EDAN65: Compilers, Lecture 04

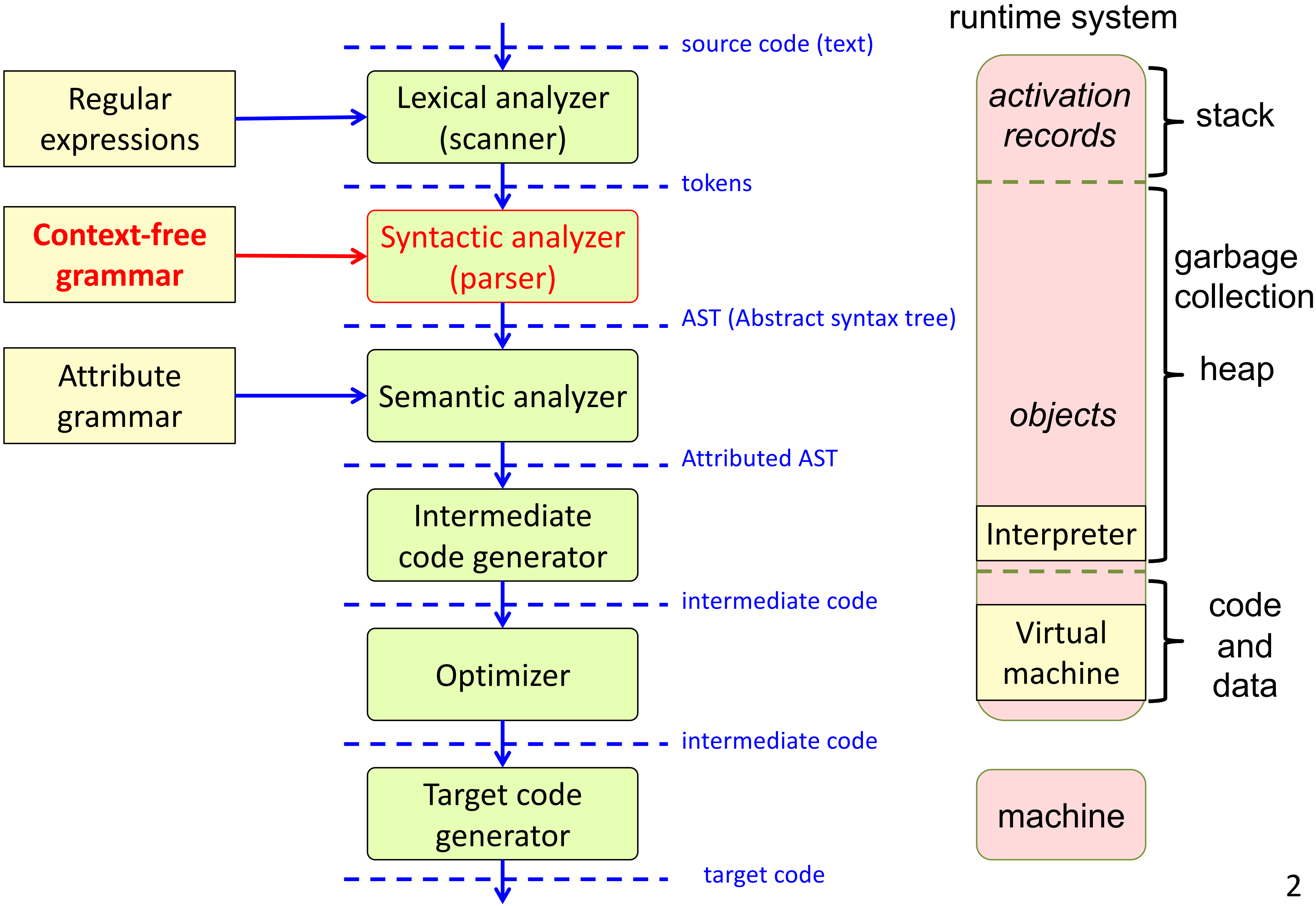
Grammar transformations:

