DOI:10.2298/CSIS111223022R

A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools

Daniel Rodríguez-Cerezo[1], Antonio Sarasa-Cabezuelo[1], and José-Luis Sierra[1]

[1] Computer Science School,
Complutense University of Madrid
Calle Profesor José García Santesmases, s/n
28040 Madrid, Spain
{drcerezo, asarasa, jlsierra}@fdi.ucm.es

Abstract. This article describes structure-preserving coding patterns to code arbitrary non-circular attribute grammars as syntax-directed translation schemes for bottom-up and top-down parser generation tools. In these translation schemes, semantic actions are written in terms of a small repertory of primitive attribution operations. By providing alternative implementations for these attribution operations, it is possible to plug in different semantic evaluation strategies in a seamlessly way (e.g., a demand-driven strategy, or a data-driven one). The pattern makes possible the direct implementation of attribute grammar-based specifications with widely-used translation scheme-driven tools for the development of both bottom-up (e.g. YACC, BISON, CUP) and top-down (e.g., JavaCC, ANTLR) language translators. As a consequence, initial translation schemes can be successively refined to yield final efficient implementations. Since these implementations still preserve the ability to be extended with new features described at the attribute grammar level, the advantages from the point of view of development and maintenance become apparent.

Keywords: Attribute Grammars, Parser Generators, Language Processor Development Method, Grammarware

1. Introduction

Attribute grammars were introduced by Donald E. Knuth [25] as an extension of context-free grammars for describing the syntax and semantics of context-free languages, and are widely used as a high-level specification method for the first stages of the design and implementation of a computer language [2][35].

In order to make an attribute grammar-based specification executable, it is possible to use one of the many specialized tools that support the formalism

(see, for instance,[12][17][31][33][35]). However, regardless the recognized advantages of these tools, in practice, traditional implementations of language processors are rarely based on artifacts directly generated from attribute grammars. On the contrary, attribute grammars are taken as initial specifications of the tasks to carry out, while final implementations are usually achieved by using scanner and parser generators (e.g., ANTLR, CUP, Flex, Bison…), general-purpose programming languages, or a suitable combination of the two techniques [2]. The process of transforming the initial specification into a final implementation is usually ill-defined, and typically depends solely on the programmer's art –a programmer who many times discards formal specifications while he or she directly hacks the final implementation. It seriously hinders the systematic development and maintenance of language processors.

In order to bridge the gap between attribute grammar-based specifications and final implementations, we propose articulating the language processor development process as the explicit transformation of the initial attribute grammar-based specification to the final implementation. According to our proposal, the first step to convey during the implementation stage is to explicitly code the attribute grammar in the input language of the development tool (usually, a parser generator like Bison, CUP, JavaCC or ANTLR). This will make it possible to yield an initial running implementation, which subsequently could be refined to achieve greater efficiency. In addition, since the refined implementation still supports the explicit incorporation and subsequent refinement of attribute grammar-based features, the incremental development and subsequent maintenance of the language processor can be greatly facilitated. Therefore, it is important to notice that the rationale of the present work is not to provide new methods to automatically generate language processors from attribute grammars (in this case, undoubtedly the best choice would be one of the pre-existing tools based on attribute grammars). Instead, the rationale is to start from an attribute grammar specification and then to systematically refine it across several stages, finishing with a final, highly efficient implementation in a conventional compiler construction tool -a process which is not the aim of any typical attribute grammar tool.

This paper is mainly focused on the first step of our proposal, i.e. how to code an attribute grammar in terms of the input language supported by a conventional parser generation tool, although we also illustrate some aspects of the latter refinement. In order to cover the most widely used parser generation tools, we address both bottom-up parser generators of the YACC and CUP type and top-down parser generators of the JavaCC or ANTLR style. Unlike works in L-attributed [28] or LR-attributed grammars [4] and similar approaches (e.g., [23]), our approach will support the implementation of arbitrary non-circular attribute grammars. In addition, the coding pattern will be independent of the final evaluation style chosen. Indeed, attribute grammars will be coded by using a small repertory of *attribution* operations. Finally, by providing alternative implementations for these operations, it will be possible to set up the semantic evaluation style that will finally be used.

The structure of the rest of the paper is as follows: section 2 introduces some preliminaries. Section 3 details the dependency description operations and outlines two alternative implementations, which makes apparent how to plug in different evaluation styles. Section 4 describes the coding pattern for bottom-up parser generation tools. Section 5 describes the pattern for top-down ones. Section 6 presents some work related to ours. Finally, section 7 concludes the paper and outlines some lines of future work. A preliminary version of this work, which only deals with a former pattern for bottom-up translation schemes, can be found in [41].

2. Preliminaries

In this section we introduce some basic concepts concerning the two main language-processing specification tools addressed in this paper: attribute grammars (subsection 2.1) and translation schemes (subsection 2.2).

2.1. Attribute grammars

The formalism of attribute grammars was initially proposed by Donald E. Knuth at the end of the 1960s to characterize the semantics of context-free languages [25]. Attribute grammars introduce a syntax-directed, dependency-driven language processing style. This processing style is syntax-directed because the processing of each sentence is driven by its syntactic structure, and it is dependency-driven because it is directed by the dependencies among the computations involved. Figure 1 shows an example of an attribute grammar that models the evaluation of simple arithmetic expressions, followed by declarations of constants. In the formalized process, declarations are used to build an *environment* (a set of variable-value pairs), which is subsequently used to determine the value of variables. For the sake of conciseness, only the addition operator is considered.

