Compiler Theory: Syntax-Directed Translation

Marc Moreno Maza
University of Western Ontario
CS447b – Winter 2002

December 1, 2004

1 Syntax-Directed Definitions

A SYNTAX-DIRECTED DEFINITION is a context-free grammar in which

- each grammar symbol X is associated with two finite sets of values: the *synthesized attributes* of X and the *inherited attributes* of X,
- each production $A \longmapsto \alpha$ is associated with a finite set of expressions of the form
$$b \ := \ f(c_1, \ldots, c_k) \tag{1}$$
called *semantic rules* where f is a function and
  - either b is a synthesized attribute of A and the values $c_1, \ldots, c_k$ are attributes of the grammar symbols of α or A,
  - or b is an inherited attribute of a grammar symbol of α and the values $c_1, \ldots, c_k$ are attributes of the grammar symbols of α or A.
- each terminal symbol has no inherited attributes.

It is usual to denote the attributes of a grammar symbol in the form $X.name$ where $name$ is an meaningful name for the attribute.

Example 1 The following syntax-directed definition generates binary strings where (one of) the attributes gives the number represented by the string. The nonterminals are B and D, the terminals are 0 and 1. The symbol B stands for *binary expansion* and the symbol D stands for *digit*.

Grammar symbol	Synthesized attribute	Inherited attribute
B	$B.pos$, $B.val$	
D	$D.val$	$D.pow$

Production	Semantic Rule
$B \longmapsto DB_1$	$B.pos := B_1.pos + 1$
	$B.val := B_1.val + D.val$
	$D.pow := B_1.pos$
$B \longmapsto D$	$B.pos := 1$
	$B.val := D.val$
	$D.pow := 0$
$D \longmapsto 0$	$D.val := 0$
$D \longmapsto 1$	$D.val := 2^{D.pow}$

<u>At a parse tree node,</u> following the previous definition,

- the value of a synthesized attribute is computed from the values of the attributes of the children of the node.
- the value of an inherited attributes is computed from the values of the attributes of its siblings (brothers and sisters) and parents.

Example 2 Let us continue Example 1 and let us consider the input string $w = 1010$. Figure 1 shows a parse tree for w. Figure 2 shows the values of the attributes at each node.

Production	Semantic Rule
$B \longmapsto DB_1$	$B.pos := B_1.pos + 1$
	$B.val := B_1.val + D.val$
	$D.pow := B_1.pos$
$B \longmapsto D$	$B.pos := 1$
	$B.val := D.val$
	$D.pow := 0$
$D \longmapsto 0$	$D.val := 0$
$D \longmapsto 1$	$D.val := 2^{D.pow}$

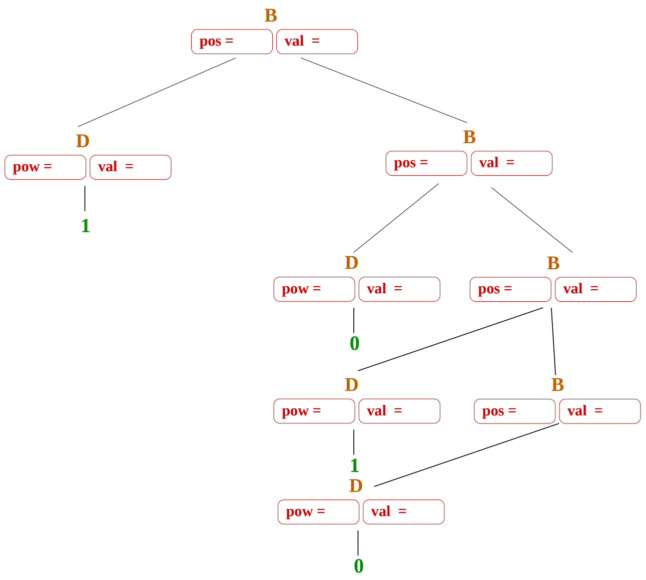

Figure 1: Parse tree for $w = 1010$.

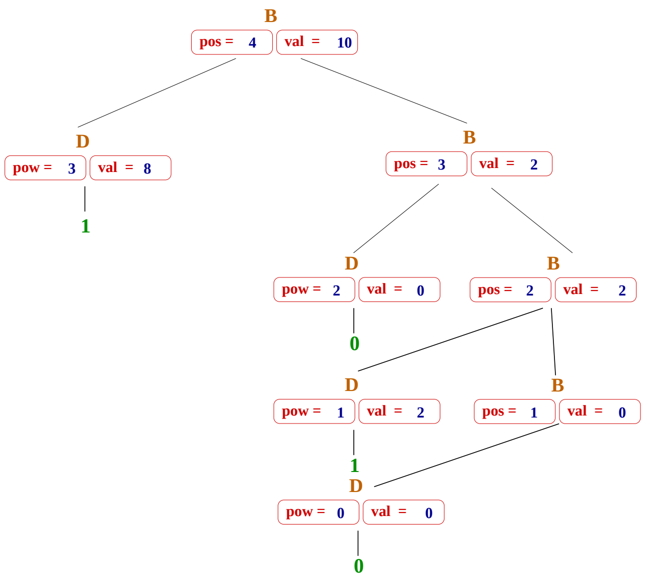

Figure 2: Annotated parse tree for $w = 1010$.

Semantic rules set up dependencies between attributes that will be represented as a *dependency graph.*

- From this dependency graph, we will derive an *evaluation order* for the semantic rules.
- This *evaluation order* defines the values of the attributes at the nodes in a parse tree.
- Terminals do not have semantic rules since their attributes are supplied by the lexical analyzer.

Attribute grammars. It is useful to allow side effects (printing a value, updating a global variable, ...) in semantic rules. Such semantic rules are written as procedure calls or program fragments like

$$p(c_1, \ldots, c_k) \tag{2}$$

However these semantic rules can be thought of as rules defining the values of dummy attributes. An *attribute grammar* is a syntax-directed definition where the semantic rules cannot have side effects.

A S-attribute grammar is an attribute grammar where all attributes are synthesized attributes. The interest of an S-attribute grammar is that any parse tree can always be annotated by evaluating the semantic rules for the attributes at each node bottom up, from the leaves to the root.

An Annotated parse tree. is a parse tree showing the values of the attributes at each node. The process of computing the attribute values at the nodes is called *annotating* or *decorating* the parse tree.

Example 3 The following syntax-directed definition is from a desk calculator. This example is of course closely related to the detailed YACC example of the previous chapter. The terminals **number** and **n** stand respectively for an integer number and the newline character.

Grammar symbol	Synthesized attribute
E	$E.val \in \mathbb{Z}$
T	$T.val \in \mathbb{Z}$
F	$F.val \in \mathbb{Z}$
number	**number**$.lexval \in \mathbb{Z}$

Production	Semantic Rule
$L \longmapsto E\mathbf{n}$	$\mathsf{print}(E.val)$
$E \longmapsto E_1 + T$	$E.val := E_1.val + T.val$
$E \longmapsto T$	$E.val := T.val$
$T \longmapsto T_1 * F$	$T.val := T_1.val * F.val$
$T \longmapsto F$	$T.val := F.val$
$F \longmapsto (E)$	$F.val := E.val$
$F \longmapsto \mathbf{number}$	$F.val := \mathbf{number}.lexval$

Observations.

