Today's Topics

Today

- Lexical & Syntactic Specification of Languages (CISC 223 review)
- Lexical structure - regular expressions, FSA, transition diagrams and matrices, regular languages, relation to S/SL
- Syntactic structure - context-free grammars, PDA, context-free languages
- Grammars - BNF, terminal vs nonterminal symbols, relation to S/SL, sentential forms, derivations, parse trees, ambiguity, precedence and associativity

Lexical & Syntactic Specification of Languages

Specifying Languages

- The syntax of programming languages is normally specified using two separate levels of structure

Lexical Structure

- Describes division of sequences of characters into tokens
- Specified using *regular expressions* and *finite state automata*

Syntactic Structure

- Describes the combination of sequences of tokens into the grammatical forms of the language (parsing)
- Specified using *context free grammars* and *push down automata*

Formal Language Terminology

Terminology

- An *alphabet* or *vocabulary* is any finite set of symbols
- Examples:

binary digits	{ 0, 1 }
decimal digits	{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
letters	{ a, b, c, ..., z, A, B, C, ..., Z }
special chars	{ +, -, *, /, <, >, ... }
keyboard chars	decimal digits ∪ letters ∪ special chars
Pascal keywords	{ **begin**, **end**, ... }

Formal Language Terminology

Terminology

- A *string* is any finite sequence of symbols in the alphabet

  Example:

  strings of binary digits: 0,1,101, 00100100, 1010101, ...

- The *empty string* (ε) – a string of zero length (containing no symbols at all)

- A *language* is a subset of the strings of a particular alphabet

  Examples:

  strings of binary digits representing odd numbers

  strings of binary digits representing prime numbers

Specification of Lexical Structure

Regular Expressions

- A *regular expression* over an alphabet **A** consists of members of **A** combined using the following operators:

*	repetition
|	alternation
•	sequence (often *implied*, like multiplication ab = a•b)
( )	grouping

- Examples:

  If **A** is the set of keyboard characters, and **l** is the subset of **A** which are letters, and **d** is the subset of **A** which are digits

