Syntax-Directed Translation

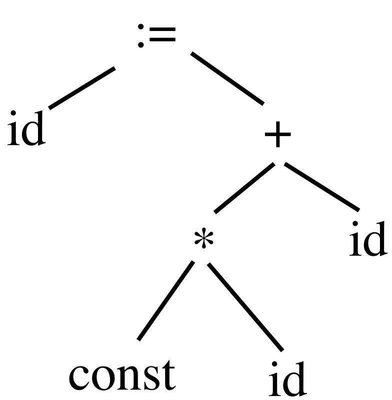
ALSU Textbook Chapter 5.1–5.4, 4.8, 4.9

What is syntax-directed translation?

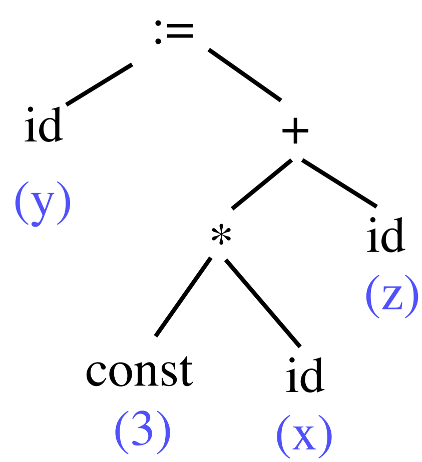
- **Definition:**
  - **The compilation process is driven by the syntax.**
  - **The semantic routines perform interpretation based on the syntax structure.**
  - **Attaching attributes to the grammar symbols.**
  - **Values for attributes are computed by semantic actions associated with the grammar productions.**

Example: Syntax-directed translation

- **Example in a parse tree:**
  - **Annotate the parse tree by attaching semantic attributes to the nodes of the parse tree.**
  - **Generate code by visiting nodes in the parse tree in a given order.**
  - **Input:** $y := 3 * x + z$



parse tree



annotated parse tree

Syntax-directed definitions

- **Each grammar symbol is associated with a set of attributes.**
  - **Synthesized attribute : value computed from its children or associated with the meaning of the tokens.**
  - **Inherited attribute : value computed from parent and/or siblings.**
  - **General attribute : value can be depended on the attributes of any nodes.**

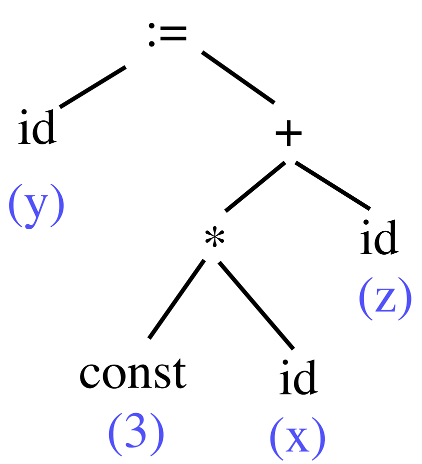
Format for writing syntax-directed definitions

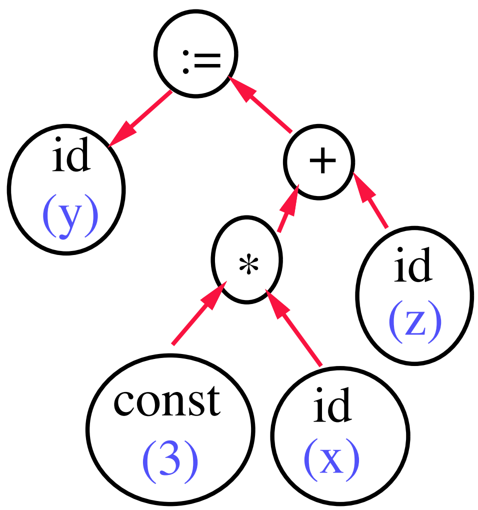
Production	Semantic actions
$L \to E$	print($E.val$)
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to digit$	$F.val := digit.lexval$

- **$E.val$ is one of the attributes of E.**
- **To avoid confusion, recursively defined nonterminals are numbered on the RHS.**
- **Semantic actions are performed when this production is "used".**

Order of evaluation (1/2)

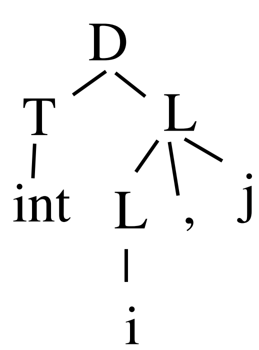
- **Order of evaluating attributes is important.**
- **General rule for ordering:**
  - **Dependency graph** :
    - *If attribute b needs attributes a and c, then a and c must be evaluated before b.*
    - *Represented as a directed graph without cycles.*
    - *Topologically order nodes in the dependency graph as n_1, n_2, ..., n_k such that there is no path from n_i to n_j with $i > j$.*

