Specifying Syntax

COMP360

“The most important thing in the programming language is the name. A language will not succeed without a good name. I have recently invented a very good name and now I am looking for a suitable language.”

Donald Knuth

Homework

- An assignment has been posted on Blackboard
- You have to write:
  - four regular expressions
  - DFAs to implement two of the regular expressions
  - a program to implement one of the DFA
- Due by noon on Friday, March 4, 2016

Pushdown Automaton

- A pushdown automaton (PDA) is an abstract model machine similar to the FSA
- It has a finite set of states. However, in addition, it has a pushdown stack. Moves of the PDA are as follows:

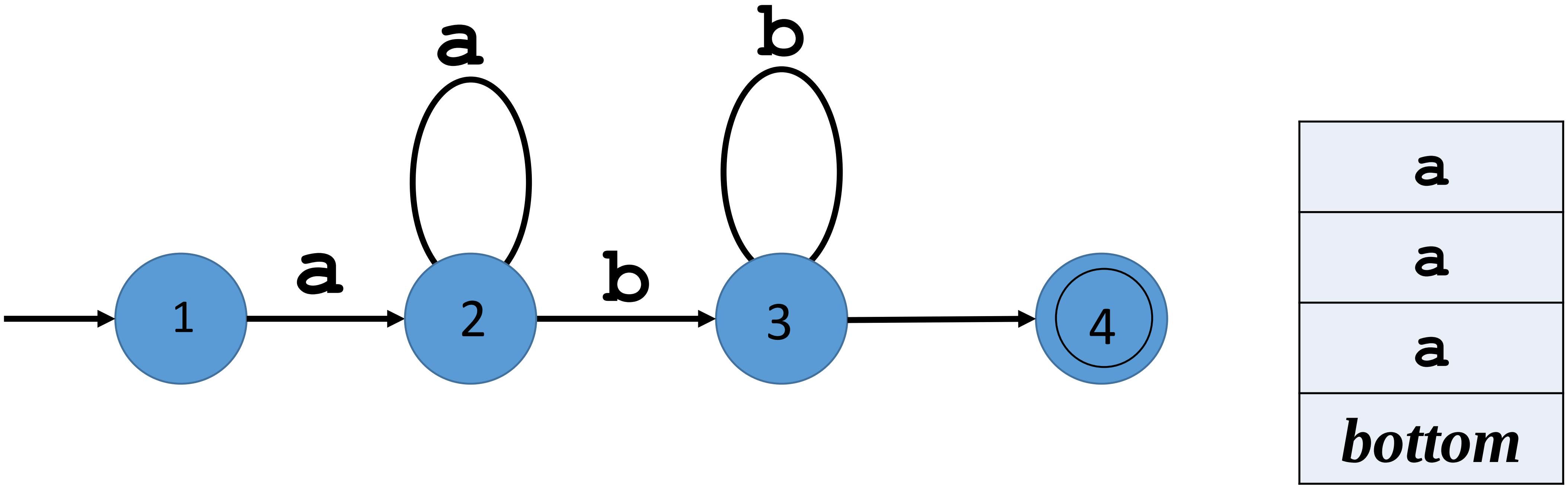
1. An input symbol is read and the top symbol on the stack is checked
2. Based on both inputs, the machine enters a new state and writes zero or more symbols onto the pushdown stack
3. Acceptance of a string occurs if the stack is ever empty. (Alternatively, acceptance can be if the PDA is in a final state. Both models can be shown to be equivalent)

Power of PDAs

- PDAs are more powerful than FSAs.
- $a^n b^n$, which cannot be recognized by an FSA, can easily be recognized by the PDA
- Stack all a symbols and, for each b, pop an a off the stack.
- If the end of input is reached at the same time that the stack becomes empty, the string is accepted

- It is less clear that the languages accepted by PDAs are equivalent to the context-free languages

Push Down Automata



$a^n b^n$ PDA

- start in state 1 with an empty stack

state	input	stack	new state	action
1	a	Empty	2	push a
2	a	a	2	push a
2	b	a	3	pop a
3	b	a	3	pop a
3	EOF	Empty	4	accept

NDPDAs are different from DPDAs

- Unlike FSA, a nondeterministic PDA is more powerful than a deterministic PDA
- Consider the set of palindromes, strings reading the same forward and backward, generated by the grammar

$$S \rightarrow 0S0 \mid 1S1 \mid 2$$

- We can recognize such strings by a deterministic PDA:
  1. Stack all 0s and 1s as read
  2. Enter a new state upon reading a 2
  3. Compare each new input to the top of stack, and pop stack