Eliminating ambiguities, adapting to LL parsing

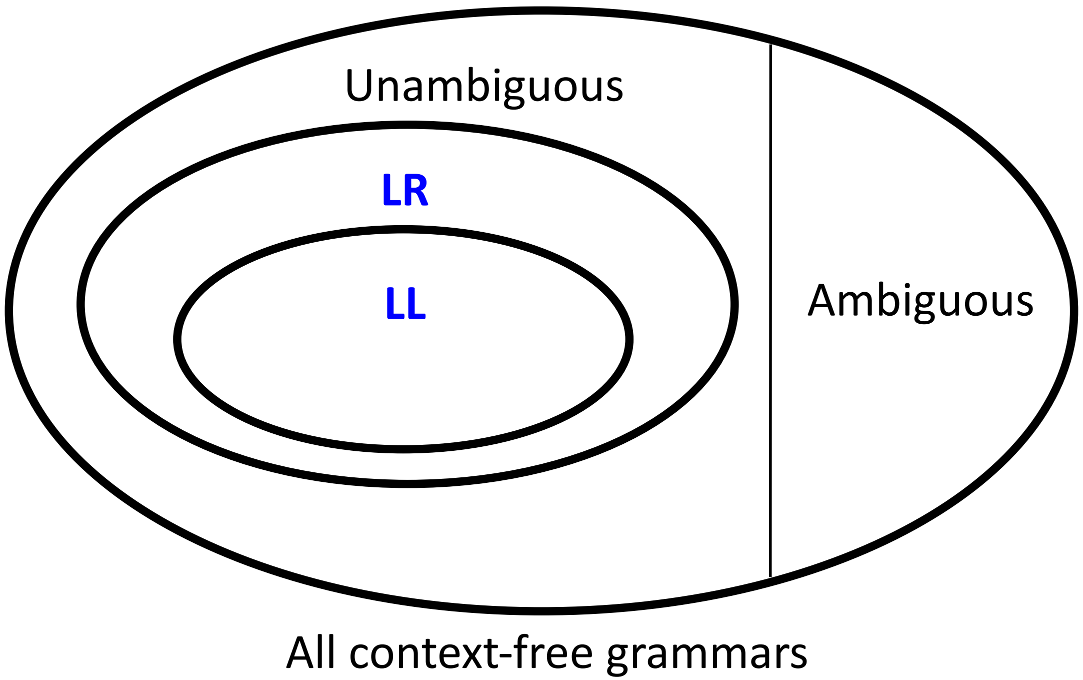
Görel Hedin

Revised: 2018-09-11

This lecture



Space of context-free grammars



LL:
Builds tree top-down
Simple to understand

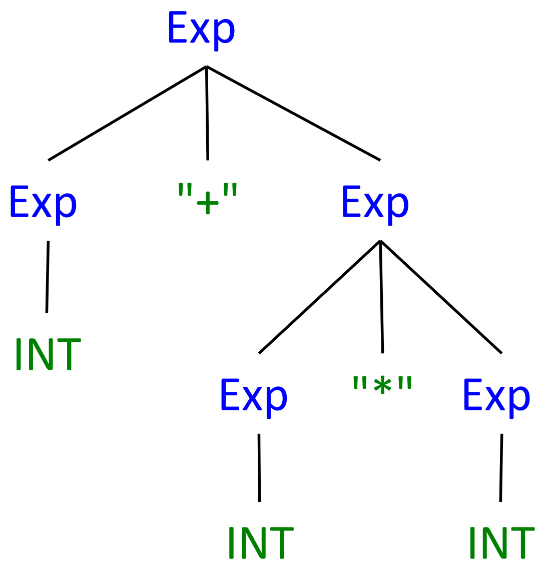
LR:
Builds tree bottom-up
More powerful

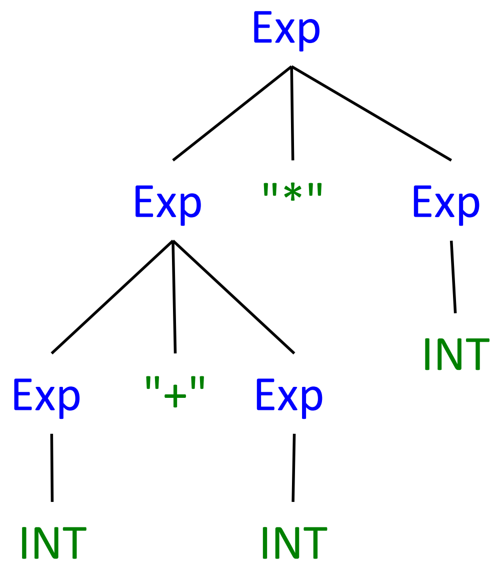
Ambiguous grammars

Recall: the definition of ambiguity

Grammar:
```
  Exp -> Exp "+" Exp
  Exp -> Exp "*" Exp
  Exp -> INT
```

A CFG is *ambiguous* if there is a sentence in the language that can be derived by two (or more) different parse trees.





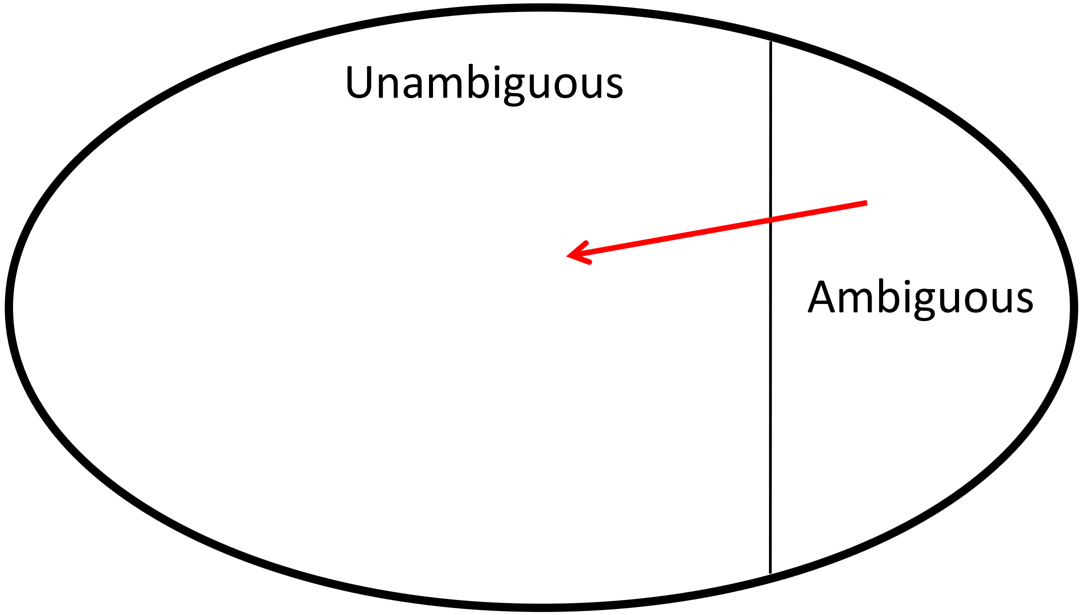
Strategies for dealing with ambiguities

First, decide which parse tree is the desired one.