Attribute grammars extend *context-free grammars* with *semantic attributes* and *semantic equations*. Indeed, *context-free* grammars are standard mechanisms to define the syntax of computer languages. In a context-free grammar:

- Syntax is defined by means of *syntax rules* (or *productions*), which determine the structure of syntactic constructions in terms of sequences of simpler constructions. For instance, in Figure 1 `Sent::= Exp` **`where`** `Decs` is a syntax rule that describes the top-level structure of the kind of sentences considered in this example.
- Syntactic constructions are represented by means of *syntax symbols*: composite structures by *non-terminal* symbols and simple structures by *terminal* symbols. For instance, in Figure 1 `Sent`, `Exp` and `Decs` are non-terminal symbols that represent, respectively, sentences,

expressions and declarations. In turn, **`where`**, **`var`** or **`num`** are terminal symbols (these symbols represent, respectively, the *where* reserved word, variables and numbers in the language considered).

- For each non-terminal there are one or several rules defining its structure. Each rule is made up of a *left-hand side rule* (*LHS*; the non-terminal whose structure is defined) and of a *right-hand side rule* (*RHS*; the sequence of symbols which define such a structure). For instance, the previously referred to rule established that a sentence (`Sent`, the rule's LHS) maybe (the rule's RHS): an expression (`Exp`), followed by the **where** reserved word, and followed by a block of declarations (`Dec`).
- There is also a distinct non-terminal (the grammar's initial symbol or the *grammar*'s *axiom*), which represents the language's highest level structure. In Figure 1, the grammar's initial symbol is `Sent`.

```
Sent ::= Exp where Decs
    Exp.env↓ = Decs.env↑
    Sent.val↑ = Exp.val↑
Exp ::= Exp + Opnd
    Exp1.env↓ = Exp0.env↓
    Opnd.env↓ = Exp0.env↓
    Exp0.val↑ = Exp1.val↑ + Opnd.val↑
Exp ::= Opnd
    Opnd.env↓ = Exp.env↓
    Exp.val↑ = Opnd.val↑
Opnd ::= num
    Opnd.val↑ = toNum(num.lex↑)
```

```
Opnd ::= var
    Opnd.val↑ = valOf(var.lex↑,Opnd.env↓)
Opnd ::= (Exp)
    Exp.env↓ = Opnd.env↓
    Opnd.val↑ = Exp.val↑
Decs ::= Decs , Dec
  Decs0.env↑ = extendWith(Dec.env↑,Decs1.env↑)
Decs ::= Dec
  Decs.env↑ = Dec.env↑
Dec ::= var = num
  Dec.env↑ = {(var.lex↑,toNum(num.lex↑))}
```

Figure 1. An example of attribute grammar

In a context-free grammar, syntax rules enable the description of the structure of each language's sentence in terms of a tree, which is called the *parse tree* of the sentence. Inner nodes are non-terminals, while leaves are terminals. Each parent node, together with its ordered sequence of child nodes, corresponds to the application of a syntax rule. Finally, the root node corresponds to the grammar's axiom. Figure 2a shows an example of sentence in the language considered in Figure 1, and Figure 2b shows the parse tree for this sentence. Notice how this tree makes the structure of the sentence explicit. Thus, subsequent processes can be driven by this structure.

As indicated before, an attribute grammar adds a set of *semantic attributes* to the symbols of an underlying context-free grammar. These attributes will take values in the corresponding nodes of the parse trees. Attributes can be of two types:

- *Synthesized attributes*: their values are computed from synthesized attributes in the owner node's child nodes and from the inherited attributes of this owner node. Thus, the value of a synthesized attribute represents (part of) the meaning of the symbol(s) to which this attribute is

associated. In the grammar of Figure 1, synthetized attributes are terminated with ↑. Thus, `val↑` is an example of synthetized attribute in this grammar, which is used to contain the values of operands (`Opnd` non-terminal), expressions (`Exp` non terminal) and sentences (`Sent` non-terminal). In turn, the synthesized attribute `env↑` is used to build the aforementioned environment from declarations. Finally, notice that terminal symbols can also have synthesized attributes; these synthesized attributes are called *lexical attributes*, and they should be set during lexical analysis. For instance, in the grammar of Figure 1 we use a lexical attribute, `lex↑`, which contains the actual string (the *lexeme*) of each token (e.g., for **`num`** it will contain the actual number, for **`var`** the actual variable, …).

