Context-Free Grammars, Languages and Parsing

Nguyen Phuong Thai
Faculty of Information Technology
The University of Engineering and Technology
VNU Hanoi

Lecture 3: Context-Free Grammars, Languages and Parsing

1. The syntax and semantics of a programming language
2. Specify a language's syntax: CFG, BNF and EBNF
3. The parsing of a program in a language:
   - Construct leftmost and rightmost derivations
   - Construct parse trees
4. The structure of a grammar:
   - How language constructs are defined
   - The precedence and associativity of operators
   - Ambiguity
5. The Chomsky hierarchy
6. The equivalence between regular grammars and finite automata

English: Syntax and Semantics

1. **John eats apples**
2. **Apples eat John**

- Syntax:
  - The form or structure of English sentences
  - No concern with the meaning of English sentences
  - Specified by the English grammar
- Semantics:
  - The meaning of English sentences
  - How is the English semantics defined?

Programming Languages: Syntax and Semantics

1. **i = i + 1;**
2. **if (door.isOpen()) System.out.println("hello");**

- Syntax:
  - The form or structure of programs
  - No concern with the meaning of programs
  - Specified by a context-free grammar (CFG)
- Semantics:
  - The meaning of programs
  - Specified by
    * operational, denotational or axiomatic semantics,
    * attribute grammars (Week 7), or
    * an informal English description as in C, Java and VC

Programming Languages: Syntax and Semantics

- Syntax:
  - The form or structure of a program and individual statements in the language
  - No concern with the meaning of a program
  - Specified by a CFG (universally used in compiler construction)
- Semantics:
  - Static Semantics: Context-sensitive restrictions enforced at compile-time
    * All identifiers declared before used
    * Assignment must be type-compatible
    * Operands must be type-compatible with operators
    * Methods called with the proper number of arguments
    * Assignment 4: context handling module for static semantics
  - Run-Time Semantics: What the program does or computes
    * The meaning of a program or what happens when it is executed.
    * Specified by code generation routines.

Static Semantics: Undeclared Variables

```
public  class Foo {
  public static void main(String argv[]) {
      i = 10;
  }
}
javac Foo.java
Foo.java:3: Undefined variable: i
      i = 10;
      ^
1 error
```

- Grammatical
- But has a semantic error: undeclared variable

Static Semantics: Assignment Incompatible

```
public  class Foo {
  public static void main(String argv[]) {
      int i;
      float f = 10;
      i = f;
  }
}
javac Foo.java
Foo.java:5: Incompatible type for =.
Explicit cast needed to convert float to int.
      i = f;
        ^
1 error
```

- Grammatical
- Semantic error: assignment incompatible

Static Semantics: Operands with Incompatible Types

```
public  class Foo {

  public static void main(String argv[]) {
     int i = 1 + main;
  }
}
javac Foo.java
Foo.java:4: Reference to method main in class Foo
as if it were a variable.
     int i = 1 + main;
                 ^
1 error
```

- Grammatical
- Semantic error: incompatible operand type

Static Semantics: Wrong Number of Arguments

```
public  class Foo {
  void sub(int i) { };

  public static void main(String argv[]) {
      (new Foo()).sub(1, 2);
  }
}
javac Foo.java
Foo.java:5: Wrong number of arguments in method.
      (new Foo()).sub(1, 2);
                     ^

1 error
```

- Grammatical
- Semantic error: wrong number of arguments

Lecture 3: Context-Free Grammars, Languages and Parsing

1. The syntax and semantics of a programming language $\surd$
2. Specify a language's syntax: CFG, BNF and EBNF
3. The parsing of a program in a language:
   - Construct leftmost and rightmost derivations
   - Construct parse trees
4. The structure of a grammar:
   - How language constructs are defined
   - The precedence and associativity of operators
   - Ambiguity
5. The Chomsky hierarchy
6. The equivalence between regular grammars and finite automata

Why Grammars at All?

- Give a precise and easy-to-understand syntactic specification of the language.
- New language constructs added easily.
- Facilitate programming language modifications and extensions
- Allow the meaning of the corresponding language to be defined in terms of the grammar (i.e., syntactical structure)
- Scanners and parsers constructed easily.
  - In 1950s, the first FORTRAN took 18 man-years
  - Now, a compiler written by a student in a semester
- Enable syntax-directed translation (Assignment 5)

CFG

- One type of grammar for specifying a language's syntax.
- Simple, widely used, and sufficient for most purposes.
- Not the most powerful syntax description tool. Powerful grammars like context-sensitive grammars and phrase-structure grammars too complex to be useful.
- Regular grammars (i.e., regular expressions) are less powerful

In this course, only CFGs and regular grammars required

Formal Definition of CFG

A grammar G is a quadruple (V_T, V_N, S, P), where

- V_T: a finite set of terminal symbols or tokens
- V_N: a finite set of nonterminal symbols ($V_T \cap V_N = \emptyset$)
- S: a unique start symbol ($S \in N$)
- P: a finite set of rules or productions of the form (A, α) where:
  - A is a nonterminal, and
  - α is a string of zero or more terminals and nonterminals

Note: zero means that $\alpha = \epsilon$ is possible

Backus-Naur Form (BNF)

- A notation for writing a CFG

- To recognise P. Naur's contributions as editor of the ALGOL60 report and J.W. Backus for applying the notation to the first FORTRAN compiler.

- Each production (A, α) is written as:

$$A \rightarrow \alpha$$

where the arrow $\rightarrow$ means "is defined to be", "can have the form of", "may be replaced with" or "derives"

- Can abbreviate the left to the right

$$\boxed{\begin{array}{ccc} A & \rightarrow & \alpha_1 \\ A & \rightarrow & \alpha_2 \\ & \vdots & \\ A & \rightarrow & \alpha_n \end{array}} \quad \Longrightarrow \quad \boxed{\begin{array}{ccc} A & \rightarrow & \alpha_1 \\ & \mid & \alpha_2 \\ & \vdots & \\ & \mid & \alpha_n \end{array}}$$

where:

- $\alpha_1, \ldots, \alpha_n$ are the alternatives of A
- the vertical bar $\mid$ reads "or else"

CFG for micro-English

1	⟨sentence⟩	$\rightarrow$	⟨subject⟩ ⟨predicate⟩
2	⟨subject⟩	$\rightarrow$	**NOUN**
3		$\mid$	**ARTICLE NOUN**
4	⟨predicate⟩	$\rightarrow$	**VERB** ⟨object⟩
5	⟨object⟩	$\rightarrow$	**NOUN**
6		$\mid$	**ARTICLE NOUN**

The four components of a CFG:

- V_N: set of nonterminals:
  - The symbol on the left-hand side of $\rightarrow$
  - The names of language constructs in the language.
- V_T: set of terminals or tokens:
  - The basic language units, parallel to the words in natural languages.
- S: ⟨sentence⟩, i.e., the left-hand side of the 1st production
- P: set of productions or rules of the form: $A \rightarrow X_1 X_2 \cdots X_n$.
  - A: a nonterminal
  - X_i: a terminal (can be ϵ) or nonterminal.

Derivations; Sentential Forms; Sentences; Languages

A grammar derives sentences by

1. beginning with the start symbol, and
2. repeatedly replacing a nonterminal by the right-hand side of a production with that nonterminal on the left-hand side, until there are no more nonterminals to replace.

- Such a sequence of replacements is called a derivation of the sentence being analysed
- The strings of terminals and nonterminals appearing in the various derivation steps are called sentential forms
- A sentence is a sentential form with terminals only
- The language: the set of all sentences thus derived

Verify if

"PETER PASSED THE TEST"

is a sentence?

The Three Derivations of PETER PASSED THE TEST

$$
\begin{array}{lll}
\langle\text{sentence}\rangle & \Longrightarrow \langle\text{subject}\rangle\ \langle\text{predicate}\rangle & \text{by P1} \\
& \Longrightarrow \textbf{NOUN}\ \langle\text{predicate}\rangle & \text{by P2} \\
& \Longrightarrow \textbf{NOUN VERB}\ \langle\text{object}\rangle & \text{by P4} \\
& \Longrightarrow \textbf{NOUN VERB ARTICLE NOUN} & \text{by P6}
\end{array}
$$

$$
\begin{array}{lll}
\langle\text{sentence}\rangle & \Longrightarrow \langle\text{subject}\rangle\ \langle\text{predicate}\rangle & \text{by P1} \\
& \Longrightarrow \langle\text{subject}\rangle\ \textbf{VERB}\ \langle\text{object}\rangle & \text{by P4} \\
& \Longrightarrow \langle\text{subject}\rangle\ \textbf{VERB ARTICLE NOUN} & \text{by P6} \\
& \Longrightarrow \textbf{NOUN VERB ARTICLE NOUN} & \text{by P2}
\end{array}
$$

$$
\begin{array}{lll}
\langle\text{sentence}\rangle & \Longrightarrow \langle\text{subject}\rangle\ \langle\text{predicate}\rangle & \text{by P1} \\
& \Longrightarrow \langle\text{subject}\rangle\ \textbf{VERB}\ \langle\text{object}\rangle & \text{by P4} \\
& \Longrightarrow \textbf{NOUN VERB}\ \langle\text{object}\rangle & \text{by P2} \\
& \Longrightarrow \textbf{NOUN VERB ARTICLE NOUN} & \text{by P6}
\end{array}
$$

- Sentence: **NOUN VERB ARTICLE NOUN**
- Sentential forms: all the others

Leftmost and Rightmost Derivations

At each step in a derivation, two choices are made:

1. Which nonterminal to replace?
2. Which alternative to use for that nonterminal?

- Two types of useful derivations:
  - Leftmost derivation: always replace the leftmost nonterminal.
  - Rightmost derivation: always replace the rightmost nonterminal.

