Formal Languages - 2

- Context-free PLs
- Grammars
  - Derivation
  - Parsing and parse trees
  - Ambiguity
  - Precedence and Associativity
- Deterministic parsing techniques
  - TD parsing - LL(1)
    - First and Follow sets
    - Parse table construction

Context-free PLs

- Describe most of the constructs in real PLs
- Form of rules
  - Each lhs contains one nonterminal
  - Each rhs contains a sequence of terminals and/or nonterminals
- PLs describable by context-free grammars are recognized by push-down automata (analogous to an FSA with a stack)

Context-free Grammars

- Can be used to generate correct sentences in the PL *(derivation)*
- Can be used to recognize syntactically correct sentences in the PL *(parse)*
  - Can be automated efficiently in a compiler (*LL or LR parsing)*
  - Is insufficient to describe all constructs of a real PL
    - E.g., type checking with declaration

Derivation

1 <letter>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
2 <digit>::= 0|1|2|3|4|5|6|7|8|9
3 <identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
4 <0-assign-stmt> ::= <identifier> = 0

Can we generate x2 = 0 from these rules?

<0-assign-stmt> →4 <identifier> = 0 sentential form
→3c <identifier> <digit> = 0
→3a <letter> <digit> = 0
→1 x <digit> = 0
→2 x 2 = 0 sentence

YES! *leftmost* or *canonical derivation.*

Parse

1 <letter>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
2 <digit>::= 0|1|2|3|4|5|6|7|8|9
3 <identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
4 <0-assign-stmt> ::= <identifier> = 0

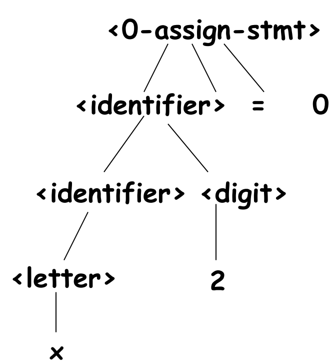
Can we recognize x2 = 0 as belonging to this PL?

x2 = 0	→	<letter> 2 = 0	rule 1
	→	<identifier> 2 = 0	rule 3a
	→	<identifier><digit> = 0	rule 2
	→	<identifier> = 0	rule 3c
	→	<0-assign-stmt>	rule 4

A *parse* of the sentence x2 = 0.

Parse Tree

x2 = 0	→	<letter> 2 = 0	rule 1
	→	<identifier> 2 = 0	rule 3a
	→	<identifier><digit> = 0	rule 2
	→	<identifier> = 0	rule 3c
	→	<0-assign-stmt>	rule 4



In parse tree, each internal node is a nonterminal; its children are the rhs of a rule for that nonterminal. Frontier of the tree is a sentence or valid PL construct.

Grammars are not Unique

1 <letter>::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v| w|x|y|z

2 <digit>::= 0|1|2|3|4|5|6|7|8|9

3' <id> ::= <letter> | <id> <letterordigit>

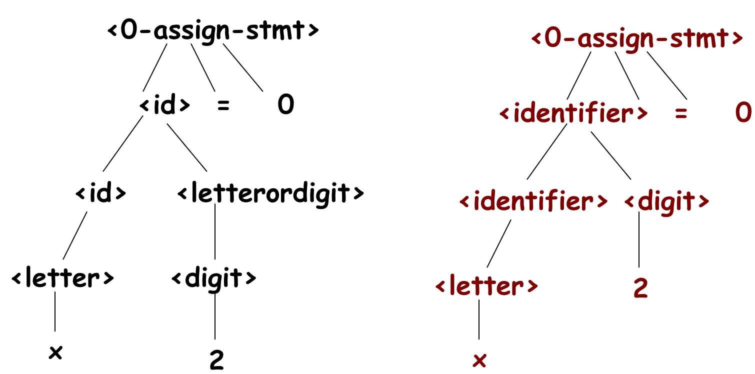
4' <0-assign-stmt> ::= <id> = 0

5' <letterordigit> ::= <letter> | <digit>

This grammar generates the same language (i.e, set of trees whose frontiers are the same), but has different parse trees than the previous grammar.

Example

2nd grammar tree **1st grammar tree**



Many grammars can correspond to 1 PL, but only 1 PL should correspond to any *useful* grammar!

