# Attribute Grammars and their Applications

Krishnaprasad Thirunarayan
Metadata and Languages Laboratory
Department of Computer Science and Engineering
Wright State University
Dayton, OH-45435

## INTRODUCTION

Attribute grammars are a framework for defining semantics of programming languages in a syntax-directed fashion. In this paper, we define attribute grammars, and then illustrate their use for language definition, compiler generation, definite clause grammars, design and specification of algorithms, etc. Our goal is to emphasize its role as a tool for design, formal specification and implementation of practical systems, so our presentation is example rich.

## BACKGROUND

The lexical structure and syntax of a language is normally defined using regular expressions and context-free grammars respectively (Aho et al, 2007, Chapters 3-4). Knuth (1968) introduced attribute grammars to specify static and dynamic semantics of a programming language in a syntax-directed way. Let $G = (N, T, P, S)$ be a context-free grammar for a language L (Aho et al, 2007). N is the set of non-terminals. T is the set of terminals. P is the set of productions. Each production is of the form A ::= $\alpha$, where A $\in$ N and $\alpha \in$ (N U T)*. S $\in$ N is the start symbol. An *attribute grammar* AG is a triple (G, A, AR), where G is a context-free grammar for the language, A associates each grammar symbol X $\in$ N U T with a set of attributes, and AR associates each production R $\in$ P with a set of attribute computation rules (Paakki, 1995). A(X), where X $\in$ (N U T), can be further partitioned into two sets: synthesized attributes S(X) and inherited attributes I(X). AR(R), where R $\in$ P, contains rules for computing inherited and synthesized attributes associated with the symbols in the production R.

Consider the following attribute grammar that maps bit strings to numbers. CFG = ({N},{0,1},P,N), where P is the left column of productions shown below. The number semantics is formalized by associating a

synthesized attribute *val* with N, and providing rules for computing the value of the attribute *val* associated with the left-hand side N (denoted $N_l$) in terms of the value of the attribute *val* associated with the right-hand side N (denoted $N_r$), and the terminal.

$$
\begin{array}{ll}
\text{N ::= 0} & \text{N}.val = 0 \\
\text{N ::= 1} & \text{N}.val = 1 \\
\text{N ::= N0} & N_l.val = 2 * N_r.val \\
\text{N ::= N1} & N_l.val = 2 * N_r.val + 1
\end{array}
$$

An attribute grammar involving only synthesized attributes is called an ==*S-attributed grammar.*== It is straightforward to parse a binary string using this grammar and then compute the value of the string using a simple top-down left-to-right traversal of the abstract syntax tree.

The above attribute grammar is not unique. One can construct a different S-attributed grammar for the same language and the same semantics as follows.

$$
\begin{array}{ll}
\text{N ::= 0} & \text{N}.val = 0,\ \text{N}.len = 1 \\
\text{N ::= 1} & \text{N}.val = 1,\ \text{N}.len = 1 \\
\text{N ::= 0N} & N_l.val = N_r.val; \\
 & N_l.len = N_r.len + 1 \\
\text{N ::= 1N} & N_l.val = 2^{\wedge} N_r.len + N_r.val; \\
 & N_l.len = N_r.len + 1
\end{array}
$$

Attribute grammars can be devised to specify different semantics associated with the same language. For instance, the bit string can be interpreted as a fraction by associating an inherited attribute *pow* to capture the left context -- the length of the bit string between left of a non-terminal and the binary point, to determine the local value or the weight of the bit.

$$
\begin{array}{ll}
\text{F ::= . N} & \text{F}.val = \text{N}.val,\quad \text{N}.pow = 1 \\
\text{N ::= 0} & \text{N}.val = 0 \\
\text{N ::= 1} & \text{N}.val = (1 / 2^{\wedge} \text{N}.pow) \\
\text{N ::= 0N} & N_l.val = N_r.val; \\
 & N_r.pow = N_l.pow + 1 \\
\text{N ::= 1N} & N_l.val = (1 / 2^{\wedge} \text{N}.pow) + N_r.val; \\
 & N_r.pow = N_l.pow + 1
\end{array}
$$

Each production is associated with ==attribute computation rules== to compute the synthesized attribute *val* of the left-hand side non-terminal and the inherited attribute *pow* of the right-hand side non-terminal. (We leave it as an exercise for the interested reader to devise an S-attributed grammar to capture this semantics.)

