

Java Implementation of Demand-Driven Attribute Grammar Evaluation

Görel Hedin

Dept of Computer Science, Lund University, Sweden.
Gorel.Hedin@cs.lth.se

Abstract Any non-circular attribute grammar can be evaluated by demand-driven evaluation where each attribute is viewed as a function. Object-oriented languages are convenient for providing such an implementation, relying on inheritance and virtual methods. This paper shows how a demand-driven evaluator can be implemented very conveniently in Java, also making use of Java's interface mechanism.

1 Introduction

There is a large variety of algorithms for evaluation of Attribute Grammars (AGs) [2]. One particularly simple and general algorithm is the demand-driven algorithm where each attribute is viewed as a function. To evaluate a particular attribute, its corresponding function is called, and this function will in turn call the functions corresponding to its dependent attributes, resulting in a recursion which is guaranteed to terminate if the grammar is non-circular. The algorithm can be made optimal by caching the value of an attribute the first time its function is called, and returning that value directly at subsequent calls (similar to lazy evaluation). The algorithm works for any non-circular attribute grammar, and it is also easy to add a dynamic circularity-check by using a mark-bit for each attribute, so that termination is guaranteed also for circular grammars.

Several systems have made use of this algorithm, e.g. Madsen [10], and a variety of different implementation styles have been proposed. The algorithm is also applicable to extended AG formalisms including non-local connections, known as occurrence attributes or reference attributes [11, 5]. In Engelfriet's survey of evaluation methods [2], the demand-driven algorithm is referred to as "P4", and the optimal variant as "P5". Other descriptions of the algorithm includes Jalili's [6]. Several researchers have showed how the algorithm lends itself to implementation in different kinds of languages: Jourdan gives an implementation in Lisp [8]. Johnsson shows how a functional language with lazy evaluation can be employed [7], Hedin shows how object-oriented constructs like inheritance and virtual methods can be used [3, 4].

In this paper we show how Java can be used to implement the algorithm. In relation to the earlier work on object-oriented implementation, we show how Java interfaces are particularly suited for the implementation of inherited attributes.

As a running example we use Knuth's example on binary numbers where numbers like 1, 101, 10.01 etc. are modelled as syntax trees and their decimal values (like 1, 5, and 2.25) are defined using attributes [9].

2 Basic translation

In this section we show how an attribute grammar can be translated to a Java program, implicitly providing the demand-driven evaluation algorithm in a basic form, without caching attribute values and without doing circularity check.

2.1 Base class

A base class, `ASTnode`, modelling an abstract syntax tree node is defined as follows:

```
abstract class ASTnode {
    ASTnode father; (* Denotes the father node of this node. *)
                  (* Null for the root node. *)
};
```

2.2 Grammar class

The nonterminals $X_1 \dots X_n$, and productions $p_1 \dots p_m$ of the grammar are modelled as inner static classes of a class `Grammar` as follows:

```
import ASTnode;

class Grammar {
    static class  $X_1$  ...
    ...
    static class  $X_n$  ...

    static class  $p_1$  ...
    ...
    static class  $p_m$  ...
};
```

The outer class `Grammar` only serves the role of a module for containing the nonterminal and production classes. Therefore, the inner classes are declared as static which means that they are possible to instantiate without any instance of the outer class `Grammar`. The alternative in Java to using inner classes would have been to put each of the nonterminal and production classes in a separate file, which we find less practical. *** is this the only alternative??? ***.

2.3 Nonterminals and productions

A nonterminal X is translated to an abstract class `X` that extends the base class `ASTnode` as follows:

```
abstract static class X extends ASTnode ...
```

A production $p: X_0 \rightarrow X_1 \dots X_{n(p)}$ (where p is the name of the production) is translated to a class p that extends the nonterminal class X_0 and declares instance variables for the children $s_1 \dots s_{n(p)}$ qualified by the types $X_1 \dots X_{n(p)}$ respectively. The names of the instance variables are chosen so that they are all unique. The class p also declares a constructor taking $n(p)$ arguments (one for each child) and sets the corresponding instance variables. The constructor also sets the **father** reference (inherited from **AST-node**) of each of the sons to denote the new node. For example, the following production

RationalNumber : Number -> List List

is translated to the following Java class:

```
static class RationalNumber extends Number {
    (* Instance variables for the son nodes *)
    List List1;
    List List2;

    (* Constructor setting the son instance variables and their father
    references. *)
    RationalNumber(List List1param, List List2param) {
        List1 = List1param;
        List2 = List2param;
        List1.father = this;
        List2.father = this;
    };
};
```

