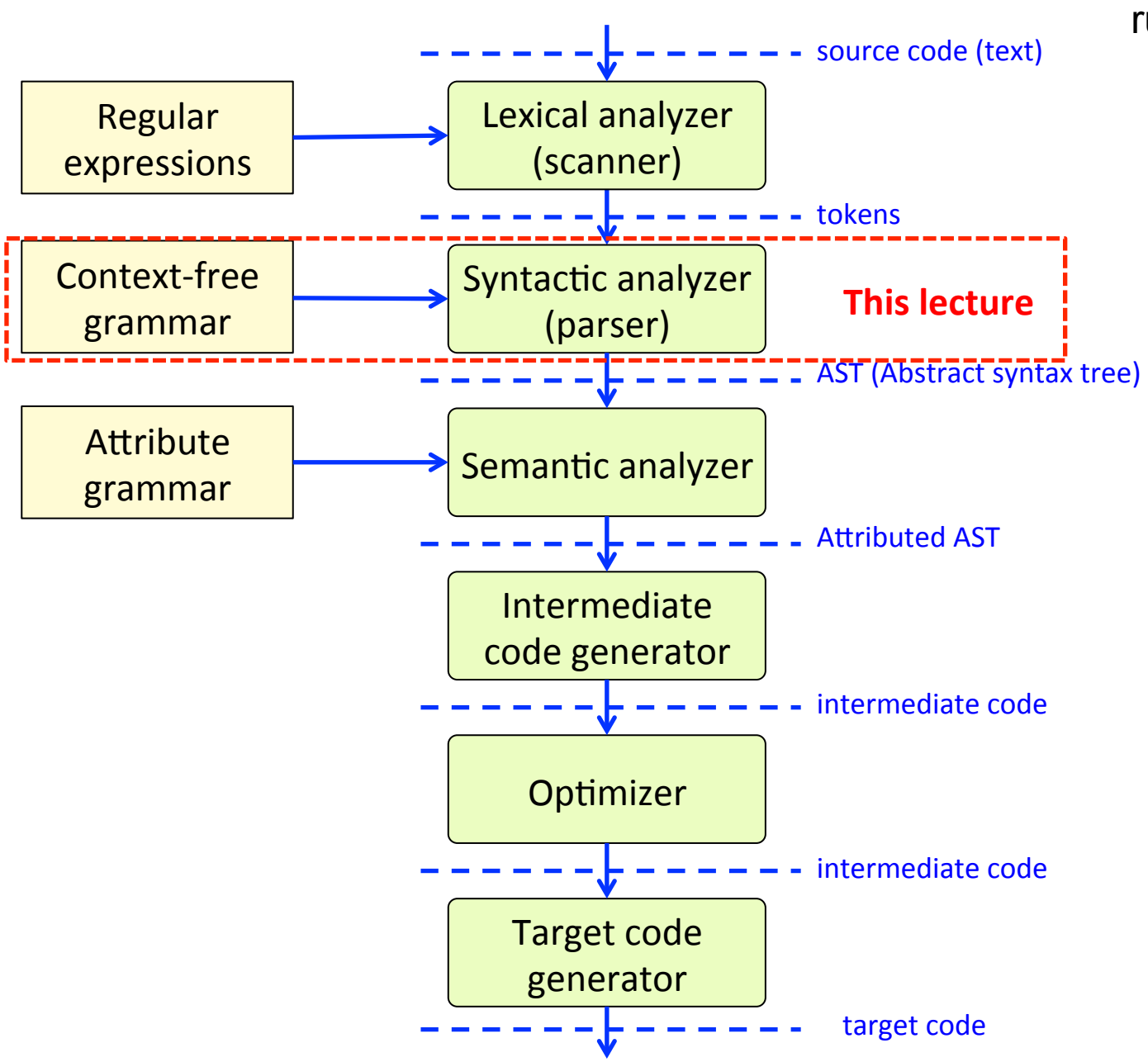
EDAN65: Compilers, Lecture 03

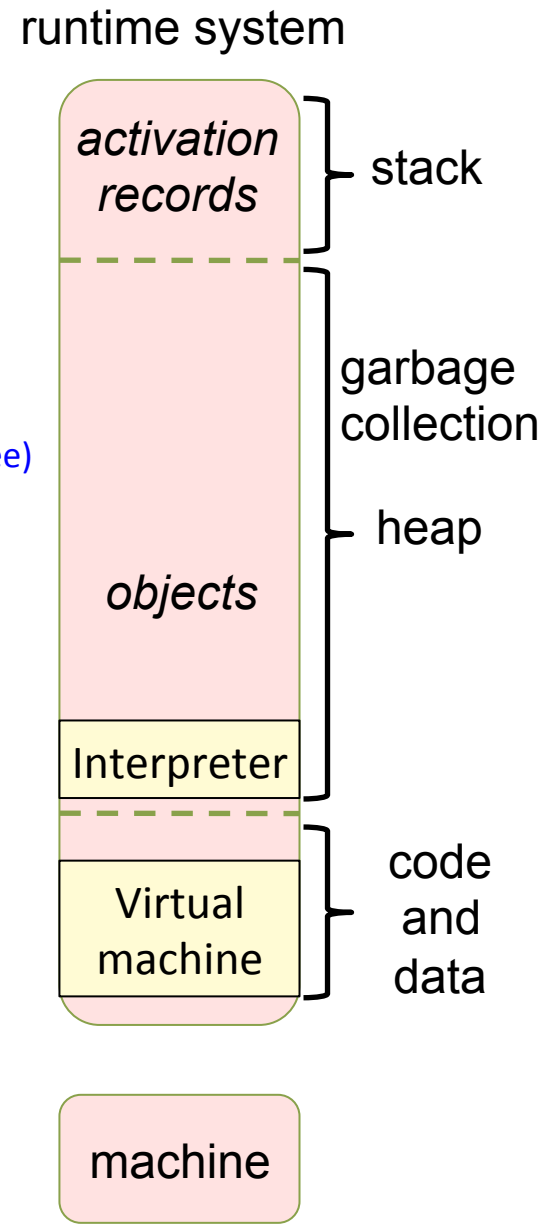
# Context-free grammars, introduction to parsing