# APPLICATIONS OF ATTRIBUTE GRAMMARS

Attribute grammars provide a *modular framework* for formally specifying the semantics of a language based on its context-free grammar (or in practice, for conciseness, on its Extended Backus Naur Formalism representation (Louden, 2003, Chapter 4)). By *modular*, we emphasize its role in structuring specification that is incremental with respect to the productions. That is, it is possible to develop and understand the attribute computation rules one production at a time. By *framework*, we emphasize its role in structuring a specification, rather than conceptualizing the meaning. For instance, denotational semantics, axiomatic semantics, and operational semantics of a language can all be specified using the attribute grammar formalism by appropriately choosing the attributes (Louden, 2003, Chapter 13). In practice, different programming languages interpret the same syntax/construct differently, and attribute grammars provide a framework for defining and analyzing subtle semantic differences.

In this section, we illustrate the uses and the subtleties associated with attribute grammars using examples of contemporary interest. Traditionally, attribute grammars have been used to specify various compiler activities formally. We show examples involving (i) type checking/inference (static semantics), (ii) code generation, and (iii) collecting distinct variables in a straight-line program. Attribute grammars can also be used to specify compiler generator activities. We show parser generator examples specifying the computation of (i) nullable non-terminals, (ii) first sets, and (iii) follow sets, in that sequence (Aho et al, 2007, Chapter 4). Definite clause grammars enable attribute grammars satisfying certain restrictions to be viewed as executable specifications (Bratko, 2001, Chapter 21). We exemplify this in SWI-Prolog. The essence of attribute grammars can be seen to underlie several database algorithms. We substantiate this by discussing the Magic sets for optimizing bottom-up query processing engine. We also discuss how attribute grammars can be used for developing and specifying algorithms for information management (Thirunarayan et al, 2005).

*Type Checking, Type Inference, and Code Generation*

Consider a simple prefix expression language containing terminals {n, x, +}. The type of variable n is int, and the type of variable x is double. The binary arithmetic operation + returns an int result if both the operands are int, otherwise it returns a double. The type of a prefix expression can be specified as follows.

$$
\begin{array}{ll}
E ::= n & E.typ = \text{int} \\
E ::= x & E.typ = \text{double} \\
E ::= +\ E\ E & E.typ = \text{if } E_{r1}.typ = E_{r2}.typ \\
& \qquad\qquad \text{then } E_{r1}.typ \text{ else double}
\end{array}
$$

The corresponding executable specification in Prolog can be obtained by defining a binary relation 'typ' between prefix expression terms and their types as follows. Observe that each line of specification that ends in a "." is an axiom (first two are Prolog facts, while last two are Prolog rules), E, F, T, T1, and T2 are universally quantified Prolog variables, ":-" stands for *logical if*, and "," stands for *logical and*.

$$typ(i,int).$$
$$typ(x,double).$$
$$typ(+(E,F),T) :- typ(E,T), typ(F,T).$$
$$typ(+(E,F),real) :- typ(E,T1), typ(F,T2),\ T1 \mathbin{\backslash}= T2.$$

A type checking query '?- typ(+(n,x),int).' verifies  if the expression '+(n,x)' is of type int, while a type inference query '?- typ(+(n,x),T).' determines the type of the expression '+(n,x)'.

Attribute grammar specifying the translation of an equivalent expression language containing infix + into Java bytecode in the context of the instance method definition:  'class { double f(int n, double i) { return E;}}' is as follows:

$E ::= n \qquad E.code = [iload\_1]$
$E ::= x \qquad E.code = [dload\_2]$
$E ::= E + E \quad E.code = $ if $E_{r1}.typ = int$
$\qquad\qquad\qquad\qquad$ then if $E_{r2}.typ = int$
$\qquad\qquad\qquad\qquad\qquad$ then $E_{r1}.code@E_{r2}.code@[iadd]$
$\qquad\qquad\qquad\qquad\qquad$ else $E_{r1}.code@[i2d]@E_{r2}.code@[dadd]$
$\qquad\qquad\qquad\qquad$ else if $E_{r2}.typ = int$
$\qquad\qquad\qquad\qquad\qquad$ then $E_{r1}.code@E_{r2}.code@[i2d,dadd]$
$\qquad\qquad\qquad\qquad\qquad$ else $E_{r1}.code@E_{r2}.code@[dadd]$

The attribute *typ* has been specified earlier. The attribute *code* is bound to a list of Java bytecode. '@' refers to list append operation. Java compiler maps the formal parameters n and x to register 1, and register pair 2 and 3, respectively. (The double value requires two registers.) iload_1 (dload_2) stands for pushing the value of the int n (double x) on top of the stack; dadd (iadd) stands for popping the top two double (int) values from the stack, adding them, and pushing the result on top of the stack; and i2d stands for coercing an int value into a double value (Lindholm & Yellin, 1999, Chapter 3). (Note that + is left associative in Java.) In practice, the code generator has to cater to variations on whether the method is static or instance, whether the formal arguments require 4 or more registers, whether the arguments are of primitive types or reference types, etc, and all this can be made explicit via attribute grammars.

