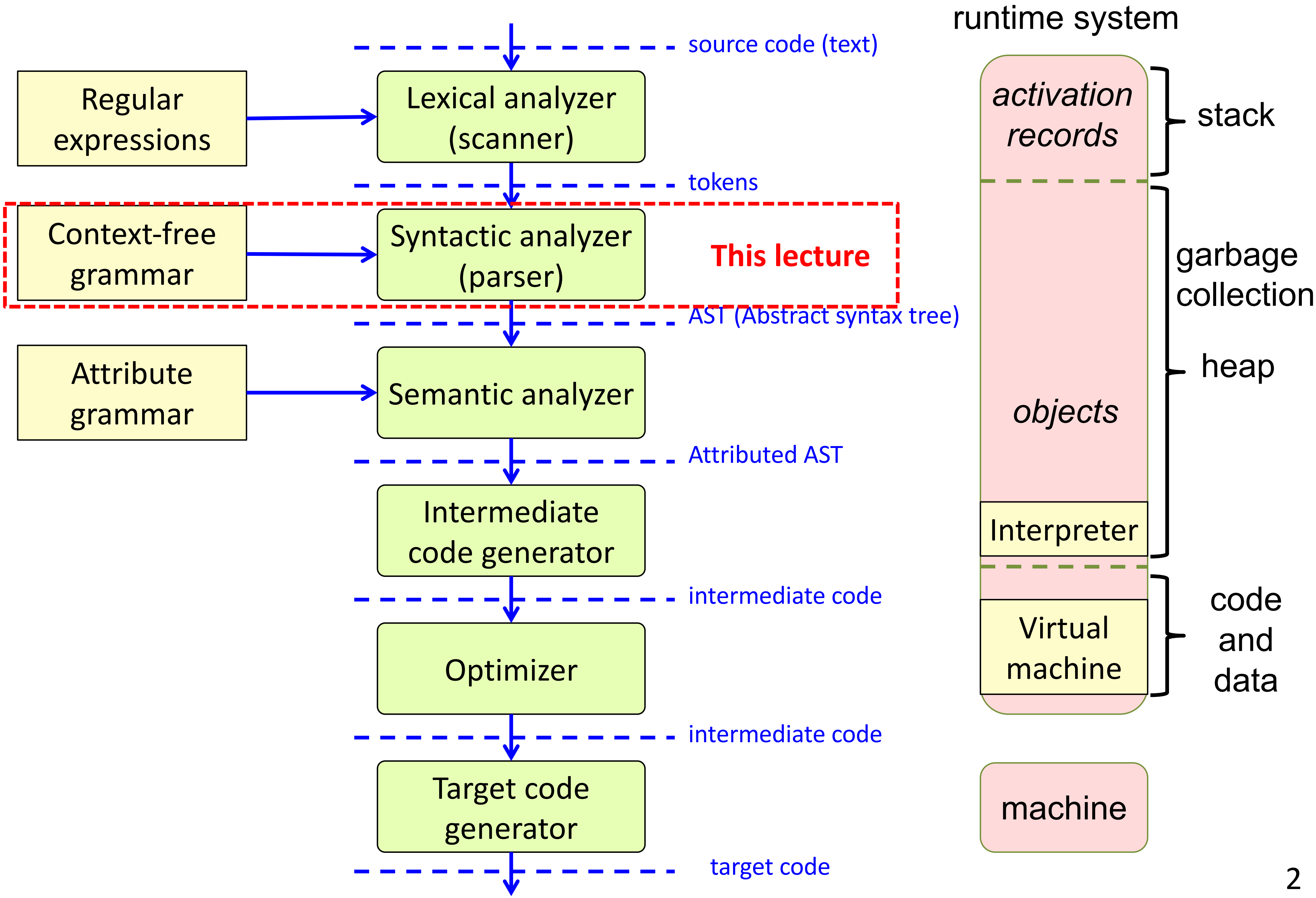
EDAN65: Compilers, Lecture 03

Context-free grammars, Introduction to parsing

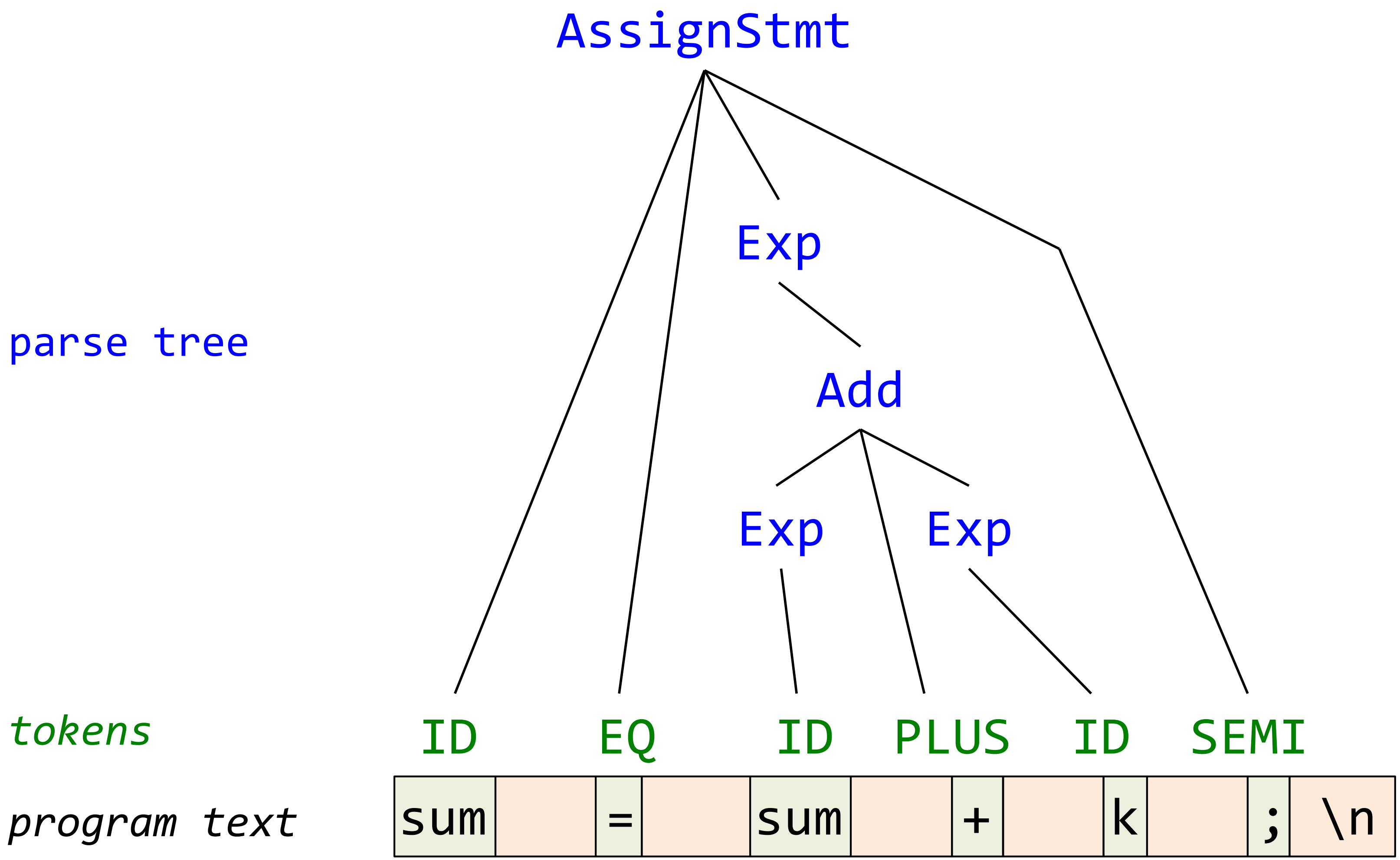
Görel Hedin

Revised: 2021-09-06

Course overview

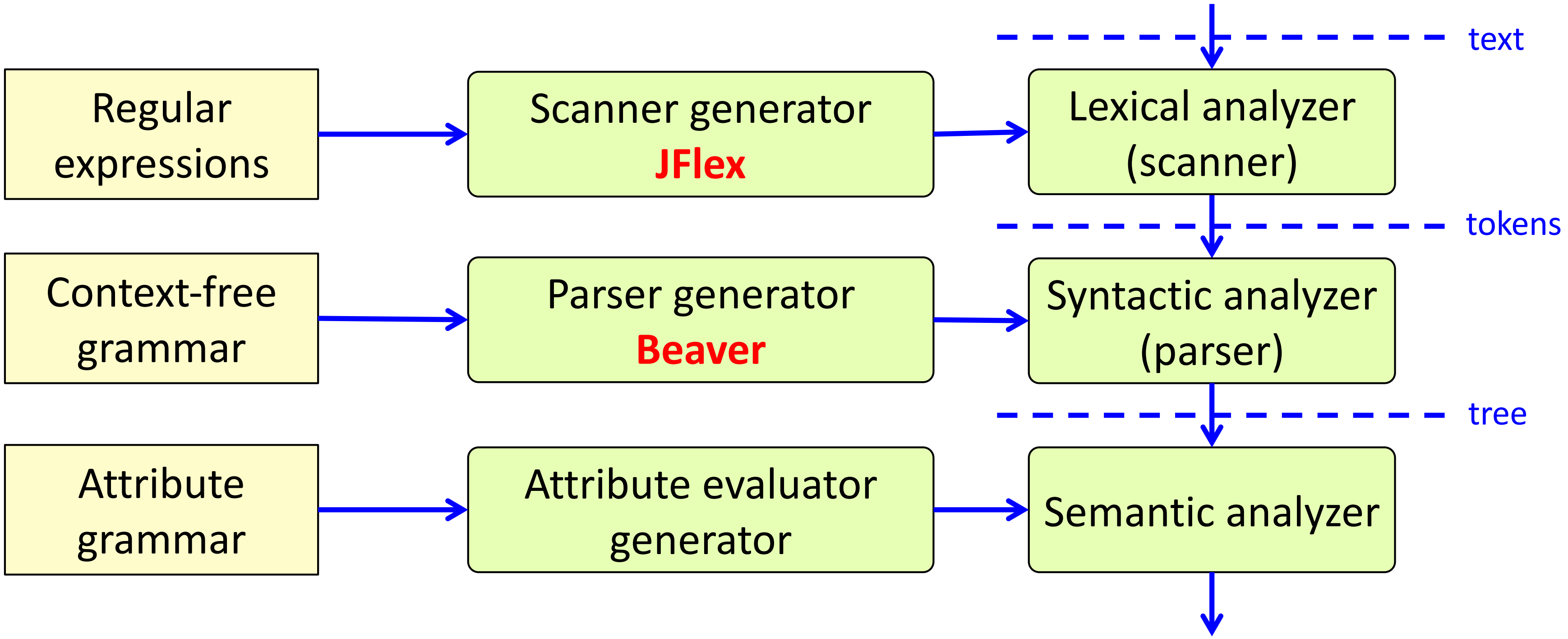


Analyzing program text



non-tokens (like white space) are discarded

Recall: Generating the compiler:



We will use a parser generator called **Beaver**

Context-Free Grammars

Regular Expressions vs Context-Free Grammars

Example REs:

```
WHILE = "while"
ID = [a-z][a-z0-9]*
LPAR = "("
RPAR = ")"
PLUS = "+"
...
```

Example CFG:

```
Stmt -> WhileStmt
Stmt -> AssignStmt
WhileStmt -> WHILE LPAR Exp RPAR Stmt
Exp -> ID
Exp -> Exp PLUS Exp
...
```

An RE can have *iteration*

A CFG can also have *recursion*

(it is possible to derive a symbol, e.g., `Stmt`, from itself)

Elements of a Context-Free Grammar

```
Example CFG:
Stmt -> WhileStmt
Stmt -> AssignStmt
WhileStmt -> WHILE LPAR Exp RPAR Stmt
AssignStmt -> ID EQ Exp SEMIC
…
```

Production rules:

$X \ \text{->}\ s_1\ s_2\ \ldots\ s_n$

where s_k is a *symbol* (terminal or nonterminal), n >= 0

Nonterminal symbols

Terminal symbols (tokens)

Start symbol

(one of the nonterminals, usually the left-hand side of the first production)

Shorthand for alternatives

```
Stmt -> WhileStmt | AssignStmt
```

is equivalent to