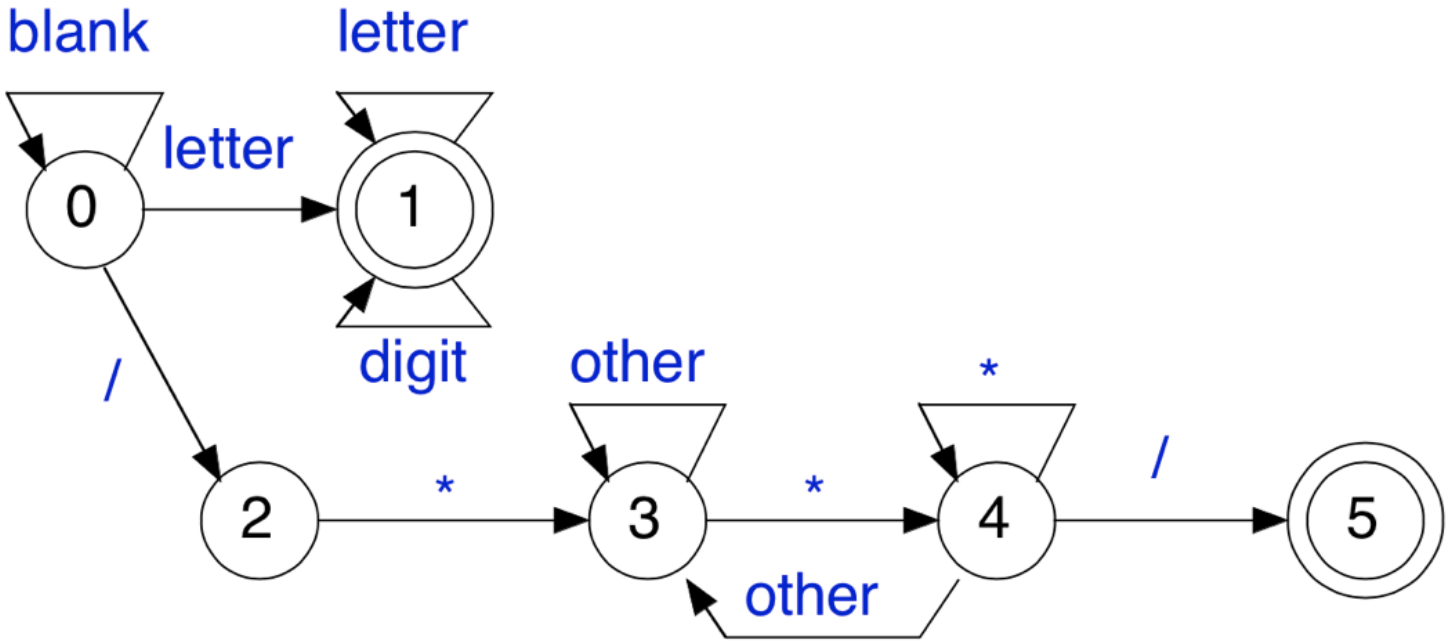
  Identifiers: `l•(l|d)* = l(l|d)*`

  Unsigned integers: $d \bullet d^* = dd^* = d^+$

Specification of Lexical Structure

Finite State Automata

- A *finite state automaton* (FSA) is:
  - a finite set of *states*
  - a finite set of *state transitions* (State x Symbol -> State)
  - a particular state designated as the *start state*
  - one or more states designated as *final states*



Specification of Lexical Structure

Transition Matrices

- A *transition matrix* represents a FSA diagram
- For each input symbol and state, gives either next state to go to, accept (Y) or reject (N)

	blank	letter	digit	/	*	other	ε
0	0	1	N	2	N	N	N
1	N	1	1	N	N	N	Y
2	N	N	N	N	3	N	N
3	3	3	3	3	4	3	N
4	3	3	3	5	4	3	N
5	N	N	N	N	N	N	Y

- Any string of input symbols which take a FSA from the start state to a final state is said to be *accepted* or *recognized* by the FSA

Regular Languages

Regular Languages

- The set of all languages that can be accepted by FSA and described by regular expressions is the same
- These are called the *regular languages*

Regular Languages in Compiler Technology

- Regular languages are used to describe the scanning part of programming language processors
- The elements of the regular language recognized by the scanner are called tokens in compiler terminology

Regular Languages in S/SL

- An *SL program* that has *no recursive rules* recognizes a regular language (prove!)

```
Scan:
   {[
       | blank:
       | letter:
          {[
              | letter, digit:
              | *:
                 >>
          ]}
                                    | '/':
                                       '*'
                                       {[
                                           | '*':
                                              [
                                                 |'/':
                                                    >>
                                                 |*:
                                                    ?
                                              ]
                                           | *:
                                              ?
                                       ]}
   ]};
```

Limitations of Regular Languages

Limitations

- Regular languages work well for Scanners
- Most Scanners are built using FSA, even automatic generators for scanners are based on FSAs (e.g. *lex, flex, regexp, Ragel*)
- But - FSA cannot recognize languages involving nesting, indefinite counting, matching or balancing symbols (hence cannot handle **begin**...**end**, parentheses, etc.)
- Example:

  Even the simple language consisting of all strings of the form

  $(^n\ x\ )^n$ (that is, balanced parentheses) is not a regular language

```
(((((((((((((((x)))))))))))))))
```

Specification of Syntactic Structure

Syntactic Structure

- The *syntactic structure* of a language deals with the combination of sequences of symbols (tokens) into language constructs such as expressions, statements and so on
- Syntactic structure is specified using *context free grammars* (CFG) or *Push Down Automata* (PDA)
- The formal languages that can be described by CFG are called the *context free languages* (CFL) and include the syntax of all modern programming languages