Görel Hedin

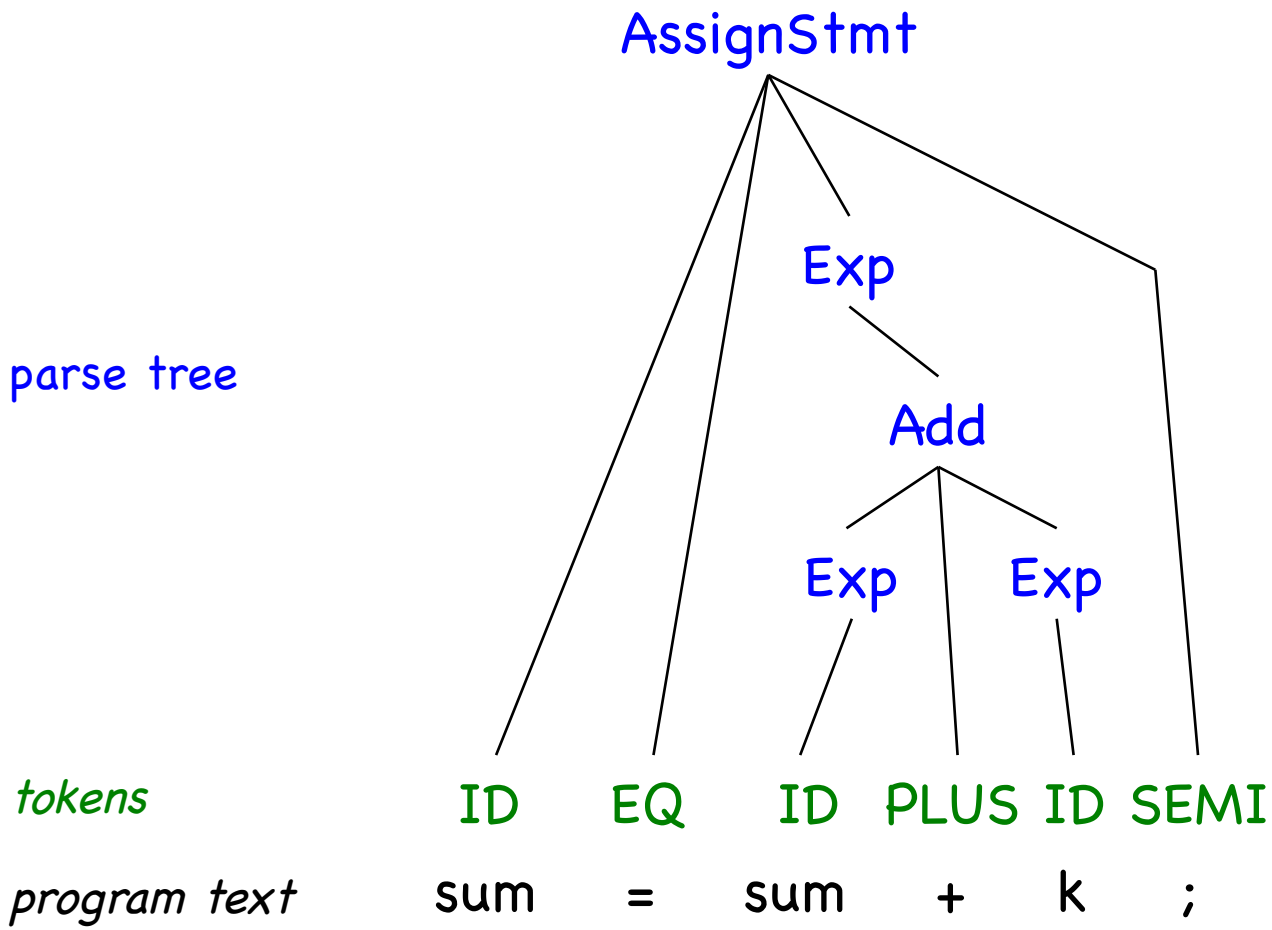
Revised: 2015-09-07

# Course overview

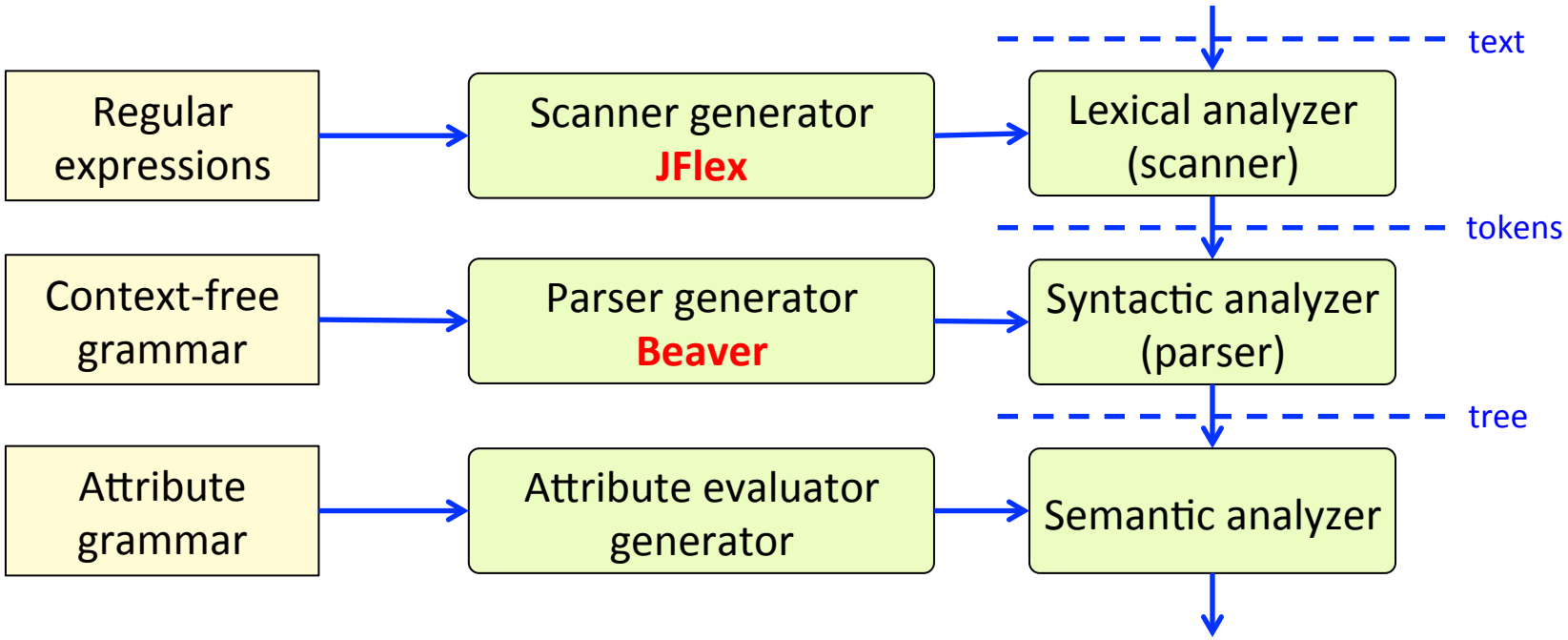


# Analyzing program text



# 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*:

$X \rightarrow s_1\ s_2\ \ldots\ s_n$

where $s_k$ is a *symbol* (terminal or nonterminal)

*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
Stmt -> AssignStmt
```

is equivalent to

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

# Exercise

Construct a grammar covering the following program:

**Example program:**

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

**CFG:**