- All attributes are synthesized.
- The attribute **number**.*lexval* of the token **number** is assumed to be supplied by the lexical analyzer.
- The only semantic rule with side effect is $\mathsf{print}(E.val)$.

Example 4 The following syntax-directed definition implements statements for declaring identifiers **id** with type **real** and **int**. Note that the comma **,** is also a terminal symbol for this grammar. The addtype procedure is used to add the type of an identifier in the symbol table (by accessing its entry in the symbol table with **id**.*entry*).

Grammar symbol	Synthesized attribute	Inherited attribute
L		$L.in$ (string)
T	$T.type$ (string)	
id	$\mathbf{id}.entry \in \mathbb{Z}$	

Production	Semantic Rule
$D \longmapsto TL$	$L.in := T.type$
$T \longmapsto \mathbf{int}$	$T.type := \mathbf{int}$
$T \longmapsto \mathbf{real}$	$T.type := \mathbf{real}$
$L \longmapsto L_1,\mathbf{id}$	$L_1.in := L.in$
	$\mathsf{addtype}(\mathbf{id}.entry, L.in)$
$L \longmapsto \mathbf{id}$	$\mathsf{addtype}(\mathbf{id}.entry, L.in)$

Observations.

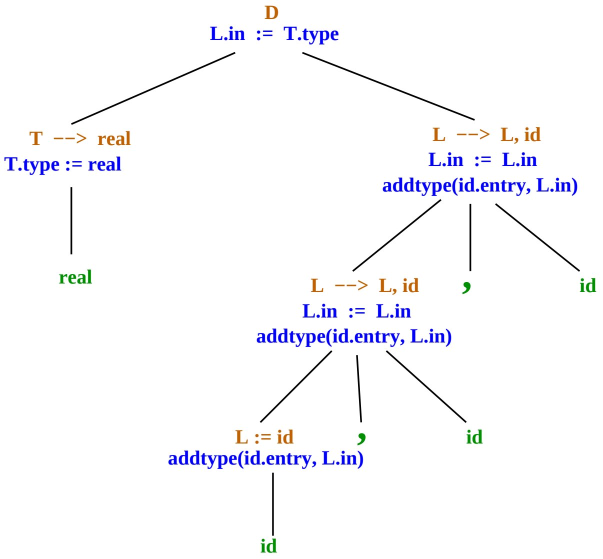

Figure 3: Parse tree for **real id**, **id**, **id**.

- The nonterminal T has a synthesized attribute whose value is received by the inherited attribute of L via the semantic rule $L.in := T.type$.
- Then the type $L.in$ is passed down a parse tree via the semantic rule $L_1.in := L.in$.

Remark 1 To compute the values of the attributes at each node of an annotated parse tree we need to define an evaluation order. This will be done by a dependency graph.

<u>DEPENDENCY GRAPH.</u> Let us a consider a syntax-directed definition. Let us assume that every semantic rule as the form

$$b \ := \ f(c_1, \ldots, c_k) \tag{3}$$

Let T be an annotated tree for this syntax-directed definition. Let N be an internal node of T and let $A \longmapsto \alpha$ be the associated production. Let $b := f(c_1, \ldots, c_k)$ be a semantic rule associated with this production. Then for $i = 1 \cdots k$ we say that the attribute b *depends* on the attribute c_i.

Let $\mathcal{A}$ be the set of all attributes. The underlying idea guiding the construction of the dependency graph of T is: if an attribute b the node N of T

depends on the attribute c_i then c_i must before evaluated before b. Thus, it is tempting to consider the graph of the relation

$$(b,\, c) \in \mathcal{A} \times \mathcal{A} \quad \longmapsto \quad b \text{ depends on } c \tag{4}$$

However a given grammar symbol X may label different nodes of T. Therefore the *dependency graph* of T is the directed graph constructed as follows.

Algorithm 1

Input: An annotated tree T

Output: The dependency graph G of T

```
for every node N of T repeat
   let X be the grammar symbol of N
   for every attribute X.b of X repeat
      create a vertex X.b(N) in the dependency graph
for every node N of T repeat
   for every semantic rule Y.b := f(Z.c, ...) attached to N repeat
      let M be the node of T (child or equal to N) labeled with Y.b
      for every parameter Z.c of f repeat
         let P be the node of T (child or equal to N) labeled with Z.c
         create an arc from Z.c(P) to Y.b(M)
```

Given a node N of the annotated tree T labeled by the grammar symbol X it is convenient to call *attribute* of N every attribute of X.

Example 5 Figures 4, 5, 6 and 7 show the construction of the vertices and the arcs of the dependency graph G of the parse tree of Figure 3. Observations.

- This graph has 10 vertices (shown on Figures 6 and 7 by rectangles with corners made round.
- The arcs of G are shown on Figure 7.
- An arc in the parse tree does not become necessarily an arc in the dependency graph.

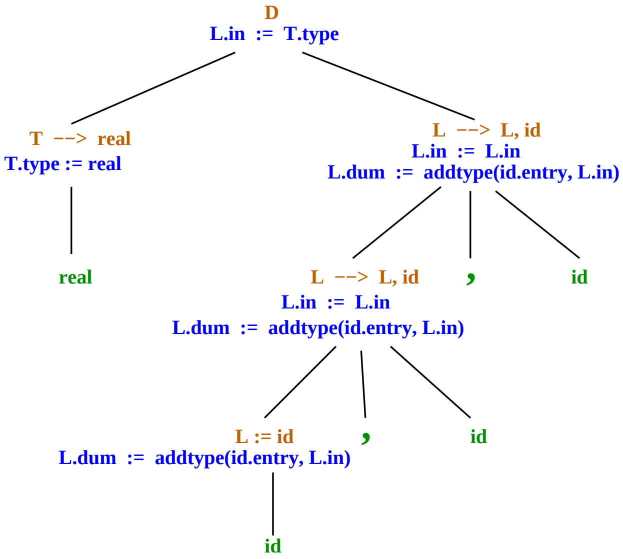

Figure 4: Parse tree for **real id**, **id**, **id** with dummy synthesized attributes.

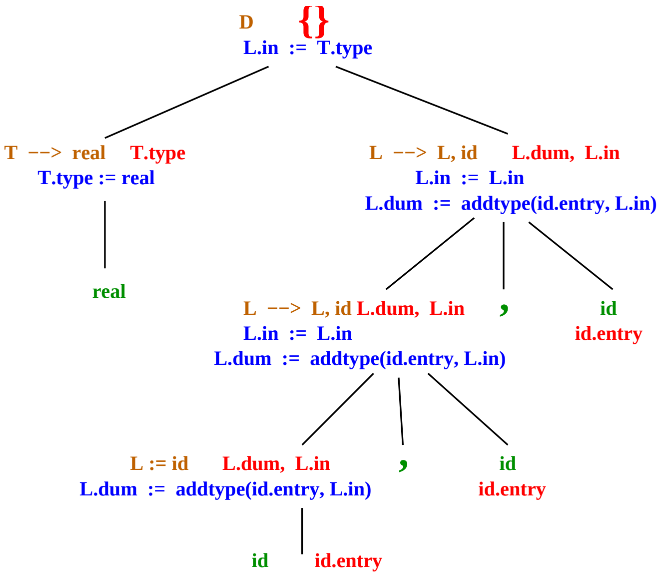

Figure 5: Parse tree for **real id**, **id**, **id** with dummy synthesized attributes and showing the attributes of each mode.