Specification of Syntactic Structure

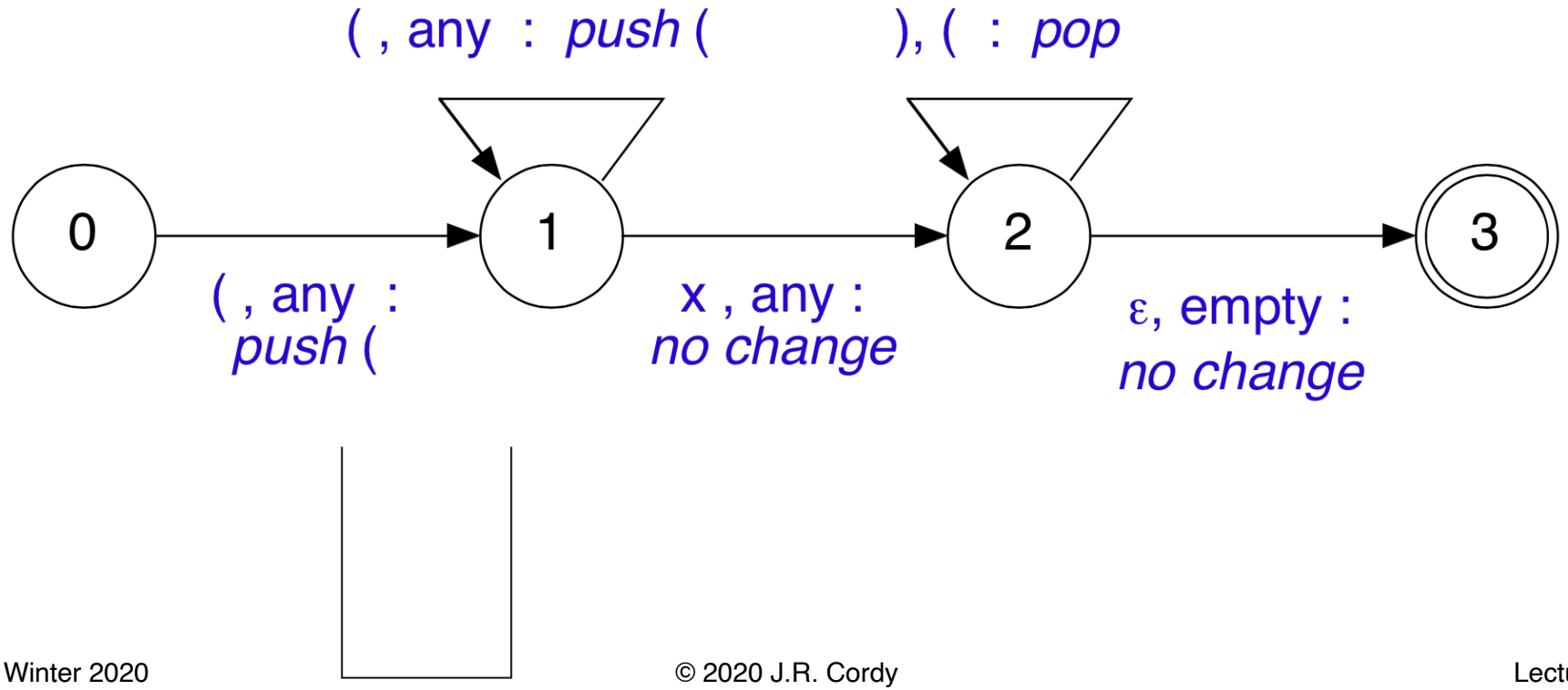
Push Down Automata

- A *push down automaton* (PDA) is a formal machine consisting of a *finite state control* (FSC) and a single *push down stack*
- The FSC is similar to the finite state automata (FSA) used for recognizing lexical structure
- But the state transitions of the FSC can now additionally depend on the *symbol on the top of the push down stack* as well as on the next input symbol
- Additionally, at every state transition the FSC may optionally modify the stack by *pushing or popping a symbol* from it

Specification of Syntactic Structure

Example PDA for $(^n\ x\ )^n$

- State transitions are determined by both the next input symbol and the top symbol in the stack, for example "),(" means next input ")", top symbol in stack "("
- At each transition a symbol may optionally be pushed or popped from the stack, for example ": push (" means push the symbol "(" onto the stack



Specification of Syntactic Structure

Backus-Nauer Form (BNF)

- A *context free grammar* (CFG) in *Backus-Nauer form* (BNF) is a 4-tuple $< T, N, S, P >$ where:

  T is a finite set of *terminal symbols* (i.e. tokens)

  N is a finite set of *non-terminal symbols* (i.e., constructs)