```
Stmt -> WhileStmt | AssignStmt | CompoundStmt
WhileStmt -> "while" "(" Exp ")" Stmt
AssignStmt -> ID "=" Exp ";"
CompoundStmt -> …
Exp -> …
LessEq -> …
Add -> …
```

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

# Solution

Construct a grammar covering the following program:

**Example program:**

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

**CFG:**

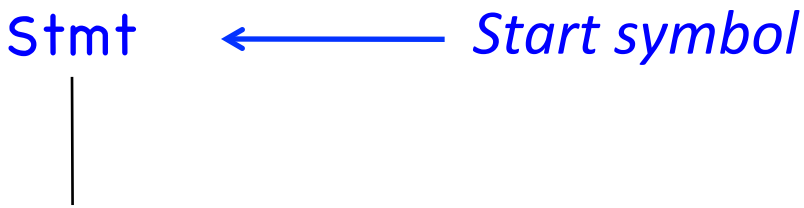
```
Stmt -> WhileStmt | AssignStmt | CompoundStmt
WhileStmt -> "while" "(" Exp ")" Stmt
AssignStmt -> ID "=" Exp ";"
CompoundStmt -> "{" Stmt* "}"
Exp -> LessEq | Add | ID | INT
LessEq -> Exp "<=" Exp
Add -> Exp "+" Exp
```

# Parsing

Use the grammar to derive a tree for a program:

**Example program:**

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



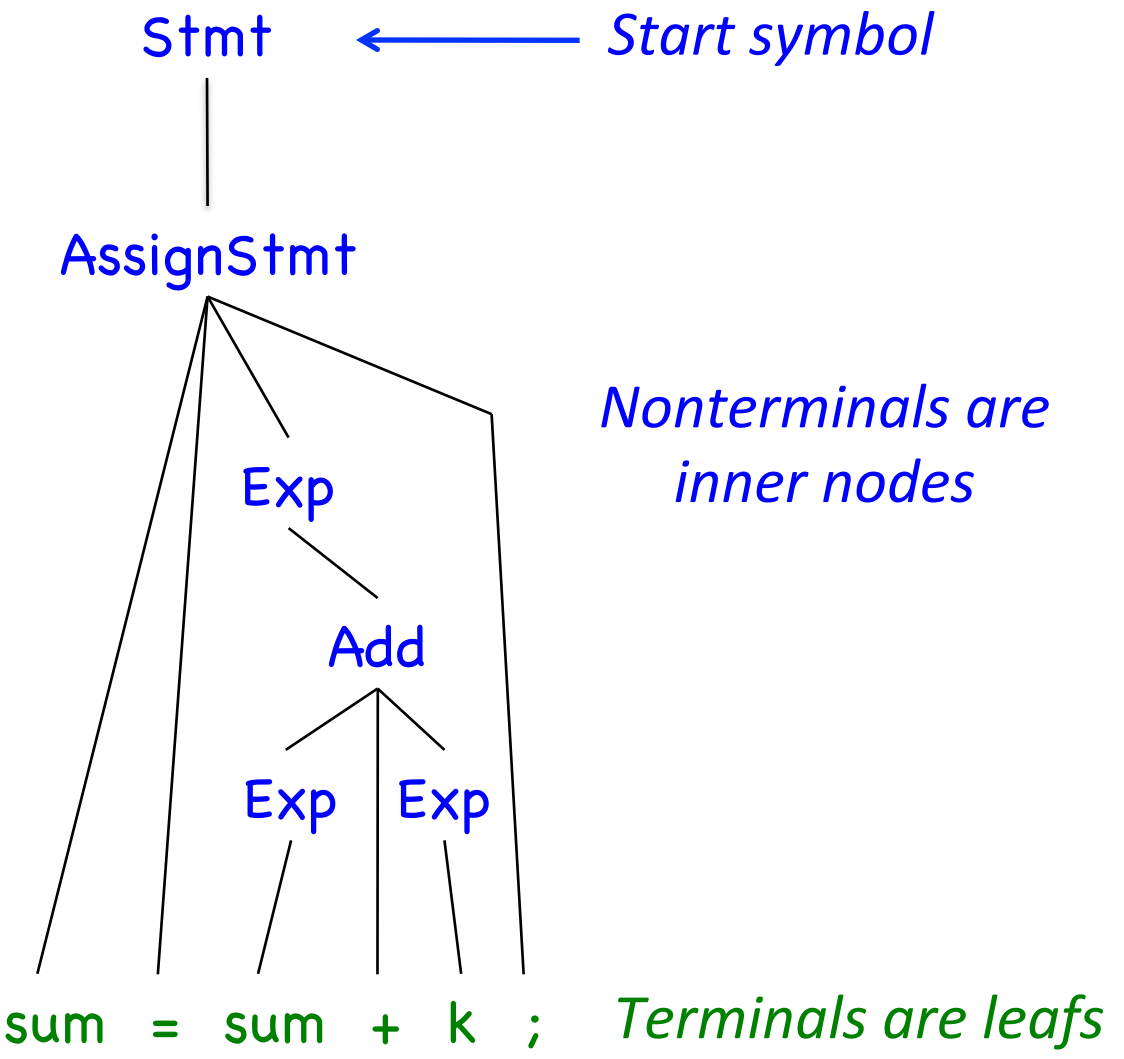

sum = sum + k ;

# Parse tree

Use the grammar to derive a parse tree for a program:

**Example program:**

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



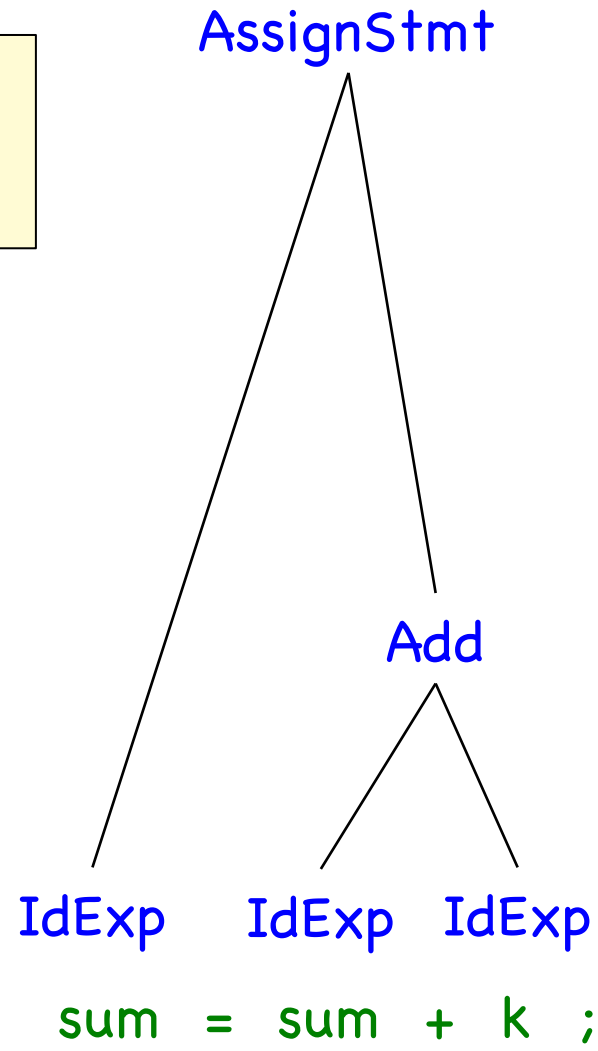

A parse tree includes *all* the tokens as leafs.

# Corresponding abstract syntax tree

(will be discussed in later lecture)

**Example program:**

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




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

It does not include all the tokens.

# EBNF vs Canonical Form

**EBNF:**

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

**(Extended) Backus-Naur Form:**

- Compact, easy to read and write
- EBNF has alternatives, 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

CompilationUnit:
  [PackageDeclaration] {ImportDeclaration} {TypeDeclaration}

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

PackageModifier:
  Annotation

…

See `http://docs.oracle.com/javase/specs/jls/se8/html/index.html`

- See Chapter 2 about the Java grammar notation.
- Look at some other chapters to see other syntax examples.

# 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

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

where $X \in N$, 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

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

where $X \in N$, 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*)

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.:

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

where each $\gamma_i$ is a sequence of terminals and nonterminals

If there is a derivation from $\gamma_0$ to $\gamma_n$, we can write this as

$\gamma_0 \Rightarrow^* \gamma_n$

So this means it is possible to get from the sequence $\gamma_0$ to the sequence $\gamma_n$ by following the 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 following the production rules $P$.

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

Exp
=>

# 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 | |
| => Exp "+" Exp | ($p_1$) |
| => INT "+" Exp | ($p_3$) |
| => INT "+" Exp "*" Exp | ($p_2$) |
| => INT "+" INT "*" Exp | ($p_3$) |
| => INT "+" INT "*" INT | ($p_3$) |

# Leftmost and rightmost derivations

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
```