Eliminate the ambiguity:
Create an equivalent unambiguous grammar.
Usually possible, but there exists grammars for which it cannot be done.
Unambiguous: Only one parse tree can be derived for any string.
However, the parse tree will be different from the original desired one.

Alternatively, use extra rules:
Use the ambiguous grammar.
Add priority and associativity rules to instruct the parser to select the desired parse tree.
Works for some ambiguities.
Supported by some parser generators.

Eliminating ambiguity



Goal: transform an ambiguous grammar to an *equivalent* unambiguous grammar.

Equivalent grammars

Two grammars, G_1 and G_2, are *equivalent* if they generate the same language.

I.e., a sentence can be derived from one of the grammars, iff it can be derived also from the other grammar:

$$L(G_1) = L(G_2)$$

Common kinds of ambiguities

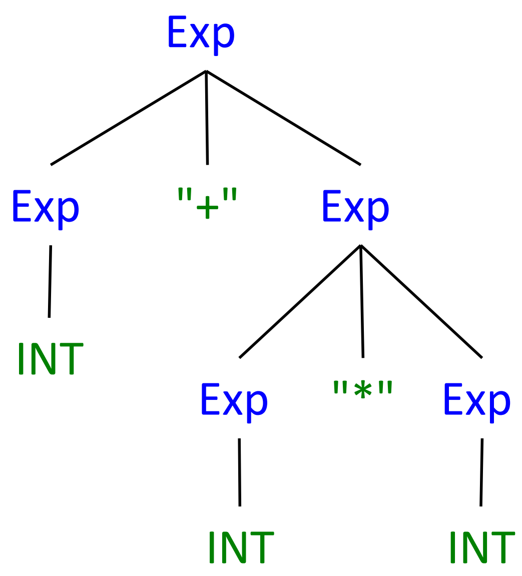
- Operators with different priorities:
```
a + b * c == d, ...
```
- Associativity of operators of the same priority:
```
a + b – c + d, ...
```
- Dangling else:
```
if (a)
if (b) c = d;
else e = f;
```

Example ambiguity:

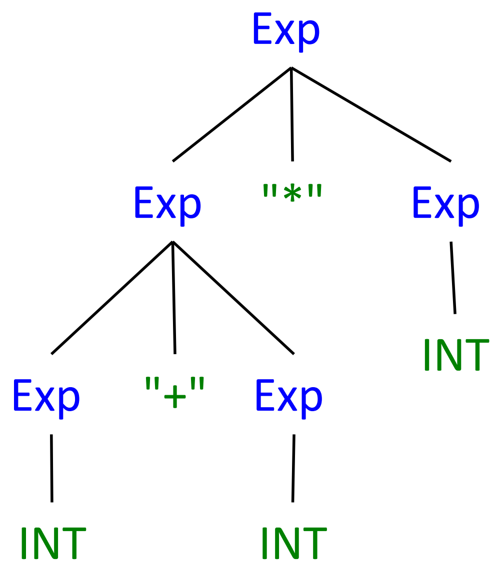
Priority (also called *precedence*)

```
Exp -> Exp "+" Exp
Exp -> Exp "*" Exp
Exp -> INT
```

Two parse trees for INT "+" INT "*" INT



prio("*") > prio("+")
(according to tradition)



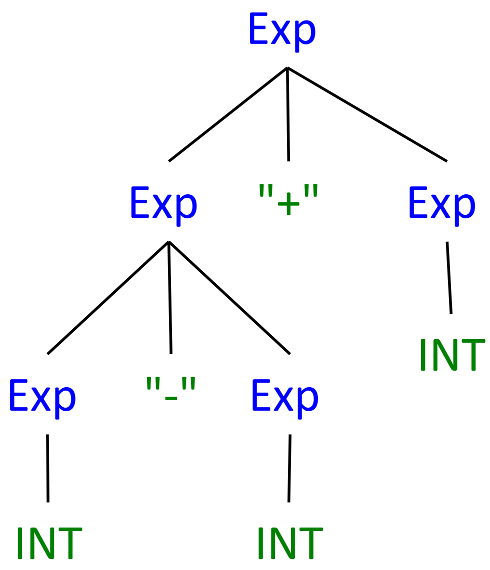
prio("+") > prio("*")
(would be unexpected and confusing)

Example ambiguity:

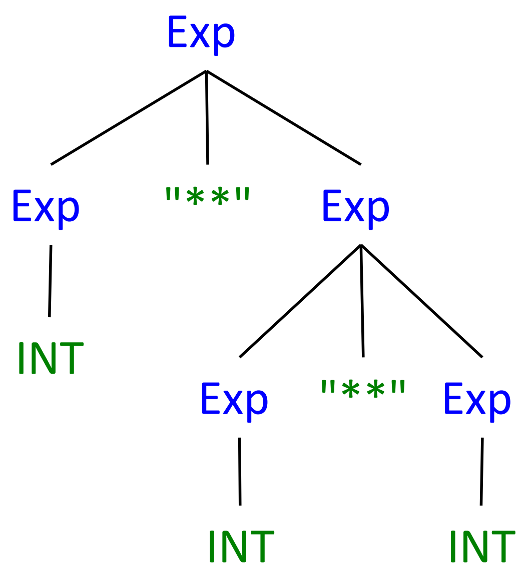
Associativity

```
Exp -> Exp "+" Exp
Exp -> Exp "-" Exp
Exp -> Exp "**" Exp
Exp -> INT
```

For operators with the same priority,
how do several in a sequence associate?