  $N \cap T = \varnothing$

  S is a distinguished member of N (the *start symbol*)

  P is a set of *rules* or *productions* of the form

  $$A \rightarrow \gamma$$

  where :

  $A \in N$ (i.e. A is a non-terminal)

  $\gamma \in (N \cup T)^*$ (i.e. a possibly empty finite string of terminal and non-terminal symbols)

Derivation Step

Derivation in One Step

- Definition: $x \Rightarrow y$ x *derives* y *in one step*

  when :

  there is a nonterminal $A \in N$, and $\alpha, \beta, \gamma \in (N \cup T)^*$

  such that :

  $x = \alpha A \gamma$

  $y = \alpha \beta \gamma$

  and there is a production $\in P$

  $A \rightarrow \beta$

Example BNF Context-Free Grammar (CFG)

$T = \{ 0, 1, 2, ..., 9, +, -, *, /, (,) \}$

$N = \{ E, OP \}$

$S = E$

$P = \{$
$E \rightarrow E\ OP\ E,$
$E \rightarrow (\ E\ ),$
$E \rightarrow -\ E,$
$E \rightarrow 0 \mid 1 \mid 2 \mid ... \mid 9,$
$OP \rightarrow + \mid - \mid * \mid /$
$\}$

Note : $A \rightarrow B \mid C$ *is shorthand for:*
$A \rightarrow B,\ A \rightarrow C$

$E * E \Rightarrow E\ OP\ E * E$ since $E \rightarrow E\ OP\ E \in P$

$E * E \Rightarrow 3 * E$ since $E \rightarrow 3 \in P$

$E\ OP\ E \Rightarrow E * E$ since $OP \rightarrow * \in P$

Derivations

Derivation

- Definition: $x \Rightarrow^* y$ x *derives* y

  when :

  $x \Rightarrow z_1$

  $z_1 \Rightarrow z_2$ in zero or more steps - that is,

  ... there is a chain of one step derivations

  $z_n \Rightarrow y$ that lead from x to y

- Note that *zero or more steps* means that $x \Rightarrow^* x$ for all x

- Example: $E \Rightarrow^* 1+2*3$ since

$$
\begin{aligned}
E &\Rightarrow E\ OP\ E \\
&\Rightarrow E * E \\
&\Rightarrow E\ OP\ E * E \\
&\Rightarrow E + E * E \\
&\Rightarrow E + E * 3 \\
&\Rightarrow E + 2 * 3 \\
&\Rightarrow 1 + 2 * 3
\end{aligned}
$$

Sentential Forms

Context-Free Languages

- Given the CFG $G = \langle T,N,S,P \rangle$, the *language* generated by G is:

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

  (that is, the set of all sequences of terminal symbols that can be derived from the start symbol)

- A sequence $s \in (N \cup T)^*$ is a *sentential form* of G if $S \Rightarrow^* s$

- A sequence $s \in T^*$ is a *sentence* of G if $S \Rightarrow^* s$

  (that is, a sentence is a sentential form consisting of only terminal symbols)

Derivation of a Sentence

Deriving from the Start Symbol

- A *derivation* of a sentence $s \in T^*$ is the sequence of sentential forms leading from the start symbol S to the sentence s
- Example: The start symbol of our example grammar G is E, and

$$
\begin{aligned}
E &\Rightarrow E\ OP\ E \\
&\Rightarrow E * E \\
&\Rightarrow E * (\ E\ ) \\
&\Rightarrow E * (\ E\ OP\ E\ ) \\
&\Rightarrow E * (\ E + E\ ) \\
&\Rightarrow E * (\ E + 9\ ) \\
&\Rightarrow 3 * (\ E + 9\ ) \\
&\Rightarrow 3 * (\ 8 + 9\ )
\end{aligned}
$$

So 3 * (8 + 9) is a sentence of L(G), and the above is a derivation of it

Also, E * (E OP E) and *all* of the other forms above are sentential forms of L(G)