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

Example derivation:

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

# A derivation corresponds to building a parse tree

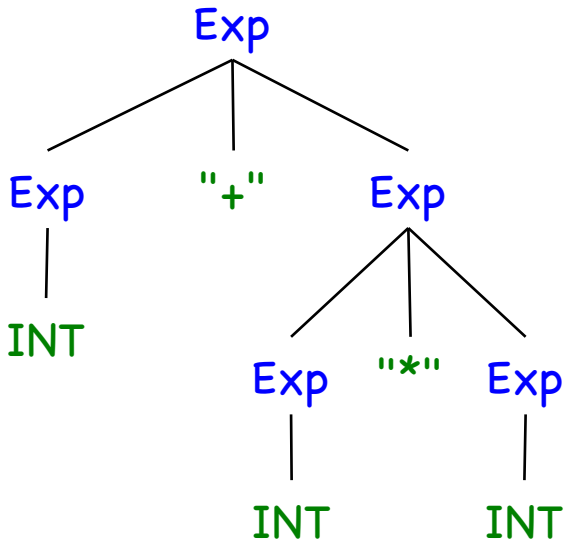
Grammar:

```
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?

Grammar:

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

Parse tree:

Another derivation:

```
Exp =>
```

# Solution:

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

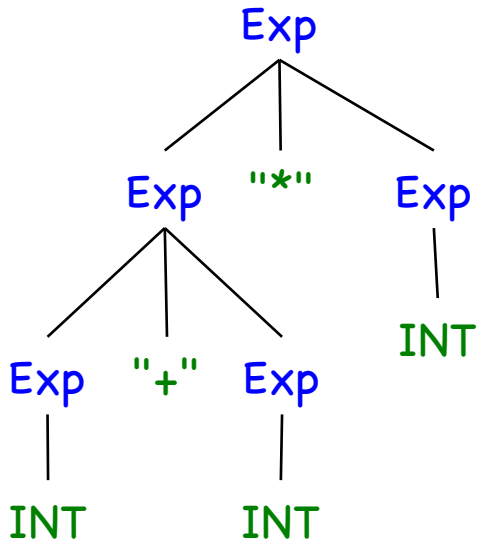
Grammar:

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

Another derivation:

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

Parse tree:




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?

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

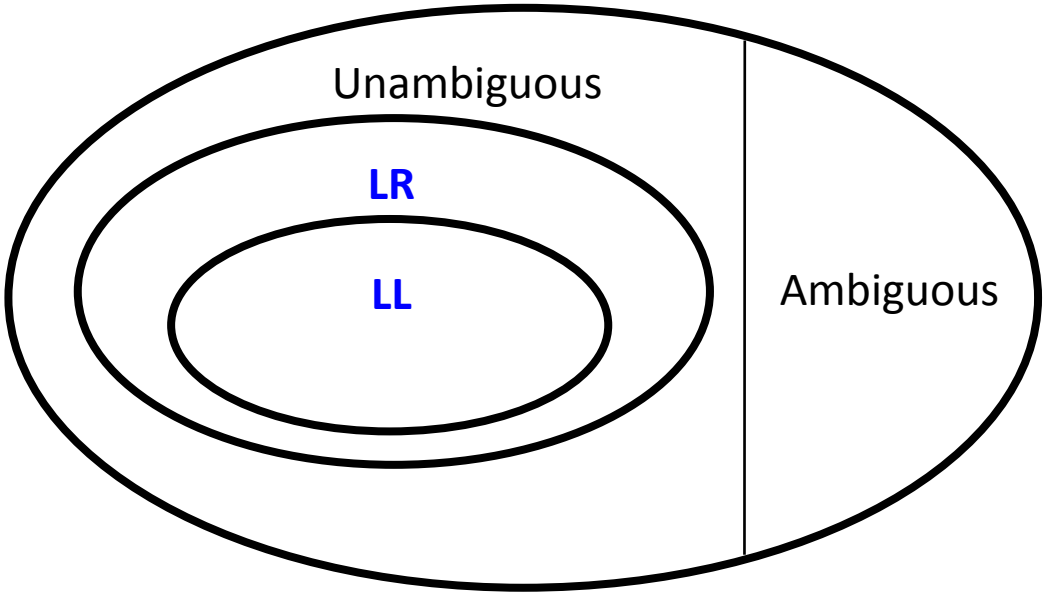