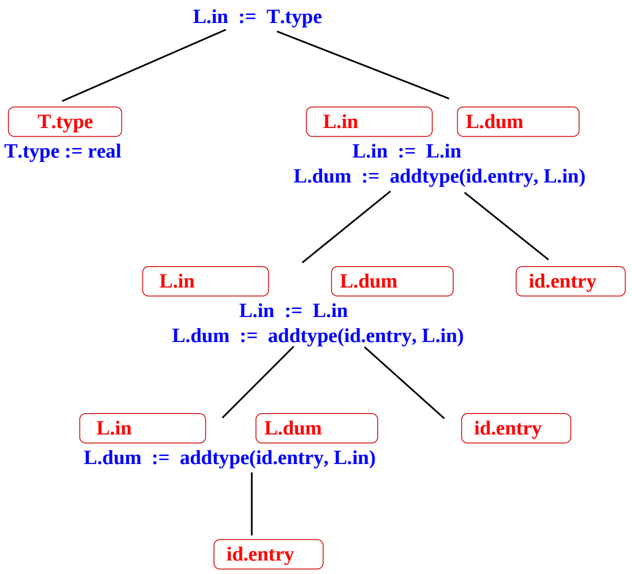

Figure 6: Construction of the vertices of the dependency graph.

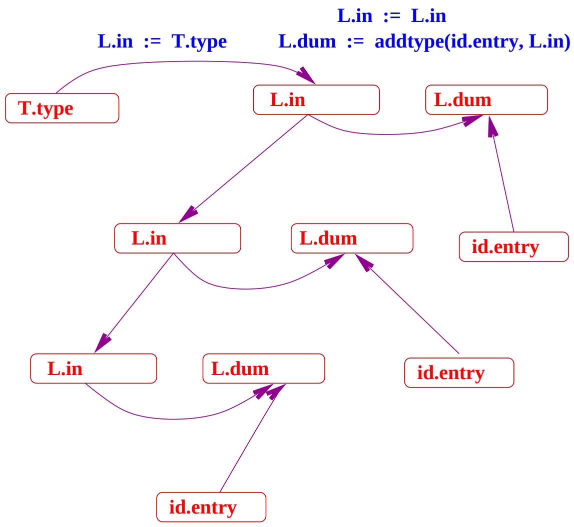

Figure 7: Construction of the arcs of the dependency graph.

- The dependency graph G has no circuits (or even cycles).

<u>DIRECTED GRAPHS.</u> let G be a directed graph with vertex set X and arc set A. The following definitions are very useful.

- A *path* in G is a sequence of vertices $(x_1, x_2, \ldots, x_n)$ such that for every $i = 1 \cdots n-1$ the couple (x_i, x_{i+1}) is an arc of G.
- A *circuit* in G is a path $(x_1, x_2, \ldots, x_n)$ such that $x_1 = x_n$.
- A *chain* in G is a sequence of vertices $(x_1, x_2, \ldots, x_n)$ such that for every $i = 1 \cdots n-1$ either the couple (x_i, x_{i+1}) or the couple (x_{i+1}, x_i) is an arc of G.
- A *cycle* in G is a chain $(x_1, x_2, \ldots, x_n)$ such that $x_1 = x_n$.
- A directed graph is said *acyclic* if it has no cycles. Such a graph is also called a DAG.

<u>SOURCES IN DIRECTED GRAPHS.</u> A *source* in the directed graph G is a vertex x such that for every other vertex y the couple (y, x) is not an arc. In broad words, there is no path leading to a source. It is not hard to see that every DAG has at least one source. (Otherwise there would be a circuit and thus a cycle).

<u>TOPOLOGICAL SORT OF A DAG.</u> Let $G = (X, A)$ be a DAG with n vertices. A *topological sort* of the DAG G is an enumeration of its vertices (a bijection from $\{1, \ldots, n\}$ to X) say $x_1, x_2, \ldots, x_n$ such that there is no path from x_j to x_i for $1 \leq i < j \leq n$.

By induction on n, it is not hard to see that every DAG has a topological sort. Indeed, the case $n = 1$ is trivial. For $n > 1$ let s be a source of G. Let $G - s$ be the graph induced from G by the cancellation of s and every arc leaving from s. By the induction hypothesis, the graph $G - s$ has a topological sort, say $x_1, x_2, \ldots, x_{n-1}$. Now observe that $s, x_1, x_2, \ldots, x_{n-1}$ is a topological sort of G.

<u>THE TRANSITIVE CLOSURE OF A DIRECTED GRAPH</u>

- The directed graph G is *transitive* if for every vertex x, y, z such that (x, y) and (y, z) are arcs of G then (x, z) is also an arc of G.

- The *transitive closure* $\overline{G}$ of the directed graph $G = (X, A)$ is the smallest graph (by its arc set) which is transitive and for which G is a subgraph. In other words, the graph $\overline{G}$ is such that
  - $\overline{G}$ is transitive,
  - every arc of G is an arc of $\overline{G}$,
  - and $\overline{G}$ is the smallest graph with these properties.
- Observe that for that two vertices x, y of G the couple (x, y) is an arc of $\overline{G}$ iff there is a path from x to y.
- Therefore the knowledge of $\overline{G}$ allow us to answer the following questions.
  1. Does G possess a circuit? If yes, a topological sort can be constructed.
  2. Is $\overline{G}$ antisymmetric? If both G and $\overline{G}$ have no loops (arcs of the form (x, x)) then $\overline{G}$ is antisymmetric. Hence $\overline{G}$ is the graph of a *pre-ordering* (that is a relation on the vertices which is antisymmetric and transitive). Then a sort algorithm (heap-sort, quick-sort, merge-sort) can be applied to sort the veritces of $\overline{G}$. This gives a topological sort for G.

Computing the transitive closure of a directed graph

- There are many techniques to compute $\overline{G}$. For instance we can compute the shortest length of a path between every couple of vertices. We describe now how to do this.
- Let G be a directed graph with n vertices that we denote by $1, 2, \ldots, n$.
- To every couple (i, j) of G we associate $\ell(i, j) \in \mathbb{N}$ the length of the arc (i, j), if (i, j) is an arc of G, and $+\infty$ otherwise.
- Then to every couple (i, j) we associate $L(i, j)$ the length of the shortest path from i to j, if any, and $+\infty$ otherwise.
- Let k, i, j be integers in the range $1 \cdots n$. We call k-path from i to j a path using intermediate vertices (excluding i and j) in the range $1 \cdots k$. Let $L^{(k)}(i, j)$ be the length of the shortest k-path from i to j. We define $L^{(0)}(i, j) = \ell(i, j)$.

- Then $L^{(n)}(i,j) = L(i,j)$. Moreover the matrices $L^{(1)}, L^{(2)}, \ldots, L^{(n)}$ can be computed iteratively by using the following relation for every $k = 1 \cdots n$

$$L^{(k)}(i,j) = \min(L^{(k-1)}(i,j), L^{(k-1)}(i,k) + L^{(k-1)}(k,j)) \tag{5}$$

  Indeed a shortest k-path from i to j is

  – either a shortest $(k-1)$-path from i to j,

  – or a shortest $(k-1)$-path from i to k followed by a shortest $(k-1)$-path from k to j.

- This leads to Algorithm 2.