```
Stmt -> WhileStmt
Stmt -> AssignStmt
```

Shorthand for repetition

```
Stmt*
```

is equivalent to

```
StmtList
```

where

```
StmtList -> ε | Stmt StmtList
```

Exercise

Construct a grammar covering this program and similar ones:

Example program:

```
while (k <= n) {sum = sum + k; k = k+1;}
```

Solution

Construct a grammar covering this program and similar ones:

Example program:

```
while (k <= n) {sum = sum + k; k = k+1;}
```

CFG:

```
Stmt -> WhileStmt | AssignStmt | Block
WhileStmt -> "while" "(" Exp ")" Stmt
AssignStmt -> ID "=" Exp ";"
Block -> "{" Stmt* "}"
Exp -> LessEq | Add | ID | INT
LessEq -> Exp "<=" Exp
Add -> Exp "+" Exp
```

(Often, simple tokens are written directly as text strings)

Parsing

Use the grammar to derive a tree for a program (top-down):

```
Stmt -> WhileStmt | AssignStmt | Block
WhileStmt -> "while" "(" Exp ")" Stmt
AssignStmt -> ID "=" Exp ";"
Block -> "{" Stmt* "}"
Exp -> LessEq | Add | ID | INT
LessEq -> Exp "<=" Exp
Add -> Exp "+" Exp
```

Start symbol → `Stmt`

```
sum = sum + k ;
```

Parsing

Use the grammar to derive a tree for a program (bottom-up):

```
Stmt -> WhileStmt | AssignStmt | Block
WhileStmt -> "while" "(" Exp ")" Stmt
AssignStmt -> ID "=" Exp ";"
Block -> "{" Stmt* "}"
Exp -> LessEq | Add | ID | INT
LessEq -> Exp "<=" Exp
Add -> Exp "+" Exp
```

