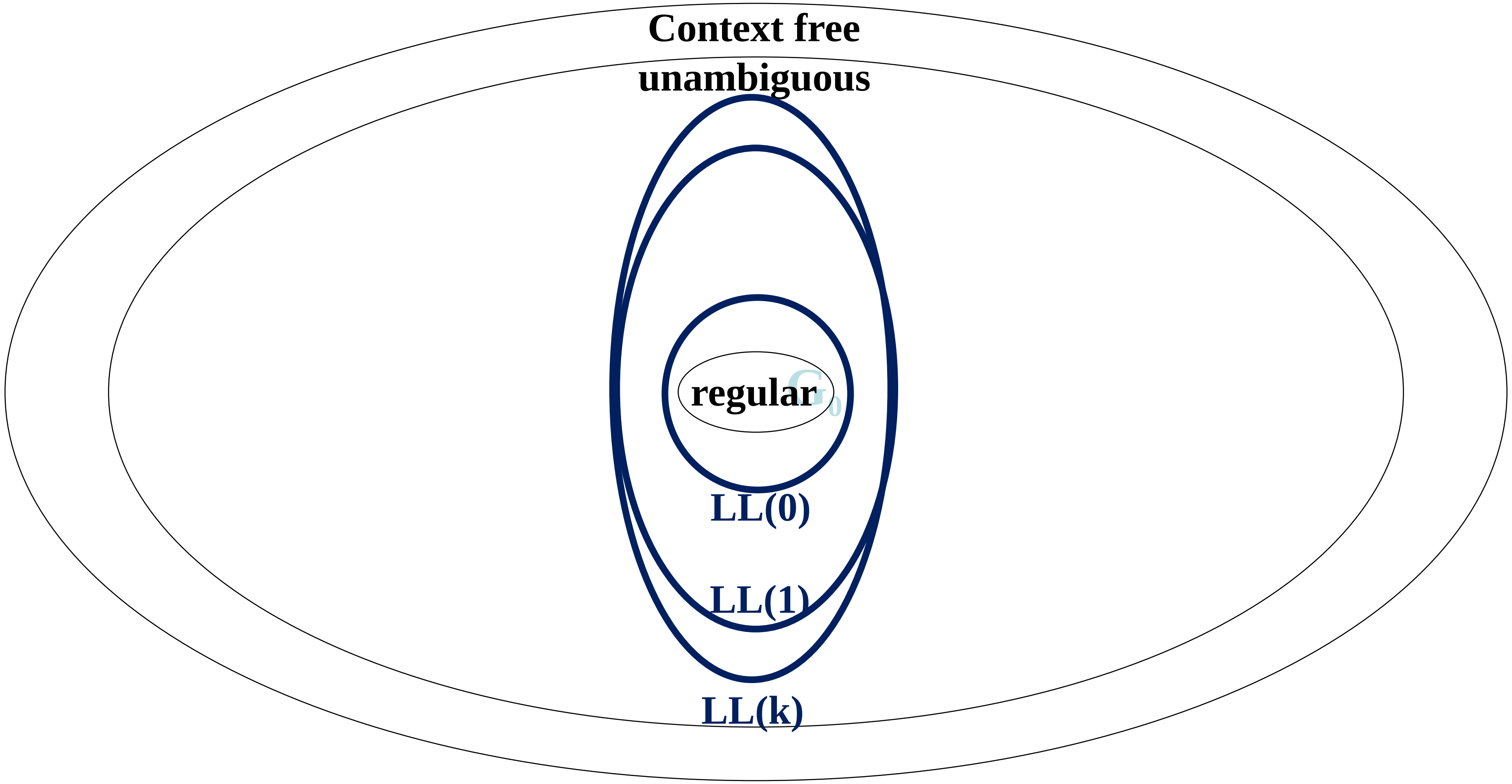
X → Y

X → "a"

- How to verify if a grammar is LL(1)?
  - If the syntactic table does not have more than one production per cell
- This gramar is not LL(1)

Non-terminals	Terminals		
	"d"	"c"	"a"
Z	Z → X Y Z Z → "d"	Z → X Y Z	Z → X Y Z
Y	Y → ε	Y → ε Y → "c"	Y → ε
X	X → Y	X → Y	X → Y X → "a"