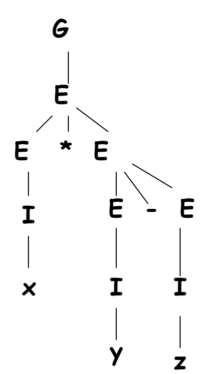
Definitions - Review

- *Grammar*
  - <finite set of terminals, non-terminals, production rules, special symbol>
- *Context-free grammar*
  - corresponds to PLs whose rules have only 1 nonterminal on the lhs
- *Sentence*
  - a finite sequence of terminals, constructed according to the rules of the grammar for that PL
- *Sentential form*
  - a finite sequence of terminals and non-terminals, constructed according to the rules of the grammar for that PL
- *Derivation*
  - A step by step procedure that substitutes righthandsides of productions for the nonterminal on their left, eventually leading to a sequence of terminals that is a sentence in a PL.
- *Parse* (basically a reverse derivation)

Ambiguity in PL Definition

1 G ::= E
  2 E ::= E - E | 3 E * E | 4 I
5 I ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

G	→1	E
	→3	E * E
	→4	I * E
	→5	x * E
	→2	x * E - E
	→4	x * I - E
	→5	x * y - E
	→4	x * y - I
	→5	x * y - z



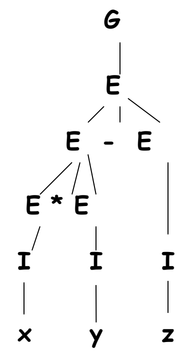
Ambiguity in PL Definition

1 G ::= E

E ::= E - E (2) | E * E (3) | I (4)

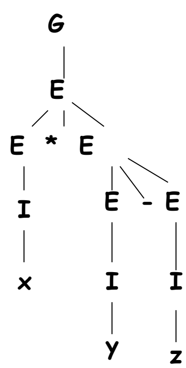
5 I ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

G	→1	E
	→2	E - E
	→3	E * E - E
	→4	I * E - E
	→5	x * E - E
	→4	x * I - E
	→5	x * y - E
	→4	x * y - I
	→5	x * y - z

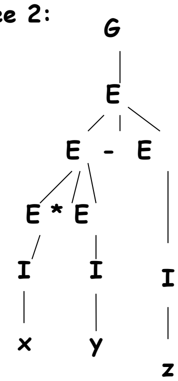


Comparison

Tree 1:



Tree 2:



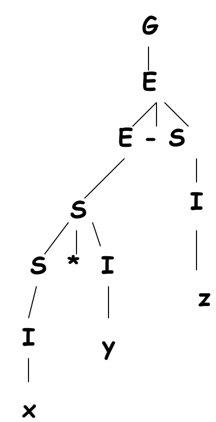
Which tree is correct?
Can we rewrite the grammar to only generate one of them?

Ambiguity

- If there are 2 different canonical derivations (or alternatively, 2 parse trees) for the same sentence then the grammar is *ambiguous*
  - Solution
    - Change grammar to reflect operator precedence
    i.e., X*Y-Z means ((X*Y) – Z)
- There is no algorithm which can tell if an arbitrary context-free grammar is ambiguous
- Also no algorithm to tell if 2 arbitrary context-free grammars generate the same language
  - But can tell if 2 regular languages are equivalent!

A Better Grammar

G::=E
E::=S | E - S
S::=I | S * I
I ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n
|o|p|q|r|s|t|u|v|w|x|y|z



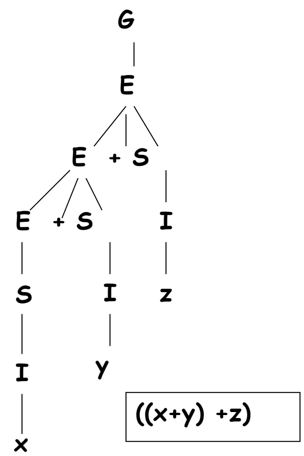
Note: since S is operand of - operation, this forces * to have higher precedence than -.

Associativity in the Grammar

G::=E
E::=E + S | S
S::=I | S * I
I ::=a|b|c|d|e|f|g|h|i|j|k|l|m|n
|o|p|q|r|s|t|u|v|w|x|y|z

How to parse x+y+z?

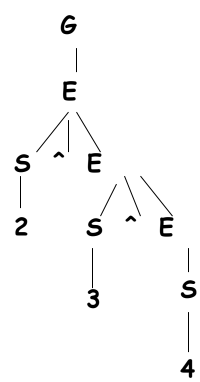
Tree shows that + is left associative because E's rule is left recursive.