There are algorithms for deciding if a CFG belongs to certain subsets of CFGs, e.g. LL, LR, etc. (See later lectures.) These grammars are unambiguous.

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

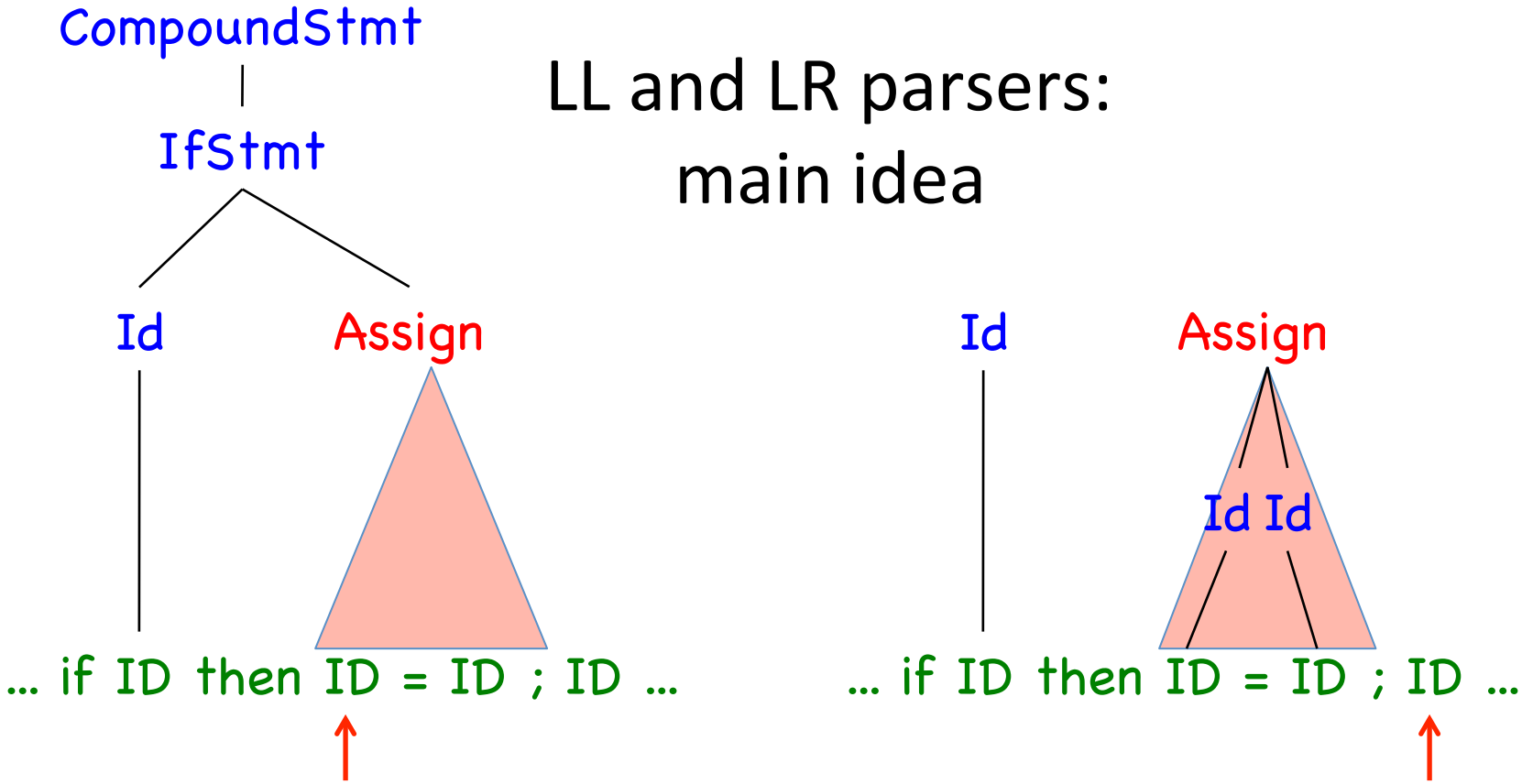
# Different parsing algorithms




**LL:**
**L**eft-to-right scan
**L**eftmost derivation
Builds tree top-down
Simple to understand

**LR:**
**L**eft-to-right scan
**R**ightmost derivation
Builds tree bottom-up
More powerful

# 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.

# Recursive-descent parsing

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

Assume an EBNF grammar with exactly *one* production rule for each nonterminal symbol.

For each  nonterminal, a method is constructed.

A nonterminal method matches tokens and calls other nonterminal methods, according to the grammar.

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