Leftmost and Rightmost Derivations

$$
\begin{array}{lll}
\langle\text{sentence}\rangle & \Longrightarrow_{\text{lm}} \langle\text{subject}\rangle\ \langle\text{predicate}\rangle & \text{by P1} \\
& \Longrightarrow_{\text{lm}} \textbf{NOUN}\ \langle\text{predicate}\rangle & \text{by P2} \\
& \Longrightarrow_{\text{lm}} \textbf{NOUN VERB}\ \langle\text{object}\rangle & \text{by P4} \\
& \Longrightarrow_{\text{lm}} \textbf{NOUN VERB ARTICLE NOUN} & \text{by P6}
\end{array}
$$

$$
\begin{array}{lll}
\langle\text{sentence}\rangle & \Longrightarrow_{\text{rm}} \langle\text{subject}\rangle\ \langle\text{predicate}\rangle & \text{by P1} \\
& \Longrightarrow_{\text{rm}} \langle\text{subject}\rangle\ \textbf{VERB}\ \langle\text{object}\rangle & \text{by P4} \\
& \Longrightarrow_{\text{rm}} \langle\text{subject}\rangle\ \textbf{VERB ARTICLE NOUN} & \text{by P6} \\
& \Longrightarrow_{\text{rm}} \textbf{NOUN VERB ARTICLE NOUN} & \text{by P2}
\end{array}
$$

$$
\begin{array}{lll}
\langle\text{sentence}\rangle & \Longrightarrow \langle\text{subject}\rangle\ \langle\text{predicate}\rangle & \text{by P1} \\
& \Longrightarrow \langle\text{subject}\rangle\ \textbf{VERB}\ \langle\text{object}\rangle & \text{by P4} \\
& \Longrightarrow \textbf{NOUN VERB}\ \langle\text{object}\rangle & \text{by P2} \\
& \Longrightarrow \textbf{NOUN VERB ARTICLE NOUN} & \text{by P6}
\end{array}
$$

neither

The Language Defined by a Grammar

- The language defined by a grammar: all the sentences derived from the grammar.
- The language defined by the micro-English grammar:

NOUN VERB NOUN

NOUN VERB ARTICLE NOUN

ARTICLE NOUN VERB NOUN

ARTICLE NOUN VERB ARTICLE NOUN

Conventions for Writing CFGs

- Start symbol:
  - The left side of the first production
  - The letter S, whenever it appears
- Nonterminals:
  - lower-case $\langle$italic$\rangle$ names such as $\langle$sentence$\rangle$ and $\langle$expr$\rangle$
  - capital letters like A, B, C
- Terminals:
  - **boldface** names such as **ID** and **INTLITERAL**
  - digits and operators such as 1 and + (sometimes in double quotes)
  - lower-case letters such as a, b, c
  - Usually anything non-italic
- Strings of terminals: lower-case letters late in the alphabet such as, $u, v, \cdots, z$
- Mixtures of nonterminals and terminals: lower-case Greek letters, such as $\alpha, \beta, \gamma, \cdots$

Some Formal Notations about Derivations

- $\Longrightarrow$: derivation in one step (one production used)
- $\Longrightarrow^+$: derivation in one or more steps
  - $\langle$sentence$\rangle \Longrightarrow^+ \langle$subject$\rangle$ **VERB** $\langle$object$\rangle$
  - $\langle$sentence$\rangle \Longrightarrow^+$ **NOUN VERB ARTICLE NOUN**
- $\Longrightarrow^*$: derivation in zero or more steps:

$$\begin{array}{lcl} \langle\text{sentence}\rangle & \Longrightarrow^* & \langle\text{sentence}\rangle \\ \langle\text{subject}\rangle\ \langle\text{predicate}\rangle & \Longrightarrow^* & \langle\text{subject}\rangle\ \langle\text{predicate}\rangle \end{array}$$

- The language $L(G)$ defined by a grammar G:

$$L(G) = \{w \mid S \Longrightarrow^+ w\}$$

- The context-free language (CFL): the language generated by a CFG

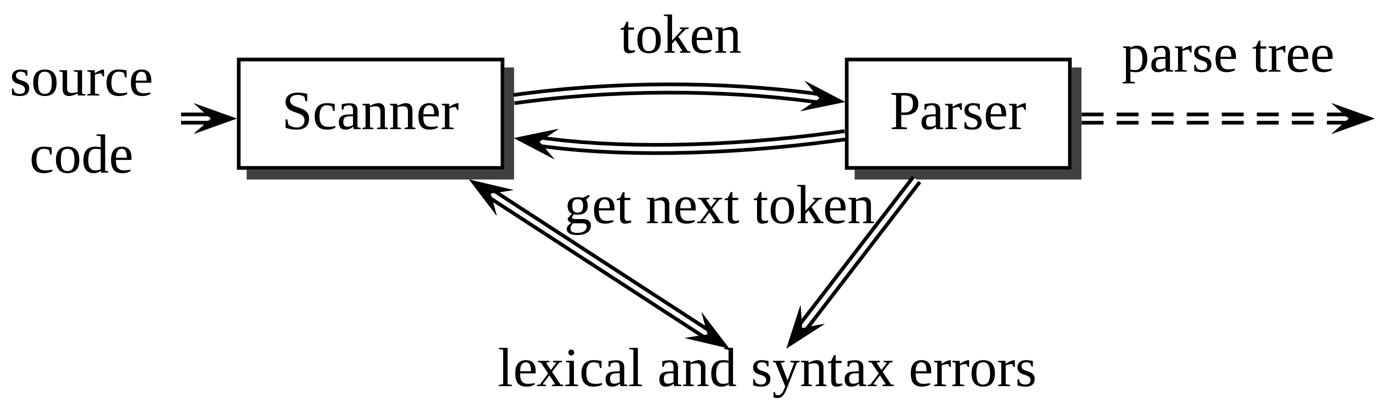
Lecture 3: Context-Free Grammars, Languages and Parsing

1. The syntax and semantics of a programming language $\surd$
2. Specify a language's syntax: CFG, BNF and EBNF $\surd$
3. The parsing of a program in a language:
   - Construct leftmost and rightmost derivations
   - Construct parse trees
4. The structure of a grammar:
   - How language constructs are defined
   - The precedence and associativity of operators
   - Ambiguity
5. The Chomsky hierarchy
6. The equivalence between regular grammars and finite automata

The Parsing of a Sentence (or Program)

- Use syntactic rules to break a sentence into its component parts and analyse their relationship.
- The term parsing used in both linguistic and compiler theory.
- A parser is a program that uses a CFG to parse a sentence or a program (Assignment 3). In particular, it
  - constructs its leftmost or rightmost derivation, or
  - builds the parse tree for the sentence.
- A recogniser is a parser that checks only the syntax (without having to built the parse tree). It outputs YES if the program is legal and NO otherwise (Assignment 2).

The Role of the Parser



- Perform context-free syntactic analysis
- Construct a tree (an AST rather than a parse tree)
- Produce some meaningful error messages
- Attempt error recovery