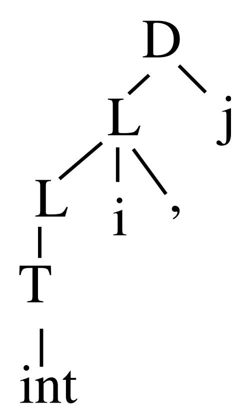




Order of evaluation (2/2)

- **It is always possible to rewrite syntax-directed definitions using only synthesized attributes, but the one with inherited attributes is easier to understand.**
  - **Use inherited attributes to keep track of the type of a list of variable declarations.**
    - ***Example:*** $int\ i, j$
  - **Grammar 1: using inherited attributes**
    - $D \rightarrow T L$
    - $T \rightarrow int \mid char$
    - $L \rightarrow L, id \mid id$
  - **Grammar 2: using only synthesized attributes**
    - $D \rightarrow L\ id$
    - $L \rightarrow L\ id, \mid T$
    - $T \rightarrow int \mid char$





Attribute grammars

- **Attribute grammar: a grammar with syntax-directed definitions and having no side effects.**
  - **Side effect: change values of others not related to the return values of functions themselves.**
- **Tradeoffs:**
  - **Synthesized attributes are easy to compute, but are sometimes difficult to be used to express semantics.**
    - ▷ *S-attributes.*
  - **Inherited and general attributes are difficult to compute, but are sometimes easy to express the semantics.**
  - **The dependence graph for computing some inherited and general attributes may contain cycles and thus not be computable.**
  - **A restricted form of inherited attributes is invented.**
    - ▷ *L-attributes.*

S-attributed definition

- **Definition: a syntax-directed definition that uses synthesized attributed only.**
  - **A parse tree can be represented using a directed graph.**
  - **A post-order traverse of the parse tree can properly evaluate grammars with S-attributed definitions.**
  - **Goes naturally with LR parsers.**
- **Example of an S-attributed definition: $3 * 5 + 4$ return**