Leftmost Derivation of a Sentence

Leftmost Derivation

- A *leftmost derivation* of a sentence is one where only the leftmost nonterminal is replaced at each step
- Example:

$$
\begin{aligned}
E &\Rightarrow E\ OP\ E \\
&\Rightarrow 3\ OP\ E \\
&\Rightarrow 3\ *\ E \\
&\Rightarrow 3\ *\ (\ E\ ) \\
&\Rightarrow 3\ *\ (\ E\ OP\ E\ ) \\
&\Rightarrow 3\ *\ (\ 8\ OP\ E\ ) \\
&\Rightarrow 3\ *\ (\ 8\ +\ E\ ) \\
&\Rightarrow 3\ *\ (\ 8\ +\ 9\ )
\end{aligned}
$$

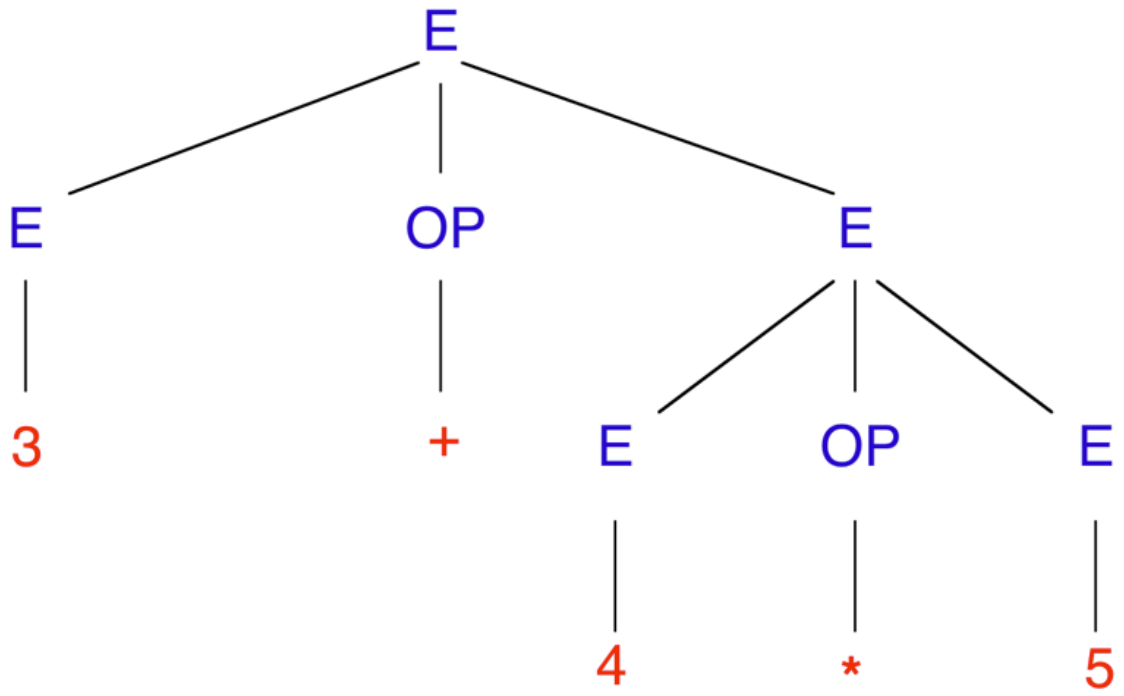
$$
\begin{aligned}
E &\Rightarrow E\ OP\ E \\
&\Rightarrow E\ *\ E \\
&\Rightarrow E\ *\ (\ E\ ) \\
&\Rightarrow E\ *\ (\ E\ OP\ E\ ) \\
&\Rightarrow E\ *\ (\ E\ +\ E\ ) \\
&\Rightarrow E\ *\ (\ E\ +\ 9\ ) \\
&\Rightarrow 3\ *\ (\ E\ +\ 9\ ) \\
&\Rightarrow 3\ *\ (\ 8\ +\ 9\ )
\end{aligned}
$$

- Both of these and many other derivations derive this same sentence, but only the one on the left above is a leftmost derivation of it
- Notice that the number of steps is the same, but the order of nonterminal replacements differs

Parse Trees

Syntactic Structure

- A *parse tree* is a graphical representation of the derivation of a sentence
- For each parse tree there is a *unique* corresponding leftmost derivation
- Example: 3 + 4 * 5

