Formal Languages and Compilers

Lecture VII—Semantic Analysis: Syntax Directed Translation

Alessandro Artale

Free University of Bozen-Bolzano
Faculty of Computer Science – POS Building, Room: 2.03
artale@inf.unibz.it
http://www.inf.unibz.it/∼artale/

Formal Languages and Compilers — BSc course

2020/21 – Second S

Summary of Lecture VII

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

Semantic Analysis

- Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a *Representation Formalism* and an *Implementation Mechanism*.
- As representation formalism this lecture illustrates what are called *Syntax Directed Translations*.

Syntax Directed Translation: Intro

- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
  - We associate Attributes to the non-terminal symbols of the grammar;
  - Values for attributes are computed by Semantic Rules associated with grammar productions.

Syntax Directed Translation: Intro (Cont.)

- Evaluation of Semantic Rules may:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages;
  - etc.
- There are two notations for attaching semantic rules:
  1. Syntax Directed Definitions. High-level specification hiding many implementation details (also called **Attribute Grammars**).
  2. Translation Schemes. More implementation oriented: Indicate the evaluation order of the semantic rules.

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

Syntax Directed Definitions

- Syntax Directed Definitions are a generalization of context-free grammars in which:
  1. Grammar symbols have an associated set of **Attributes**;
  2. Productions are associated with **Semantic Rules** for computing the values of attributes.
- Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g., $X.a$ indicates the attribute a of the grammar symbol X).

Syntax Directed Definitions (Cont.)

- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.
- We distinguish between two kinds of attributes:
  1. Synthesized Attributes. They are computed from the values of the attributes of the children nodes.
  2. Inherited Attributes. They are computed from the values of the attributes of both the siblings and the parent nodes.

Form of Syntax Directed Definitions

- Each production, $A \to \alpha$, is associated with a set of semantic rules: $b := f(c_1, c_2, \ldots, c_k)$, where f is a function and either
  1. b is a **synthesized** attribute of A, and $c_1, c_2, \ldots, c_k$ are attributes of the grammar symbols of the production (including A itself), or
  2. b is an **inherited** attribute of a grammar symbol in α, and $c_1, c_2, \ldots, c_k$ are attributes of grammar symbols in α or attributes of A.
- **Note 1.** Terminal symbols are assumed to have an attribute which coincides with the attribute supplied by the lexical analyzer.
- **Note 2.** Procedure calls (e.g. *print* in the next slide) define values of *Dummy* synthesized attributes of the non terminal on the left-hand side of the production.

Syntax Directed Definitions: An Example

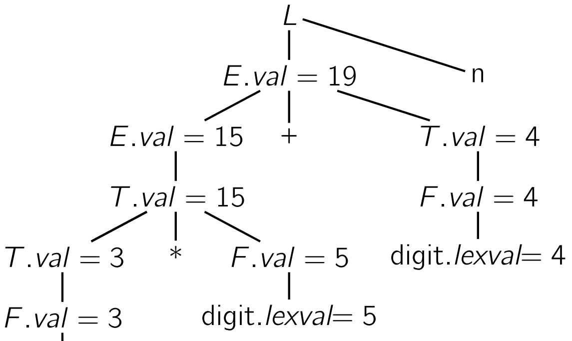
- **Example.** Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val*.

Production	Semantic Rule
$L \rightarrow E\mathrm{n}$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \mathrm{digit}$	$F.val := \mathrm{digit}.lexval$

S-Attributed Definitions

Definition. An S-Attributed Definition is a Syntax Directed Definition that uses only synthesized attributes.

- **Evaluation Order.** Semantic rules in an S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.
- **Example.** The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



Inherited Attributes

- Inherited Attributes are useful for expressing the dependence of a construct on the context in which it appears.
- **Note:** It is always possible to rewrite a syntax directed definition to use only synthesized attributes, but it is often more natural to use both synthesized and inherited attributes.
- **Evaluation Order.** Inherited attributes **can not** be evaluated by a simple PreOrder traversal of the parse-tree:
  - Unlike synthesized attributes, the order in which the inherited attributes of the children are computed is important!!! Indeed:
  - Inherited attributes of the children can depend from both left and right siblings!