Left-associative
(usual for most operators)



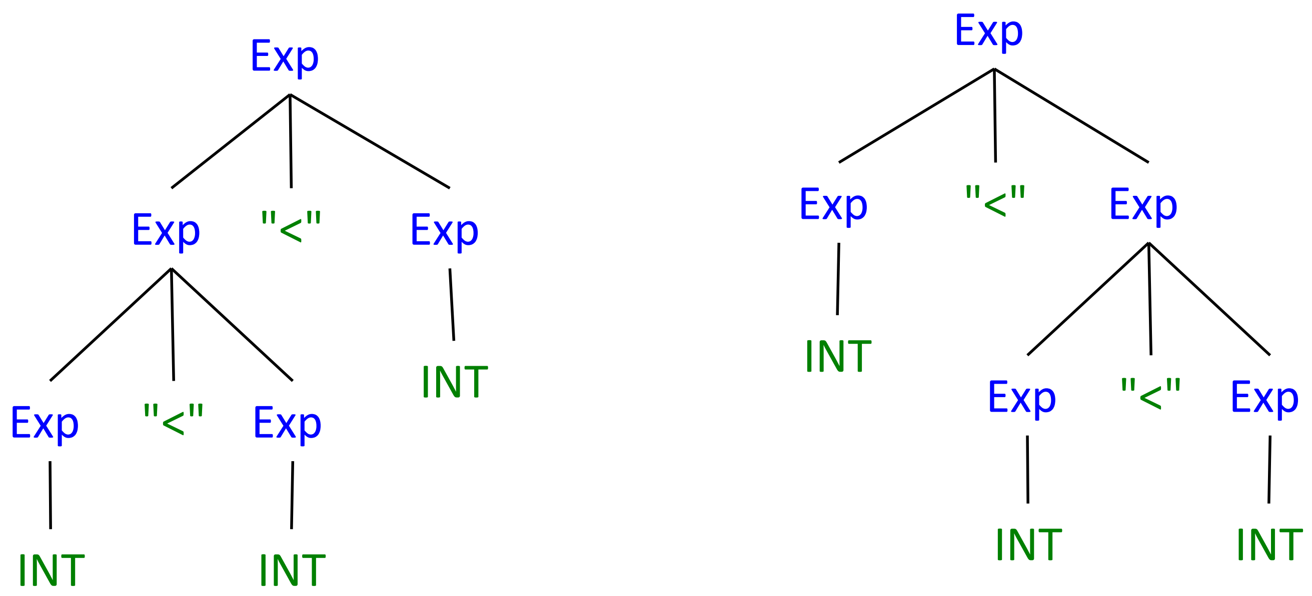
Right-associative
(usual for the power operator)

Example ambiguity:

Non-associativity

```
Exp -> Exp "<" Exp
Exp -> INT
```

For some operators, it does not make sense to have several in a sequence at all. They are *non-associative*.



We would like to forbid both trees.
I.e., rule out the sentence from the langauge.

Disambiguating expression grammars

How can we change the grammar so that only the desired trees can be derived?

Idea: Restrict certain subtrees by introducing new nonterminals.

Priority: Introduce a new nonterminal for each priority level:
Term, Factor, Primary, ...

Left associativity: Restrict the right operand so it only can contain expressions of higher priority

Right associativity: Restrict the left operand ...

Non-associativity: Restrict both operands

Exercise

Ambiguous grammar:

```
Expr -> Expr "+" Expr
Expr -> Expr "*" Expr
Expr -> ID
Expr -> "(" Expr ")"
```

Equivalent unambiguous grammar:

Solution

You will do this in Assignment 2!

Ambiguous grammar:

```
Expr -> Expr "+" Expr
Expr -> Expr "*" Expr
Expr -> ID
Expr -> "(" Expr ")"
```

Equivalent unambiguous grammar:

```
Expr -> Expr "+" Term
Expr -> Term
Term -> Term "*" Factor
Term -> Factor
Factor -> ID
Factor -> "(" Expr ")"
```

Here, we introduce a new nonterminal, Term, that is more restricted than Expr. That is, from Term, we can not derive any new additions.

For the addition production, we use Term as the right operand, to make sure no new additions will appear to the right. This gives left-associativity.

For the multiplication production, we use Term, and the even more restricted nonterminal Factor to make sure no additions can appear as children (without using parentheses). This gives multiplication higher priority than addition.

Real-world example: The Java expression grammar

```
Expression -> LambdaExpression | AssignmentExpression
AssignmentExpression -> ConditionalExpression | Assignment
ConditionalExpression -> ...

AdditiveExpression ->
    MultiplicativeExpression
  | AdditiveExpression + MultiplicativeExpression
  | AdditiveExpression – MultiplicativeExpression

MultiplicativeExpression ->
    UnaryExpression
  | MultiplicativeExpression * UnaryExpression
  | ...

UnaryExpression -> ...
...
Primary -> PrimaryNoNewArray | ArrayCreationExpression
PrimaryNoNewArray -> Literal | this | ( Expression ) | FieldAccess ...
```

More than 15 priority levels.