*Collecting distinct identifiers*

We use the example of collecting distinct identifiers in an expression to illustrate the influence of primitive data types available for specifying the

semantics on the ease of writing specifications, and the rules of thumb to be used to enable sound and complete <mark>attribute computation</mark> in one-pass using top-down left-to-right traversal of the abstract syntax tree.

$$<exp> \ ::= \ <var> \ | \ <exp> + <exp>$$

Suppose we have ADT SET available to us as a primitive. We associate synthesized attributes *id* and *ids* with <var> and <exp> respectively, to obtain the following attribute grammar. ('$\cup$' refers to set-union.)

$$<exp> \ ::= <var> \qquad <exp>.ids = \{ <var>.id \}$$
$$<exp> \ ::= <exp> + <exp> \qquad <exp>.ids = <exp>.ids \ \cup <exp>.ids$$

Instead, if we have only ADT LIST available to us, we associate synthesized attribute *envo* and inherited attribute *envi* of type list of symbols with <exp>, to obtain the following attribute grammar.

$$<exp> \ ::= <var> \qquad <exp>.envo = \text{if } <var>.id \in <exp>.envi$$
$$\text{then } <env>.envi$$
$$\text{else cons}(<var>.id, <env>.envi)$$
$$<exp> \ ::= <exp> + <exp> \qquad <exp_{r1}>.envi = <exp_l>.envi$$
$$<exp_{r2}>.envi = <exp_{r1}>.envo$$
$$<exp_l>.envo = <exp_{r2}>.envo$$

Observe that, given the definition of the attributes and the production rule, one can *automatically* determine the left-hand sides of the <mark>attribute computation rules</mark> required. *There is one rule for each synthesized attribute of the left-hand side non-terminal, and one rule for each inherited attribute of the right-hand side symbol (non-terminal).*

$$<exp> \ ::= \ <exp> \ + \ <exp>$$
$$\downarrow \textbf{envi} \qquad \downarrow envi \qquad \downarrow envi$$
$$\uparrow envo \qquad \uparrow \textbf{envo} \qquad \uparrow \textbf{envo}$$

For the above production, the attributes shown in italics need to be determined using the attributes given in italics as explained below. *To enable one-pass top-down left-to-right computation of the <mark>attributes</mark>, each inherited attribute of the right-hand side symbol can depend on all the attributes associated with preceding right-hand side symbols and the inherited attribute of the left-hand side non-terminal. Similarly, the synthesized attribute of the left-hand side non-terminal can depend on all the attributes associated with all the right-hand side symbols and the inherited attribute of the left-hand side non-terminal.* Effectively, the inherited attribute associated with the left-hand side non-terminal provides context information from the parent and the left siblings, while the synthesized attributes of the right-hand side non-terminals provide information from their descendants.

From modularity perspective, if the attribute computation rules associated with each production satisfies the above constraints, it can be argued using induction principle that all the attributes associated with each node in the abstract syntax tree will be well-defined after one-pass of computation.

In practical compiler construction, a programming language is specified to a parser generator as an S-attributed grammar that transforms a program into its abstract syntax tree. This abstract syntax tree is traversed multiple times for semantic analysis and code generation. The APIs of the meta-language provide primitive functions and data types for defining and implementing semantic actions (attribute computations). For example, bottom-up parser generator Bison is based on C/C++ libraries, top-down parser generator ANTRL is based on Java APIs etc.

*Definite Clause Grammars*

Prolog's definite clause grammars can be used to execute the set-based and the list-based attribute grammars discussed earlier. Each DCG rule resembles a production with the predicate name corresponding to the non-terminal and the formal arguments corresponding to the attributes. The input string is encoded as a list of symbols (tokens). Upon loading, the DCG rules are automatically translated into ordinary Prolog rules using difference-list implementation (Bratko, 2001, Chapter 21). As a result, the predicate exp in the query has two additional arguments than are found in the DCG specification. The semantic action code inside curly braces incorporates calls to SWI-Prolog library functions.