Inherited Attributes: An Example

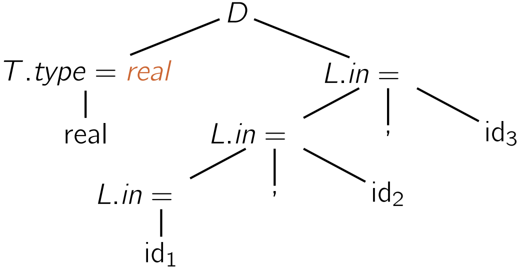
- **Example.** Let us consider the syntax directed definition with both inherited and synthesized attributes for the grammar for "type declarations":

Production	Semantic Rule
$D \to T\ L$	$L.in := T.type$
$T \to$int	$T.type := integer$
$T \to$real	$T.type := real$
$L \to L_1$, id	$L_1.in := L.in$; $addtype($id$.entry, L.in)$
$L \to$ id	$addtype($id$.entry, L.in)$

- The non terminal T has a synthesized attribute, *type*, determined by the tokens int/real in the corresponding production.
- The production $D \to T\ L$ is associated with the semantic rule $L.in := T.type$ which set the *inherited* attribute $L.in$.
- **Note:** The production $L \to L_1$, id distinguishes the two occurrences of L.

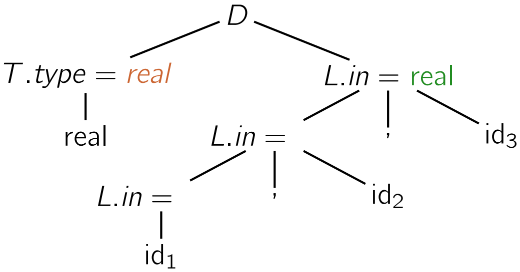
Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.
- Inherited attributes that *do not depend from right children* can be evaluated by a PreOrder traversal.
- The annotated parse-tree for the input real id_1, id_2, id_3 is:



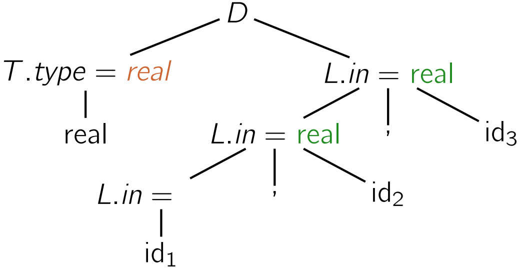
Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.
- Inherited attributes that *do not depend from right children* can be evaluated by a PreOrder traversal.
- The annotated parse-tree for the input real id_1, id_2, id_3 is:



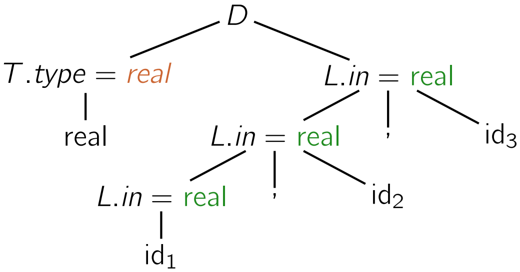
Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.
- Inherited attributes that *do not depend from right children* can be evaluated by a PreOrder traversal.
- The annotated parse-tree for the input real id_1, id_2, id_3 is:



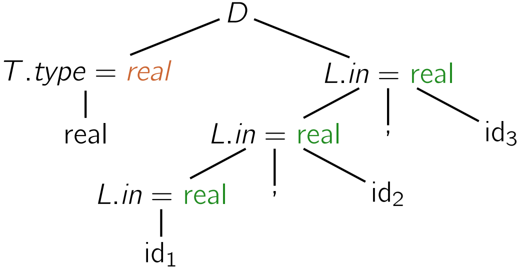
Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.
- Inherited attributes that *do not depend from right children* can be evaluated by a PreOrder traversal.
- The annotated parse-tree for the input real id_1, id_2, id_3 is:



Inherited Attributes: An Example (Cont.)

- Synthesized attributes can be evaluated by a PostOrder traversal.
- Inherited attributes that *do not depend from right children* can be evaluated by a PreOrder traversal.
- The annotated parse-tree for the input real id_1, id_2, id_3 is:



- $L.in$ is then inherited top-down the tree by the other L-nodes.
- At each L-node the procedure *addtype* inserts into the symbol table the type of the identifier.

