Chapter 2

ATTRIBUTED TRANSLATIONS*

P. M. LEWIS, D. J. ROSENKRANTZ AND R. E. STEARNS
General Electric Company, Research and Development Center, Schenectady, NY 12345, USA

Reprinted from: *J. Computer and System Sciences*, Vol. 9, No. 3, Dec. 1974, pp. 279–307.

Abstract Attributed translation grammars are introduced as a means of specifying a translation from strings of input symbols to strings of output symbols. Each of these symbols can have a finite set of attributes, each of which can take on a value from a possibly infinite set. Attributed translation grammars can be applied in depth to practical compiling problems.

Certain augmented pushdown machines are defined and characterizations are given of the attributed translations they can perform both deterministically and nondeterministically. Classes of attributed translation grammars are defined whose translation can be performed deterministically while parsing top down or bottom up.

Received August 30, 1973.

1. Introduction

The purpose of this paper is to develop the concept of an "attributed translation," particularly attributed translations which can be described in a syntax-directed manner. The theory is developed with a particular application in mind, namely the specification of input-output relations of language processing devices such as the lexical and syntax boxes of a compiler. This application is reflected in our choice of mathematical terminology and in our illustrative examples.

* A preliminary version of this paper was presented at the 1973 Fifth Annual ACM Symposium on the Theory of Computing.

S.S. Ravi, S.K. Shukla (eds.), *Fundamental Problems in Computing*,

The concept underlying the mathematics of this paper is the concept of an *attributed symbol*. A set of attributed symbols is specified by giving a finite set of basic symbols, a finite set of attributes for each basic symbol, and a set (possibly infinite) of values for each attribute. A particular attributed symbol consists of a basic symbol together with an associated attribute value for each attribute. Our customary notation is to display the attribute values as subscripts of the basic symbol. Our customary interpretation is that the values are "semantic" information associated with a particular occurrence of a basic symbol. Suppose, for example, that one basic symbol is the symbol CONSTANT specified to have one attribute and suppose it is specified that the attribute can take any integer as its value. Then the attributed symbol consisting of the basic symbol CONSTANT with associated attribute value 37 would be written CONSTANT_{37}. In a particular application, the subscript might be interpreted as semantic information giving the numerical value of a constant. (In other applications, an attribute might be interpreted as a pointer to a symbol table entry.)

By an "attributed translation," we mean a mapping of certain strings of attributed "input symbols" (i.e. an input language) into strings of attributed "action symbols." The terminology "action symbol" is in deference to the interpretation that an action symbol represents the performance of an arbitrary semantic action. In the simple applications illustrating this paper, the semantic actions are simply to emit a corresponding output. Thus for purposes of understanding this paper, it is satisfactory to think of the action symbols as "output symbols."

The attributed translations studied in this paper are translations that can be described by a grammar we call an "attributed translation grammar," which is a generalization of context-free grammar. The generalization is achieved in two steps. First a context-free grammar is generalized to a "translation grammar" describing translations without attributes. Then the attributes are added.

After considering attributed translation grammars as a means of specifying translations, we concentrate on performing these translations with augmented pushdown machines. Characterizations are given of the attributed translations that can be performed by both nondeterministic and deterministic augmented pushdown machines. Certain classes of attributed translation grammars are defined whose specified translation can always be performed by a deterministic augmented pushdown machine while parsing top down or bottom up.

Attributed translations are based on the ideas of attributed grammars [10] and syntax directed translations [7, 12]. The computation of attributes is also considered in [1]. Other relevant concepts are property grammars and table machines [14], attributed grammars with relations [3], and affix grammars [2, 11].

2. Translation Grammars

We begin by introducing a new mechanism, called a translation grammar. The translation grammar concept is introduced as a way of specifying translations of input strings (without attributes) into action or output symbol strings (without attributes).

A *translation grammar* is a context free grammar in which the set of terminal symbols is partitioned into a set of *input symbols* and a set of *action symbols*. The strings in the language generated by a translation grammar are called *activity sequences*. The *input grammar* of a translation grammar is the grammar obtained by deleting all action symbols from the productions of the given grammar.

Given an activity sequence of input and action symbols, we use the term *input part* to refer to the sequence of input symbols obtained from the activity sequence by deleting all action symbols and we use the term *action part* to refer to the sequence of action symbols obtained from the activity sequence by deleting all input symbols. For each activity sequence, the action part is called a *translation* of the input part.

Given a translation grammar, each activity sequence in the language defined by that grammar pairs an input part with an action part. The set of all pairs that can be obtained in this way is called the *syntax directed translation* defined by that translation grammar.

The set of translations defined by translation grammars is exactly the same set as defined by the simple syntax directed transductions of [12], because the translation grammar provides an alternate notation for indicating "simple transduction elements." However, the activity sequence is a new mathematical object amenable to theoretical study. In practice, an activity sequence can be interpreted as a scenario specifying the operation of a language processor. An occurrence of an input symbol in an activity sequence can be interpreted (roughly) as the reading of that symbol by the processor. The occurrence of an action symbol in an activity sequence can be interpreted as the emitting of that symbol by the processor. Alternatively, the action symbols can be interpreted as the names of action (or semantic) routines that are to be called while processing the input sequence. The activity sequence can thus be interpreted as specifying both the sequence of action routine calls (or emitting of symbols) corresponding to the input sequence, and the timing of these action routine calls with respect to reading the input symbols.

The primary use of translation grammars in this paper is as a vehicle for describing translations.

3. Attributed Translations

We now start the study of translations where the input and action symbols have associated attributes. As an aid to understanding the objectives of the theory, we begin with an English description of a particular language processor.

> The input set of the processor is the set
>
> $$\{(,), +, *, C\}$$
>
> where C represents a constant. Furthermore, each occurrence of input C presented to the processor is accompanied by information giving the value of that constant. The processor accepts input sequences which constitute valid arithmetic expressions and emits the numerical value of the input expression.

To model the input of this processor as a string of attributed input symbols, we simply treat the value of the constant as an attribute. Under our convention that attributes are shown as subscripts, one of the permissible attributed input strings is

$$(C_2 + C_5) * (C_{11} + C_3)$$

To model the output activity of the processor, we invent the symbol ANSWER to represent the action of emitting the answer. We let ANSWER have an attribute which is to be the numerical answer emitted. The action sequence corresponding to the above input sequence would therefore be

$$\text{ANSWER}_{98}$$

In the next section, we present a method of describing certain attributed translations in a grammatical way. It will then be possible to replace the above English description of a processor with a precise grammatical specification of its input-output relation. In later sections, we show how suitable grammatical specifications can be used to obtain processors for performing the specified attributed translation.

4. Attributed Translation Grammars

We now generalize translation grammars to accommodate attributes. Each symbol in the translation grammar (input, nonterminal or action symbol) is allowed to have attributes. Rules are then given by which values for the attributes of all the symbols on a derivation tree can be computed.

An *attributed translation grammar* is a translation grammar for which the following additional specifications are made.

1. Each input, nonterminal, and action symbol has an associated finite set of attributes, and each attribute has a (possibly infinite) set of permissible values.

2. Each nonterminal and action symbol attribute is classified as being either *inherited* or *synthesized*.

3. Rules for inherited attributes are specified as follows.

   (a) For each occurrence of an inherited attribute on the right-hand side of a given production, there is an associated rule which says how to compute a value for that attribute as a function of certain other attributes of symbols occurring in the left- or right-hand sides of the given production.

   (b) An initial value is specified for each inherited attribute of the starting symbol.

4. Rules for synthesized attributes are specified as follows.

   (a) For each occurrence of a synthesized nonterminal attribute on the left-hand side of a given production, there is an associated rule which says how to compute a value for that attribute as a function of certain other attributes of symbols occurring in the left- or right-hand sides of the given production.

   (b) For each synthesized action symbol attribute, there is an associated rule which says how to compute a value for that attribute as a function of certain other attributes of the action symbol.

Attributed translation grammars are to be used to define attributed derivation trees and then attributed activity sequences and attributed translations. The basic idea is as follows.

1. An unattributed derivation tree is constructed from the underlying translation grammar.

2. For each occurrence of an input symbol in the derivation tree, arbitrary permissible values are assigned to its attributes.