Nondeterministic PDA

- However, consider the following set of palindromes:

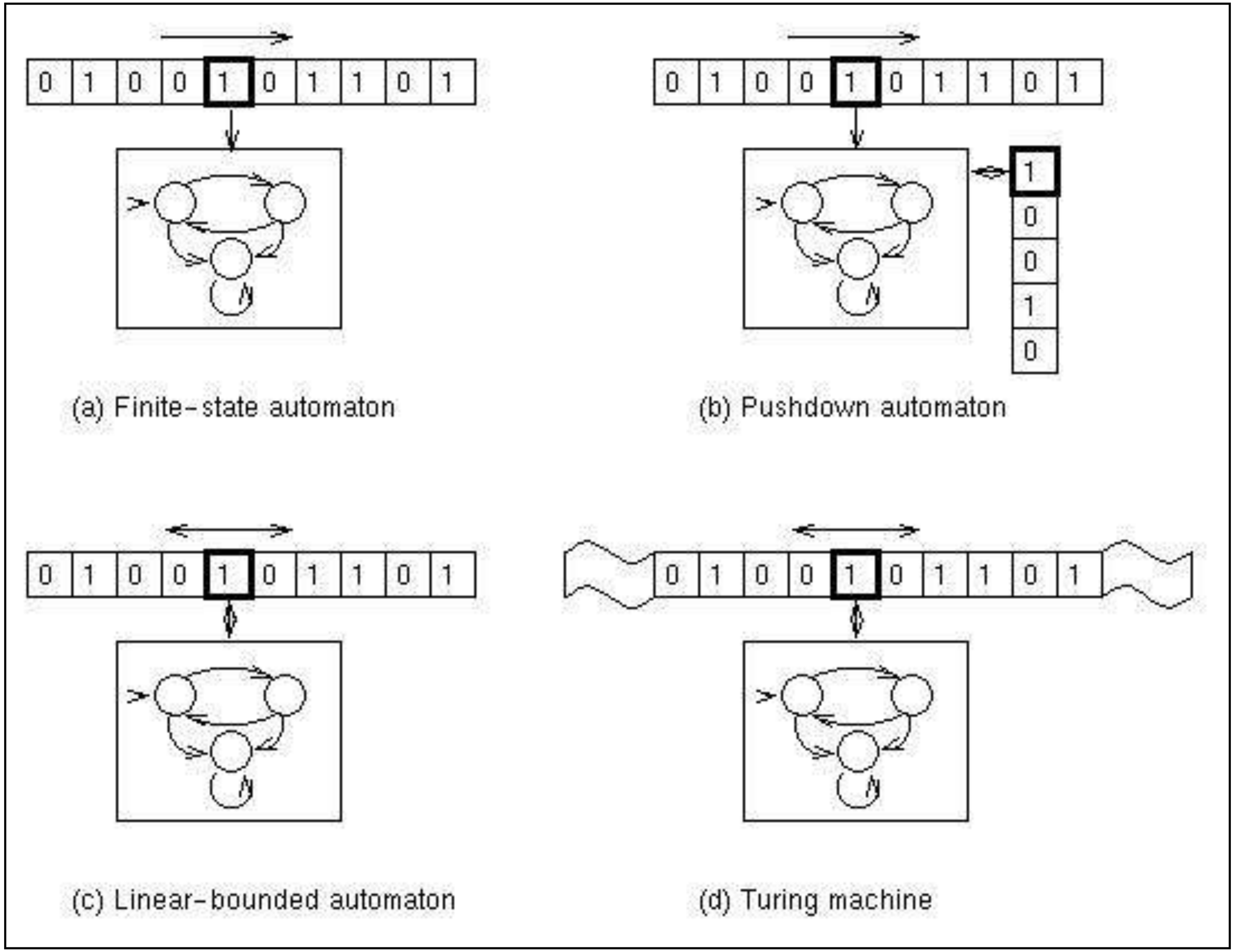
$$S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1$$

- In this case, we never know where the middle of the string is. To recognize these palindromes, the automaton must guess where the middle of the string is (i.e., is nondeterministic)

Language-machine equivalence

- Regular languages = FSA = NDFSA
- Context free languages = NDPDA. It can be shown that NDPDA not the same as DPDA
- For context sensitive languages, we have Linear Bounded Automata (LBA)
- For unrestricted languages we have Turing machines (TM)
- Unrestricted languages = TM = NDTM
- Context sensitive languages = NDLBA.
- It is still unknown if NDLBA=DLBA