2.4 Synthesized attributes

Synthesized attributes correspond exactly to parameterless virtual methods: Declarations of synthesized attributes are translated to virtual method specifications, and equations defining such attributes are translated to virtual method implementations. For example, consider a nonterminal **Bit** with a synthesized attribute **val**:

```
Bit:
    syn val: double
```

This attribute declaration is translated to an abstract virtual method in the nonterminal class **Bit** as follows:

```
abstract class Bit extends ANYnode {
    abstract double val();
};
```

Equations defining the `val` attribute occur in the productions for `Bit`, namely `Zero` and `One`:

`Zero:`
`val = 0`

`One:`
`val = 2.0scale`

where `scale` is an inherited attribute in `Bit`. These equations are translated to virtual method implementations in the production classes `Zero` and `One` as follows:

```
class Zero extends Bit {  
    double val() {return 0.0; };  
}  
  
class One extends Bit {  
    double val() {return Math.pow(2.0, scale()); };  
}
```

Note that the use of the `scale` attribute in the equation in production `One` is translated to a call “`scale()`” to the method implementing that attribute. There is no built-in operator for power in Java, so the standard method `Math.pow(x,y)` is used to implement x^y .

2.5 Inherited attributes

An inherited attribute is defined by an equation in the father node. To represent this in Java, we add an interface `FatherOfX` for each nonterminal `X` declaring an inherited attribute, and each production which has son nodes of type `X` must implement this interface. The interface contains virtual methods for computing the inherited attributes. In addition, a method for each inherited attribute is added to the nonterminal `X` which calls the corresponding method of the father node in an appropriate way.

For example, consider the following grammar fragment where the nonterminal `List` has two attributes: a synthesized attribute `length` and an inherited attribute `scale`, and the production `RationalNumber` has two sons of type `List` and an equation defining the `scale` attribute of each of those `List` sons.

`List:`
`syn length: integer;`
`inh scale: integer;`

`RationalNumber: Number -> List List`
`List1.scale = 0`
`List2.scale = - List2.length`

This grammar is translated in the following steps.

First, a new interface `FatherOfList` is defined which has a virtual method `List_scale` taking a `List` as a parameter:

```
static interface FatherOfList {
    int List_scale(List theList);
};
```

Second, each production which has sons of type `List` must implement this interface, and return the defined value for `scale`. For the production `RationalNumber`, this implementation looks as follows:

```
static class RationalNumber extends Number implements FatherOfList {
    ...
    public int List_scale(List theList) {
        if (theList==List1)
            return 0
        else (* theList== List2 *)
            return -List2.length();
    };
};
```

Note that the `List` parameter is used to test which of the two sons is “asking” for its `scale` attribute. If there is only one `List` son, or if the equations are the same for all `List` sons, the test is of course unnecessary.

Finally, a (non-virtual) method is added to the nonterminal class `List` to represent the `scale` attribute as follows:

```
abstract static class List extends ASTnode {
    final int scale() {return ((FatherOfList) father).List_scale(this); };
};
```

The `scale` method is declared as `final`, meaning that it is not virtual (cannot be overridden). The cast of the `father` to the type `FatherOfList` is always safe since all nonterminals that have sons of type `List` have implemented this interface.

2.6 Creating ASTs and evaluating attributes

ASTs can now be created using the constructor interface of the production classes. For example, to create the AST for the binary number 10.01 we can write

```

Number n;

n =
    new Grammar.RationalNumber(
        new Grammar.CompoundList(
            new Grammar.SimpleList(
                new Grammar.One()
            ),
            new Grammar.Zero()
        ),
        new Grammar.CompoundList(
            new Grammar.SimpleList(
                new Grammar.Zero()
            ),
            new Grammar.One()
        )
    );

```

To evaluate some attribute of the tree, we simply call the corresponding attribute method. For example, the `val` attribute of the `Number n` is computed simply by calling its `val` method as follows:

```

int v;

v = n.val();

```

and `v` gets the value 2.25 as expected.