The DCG corresponding to the set-based attribute grammar is as follows.

```
exp(Ids)    --> aexp(Ids).
exp(Ids)    --> aexp(Ids1), ['+'] , exp(Ids2), {union(Ids1,Ids2,Ids)}.
aexp([Id])  --> [Id], {atom(Id)}.

/* ?- exp(Vs, [x, '+', y, '+', z, +, y],[]). */
 /* Vs = [x, z, y] ; */
```

The DCG corresponding to the list-based attribute grammar is as follows.

```
exp(Envi,Envo)    --> aexp(Envi,Envo).
exp(Envi,Envo)    --> aexp(Envi,EnvT), ['+'] , exp(EnvT,Envo).
aexp(Envi,Envo)   --> [Id], {atom(Id)},
                      {member(Id,Envi) -> Envo = Envi;
                                          Envo = [Id | Envi]}.

 /* ?- exp([],Vs, [x, '+', y, '+', z, +, y],[]). */
 /* Vs = [z, y, x] ; */
```

The sample queries and the results have been commented out. Refer to (Bratko, 2001, Chapter 21) for DCG details.

*Specifying Compiler Generator Operations*

Attribute grammars can be used to formalize and implement parser generators. For instance, consider the computation of nullable non-terminals, first-sets associated with non-terminals and follow-sets associated with non-terminals. A non-terminal is *nullable* if it can derive a null string. The *first-set* associated with a non-terminal is the set of tokens that can begin a string derivable from the non-terminal. The *follow-set* associated with a non-terminal is the set of tokens that can come after the non-terminal in a sentential form. These can be specified for an expression grammar along the lines indicated below. (Only partial specification has been given.)

$$S ::= T\ E$$
$$E ::= \varepsilon \mid +\ S$$
$$T ::= x \mid y$$

$$S.nullable = T.nullable \ \text{ and } E.nullable$$
$$E.nullable = \text{true}$$
$$T.nullable = \text{false}$$

$$S.first\text{-}set \ = \ \text{if } T.nullable \ \text{ then } T.first\text{-}set \cup E.first\text{-}set \ \text{ else } T.first\text{-}set$$
$$E.first\text{-}set \ = \ \{\ \varepsilon\ ,\ +\ \}$$
$$T.first\text{-}set \ = \ \{\ x\ ,\ y\ \}$$

$$T.follow\text{-}set \ = \ \text{if } E.nullable \ \text{ then } S.follow\text{-}set \cup E.first\text{-}set \ \text{ else } E.first\text{-}set$$
$$S.follow\text{-}set \ = \ E.follow\text{-}set$$
$$E.follow\text{-}set \ = \ S.follow\text{-}set$$

The dependencies among the various attributes can be exploited to sequence their computation in three phases: compute *nullable* first, followed by *first-sets*, followed by *follow-sets* (Bochmann, 1976). Each phase can potentially require multiple iterations to converge to a fixed-point. The *nullable* is a synthesized attribute and requires multiple bottom-up pass. The *first-set* is also a synthesized attribute computed using multiple bottom-up left-to-right pass. The *follow-set* is an inherited attribute computed using multiple top-down right-to-left pass.

*Optimizing Bottom-up Database Query Evaluation*

Top-down query evaluation and bottom-up query evaluation are two well-known deductive database query implementation strategies (Ramakrishnan & Sudarshan, 1991). In top-down approach, the query solution tree is grown from the root (goal) to the leaves (data), applying the datalog (function-free Horn-logic or Prolog) rules from left to right. It is potentially incomplete in

the presence of left-recursive rules, but it is efficient because the bindings in the queries are propagated to the base relations (data). In bottom-up approach, the query solution tree is grown from the leaves (data) to the root (goal), applying the datalog rules from right to left. It is complete, but can be inefficient because the search is not goal directed. Memoing techniques can be used to improve top-down strategy by making its search complete, while magic predicates can be employed to improve bottom-up strategy by making it more focused and efficient (Beeri & Ramakrishnan, 1987). The ideas underlying the definition of magic predicates and sideways information processing are reminiscent of attribute computation rules involving inherited and synthesized attributes as explained and illustrated below.