Parsing: The Derivational View

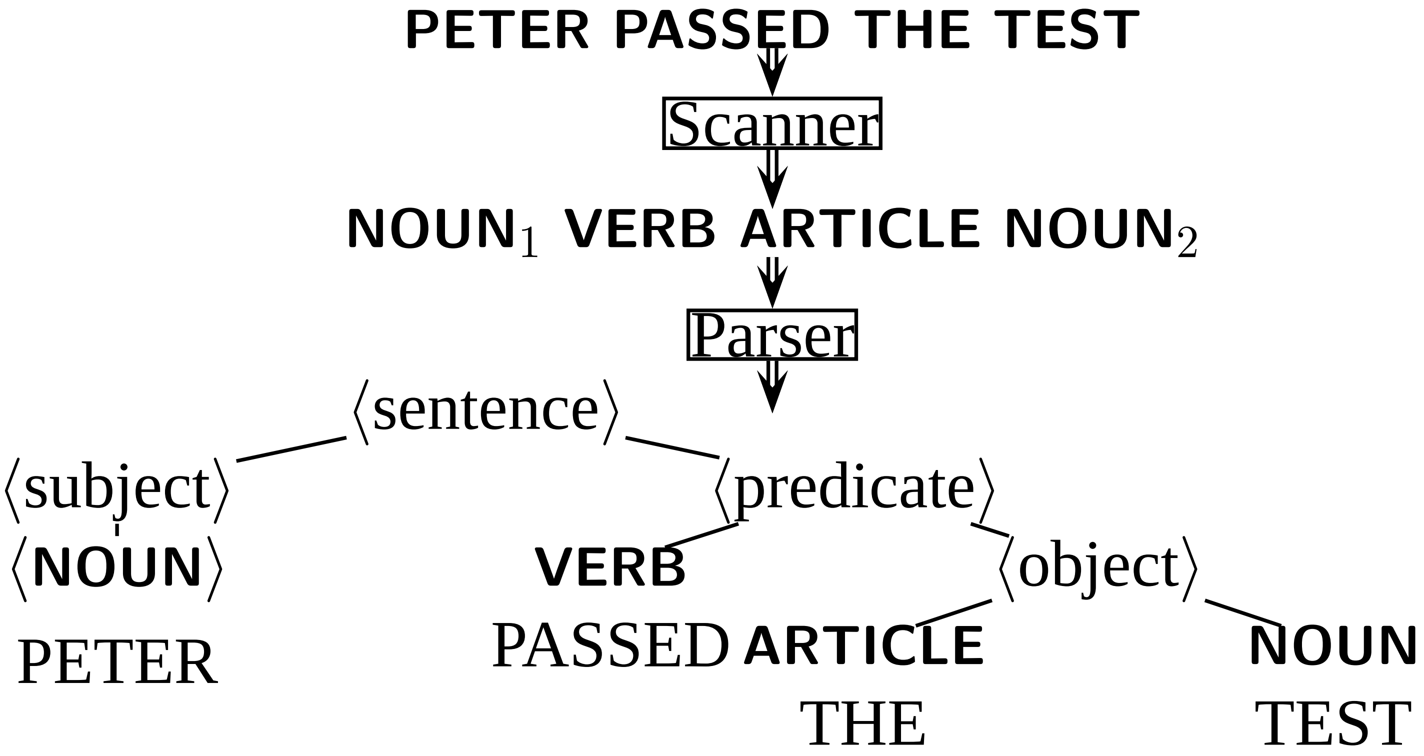
- Parsing: A process of constructing the leftmost or rightmost derivation of the sentence being analysed.
- **PETER PASSED THE TEST** $\stackrel{scanner}{\Longrightarrow}$ **NOUN**$_1$ **VERB ARTICLE NOUN**$_2$

$$
\begin{array}{lll}
\langle\text{sentence}\rangle & \Longrightarrow_{\text{lm}} \langle\text{subject}\rangle\ \langle\text{predicate}\rangle & \text{by P1} \\
 & \Longrightarrow_{\text{lm}} \textbf{NOUN}\ \langle\text{predicate}\rangle & \text{by P2} \\
 & \Longrightarrow_{\text{lm}} \textbf{NOUN VERB}\ \langle\text{object}\rangle & \text{by P4} \\
 & \Longrightarrow_{\text{lm}} \textbf{NOUN VERB ARTICLE NOUN} & \text{by P6}
\end{array}
$$

$$
\begin{array}{lll}
\langle\text{sentence}\rangle & \Longrightarrow_{\text{rm}} \langle\text{subject}\rangle\ \langle\text{predicate}\rangle & \text{by P1} \\
 & \Longrightarrow_{\text{rm}} \langle\text{subject}\rangle\ \textbf{VERB}\ \langle\text{object}\rangle & \text{by P4} \\
 & \Longrightarrow_{\text{rm}} \langle\text{subject}\rangle\ \textbf{VERB ARTICLE NOUN} & \text{by P6} \\
 & \Longrightarrow_{\text{rm}} \textbf{NOUN VERB ARTICLE NOUN} & \text{by P2}
\end{array}
$$

Parsing: Graphical Representation via Parse Trees

- Parsing: A process of constructing the parse tree for the sentence being analysed.



The Structure of Parse Trees

- The start symbol is always at the root of the tree.
- Nonterminals are always interior nodes.
- Terminals are always leaves in the tree.
- The sentence being analysed is the the leaves read from left to right.

Derivations v.s. Parse Trees

- The parsing of a sentence is to construct for the sentence
  - its leftmost or rightmost derivation, or
  - its parse tree
- The derivation and parse tree are two different views of the parsing of a sentence.
- The parse tree:
  - A graphical representation for a derivation.
  - The choice regarding to replacement order filtered out.

Summary So Far

- A language has two components: syntax and semantics.
  - Syntax: the form or structure of a program.
  - Semantics: the meaning of a program.
- A language's syntax is specified by a CFG.
- A CFG has four components.
- A BNF is a notation for writing a CFG.
- Parsing: discover a leftmost or rightmost derivation or build a parse tree
- Concepts
  - Sentential form
  - Sentence
  - Derivation: leftmost and rightmost
  - parse tree
  - Language and context-free language

Lecture 3: Context-Free Grammars, Languages and Parsing

1. The syntax and semantics of a programming language $\surd$
2. Specify a language's syntax: CFG, BNF and EBNF $\surd$
3. The parsing of a program in a language: $\surd$
   - Construct leftmost and rightmost derivations $\surd$
   - Construct parse trees $\surd$
4. The structure of a grammar:
   - How language constructs are defined
   - The precedence and associativity of operators
   - Ambiguity
5. The Chomsky hierarchy
6. The equivalence between regular grammars and finite automata

Extended Backus-Naur Form (EBNF)

- EBNF = BNF + regular expressions
- (*something* $)^*$ means that the stuff inside can be repeated zero or more times:
- (*something* $)^+$ means that the stuff inside can be repeated one or more times:
- (*something*)? means that the stuff inside is optional
- The parentheses omitted if $\langle$something$\rangle$ is a single symbol
- More compact and readable than the BNF
- Convenient for writing recursive-descent parsers
- The VC grammar is given in the form of EBNF

An ENBF Example from the VC Grammar: Kleene Closure

- A VC program is a sequence of zero or more functions
- The BNF productions:

$$
\begin{array}{lcl}
\textit{program} & \longrightarrow & \textit{decl-list} \\
\textit{decl-list} & \longrightarrow & \textit{decl-list func-decl} \\
 & \mid & \textit{decl-list var-decl} \\
 & \mid & \epsilon
\end{array}
$$

- The EBNF productions:

$$\textit{program} \longrightarrow (\textit{func-decl} \mid \textit{var-decl})^*$$

An ENBF Example: Positive Closure

- A program is a sequence of one or more functions
- The BNF productions:

$$
\begin{array}{lcl}
\textit{program} & \longrightarrow & \textit{decl-list} \\
\textit{decl-list} & \longrightarrow & \textit{decl-list func-decl} \\
 & | & \textit{decl-list var-decl} \\
 & | & \textit{func-decl} \\
 & | & \textit{var-decl}
\end{array}
$$

- The EBNF productions:

$$\textit{program} \longrightarrow (\textit{func-decl} \ | \ \textit{var-decl})^{+}$$

An ENBF Example from the VC Grammar: Optional Operator

- The if statement where the else-part is optional
- The BNF productions:

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textsf{\textbf{IF}} \text{ “(” } \mathit{expr} \text{ “)” } \mathit{stmt} \\
& | & \textsf{\textbf{IF}} \text{ “(” } \mathit{expr} \text{ “)” } \mathit{stmt} \ \textsf{\textbf{ELSE}} \ \mathit{stmt} \\
& | & \textsf{\textbf{other}}
\end{array}
$$

- The EBNF productions:

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \textsf{\textbf{IF}} \text{ “(” } \mathit{expr} \text{ “)” } \mathit{stmt} \ (\textsf{\textbf{ELSE}} \ \mathit{stmt})? \\
& | & \textsf{\textbf{other}}
\end{array}
$$

The Structure Of Grammars

- Top-Down Definition Of Language Constructs, as in VC:

```
program          -> ( func-decl | var-decl)*
func-decl        -> type identifier para-list compound-stmt
var-decl         -> type  init-declarator
type             -> void | boolean | int | float
compound-stmt    -> "{"  var-decl*  stmt*  "}"
stmt             -> compound-stmt
                  |  if-stmt
                     ...
                  |  expression-stmt
if-stmt          -> IF "(" expr ")" stmt ( ELSE stmt )?
expr-stmt        -> expr? ";"
expr             -> assignment-expr
assignment-expr  -> ...
```

- See the grammars for C (Kernighan and Ritchie's book) and Java (on-line)
- Bottom-Up Processing Of Language Constructs (Assignment 5). Roughly:
  - The deeper nodes in the parse tree processed first.
  - The deeper operators in the parse tree have higher precedence