3. The attribute rules are then employed wherever possible in an attempt to supply attribute values for all the attributes of all the occurrences of non-terminal and action symbols in the derivation tree.

Before discussing the ramifications of Step 3, we first discuss and interpret the attributed translation grammar definition.

Part 1 of the definition simply says that the input, nonterminal, and action symbols are to be attributed symbols.

In part 2, a distinction is made between inherited and synthesized attributes to indicate whether their values are to be computed by rules specified by part 3 or by rules specified by part 4. The terms “inherited” and “synthesized” were

introduced in [10], as was the term "attribute." A more detailed comparison with [10] is given at the end of this section.

Part 3 states what rules are needed to compute values for inherited attributes in a derivation tree. Each symbol in a derivation tree is either associated with the right-hand side of a production (i.e. the production which attaches the symbol to its parent in the tree) or is designated as the root of the tree (in which case the symbol is an occurrence of the starting symbol). These two cases account for the two sections A and B of part 3.

Section A says that each inherited attribute associated with a right-hand occurrence has a rule for computing its value based on some of its parent's attribute values, some of its sibling's attribute values, and even some of its own attribute values. The term "inherited" is suggestive of the idea that the rule is based on information obtained from the parent. The evaluation of the attribute rule can of course only be performed if the attribute values on which the rule depends have previously been computed.

Section B of part 3 says that initial values must be supplied for inherited attributes of the root of the derivation tree.

Part 4 states what rules are needed to compute values for synthesized attributes in a derivation tree. The case of a nonterminal attribute and an action symbol are treated separately.

Section A of part 4 deals with the nonterminal case. Because each nonterminal node in a derivation tree is associated with a left-hand side of a production, namely the production applied to that node, Section A ensures that there is a rule for each non- terminal synthesized attribute. The rule computes a value using some of the attribute values of the nonterminal's immediate descendants and possibly some of the non- terminal's own attribute values. The term "synthesized" is suggestive of the idea that a value is synthesized from the attributes of the descendants.

Section B of part 4 deals with the action symbol case. Here the rule is associated with the symbol itself (because the action symbol is not a left-hand side) and the rule is based solely on other attributes of the symbol (because the action symbol has no descendants). Synthesized action symbol attributes are almost completely neglected in the rest of the paper since an equivalent formulation with only inherited action symbol attributes can always be found for purposes of specifying a translation. Nevertheless, we believe it natural to include such attributes in modeling compilers.

Now we return to the problem of adding nonterminal and action symbol attributes to a derivation tree for which input symbol attribute values have been supplied. As a first step, values can be assigned to the inherited attributes of the root in accordance with the initial values required by Section 3B. Then perhaps rules can be found which depend only on the input attributes or the inherited attributes of the root, and the resulting values can be added to the tree.

Hopefully, as attribute values are added to the tree, the arguments of additional rules will be available, and still more values can be added until finally every attribute of each symbol on the derivation tree has an assigned value.

We say that an attributed translation grammar is *well defined* if and only if, for any derivation tree obtained from the underlying translation grammar, the process described above can be used to compute a value for each attribute of each symbol occurring in the derivation tree. This concept of "well defined" was introduced in [10], and the test given in [10] can be used with straightforward extensions to test an attributed translation grammar for the "well defined" condition. For application purposes, we are only interested in well defined attributed translation grammars, and our examples are all from this class.

Given an attributed translation grammar and given a derivation tree obtained from the grammar, the sequence of attributed input and action symbols obtained from the derivation tree is an *attributed activity sequence*. The attributed action part of this activity sequence is called a translation of the attributed input part. The set of attributed input part and action part pairs obtainable from the given grammar is called the *attributed translation* specified by the grammar. If an attributed translation grammar has an unambiguous input grammar, then each attributed input sequence has only one derivation tree and only one attributed translation.

Comparing the attributed translation grammars presented here with those of Knuth in [10], the principal difference is that we permit and require a certain class of terminal symbols (namely the input symbols) to have attributes whose values are not given by rules. There are also two minor differences. Knuth restricts terminals to have inherited attributes whereas we also permit synthesized attributes for our action terminals. Knuth also restricts the starting symbol to synthesized attributes only whereas we permit initialized inherited attributes. These two differences are minor in the sense that given any attributed translation grammar, the translation can be specified by an equivalent attributed translation grammar with all action symbol attributes inherited and all starting symbol attributes synthesized.

5. Examples

EXAMPLE 1. As a first example, we give an attributed translation grammar specifying the translation of expressions over constants mentioned previously.

The nonterminals $\langle E\rangle$, $\langle T\rangle$, and $\langle P\rangle$, each have an integer valued synthesized attribute. The input symbol C has one integer valued attribute and the action symbol ANSWER has an inherited integer valued attribute. The starting symbol is $\langle S\rangle$.

1. $\langle S\rangle \rightarrow \langle E\rangle_a \, \mathrm{ANSWER}_b$
   $b \leftarrow a$

2. $\langle E\rangle_d \rightarrow \langle E\rangle_e + \langle T\rangle_f$
   $d \leftarrow e + f$
3. $\langle E\rangle_g \rightarrow \langle T\rangle_h$
   $g \leftarrow h$
4. $\langle T\rangle_i \rightarrow \langle T\rangle_j * \langle P\rangle_k$
   $i \leftarrow j * k$
5. $\langle T\rangle_m \rightarrow \langle P\rangle_n$
   $m \leftarrow n$
6. $\langle P\rangle_p \rightarrow (\langle E\rangle_q)$
   $p \leftarrow q$
7. $\langle P\rangle_r \rightarrow \langle C\rangle_s$
   $r \leftarrow s$

The notation used to describe the rules for computing attributes is that each attribute of a symbol in a production is given a name and the rules are written below the productions in terms of these names. For instance the rule

$$d \leftarrow e + f$$

below production 2 specifies that attribute d is computed by evaluating the sum $e + f$.

In any derivation tree obtained from this grammar, the value of the attribute of each nonterminal $\langle E\rangle$, $\langle T\rangle$ and $\langle P\rangle$ equals the numerical value of the subexpression generated by that nonterminal. The value of the attribute of ANSWER is the numerical value of the entire expression.

The input sequence

$$(C_2 + C_5) * (C_{11} + C_3)$$

has the attributed derivation tree shown in Fig. 2.1. The activity sequence corresponding to the tree is

$$(C_2 + C_5) * (C_{11} + C_3)\mathrm{ANSWER}_{98}$$

and the action sequence is

$$\mathrm{ANSWER}_{98}.$$

To see that the attribute values in Fig. 2.1 are in fact obtainable by successive applications of attribute rules, observe that the values can be added to the unattributed tree simply by computing the values in a bottom up order. In

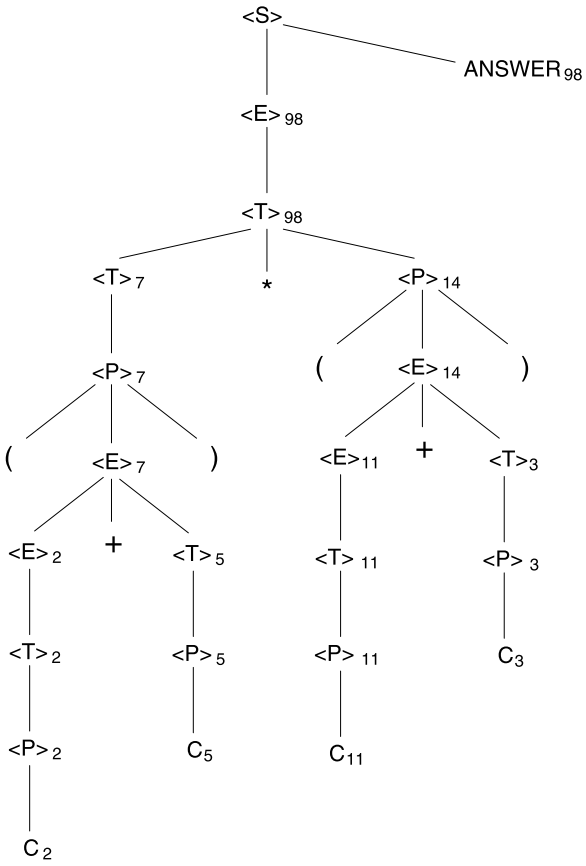

Figure 2.1.

other words, each nonterminal attribute can be computed as soon as the attribute values have been determined for the symbols below it, so its value can be computed by starting from the terminal attributes and working up the tree. The value of action ANSWER can be computed as the final step.