(a) `x+y+5` **`where`** `x=5,y=6`

(b)

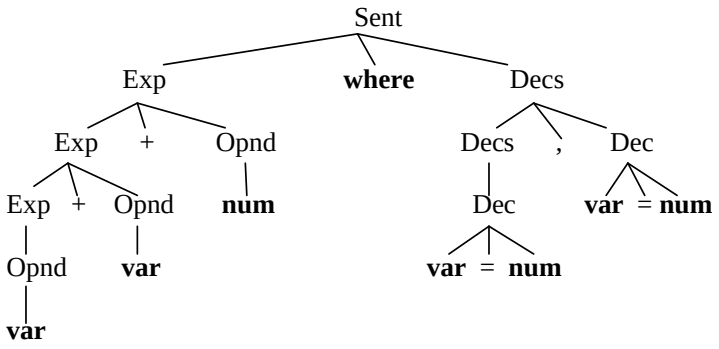

Figure 2. (a) A sentence of the language defined by the context-free grammar behind Figure 1, (b) Parse tree for the sentence in (a)

- *Inherited attributes*: their values are computed from inherited attributes in the parent and/or from synthesized attributes in the siblings. Thus, inherited attributes provide additional contextual information needed to determine the meanings of the symbols to which they are associated. In the grammar of Figure 1, we use an `env↓` inherited attribute to propagate the environment to the *expression* part of the input sentence, since this information is necessary to correctly determine the value of the constant appearing in such an expression part.

The attribute grammar will also add a set of *semantic equations* to each syntax rule. These equations will indicate how to compute the values of synthesized attributes in the rule's LHS, as well as the inherited attributes in the RHS symbols. More precisely:

- There will be exactly one semantic equation for each synthetized attribute on the LHS, and another one for each inherited attribute on the RHS.
- Each equation will apply *semantic functions* to other attributes in the rule. We will assume that, in the computation expressed by each equation, it

will only be possible to use inherited attributes from the LHS and synthesized attributes from the RHS (i.e., we will consider attribute grammars in Bochmann's normal *form* [9]).

For instance, the semantic equation $\texttt{Exp}_0\texttt{.val}\uparrow = \texttt{Exp}_1\texttt{.val}\uparrow + \texttt{Opnd.val}\uparrow$ for the syntax rule `Exp ::= Exp + Opnd` in the grammar of Figure 1 establishes that, in order to compute the value of a sum ($\texttt{Exp}_0\texttt{.val}\uparrow$)[1], it is necessary to add the value of the first operand ($\texttt{Exp}_1\texttt{.val}\uparrow$) to the value of the second operand ($\texttt{Opnd.val}\uparrow$).

Attribute grammars enable *semantic evaluation* on *attributed parse trees* (i.e., parse trees along with the semantic attributes for each node). Semantic evaluation is *dependency-driven*, since it is solely constrained by the dependencies that exist among these semantic attributes (i.e., to compute the value of an attribute, the only rule that must be obeyed is to have the values available of all the other attributes required by this computation according to a suitable semantic equation). Aside from this basic constraint, evaluation order does not matter. In consequence, attribute grammars result in a high-level specification formalism, since it is possible to specify language-processing tasks by focusing on the meaning of the syntax structures, without being distracted by lower-level implementation details, like the exact order in which attribute instances must finally be evaluated. In addition, the formalism is highly modular: it facilitates the addition of new attributes and semantic equations without affecting the existing ones, since the dependencies among attribute instances will be responsible for automatically rearranging the order in which to carry out the evaluation.

A convenient way of describing dependencies between attributes in an attributed parse tree is by means of a *dependency graph.* Nodes in this graph are the attributes in the symbols on the tree. Each arc denotes that the source attribute must be used to compute the value of the target one. Figure 3 shows the attributed parse tree and the dependency graph for the sentence in Figure 2a.

An attribute grammar is *non-circular* when it is not possible to find an attribute instance in a parse tree depending (directly or indirectly) on itself. For the contrary, the grammar is called a *circular* attribute grammar. Although semantic evaluation can be extended to manage circular attribute grammars (see, for instance [19]), for translation purposes non-circular attribute grammars usually suffice. Therefore, in this paper we will deal with non-circular attribute grammars. Semantic evaluation in these grammars can be meaningfully explained as follows [2]:

- First, find a topological order of the nodes in the dependency graph for the sentence being processed (since the grammar is non-circular, the

[1] Notice that, in order to refer to particular occurrences of a non-terminal symbol in a rule, it is possible to use subscripts: thus, Exp_0 refers to the first occurrence of Exp, Exp_1 to the second occurrence, etc.

dependency graph will be acyclical). In this order, each attribute instance will precede all the attribute instances depending upon it.

- Then, evaluate the attribute instances according to this order:

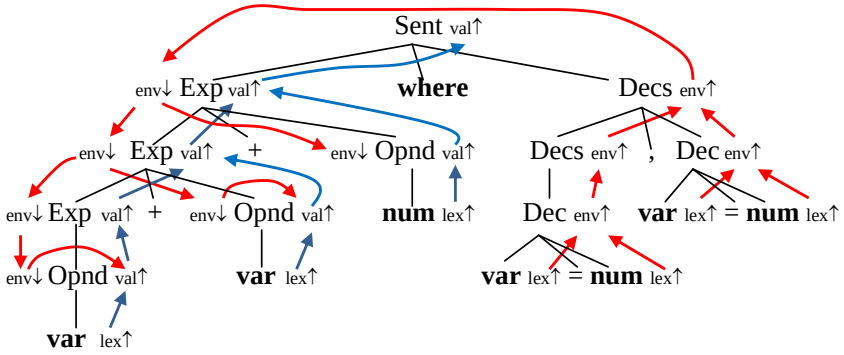

Figure 3. Attributed parse tree and dependency graph for the sentence in Figure 2a