Grammar Classification



Other Topics

- Attribute Grammars
- Generalized Context-Free Grammars

Lookahead Extensions

- Syntactic Lookahead
- Semantic Lookahead
- Both included in the JavaCC parser generator:
  - Syntactic:

```
LOOKAHEAD(“(“ Type1() “[“)
“(“ Type1() “[“ Other1()
| (“(“ Type2() “(“ Other2()
```

  - Semantic:

```
LOOKAHEAD( { getToken(1).kind == C && getToken(2).kind != C } )
<C:"c">
```

Lookahead Extensions

- LL(*)
  - Used in the ANTLR parser generator: http://www.antlr.org/
  - Paper: Terence Parr and Kathleen Fisher. 2011. **LL(*): the foundation of the ANTLR parser generator**. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, NY, USA, 425-436. DOI=http://dx.doi.org/10.1145/1993498.1993548

LL(*): The Foundation of the ANTLR Parser Generator

Terence Parr
University of San Francisco
parrt@cs.usfca.edu

Kathleen Fisher *
Tufts University
kfisher@eecs.tufts.edu

Abstract

Despite the power of Parser Expression Grammars (PEGs) and GLR, parsing is not a solved problem. Adding nondeterminism (parser speculation) to traditional *LL* and *LR* parsers can lead to unexpected parse-time behavior and introduces practical issues with error handling, single-step debugging, and side-effecting embedded grammar actions. This paper introduces the *LL(*)* parsing strategy and an associated grammar analysis algorithm that constructs *LL(*)* parsing decisions from ANTLR grammars. At parse-time, decisions gracefully throttle up from conventional fixed $k \geq 1$ lookahead to arbitrary lookahead and, finally, fail over to backtracking depending on the complexity of the parsing decision and the input symbols. *LL(*)* parsing strength reaches into the context-sensitive languages, in some cases beyond what GLR and PEGs can express. By statically removing as much speculation as possible, *LL(*)* provides the expressivity of PEGs while retaining *LL*'s good error handling and unrestricted grammar actions. Widespread use of ANTLR (over 70,000 downloads/year) shows that it is effective for a wide variety of applications.

trast, modern computers are so fast that programmer efficiency is now more important. In response to this development, researchers have developed more powerful, but more costly, nondeterministic parsing strategies following both the "bottom-up" approach (*LR*-style parsing) and the "top-down" approach (*LL*-style parsing).

In the "bottom-up" world, *Generalized LR* (GLR) [19] parsers parse in linear to cubic time, depending on how closely the grammar conforms to classic *LR*. GLR essentially "forks" new subparsers to pursue all possible actions emanating from nondeterministic *LR* states, terminating any subparsers that lead to invalid parses. The result is a parse forest with all possible interpretations of the input. *Elkhound* [12] is a very efficient GLR implementation that achieves *yacc*-like parsing speeds when grammars are *LALR*(1). Programmers unfamiliar with *LALR* parsing theory, though, can easily get nonlinear GLR parsers. Since GLR parser generators do not issue *LR* conflict warnings, programmers can unwittingly specify non-*LALR* grammars that lead to parsers with poor performance.

In the "top-down" world, Ford introduced *Packrat* parsers and the associated *Parser Expression Grammars* (PEGs) [6,

Parser Generators

- Generate C, http://dinosaur.compilertools.net/
  - Lex & Yacc
  - flex e bison
- Generate Java:
  - JLex e CUP
    - http://www.cs.princeton.edu/~appel/modern/java/JLex/
    - http://www.cs.princeton.edu/~appel/modern/java/CUP/
  - SableCC, http://sablecc.org/
  - JavaCC (version 6 includes C++ generation):
    http://www.experimentalstuff.com/Technologies/JavaCC/index.html
- ANTLR Parser Generator (generates Java, C#, JavaScript, Python):
  - http://www.antlr.org/
- List with other parser generators
  - http://catalog.compilertools.net/lexparse.html
  - http://catalog.compilertools.net/java.html