EXAMPLE 2. To show how an attributed translation grammar might be used in a compiler design, we consider the processing of declarations in a hypothetical programming language. The translation is one that the syntax box of a compiler might be required to perform. The input set consists of the three symbols:

1. REAL
2. *I*
3. ,

where I represents an identifier having one attribute. The value of this attribute is to be a pointer to a table entry for the identifier. The input language consists of the word REAL followed by a sequence of identifiers separated by commas. For each identifier, an action routine named ALLOCATE is to be called. This action routine is to fill in the table entry for the identifier with the run time location corresponding to the identifier. The identifiers are to be allocated consecutive locations beginning at location 50. Routine ALLOCATE has two parameters: a pointer to the table entry for the identifier and the value of the run time location. To represent the act of calling this routine, we use ALLOCATE as an action symbol with two inherited attributes, which take on the values of the routine's parameters.

The grammar has two nonterminals, ⟨DECLARATION⟩ and ⟨IDENTIFIER LIST⟩, of which the first is the starting symbol. Each nonterminal has two pointer-valued attributes, of which the first is inherited and the second is synthesized. The initial value of the inherited attribute of the starting symbol is 50.

The grammar is:

1. $\langle \text{DECLARATION} \rangle_{x1,z2} \rightarrow \text{REAL} I_{a1} \text{ALLOCATE}_{a2,x2} \langle \text{IDENTIFIER LIST} \rangle_{y,z1}$
   $a2 \leftarrow a1 \qquad y \leftarrow x1 + 1$
   $x2 \leftarrow x1 \qquad z2 \leftarrow z1$
2. $\langle \text{IDENTIFIER LIST} \rangle_{x1,z2} \rightarrow I_{a1} \text{ALLOCATE}_{a2,x2} \langle \text{IDENTIFIER LIST} \rangle_{y,z1}$
   $a2 \leftarrow a1 \qquad y \leftarrow x1 + 1$
   $x2 \leftarrow x1 \qquad z2 \leftarrow z1$
3. $\langle \text{IDENTIFIER LIST} \rangle_{x,z} \rightarrow \epsilon$
   $z \leftarrow x$

The inherited attribute of each nonterminal equals the run time location available for the first identifier generated from the nonterminal. The synthesized attribute equals the next available runtime location after space has been allocated to all the identifiers generated from the nonterminal. In this example, the synthesized attributes do not affect the attributes of the action symbols, but they might if this grammar were part of some larger grammar.

The input sequence

$$\text{REAL } I_3, I_9, I_2$$

has the derivation tree shown in Fig. 2.2. The activity sequence is

$$\text{REAL } I_3 \text{ ALLOCATE}_{3,50}, I_9 \text{ ALLOCATE}_{9,51}, I_2 \text{ ALLOCATE}_{2,52}$$

The attribute values shown in Fig. 2.2 were obtained by first computing the inherited values and then the synthesized attributes. The inherited attributes were evaluated starting with the initial value of the top node and evaluating each attribute after those above and to the left were evaluated. The first synthesized attribute evaluated was the one lowest on the tree and then the other synthesized attributes were evaluated working up the tree. The order of evaluation

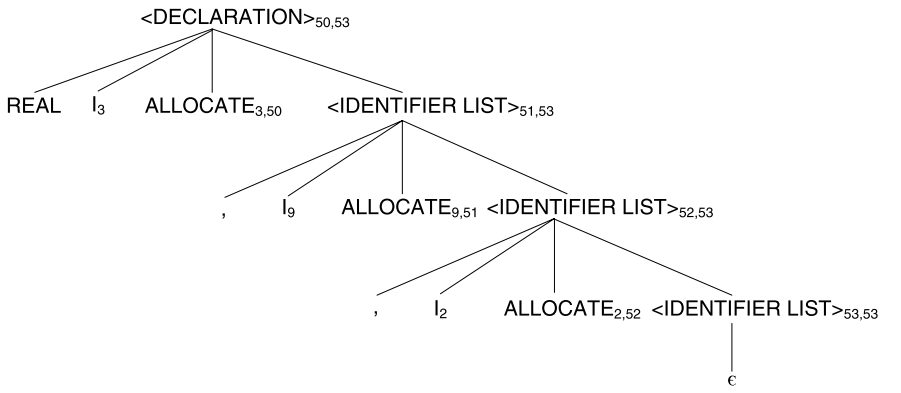

Figure 2.2.

illustrates a technique of sending information down the tree using inherited attributes and then sending it back up using synthesized attributes. Observe how the downward information is turned back up with the application of production 3.

EXAMPLE 3. As another example, we consider the translation of assignment statements in a hypothetical programming language. The input set is

$$\{(,),+,*,I,=\}$$

where I represents an identifier having one attribute whose value is to be a pointer to a table entry for the identifier.

The set of action symbols is

$$\{\text{ADD, MULTIPLY, ASSIGN}\}$$

where ADD and MULTIPLY each have three inherited attributes and ASSIGN has two inherited attributes. The attributes of ADD and MULTIPLY are to be pointers to the table entries for the left operand, right operand, and result of the operator. The attributes of ASSIGN are to be pointers to the table entries for an identifier being assigned to and the expression which is being assigned to the identifier.

The nonterminal set is

$$\{\langle S\rangle, \langle E\rangle, \langle T\rangle, \langle P\rangle, \langle E\text{-LIST}\rangle, \langle T\text{-LIST}\rangle\}.$$

Nonterminal $\langle S\rangle$ has no attributes. Nonterminals $\langle E\rangle$, $\langle T\rangle$, and $\langle P\rangle$ each have one attribute, which is synthesized. This attribute is to be a pointer to the table entry for the result of the subexpression generated by the nonterminal.

Nonterminals $\langle E\text{-LIST}\rangle$ and $\langle T\text{-LIST}\rangle$ each have two attributes, of which the first is inherited and the second is synthesized.

The attributed grammar is the following, with starting symbol $\langle S\rangle$.

1. $\langle S\rangle \rightarrow I_{a1} = \langle E\rangle_{b1}\,\text{ASSIGN}_{a2,b2}$
$$a2 \leftarrow a1 \qquad b2 \leftarrow b1$$
2. $\langle E\rangle_{b2} \rightarrow \langle T\rangle_{a1}\langle E\text{-LIST}\rangle_{a2,b1}$
$$a2 \leftarrow a1 \qquad b2 \leftarrow b1$$
3. $\langle E\text{-LIST}\rangle_{a1,d2} \rightarrow +\langle T\rangle_{b1}\,\text{ADD}_{a2,b2,c1}\langle E\text{-LIST}\rangle_{c2,d1}$
$$a2 \leftarrow a1 \qquad c2 \leftarrow c1$$
$$b2 \leftarrow b1 \qquad d2 \leftarrow d1$$
$$c1 \leftarrow \text{GETNEW}$$
4. $\langle E\text{-LIST}\rangle_{a1,a2} \rightarrow \epsilon$
$$a2 \leftarrow a1$$
5. $\langle T\rangle_{b2} \rightarrow \langle P\rangle_{a1}\langle T\text{-LIST}\rangle_{a2,b1}$
$$a2 \leftarrow a1 \qquad b2 \leftarrow b1$$
6. $\langle T\text{-LIST}\rangle_{a1,d2} \rightarrow *\langle P\rangle_{b1}\,\text{MULTIPLY}_{a2,b2,c1}\langle T\text{-LIST}\rangle_{c2,d1}$
$$a2 \leftarrow a1 \qquad c2 \leftarrow c1$$
$$b2 \leftarrow b1 \qquad d2 \leftarrow d1$$
$$c1 \leftarrow \text{GETNEW}$$
7. $\langle T\text{-LIST}\rangle_{a1,a2} \rightarrow \epsilon$
$$a2 \leftarrow a1$$
8. $\langle P\rangle_{a2} \rightarrow I_{a1}$
$$a2 \leftarrow a1$$
9. $\langle P\rangle_{a2} \rightarrow (\langle E\rangle_{a1})$
$$a2 \leftarrow a1$$

GETNEW is assumed to be a parameterless function procedure which supplies a pointer to some unused table entry that can be used to keep track of a partial result. Because different calls on GETNEW return different answers, GETNEW is not strictly speaking a function. Thus in using GETNEW, we are taking a small liberty with our formal definition. As an alternative to using GETNEW, extra attributes could be used to keep track of available table entries. However, the use of GETNEW is simpler and would be the likely choice in an actual design application.