Grammar-machine equivalence



Programming Language design and Implementation -4th Edition Copyright© Prentice Hall, 2000

BNF

- Backus-Naur Form (BNF) is used to express a context-free grammar
- John Backus and Peter Naur developed a notational system for describing context-free grammars
- First used to describe the syntax of Algol60

Terminals and Non-Terminals

- Terminal values are symbols that actually appear in strings of the language
- Terminals are often tokens from the scanner
- BNFs introduce variables to describe portions of the grammar
- These variables do not appear in strings of the language

Format of a BNF

- The general format of a BNF is

non-terminal → terminals and non-terminals
| terminals and non-terminals

- Sometimes ::= is used instead of →
- Some books like to put non-terminals inside brackets such as <expression>

Example BNF

- Consider the syntax of a Haskell function

function → **name** parmlist

parmlist → parm | parm parmlist

parm → **name** | **(** function **)**

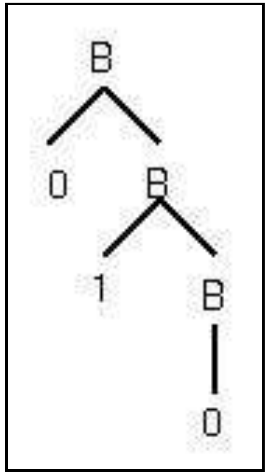
- We can assume that name is a terminal containing a string with a properly formatted name

Derivations

- A derivation is a sequence of steps starting from start symbol applying the syntax rules
- Derivation trees:

  Grammar: $B \rightarrow 0B \mid 1B \mid 0 \mid 1$

  Derivation: $B \Rightarrow 0B \Rightarrow 01B \Rightarrow 010$

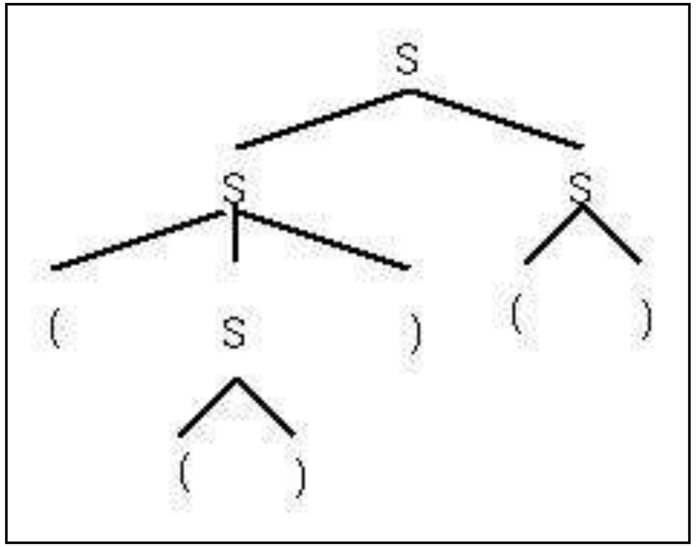


- Derivations can be expressed as a parse tree
- But derivations may not be unique

  $S \rightarrow SS \mid (S) \mid ()$

  $S \Rightarrow SS \Rightarrow (S)S \Rightarrow (())S \Rightarrow (())()$

  $S \Rightarrow SS \Rightarrow S() \Rightarrow (S)() \Rightarrow (())()$



- Different derivations but get the same parse tree

Example BNF Derivations

function → **name** parmlist
parmlist → parm | parm parmlist
parm → **name** | **(** function **)**

is myfunc dog (sqrt cat) a proper Haskell function?

function => name parmlist => name parm parmlist => name parm parm => name name parm => name name (function) => name name (name parmlist) => name name (name parm) => name name (name name) => myfunc name (name name) => myfunc dog (name name) => myfunc dog (sqrt name) => myfunc dog (sqrt cat)

Show a Derivation

- Prove that `dog (cat mouse)` is a properly formatted Haskell function

function → **name** parmlist

parmlist → parm | parm parmlist

parm → **name** | **(** function **)**

Recursion

- Many BNFs involve recursion

  expression → factor + expression

- The recursion allows the language to repeat parts

  expression → factor + expression

  expression → factor + factor + expression

  expression → factor + factor + factor

Write a BNF

- Write a BNF to define the date as written in any one reasonable way

Precedence

- In programming languages there are often semantic rules about how equations are evaluated
- Some syntax specifications will support some semantic rules
- Not all semantic rules can be defined by syntax