See the Java Language Specification, Java SE 8, Chapter 19, Syntax

http://docs.oracle.com/javase/specs/jls/se8/html/jls-19.html

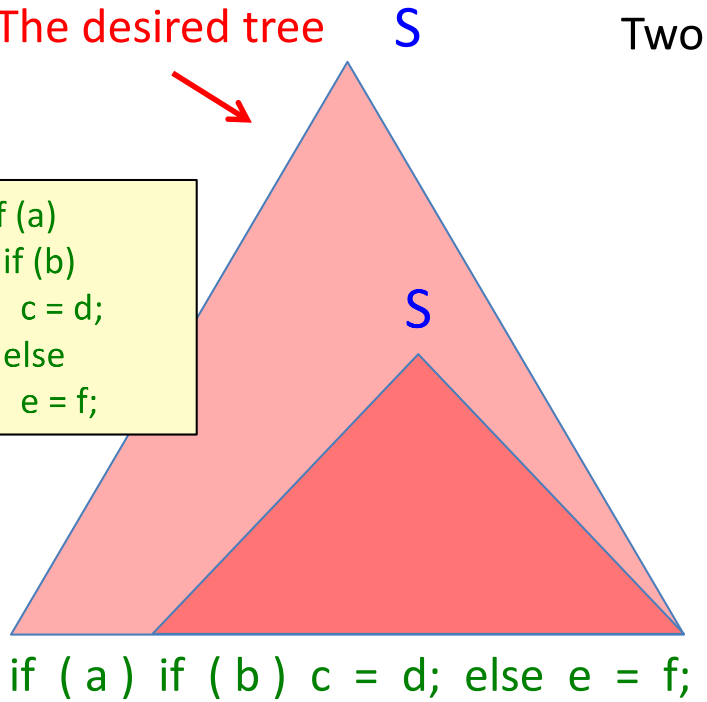
The "dangling else" problem

```
S -> "if" "(" E ")" S ["else" S]
S -> ID "=" E ";"
E -> ID
```

Construct a parse tree for:

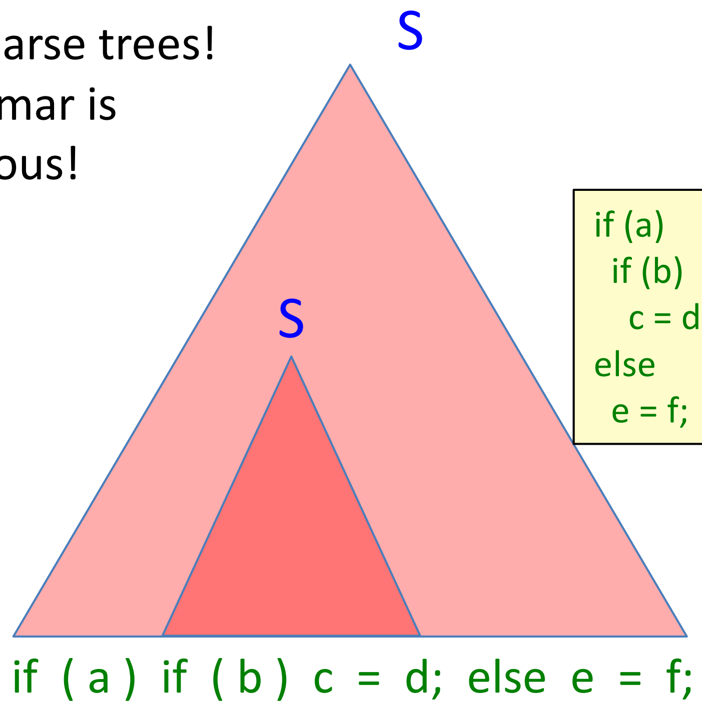
```
if (a)
if (b) c = d;
else e = f;
```

The desired tree




```
if (a)
  if (b)
    c = d;
  else
    e = f;
```

Two possible parse trees!
The grammar is ambiguous!




```
if (a)
  if (b)
    c = d;
else
  e = f;
```

Solutions to the "dangling else" problem

Rewrite to equivalent unambiguous grammar
- possible, but results in more complex grammar (several similar rules)

Use the ambiguous grammar
- use "rule priority", the parser can select the correct rule.
- works for the dangling else problem, but not for ambiguous grammars in general
- not all parser generators support it well

Change the language
- e.g., add a keyword "fi" that closes the "if"-statement
- restrict the "then" part to be a block: "{ ... }".
- only an option if you are designing the language yourself.

The Java Language Specification rewrites the grammar to be unambiguous.

Finding ambiguities in practice

You try to run a CFG through an LL or LR parser generator

- If it is accepted by the parser generator, the grammar is unambiguous
- If not, the grammar *could* be ambiguous, or unambiguous, but outside of the parser generator grammar class. In any case, you need to analyze that particular problem. This can be quite tricky, especially for large grammars. Perhaps you can find an ambiguity, or some other known LL/LR difficulty.

EBNF, BNF, Canonical form

Recall: different notations for CFGs

```
A -> B d e C f
A -> g A
```

Canonical form

- *sequence* of terminals and nonterminals

```
C -> D a b | b E F | a C
```

BNF (Backus-Naur Form)

- alternative productions (... | ... | ...)

```
G -> H* i | (d E)+ F | [d C]
```

EBNF (Extended Backus-Naur Form)

- *repetition* (* and +)
- *optionals* [...]
- *parentheses* (...)

Writing the grammar in different notations

Canonical form:

Expr -> Expr "+" Term
Expr -> Term
Term -> Term "*" Factor
Term -> Factor
Factor -> INT
Factor -> "(" Expr ")"

Equivalent BNF (Backus-Naur Form):

Use *alternatives* instead of several productions per nonterminal.