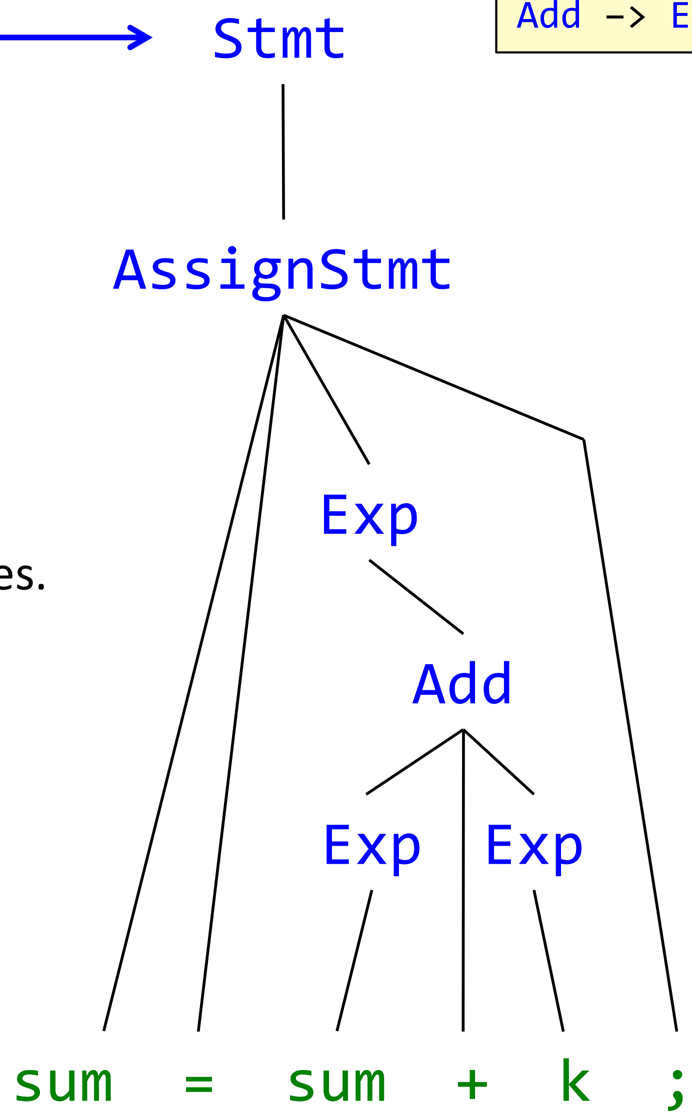
```
sum = sum + k ;
```

Parsing

Use the grammar to derive
a tree for a program:

```
Stmt -> WhileStmt | AssignStmt | Block
WhileStmt -> "while" "(" Exp ")" Stmt
AssignStmt -> ID "=" Exp ";"
Block -> "{" Stmt* "}"
Exp -> LessEq | Add | ID | INT
LessEq -> Exp "<=" Exp
Add -> Exp "+" Exp
```

Start symbol →

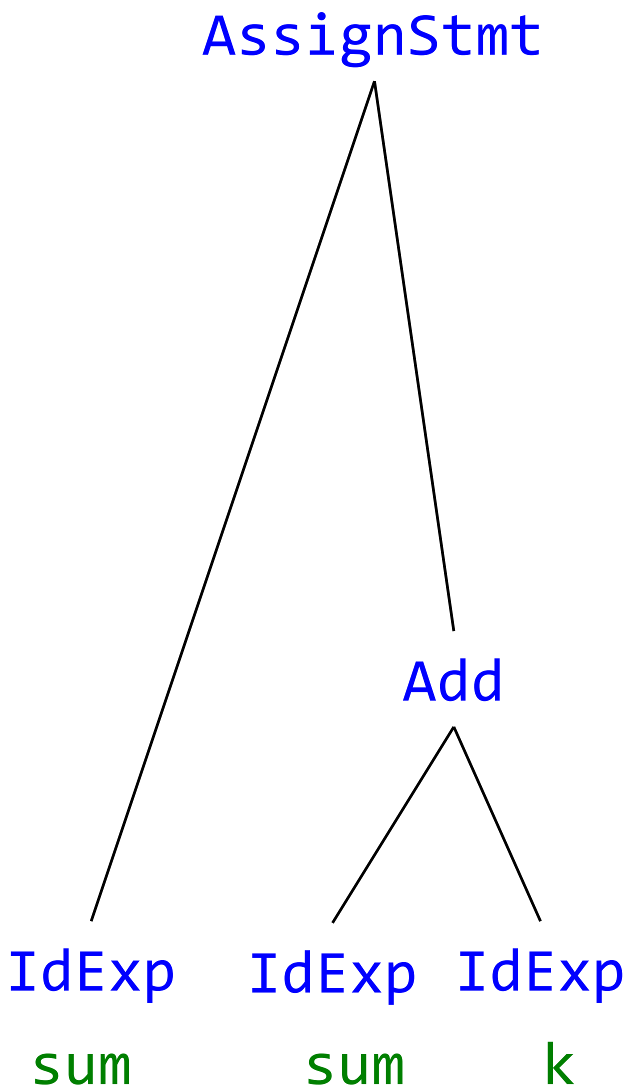


Nonterminals are inner nodes

A parse tree includes *all* the input tokens as leaves.

Terminals are leaves

Corresponding abstract syntax tree
(will be discussed in later lecture)



An abstract syntax tree is similar to a parse tree, but simpler.

It includes only some of the tokens.

(Tokens that could be generated from the tree are excluded.)

EBNF vs Canonical Form

EBNF:

```
Stmt -> AssignStmt | Block
AssignStmt -> ID "=" Exp ";"
Block -> "{" Stmt* "}"
Exp -> Add | ID
Add -> Exp "+" Exp
```

(Extended) Backus-Naur Form:

- Compact, easy to read and write
- BNF has alternatives
- EBNF has additionally repetition, optionals, parentheses (like REs)
- Common notation for practical use

Canonical form:

```
Stmt -> ID "=" Exp ";"
Stmt -> "{" Stmts "}"
Stmts -> ε
Stmts -> Stmt Stmts
Exp -> Exp "+" Exp
Exp -> ID
```

Canonical form:

- Core formalism for CFGs
- Useful for proving properties and explaining algorithms

Real world example:
The Java Language Specification

```
OrdinaryCompilationUnit:
  [PackageDeclaration] {ImportDeclaration} {TypeDeclaration}

PackageDeclaration:
  {PackageModifier} package Identifier {. Identifier} ;

PackageModifier:
  Annotation


…
```

See `https://docs.oracle.com/javase/specs/jls/se11/html`

- See Chapter 2 about the Java grammar notation.
- See Chapter 19 for the full syntax

Formal definition of CFGs

Formal definition of CFGs (canonical form)

A context-free grammar $G = (N, T, P, S)$, where

N – the set of nonterminal symbols

T – the set of terminal symbols

P – the set of production rules, each with the form

$$X \rightarrow Y_1\ Y_2\ \dots\ Y_n$$

where $X \in N$, $n \geq 0$, and $Y_k \in N \cup T$

S – the start symbol (one of the nonterminals). I.e., $S \in N$

Formal definition of CFGs (canonical form)

A context-free grammar $G = (N, T, P, S)$, where

N – the set of nonterminal symbols

T – the set of terminal symbols

P – the set of production rules, each with the form

$$X \rightarrow Y_1\ Y_2\ \ldots\ Y_n$$

where $X \in N$, $n \geq 0$, and $Y_k \in N \cup T$

S – the start symbol (one of the nonterminals). I.e., $S \in N$

So, the *left-hand side* X of a rule is a nonterminal.

And the *right-hand side* $Y_1\ Y_2\ \ldots\ Y_n$ is a sequence of nonterminals and terminals.