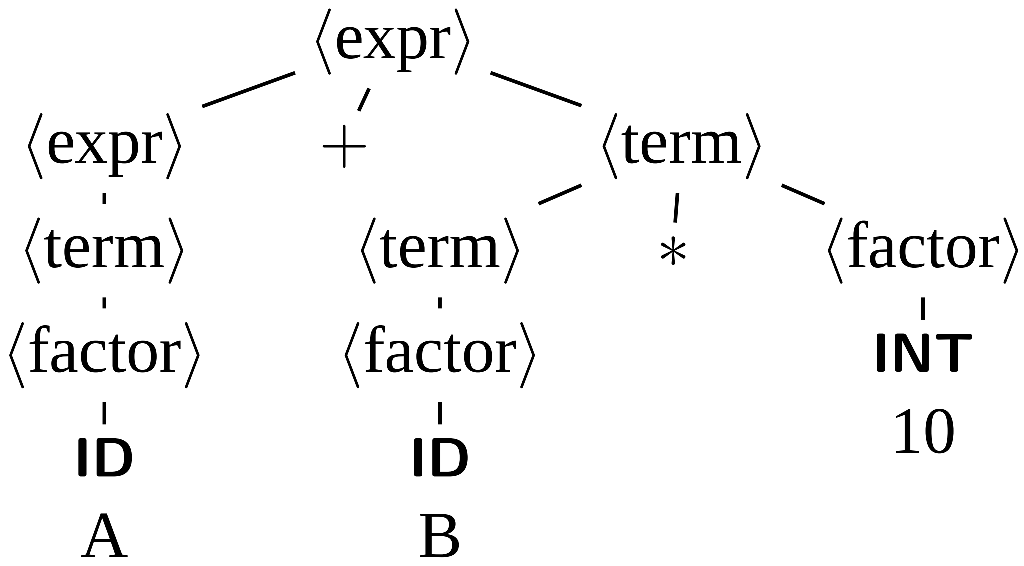
The Classic Expression Grammar

$$
\begin{array}{lrcl}
1 & \langle\text{expr}\rangle & \rightarrow & \langle\text{expr}\rangle + \langle\text{term}\rangle \\
2 & & | & \langle\text{expr}\rangle - \langle\text{term}\rangle \\
3 & & | & \langle\text{term}\rangle \\
4 & \langle\text{term}\rangle & \rightarrow & \langle\text{term}\rangle * \langle\text{factor}\rangle \\
5 & & | & \langle\text{term}\rangle \, / \, \langle\text{factor}\rangle \\
6 & & | & \langle\text{factor}\rangle \\
7 & \langle\text{factor}\rangle & \rightarrow & (\ \langle\text{expr}\rangle \ ) \\
8 & & | & \textbf{ID} \\
9 & & | & \textbf{INT} \text{ // Note: integer numbers not the type}
\end{array}
$$

- Left-Recursive Productions: $A \rightarrow A\alpha$
- Right-Recursive Productions: $A \rightarrow \alpha A$

Operator Precedence: A + B $*$ 10

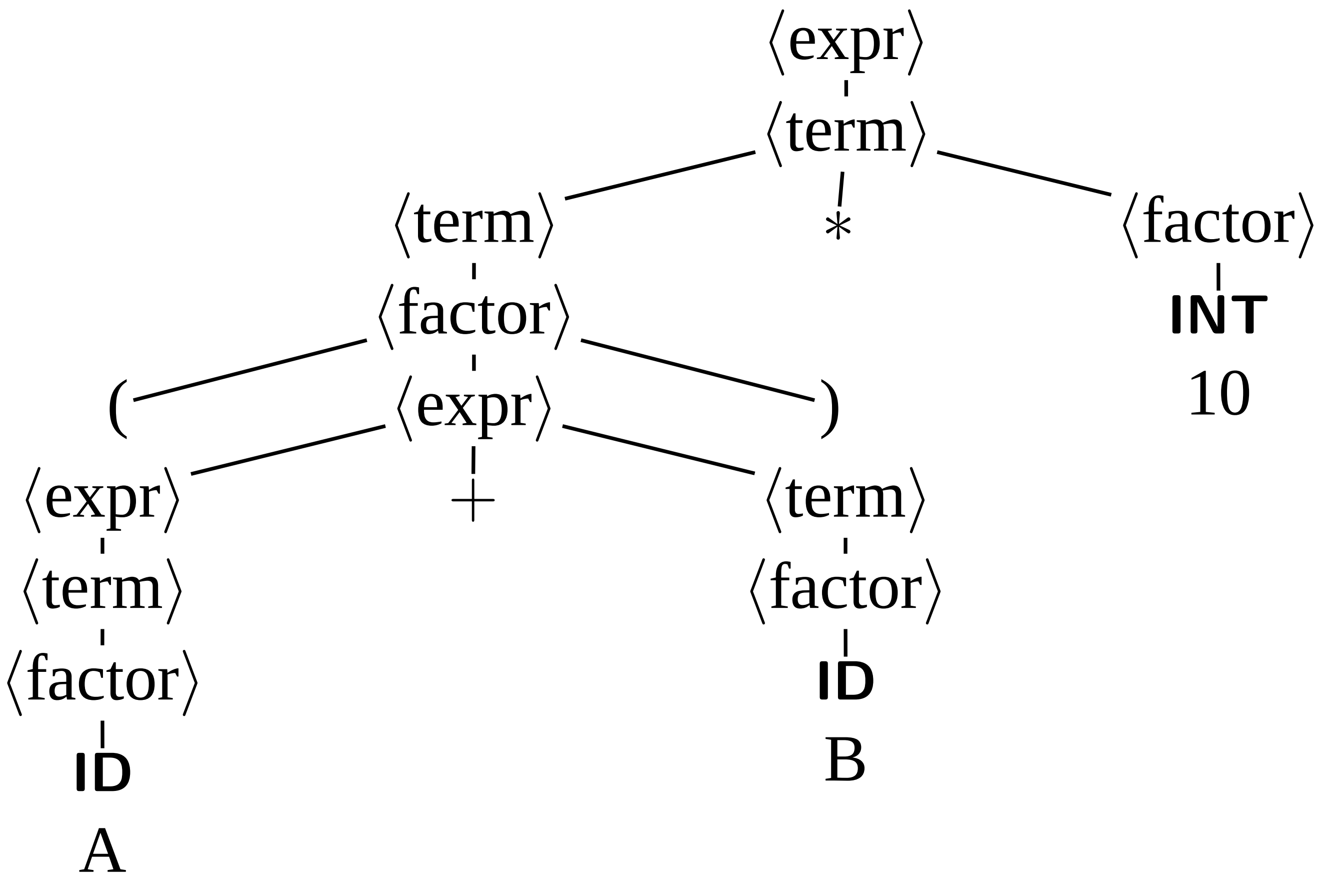
- Rules for binding operators to operands
- Higher precedence operators bind to their operands before lower precedence operators
- Higher precedence operators appear lower in the tree



- $A + B * 10$ evaluated as $A + (B * 10)$ as desired

Operator Precedence Changed by Parentheses: $(A + B) * 10$

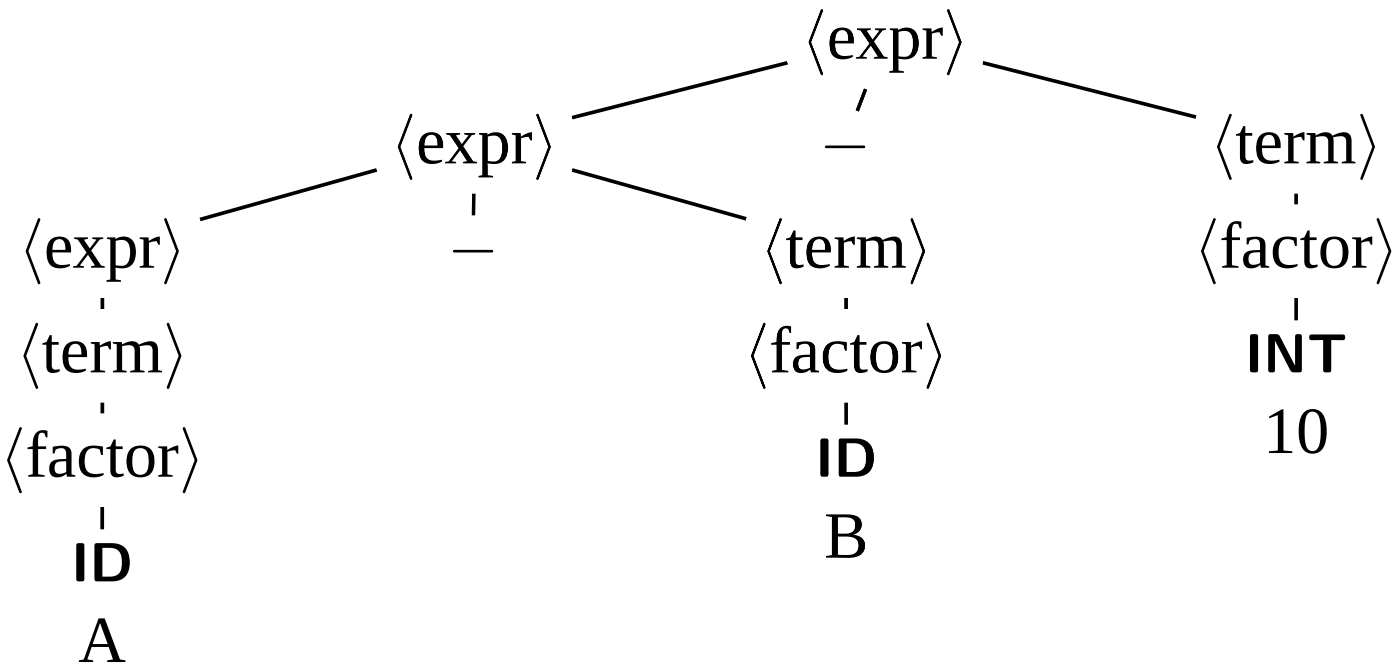
- $+$ appears lower than $*$ because of the use of (and):