Nonterminal $\langle E\text{-LIST}\rangle$ can be thought of as generating a list consisting of $+\langle T\rangle$ ADD repeated zero or more times. The inherited attribute of $\langle E\text{-LIST}\rangle$ corresponds to the left operand of the first $+$ (if any) on the list. The synthesized attribute of $\langle E\text{-LIST}\rangle$ corresponds to the result of the subexpression obtained by appending the string generated from $\langle E\text{-LIST}\rangle$ to the string representing the left operand. Nonterminal $\langle T\text{-LIST}\rangle$ is similar to $\langle E\text{-LIST}\rangle$.

For illustrative purposes, assume that GETNEW supplies consecutive locations beginning with location 200. Then the input sequence

$$I_7 = I_5 + I_2 * I_3$$

has the derivation tree shown in Fig. 2.3. The activity sequence is

$$I_7 = I_5 + I_2 * I_3 \text{MULTIPLY}_{2,3,200}\text{ADD}_{5,200,201}\text{ASSIGN}_{7,201}$$

and the action sequence is

$$\text{MULTIPLY}_{2,3,200}\text{ADD}_{5,200,201}\text{ASSIGN}_{7,201}$$

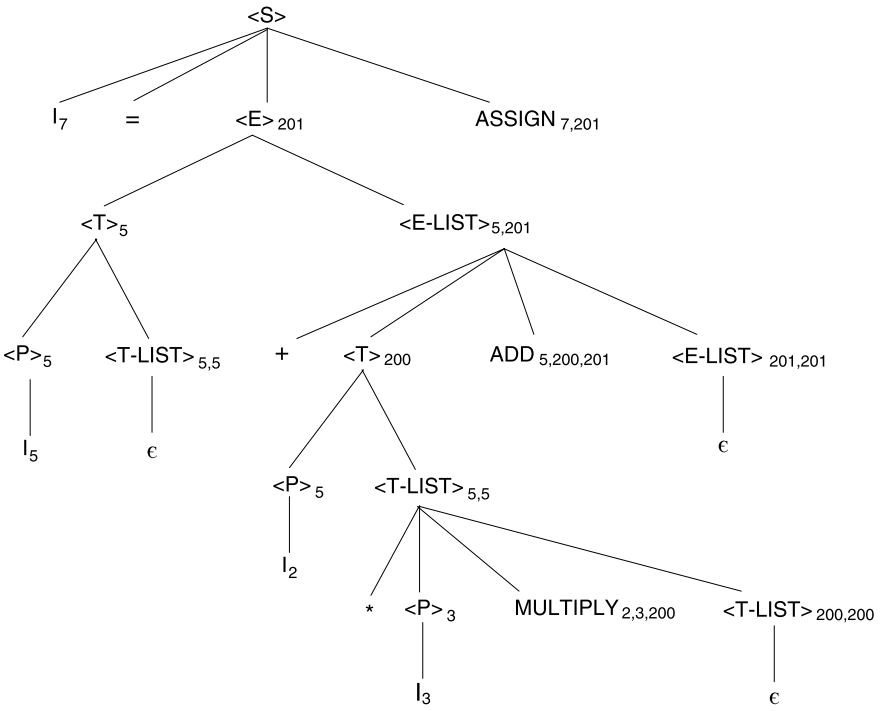

Figure 2.3.

The order of attribute evaluation in Fig. 2.3 is more complex than in the previous two examples. The most systematic order is to evaluate the inherited

attributes of a given symbol before evaluating attributes of its descendants, to evaluate the synthesized attributes of a symbol after evaluating attributes for the descendants, and to evaluate all attributes of a left sibling before a right sibling. The $\langle E\text{-LIST}\rangle$ and $\langle T\text{-LIST}\rangle$ portions of the tree again illustrate the technique of sending down inherited information and then passing back up synthesized information. Productions 4 and 7 are the productions which turn this information around.

6. Attributed Pushdown Machines

We are interested in devices that "perform" the attributed translation specified by an attributed translation grammar. By a device performing an attributed translation we mean the device reads the input symbols including their attributes, verifies that the input sequence is in the language specified by the input grammar and outputs the attributed action symbols specified by the activity sequence corresponding to the input sequence.

We are particularly interested in performing attributed translations with *attributed pushdown machines*. Attributed pushdown machines are similar to ordinary pushdown machines, except that the symbols and states of the machine have attributes that can be manipulated during the moves of the machine. Informally, an attributed pushdown machine is the same as a conventional pushdown machine except that:

1. Each input symbol, output symbol, state, and stack symbol has an associated fixed number of attributes.

2. Associated with each move of the machine is a specification of the attributes of the new state, the new top stack symbols (if the move is not a pop) and the outputs (if any) as a function of the attributes of the old state, top stack symbol and input symbol (if the move is not an ϵ-move).

Formally, a (nondeterministic) attributed pushdown transducer is an 11-tuple

$$(Q, I, Y, \Gamma, \delta, q, Z, A, C, u, v)$$

where:

Q is a finite state of *states*.

I is a finite set of *input symbols*.

Y is a finite set of *output symbols*, disjoint from I.

Γ is a finite set of *stack symbols*.

q in Q is the *initial state*.

Z in Γ is the *initial stack symbol*.

A is the set of possible *attribute values*.

C is a function from $Q \cup I \cup Y \cup \Gamma$ to the nonnegative integers, specifying how many attributes each of these symbols have. We let $\bar{C}$ denote the extension of C to $(Q \cup I \cup Y \cup \Gamma)^*$ defined by $\bar{C}(\epsilon) = 0$ and $\bar{C}(\alpha\beta) = C(\alpha) + \bar{C}(\beta)$ for α a single symbol.

u in $A^{C(q)}$ is the attribute set of the starting state.

v in $A^{C(Z)}$ is the attribute set of the starting stack symbol.

δ is a mapping of $Q \times (I \cup \{\epsilon\}) \times \Gamma$ into a finite set of 4-tuples such that if $\delta(r, a, \beta)$ contains (p, γ, ξ, f) then p is in Q, γ is in Γ^*, ξ is in Y^*, and f is a computable function from $A^{C(r)+\bar{C}(a)+C(\beta)}$ into $A^{C(p)+\bar{C}(\gamma)+C(\xi)}$. Furthermore each pair of 4-tuplets in $\delta(r, a, \beta)$ differs in at least one of the first three components.

We say that an attributed pushdown transducer is deterministic if

1. For each r in Q and β in Γ, whenever $\delta(r, \epsilon, \beta)$ is nonempty, then $\delta(r, a, \beta)$ is empty for all a in I;

2. δ never maps its argument into more than one element.

A configuration of an attributed pushdown translator is a 4-tuple (r, x, γ, y) where r is an attributed state, x is a string of attributed input symbols, γ is a string of attributed stack symbols, and y is a string of attributed output symbols. If a configuration is of the form $(r_g, a_h x, \beta_i \gamma, y)$ where r is a state with attributes g, a is in $I \cup \{\epsilon\}$ and has attributes h, β is a stack symbol with attributes i, and $\delta(r, a, \beta)$ contains (p, η, ξ, f) then we write $(r_g, a_h x, \beta_i \gamma, y) \vdash (\bar{p}, x, \bar{\eta}\gamma, y\bar{\xi})$ where $\bar{p}$, $\bar{\eta}$, and $\bar{\xi}$ are p, η, and ξ respectively with attributes computed by applying the function f to the attributes obtained by composing g, h, and i.

Let $\overset{*}{\vdash}$ denote the transitive reflexive closure of $\vdash$. Then if

$$(q_u, x, Z_v, \epsilon) \overset{*}{\vdash} (p, \epsilon, \epsilon, y)$$

we say that y is a translation of x performed by the machine. The *translation performed* by the machine is the set of all such pairs (x, y).