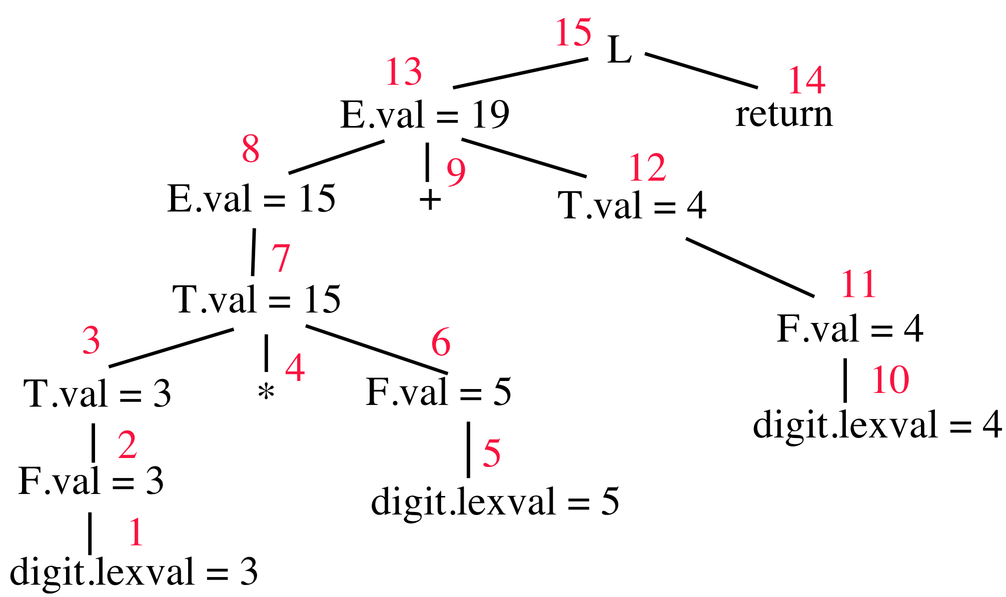
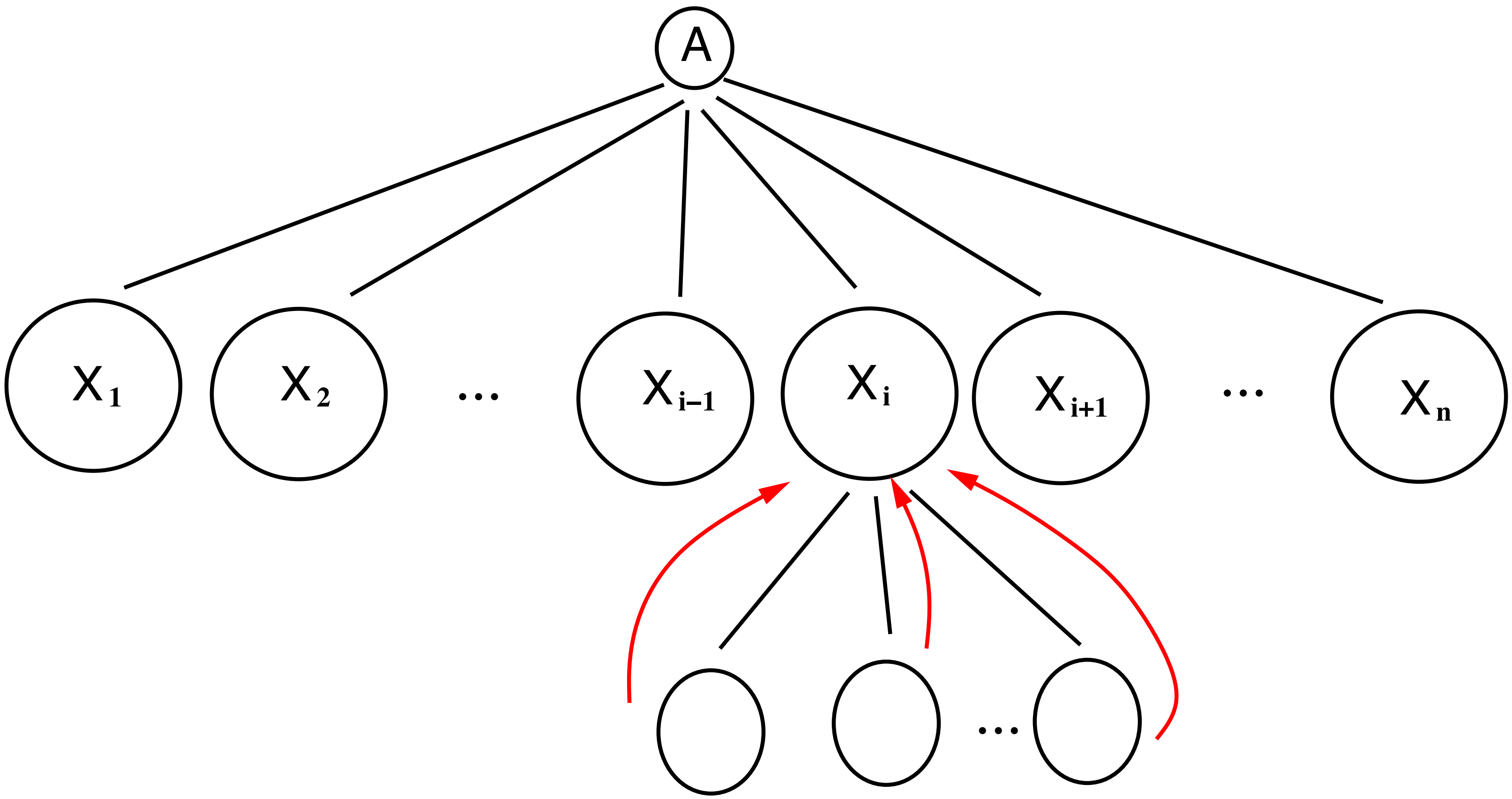