```
A -> B | C | D
B -> e C f D
C -> …
D -> …
```

# Example Java implementation: overview

```
statement -> assignment | compoundStmt
assignment-> ID ASSIGN expr SEMICOLON
compoundStmt -> 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 compoundStmt () {...}
  ...




}
```

# Example: recursive descent methods

```
statement -> assignment | compoundStmt
assignment-> ID ASSIGN expr SEMICOLON
compoundStmt -> LBRACE statement* RBRACE
```

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

# Example: Parser skeleton details

```
statement -> assignment | compoundStmt
assignment-> ID ASSIGN expr SEMICOLON
compoundStmt -> 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() …
  …

}
```

# Are these grammars LL(1)?

```
expr -> name params | name
```

Common prefix

```
expr -> expr "+" term
```

Left recursion

What would happen in a recursive-descent parser?

Could they be LL(2)? LL(k)?

# Dealing with common prefix of limited length:

Local lookahead

LL(2) grammar:

```
statement -> assignment | compoundStmt | callStmt
assignment-> ID ASSIGN expr SEMICOLON
compoundStmt -> LBRACE statement* RBRACE
callStmt -> ID LPAR expr RPAR SEMICOLON
```

```
void statement() …
```

# Dealing with common prefix of limited length:

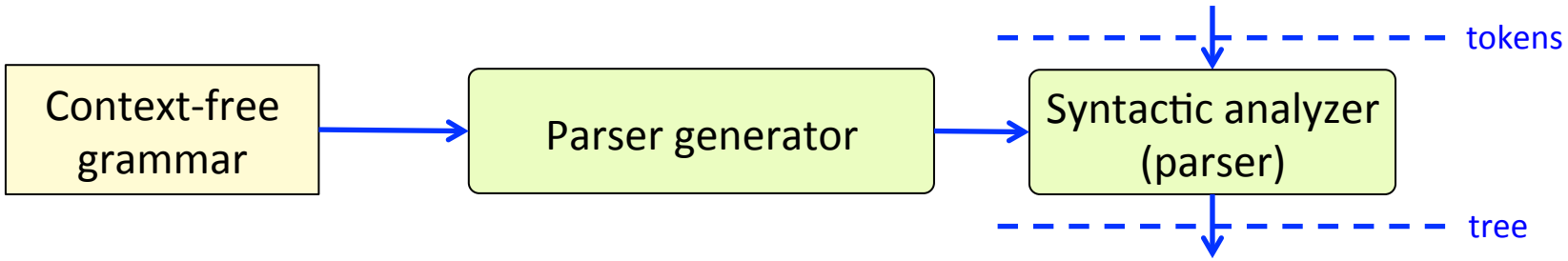
## Local lookahead

LL(2) grammar:

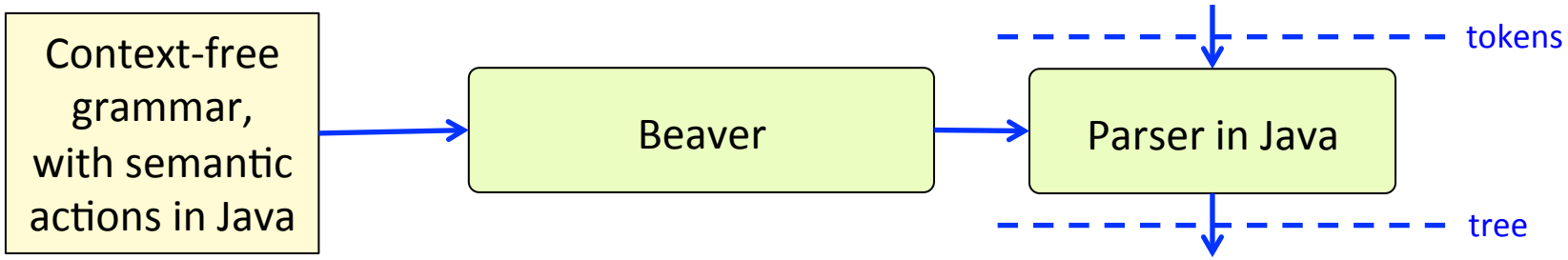
```
statement -> assignment | compoundStmt | callStmt
assignment-> ID ASSIGN expr SEMICOLON
compoundStmt -> LBRACE statement* RBRACE
callStmt -> ID LPAR expr RPAR SEMICOLON
```