We say that a machine has an *endmarker* $\sharp$ for $\sharp$ in I if $C(\sharp) = 0$ and all input sequences for which the machine performs a translation are of the form $z\sharp$ where z is in $(I - \sharp)^*$. Note that the machine reads the endmarker in producing a translation. If a machine has endmarker $\sharp$, then we say that the translation

performed using an endmarker by the machine is the set of pairs (z, y) such that $(z\sharp, y)$ is in the translation performed by the machine.

Note that the set of translations performed by nondeterministic attributed pushdown machines is identical to the set of translations performed using an endmarker by nondeterministic attributed pushdown machines.

7. Performing Translations Nondeterministically

We define a subclass of attributed translation grammars and relate it to attributed pushdown machines.

An attributed translation grammar is called *L-attributed* if and only if the following three conditions hold.

1. For each attribute evaluation rule associated with an inherited attribute of some given symbol in the right-hand side of some given production, each argument of that rule is either an inherited attribute of the left-hand side or an arbitrary attribute of some right-hand side symbol appearing to the left of the given symbol.

2. For each attribute evaluation rule associated with a synthesized attribute of the left-hand side of some given production, each argument of that rule is either an inherited attribute of the given left-hand side or an arbitrary attribute of some right-hand side symbol.

3. For each attribute evaluation rule associated with a synthesized attribute of an action symbol, each argument of that rule is an inherited attribute of the given action symbol.

Comparing the above three conditions with the definition of attributed translation grammars, we see that 1, 2, and 3 above are restrictions on Sections 3A, 4A, and 4B, respectively. The only evaluation rules not constrained by the above three conditions are the initialization rules of Section 3B.

The *L* in the name "*L*-attributed" refers to the restriction (in condition 1 of the definition) that a rule for the inherited attribute of a given symbol in a production can use attributes of symbols to the left of the given symbol, but not attributes of symbols to the right. The intent of condition 1 is that the inherited attributes of a given node in the derivation tree should depend (either directly or indirectly) only on those input symbol attributes occurring to left of the given node, and be independent of the input symbol attributes below or to the right of the given node. A consequence of this intent is that the synthesized attributes of the given node should only depend on the input symbol attributes to the left or below the given node, and be independent of input symbol attributes to the right of the given node.

The purpose of conditions 2 and 3 is to ensure that the grammar is well defined. Together, the three conditions ensure that given a production such as

$A \rightarrow BC$, the attributes of A, B, and C can be evaluated in the following order:

1. Inherited attributes of A,
2. Inherited attributes of B,
3. Synthesized attributes of B,
4. Inherited attributes of C,
5. Synthesized attributes of C,
6. Synthesized attributes of A.

THEOREM 1. *Any translation specified by an L-attributed translation grammar can be performed by a nondeterministic attributed pushdown machine.*

Proof. We construct a one state machine which operates in a top down fashion. Let the translation grammar have input set I, action set Y, and nonterminal set N. If there are m productions, order them from 1 to m and let the ith production in the translation grammar have n_i symbols.

The machine is

$$(\{q\}, I, Y, \{Z\} \cup \{(i,j) | 1 \leq i \leq m \text{ and } 0 \leq j \leq n_i\}, \delta, q, Z, A, C, u, v)$$

where q and Z are arbitrary new names, A is the set of values that the attributes of the grammar can take us on, u is arbitrary, and C, v, and δ will be specified below.

For each symbol α in $I \cup Y$, $C(\alpha)$ equals the number of attributes α has in the grammar. For Z, $C(Z)$ equals 0, and so v is trivially a null vector. For q, $C(q)$ equals the maximum number of attributes of any symbol in the grammar. For each stack symbol of the form (i,j), $C((i,j))$ is equal to the sum of the number of attributes of the first $j-1$ symbols on the right-hand side of production i plus the number of inherited attributes of the left-hand nonterminal.

The machine parses top down, with stack symbol (i,j) representing a prediction of the rest of production i after the first j symbols. The machine operates so that when the top stack symbol is (i,j), the attributes of the stack symbol equal the inherited attributes of the left-hand nonterminal of production i, and the attributes of the first $j-1$ symbols on the right-hand side. Also, when $j > 0$ an appropriate number of attributes of the state will equal the attributes of the jth symbol on the right-hand side. Thus when (i,j) is on top of the stack the inherited attributes of the left-hand side of production i and all the attributes of the first j symbols of the right-hand side are available as attributes of the state and top stack symbol.

Stack symbol Z is used only to initialize the stack and disappears forever with the first machine operation. The first operation is to predict the production i applied to the starting symbol and replace the Z with the corresponding $(i, 0)$. Symbol $(i, 0)$ has an attribute for each inherited attribute of the starting symbol (left-hand side of production i) and these are initialized with the values specified as part of the grammar. Stated symbolically,

$$\delta(q, \epsilon, Z) = \{(q, (i, 0), \epsilon, f_i) \text{ for all productions } i \text{ with the starting symbol as left-hand side}\}$$

where f_i assigns the initial starting symbol inherited attribute values to the attributes of $(i, 0)$ and assigns arbitrary values to attributes of q.

When the top stack symbol of the machine has the form (i, j) where $j = n_i$, the machine predicts that an example of production i is over. The machine operation is to assign the attribute values of the left-hand side to a subset of the state attributes and to pop the stack to the symbol below. The inherited attributes of the left-hand side are immediately known since their values are given by corresponding attributes of (i, j). The synthesized attributes must now be computed, but this is easily done because of condition 2 of the L-attributed definition which says they can be computed from attribute values of the top stack symbol and the attributes of q.

$$\delta(q, \epsilon, (i, n_i)) \text{ is the one element set } \{(q, \epsilon, \epsilon, f)\}$$

where f is a function computing the left-hand side attributes of production i and assigning them to attributes of q (and assigning arbitrary values to any remaining state q attributes).

For a stack symbol of the form (i, j) where $j < n_i$ we consider three cases, depending on whether the $(j + 1)$st symbol on the right-hand side production i is in I, Y, or N. All three cases have the property that (i, j) is to be replaced with $(i, j + 1)$ and that the attribute values for this replacement symbol are already computed and are available as attributes of (i, j) and q. The actions taken in each case must also provide that the attributes for the j'th symbol are assigned to q, but the mechanism is different in each case. Letting α be the $(j + 1)$st symbol of production i, the three cases are as follows.

Case 1. α is an input symbol. In this case, the machine has an operation if and only if predicted input symbol α matches the current input. The obligation to make the attributes of state q equal the attributes of α is met simply by assigning the attributes of the machine input to q. Symbolically,

$$\delta(q, \alpha, (i, j)) \text{ is the one element set } \{(q, (i, j + 1), \epsilon, f)\}$$

where f fills in the values of $(i, j + 1)$ and q as described above.

Case 2. α is an action symbol. Conditions 1 and 3 of the L-attributed definition ensure that the attributes of α can be computed from the information at hand and be assigned as attributes of the state. Condition 1 says that the inherited attributes of α can be computed from the attribute values of top stack symbol (i, j) and the attributes of q. Condition 3 says that the synthesized attributes of α can then be computed from the inherited attributes. One other action associated with α is to put out α with its attribute values. Symbolically,

$$\delta(q, \epsilon, (i, j)) \text{ is the one element set } \{(q, (i, j+1), \alpha, f)\}$$

where f fills in the values of stack symbol $(i, j+1)$, state q, and output α as described above.

Case 3. α is a nonterminal symbol. Condition 1 of the L-attributed definition ensures that the inherited attributes can be computed from the attributes of the state and top stack symbol. The machine predicts a production k that generates the predicted occurrence of α, and places a symbol $(k, 0)$ on top of the stack (above the $(i, j+1)$) assigning to its attributes the inherited attributes of α. Later, when the symbol $(i, j+1)$ is exposed (due to popping a (k, n_k)), the attributes of α will appear as attributes of the state thus fulfilling the obligation to have $(i, j+1)$ appear with the attributes of α as state attributes. Symbolically,

$$\delta(q, \epsilon, (i, j)) \text{ equals } \{(q, (k, 0)(i, j+1), \epsilon, f_k) \text{ for all productions } k \text{ with left-hand nonterminal } \alpha\}$$

where f_k computes the attributes of $(k, 0)$ and $(i, j+1)$ and arbitrary values for q as described above.