If the rhs for a production is empty, i.e., $n = 0$, we write

$$X \rightarrow \varepsilon$$

A grammar G defines a language $L(G)$

~~A~~ context-free grammar $G = (N, T, P, S)$, where

N – the set of nonterminal symbols

T – the set of terminal symbols

P – the set of production rules, each with the form

$$X \rightarrow Y_1\ Y_2\ \dots\ Y_n$$

where $X \in N$, $n \geq 0$, and $Y_k \in N \cup T$

S – the start symbol (one of the nonterminals). I.e., $S \in N$

A grammar G defines a language L(G)

~~A~~ context-free grammar $G = (N, T, P, S)$, where

N – the set of nonterminal symbols

T – the set of terminal symbols

P – the set of production rules, each with the form

$$X \text{ –> } Y_1\ Y_2 \ldots Y_n$$

where $X \in N$, $n \geq 0$, and $Y_k \in N \cup T$

S – the start symbol (one of the nonterminals). I.e., $S \in N$

G defines a *language* $L(G)$ over the alphabet T

T^* is the set of all possible sequences of T symbols.

$L(G)$ is the subset of T^* that can be derived from the start symbol S, by following the production rules P.

Exercise

```
G = (N, T, P, S)

P = {
  Stmt -> ID "=" Exp ";",
  Stmt -> "{" Stmts "}" ,
  Stmts -> ε ,
  Stmts -> Stmt Stmts ,
  Exp -> Exp "+" Exp ,
  Exp -> ID
}

N =

T =

S =
```

```
L(G) =
```

Solution

```
G = (N, T, P, S)

P = {
  Stmt -> ID "=" Exp ";",
  Stmt -> "{" Stmts "}" ,
  Stmts -> ε ,
  Stmts -> Stmt Stmts ,
  Exp -> Exp "+" Exp ,
  Exp -> ID
}

N = {Stmt, Exp, Stmts}

T = {ID, "=", "{", "}", ";", "+"}

S = Stmt
```

```
L(G) = {
 "{" "}",
 "{" "{" "}" "}",
 ID "=" ID ";",
 "{" ID "=" ID ";" "}",
 ID "=" ID "+" ID ";",
 "{" "{" "}" "{" "}" "}",
 "{" "{" "{" "}" "}" "}",
 "{" ID "=" ID "+" ID ";" "}",
 ID "=" ID "+" ID "+" ID ";",
 ...
}
```

The sequences in L(G) are usually called *sentences* or *strings*

Derivations

Derivation step

If we have a sequence of terminals and nonterminals, e.g.,

X a Y Y b

we can replace one of the nonterminals, applying a production rule. This is called a *derivation step*.
(Swedish: *Härledningssteg*)

Derivation step

If we have a sequence of terminals and nonterminals, e.g.,

X a Y Y b

we can replace one of the nonterminals, applying a production rule. This is called a *derivation step*.
(Swedish: *Härledningssteg*)

Suppose there is a production

Y –> X a

and we apply it for the first Y in the sequence. We write the derivation step as follows:

X a Y Y b => X a X a Y b

Derivation

A *derivation*, is simply a sequence of derivation steps, e.g.:

$\gamma_0 => \gamma_1 => \ldots => \gamma_n \qquad (n \geq 0)$

where each γ_i is a sequence of terminals and nonterminals

If there is a derivation from γ_0 to γ_n, we can write this as

$\gamma_0 =>^* \gamma_n$

So this means it is possible to get from the sequence γ_0 to the sequence γ_n by applying 0 or more production rules.

Definition of the language L(G)

Recall that:

G = (N, T, P, S)

T* is the set of all possible sequences of T symbols.

L(G) is the subset of T* that can be derived from the start symbol S, by applying production rules in P.

Definition of the language L(G)

Recall that:

$G = (N, T, P, S)$

T^* is the set of all possible sequences of T symbols.

$L(G)$ is the subset of T^* that can be derived from the start symbol S, by applying production rules in P.

Using the concept of derivations, we can formally define $L(G)$ as follows:

$$L(G) = \{ w \in T^* \mid S \Rightarrow^* w \}$$

Exercise:
Prove that a sentence belongs to a language

Prove that

INT + INT * INT

belongs to the language of the following grammar:

p_1: Exp –> Exp "+" Exp
p_2: Exp –> Exp "*" Exp
p_3: Exp –> INT

Proof:

Solution:

Prove that a sentence belongs to a language

Prove that

INT + INT * INT

belongs to the language of the following grammar:

p_1: Exp –> Exp "+" Exp
p_2: Exp –> Exp "*" Exp
p_3: Exp –> INT

Proof:
(by showing all the derivation steps from the start symbol Exp)

Exp
=>^{p1} Exp "+" Exp
=>^{p3} INT "+" Exp
=>^{p2} INT "+" Exp "*" Exp
=>^{p3} INT "+" INT "*" Exp
=>^{p3} INT "+" INT "*" INT

Leftmost and rightmost derivations

p_1: Exp –> Exp "+" Exp
p_2: Exp –> Exp "*" Exp
p_3: Exp –> INT

In a *leftmost* derivation, the leftmost nonterminal is replaced in each derivation step, e.g.,:

Exp =>
Exp "+" Exp =>
INT "+" Exp =>
INT "+" Exp "*" Exp =>
INT "+" INT "*" Exp =>
INT "+" INT "*" INT

Leftmost and rightmost derivations

p_1: Exp –> Exp "+" Exp
p_2: Exp –> Exp "*" Exp
p_3: Exp –> INT

In a *leftmost* derivation, the leftmost nonterminal is replaced in each derivation step, e.g.,:

Exp =>
Exp "+" Exp =>
INT "+" Exp =>
INT "+" Exp "*" Exp =>
INT "+" INT "*" Exp =>
INT "+" INT "*" INT

In a *rightmost* derivation, the rightmost nonterminal is replaced in each derivation step, e.g.,:

Exp =>
Exp "+" Exp =>
Exp "+" Exp "*" Exp =>
Exp "+" Exp "*" INT =>
Exp "+" INT "*" INT =>
INT "+" INT "*" INT

LL parsing algorithms use leftmost derivation.
LR parsing algorithms use rightmost derivation.
Will be discussed in later lectures.

A derivation corresponds to building a parse tree

Grammar:

Exp –> Exp "+" Exp
Exp –> Exp "*" Exp
Exp –> INT

Example derivation:

Exp =>
Exp "+" Exp =>
INT "+" Exp =>
INT "+" Exp "*" Exp =>
INT "+" INT "*" Exp =>
INT "+" INT "*" INT

Exercise: draw the parse tree (also called derivation tree).