Equivalent EBNF (Extended BNF):

Use *repetition* instead of recursion, where possible.

Writing the grammar in different notations

Canonical form:

```
Expr -> Expr "+" Term
Expr -> Term
Term -> Term "*" Factor
Term -> Factor
Factor -> INT
Factor -> "(" Expr ")"
```

Equivalent BNF (Backus-Naur Form):

```
Expr -> Expr "+" Term | Term
Term -> Term "*" Factor | Factor
Factor -> INT | "(" Expr ")"
```

Use *alternatives* instead of several productions per nonterminal.

Equivalent EBNF (Extended BNF):

```
Expr -> Term ("+" Term)*
Term -> Factor ("*" Factor)*
Factor -> INT | "(" Expr ")"
```

Use *repetition* instead of recursion, where possible.

Translating EBNF to Canonical form

EBNF

Equivalent canonical form

Top level repetition
$X \rightarrow \gamma_1\ \gamma_2^*\ \gamma_3$

Top level alternative
$X \rightarrow \gamma_1 \mid \gamma_2$

Top level parentheses
$X \rightarrow \gamma_1\ (\ldots)\ \gamma_2$

Where γ_k is a sequence of terminals and nonterminals

Translating EBNF to Canonical form

EBNF	Equivalent canonical form
Top level repetition $X \to \gamma_1\ \gamma_2{*}\ \gamma_3$	$X \to \gamma_1\ N\ \gamma_3$ $N \to \varepsilon$ $N \to \gamma_2\ N$
Top level alternative $X \to \gamma_1 \mid \gamma_2$	$X \to \gamma_1$ $X \to \gamma_2$
Top level parentheses $X \to \gamma_1\ (\ldots)\ \gamma_2$	$X \to \gamma_1\ N\ \gamma_2$ $N \to \ldots$

Exercise:
Translate from EBNF to Canonical form

EBNF:

Expr -> Term ("+" Term)*

Equivalent Canonical Form

Solution:
Translate from EBNF to Canonical form

EBNF:

Expr -> Term ("+" Term)*

Equivalent Canonical Form

Expr -> Term N
N -> ε
N -> "+" Term N

Can we prove that these are equivalent?

trivial

Equivalent Canonical Form

Expr -> Term N
N -> ε
N -> "+" Term N

EBNF:

Expr -> Term ("+" Term)*

non-trivial

Alternative Equivalent Canonical Form

Expr -> Expr "+" Term
Expr -> Term

Example proof

1. We start with this:
Expr -> Term ("+" Term)*

We would like this:
Expr -> Expr "+" Term
Expr -> Term

2. We can move the repetition:
Expr -> (Term "+")* Term

3. Eliminate the repetition:
Expr -> N Term
N -> ε
N -> N Term "+"

4. Replace N Term by Expr in the third production:
Expr -> N Term
N -> ε
N -> Expr "+"

5. Eliminate N:
Expr -> Expr "+" Term
Expr -> Term

Done!

Equivalence of grammars

Given two context-free grammars, G1 and G2.
Are they equivalent?

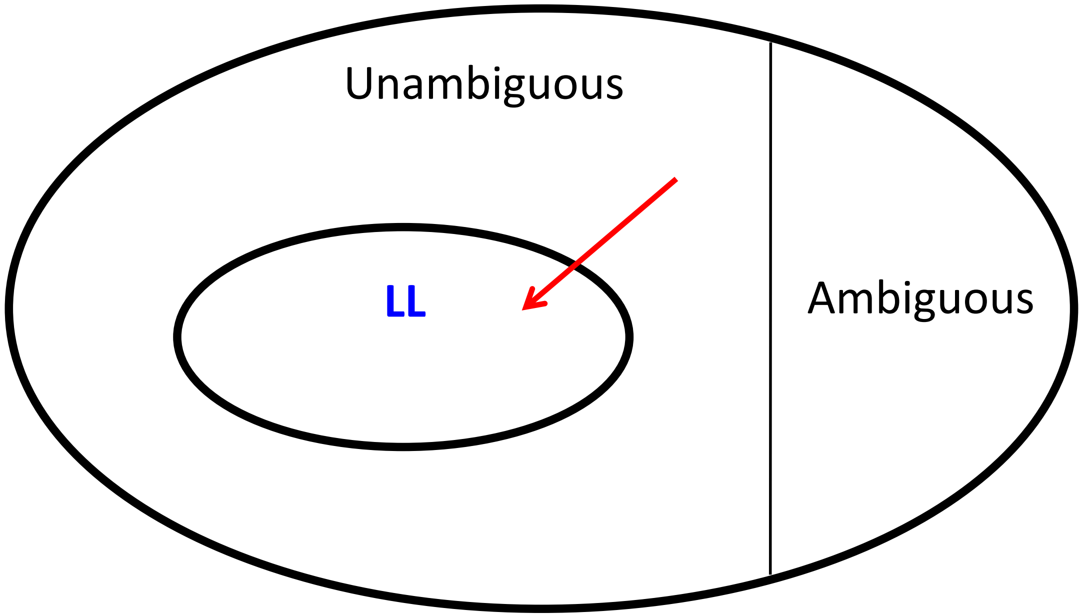
I.e., is L(G1) = L(G2)?

Undecidable problem:
a general algorithm cannot be constructed.

We need to rely on our ingenuity to find out.
(In the general case.)