Semantic Implications

- Considering any describable semantics, the possible values for `2 * 3 + 4 * 5` include

26	Multiplication before addition
46	Right to left
50	Left to right
70	Addition before multiplication

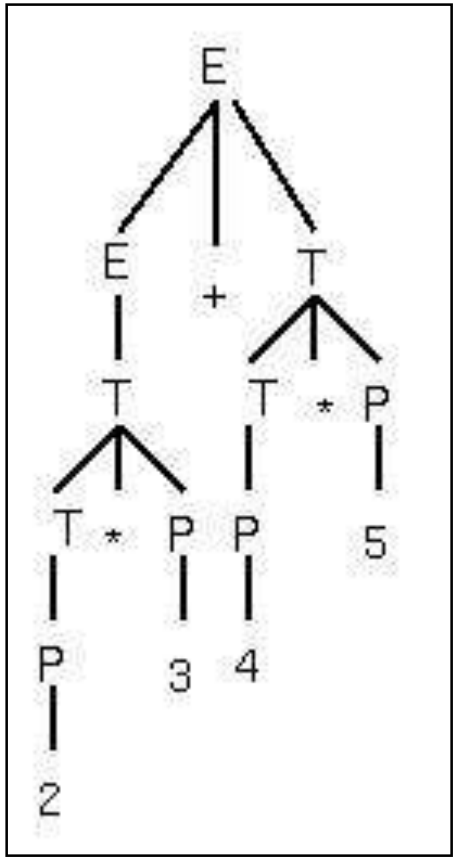
Usual Grammar for Expressions

$E \rightarrow E + T \mid T$

$T \rightarrow T * P \mid P$

$P \rightarrow$ number $\mid (E)$

- result for **2 * 3 + 4 * 5**
- "Natural" value of expression is 26
- Multiply 2 * 3 = 6
- Multiply 4 * 5 = 20
- Add 6 + 20 = 26



Alternate Grammar for Expressions

$$E \rightarrow E * T \mid T$$

$$T \rightarrow T + P \mid P$$

$$P \rightarrow \text{number} \mid (E)$$

- Result for **2 * 3 + 4 * 5**
- add 3 + 4 = 7
- Multiply 2 * 7 = 14
- Multiply 14 * 5 = 70

Draw a parse tree for `2 * 3 + 4 * 5`

$$E \rightarrow E \ * \ T \ | \ T$$

$$T \rightarrow T \ + \ P \ | \ P$$

$$P \rightarrow \text{number} \ | \ (\ E \ )$$

Another Grammar for Expressions

$$E \rightarrow E + T \mid E * T \mid T$$

$$T \rightarrow i \mid (E)$$

- Result for **`2 * 3 + 4 * 5`**
- Multiply 2 * 3 = 6
- add 6 + 4 = 10
- Multiply 10 * 5 = 50

Draw a parse tree for **2 * 3 + 4 * 5**

$$E \rightarrow E + T \mid E * T \mid T$$

$$T \rightarrow i \mid (E)$$

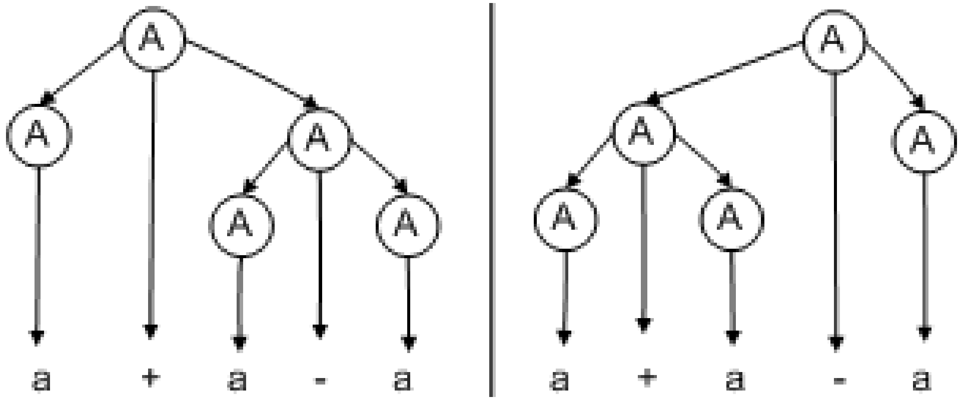
Ambiguous Grammars

- A grammar is ambiguous if a given string can have more than one derivation
- If there is more than one way to generate a string, there can be multiple meanings for the same program
- In general, language specifications should be unambiguous
- The question of whether a grammar is ambiguous is undecidable