A derivation corresponds to building a parse tree

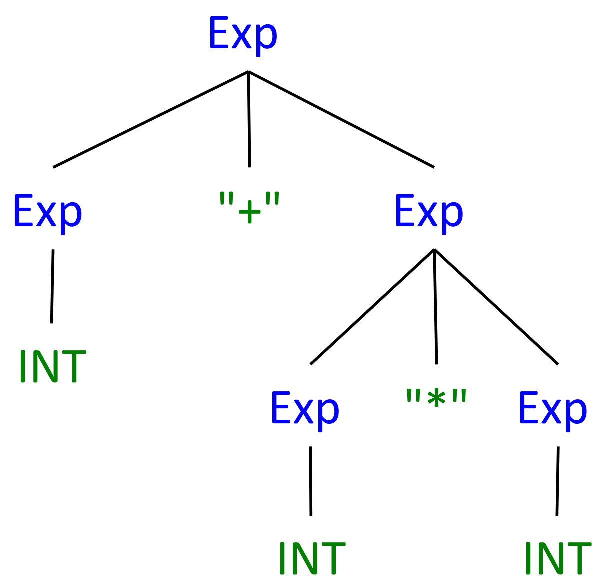
Grammar:

Exp –> Exp "+" Exp
Exp –> Exp "*" Exp
Exp –> INT

Example derivation:

Exp =>
Exp "+" Exp =>
INT "+" Exp =>
INT "+" Exp "*" Exp =>
INT "+" INT "*" Exp =>
INT "+" INT "*" INT

Parse tree (derivation tree):



Ambiguities

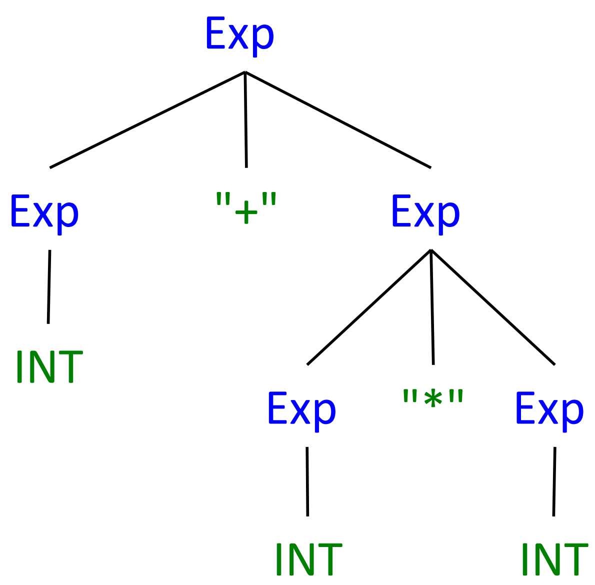
Exercise:

Can we do another derivation of the same sentence, that gives a different parse tree?

```
Exp –> Exp "+" Exp
Exp –> Exp "*" Exp
Exp –> INT
```

One derivation and parse tree

```
Exp =>
Exp "+" Exp =>
INT "+" Exp =>
INT "+" Exp "*" Exp =>
INT "+" INT "*" Exp =>
INT "+" INT "*" INT
```



Other derivation that gives *different* parse tree

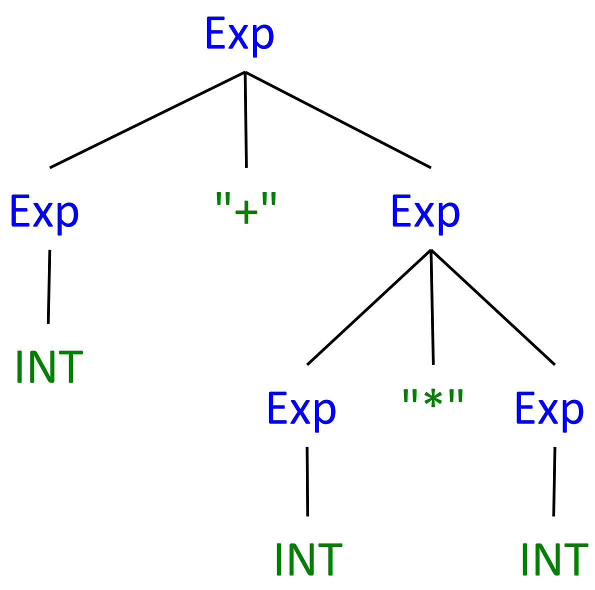
Solution:

Can we do another derivation of the same sentence, that gives a different parse tree?

```
Exp –> Exp "+" Exp
Exp –> Exp "*" Exp
Exp –> INT
```

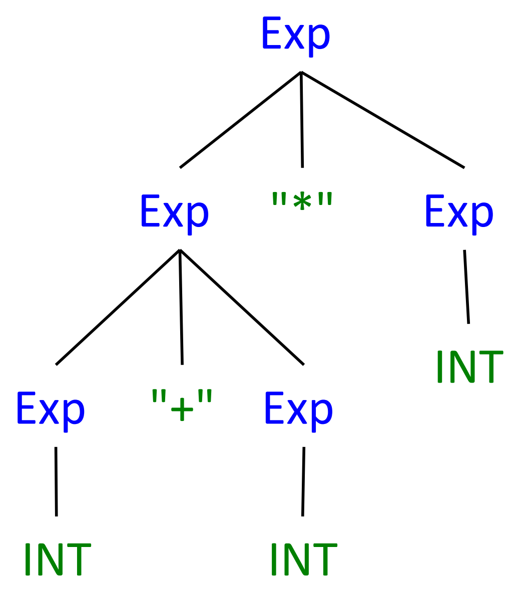
One derivation and parse tree

```
Exp =>
Exp "+" Exp =>
INT "+" Exp =>
INT "+" Exp "*" Exp =>
INT "+" INT "*" Exp =>
INT "+" INT "*" INT
```



Other derivation that gives *different* parse tree

```
Exp =>
Exp "*" Exp =>
Exp "+" Exp "*" Exp =>
INT "+" Exp "*" Exp =>
INT "+" INT "*" Exp =>
INT "+" INT "*" INT
```



Which parse tree would we prefer?

Ambiguous context-free grammars

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

A CFG is *unambiguous* if each sentence in the language can be derived by only *one* parse tree.

(Swedish: *tvetydig, otvetydig*)

Note! There can be many different derivations that give the same parse tree.

How can we know if a CFG is ambiguous?

How can we know if a CFG is ambiguous?

If we find an example of an ambiguity, we know the grammar is ambiguous.

There are algorithms for deciding if a CFG belongs to certain subsets of CFGs, e.g. LL, LR, etc. (See later lectures.) These grammars are unambiguous.

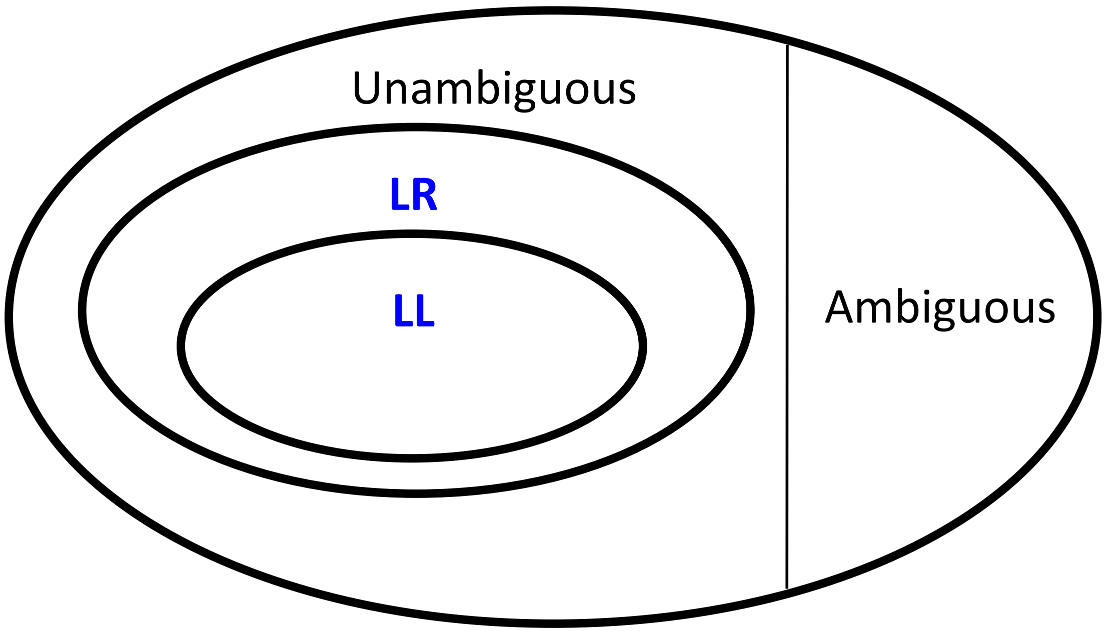
But in the general case, the problem is *undecidable*: it is not possible to construct a general algorithm that decides ambiguity for an arbitrary CFG.