Consider the definition of the ancestor relation based on the parent relation, and the ancestor query with the first argument bound.

$$ancestor(X,Y) :- parent(X,Y).$$
$$ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).$$
$$?- ancestor(john, N).$$

In order to explore only a small fragment of the necessary parent facts to compute the query answers using naïve bottom-up approach, it is important to restrict the "firing" of the second rule. This can be done by transforming the second rule using *magic* predicates so that it is satisfied only for the ancestors of john as follows.

$$magic(john).$$
$$magic(Z) :- magic(X), parent(X,Z).$$
$$ancestor(X,Y) :- magic(X), parent(X,Y).$$
$$ancestor(X,Y) :- magic(X), parent(X,Z), ancestor(Z,Y).$$

The rules for *magic* predicate are analogous to the computation of attributes. The first fact corresponds to the inherited attribute value corresponding to the first argument of ancestor initialized by the top-level query. The second rule corresponds to the computation of additional inherited attribute values corresponding to the first argument of ancestor obtained through sideways information passing of the synthesized attribute from the parent relation. Beeri and Ramakrishnan (1987) provide several additional examples embodying attribute grammars ideas for query optimization.

*Specifying Algorithms for Customized Information Extraction*

In this section, we discuss an information extraction problem of industrial significance that has benefited from attribute grammar based algorithm specification. We skip the detailed specification *per se* due to space constraints.

A typical materials and process spec (from authoring organizations such as GE, Pratt and Whitney, ASTM, AMS, etc) contains requirements for making and testing a variety of alloys. A typical customer order specifies the

desired material in terms of the specs it must conform to, and a collection of domain-specific product parameters such as product type, spec class, product dimension and cross-section, etc. A fundamental operation of interest on a spec is the determination of applicable fragments of the spec for a customer order. The goal of coarse-grain extraction is to convert a spec into a form that can be evaluated against the order parameters to determine applicable fragments.

To balance the commercial viability of the extraction task and its tractability, a spec is transformed into a possibly, nested sequence of conditioned notes of the form "If CONDITIONS Then [Note = " ... "]", where the note contains contiguous block of spec text. These extractions are cheap to produce because the detailed requirements are still in text.

In order to formalize conditioned notes, we need to propose a structure for the conditional expression and the note it qualifies. The conditional expression can be formed using characteristic names (e.g., spec class, alloy, product type, etc), constants, relational symbols (e.g., `=', `$>$', etc), and boolean connectives (e.g., `and', `or', etc) in the standard way. The note can be defined in quanta of (sub)-sections and paragraphs. Thus, there are two important technical problems to be solved for carrying out extraction: (1) Identification of the values of a characteristic that can appear as a condition, and (2) Transformation of the relevant spec text into a sequence of conditioned notes.

The conditioned notes can be specified in terms of the scope rules of applicability of charatcteristic-value pairs to the spec text fragments. For instance, to associate a spec class with a section or a paragraph, we use the following heuristic: Every (sub-)section is conditioned on all spec classes named in section 'Scope'. Explicit spec class references in a paragraph override the default condition. Otherwise, a paragraph inherits the condition from its left sibling (earlier paragraph), or transitively from its parent (enclosing (sub-)section). The rationale behind the heuristic is that, when the conditionals in an extraction are evaluated against the given condition values, it should generate all applicable fragments of the spec.

To abstract the algorithmic details of the extraction, the structure of a spec can be captured using EBNF as

&lt;document&gt; ::= &lt;document-header&gt; &lt;section&gt;+
&lt;section&gt; ::= &lt;sectionNumber&gt; &lt;sectionHeading&gt; &lt;paragraph&gt;+ &lt;section&gt;*
and the computation of the conditioned notes involving spec classes can be given using attribute grammars. There are many other qualifiers of interest besides spec classes such as products, product types, alloys, etc. (See (Thirunarayan et al, 2005) for a detailed attribute grammar specification.)

# FUTURE TRENDS

Information flow ideas underlying attribute grammars can provide a general framework for designing and specifying algorithms. For example,

Neven (2005) and Neven, F., and den Bussche, J. V. (2002) demonstrate the influence of attribute grammars on query languages.