This completes the construction. We have given arguments at each step to show that appropriate attribute values are always available and computed. The machine is otherwise a standard top down translator so we omit further arguments that it performs the desired attributed translation. □

EXAMPLE 4. Consider the following L-attributed translation grammar with input set $\{a, b\}$, action set $\{d\}$, nonterminal set $\{S, B\}$, and starting symbol S. Symbols a and b each have one attribute; S and d each have one inherited attribute; and B has two attributes, of which the first is inherited and the second synthesized. The starting value of the attribute of S is 4. The productions are

1. $S_r \rightarrow a_s B_{t,u} d_v$

   $t \leftarrow r + s$ $\qquad$ $v \leftarrow 3 * u$

2. $B_{r,s} \rightarrow b_t$

   $s \leftarrow r * t$

The machine constructed by the procedure described above would have the following sequence of configurations for input sequence a_2b_5. Wherever the machine can specify an arbitrary value for an attribute, the value 0 has been specified. The output sequence is d_{90}.

$$\begin{aligned}(q_{0,0}, a_2b_5, Z, \epsilon) &\vdash (q_{0,0}, a_2b_5, (1,0)_4, \epsilon)\\ &\vdash (q_{2,0}, b_5, (1,1)_4, \epsilon) \vdash (q_{0,0}, b_5, (2,0)_6(1,2)_{4,2}, \epsilon)\\ &\vdash (q_{5,0}, \epsilon, (2,1)_6\,(1,2)_{4,2}, \epsilon) \vdash (q_{6,30}, \epsilon, (1,2)_{4,2}, \epsilon)\\ &\vdash (q_{90,0}, \epsilon, (1,3)_{4,2,6,30}, d_{90}) \vdash (q_{4,0}, \epsilon, \epsilon, d_{90})\end{aligned}$$

THEOREM 2. *Any translation performed by a nondeterministic attributed pushdown machine can be specified by an L-attributed translation grammar.*

Proof. We modify a standard technique for picking a grammar off a machine [6]. Let the machine be $(Q, I, Y, \Gamma, \delta, q, Z, A, C, u, v)$. The grammar has input set I, action set Y, and nonterminal set $(Q \times \Gamma \times Q) \cup \{S\}$ where S is a new symbol and is also the starting nonterminal. The productions are of two forms

1. $S \rightarrow (q, Z, p)$ for each p in Q,

2. $(r, A, p) \rightarrow a\xi(q_1, B_1, q_2)(q_2, B_2, q_3)\cdots(q_m, B_m, q_{m+1})$ for each r, q_1, q_2, …, q_{m+1} in Q where $p = q_{m+1}$, each a in $I \cup \{\epsilon\}$, and A, B_1, B_2, …, B_m in Γ, such that $\delta(r, a, A)$ contains $(q_1, B_1B_2\cdots B_m, \xi, f)$. (If $m = 0$ then $q_1 = p$, $\delta(r, a, A)$ contains (p, ϵ, ξ, f) and the production is $(r, A, p) \rightarrow a\xi$).

Each input and action symbol in the grammar has the same number of attributes as the corresponding symbol in the machine, and all action symbol attributes are inherited. Nonterminal S has no attributes. A nonterminal of the form (r, A, p) has $C(r) + C(A)$ inherited attributes and $C(p)$ synthesized attributes.

For a form 1 production, the rules for the inherited attributes of (q, Z, p) are that they equal u and v.

For a form 2 production, the function f from the machine specifies the attributes of q_1, ξ and $B_1B_1\cdots B_m$ in terms of the attributes of r, a, and A. The rules associated with the production for computing the inherited attributes of ξ are obtained from f, with the inherited attributes of the left-hand non-terminal used instead of the attributes of the symbols r and A in the machine. If $m = 0$ the rules for computing the synthesized attributes of the left-hand nonterminal are similarly obtained from f. If $m > 0$, the rules for the synthesized attributes of the left-hand nonterminal specify that they equal the synthesized attributes of (q_m, B_m, q_{m+1}). The rules for computing the inherited attributes of symbol

(q_i, B_i, q_{i+1}) use the rules from f to compute the attributes corresponding to B_i. For $i = 1$, the rules for the inherited attributes corresponding to q_i are obtained from f. For $i > 1$, the inherited attribute rules specify that these attributes equal the synthesized attributes of the symbol (q_{i-1}, B_{i-1}, q_i). □

Note that the grammar is L-attributed.

THEOREM 3. *There exists a translation specified by an attributed translation grammar that cannot be performed by any nondeterministic attributed pushdown machine.*

Proof. The proof uses the following grammar, which is not L-attributed. The input set is $\{a, b, c\}$, action set is $\{1, 2, 3\}$, and starting nonterminal is S.

$$\begin{aligned} S &\rightarrow 1_y A c_x \\ &\quad\ y \leftarrow x \\ A &\rightarrow a2A \\ A &\rightarrow b3A \\ A &\rightarrow \epsilon \end{aligned}$$

Suppose this translation can be performed by a nondeterministic machine and that for some input string, the machine can produce the translation by emitting the 1 before reading the c, i.e.,

$$(q, stc_j, Z, \epsilon) \overset{*}{\vdash} (p, tc_j, \gamma, 1_j\xi) \overset{*}{\vdash} (r, \epsilon, \epsilon, 1_j\xi\eta)$$

But then for some other attribute k

$$(q, stc_k, Z, \epsilon) \overset{*}{\vdash} (p, tc_k, \gamma, 1_j\xi) \overset{*}{\vdash} (r, \epsilon, \epsilon, 1_j\xi\eta)$$

which is an incorrect translation.

If, on the other hand, 1 is never emitted before reading the c, then no output is produced until all inputs are read (1 being the first output symbol and c the last input symbol). Picking a grammar off this machine by the proof of Theorem 2, the underlying translation grammar would generate the set

$$L = \{wc1h(w) | w \text{ in } \{a, b\}^*\}$$

where h is the string homomorphism mapping a into 2 and b into 3. There is a string homomorphism which maps L into $\{ww | w \text{ in } \{a, b\}^*\}$, which is known to be not context free. Since context free languages are closed under homomorphisms, L is not a context free language. We conclude that no such grammar can be picked off a machine and hence no such machine can exist. □

8. Performing Translations Deterministically

In this section we study the attributed translations that can be performed using an endmarker by deterministic attributed pushdown machines. First we note that any translation that can be performed by a deterministic machine can also be performed using an endmarker by a deterministic machine. However, there are translations that can be performed using an endmarker by a deterministic machine, but that cannot be performed by a deterministic machine, simply because more languages can be accepted when the endmarker is used [5]. First we consider the case when the input grammar is $LL(k)$ [12, 13], i.e., can be parsed top down without backtrack.

THEOREM 4. *Any translation specified by an L-attributed translation grammar with an $LL(k)$ input grammar can be performed using an endmarker by a deterministic attributed pushdown machine.*

Proof. First construction 1 of [13] can be applied to the grammar so that the input grammar is strong $LL(k)$. For this input grammar, the next k input symbols always determine which production should be applied to a nonterminal [13]. Now a construction similar to that for Theorem 1 can be used to obtain the attributed pushdown machine, assuming that the machine is capable of looking ahead at the next k input symbols when selecting a move. The construction is modified so that the next k input symbols are used to determine which production to apply to a nonterminal. The resulting machine is deterministic and performs the attributed translation.

Since attributed pushdown machines as defined in this paper are not capable of lookahead, the standard lookahead machine must be simulated by the type of machine defined in this paper. This can be done in a straightforward manner using the machine state to remember k inputs and the attributes of the state to remember the attributes of k inputs. The simulating machine needs an endmarker and so the translation is performed using an endmarker by the resultant deterministic machine. □

Note that Examples 2, 3, and 4 are all L-attributed and all have an $LL(1)$ input grammar.

L-attributed translations with an $LL(k)$ input grammar can also be performed using the method of recursive descent [4]. In this method there is a procedure for recognizing each nonterminal in the grammar. To perform an attributed translation, the procedure has a parameter for each attribute of the corresponding nonterminal. In terms of ALGOL 60, the parameters corresponding to inherited attributes can be called by value, and the parameters corresponding to synthesized values must be called by name. In the call of one of the procedures, an actual parameter corresponding to an inherited attribute

is the value of the attribute, and an actual parameter corresponding to a synthesized attribute is a variable to which the value of the synthesized attribute should be assigned during the execution of the called procedure.

