Attribute Grammars and Their Applications

Krishnaprasad Thirunarayan
Wright State University, USA

INTRODUCTION

Attribute grammars are a framework for defining semantics of programming languages in a syntax-directed fashion. In this chapter, we define attribute grammars, and then illustrate their use for language definition, compiler generation, definite clause grammars, design and specification of algorithms, and so forth. 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, Lam, Sethi & Ullman et al., 2007,). 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::= α, where A ∈ N and α ∈ (N U T)*. S ∈ 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 ∈ N U T with a set of attributes, and AR associates each production R ∈ P with a set of attribute computation rules (Paakki, 1995). A(X), where X ∈ (N U T), can be further partitioned into two sets: synthesized attributes S(X) and inherited attributes I(X). AR(R), where R ∈ 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.

N ::= 0	N.*val* = 0
N ::= 1	N.*val* = 1
N ::= N0	N_l.*val* = 2 * N_r.*val*
N ::= N1	N_l.*val* = 2 * N_r.*val* + 1

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.

N ::= 0	N.*val* = 0, N.*len* = 1
N ::= 1	N.*val* = 1, N.*len* = 1
N ::= 0N	N_l.*val* = N_r.*val*; N_l.*len* = N_r.len +1
N ::= 1N	N_l.*val* = 2^ N_r.*len* + N_r.*val*; N_l.*len* = N_r.len +1

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.

F ::= . N	F.*val* = N.*val*, N.*pow* = 1
N ::= 0	N.*val* = 0
N ::= 1	N.*val* = (1 / 2^ N.*pow*)
N ::= 0N	N_l.*val* = N_r.*val*; N_r.*pow* = N_l.pow +1
N ::= 1N	N_l.*val* = (1 / 2^ N.*pow*) + N_r.*val*; N_r.*pow* = N_l.pow +1

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)). By *modular*, we emphasize its role in structuring specification that is incremental with respect to the produc-

tions. 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). 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). Definite clause grammars enable attribute grammars satisfying certain restrictions to be viewed as executable specifications (Bratko, 2001). 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, Berkovich, & Sokol, 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.

E ::= n E.*typ* = int
E ::= x E.*typ* = double
E ::= + E E E.*typ* = if E_{r1}.*typ* = E_{r2}.*typ*
then E_{r1}.*typ* else double

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 \= 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 E.*code* = [iload_1]
E ::= x E.*code* = [dload_2]
E ::= E + E E.*code* = if E_{r1}.*typ* = int
then if E_{r2}.*typ* = int
then E_{r1}.*code*@E_{r2}.*code*@[iadd]
else E_{r1}.*code*@[i2d]@E_{r2}.*code*@[dadd]
else if E_{r2}.*typ* = int
then E_{r1}.*code*@E_{r2}.*code*@[i2d,dadd]
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). (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 attribute computation 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. ('∪' refers to set-union.)