Algorithm 2

Input: A directed graph G over $X = \{1, \ldots, n\}$ and arc lengths $\ell(i,j) \in \mathbb{N} \cup \{+\infty\}$ such that $\ell(i,j) = +\infty$ iff (i,j) is not an arc.

Output: $L^{(n)}(i,j)$ the shortest length of a path from i to j, if there is one, otherwise $+\infty$.

for $i = 1 \cdots n$ **repeat**
  for $j = 1 \cdots n$ **repeat**
    $L^{(0}(i,j) = \ell(i,j)$
for $k = 1 \cdots n$ **repeat**
  for $i = 1 \cdots n$ **repeat**
    for $j = 1 \cdots n$ **repeat**
      $L^{(k)}(i,j) = \min(L^{(k-1)}(i,j), L^{(k-1)}(i,k) + L^{(k-1)}(k,j))$.
return $L^(n)$.

<u>COMPUTING A TOPOLOGICAL SORT OF A DAG.</u> In our simple example we obtain a topological sort easily by *canceling* the sources one after another. This leads to Figure 8. Coming back to the parse tree (where the nodes without attributes are removed, for clarity) we obtain an order for evaluating the attributes. This leads to the annotated parse tree on Figure 9 where the order for executing the semantic rules is shown by increasing numbers. Let us denote by a_i the attribute associated with the i-th node in the topological sort of the dependency graph. We finally obtain the following program.

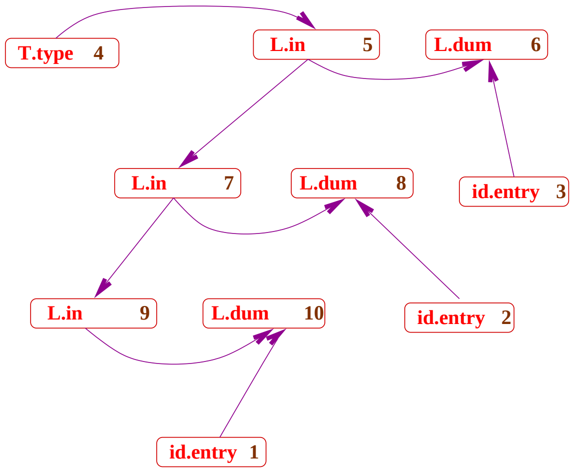

Figure 8: Topological sort for the dependency graph.

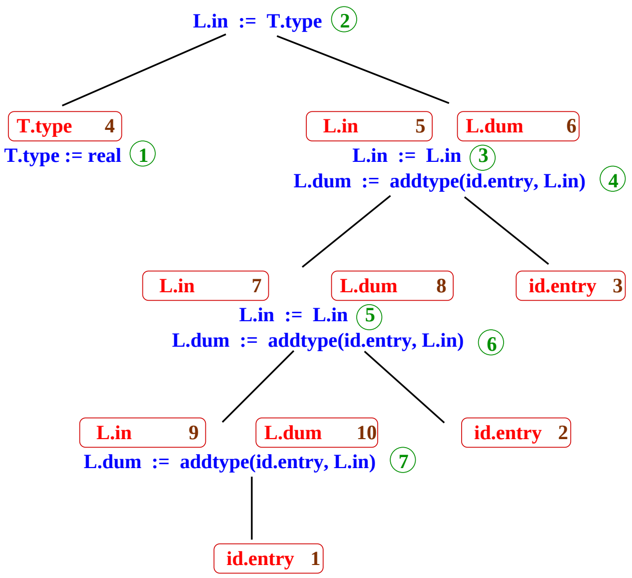

Figure 9: Annotated parse tree (where nodes without attributes are removed).

$$
\begin{aligned}
&a_4 := \mathbf{real} \\
&a_5 := a_4 \\
&a_7 := a_5 \\
&\mathsf{addtype}(\mathbf{id}_3.entry, a_5) \\
&a_9 := a_7 \\
&\mathsf{addtype}(\mathbf{id}_2.entry, a_7) \\
&\mathsf{addtype}(\mathbf{id}_1.entry, a_9)
\end{aligned}
$$

Remark 2 A grammar which is suitable for parsing (as the one of the detailed Example 4) may not be suitable for translation (i.e. may lead to an expensive translation scheme). In the following example, which also implements statements for declaring identifiers, inherited attributes are avoided. Thus the construction of a dependency graph is not needed.

Production	Semantic Rule
$D \longmapsto L$	
$T \longmapsto \mathbf{int}$	$T.type := \mathbf{int}$
$T \longmapsto \mathbf{real}$	$T.type := \mathbf{real}$
$L \longmapsto \mathbf{id},\ L_1$	$L.type := L_1.type$
	$\mathsf{addtype}(\mathbf{id}.entry, L_1.type)$
$L \longmapsto :T$	$L.type := T.type$

If for the same language we use the following grammar

$$
\begin{aligned}
&D \longmapsto L\colon T \\
&T \longmapsto \mathbf{int} \\
&T \longmapsto \mathbf{real} \\
&L \longmapsto L,\ \mathbf{id} \\
&L \longmapsto \mathbf{id}
\end{aligned}
$$

then we get into the same difficulties as in Example 4.

Example 6 The following syntax-directed definition implements statements for performing substitutions.

Grammar symbol	Synthesized attribute	Inherited attribute
S		
E	$E.val \in \mathbb{Z}$	$E.env$

- The attribute $E.val$ is an integer value

- The attribute $E.env$ is a list of *bindings* where a binding is a pair (**id**, *value*). Hence $E.env$ is a symbol table.

There are three functions used in the semantic rules.

initialEnv which returns the empty *environment*.

LookUp which determines the value of an identifier by consulting the environment.

Update which adds a binding to the environment.

In the above syntax-directed definition E_1, E_2 and E_3 correspond to the same nonterminal E.

Production	Semantic Rule
$S \longmapsto E$	$E.env := \mathsf{initialEnv}()$
$E_1 \longmapsto \textbf{let id} = E_2 \textbf{ in } E_3 \textbf{ end}$	$E_2.env := E_1.env$
	$E_3.env := \mathsf{Update}(E_1.env, \textbf{id}, E_2.val)$
	$E_1.val := E_3.val$
$E_1 \longmapsto (E_2 + E_3)$	$E_1.val := E_2.val + E_3.val$
	$E_2.env := E_1.env$
	$E_3.env := E_1.env$
$E \longmapsto \textbf{id}$	$E.val := \mathsf{LookUp}(\textbf{id}, E.env)$
$E \longmapsto \mathbf{1}$	$E.val := \mathbf{1}$
$E \longmapsto \mathbf{0}$	$E.val := \mathbf{0}$

Exercise. Translate the following statement.

$$\textbf{let id} = 1 \textbf{ in id} + \textbf{id end}$$

CONCLUSIONS

- We have provided a general method for evaluating semantic rules.
- To build an annotated tree with n nodes in its dependency graph the cost is $\mathcal{O}(n^3)$.
- We aim to reduce this cost by several *ad-hoc* techniques in the next sections.

2 Syntax Trees for Expressions

In this section, we consider the case where

- the input language consists of arithmetic expressions (in a general sense that we will make precise below) like arithmetic expressions in the usual sense, or regular expressions, or S-expressions, etc.
- the output intermediate representation for an input sentence is a tree reflecting the syntax structure of the sentence.