Web technologies such as <mark>XML/XSLT</mark> that are based on adorned context-free grammars can benefit from techniques developed for attribute grammars (Harold, 2004). Conceptually, an XML document consists of annotations, where each annotation consists of an associated XML-element that reflects the semantic category to which the corresponding text fragment belongs, and the associated XML-attributes that are bound to relevant semantic values. Overlaying domain-specific XML tags on a text document enables abstraction, formalization, and in-place embedding of machine-processable semantics. In the future, we can expect the annotated data to be interpreted by viewing it as a function/procedure call, and defining the XML-element as a function/procedure in a language such as XSLT or Water for associating different collections of behaviors with XML-elements (Thirunarayan, 2005).

# C O N C L U S I O N

Historically, attribute grammars were developed in the context of compiler construction. We provided several examples illustrating the application of attribute grammars for static analysis of programs, for program translation, and for specifying certain phases of a parser generator. We also provided general principles for developing attribute grammar specifications for efficient one-pass computation of attributes. We introduced Prolog's definite clause grammars to enable attribute grammars to be viewed as executable specifications. We showed how the information flow ideas implicit in the attribute computation rules can be exploited to optimize bottom-up evaluation of datalog programs. Finally, we discussed the application of attribute grammars for specifying information extraction algorithms, and its future role in XML technologies.

# R E F E R E N C E S

Aho, A. V., Lam, M., Sethi, R. and Ullman, J. D. (2007) *Compilers: Principles, Techniques, and Tools*, Addison Wesley.

Beeri, C., and Ramakrishnan, R. (1987) On the Power of Magic, *Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 269-283.

Bratko, I. (2001) *Prolog Programming for Artificial Intelligence*, Third Edition, Addison Wesley.

Bochmann, G. V. (1976) Semantic evaluation from left to right, *Communications of the ACM, 19*(2), 55-62.

Harold, E. R. (2004). *XML in a Nutshell,* Third Edition, O'Reilly.

Knuth, D. E. (1968) Semantics of context-free languages. *Mathematical Systems Theory 2*, 2, 127-145. (Corrigenda: *Mathematical Sysems Theory 5*, 1, 1971, 95-96.)

Lindholm, T. and Yellin, F. (1999). *Programming Java Virtual Machine Specification,* Second Edition, Addison-Wesley.

Louden, K. C. (2003). *Programming Languages : Principles and Practice*, Second Edition, Thomson – Course Technology.

Neven, F. (2005) Attribute grammars for unranked trees as a query language for structured documents, *Journal of Computer and System Sciences, 70,* 221-257.

Neven, F., & den Bussche, J. V. (2002) Expressiveness of structured document query languages based on attribute grammars, *Journal of the ACM, 49*(1), 56-100.

Paakki, J. (1995) Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation, *ACM Computing Surveys, 27*(2)*,* 196-255.

Ramakrishnan, R., and Sudarshan, S. (1991) Top-Down versus Bottom-Up Revisited, *Proceedings of the 1991 International Symposium on Logic Programming*, 321-336.

Thirunarayan, K., Berkovich, A., and Sokol, D. (2005) An information extraction approach to reorganizing and summarizing specifications, *Information and Software Technology Journal*, *47*(4), 215-232, 2005.

Thirunarayan, K. (2005) On Embedding Machine-Processable Semantics into Documents, *IEEE Transactions on Knowledge and Data Engineering*, *17*(7), 1014-1018.

Thirunarayan, K. (1984) *A Compiler-Generator Based on Attributed Translation Grammars*, M. E. Thesis, Indian Institute of Science, Bangalore. (Advisors: Priti Shankar and Y. N. Srikant)

## Terms and Definitions

**EBNF**: Extended Backus Naur Formalism is an extension of context-free grammar with regular expression operations for defining context-free languages. It provides a more concise syntax specification.

**Attribute Grammar**: An attribute grammar is an extension of context-free grammar that enables definition of context-sensitive aspects of a language and its translation.

**Synthesized Attributes**: These attributes pass information from leaves to the root of a parse tree.

**Inherited Attributes**: These attributes pass information from root to the leaves of a parse tree, or sideways among siblings.

**Attribute Computation Rules**: Rules used to compute attributes, usually defined in terms of other attributes and standard primitives.

**DCG:** A Definite Clause Grammar is a Prolog built-in mechanism for implementing attribute grammars efficiently using difference lists.

**Machine-processable Semantics:** Metadata added to the documents to enable machines to understand and reason with text or multi-media content.

**XML/XSLT**: eXtensible Markup Language is a meta-language for creating markup languages. XML is a subset of SGML. XHTML is an instance of XML. eXtensible Stylesheet Language Transformations is a language for manipulating XML documents.