- The addition will be evaluated first now

Operator Associativity: $A - B - 10$

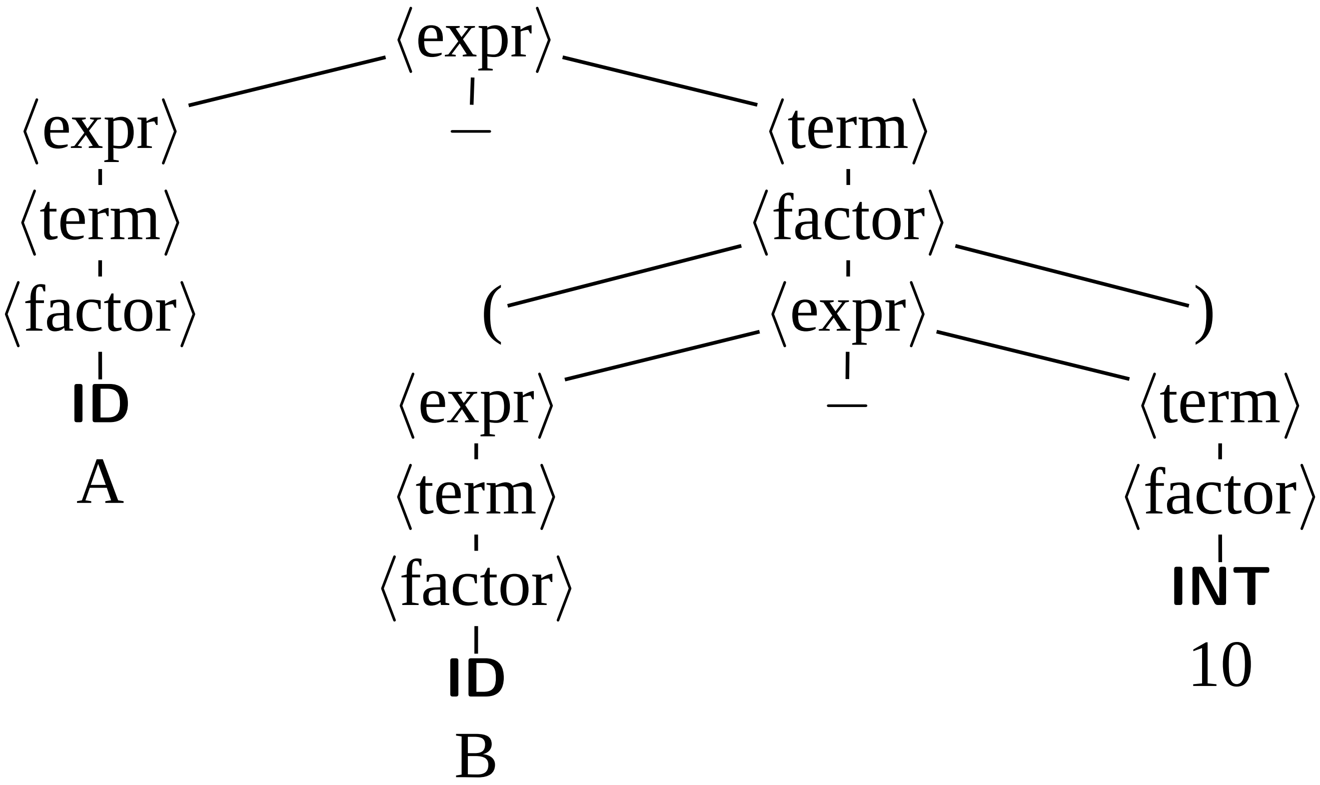
- Rules for grouping operators with equal precedence
- Given $\cdots - 1 - \ldots$, determines which $-$ takes the 1
- Left-recursive productions enforce left-associativity



- $A - B - 10$ evaluated as $(A - B) - C$ as desired

Operator Associativity Changed by Parentheses: $A-(B-10)$

- The 2nd $-$ appears lower than the 1st $-$ in the tree:



- The 2nd subtraction will be evaluated first

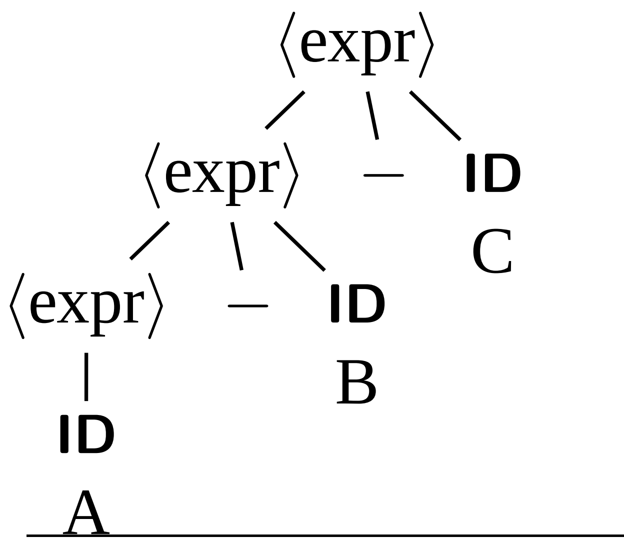
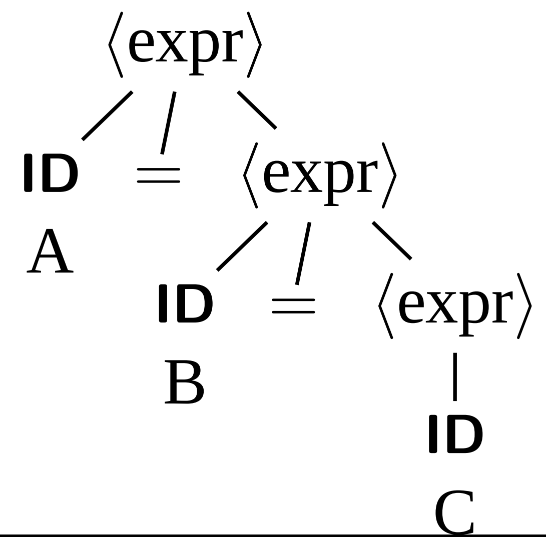
Operator Associativity: Summary

- A grammar consisting of left-recursive productions:

$$\langle\text{expr}\rangle \;\rightarrow\; \textbf{ID} \mid \langle\text{expr}\rangle - \textbf{ID}$$

- A grammar consisting of right-recursive productions:

$$\langle\text{expr}\rangle \;\rightarrow\; \textbf{ID} \mid \textbf{ID} = \langle\text{expr}\rangle$$

Parse tree of $A - B - C$	Parse tree of $A = B = C$
	

Precedence and Associativity Tables for Some Languages

```
C++: www-agrw.informatik.uni-kl.de/~jmayer/c-operator-precedence.html

C:   www.isthe.com/chongo/tech/comp/c-precedence.html

Java: http://www.uni-bonn.de/~manfear/javaoperators.php

Perl: http://www.ictp.trieste.it/texi/perl/perl_43.html#SEC34

Javascript: http://developer.mozilla.org/en/docs/
            Core_JavaScript_1.5_Reference:Operators:Operator_Precedence
```

Ambiguous Grammars

- A grammar is ambiguous if it permits
  - more than one parse tree for a sentence,
    or in other words,
  - more than one leftmost derivation or more than one rightmost derivation for a sentence.