Right Associativity

G::= E
E::= S ^ E | S
S::= 0|1|2|3|4|5|6|7|8|
9

What is 2^3^4?
8^4 or 2^{81}?



TD Parsing

Elimination of left recursion to prevent infinite loops in the parse .

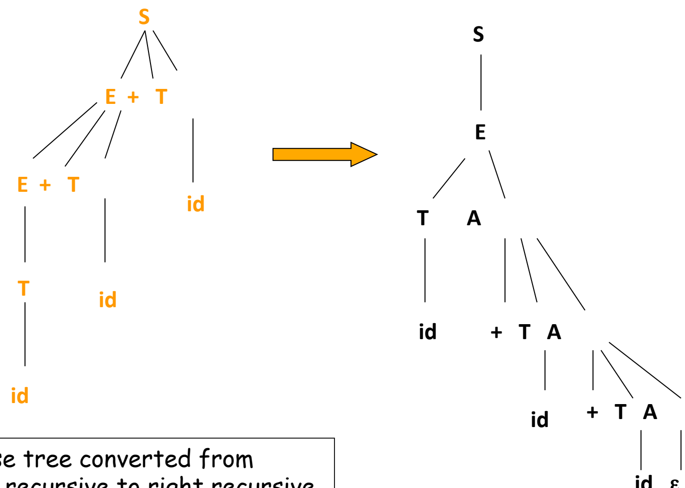
$E \rightarrow E\,\alpha \mid \beta \Longrightarrow E \rightarrow \beta\,A$
$A \rightarrow \alpha\,A \mid \varepsilon$

Example:

$S \rightarrow E$	$S \rightarrow E$
$E \rightarrow E + T$	$E \rightarrow T\ A$
$E \rightarrow T$	$A \rightarrow +\ T\ A \mid \varepsilon$
$T \rightarrow id$	$T \rightarrow id$

Can also left factor the grammar removing shared prefixes of right-hand-sides.

Parse Tree



Parse tree converted from left recursive to right recursive.

TD Parsing

- *Problem:* predicting which nonterminal to expand next, from a leading string of symbols
- *Idea:* generate parse tree top down so its frontier is always a sentential form
  - Use First and Follow sets to understand the shape of sentential forms possibly generated by the grammar

TD Stack Parser, EG

$S \to E$
$E \to T\ A$
$A \to +\ T\ A \mid \varepsilon$
$T \to id$

Stack	Input	Production
$S	*id+id+id$*	
$E	*id+id+id$*	$S \to E$
$A T	*id+id+id$*	$E \to T\ A$
$A	*+id+id$*	$T \to id$
$A T	*id+id$*	$A \to +\ T\ A$
$A	*+id$*	$T \to id$
$A T	*id$*	$A \to +\ T\ A$
$A	$	$T \to id$
$	$	$A \to \varepsilon$

See algm in ASU Fig 4.14, p 187

How to mechanize?

- Define α to be string of non-terminals and terminals
- First(α) is the set of terminals that begin strings derivable from α.

  If $\alpha \stackrel{*}{\Rightarrow} \varepsilon$, then ε is in First(α).
- Follow(A) is the set of terminals that can appear directly to the right of A in a sentential form

  S $\stackrel{*}{\Rightarrow}$ α A a β means a is in Follow(A).

  If A can be rightmost symbol in a sentential form, that is, X $\stackrel{*}{\Rightarrow} \alpha\, A\, \delta$ where $\delta \stackrel{*}{\Rightarrow} \varepsilon$, then

  Follow(X)$\subseteq$ Follow(A)because whatever can follow an X can follow an A too.

Example

- First(S) = First(E) = First(T) = {*id*}
- First(A) = { +, ε }
- Follow(S) = Follow(E) = Follow(A) = {$}
- Follow(T) = {+, $}

  S → E

  E → T A

  A → + T A | ε

  T → *id*

LL(k) Grammars

- Can choose next production to expand by during TD phase, by looking k symbols ahead into input
- Use First sets to choose production
- Use Follow sets to handle ε cases

Example: LL(1)

Nonterms\Inputs:	id	+	$
S	S → E		
E	E → T A		
T	T → id		
A		A → + T A	A → ε

Ambiguous or left recursive grammars result in multiply defined entries in table – a problem!

First(S) = First(E) = First(T) = {id}
First(A) = { +, e }
Follow(S) = Follow(E) = Follow(A) = {$}
Follow(T) = {+, $}

S → E
E → T A
A → + T A | ε
T → id