In this case,

- we will forbid inherited attributes,
- we will simplify the constructions of the parse tree and the annotated tree,
- and we will avoid the construction of the dependency graph.

<u>Arithmetic expressions.</u> A language L over an alphabet Σ is a language of *arithmetic expressions* (or *expressions* for short) if there exists

1. a partition $\Sigma = \Sigma_{op} \cup \Sigma_{val} \cup \{(,)\}$,
2. a grammar G generating L such that each production has one of the forms
   - $A \longmapsto a$ for $a \in \Sigma_{val}$
   - $A \longmapsto (B)$
   - $A \longmapsto \alpha_1\, \alpha_2, \ldots \alpha_n$ where $n \geq 2$ and where the α_i's are grammar symbols and such that at most one of them belongs to Σ_{op}, the other symbols being nonterminals.

The elements of Σ_{op} are called *operators* and those of Σ_{val} are called *values*. The nonterminals on the right side of a production where an operator appears are called *operands* of the production.

The language L is said

- *binary infix* if every production has the form $A \longmapsto a$ for $a \in \Sigma_{val}$, $A \longmapsto (B)$ or $A \longmapsto BfC$ where $f \in \Sigma_{op}$.

- *binary postfix* if every production has the form $A \longmapsto a$ for $a \in \Sigma_{val}$, $A \longmapsto (B)$ or $A \longmapsto BCf$ where $f \in \Sigma_{op}$.
- *binary prefix* if every production has the form $A \longmapsto a$ for $a \in \Sigma_{val}$, $A \longmapsto (B)$ or $A \longmapsto fBC$ where $f \in \Sigma_{op}$.

KEY IDEAS.

- The first idea is to condense the double construction *parse tree + annotated tree* in a single construction. For arithmetic expressions this is quite easy as we shall see
- The second idea is to avoid the duplications of nodes that stand for two (syntactically) equal subexpressions. For (arithmetic) expressions this is again quite easy by using dags.

CONSTRUCTING SYNTAX TREES FOR EXPRESSIONS. Each node in a syntax tree for an (arithmetic) expression is a record with several fields.

- In the node for an operator, one field identifies the operator and the remaining fields contain pointers to the nodes of the operands. The operator is often called the *label* of the node.
- In the leaf for an identifier, one field (the label) indicates that this is an identifier and another field is a pointer to the symbol-table entry for this identifier.
- In the leaf for a value, one field (the label) indicates that this is a value (may be this field gives the type of the value) and another field is the value (or a pointer to this value).

We use here the following functions to create the nodes of syntax trees for expressions with binary operators. Each function returns a pointer to a newly created node.

make_node (op, $left$, $right$) creates an *operator node* with label op and two fields containing pointers to $left$ and $right$.

make_leaf (**id**, $entry$) creates an *identifier leaf* with label **id** and a field containing $entry$ a pointer to the symbol-table $entry$ for the identifier.

make_leaf (**num**, *val*) creates a *number leaf* with label **num** and a field containing *val*.

Example 7 The following syntax-directed definition allows the construction of a syntax tree for a binary infix arithmetic expression. The synthesized attributes $E.ptr$, $T.ptr$ and $F.ptr$ keep track of the pointers returned by the function calls.

Production	Semantic Rule
$E \longmapsto E_1 + T$	$E.ptr := \mathsf{make\_node}(+, E_1.ptr, T.ptr)$
$E \longmapsto E_1 - T$	$E.ptr := \mathsf{make\_node}(-, E_1.ptr, T.ptr)$
$E \longmapsto T$	$E.ptr := T.ptr$
$T \longmapsto T_1 * F$	$T.ptr := \mathsf{make\_node}(*, T_1.ptr, F.ptr)$
$T \longmapsto F$	$T.ptr := F.ptr$
$F \longmapsto (E)$	$F.ptr := E.ptr$
$F \longmapsto \mathbf{id}$	$F.ptr := \mathsf{make\_leaf}(\mathbf{id}, \mathbf{id}.entry)$
$F \longmapsto \mathbf{num}$	$F.ptr := \mathsf{make\_leaf}(\mathbf{num}, \mathbf{num}.val)$

USING DAGS FOR SYNTAX TREES OF EXPRESSIONS.

- A dag for an expression identifies the common subexpressions in the expression.
- Like a syntax tree, a dag has a node for every subexpression of the expression.
- The difference is that a node in a dag representing a common subexpression has more than one parent.
- Thus, before constructing a node with label *op* and fields with pointers to $left$ and $right$, the operation make_node(*op*, $left$, $right$) checks if such a node has already been constructed.

IMPLEMENTING DAGS FOR SYNTAX TREES OF EXPRESSIONS.

- To construct dags for expressions in an efficient manner one could use an array of records.
- To avoid (explicit) pointers we can refer to a node by its *index* (or position) in this array of records.

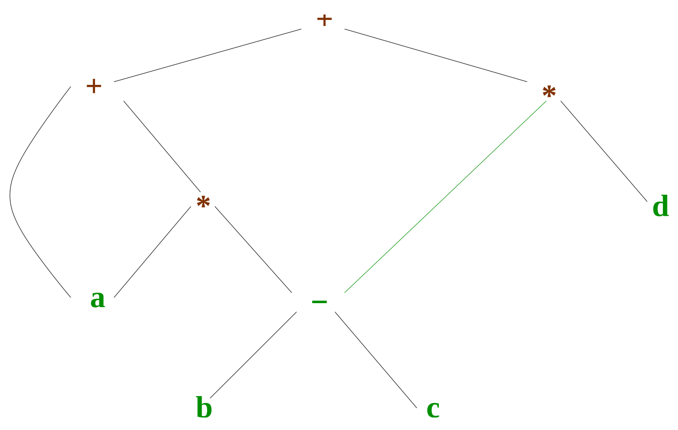

Figure 10: A dag for the expression $a + a * (b - c) + (b - c) * d$.

- Each record implements a node (or leaf). For an operator node this record will contain
  - the label of the operator,
  - the indices of the children.
- This kind of representations allows search for existing subexpressions in linear time.
- This can be improved by using hash-tables.
  - A hash-table will be in our case an array of *buckets* where a bucket is a linked list of nodes (each node being represented by a record as in the previous data structure).
  - We assume that we are given a *hash-function* h which converts a label into a non-negative machine integer.
  - To perform make_node(op, $left$, $right$) we first compute $h(op)$ which returns the *bucket number* in which to search the node (and may be store the node if not yet present).
  - In practice we expect each bucket to contain a small number of nodes.
  - If α is the expected maximum length of a bucket, then search/insertion of nodes can be done in $\mathcal{O}(\alpha)$ rather than in $\mathcal{O}(n)$ where n is the number of nodes currently stored.

3 Translation Schemes

In this section, we enhance the notion of a syntax-directed definitions in order to specify the order of evaluation of the semantic rules, leading to the concept of a translation scheme.

KEY FEATURES.

- For each sentence of the underlying grammar, a parse tree (decorated with attributes and semantic rules) is constructed.
- But a dependency graph is not computed.

Translation schemes are well adapted when the semantic rules are procedure calls.

A TRANSLATION SCHEME is a context-free grammar in which semantic rules are embedded within the right sides of the productions.