As an example, the following ALGOL-like program is a recursive descent processor based on the grammar of Example 4, assuming the attribute values are integers.

```
begin
procedure S(r); value r; integer r;
    comment This procedure translates an example of nonterminal S.
    All examples of S begin with input symbol a;
    if input symbol = a
    then begin integer s, t, u, v;
        s := attribute of input symbol;
        advance to next input symbol;
        t := r + s;
        B(t, u);
        v := 3 * u;
        output ("d", v)
        end
    else reject;
procedure B(r, s); value r; integer r, s;
    comment This procedure translates an example of nonterminal B.
    All examples of B begin with input symbol b;
    if input symbol = b
    then begin integer t;
        t := attribute of input symbol;
        advance to next input symbol;
        s := r * t;
        end
    else reject;
    comment execution starts here;
    S(4);
    if input symbol = end marker then accept else reject
    end
```

Bochman [1] independently shows that, in his model, if the attribute rules satisfy conditions similar to those in our definition of L-attributed grammars, the attributes can be evaluated in a top down scan of a derivation tree by calling recursive procedures.

We now study attributed translations that can be performed while parsing bottom up. First we need the following definition.

> An attributed grammar is called *Polish* if and only if all action symbols occur only at the extreme right end of the right-hand sides of productions.

Any unattributed translation specified by a Polish translation grammar with an $LR(k)$ input grammar can be performed using an endmarker by a deterministic pushdown machine [12]. However this result does not hold when the grammar is L-attributed.

THEOREM 5. *There exists a translation specified by an L-attributed Polish translation grammar with an $LR(0)$ input grammar that cannot be performed by any deterministic attributed pushdown machine.*

Proof. Consider the following L-attributed grammar with input set $\{a, b, c, d\}$, action set $\{0, 1, 2\}$ and nonterminal set $\{S\}$. Nonterminal S has one inherited attribute for which the initial value is 1. Action symbol 2 has an inherited attribute.

$$\begin{aligned} S_x &\rightarrow aS_y c0 \\ &\qquad y \leftarrow 2 * x \\ S_x &\rightarrow aS_y d1 \\ &\qquad y \leftarrow 2 * x + 1 \\ S_x &\rightarrow b2_y \\ &\qquad y \leftarrow x \end{aligned}$$

Suppose this translation can be performed by a deterministic machine. The attribute of action symbol 2 cannot be determined by the machine until after the entire input sequence has been read, and so the machine cannot produce any output until after it reads the entire input sequence. The machine must therefore be able to read a sequence in $\{c, d\}^*$ and then output the same sequence with c replaced by zero and d by 1. However when the machine reaches the end of the input string, the first part of its output string is determined by the upper portion of its stack contents, and this upper portion can only reflect the end of the sequence in $\{c, d\}^*$. Therefore such a deterministic machine does not exist. □

> An L-attributed grammar is called *S-attributed* if and only if all attributes of nonterminals are synthesized.

Many compilers that parse bottom up use a design method that only permits the call of a "semantic action" when a production is recognized. If furthermore, the information available to the semantic action is associated with the right-hand side of the recognized production, and the information returned by

the semantic action is associated with the left-hand side, the design method corresponds to S-attributed Polish translation grammars.

THEOREM 6. *Any translation specified by an S-attributed Polish translation grammar with an $LR(k)$ input grammar can be performed using an endmarker by a deterministic attributed pushdown machine.*

Proof. The machine is based on the standard $LR(k)$ machine [6, 8] for recognizing the unattributed version of the input grammar in a bottom up fashion. Each stack symbol has a set of attributes equal to the attributes of the grammatical symbol it represents. When a production is recognized, the attributes of the action symbols and left-hand nonterminal are computed, and the outputs are emitted. □

However, S-attributed translation grammars cannot specify all translations that deterministic attributed pushdown machines can perform using an endmarker.

THEOREM 7. *There exists a translation specified by an L-attributed translation grammar with an $LL(1)$ input grammar that cannot be specified by any S-attributed translation grammar.*

Proof. Consider the following translation grammar with input set $\{a, b, c\}$, action set $\{1, 2, 3\}$, and starting nonterminal S. Nonterminal A and action symbol 2 each have one inherited attribute; and input a has one attribute. No other symbols have attributes.

$$\begin{aligned} S &\rightarrow a_x A_y \\ &\quad\ y \leftarrow x \\ A_x &\rightarrow b1A_y c3 \\ &\quad\ y \leftarrow x \\ A_x &\rightarrow d2_y \\ &\quad\ y \leftarrow x \end{aligned}$$

Observe that the grammar is L-attributed and has an $LL(1)$ input grammar. The activity sequences generated by this grammar have input part $a_x b^n d c^n$ and action part $1^n 2_x 3^n$ where $n \geq 0$ and the attribute of 2 equals the attribute of a.

Suppose this translation can be specified by an S-attributed translation grammar. Then it can be shown (see for instance the proof of the "$uvwxy$" theorem in [5]) that associated with the grammar there is an integer p such that all activity sequences of length greater than p can be written in the form $uvwxy$ where v and x are not both ϵ, and there is a nonterminal A such that

the starting symbol of the grammar generates uAy and $A \stackrel{*}{\Rightarrow} vAx \stackrel{*}{\Rightarrow} vwx$. An implication of this is that all sequences of the form $uv^m wx^m y$ are generated by the grammar. Since an activity sequence containing $n+2$ input symbols must contain exactly $n+1$ action symbols, vx must contain an equal number of input symbols and action symbols.

Now consider an activity sequence generated from the hypothetical S-attributed grammar and having length greater than p. We wish to show that the single occurrence of a from the input part must be part of u and the single occurrence of 2 from the action part must be part of w.

The one occurrence of a cannot be in v or x because these sequences are repeated. The a could not occur in y because then y would contain all the input symbols and vx would contain only action symbols. Finally, the a cannot occur in w, because then all the input symbols in vx would be in x, and $uvvwxxy$ would have an input part that is not of the form $a_x b^n dc^n$. We conclude that a is in u.

The one occurrence of 2 cannot be in v or x because these sequences are repeated. The 2 cannot occur in u, because then vx would contain action symbol 3, but not action symbol 2. Similarly, 2 in y would imply that vx contains action symbol 1, but not action symbol 3. We conclude that 2 is in w.

From the "$uvwxy$" theorem, we now conclude that there is a derivation of an activity sequence where $A \stackrel{*}{\Rightarrow} w$ and w contains 2, but not a. Since the grammar is assumed to be S-attributed, the nonterminals have only synthesized attributes. Therefore the attributes of any action symbols generated from a nonterminal can only be computed in terms of the attributes of the input symbols actually generated from that nonterminal. Since a is not generated from nonterminal A, there is no way of specifying that the attribute of 2 equals the attribute of a.

We now give a characterization of the translations that can be performed by deterministic attributed pushdown machines, i.e., we define a class of attributed translation grammars which specify exactly the set of attributed translations that can be performed by deterministic attributed pushdown machines. The characterization is in terms of an extension of strict deterministic grammars [5] in which we take the attributes and action symbols into account.

An attributed translation grammar (with terminals and nonterminals V) is called *SD-attributed* if and only if it is L-attributed and there exists a partition π on V such that

1. All input symbols are in the same block of π.

2. For each action symbol y, $\{y\}$ is a block of π.

3. All the nonterminals in the same block of π have the same number of inherited attributes.

4. The inherited attributes of each nonterminal can be ordered so that for any nonterminals A and A' in the same block of π, if $A \to \alpha\beta$ and $A' \to \alpha\beta'$ are productions (α, β, β', in V^*) then either

   (a) both β and $\beta' \neq \epsilon$, in which case the first symbol of β and β' are in the same block of π, and the rules for computing corresponding inherited attributes of these symbols (in terms of the attributes of α and corresponding inherited attributes of A and A') are the same, or

   (b) $\beta = \beta' = \epsilon$ and $A = A'$. □

THEOREM 8. *Any translation performed by a deterministic attributed pushdown machine can be specified by an SD-attributed translation grammar.*

Proof. The grammar obtained from the machine by the construction used in the proof of Theorem 2 is SD-attributed. To construct the partition whose existence is required by the definition, place nonterminals of the form (r, A, p) in the same block if and only if they have the same first two components. Then place nonterminal S in a one element block, place the input symbols together as a block, and put each action symbol in a separate one element block. The attribute ordering required by condition 4 is then easily supplied. □

THEOREM 9. *Any translation specified by an SD-attributed translation grammar can be performed by a deterministic attributed pushdown machine.*

Proof. We extend the construction in [5]. Let the grammar have partition π and vocabulary V consisting of input set I, action set Y, and nonterminal set N.