- An ambiguous expression grammar:

$$\langle\text{expr}\rangle \longrightarrow \langle\text{expr}\rangle\ \langle\text{op}\rangle\ \langle\text{expr}\rangle \ \mid \textbf{ID} \mid \textbf{INT} \mid (\ \langle\text{expr}\rangle\ )$$
$$\langle\text{op}\rangle \longrightarrow + \mid - \mid * \mid /$$

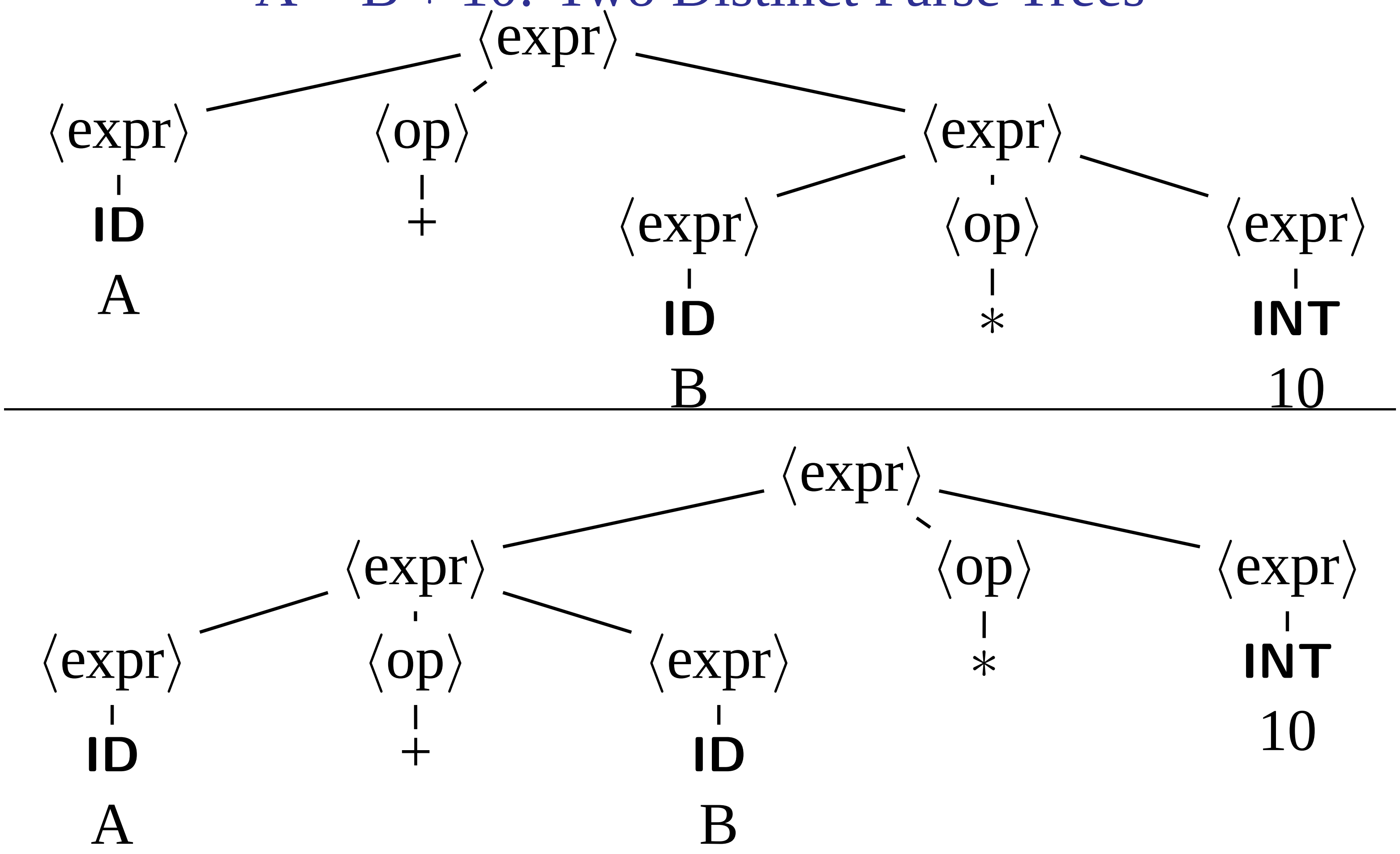
A + B ∗ 10: Two Distinct Leftmost Derivations

$$\begin{aligned}
\langle \text{expr} \rangle &\Longrightarrow_{lm} \langle \text{expr} \rangle\ \langle \text{op} \rangle\ \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID}\ \langle \text{op} \rangle\ \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID} + \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID} + \langle \text{expr} \rangle\ \langle \text{op} \rangle\ \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID} + \mathbf{ID}\ \langle \text{op} \rangle\ \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID} + \mathbf{ID} * \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID} + \mathbf{ID} * \mathbf{ID}
\end{aligned}$$

$$\begin{aligned}
\langle \text{expr} \rangle &\Longrightarrow_{lm} \langle \text{expr} \rangle\ \langle \text{op} \rangle\ \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \langle \text{expr} \rangle\ \langle \text{op} \rangle\ \langle \text{expr} \rangle\ \langle \text{op} \rangle\ \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID}\ \langle \text{op} \rangle\ \langle \text{expr} \rangle\ \langle \text{op} \rangle\ \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID} + \langle \text{expr} \rangle\ \langle \text{op} \rangle\ \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID} + \mathbf{ID}\ \langle \text{op} \rangle\ \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID} + \mathbf{ID} * \langle \text{expr} \rangle \\
&\Longrightarrow_{lm} \mathbf{ID} + \mathbf{ID} * \mathbf{ID}
\end{aligned}$$

Exercise: Find two distinct rightmost Derivations.

A + B $*$ 10: Two Distinct Parse Trees

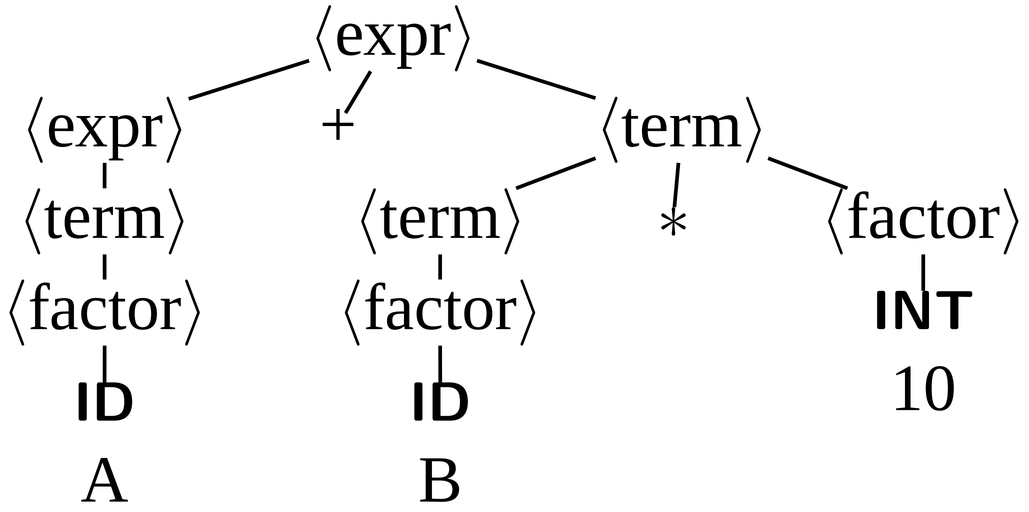


- The top tree means: $A + (B * 10)$
- The bottom tree means: $(A + B) * 10$

Coping With Ambiguous Grammars

- Method 1: Rewrite the grammar to make it unambiguous.

$$
\begin{aligned}
\langle \text{expr} \rangle &\rightarrow \langle \text{term} \rangle \mid \langle \text{expr} \rangle + \langle \text{term} \rangle \mid \langle \text{expr} \rangle - \langle \text{term} \rangle \\
\langle \text{term} \rangle &\rightarrow \langle \text{factor} \rangle \mid \langle \text{term} \rangle * \langle \text{factor} \rangle \mid \langle \text{term} \rangle / \langle \text{factor} \rangle \\
\langle \text{factor} \rangle &\rightarrow \textbf{ID} \mid \textbf{INT} \mid (\langle \text{expr} \rangle)
\end{aligned}
$$



- Un-ambiguous grammars preferred in practice

Coping With Ambiguous Grammars (Cont'd)

- Method 2: Use disambiguating rules to throw away undesirable parse trees, leaving only one tree for each sentence.
  - Rule 1: $*$ and / have higher precedence than + and $-$.
  - Rule 2: The operators of equal precedence associate to the left.
  - The desired parse tree: The one on the top of Slide 171.