However, it is only a conceptual execution model. In practice, semantic evaluation can be carried out by following different strategies which are only constrained by dependencies among attributes. Also, a particular evaluation strategy may not require the explicit construction of a parse tree. In fact, for remarkable subclasses of attribute grammars (many *s-attributed grammars*, which only involve synthesized attributes, and some classes of *l-attributed grammars*, in which inherited attributes of symbols only depend on the inherited attributes of their parents and synthesized attributes of their preceding siblings), it is possible to yield implementations that evaluate the attributes *on-the-fly* during parsing of the input sentence, without requiring the explicit construction of the syntax tree. Notice the grammar in Figure 1 is not s-attributed (it is needed to propagate the environment to the expression in order to evaluate it), nor l-attributed (because declarations are placed after the expression, and constant values are required to compute the value of such an expression).

2.2. Translation schemes

Translation schemes constitute another formalism that extends context-free grammar to allow the specification of syntax-directed processing [2]. For this purpose:

- Translation schemes adopt explicit visit orders for the nodes of the parse trees. Although many others are possible, two well-known visit orders are *left-to-right bottom-up* and *top-down* ones. In both of them child nodes are visited from left-to-right. However, in a bottom-up visit, nodes are visited in post-order, while in a top-down visit are visited in pre-order. In

addition, in a bottom-up visit order the visit to each node has only one *significant point*, once all its children have been visited. On the other hand, in a top-down one there are many significant points: (i) when the node is entered the first time, (ii) after a child has been exited and before the next one is entered, and (iii) when the node itself is exited.

- Translation schemes also adopt explicit ways of storing computed semantic information. For this purpose, it can be stored in semantic attributes, as in the case of attribute grammars, but also by using other means. For instance, typical execution models for bottom-up translation schemes use stacks for storing semantic information, while typical execution models for top-down ones assume implementations based on mutually recursive subprograms and use subprogram parameters and the runtime stack as a semantic storage mechanism. In addition, both bottom-up and top-down translation schemes can use global variables to facilitate some translation tasks.
- These artifacts conceive of the syntax rules as *visit plans*. For this purpose, they introduce a *semantic reference* mechanism to consult and update semantic information, as well as interleave chunks of code (*semantic actions*) at those points of the rule's RHS corresponding to significant visit points. Semantic actions will be executed each time the corresponding significant visit point is reached during the translation process. In particular, in bottom-up translators it will be possible to place a semantic action at the end of each syntax rule, while in top-down ones it will be possible to place semantic actions in any point of the rules' RHSs. In consequence, the latter will allow more natural translation patterns than the former. This is particularly true for the managing of inherited semantic information.

Although, in principle, translation schemes are independent of parser generation tools, as they can be conceived of as artifacts for processing parse trees, they are usually used as input specification formalisms for these tools. The resulting tree processors are then coupled with the parsing algorithms, and the explicit construction of the parse trees is definitively avoided. In particular:

- Bottom-up translation schemes are used as input to shift-reduce, LR parser generation tools of the YACC type (e.g., YACC, Bison, CUP, …). The resulting parsers use a stack to attach a semantic value to each syntax symbol, and they can also use global variables to manage additional semantic information. These tools constrain underlying context-free grammars to the LR type (usually, LALR(1) grammars) [2], although there are tools accepting more general grammars (e.g.,30).
- Top-down translation schemes are used as input to predictive descent parser generation tools of the JavaCC or ANTLR type. Since these tools

usually generate recursive descent parsers[2], semantic information is managed as parameters and return values of the subprograms generated, as well as in global variables, and the explicit construction of the parse tree is also avoided. These tools usually impose stronger constraints on the underlying context-free grammars: LL grammars. Although modern generation tools like ANTLR provide many useful extensions to basic LL(k) grammars (in particular, it supports the so-called LL(*) parsing method, which provides unbounded look-ahead enabled by finite-state predictors [37][38]), they are unable to manage features like left-recursion. However, as indicated before, they enable more natural mechanisms for dealing with inherited information.

Figure 4a shows an example of a bottom-up translation scheme. The language processed is the classical language of binary numbers proposed by Knuth in [25] to illustrate basic concepts in attribute grammars, and the processing task is to compute the values of the numbers. As in the other bottom-up translation schemes in this paper, we do not commit to any particular generation tool, and we do use a YACC-like notation [2] to refer to semantic values of symbols in the parse stack. Figure 4b shows a top-down, predictive-recursive translation scheme for this task. The underlying grammar is changed to LL(1), and the semantic actions are changed in consequence. Therefore, it will allow its implementation by using any of the mentioned top-down parser generation tools. As in the case of bottom-up translation schemes, we will not commit to particular generators. In addition, we will use ↓ to annotate input parameters and ↑ to annotate output ones.