3 Optimal evaluation

*** to be written ***

4 Circularity check

*** to be written ***

5 Discussion

*** to be written ***

References

1. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley. 1996.
2. J. Engelfriet. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pp 103-137. Cambridge University Press, 1984.
3. G. Hedin. An object-oriented notation for attribute grammars. *ECOOP'89*. BCS Workshop Series, pp 329-345, Cambridge University Press. 1989.
4. G. Hedin. *Incremental Semantic Analysis*. PhD thesis, Department of Computer Science, Lund University, Sweden, March 1992.
5. G. Hedin. Reference Attributed Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 153-172, Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.
6. F. Jalili. A general linear time evaluator for attribute grammars. *ACM SIGPLAN Notices, Vol 18(9):35-44, September 1983*.
7. T. Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Languages and Computer Architecture*. LNCS 274, Springer-Verlag.
8. M. Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming, 6th Colloquium*, volume 167 of *Lecture Notes in Computer Science*, pages 167-178. Springer-Verlag, 1984.
9. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127-145, June 1968.
10. O. L. Madsen. On defining semantics by means of extended attribute grammars. In *Semantics-Directed Compiler Generation*, pp 259-299, LNCS 94, Springer-Verlag, January 1980.
11. A. Poetsch-Heffter. Prototyping realistic programming languages based on formal specifications. *Acta Informatica* 34, 737-772 (1997).

Appendix A Attribute grammar for binary numbers

Nonterminals:

Number, List, Bit

Productions:

IntegerNumber: Number -> List
RationalNumber: Number -> List List
SimpleList: List -> Bit
CompoundList: List -> List Bit
Zero: Bit ->
One: Bit ->

Attribute declarations:

Number:
syn val: real

List:
syn val: real
syn length: integer
inh scale: integer

Bit:
syn val: real
inh scale: integer

Attribute Equations:

IntegerNumber:
this.val = List.val
List.scale = 0;

RationalNumber:
this.val = List1.val + List2.val
List1.scale = 0
List2.scale = - List2.length

SimpleList:
this.val = Bit.val
this.length = 1
Bit.scale = this.scale

CompoundList:
this.val = List.val + Bit.val
this.length = List.length + 1
List.scale = this.scale + 1
Bit.scale = this.scale

Zero:
this.val = 0

One:
this.val = 2.0**scale

Appendix B Java implementation of binary number grammar

```

import ASTnode;

class Grammar {
    abstract static class Number extends ASTnode {
        abstract double val();
    };

    static class IntegerNumber extends Number
        implements FatherOfList{
        List List1;
        IntegerNumber(List List1Param) {
            List1 = List1Param;
            List1.father = this;
        };
        double val() { return List1.val(); };
        public int List_scale(List theList) { return 0; };
    };

    static class RationalNumber extends Number
        implements FatherOfList{
        List List1, List2;
        RationalNumber(List List1Param, List List2Param) {
            List1 = List1Param;
            List2 = List2Param;
            List1.father = this;
            List2.father = this;
        };
        double val() { return List1.val() + List2.val(); };
        public int List_scale(List theList) {
            if (theList==List1)
                return 0;
            else
                return - List2.length();
        };
    };

    abstract static class List extends ASTnode {
        abstract double val();
        abstract int length();
        final int scale() {
            return ((FatherOfList)father).List_scale(this);
        };
    };

    static class SimpleList extends List
        implements FatherOfBit{
        Bit Bit1;
        SimpleList(Bit Bit1Param) {
            Bit1 = Bit1Param;
            Bit1.father = this;
        };
        double val() { return Bit1.val(); };
        int length() { return 1; };

        public int Bit_scale(Bit theBit) { return scale(); };
    };

    static class CompoundList extends List
        implements FatherOfList, FatherOfBit{
        List List1;
        Bit Bit1;
        CompoundList(List List1Param, Bit Bit1Param) {
            List1 = List1Param;
            Bit1 = Bit1Param;
            List1.father = this;
            Bit1.father = this;
        };
        double val() { return List1.val() + Bit1.val(); };
        int length() { return List1.length() + 1; };
        public int List_scale(List theList)
            { return scale() + 1; };
        public int Bit_scale(Bit theBit)
            { return scale(); };
    };

    abstract static class Bit extends ASTnode {
        abstract double val();
        final int scale() {
            return ((FatherOfBit)father).Bit_scale(this);
        };
    };

    static class Zero extends Bit {
        Zero() {};
        double val() { return 0.0; };
    };

    static class One extends Bit {
        One() {};
        double val() { return Math.pow(2.0,scale()); };
    };

    static interface FatherOfList {
        int List_scale(List theList);
    };

    static interface FatherOfBit {
        int Bit_scale(Bit theBit);
    };
};

```