Ambiguous Context-Free Languages

- There exists a CFL such that every grammar generating the language is ambiguous
- Such a language is called an ambiguous CFL
- The following language is inherently ambiguous:

$$L = \{a^n b^n c^m d^m \mid n \geqslant 1, m \geqslant 1\} \cup \{a^n b^m c^m d^n \mid n \geqslant 1, m \geqslant 1\}$$

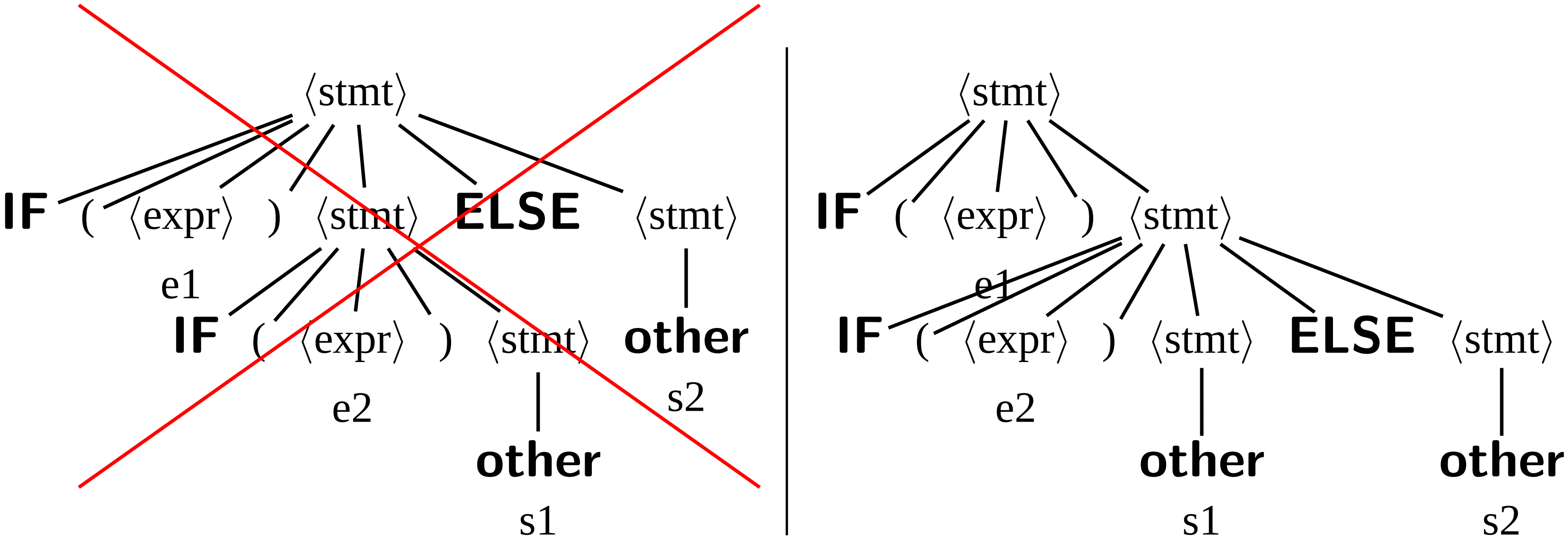
- Theoretical Result: There does not exist an algorithm that takes any CFL and tells us if it is ambiguous or not.

The "Dangling-Else" Grammar

- The grammar

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & \rightarrow & \textbf{IF} \text{ “(” } \langle \text{expr} \rangle \text{ “)” } \langle \text{stmt} \rangle \\
 & | & \textbf{IF} \text{ “(” } \langle \text{expr} \rangle \text{ “)” } \langle \text{stmt} \rangle \textbf{ ELSE } \langle \text{stmt} \rangle \\
 & | & \textbf{other}
\end{array}
$$

- Two parse trees for **IF (e1) if (e2) s1 else s2**



- Match **else** with the closest previous unmatched **then**
- A parser disambiguates the two cases easily using this rule

Formal Grammar

A grammar G is a quadruple (V_T, V_N, S, P), where

- V_T: a finite set of terminal symbols or tokens
- V_N: a finite set of nonterminal symbols ($V_T \cap V_N = \emptyset$)
- S: a unique start symbol ($S \in N$)
- P: a finite set of rules or productions of the form:

$$\alpha \to \beta \qquad\qquad (\alpha \neq \epsilon)$$

  - α is a string of one or more terminals and nonterminals
  - β is a string of zero or more terminals and nonterminals

Chomsky's Hierarchy

Depending on $\alpha \rightarrow \beta$, four types of grammars distinguished:

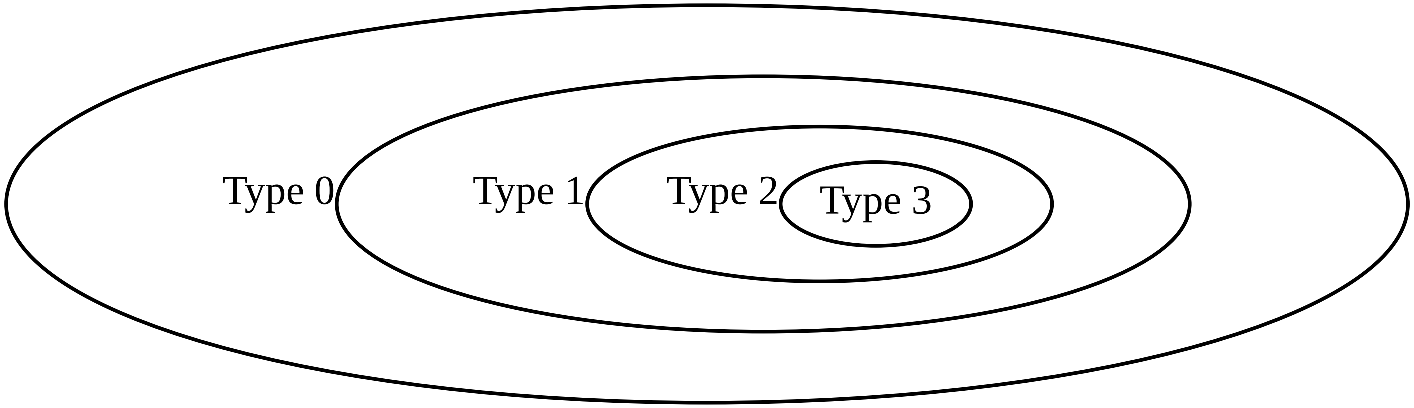
GRAMMAR	KNOWN AS	DEFINITION	MACHINE
Type 0	phrase-structure grammar	$\alpha \neq \epsilon$	Turing machine
Type 1	context-sensitive grammar CSGs	$\lvert\alpha\rvert \leq \lvert\beta\rvert$	linear bounded automaton
Type 2	context-free grammar CFGs	$A \rightarrow \alpha$	stack automaton
Type 3	right-linear grammar regular grammars	$A \rightarrow a \mid aB$	finite automaton

Note:

- a is a terminal.
- regular grammars can also be specified by left-linear grammars:

$$A \rightarrow a \mid Ba$$

Relationships between the Four Types of Languages



- Type k language is a proper subset of Type $k-1$ language.
- The existence of a Type 0 language is proved:

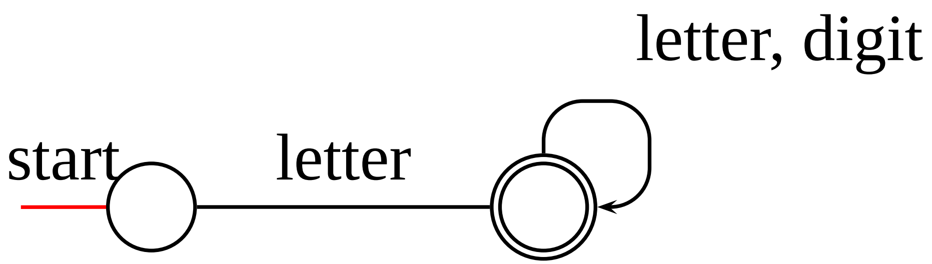
  page 228, J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

  But any language one can think of turns out to be context-sensitive.

Regular Expressions, Regular Grammars and Finite Automata

- All three are equivalent:
- Example:
  - Regular expression: $[A-Za-z\_][A-Za-z0-9\_]^*$
  - Regular grammar:

    identifier `->` *letter* | *identifier letter* | *identifier digit*
    letter `-> A|B| ... |Z|a|b| ... |z|_`
    digit `-> 0|1| ... |9`
  - DFA:



Limitations of Regular Grammars

- Cannot generate nested constructs
- The following language is not regular
$$L = \{a^n b^n \mid n \geqslant 1\}$$
- But L is context-free: $S \rightarrow \epsilon \mid aSb$
- Regular grammars (expressions) powerful enough for specifying tokens, which are not nested
- By replacing "a" and "b" with "(" and ")", the following
$$L = \{(^n)^n \mid n \geqslant 1\}$$
is not regular
- Formal proof: Pages 180 – 181 of Red / $4.2.7 of Purple
- Regular grammars (finite automata) cannot count

Limitations of CFGs

- CFLs only include a subset of all languages
- Examples of non-CFL constructs:
  - An abstraction of variable declaration before use:

$$L_1 = \{wcw \mid \text{w is in } (a|b)^*\}$$

    where the 1st w represents a declaration and the 2nd its use
  - a method called with the right number of arguments:

$$L_2 = \{a^n b^m c^n d^m \mid n \geqslant 1, m \geqslant 1\}$$

    where a^n and b^m represent formal parameter lists in two methods with n and m arguments, respectively, and c^n and d^m represent actual parameter lists in two calls to the two methods.
- Can count two but not three:

$$L_3 = \{a^n b^n c^n \mid n \geqslant 0\}$$

Limitations of CFGs (Cont'd)

- L_3 is not context-free
  - The language:

$$L_3 = \{a^n b^n c^n \mid n \geqslant 0\}$$

  - The grammar:

- A Context-Sensitive Grammar (CSG) for L_3:

CSG:	A derivation for $aabbcc$
$S \rightarrow aSBC$	$S \Longrightarrow aSBC$
$S \rightarrow abC$	$\Longrightarrow aabCBC$
$CB \rightarrow BC$	$\Longrightarrow aabBCC$
$bB \rightarrow bb$	$\Longrightarrow aabbCC$
$bC \rightarrow bc$	$\Longrightarrow aabbcC$
$cC \rightarrow cc$	$\Longrightarrow aabbcc$

Why CFGs in Parser Construction?