(a)

```
Num ::= Num Bit {$$ := $1*2+$2}
Num ::= Bit {$$ := $1}
Bit ::= 0 {$$:=0}
Bit ::= 1 {$$:=1}
```

(b)

```
N(↑v) ::= Num(0,v)
Num(↓cv,↑v) ::= Bit(vb) RNum(vb,v)
RNum(↓cv,↑v) ::= Bit(vb) RNum(cv*2+vb,v)
RNum(↓cv,↑v) ::= {v := cv}
Bit(↑v) ::= 0 {v := 0}
Bit(↑v) ::= 1 {v := 1}
```

Figure 4. (a) An example of bottom-up translation scheme

3. The Attribute Evaluation Framework

Our coding pattern is largely based on the explicit description of the attribution structure of each grammar rule. For this purpose, we needed to develop an attribute evaluation framework, to be used in the semantic actions of the translation schemes. In this section we describe such a framework. For this purpose:

[2] It is also possible to generate non-recursive, table-driven descent parsers [2], but the mainstream in top-down parser generators is geared to the *recursive* model.

- Subsection 3.1 describes the set of basic *attribution operations* used in the translation schemes. These attribution operations make it possible to describe, for each syntax rule: (i) the dependencies between attribute occurrences in the symbols of this rule, and (ii) the functions to be used in order to compute the value of the attributes. They also make it possible to build *semantic contexts* for syntax rules (i.e., tables of references to attributes), to consult and set the value of individual attributes, and to control garbage collection.
- Subsection 3.2 introduces *semantic function managers* as the main extension points of the framework. Semantic function managers are the components used to execute semantic functions.
- Finally, subsections 3.3 and 3.4 describe two alternative implementations of the attribution operations, each based on a different *evaluation style* (a *demand-driven* style in subsection 3.3, and a *data-driven* one in subsection 3.4). In the demand-driven evaluation style, the values of attributes are computed in a lazy way, as they are required. On the other hand, in the data-driven style, values of attributes are computed in an eager way, as soon as the values of the attributes on which they depend become available. These implementations can be interchanged in a transparent way, without further changes in the translation schemes.

3.1. Attribution Operations

Table 1 outlines the repertory of basic attribution operations along with their intended meanings. As such a description makes apparent, the purpose of these operations is to provide the developer with the tools necessary to describe how the *attribute dependency graph* associated with a sentence can be built as this sentence is analyzed by the parser. In addition, it also lets the developer indicate the semantic functions for computing each attribute instance. It does not necessarily mean the graph must be fully stored in memory: depending on the actual implementation of the attribution operations, it will be possible to optimize, to a greater or lesser extent, the heap footprint, as the following subsections make apparent.

Table 1. Attribution operations

Operation	Intended Meaning
mkCtx(*n*)	It creates and initializes a *semantic context*: the list of attribute instances for a syntax symbol.
mkDep (a_0, a_1)	It sets a dependency between two attribute instances. Indeed, it declares that the attribute instance a_0 depends on the attribute instance a_1.
inst(*a*,*f*)	It *instruments* the attribute instance *a* by establishing *f* as the semantic function to be applied during evaluation (*f* is actually an integer identifier of such a semantic function)
release(*as*)	It invokes garbage collection on the attribute instance list *as*.
release(*a*)	It invokes garbage collection on the attribute instance *a*
set(*a*,*val*)	It fixes the value of the attribute instance *a* to *val*.
val(*a*)	It retrieves the value of the attribute instance *a*.

3.2. Semantic Function Managers

Before proceeding with the implementation of the attribution operations, it is convenient to introduce the concept of *semantic function manager*. In our approach, given a particular attribute grammar, the *semantic function manager* is an auxiliary component that supports the execution of semantic functions. Therefore, it is the main extension point of the evaluation framework, since it makes it possible to tailor it to each particular attribute grammar.

A semantic function manager can be conceived as a procedure that, taking the semantic function's identifier and the sequence of attribute instances as input, returns the result of applying the function to the attribute instances. It is important to remark that this component must be provided for each particular attribute grammar. Nevertheless, in our minimalistic conceptualization, we will assume this manager has the pre-established name `exec`. The implementation of this `exec` procedure will be changed from coding to coding[3].

As an example, Figure 5 depicts the pseudo-code for a semantic function manager for the grammar in Figure 1. Notice that, for each equation it is necessary to: (i) substitute attribute references in the equation's RHS for values of the semantic function manager's attribute arguments (e.g., Exp_1`.val`↑ `+ Opnd.val`↑ becomes `val(ARGS[0])+ val(ARGS[1])`, and (ii) associate a suitable integer number to the underlying semantic function (e,g., the `ADD` constant in Figure 4).

```
def IDEN=0; def ADD=1; def TONUM=2; def VALOF=3;
def EXTEND=4; def SINGLEENV=5;

procedure exec(FUN,ARGS) {
case FUN of
  IDEN →
   return val(ARGS[0]);
  ADD →
   return val(ARGS[0])+ val(ARGS[1]);
  TONUM →
   return toNum(val(ARGS[0]));
  VALOF →
   return valOf(val(ARGS[0]),val(ARGS[1]));
  SINGLEENV →
   return {( val(ARGS[0]), toNum(val(ARGS[1])) ) }
  EXTEND →
   return extendsWith(val(ARGS[0],val(ARGS[1]))
end case
}
```

Figure 5. Semantic function manager for the attribute grammar in Figure 1

[3] Although it is possible to achieve more elegant solutions by using a programming language with minimal higher-order support (e.g., a conventional object-oriented language), our conceptualization is deliberately maintained as simple as possible to preserve the essence of the evaluation approaches.