- So a translation scheme is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly shown.
- The position at which an action is to be executed
  - is shown by enclosing it between braces
  - and writing it within the right side of the production.
- When drawing a parse tree for a translation scheme, an action is indicated by constructing an extra child, connected by a dashed line to the node of the production.
- The order in which the actions are executed is the order in which they appear during a depth-first traversal of a parse tree.

Example 8 Let L be a language of arithmetic expressions over the alphabet

$$\Sigma \; = \; \Sigma_{op} \cup \Sigma_{val} \cup \{(,)\}. \tag{6}$$

where Σ_{op} is the set of operators and Σ_{val} the set of values. We assume that L is binary infix. We associate to L a language L' of arithmetic expressions, binary postfix. This is given by a map p defines as follows.

- for $a \in \Sigma_{val}$ we have $p(a) = a$

- for nonterminals B we have $p(\,(B)\,) = B$.
- for nonterminals B, C and $f \in \Sigma_{op}$ we have $p(B\,f\,C) = B\,C\,f$.

The translation from L to L' can be made by a syntax-directed definition of the form

Production	Semantic Rule
$A \longmapsto a$	$A.str := a$
$A \longmapsto (B)$	$A.str := B.str$
$A \longmapsto B\,f\,C$	$A.str :=$ concat$(B.str,\, C.str,$ " f")

where $A.str$, $B.str$ and $C.str$ are synthesized attributes of type string and where concat is a concatenation function for strings. Since this syntax-directed definition has no inherited attributes, the evaluation of the attributes (and consequently the translation) will be easy. However, observe that for a large expression, a certain amount of time and space will be needed to create all the attributes.

It is possible to speed up translations from L to L' by considering the following translation scheme.

A	$\longmapsto$	a { output(a) }
A	$\longmapsto$	(B) { }
A	$\longmapsto$	$B\,f\,C$ { output(f) }

We illustrate the notions of syntax tree and translation scheme with the expression $a + (a + d) * (b - c)$.

- Figure 11 shows a syntax tree for this expression. Observe that there is no need to show parenthesis on a syntax tree since there is no ambiguity for evaluating the expression represented by this tree.
- Figure 12 shows a parse tree for $a+(a+d)*(b-c)$. This tree has more nodes and arcs since it shows the productions used.
- Figure 13 shows the previous parse tree decorated by means of the above translation scheme.

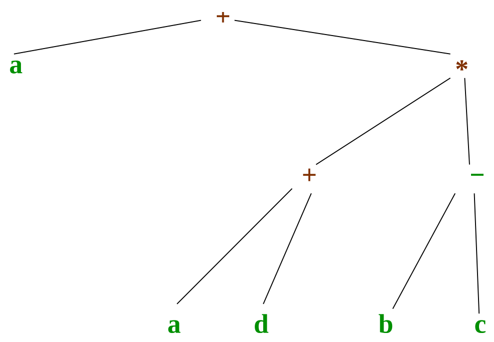

Figure 11: A syntax tree for the expression $a + (a + d) * (b - c)$.

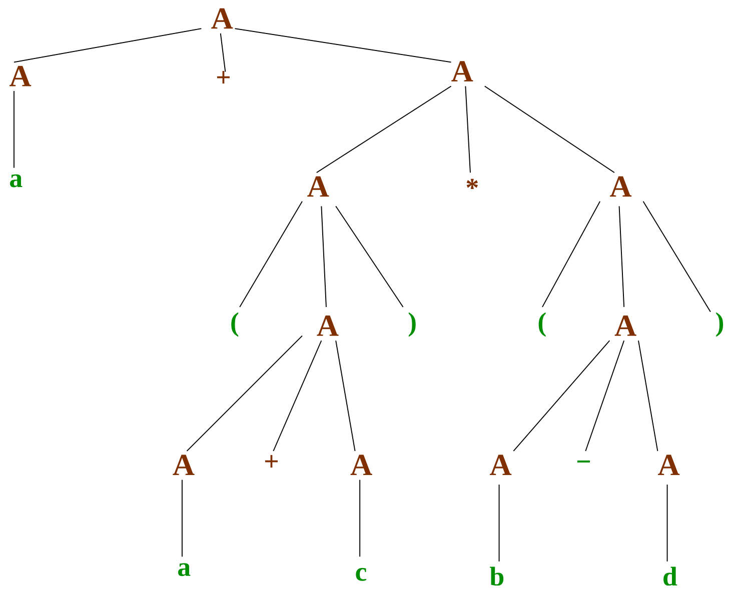

Figure 12: A parse tree for the expression $a + (a + d) * (b - c)$.

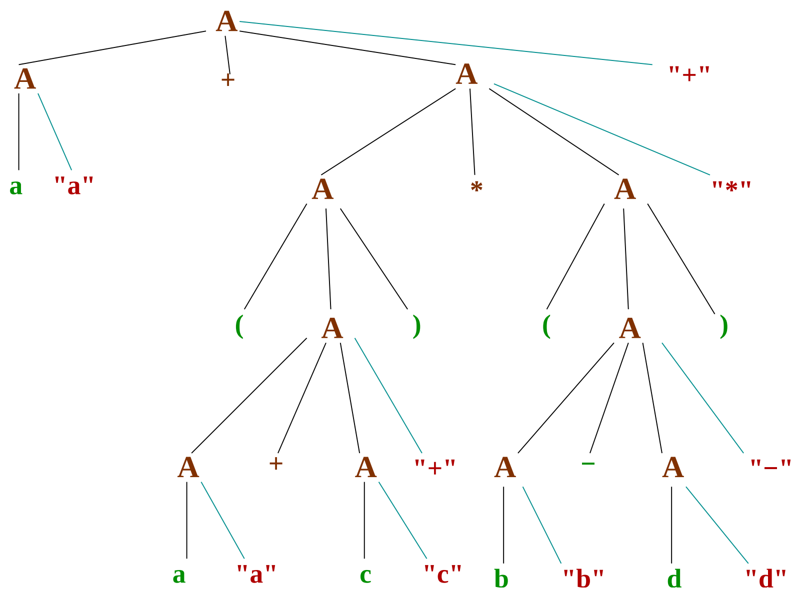

Figure 13: A parse tree decorated with translation rules for the expression $a + (a + d) * (b - c)$.

4 L-attributed Definitions

As in the previous section, we enhance the notion of syntax-directed definitions in order to specify the order of evaluation of the semantic rules. This leads to the concept of a L-attributed definition.

<u>Key ideas.</u> Both notions of *translation scheme* and *L-attributed definition* are close. However

- the later provides more care for the treatment of inherited attributed, leading to a more powerful theoretical concept.
- whereas the former is closer to a true computer program, leading to a more practical concept.

<u>The depth-first evaluation order.</u> We consider a syntax-directed definition S and a parse tree T for S showing the attributes of the grammar symbols of T. Figure 1 shows an example of such a tree. Algorithm 3 provides an order, called *depth-first evaluation order*, for evaluating attributes shown by T.

Algorithm 3

Input: A node n in a parse tree T.

Output: An evaluation order for the attributes of the subtree of T rooted at n.

```
procedure depth_first(n);
begin
    for every child m of n from left to right do begin
      evaluate the inherited attributes of m;
      depth_first(m);
   end
   evaluate the synthesized attributes of n;
end.
```

Remark 3 We now introduce a class of syntax-directed definitions, called L-attributed definitions, whose attributes can always be evaluated in depth-first order.