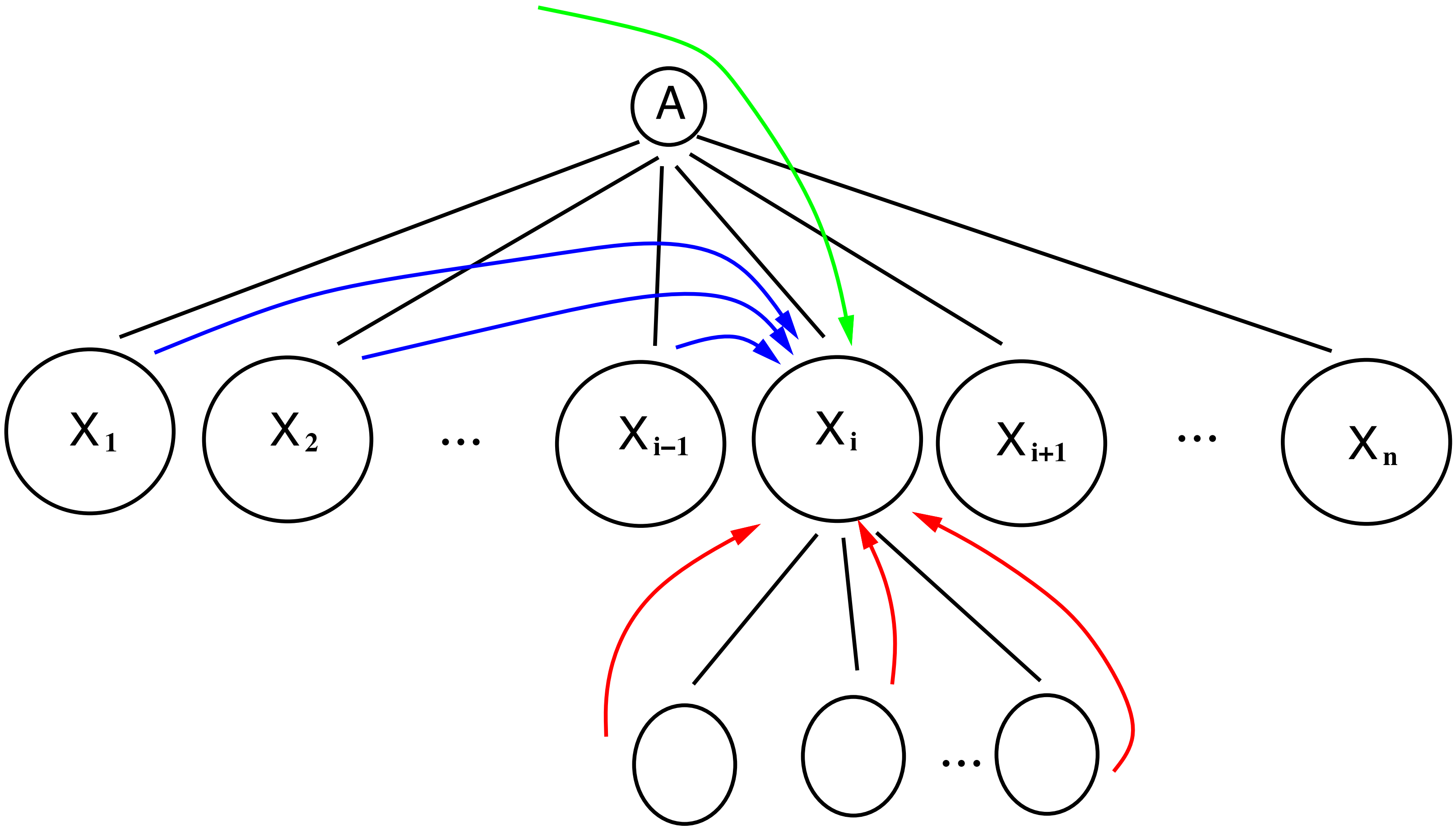
Illustration: S-attributed definition



L-attributed definitions

- Each grammar symbol can have many attributes. However, each attribute must be either
  - a synthesized attribute, or
  - an inherited attribute with the following constraints.
    Assume there is a production $A \to X_1 X_2 \cdots X_n$ and the inherited attribute is associated with X_i. Then this inherited attribute depends only on
    - *the inherited attributes of its parent node A;*
    - *either inherited or synthesized attributes from its elder siblings $X_1, X_2, \ldots, X_{i-1}$;*
    - *inherited or synthesized attributed associated from itself X_i, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.*
- Every S-attributed definition is an L-attributed definition.

Illustration: L-attributed definition



Evaluations of L-attributed definitions

- **For grammars with L-attributed definitions, special evaluation algorithms must be designed.**
- **L-attributes are always computable.**
  - **Similar arguments as the one used in discussing Algorithm 4.19 for removing left recursion.**
- **Evaluation of L-attributed grammars.**
  - **Goes together naturally with LL parsers.**
    - ▷ ***Parse tree generate by recursive descent parsing corresponds naturally to a top-down tree traversal using DFS by visiting the sibling nodes from left to right.***
- **High level ideas for tree traversal.**
  - **Visit a node v first.**
    - ▷ ***Compute inherited attributes for v if they do not depend on synthesized attributes of v.***
  - **Recursively visit each children of v one by one from left to right.**
  - **Visit the node v again.**
    - ▷ ***Compute synthesized attributes for v.***
    - ▷ ***Compute inherited attributes for v if they depend on synthesized attributes of v.***