The machine is

$$(Q, I, Y, \Gamma, \delta, Z, A, C, u, v)$$

where $Q = \{q_j \mid 0 \leq j <$ maximum number of symbols in a block of $\pi\}$. $\Gamma = \{(V_i, \alpha) \mid A \to \alpha\beta$ for some A in block V_i and α, β in $V^*\} \cup \{(V_i, \alpha, V_j) \mid A \to \alpha B \beta$ for some A in block V_i, nonterminal B in block V_j and α, β in $V^*\}$. $Z = (V_0, \epsilon)$ where V_0 is the block containing the starting nonterminal.

A is the set of values that the attributes of the grammar can take on.

$C(a)$ for a in $I \cup Y$ equals the number of attributes a has in the grammar. $C((V_i, \alpha))$ equals the sum of the number of attributes of α in the grammar and the number of inherited attributes of a symbol in V_i. $C((V_i, \alpha, V_j))$ equals the sum of the number of attributes of α, the number of inherited attributes of a symbol in V_i, and the number of inherited attributes of a symbol in V_j. $C(q)$ for q in Q equals the maximum number of synthesized attributes of any nonterminal.

u is arbitrary.

v equals the inherited attributes of the starting nonterminal.

δ consists of the following five types of moves.

For any V_i, V_k blocks of nonterminals, α in V^*, a in I, y in Y, and q in Q.

(i) $\delta(q_0, a, (V_i, \alpha)) = \{(q_0, (V_i, \alpha a), \epsilon, f)\}$ if $A \to \alpha a \beta$ for some A in V_i and β in V^*.

(ii) $\delta(q_0, \epsilon, (V_i, \alpha)) = \{(q_0, (V_i, \alpha y), y, f)\}$ if $A \to \alpha y \beta$ for some A in V_i and β in V^*.

(iii) $\delta(q_0, \epsilon, (V_i, \alpha)) = \{(q_0, (V_k, \epsilon)(V_i, \alpha, V_k), \epsilon, f)\}$ if $A \to \alpha B \beta$ for some A in V_i, nonterminal B in V_k and β in V^*.

(iv) $\delta(q_0, \epsilon, (V_i, \alpha)) = \{(q_j, \epsilon, \epsilon, f)\}$ if $A \to \alpha$ is a production and A is the jth nonterminal in its block.

(v) $\delta(q_j, a, (V_i, \alpha, V_k)) = \{(q_0, (V_i, \alpha B), \epsilon, f)\}$ if B is the jth nonterminal in block V_k.

In case (iv), function f computes the synthesized attributes of A (from the attributes of (V_i, α)) and assigns them to q_j. In all other cases, f assigns arbitrary values to the attributes of the new state.

In case (i), f assigns to $(V_i, \alpha a)$ the attributes of (V_i, α) plus the attributes of a. In case (ii), f computes the attributes of y. It assigns these attributes to the output and (together with the attributes of (V_i, α)) to $(V_i, \alpha y)$. In case (iii), f computes the inherited attributes of B. Because the grammar is SD-attributed, all such B have the same rule for computing their attributes. Function f assigns these attributes to (V_k, ϵ) and (together with the attributes of (V_i, α)) to (V_i, α, V_k). In case (v), f assigns the attributes of q_j and (V_i, α, V_k) to $(V_i, \alpha B)$. □

9. Performing Arbitrary Translations

In this section we show that if the attribute rules specify all the attribute values in a derivation tree, the attributes can be computed on a random access device in an amount of time proportional to the number of edges in the derivation tree. When the grammar is well defined, the attribute rules specify all the attribute values in all derivation trees.

THEOREM 10. *Given a derivation tree for which the attribute rules specify all the attribute values, and assuming that one unit of time is charged for the evaluation of an attribute rule, then the attributes can be computed in time linear with the number of edges in the derivation tree.*

Proof. Construct a directed graph containing a node for each attribute of each node of the derivation tree. The graph contains an edge from node a to node b if the rule for computing attribute b uses the value of attribute a. Since the rules for computing attributes can only depend on other attributes in the same production, the number of edges and nodes in the graph is bounded by some constant (based on the attributed grammar) times the number of edges in the derivation tree.

Since the attributed grammar specifies all the attributes of the tree (given the values of the input symbol attributes and starting symbol inherited attributes) the constructed graph has no cycles. Therefore a topological sort can be performed on the graph, using an algorithm whose time is linear with the size of the graph [9]. The attributes associated with the nodes of the graph can then be evaluated in the order produced by the topological sort. Each attribute will be evaluated after the attributes on which the rule for computing it depends. □

10. Summary

A grammatical method of specifying attributed translations has been presented. The traditional top down and bottom up pushdown translators have been generalized to perform these translations. As with unattributed pushdown machines, the generalizations also operate in linear time (excluding the time required to evaluate the attribute evaluation functions).

Generalizations of $LL(k)$ and $LR(k)$ grammars are L-attributed $LL(k)$ and S-attributed $LR(k)$ grammars respectively. Neither of these grammars is sufficient to characterize deterministic attributed pushdown translations since $LL(k)$ grammars do not have sufficient syntactic power and S-attributed grammars do not have sufficient semantic power (Theorem 7). However, a characterization of deterministic attributed pushdown translations can be obtained by merging a top down attribute concept (L-attributed grammars) with a bottom up grammatical concept (SD grammars of [5]).

Taken together, the results show that grammatical specification and translation techniques can be generalized in a natural way to handle attributed translations without significant increases in processing cost. Thus attributed translation grammars can be a suitable basis for a theory of formal semantics of translation.

Acknowledgment

The authors wish to thank Professor Michael M. Hammer of the Massachusetts Institute of Technology for many useful comments concerning the presentation of this paper.

References

[1] G. V. Bochman. Semantics evaluated from left to right. Technical report, Departement d'Informatique, Univ. de Montreal, 1973.

[2] D. Crowe. Generating parsers for affix grammars. *Comm. Assoc. Computing Mach.*, 15:728–734, 1972.

[3] K. Culik. Attributed grammars and languages. Technical report, Departement d'Informatique, Univ. de Montreal, 1969.

[4] D. Gries. *Compiler Construction for Digital Computers*. Wiley, New York, 1971.

[5] M. A. Harrison and I. M. Havel. Strict deterministic grammars. *J. Comput. System Sci.*, 7:237–277, 1973.

[6] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison–Wesley, Reading, 1969.

[7] E. T. Irons. A syntax directed compiler for ALGOL 60. *Comm. Assoc. Comput. Mach.*, 4:51–55, 1961.

[8] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[9] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison–Wesley, Reading, 1968.

[10] D. E. Knuth. Semantics of context free languages. *Math. Systems Theory*, 2:127–145, 1968.

[11] C. H. A. Koster. Affix grammars. In *ALGOL 68 Implementation*. North-Holland, Amsterdam, 1971.

[12] P. M. Lewis and R. E. Stearns. Syntax directed transduction. *J. Assoc. Comput. Mach.*, 15:465–488, 1968.

[13] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top-down grammars. *Information and Control*, 17:226–256, 1970.

[14] R. E. Stearns and P. M. Lewis. Property grammars and table machines. *Information and Control*, 14:524–549, 1969.

http://www.springer.com/978-1-4020-9687-7

Fundamental Problems in Computing
Essays in Honor of Professor Daniel J. Rosenkrantz
Ravi, S.S.; Shukla, S.K. (Eds.)
2009, XXII, 516 p., Hardcover
ISBN: 978-1-4020-9687-7