```
void statement() {
  switch(token) {
    case ID:
      if (lookahead(2) == ASSIGN) {
        assignment();
      } else {
        callStmt();
      }
      break;
    case LBRACE: compoundStmt(); 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** | 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

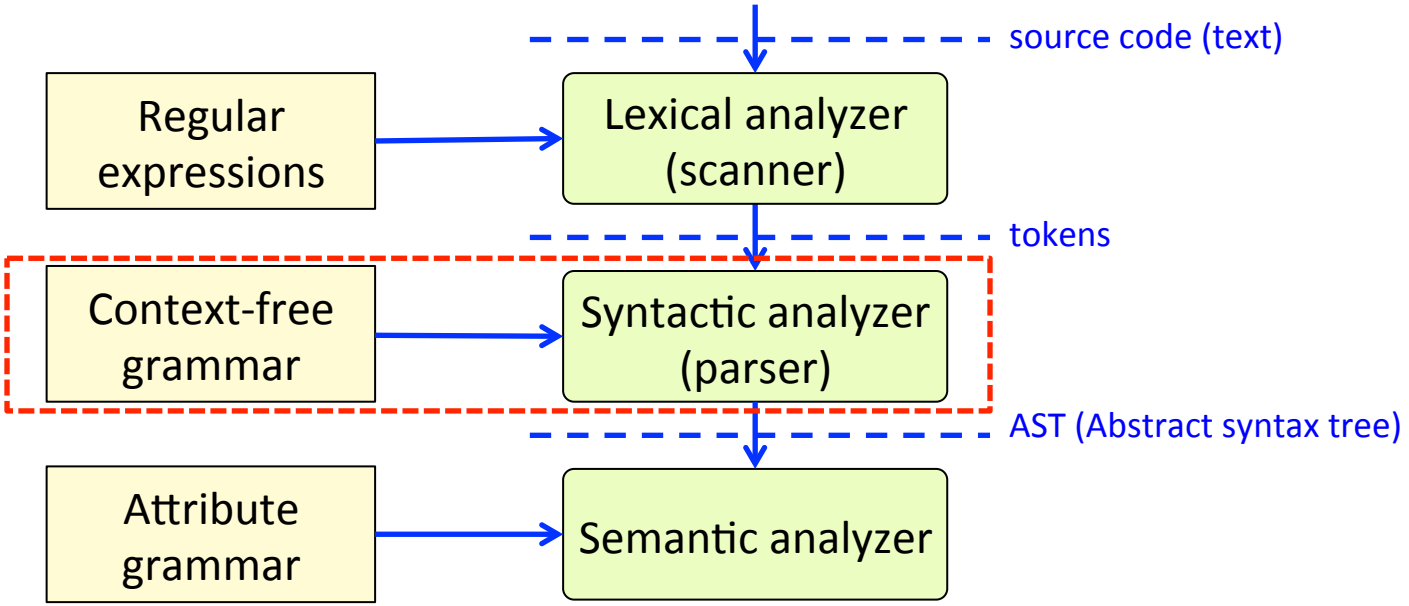
| Grammar | Rule patterns | Type |
|---|---|---|
| regular | X -> aY or X -> a or X -> ε | 3 |
| context free | X -> γ | 2 |
| context sensitive | α X β -> α γ β | 1 |
| arbitrary | γ -> δ | 0 |

Type(3) ⊂ Type (2) ⊂ Type(1) ⊂ 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"?