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

Dependency Graphs

- Implementing a Syntax Directed Definition consists primarily in finding an order for the evaluation of attributes
  - Each attribute value must be available when a computation is performed.
- Dependency Graphs are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.
- A Dependency Graph shows the interdependencies among the attributes of the various nodes of a parse-tree.
  - There is a node for each attribute;
  - If attribute b depends on an attribute c there is a link from the node for c to the node for b ($b \leftarrow c$).
- **Dependency Rule:** If an attribute b depends from an attribute c, then we need to fire the semantic rule for c first and then the semantic rule for b.

Evaluation Order

- The evaluation order of semantic rules depends from a *Topological Sort* derived from the dependency graph.
- **Topological Sort:** Any ordering $m_1, m_2, \ldots, m_k$ such that if $m_i \rightarrow m_j$ is a link in the dependency graph then $m_i < m_j$.
- Any topological sort of a dependency graph gives a valid order to evaluate the semantic rules.

Dependency Graphs: An Example

- **Example.** Build the dependency graph for the parse-tree of real id_1, id_2, id_3.

Implementing Attribute Evaluation: General Remarks

- Attributes can be evaluated by building a dependency graph at compile-time and then finding a topological sort.
- **Disavantages**
  1. This method fails if the dependency graph has a cycle: We need a test for non-circularity;
  2. This method is time consuming due to the construction of the dependency graph.
- **Alternative Approach.** Design the syntax directed definition in such a way that attributes can be evaluated with a *fixed order* avoiding to build the dependency graph (method followed by many compilers).

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

Evaluation of S-Attributed Definitions

- Synthesized Attributes can be evaluated by a bottom-up parser as the input is being analyzed avoiding the construction of a dependency graph.
- The parser keeps the values of the synthesized attributes in its stack.
- Whenever a reduction $A \to \alpha$ is made, the attribute for A is computed from the attributes of α which appear on the stack.
- Thus, a translator for an S-Attributed Definition can be simply implemented by extending the stack of an LR-Parser.

Extending a Parser Stack

- Extra fields are added to the stack to hold the values of synthesized attributes.
- In the simple case of just one attribute per grammar symbol the stack has two fields: *state* and *val*

state	*val*
Z	$Z.x$
Y	$Y.x$
X	$X.x$
$\ldots$	$\ldots$

- The current top of the stack is indicated by the pointer variable *top*.
- Synthesized attributes are computed just before each reduction:
  - Before the reduction $A \to XYZ$ is made, the attribute for A is computed: $A.a := f(val[top], val[top-1], val[top-2])$.

Extending a Parser Stack: An Example

- **Example.** Consider the S-attributed definitions for the arithmetic expressions. To evaluate attributes the parser executes the following code

Production	Code
$L \to E\mathrm{n}$	$print(val[top-1])$
$E \to E_1 + T$	$val[ntop] := val[top] + val[top-2]$
$E \to T$	
$T \to T_1 * F$	$val[ntop] := val[top] * val[top-2]$
$T \to F$	
$F \to (E)$	$val[ntop] := val[top-1]$
$F \to \mathrm{digit}$	

- The auxiliary variable *ntop* is set to the *new top of the stack*: when a reduction $A \to \alpha$ is done, with $|\alpha| = r$, then $ntop = top - r + 1$. After the reduction is done *top* is set to *ntop*.
- During a shift action both the token and its attribute (as returned by the lexical analyzer) are pushed into the stack.

Extending a Parser Stack: An Example (Cont.)

- The following Figure shows the moves made by the parser on input 3*5+4n.
  - Stack states are replaced by their corresponding grammar symbol;
  - Instead of the token digit the actual value is shown.