3.3. Demand-Driven Evaluation

According to the *demand-driven* evaluation style, semantic evaluation starts once the sentence has been completely parsed (see, for instance [18][29]). At this point, there is an in-memory representation of the part of the dependency graph required for performing semantic evaluation. During evaluation, the values of the attribute instances will be calculated only when they are required. For the sake of simplicity, we will ignore the detection of potential circularities in the underlying dependency graphs, although it would not be difficult to extend the framework to support it.

The first step in setting this implementation is to decide how to represent semantic attributes. For this purpose, the instances of the semantic attributes can be conceived as records. Table 2 outlines the fields required together with their intended purposes. Thus, this representation makes it possible to build a dependency structure in which:

Table 2. Structure of attribute instances in the demand-driven evaluation framework.

Field	Purpose	Initial value
value	It keeps the value of the instance of the semantic attribute.	$\perp$
available	A boolean flag that indicates whether the value is available.	`false`
deps	It keeps the links to those attribute instances required to compute the value.	*The empty list*
semFun	It stores the integer code of the semantic function required to compute the value.	$\perp$
refcount	A counter of references to this attribute instance (used to enable garbage collection).	1

- Each attribute instance points to those attribute instances required to compute it (in a similar way to the *reversed* dependency graph used in [18]).
- In addition, it explicitly stores the identifier of the semantic function to be used in this computation.

Once this representation is decided, it is possible to proceed with the coding of the operations themselves. Table 3 outlines it using pseudo-code. In this pseudo-code, references are intended to work as in Java, although we do not assume automatic garbage collection (instead, a `delete` primitive is explicitly invoked). Indeed, this is why we explicitly include `release` attribution operations.

The different operations behave as follows:

- `mkCtx` collects, in a list, as many fresh attribute instances as needed. This list actually represents a *semantic context* for a syntax symbol, since it gives access to all its semantic attributes.
- `mkDep` adds the second attribute instance in the `deps` list of the first one.
- `inst` stores the semantic function code in the `semFun` field.
- `release`, when applied to a list of semantic attribute instances, releases each instance and de-allocates the list itself.

- On the other hand, when `release` is applied to an attribute instance, it decreases its reference count by 1. If this count reaches 0, the instances on which it depends are released; finally, the original instance itself is de-allocated.
- `set` sets the `value` field and records its availability.
- `val` recovers the value of an attribute instance as follows: (i) if the value is available, it returns such a value, (ii) otherwise, it calls the semantic function manager to compute such a value and sets and returns it.

Table 3. Implementation of the attribution operations to allow a demand-driven evaluation style

Operation	Implementation	Operation	Implementation

mkCtx(*n*)	*as* := **new list**
	for i := 1 **to** *n* **do**
	add(*as*, **new attribute**)
	end for
	return *as*
		release(*a*)	*a*.refcount:= *a*.refcount-1
			if *a*.refcount = 0 **then**
			foreach *a'* **in** *a*.deps **do**
			release(*a'*)
			end foreach
			delete *a*.deps
			delete *a*
			end if

mkDep (a_0, a_1)	**add** (a_0.deps, a_1) a_1.refcount := a_1.refcount + 1	set(*a*,*val*)	*a*.value := *val* *a*.available := **true**

inst(*a*,*f*)	*a*.semFun := *f*	val(*a*)	**if** ¬ *a*.available **then**
			set(*a*,
			exec(*a*.semFun,*a*.deps))
			release(*a*.deps)
			end if
			return *a*.value

release(*as*)	**foreach** *a* **in** *as* **do** release(*a*) **end foreach** **delete** *as*		

Thus, the demand-driven evaluation process arises from the interplay of the `val` attribution operation and the semantic function manager. Also notice how explicit garbage collection can be readily interleaved in the implementation of the attribution operation by appropriately managing the reference counters and by de-allocating lists and records as soon as they become unreachable. Although in this evaluation style, most of the dependency graph remains in memory until parsing is finished, automatic garbage collection makes it possible to de-allocate useless parts of the graph when they become unreachable. This can be due to attribute instances that are not ultimately required in any computation, or to successive evolutions of the implementation, combining pure attribute grammar features with implementation-oriented optimizations (e.g., global variables, on-the-fly evaluation of semantic attributes, …).

3.4. Data-Driven Evaluation

In the *data-driven* evaluation style, attribute instances are scheduled to be evaluated as soon as the values for all the instances on which it depends are available (see, for instance, [24]). Thus, this method can shorten the duration of attribute instances. Additionally, it can interleave evaluation with parsing. These features can be of interest while processing very long sentences, or sentences made available asynchronously (e.g., on a network communication channel). However, this method can do useless evaluations on attribute instances not required to yield the final results.

Table 4 outlines the representation of attribute instances in this case. Notice that, in addition to the list of instances on which an instance depends, the reverse relationship needs to be maintained (i.e., each attribute instance must refer to those instances which depend on it). Indeed, this representation is similar to that used by networks of *observables-observers* in the *observer* object-oriented pattern [14][4].