Strategies for eliminating ambiguities, next lecture.

Parsing

Different parsing algorithms

Different parsing algorithms

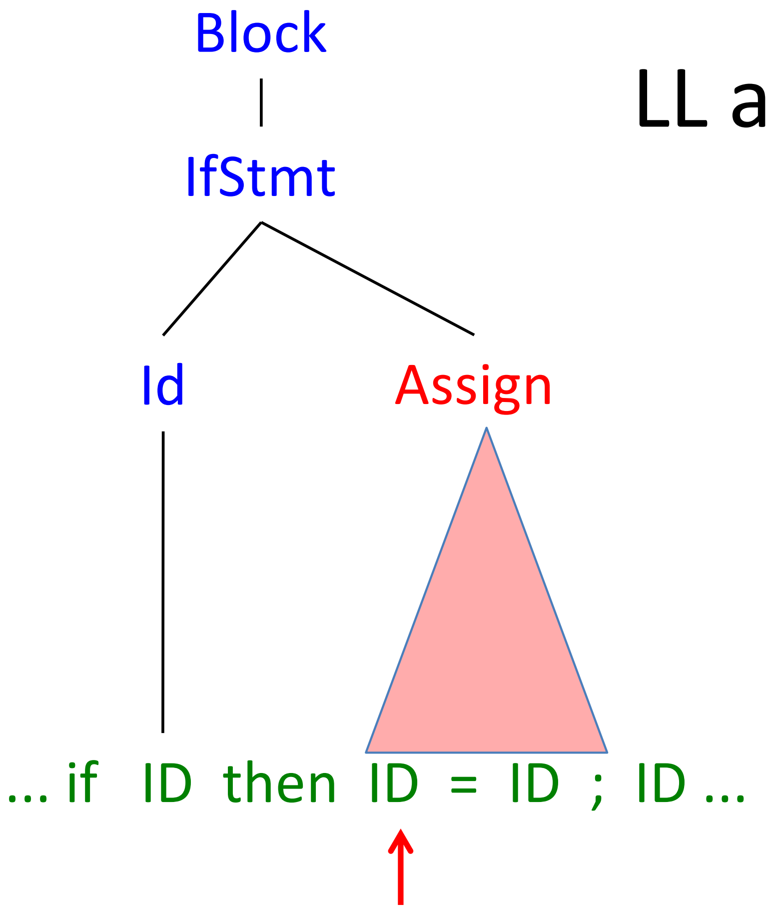


All context-free grammars

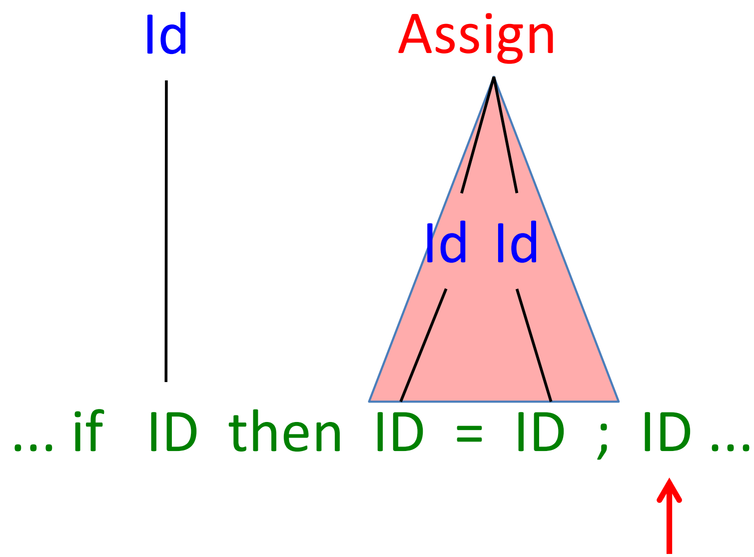
LL:
Left-to-right scan
Leftmost derivation
Builds tree top-down
Simple to understand