INPUT	*state*	*val*	PRODUCTION USED
3*5+4**n**	–	–	
*5+4**n**	3	3	
*5+4**n**	F	3	$F \to$ **digit**
*5+4**n**	T	3	$T \to F$
5+4**n**	T *	3 _	
+4**n**	T * 5	3 _ 5	
+4**n**	T * F	3 _ 5	$F \to$ **digit**
+4**n**	T	15	$T \to T * F$
+4**n**	E	15	$E \to T$
4**n**	E +	15 _	
n	E + 4	15 _ 4	
n	E + F	15 _ 4	$F \to$ **digit**
n	E + T	15 _ 4	$T \to F$
n	E	19	$E \to E + T$
	E **n**	19 _	
	L	19	$L \to E$ **n**

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

L-Attributed Definitions

- L-Attributed Definitions contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.
- **Definition.** A syntax directed definition is *L-Attributed* if each *inherited attribute* of X_j in a production $A \to X_1 \ldots X_j \ldots X_n$, depends only on:
  1. The synthesised and inherited attributes of the symbols to the **left** (this is what L in *L-Attributed* stands for) of X_j, i.e., $X_1 X_2 \ldots X_{j-1}$, and
  2. The *inherited* attributes of A.
- **Theorem.** Inherited attributes in L-Attributed Definitions can be computed by a PreOrder traversal of the parse-tree.

Evaluating L-Attributed Definitions

- L-Attributed Definitions are a class of syntax directed definitions whose attributes can always be evaluated by single traversal of the parse-tree.
- The following procedure evaluate L-Attributed Definitions by mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

  Algorithm: L-Eval(n: Node)

  Input: Node of an annotated parse-tree.

  Output: Attribute evaluation.

  Begin
      For each child m of n, from left-to-right Do
      Begin
          Evaluate inherited attributes of m;
          L-Eval(m)
      End;
      Evaluate synthesized attributes of n
  End.

Summary

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes

Translation Schemes

- Translation Schemes are more implementation oriented than syntax directed definitions since they indicate the order in which semantic rules and attributes are to be evaluated.
- **Definition.** A Translation Scheme is a context-free grammar in which
  1. Attributes are associated with grammar symbols;
  2. Semantic Actions are enclosed between braces {} and are inserted within the right-hand side of productions.
- **Note:** Yacc uses Translation Schemes.

Translation Schemes (Cont.)

- Translation Schemes deal with both synthesized and inherited attributes.
- Semantic Actions are treated as terminal symbols: Annotated parse-trees contain semantic actions as children of the node standing for the corresponding production.
- Translation Schemes are useful to evaluate L-Attributed definitions at parsing time (even if they are a general mechanism).
  - An L-Attributed Syntax-Directed Definition can be turned into a Translation Scheme.

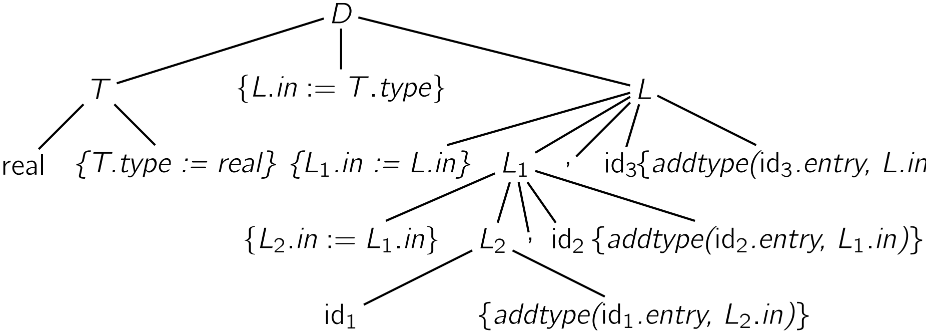
Translation Schemes: An Example

- Consider the Translation Scheme for the L-Attributed Definition for "type declarations":

$$\begin{array}{ll}
D \rightarrow & T \;\; \{L.in := T.type\} \;\; L \\
T \rightarrow & \text{int} \;\; \{T.type := integer\} \\
T \rightarrow & \text{real} \;\; \{T.type := real\} \\
L \rightarrow & \{L_1.in := L.in\} \;\; L_1, \text{id} \;\; \{addtype(\text{id}.entry, L.in)\} \\
L \rightarrow & \text{id} \;\; \{addtype(\text{id}.entry, L.in)\}
\end{array}$$

Translation Schemes: An Example (Cont.)

- **Example (Cont).** The parse-tree with semantic actions for the input real id_1, id_2, id_3 is:



- Traversing the Parse-Tree in depth-first order (PostOrder) we can evaluate the attributes.