Format for writing L-attributed definitions

- $D \to T \ \{L.in := T.type\} \ L$
- $T \to int \ \{T.type := integer\}$
- $T \to real \ \{T.type := real\}$
- $L \to \ \{L_1.in := L.in\} \ L_1, id \ \{addtype(id.entry, L.in)\}$
- $L \to id \ \{addtype(id.entry, L.in)\}$

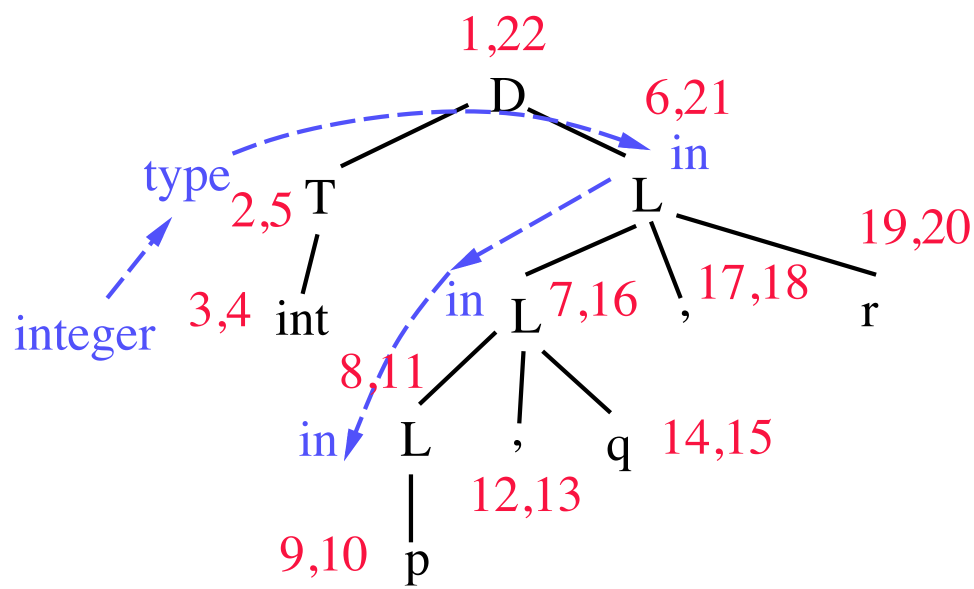
- **Some semantic actions can be inserted between symbols on the RHS of a production.**
  - $A \to B \ \{action\} \ C$
  - **When A expands to B and C, after finishes expanding B, performs $action$, then expands C.**

Example: L-attributed definitions

- $D \to T$ $\{L.in := T.type\}$ L
- $T \to int$ $\{T.type := integer\}$
- $T \to real$ $\{T.type := real\}$
- $L \to$ $\{L_1.in := L.in\}$ L_1, id $\{addtype(id.entry, L.in)\}$
- $L \to id$ $\{addtype(id.entry, L.in)\}$

Parsing and dependency graph:

STACK	input	production used
	int p, q, r	
D	**int** p, q, r	
$L\ T$	**int** p, q, r	$D \to T L$
$L\ int$	**int** p, q, r	$T \to int$
L	p, q, r	
$id\ ,\ L$	p, q, r	$L \to L, id$
$id\ ,\ id\ ,\ L$	p, q, r	$L \to L, id$
$id\ ,\ id\ ,\ id$	p, q, r	$L \to id$
$id\ ,\ id$	q, r	
id	q	



Problems with L-attributed definitions

- **Comparisons:**
  - L-attributed definitions go naturally with LL parsers.
  - S-attributed definitions go naturally with LR parsers.
  - L-attributed definitions are more flexible than S-attributed definitions.
  - LR parsers are more powerful than LL parsers.
- **Some cases of L-attributed definitions cannot be in-cooperated into LR parsers.**
  - Assume the next handle to take care is $A \to X_1 X_2 \cdots X_i \cdots X_k$, and $X_1, \ldots, X_i$ is already on the top of the STACK.
  - Attribute values of $X_1, \ldots, X_{i-1}$ can be found on the STACK at this moment.
  - No information about A can be found anywhere at this moment.
  - Thus the attribute values of X_i cannot be depended on the value of A.
- **L^--attributed definitions:**
  - Same as L-attributed definitions, but **do not** depend on
    - ▷ *the inherited attributes of parent nodes, or*
    - ▷ *any attributes associated with itself.*
  - Can be handled by LR parsers.

Illustration: L^--attributed definition