- the *L-attributed* stands for *one pass from left-to-right.*
- Intuitively, there are no right-to-left dependencies between attribute occurrences in the productions.
- L-attributed definitions include all syntax-directed definitions based on $LL(1)$ grammars. We will admit this statement.

L-ATTRIBUTED DEFINITIONS. A syntax-directed definition is *L-attributed* if for each production $A \longmapsto X_1 \cdots X_n$, for each $j = 1 \cdots n$, each inherited attribute of X_j depends only

- the attributes of $X_1 \cdots X_{j-1}$,
- the inherited attributes of A.

Theorem 1 *The attributes of an L-attributed definition can always be evaluated in depth-first order.*

PROOF $\triangleright$ By induction on the depth of a parse tree T. Observe that a parse tree has at least depth 1 and two nodes.

If T has depth 1, then its root is labeled by S and the associated production must have the form $S \longmapsto a_1 \cdots a_n$ where $a_1, \ldots, a_n$ are terminals. Recall that terminals do not have inherited attributes. Observe that every synthesized attribute of a terminal must be known in advance since semantic rules are only associated with productions (and thus cannot define a synthesized attribute of a terminal). Moreover, observe also that any inherited attribute of S must be known in advance, too. Now Algorithm 3 computes the synthesized attributes of S and thus evaluates all the attributes in T.

Assume now that T has depth $d > 1$. The production associated with the root has the form $S \longmapsto X_1 \cdots X_n$ where $X_1, \ldots, X_n$ are grammar symbols. Each subtree rooted at a child of S can be viewed as a parse tree of a L-attributed definition. However the root of each of these subtrees may have inherited attributes that must be computed (we cannot assume that they are some known constants).

Let j be in $1 \cdots n$. By virtue of the induction hypothesis and by virtue of Algorithm 3 we can assume that all attributes of $X_1, \ldots, X_{j-1}$ are known Since the inherited attributes of S must be known constants, by definition of an L-attributed syntax-directed definition we can compute the inherited attributes of X_j. When all inherited attributes of $X_1, \ldots, X_n$ are computed, then it is clear that all synthesized attributes of S can be computed. Therefore Algorithm 3 evaluates all the attributes in T. $\triangleleft$

Example 9 The syntax-directed definition of the **let** language is an L-attributed syntax-directed definition.

Production	Semantic Rule
$S \longmapsto E$	$E.env := \mathsf{initialEnv}()$
$E_1 \longmapsto$ **let id** $=$ E_2 **in** E_3 **end**	$E_2.env := E_1.env$
	$E_3.env := \mathsf{Update}(E_1.env, \mathbf{id}, E_2.val)$
	$E_1.val := E_3.val$
$E_1 \longmapsto (E_2 + E_3)$	$E_1.val := E_2.val + E_3.val$
	$E_2.env := E_1.env$
	$E_3.env := E_1.env$
$E \longmapsto$ **id**	$E.val := \mathsf{LookUp}(\mathbf{id}, E.env)$
$E \longmapsto$ **1**	$E.val := \mathbf{1}$
$E \longmapsto$ **0**	$E.val := \mathbf{0}$

However the following syntax-directed definition is not L-attributed.

Production	Semantic Rule
$A \longmapsto L\,M$	$L.i := f_1(A.i)$
	$M.i := f_2(L.s)$
	$A.s := f_3(M.s)$
$A \longmapsto Q\,R$	$R.i := f_4(A.i)$
	$Q.i := f_5(R.s)$
	$A.i := f_6(Q.s)$

Indeed the inherited attribute $Q.i$ depends on the attribute $R.s$ although R appears on the right of Q in the production $A \longmapsto Q\,R$.

FROM L-ATTRIBUTED DEFINITIONS TO TRANSLATION SCHEMES. Every L-attributed definition can be converted into a translation scheme that will always succeed in evaluating the attributes in a parse tree, provided that the following rules are observed. Let $A \longmapsto X_1 \cdots X_n$ be any production and j be in the range $1 \cdots n$.

Rule 1. The inherited attributes of X_j must be computed in actions occurring to the left of X_j in the translation. (Indeed they may be needed when visiting the subtree rooted at X_j.)

Rule 2. An action can refer to a synthesized attribute $X_j.s$ of X_j only if X_j appears to the left of this action. (Indeed $X_j.s$ will be computed when visiting the subtree rooted at X_j.)

Rule 3. A synthesized attribute of A should be computed in an action occurring at the end of the translation. (Indeed all attributes of $X_1, \ldots, X_n$ may be needed.)

Example 10 The following translation scheme does **not** satisfy **Rule 1**.

S	$\longmapsto$	$A_1\,A_2\ \{A_1.in := 1;\ A_2.in := 1\}$
A	$\longmapsto$	$a\ \{\mathsf{print}(A.in)\}$

Indeed consider the parse tree T of $w = aa$ decorated with the necessary attributes and semantic rules. A depth-first traversal of T will call $\mathsf{print}(A_1.in)$ and $\mathsf{print}(A_2.in)$ before the attributes $A_1.in$ and $A_2.in$ are assigned by the semantic rules $A_1.in := 1;\ A_2.in := 1$. In fact the above translation scheme needs to be changed to

$S \longmapsto \{A_1.in := 1;\ A_2.in := 1\}\ \ A_1\, A_2$
$A \longmapsto a\ \ \{\mathsf{print}(A.in)\}$

Example 11 Consider the following syntax-directed definition for the point size and height of boxes containing mathematical formulas.

Production	Semantic Rule
$S \longmapsto B$	$B.ps := 10$
	$S.ht := B.ht$
$B \longmapsto B_1\, B_2$	$B_1.ps := B.ps$
	$B_2.ps := B.ps$
	$B.ht := \mathsf{max}(B_1.ht, B_2.ht)$
$B \longmapsto B_1\, \textbf{sub}\, B_2$	$B_1.ps := B.ps$
	$B_2.ps := \mathsf{shrink}(B.ps)$
	$B.ht := \mathsf{disp}(B_1.ht, B_2.ht)$
$B \longmapsto \textbf{text}$	$B.ht := \textbf{text}.h \times B.ps$

Some comments.

- The nonterminal B represents a *box* which has been assigned to a mathematical formula.
- The production $B \longmapsto B\, B$ represents the juxtaposition of two boxes.
- When $B \longmapsto B_1\, B_2$ is applied,
  - the boxes B_1 and B_2 inherit the *point size* of B,
  - and the *height* of B represented by the synthesized attribute $B.ht$ is given by $\mathsf{max}(B_1.ht, B_2.ht)$.
- The production $B \longmapsto B_1\, \textbf{sub}\, B_2$ represents a subscripted expression.
- When $B \longmapsto B_1\, \textbf{sub}\, B_2$ is applied,
  - the $\mathsf{shrink}(B.ps)$ returns a fraction of the $B.ps$, say 30%,
  - and $\mathsf{disp}(B_1.ht, B_2.ht)$ allows for downward displacement of the box B_2 and computes the height of B.

Consider the production $S \longmapsto B$ together with its semantic rules $B.ps := 10$ and $S.ht := B.ht$.

- **Rule 1** implies that the action $\{B.ps := 10\}$ must occur before B is visited.
- **Rule 2** and **Rule 3** imply that the action $\{S.ht := B.ht\}$ must occur after B is visited.