LR:
Left-to-right scan
Rightmost derivation
Builds tree bottom-up
More powerful

LL and LR parsers: main idea

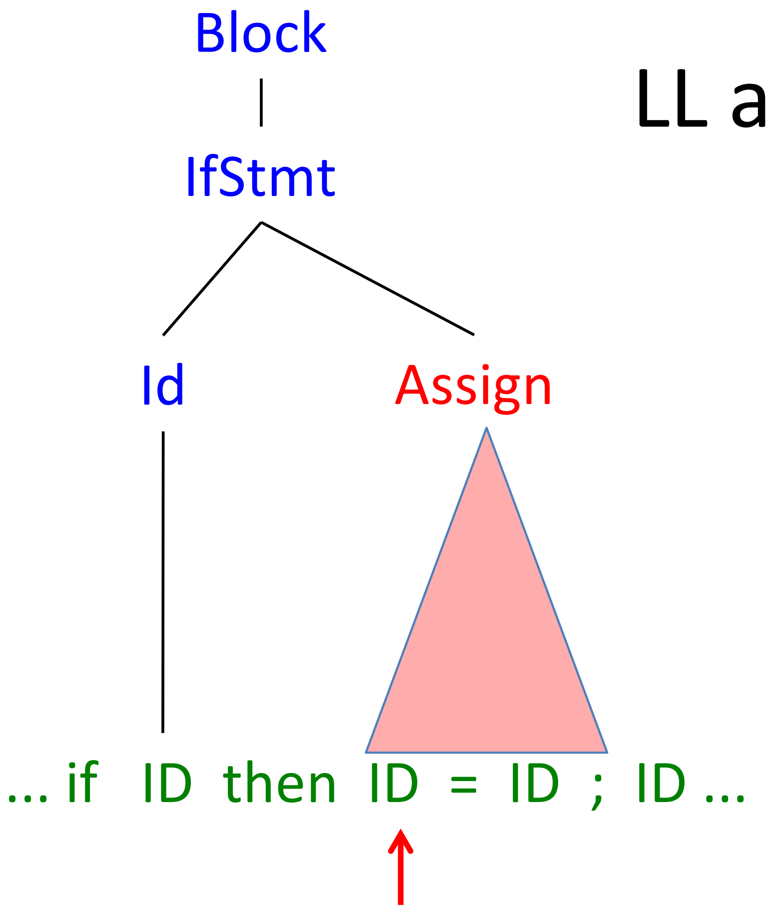


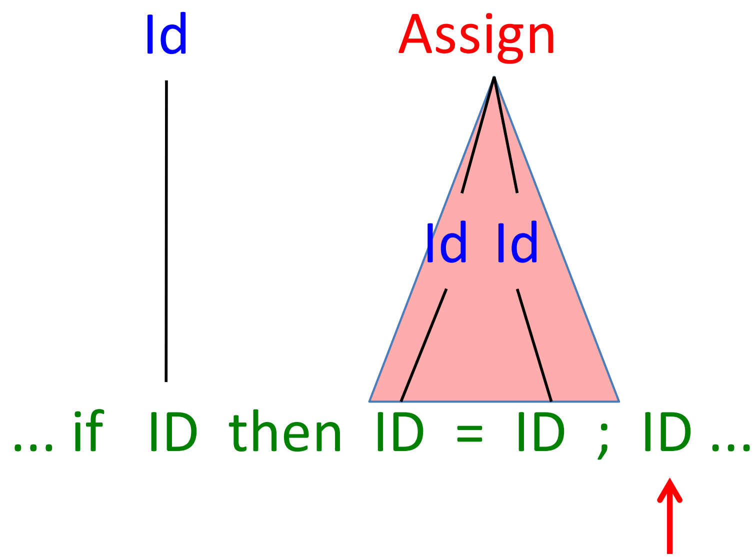
LL(1)



LR(1)

LL and LR parsers: main idea





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

LR(1): decides to build Assign after seeing the first token following its subtree. The tree is built bottom up.

The token is called lookahead.
LL(*k*) and LR(*k*) use *k* lookahead tokens.

In practice, k=1 is usually used

Recursive-descent parsing

A way of programming an LL(1) parser by recursive method calls

```
A –> B | C | D
B –> e C f D
C –> ...
D –> ...
```

Recursive-descent parsing

A way of programming an LL(1) parser by recursive method calls

A –> B | C | D
B –> e C f D
C –> ...
D –> ...

Assume a BNF grammar with exactly *one* production rule for each nonterminal. (Can easily be generalized to EBNF.)

Each production rule RHS is either

1. a sequence of token/nonterminal symbols, or
2. a set of nonterminal symbol alternatives

For each nonterminal, a method is constructed. The method

1. matches tokens and calls nonterminal methods, or
2. calls one of the nonterminal methods – which one depends on the lookahead token.

If the lookahead token does not match, a parsing error is reported.

Example Java implementation: overview

```
statement –> assignment | block
assignment –> ID ASSIGN expr SEMICOLON
block –> LBRACE statement* RBRACE
...
```

```
class Parser {
  private int token;              // current lookahead token
  void accept(int t) {...}        // accept t and read in next token
  void error(String str) {...}    // generate error message
  void statement() {...}
  void assignment() {...}
  void block() {...}
  ...
}
```

Example: Parser skeleton details

```
statement –> assignment | block
assignment–> ID ASSIGN expr SEMICOLON
block –> LBRACE statement* RBRACE
expr –> ...
```

```
class Parser {
  final static int ID=1, WHILE=2, DO=3, ASSIGN=4, ...;
  private int token;                 // current lookahead token
  void accept(int t) {               // accept t and read in next token
    if (token==t) {
      token = nextToken();
    } else {
      error("Expected " + t + " , but found " + token);
    }
  }
  void error(String str) {...}       // generate error message
  private int nextToken() {...}      // read next token from scanner
  void statement() ...
  ...

}
```

Example: recursive descent methods

```
statement –> assignment | block
assignment–> ID ASSIGN expr SEMICOLON
block –> LBRACE statement* RBRACE
```

```
class Parser {
  void statement() {
    switch(token) {
      case ID: assignment(); break;
      case LBRACE: block(); break;
      default: error("Expecting statement, found: " + token);
    }
  }
  void assignment() {
    accept(ID); accept(ASSIGN); expr(); accept(SEMICOLON);
  }
  void block() {
    accept(LBRACE);
    while (token!=RBRACE) { statement(); }
    accept(RBRACE);
  }
  ...
}
```

Is this grammar LL(1)?

expr –> name params | name

What would happen in a recursive-descent parser?

Could the grammar be LL(2)? LL(k)?

Is this grammar LL(1)?

expr –> name params | name

This is called *common prefix*

What would happen in a recursive-descent parser?
Answer: The expr method would not know which alternative to call

Could the grammar be LL(2)? LL(k)?
Answer: This depends on the definition of *name*

Is this grammar LL(1)?

```
expr –> expr "+" term
```

What would happen in a recursive-descent parser?

Could the grammar be LL(2)? LL(k)?

Is this grammar LL(1)?

```
expr –> expr "+" term
```

This is called *left recursion*

What would happen in a recursive-descent parser?
Answer: The expr method would call expr recursively without reading any token, resulting in an endless recursion.

Could the grammar be LL(2)? LL(k)?
Answer: No.

Dealing with common prefix of limited length:

Local lookahead

LL(2) grammar:

statement –> assignment | block | callStmt
assignment–> ID ASSIGN expr SEMICOLON
block –> LBRACE statement* RBRACE
callStmt –> ID LPAR expr RPAR SEMICOLON

```
void statement() ...
```

Dealing with common prefix of limited length:

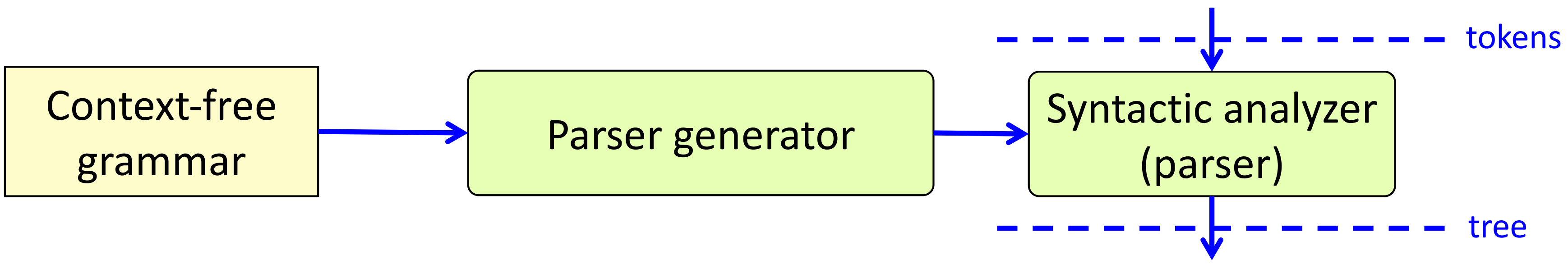
Local lookahead

LL(2) grammar:

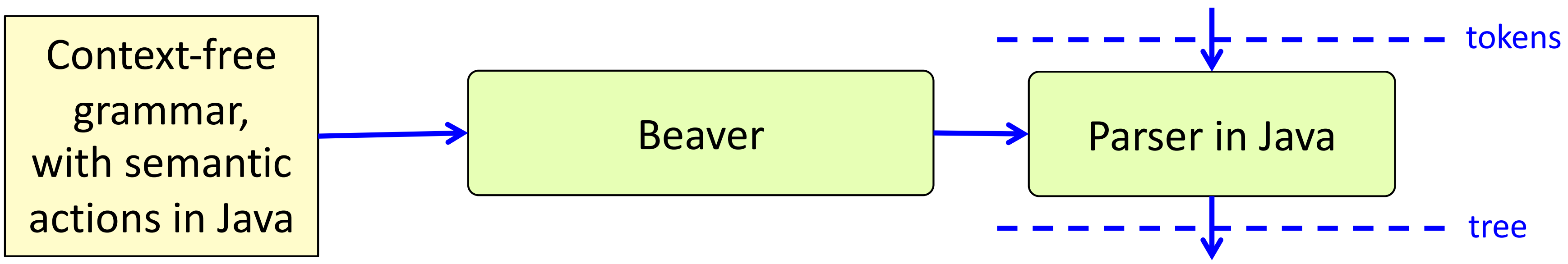
statement –> assignment | block | callStmt
assignment–> ID ASSIGN expr SEMICOLON
block –> LBRACE statement* RBRACE
callStmt –> ID LPAR expr RPAR SEMICOLON

```
void statement() {
 switch(token) {
  case ID:
   if (lookahead(2) == ASSIGN) {
    assignment();
   } else {
    callStmt();
   }
   break;
  case LBRACE: block(); break;
  default: error("Expecting statement, found: " + token);
 }
}
```

Generating the parser:



Beaver: an LR-based parser generator



Example beaver specification

```
%class "LangParser";
%package "lang";
...
%terminals LET, IN, END, ASSIGN, MUL, ID, NUMERAL;

%goal program; // The start symbol

// Context-free grammar
program = exp;
exp = factor | exp MUL factor;
factor = let | numeral | id;
let = LET id ASSIGN exp IN exp END;
numeral = NUMERAL;
id = ID;
```

Later on, we will extend this specification with semantic actions to build the syntax tree.

Regular Expressions vs Context-Free Grammars

	RE	**CFG**
Typical Alphabet	characters	terminal symbols (tokens)
Language is a set of ...	strings (char sequences)	sentences (token sequences)
Used for...	tokens	parse trees
Power	iteration	recursion
Recognizer	DFA	DFA with stack

The Chomsky hierarchy of formal grammars

Grammar	**Rule patterns**	**Type**
regular	$X \rightarrow aY$ or $X \rightarrow a$ or $X \rightarrow \varepsilon$	3
context free	$X \rightarrow \gamma$	2
context sensitive	$\alpha X \beta \rightarrow \alpha \gamma \beta$	1
arbitrary	$\gamma \rightarrow \delta$	0

a – terminal symbol

$\alpha, \beta, \gamma, \delta$ – *sequences* of (terminal or nonterminal) symbols

Type(3) $\subset$ Type (2) $\subset$ Type(1) $\subset$ Type(0)

The Chomsky hierarchy of formal grammars

Grammar	Rule patterns	Type
regular	$X \to aY$ or $X \to a$ or $X \to \varepsilon$	3
context free	$X \to \gamma$	2
context sensitive	$\alpha X \beta \to \alpha \gamma \beta$	1
arbitrary	$\gamma \to \delta$	0

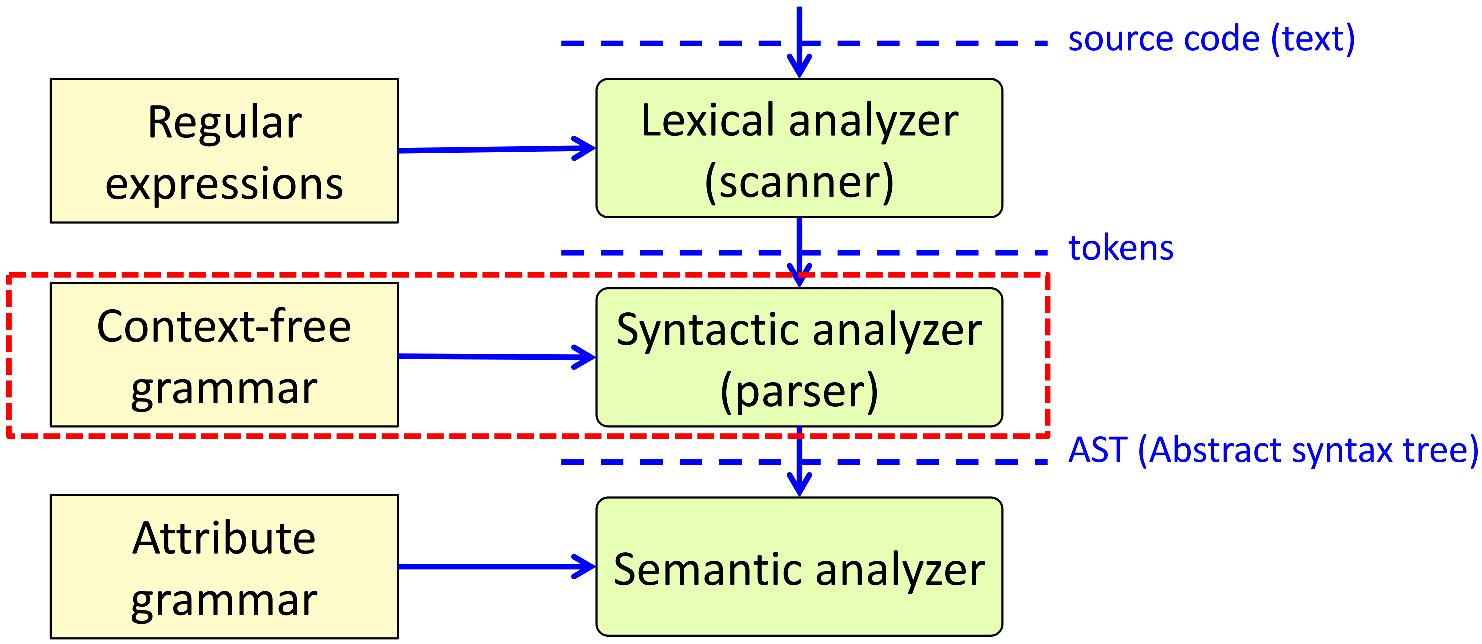
a – terminal symbol
$\alpha, \beta, \gamma, \delta$ – *sequences* of (terminal or nonterminal) symbols

Type(3) $\subset$ Type (2) $\subset$ Type(1) $\subset$ Type(0)

Regular grammars have the same power as regular expressions (tail recursion = iteration).

Type 2 and 3 are of practical use in compiler construction.
Type 0 and 1 are only of theoretical interest.

Course overview



What we have covered:

- **Context-free grammars, derivations, parse trees**
- **Ambiguous grammars**
- **Introduction to parsing, recursive-descent**

You can now finish assignment 1

Summary questions

- Construct a CFG for a simple part of a programming language.
- What is a nonterminal symbol? A terminal symbol? A production? A start symbol? A parse tree?
- What is a left-hand side of a production? A right-hand side?
- Given a grammar G, what is meant by the language L(G)?
- What is a derivation step? A derivation? A leftmost derivation? A righmost derivation?
- How does a derivation correspond to a parse tree?
- What does it mean for a grammar to be ambiguous? Unambiguous?
- Give an example an ambiguous CFG.
- What is the difference between an LL and an LR parser?
- What is the difference between LL(1) and LL(2)? Or between LR(1) and LR(2)?
- Construct a recursive descent parser for a simple language.
- Give typical examples of grammars that cannot be handled by a recursive-descent parser.
- Explain why context-free grammars are more powerful than regular expressions.
- In what sense are context-free grammars "context-free"?