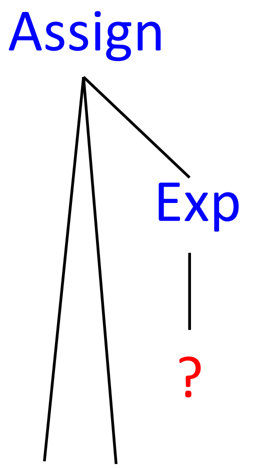
Adapting grammars to LL parsing

Create equivalent LL grammar



Typically, need to eliminate Left Recursion and Common Prefixes.
(But this may not be enough.)
The parse trees will be different from the original desired ones.
Try to build the desired ASTs anyway.
EBNF helps: relatively easy to build the desired AST.

Recall: LL(1) parsing



What node should be built?

ID = ID.ID; ID = ID.ID (ID);

↑

LL(1): decides to build the node after seeing the first token of its subtree.
The tree is built top down.

Assign -> ID = Exp ;
Exp -> Name Params | Name | ...
Name -> ID (. ID)*

Common prefix!
Cannot be handled by LL(1).
This grammar is not even LL(k).

Eliminating the common prefix

Rewrite to an equivalent grammar without the common prefix

Exp -> Name Params | Name

With common prefix - not LL(1)

Eliminating the common prefix

Rewrite to an equivalent grammar without the common prefix

```
Exp -> Name Params | Name
```

With common prefix - not LL(1)

```
Exp -> Name OptParams
OptParams -> Params | ε
```

Without common prefix - LL(1)

Eliminating a common prefix this way is called *left factoring*.

Exercise

If two productions of *the same* nonterminal can derive a sentence starting in the same way, they share a *common prefix*.

```
A -> s B
A -> s C
B -> t
C -> u
```

```
A -> B s
A -> B t
B -> u v
```

```
A -> s B
B -> s C
B -> t C
C -> u
```

Which nonterminals have common prefix productions?
What is the common prefix? Is the grammar LL(1), LL(2), ...?

Solution

If two productions of a nonterminal can derive a sentence starting in the same way, they share a *common prefix*.

A -> s B
A -> s C
B -> t
C -> u

A has two rules that can derive the prefix s
The grammar is LL(2)

A -> B s
A -> B t
B -> u v

A has two rules that can derive the prefix u v
The grammar is LL(3)

A -> s B
B -> s C
B -> t C
C -> u

This is not a common prefix problem. The two rules that start the same cannot be derived from the same nonterminal.
The grammar is LL(1)

Which nonterminals have common prefix productions?
What is the common prefix? Is the grammar LL(1), LL(2), ...?

The common prefix can be indirect

A -> B
A -> C
A -> D
B -> t s
C -> t v
D -> x

A has two rules that can derive the prefix t
The grammar is LL(2)

A -> B s
A -> B t
B -> B u
B -> v

A has two rules that can derive the prefix v u*
So, the prefix can become arbitrarily long.
The grammar is not LL(*k*), no matter what *k* we use.
We need to rewrite the grammar, or use another parsing method than LL. (For example, LR has no problem with common prefixes)

Which nonterminals have common prefix productions?
What is the common prefix?
Is the grammar LL(1), LL(2), ...?

Eliminating the common prefix

Rewrite to an equivalent grammar without the common prefix

A -> B
A -> C
B -> t s
B -> x D
B -> y
C -> t v
D -> B C

Indirect common prefix

Eliminating the common prefix

Rewrite to an equivalent grammar without the common prefix

A -> B
A -> C
B -> t s
B -> x D
B -> y
C -> t v
D -> B C

Indirect common prefix

First, make the common prefix directly visible:

Substitute all B right-hand sides into the A -> B rule

We can't remove the B rules since B is used in other places.

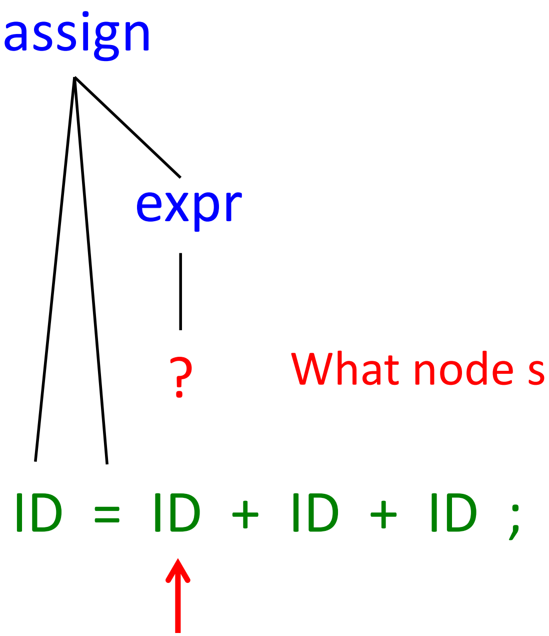
Similarly for the A -> C rule

A -> t s
A -> x D
A -> y
B -> t s
B -> x D
B -> y
A -> t v
C -> t v
D -> B C

Direct common prefix

Then, eliminate the direct common prefix, as previously.

Left recursion



What node should be built?

assign -> ID "=" expr ";"
expr -> expr "+" term | term
term -> ID

The grammar is *left recursive*.
The grammar is not LL(k).
An LL parser would go into endless recursion.

(LR parsers can handle left recursion.)