Ambiguous Grammar Example

- Consider the ambiguous context free grammar

1. $A \rightarrow A + A$
2. $\quad | \; A - A$
3. $\quad | \; a$



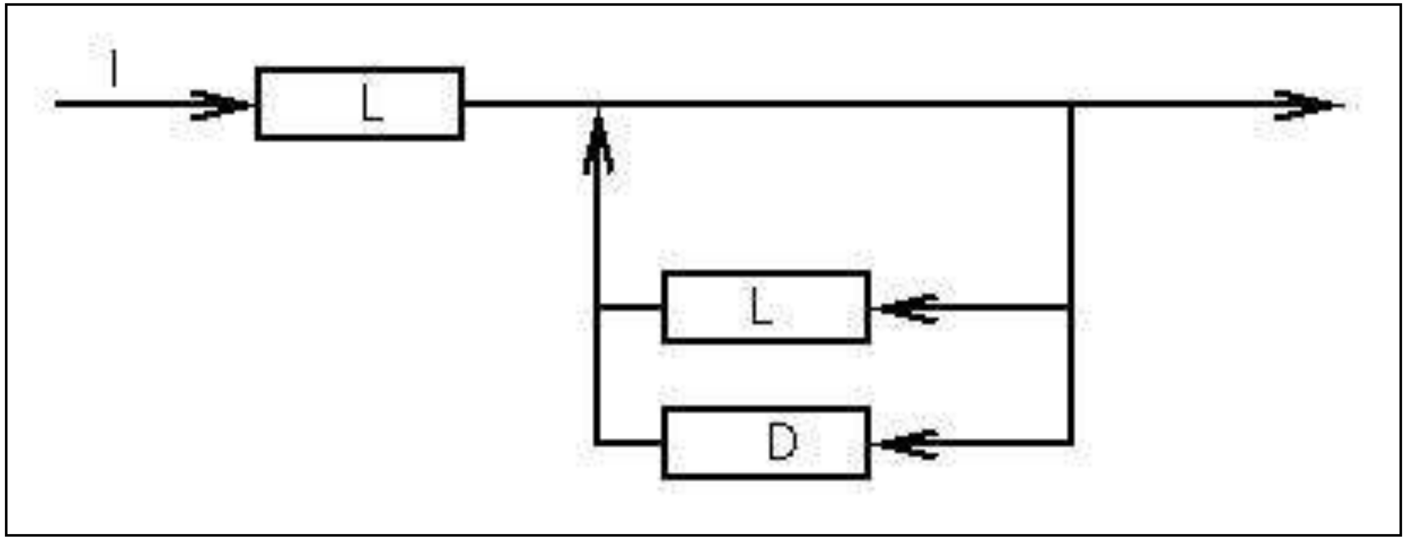
$A \rightarrow A + A$	1	$A \rightarrow A + A$	1
$\rightarrow a + A$	3	$\rightarrow A + A + A$	1
$\rightarrow a + A + A$	1	$\rightarrow a + A + A$	3
$\rightarrow a + a + A$	3	$\rightarrow a + a + A$	3
$\rightarrow a + a + a$	3	$\rightarrow a + a + a$	

Extended BNF

- Nicholas Wirth extended BNF which, while simpler, does not have any more power than regular BNF
- EBNF has
  - repetition, such as dog* or dog^{+}
  - options with commas
  - grouping