- Types 0 and 1 are less understood, no simple ways of constructing parsers for them, and parsers for these languages are slow
- Type 3 cannot define recursive language constructs
- Type 2 – context-free grammars (CFGs):
  - Easily related to the structure of the language; productions give us a good idea of what to expect in the language
  - Close relationships between the productions and the corresponding computations, which is the basis of syntax-directed translation
  - Efficient parsers can be built automatically from CFGs

Lecture 3: Context-Free Grammars, Languages and Parsing

1. The syntax and semantics of a programming language $\surd$
2. Specify a language's syntax: CFG, BNF and EBNF $\surd$
3. The parsing of a program in a language: $\surd$
   - Construct leftmost and rightmost derivations $\surd$
   - Construct parse trees $\surd$
4. The structure of a grammar: $\surd$
   - How language constructs are defined $\surd$
   - The precedence and associativity of operators $\surd$
   - Ambiguity $\surd$
5. The Chomsky hierarchy $\surd$
6. The equivalence between regular grammars and finite automata

Equivalence between Regular Grammars and FAs

- Lecture 2: the equivalence among REs and FAs
- Slides 186 – 191: NFAs $\equiv$ Regular Grammars

Converting NFAs to Right-Linear Grammars

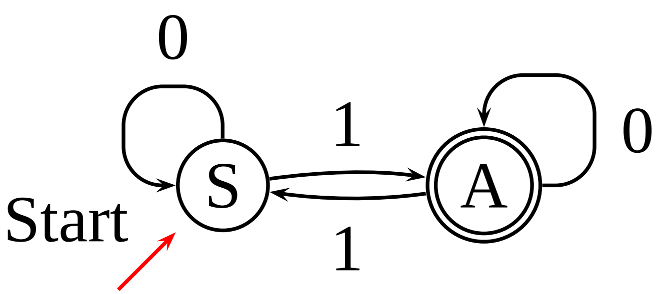
- The alphabet: the same
- For each state in the NFA, create a nonterminal with the same name.
- The start state will be the start symbol
- Then

TRANSITION		PRODUCTION
$A \xrightarrow{a} B$	$\Longrightarrow$	$A \rightarrow aB$
A	$\Longrightarrow$	$A \rightarrow \epsilon$

where $a \in \sum$ or $a = \epsilon$

Example 1

- The DFA:

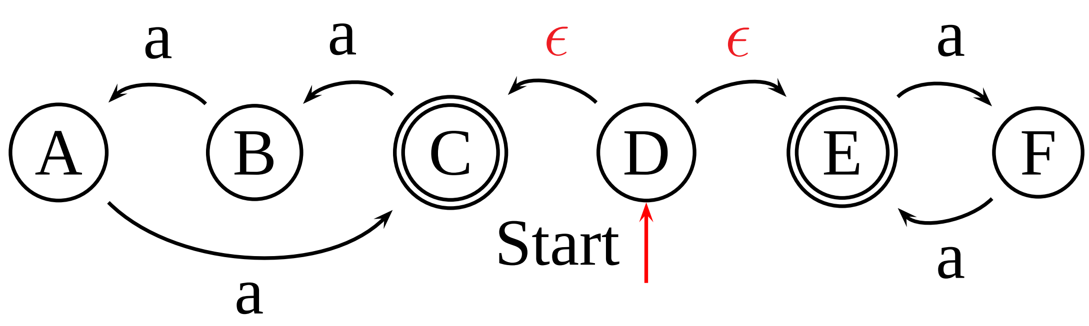


- The grammar:

$$
\begin{array}{rcl}
S & \longrightarrow & 0S \\
S & \longrightarrow & 1A \\
A & \longrightarrow & 0A \\
A & \longrightarrow & 1S \\
A & \longrightarrow & \epsilon
\end{array}
$$

Example 2

- The NFA:



- The grammar:

$$
\begin{array}{lcl@{\qquad}lcl}
D & \longrightarrow & C & A & \longrightarrow & aC \\
D & \longrightarrow & E & E & \longrightarrow & aF \\
C & \longrightarrow & aB & E & \longrightarrow & \epsilon \\
C & \longrightarrow & \epsilon & F & \longrightarrow & aE \\
B & \longrightarrow & aA & & &
\end{array}
$$

Converting Right-Linear Grammars to NFAs

- The alphabet: the same
- For each nonterminal, create a state in the NFA with the same name. The start symbol will be the start state
- Add one new state and make it the only final state $\mathcal{F}$
- Then

PRODUCTION		TRANSITION	
$A \rightarrow aB$	$\Longrightarrow$	$A \xrightarrow{a} B$	$T(A, a) = B$
$A \rightarrow a$	$\Longrightarrow$	$A \xrightarrow{a} \mathcal{F}$	$T(A, a) = \mathcal{F}$

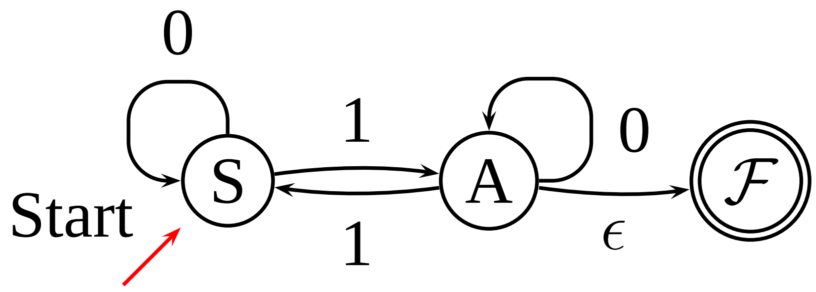
where $a \in \sum$ or $a = \epsilon$

Example 1

- The grammar:

$$
\begin{aligned}
S &\rightarrow 0S \\
S &\rightarrow 1A \\
A &\rightarrow 0A \\
A &\rightarrow 1S \\
A &\rightarrow \epsilon
\end{aligned}
$$

- The NFA:



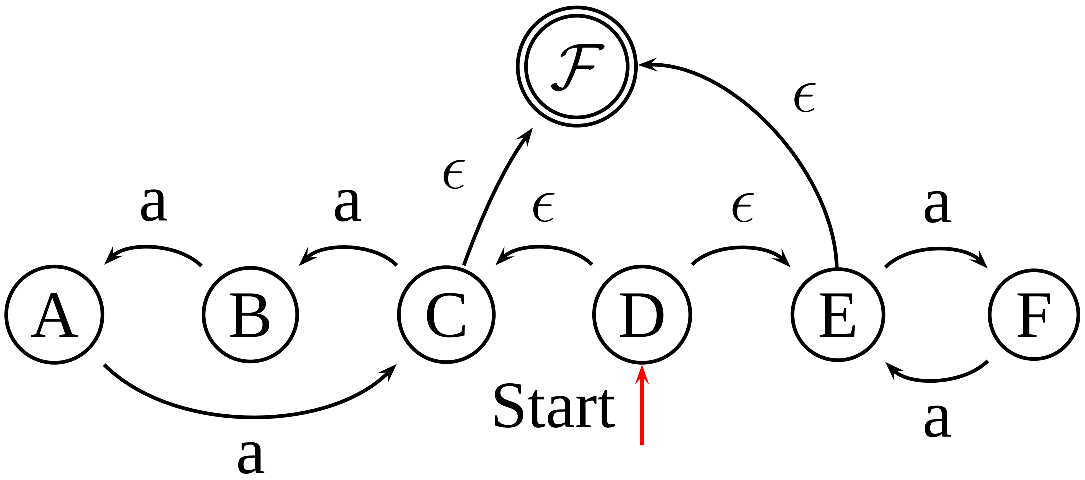
- This NFA accepts the same language as the one in Slide 187

Example 2

- The grammar:

$$
\begin{array}{lllllll}
D & \longrightarrow & C & \quad & A & \longrightarrow & aC \\
D & \longrightarrow & E & & E & \longrightarrow & aF \\
C & \longrightarrow & aB & & E & \longrightarrow & \epsilon \\
C & \longrightarrow & \epsilon & & F & \longrightarrow & aE \\
B & \longrightarrow & aA & & & &
\end{array}
$$

- The NFA:



- This NFA accepts the same language as the NFA in Slide 188

Summary of Lecture 3

1. Understand the difference between the syntax and semantics of a programming language
2. Be able to write a CFG for simple languages
3. Understand the syntax defined by a CFG, BNF and EBNF
4. Be able to parse simple sentences manually by
   - Constructing leftmost and rightmost derivations
   - Constructing parse trees
5. Be able to infer the operator precedence and associativity from a grammar
6. Be able to convert between NFAs and regular grammars