Design of Translation Schemes

- When designing a Translation Scheme we must be sure that an attribute value is available when a semantic action is executed.
- When the semantic action involves synthesized attributes: The action can be put at the end of the production.
  - **Example.** The following Production and Semantic Rule:
    $$T \to T_1 * F \quad T.val := T_1.val * F.val$$
    yield the translation scheme:
    $$T \to T_1 * F \quad \{T.val := T_1.val * F.val\}$$

Design of Translation Schemes (cont.)

- When the semantic action involves inherited attributes of a grammar symbol: The action must be put before the symbol itself.
  - **Example.** The following Production and Semantic Rule:

    $$D \rightarrow \;\; T \;\, L \qquad L.in := T.type$$

    yield the translation scheme:

    $$D \rightarrow \;\; T \;\; \{L.in := T.type\} \;\; L$$

Design of Translation Schemes: Summary

- **Rules for Implementing L-Attributed SDD's.** If we have an L-Attibuted Syntax-Directed Definition we must enforce the following restrictions:
  1. An inherited attribute for a symbol in the right-hand side of a production must be computed in an action **before** the symbol;
  2. A synthesized attribute for the non terminal on the left-hand side can only be computed when all the attributes it references have been computed: The action is usually put **at the end** of the production.

Parsing-Time Evaluation of Translation Schemes

- Attributes in a Translation Scheme following the above rules can be computed at parsing time similarly to the evaluation of S-Attributed Definitions.
- **Main Idea.** Starting from a Translation Scheme (with embedded actions) we introduce a transformation that makes all the actions occur at the right ends of their productions.
    - For each embedded semantic action we introduce a new *Marker* (i.e., a non terminal, say M) with an empty production ($M \to \epsilon$);
    - The semantic action is attached at the end of the production $M \to \epsilon$.

Parsing-Time Evaluation of Translation Schemes (Cont.)

- **Example.** Consider the following translation scheme:
  $S \to aA\{C.i = f(A.s)\}C$
  $S \to bAB\{C.i = f(A.s)\}C$
  $C \to c\{C.s = g(C.i)\}$
  Then, we add new markers M_1, M_2 with:
  $S \to aAM_1C$
  $S \to bABM_2C$
  $M_1 \to \epsilon \qquad \{M_1.s := f(val[top])\}$
  $M_2 \to \epsilon \qquad \{M_2.s := f(val[top-1])\}$
  $C \to c \qquad \{C.s := g(val[top-1])\}$
  The inherited attribute of C is the synthesized attribute of either M_1 or M_2: The value of $C.i$ is *always* in $val[top\ \text{-}1]$ when $C \to c$ is applied.

Parsing-Time Evaluation of Translation Schemes (Cont.)

General rules to compute translations schemes during bottom-up parsing assuming an L-attributed grammar.

- For every production $A \rightarrow X_1 \dots X_n$ introduce n new markers $M_1, \dots, M_n$ and replace the production by $A \rightarrow M_1 X_1 \dots M_n X_n$.
- Thus, we know the position of every synthesized and inherited attribute of X_j and A:
  1. $X_j.s$ is stored in the *val* entry in the parser stack associated with X_j;
  2. $X_j.i$ is stored in the *val* entry in the parser stack associated with M_j;
  3. $A.i$ is stored in the *val* entry in the parser stack immediately before the position storing M_1.
- **Remark 1.** Since there is only one production for each marker a grammar remains LL(1) with addition of markers.
- **Remark 2.** Adding markers to an LR(1) Grammar can introduce conflicts for not L-Attributed SDD's!!!

Parsing-Time Evaluation of Translation Schemes (Cont.)

Example. Computing the inherited attribute $X_j.i$ after reducing with $M_j \to \epsilon$.

	M_j	$X_j.i$
$top \to$	X_{j-1}	$X_{j-1}.s$
	M_{j-1}	$X_{j-1}.i$
	$\ldots$	$\ldots$
	X_1	$X_1.s$
	M_1	$X_1.i$
$(top\text{-}2j\text{+}2) \to$	M_A	$A.i$
$(top\text{-}2j) \to$		

- $A.i$ is in $val[top - 2j + 2]$;
- $X_1.i$ is in $val[top - 2j + 3]$;
- $X_1.s$ is in $val[top - 2j + 4]$;
- $X_2.i$ is in $val[top - 2j + 5]$;
- and so on.

Summary of Lecture VII

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions
- Translation Schemes