Consider the second production $B \longmapsto B_1\, B_2$ together with its semantic rules $B_1.ps := B.ps$, $B_2.ps := B.ps$ and $B.ht := \mathsf{max}(B_1.ht, B_2.ht)$.

- **Rule 1** implies that the actions $\{B_1.ps := B.ps\}$ and $\{B_2.ps := B.ps\}$ must occur before B_1 and B_2 are visited respectively.
- **Rule 2** and **Rule 3** imply that the action $\{B.ht := \mathsf{max}(B_1.ht, B_2.ht)\}$ must occur after B_1 and B_2 are visited.

Similar considerations with the other two productions lead to the following translation scheme.

S	$\longmapsto$		$\{B.ps := 10\}$
		B	$\{S.ht := B.ht\}$
B	$\longmapsto$		$\{B_1.ps := B.ps\}$
		B_1	$\{B_2.ps := B.ps\}$
		B_2	$\{B.ht := \mathsf{max}(B_1.ht, B_2.ht)\ \}$
B	$\longmapsto$		$\{B_1.ps := B.ps\}$
		B_1	
		sub	$\{\ B_2.ps := \mathsf{shrink}(B.ps)\ \}$
		B_2	$\{\ B.ht := \mathsf{disp}(B_1.ht, B_2.ht)\ \}$
B	$\longmapsto$	**text**	$\{\ B.ht := \mathbf{text}.h \times B.ps\ \}$

Example 12 To conclude this section we give a translation scheme for the **let** language

S	$\longmapsto$		$\{\ E.env := \mathsf{initialEnv}()\ \}$
		E	
E_1	$\longmapsto$		$\{\ E_2.env := E_1.env\ \}$
		let id $= E_2$	$\{\ E_3.env := \mathsf{Update}(E_1.env, \mathbf{id}, E_2.val)\ \}$
		in E_3 **end**	$\{\ E_1.val := E_3.val\ \}$
E_1	$\longmapsto$		$\{\ E_2.env := E_1.env\ \}$
			$\{\ E_3.env := E_1.env\ \}$
		$(E_2 + E_3)$	$\{\ E_1.val := E_2.val + E_3.val\ \}$
E	$\longmapsto$	**id**	$\{\ E.val := \mathsf{LookUp}(\mathbf{id}, E.env)\ \}$
E	$\longmapsto$	**1**	$\{\ E.val := \mathbf{1}\ \}$
E	$\longmapsto$	**0**	$\{\ E.val := \mathbf{0}\ \}$

5 Implementing translation schemes with YACC

<u>WRITING TRANSLATION SCHEMES WITH YACC.</u>

In all our previous examples with YACC we were

- evaluating synthesized attributes
- by putting actions at the end of each production rule.

This works because a YACC parser is a bottom-up parser. But in fact YACC can do more.

- A YACC program (where productions are embedded with actions) is a **translation scheme.**
- Thus, a YACC parser evaluates the actions by a **depth-first traversal** of the parse tree.

<u>HOW YACC HANDLES PRODUCTIONS EMBEDDED WITH ACTIONS?</u> An action occurring inside the right side of a production

- can be replaced by a new *dummy* non-terminal
- that generates the *empty string*
- and produces the corresponding action.

For instance the following translation scheme fragment

$$\boxed{A \quad \longmapsto \quad B_1 \; \{e_1\} \; B_2 \; \{e_2\} \; B_3 \; \{e_3\}}$$

is equivalent to

$$\begin{array}{|lcl|}\hline A & \longmapsto & B_1 \; C_1 \; B_2 \; C_2 \; B_3 \; C_3 \\ C_1 & \longmapsto & \varepsilon \; \{e_1\} \\ C_2 & \longmapsto & \varepsilon \; \{e_2\} \\ C_3 & \longmapsto & \varepsilon \; \{e_3\} \\ \hline\end{array}$$

In YACC

- the parser implemented in YACC requires that the actions occur at the end of the right side of the productions.

- Then, for actions occurring inside the right side, YACC automatically generates *marker non-terminals* generating the empty word. Hence

```
thing:   A { printf("seen on A"); } B ;
```

is automatically transformed into something like

```
thing:      A fakename B ;
fakename:   /* empty */  {printf("seen on A"); } ;
```

- Therefore, YACC interprets an embedded action as a symbol in the rule. So one can have:

```
thing:   A { $$ = 17; } B C { printf("%d",$2); }
```

where \$1, \$2, \$3, \$4 refers respectively to A, $\{\cdots\}$, B, C.

<u>PASSING IDENTIFIER TYPES</u> Consider the following translation scheme.

D	$\longmapsto$	$T\ \{L.in := T.type\}\ L$
T	$\longmapsto$	$\mathbf{int}\ \{T.type := \mathbf{int}\}$
T	$\longmapsto$	$\mathbf{real}\ \{T.type := \mathbf{real}\}$
L	$\longmapsto$	$\{\ L_1.in := L.in\ \}\ L_1,\mathbf{id}$
		$\{\ \mathsf{addtype}(\mathbf{id}.entry, L.in)\ \}$
L	$\longmapsto$	$\mathbf{id}\ \{\ \mathsf{addtype}(\mathbf{id}.entry, L.in)\ \}$

Consider the following sentence generated by the above grammar.

real p, q, r

A bottom-up parse of this sentence will result in applying *backwards* the rules below in this order:

1. $T \longmapsto \mathbf{real}$
2. $L \longmapsto \mathbf{id}$
3. $L \longmapsto L, \mathbf{id}$
4. $L \longmapsto L, \mathbf{id}$

5. $D \longmapsto TL$

When

- considering $L \longmapsto \mathbf{id}$ or $L \longmapsto L, \mathbf{id}$
- together with the action $\mathsf{addtype}(\mathbf{id}.entry, L.in)$

one needs the inherited attribute $L.in$. Observe that

- $L.in$ was computed just before considering $L \longmapsto \mathbf{id}$ or $L \longmapsto L, \mathbf{id}$.
- So it will be the immediate **previous** attribute computed by a depth-first traversal.
- YACC refers to it as \$-1
- $\$n$ with $n = -2, -3 \ldots$ can be used also.
- If a production is embedded with more than one actions $a_1, a_2, \ldots, a_n$ and if an action a_i with $i < n$ defines \$\$ then a_j with $j > i$ can refer to it as \$0.

PASSING INHERITED ATTRIBUTES. In a situation like

S	$\longmapsto$	$aA \ \{C.i := A.s\} \ C$
S	$\longmapsto$	$bAB \ \{C.i := A.s\} \ C$
C	$\longmapsto$	$c \ \{C.s := g(C.i)\}$

In a YACC program for this translation scheme how to access to $C.i$

- in the third rule
- if we do not know which rule was applied before?

The solution is to use a *dummy* nonterminal, called a marker. We change the above translation scheme to

S	$\longmapsto$	$aA \ \{C.i := A.s\} \ C$
S	$\longmapsto$	$bAB \ \{M.i := A.s\} \ M \ \{C.i := M.s\} \ C$
M	$\longmapsto$	$\varepsilon \ \{M.s := M.i\}$
C	$\longmapsto$	$c \ \{C.s := g(C.i)\}$

Then a YACC program for the above translation scheme will look like:

```
S       : a A C
        | b A B M C
        ;
M       :   /* empty */  {$$ = $-2}
        ;
C       :  c {$$ = g($-1)} ;
```