Dealing with left recursion in LL parsers

Method 1: Eliminate the left recursion

(A bit cumbersome)

Left-recursive grammar. Not LL(k)

```
E -> E "+" T
E -> T
T -> ID
```

Rewrite to right-recursion! But there is now a common prefix! Still not LL(k).

```
E -> T "+" E
E -> T
T -> ID
```

Eliminate the common prefix. The grammar is now LL(1)

```
E -> T E'
E' -> "+" E
E' -> ε
T -> ID
```

With a little work, it is possible to write code that builds a left-recursive AST, even if the parse is right-recursive.

Dealing with left recursion in LL parsers

Method 2: Rewrite to EBNF (Easy!)

Left-recursive grammar. Not LL(k)

```
E -> E "+" T
E -> T
T -> ID
```

Rewrite to EBNF!

```
E -> T ( "+" T )*
T -> ID
```

A left-recursive AST can easily be built during the iteration.

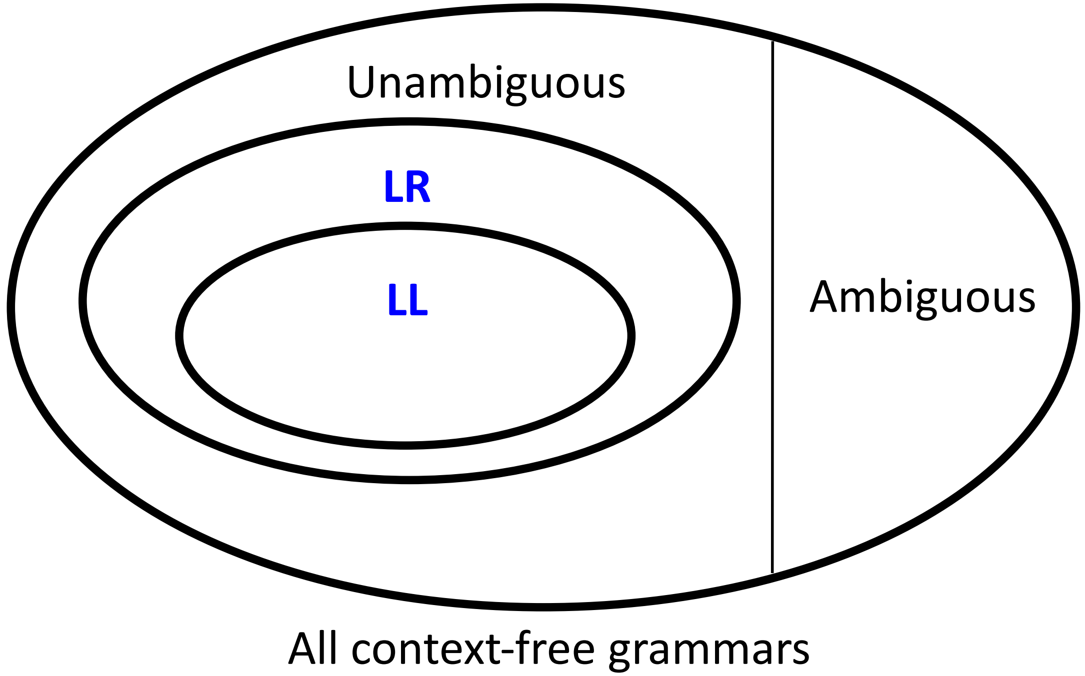
Advice when using an LL-based parser generator

If the LL parser generator does not accept your grammar, the reason might be

- Ambiguity – usually eliminate it. In some cases, rule priority can be used.
- Left recursion – can you use EBNF instead? Otherwise, eliminate.
- Common prefix – is it limited? You can then use a local lookahead, for example 2. Otherwise, factor out the common prefix.

You might be able to solve the problem, but the grammar might become large and less readable.

Different parsing algorithms



LL:

Left-to-right scan

Leftmost derivation

Builds tree top-down

Simple to understand

LR:

Left-to-right scan

Rightmost derivation

Builds tree bottom-up

More powerful

LL(*k*) vs LR(*k*)

	LL(*k*)	**LR(*k*)**
Parses input	Left-to-right	
Derivation	Leftmost	Rightmost
Lookahead	*k* symbols	
Build the tree	top down	bottom up
Select rule	after seeing its first *k* tokens	after seeing all its tokens, and an additional *k* tokens
Left recursion	Cannot handle	Can handle!
Unlimited common prefix	Cannot handle	Can handle!
Can resolve some ambiguities through rule priority	Dangling else	Dangling else, associativity, priority
Error recovery	Trial-and-error	Good algorithms exist
Implement by hand?	Possible. But better to use a generator.	Too complicated. Use a generator.

Summary questions

- What does it mean for a grammar to be ambiguous?
- What does it mean for two grammars to be equivalent?
- Exemplify some common kinds of ambiguities.
- Exemplify how expression grammars with can be disambiguated.
- What is the "dangling else"-problem, and how can it be solved?
- When should we use canonical form, and when BNF or EBNF?
- Translate an example EBNF grammar to canonical form.
- Can we write an algorithm to check if two grammars are equivalent?
- What is a ”common prefix”?
- Exemplify how a common prefix can be eliminated.
- What is ”left factoring”?
- What is ”left recursion”?
- Exemplify how left recursion can be eliminated in a grammar on canonical form.
- Exemplify how left recursion can be eliminated using EBNF.
- Can LL(*k*) parsing algorithms handle common prefixes and left recursion?
- Can LR(*k*) parsing algorithms handle common prefixes and left recursion?