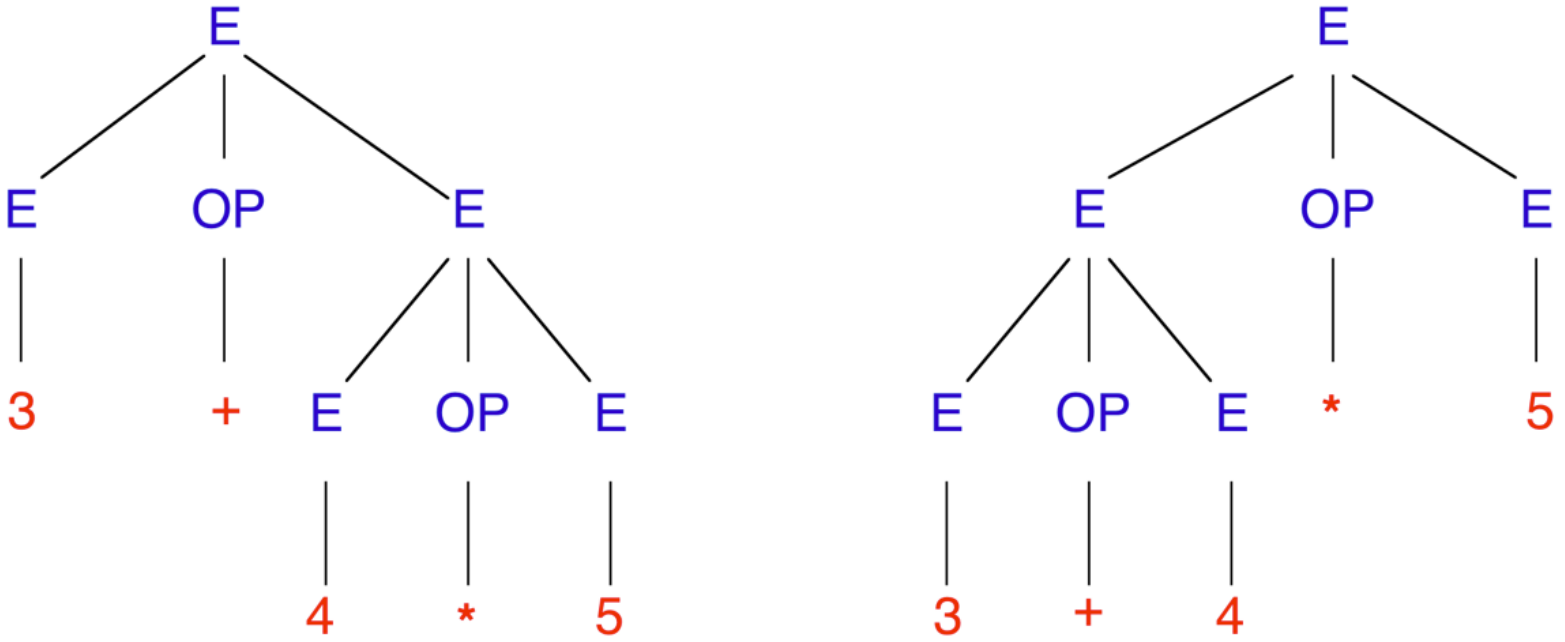


- This is interesting because parsing consists of discovering the parse tree of an input, and we would like parsing to be deterministic

Ambiguity

Which Parse Tree?

- An *ambiguous* grammar is one in which there is more than one leftmost derivation (i.e. more than one parse tree) for some sentence of the language
- Our example grammar G is ambiguous because it allows more than one parse tree / leftmost derivation of the sentence 3 + 4 * 5
- Ambiguity also causes problems for deterministic parsing - which parse tree is the right one?



Summary

Lexical and Syntactic Structure of Languages

- Lexical structure - regular expressions, FSA, transition diagrams and matrices, regular languages, relation to S/SL
- Syntactic structure - context-free grammars, PDA, context-free languages
- Grammars - BNF, terminal vs nonterminal symbols, sentential forms, derivations, parse trees, ambiguity

Next

- More on context free grammars - resolving ambiguity using precedence and associativity