Syntax diagrams

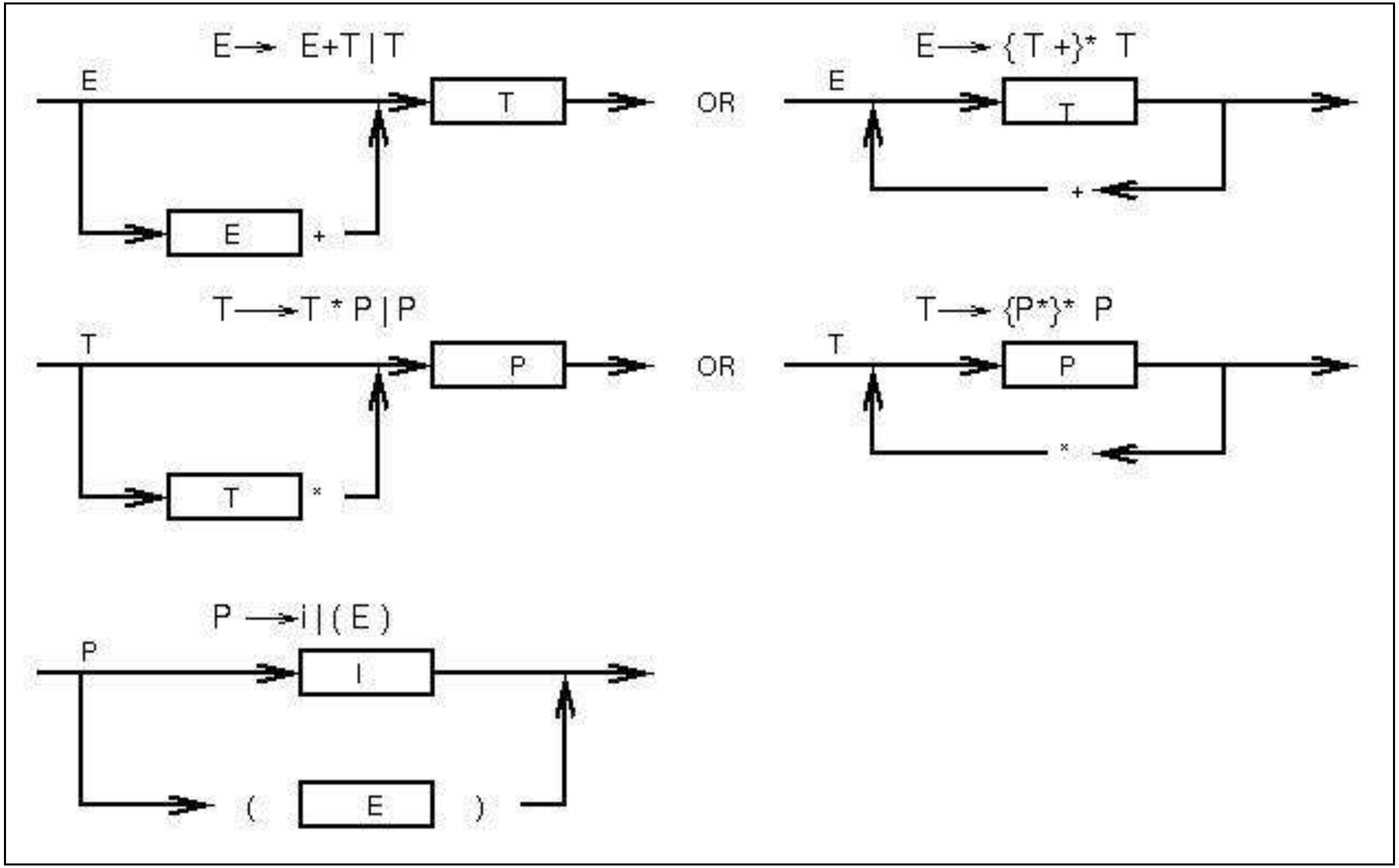
- Also called railroad charts since they look like railroad switching yards.



- Trace a path through network: An L followed by repeated
- loops through L and D, i.e., extended BNF:

$$L \rightarrow L\ (L \mid D)^*$$

Syntax charts for expression grammar



Programming Language design and Implementation -4th Edition Copyright© Prentice Hall, 2000

Other classes of grammars

- The context free and regular grammars are important for programming language design. We study these in detail
- Other classes have theoretical importance, but not in this course
- **Context sensitive grammar:** Rules: $\alpha \rightarrow \beta$ where $| \alpha | \leq | \beta |$ That is, length of $\alpha \leq$ length of β, i.e., all sentential forms are length non-decreasing
- **Unrestricted, recursively enumerable:**

  Rules: $\alpha \rightarrow \beta$. No restrictions on α and β

Looking for Ideas

- An upcoming assignment will ask you to write a scanner and a parser for a language
- We need a simple language to implement
- It must:
  - Not be trivial or too difficult
  - Not be a subset of Java arithmetic
  - Have more than one possible program

Homework

- An assignment has been posted on Blackboard
- You have to write:
  - four regular expressions
  - DFAs to implement two of the regular expressions
  - a program to implement one of the DFA
- Due by noon on Friday, March 4, 2016