Table 4. Structure of attribute instances in the data-driven evaluation framework

Field	Purpose	Initial value
value	It keeps the value of the instance of the semantic attribute.	⊥
available	A boolean flag that indicates whether the value is available.	`false`
deps	It keeps the links to those attribute instances required to compute the value.	*The empty list*
obs	It keeps the links to those attribute instances *observing* it (i.e., which depend on it to compute their values).	*The empty list*
required	Counter which records the number of attribute instances in *deps* whose values have not yet been determined.	0
semFun	It stores the integer code of the semantic function required to compute the value.	⊥
instrumented	*True* if *semFun* was set, *false* otherwise.	`false`
refcount	A counter of references to this attribute instance (used to enable garbage collection).	1

Table 5 outlines the pseudo-code of the attribution operations whose implementation differs from those in the demand-driven style. This way, we only need to redefine `mkDep`, `inst`, `set` and `val`:

- In addition to updating `deps` in the first instance, `mkDep` must test whether the second instance has already been computed. If it is not available, the first instance must be added to its `obs` list, since such an instance depends on its value, which is not yet available.
- Note `inst` must take care of whether the value can be computed. Indeed, if the corresponding attribute instance has all the instances on which it depends computed, it can thereby be computed. It assumes the

[4] As with the demand-driven style, this representation could be simplified by inferring the values of flags (in this case, *available* and *instrumented*) from the other fields. However, we prefer to explicitly preserve these flags to increase the readability of pseudo-code.

establishment of all the required dependencies before instrumentation, which is ensured by our coding pattern.

- `Set` must take care to decrement the `required` counters in all the instances depending on the current one. In addition, if a counter reaches 0, it must force the evaluation of the corresponding instance.
- Finally, `val` immediately computes the value, unless the instance has not yet been instrumented.

Notice how, in this case, evaluation can be interleaved with parsing. Indeed, evaluation is fired when the values of attribute instances are explicitly set, and also when attributes are instrumented. In consequence, garbage collection also interplays with parsing, and, therefore, this method can mean less heap usage. However, this method assumes all the semantic functions used are *strict,* in the sense that all their arguments must be evaluated before they are applied. On the contrary, the demand-driven method described in the previous subsection also supports *non-strict* functions, in which the way of evaluating the arguments can differ from function to function.

Table 5. Implementation of the attribution operations to allow a data-driven evaluation style (only those implementations differing from Table 3 are presented)

Operation	Implementation	Operation	

mkDep (a_0, a_1)	**add** (a_0.deps, a_1)
	a_1.refcount := a_1.refcount + 1
	if ¬ a_1.available **then**
	  add (a_1.obs, a_0)
	  a_0.required := a_0.required + 1
	  a_0.refcount := a_0.refcount + 1
	end if
		set(a,*val*)	a.value := *val*
			a.available := **true**
			foreach a' **in** a.obs **do**
			  a'.required := a'.required – 1
			  if a'.required = 0 **then**
			  val(a')
			  end if
			end foreach
			release(a.obs)

inst(a,f)	a.semFun := f
	a.instrumented := **true**
	if a.required = 0 **then**
	  val(a)
	end if
		val(a)	**if** ¬ a.available ∧
			  a.instrumented **then**
			  set(a, exec(a.semFun,a.deps))
			  a.available := **true**
			  release(a.deps)
			end if
			return a.value

4. A Coding Pattern for Bottom-up Parser Generation Tools

In this section we introduce a coding pattern for bottom-up parser generation tools. In this way:

- In order to keep the translation scheme as independent as possible of changes in the attribute grammar's semantic part, we will promote an intermediary representation of the attribute grammar based on *attribution functions* (subsection 4.1). For this purpose, with each rule will be assigned a function that takes the semantic contexts of the rule's RHS as arguments and builds and returns the semantic context for the rule's LHS. In addition,

using the basic attribution operations introduced in the previous section, attribution functions establish dependencies among attributes, associate semantic functions with attributes as necessary, and control garbage collection.
- Then, these functions will be used in the actions of the resulting bottom-up translation scheme (subsection 4.2). More precisely, the semantic action associated with each rule will invoke the attribution function for this rule with the suitable set of arguments.
- The analysis of the memory footprint required by the overall method will be depicted in subsection 4.3 by considering both the demand-driven and the data-driven evaluation styles.
- Finally, subsection 4.4 briefly illustrates some potential refinements of the initial implementation. These refinements will be oriented to anticipate the computation of inherited attributes by using *marker non-terminals* (i.e., new non-terminals defined by rules with empty RHS), and to simplify implementation by means of global variables.

4.1. The attribution functions

The implementation of the attribute grammar using a bottom-up parser generation tool can be naturally thought of as the *bottom-up* construction of the attribute dependency graph for each processed sentence using basic attribution operations. In this construction, the dependency graph for a syntactic structure is built by taking the dependency graphs of the substructures as building components. Thus, the process can be facilitated by introducing a set of *attribution functions*, which, for each rule in the grammar, take cares of this construction. These attribution functions will be used to set up the semantic actions of the bottom-up translation scheme that feeds the parser generation tool. Therefore, the set of attribution functions can be conceived of as the implementation of a sort of *abstract* version of the attribute grammar, which subsequently can be attached to a concrete syntax by using a suitable translation scheme.

Each attribution function takes the semantic contexts of the symbols in the rule's RHS as input, and it outputs the semantic context for the LHS non-terminal using basic attribution operations. In order to do so, it is possible to apply the following guidelines:
- First at all, we need to create the semantic context for the LHS. This is done by using an `mkCtx` operation. We only need to indicate the number of semantic attributes for the LHS non-terminal.
- Next, we need to describe the dependencies among the attribute instances. Such dependencies are directly determined by examining the semantic equations, and they must be stated by using the `mkDep` operation.
- Once this has been done, it is necessary to *instrument* the synthesized attribute instances in the rule's LHS, as well as the inherited attribute

instances of the RHS symbols. Once more, the code is straightforward: an `inst` operation for each equation. Notice we need to code the semantic functions with integer identifiers, which can be interpreted by the semantic function manager.

- Finally, we need to release the attribute instance lists for the symbols in the rule's RHS.

This process can be further facilitated by using a procedure establishing the corresponding dependencies for each attribute as well as the instrumentation. This procedure, which will be called `eq` (since it actually serves to represent semantic equations), is sketched in Figure 6. Finally, notice that, although we need to provide an attribution function for each rule in the grammar, the same function can be shared by several rules. Therefore, in addition to contributing to more readable translation schemes, attribution functions also make it possible to reuse common attribution patterns. Indeed, it is possible to provide attribution functions with additional parameters in order to increase the reuse degree.

```
procedure eq(lhsAtr,rhsAtrs,semFun) {
   foreach rhsAtr in rhsAtrs
     mkDep(lhsAtr,rhsAtr)
   end foreach
   inst(lhsAtr,semFun)
}
```

Figure 6. The `eq` procedure

As an example, Figure 7 depicts the attribution functions for the attribute grammar in Figure 1. For instance, the `addition` function codes the attribution for the rule `Exp ::= Exp + Opnd` in the grammar of Figure 1 as follows:

- Since `Exp`, the rule's LHS, has two semantic attributes (`env` and `val`), we need to invoke `mkCtx` with 2 as the number of attributes to be allocated.
- From the first equation, we get Exp_1`.env` depends on Exp_0`.env`. In addition, the semantic function to be applied is the identity. Therefore, the equation is coded by `eq(Exp1[env], (Exp0[env]),IDEN).`
- The other equations are coded in a similar manner. For instance, the equation Exp_0`.val = `Exp_1`.val + Opnd.val` is coded by `eq(Exp0[val], (Exp1[val],Opnd[val]),ADD).` Notice that, for each equation, it is important to establish the dependencies in the order in which the attribute references appear in its RHS, and therefore it must be taken into account in the coding of each equation.
- Finally, we include a `release` action for each symbol in the rule's RHS having semantic attributes.

Concerning the allocation of lexical attribute instances, it must be performed by the scanner, which will return the corresponding attribute instance list using a suitable field in the token.

```
def env=0; def val=1; def vs=0; def lex=0;
function init(Exp,Decs) {
   Sent := mkCtx(1)
   eq(Sent[vs], (Exp[val]),IDEN)
   eq(Exp[env],(Decs[env]),IDEN)
   release(Exp)
   release(Decs)
   return Sent
 }
function addition(Exp1,Opnd){
   Exp0 := mkCtx(2)
   eq(Exp1[env], (Exp0[env]),IDEN)
   eq(Opnd[env], (Exp0[env]),IDEN)
   eq(Exp0[val],
        (Exp1[val],Opnd[val]),ADD)
   release(Exp1)
   release(Opnd)
   return Exp0
 }
 function chain(Child) {
  Parent := mkCtx(2)
  eq(Child[env],(Parent[env]),IDEN)
  eq(Parent[val],(Child[val]),IDEN)
  release(Child)
  return Parent
}
function num(num) {
   Opnd := mkCtx(2)
   eq(Opnd[val], (num[lex]),TONUM)
   release(num)
   return Opnd
 }
function var(var) {
   Opnd := mkCtx(2)
   eq(Opnd[val],
       (var[lex],Opnd[env]),VALOF)
   release(var)
   return Opnd
 }
function mutiEnv(Dec,Decs1) {
  Decs0 = mkCtx(1)
  eq(Decs0[env],
     (Dec[env],Decs1[env]),EXTEND)
  release(Dec)
  release(Decs1)
  return Decs0
}
function singleEnv(Dec) {
  Decs = mkCtx(1)
  eq(Decs[env],(Dec[env]),IDEN)
  release(Dec)
  return Decs
}
function entry(var,num) {
  Dec = mkCtx(1)
  eq(Dec[env],
      (var[lex],num[lex]),SINGLEENV)
  release(var)
  release(num)
  return Dec
}
```

Figure 7. Attribution functions for the attribute grammar in Figure 1

4.2. The bottom-up translation scheme

In order to finish the coding, it is necessary to provide a suitable translation scheme. It can be done in a straightforward way, by using the attribution function that corresponds to each rule. Indeed, for each syntax rule A ::= α in the grammar, we only need to add a rule A::= α {\$\$:= ϕ($\$_\alpha$)} to the translation scheme. Here, ϕ is the attribution function for A::= α, and $\$_\alpha$ denotes the list of RHS semantic contexts. This pattern makes further advantages to using attribution functions apparent, instead of directly coding the semantic equations in the rule's actions (like we did in our previous work [41]): the concrete syntax can be readily changed without changing the attribution functions (which, as indicated before, are actually the implementation of an abstract version of the original attribute grammar).

Figure 8 exemplifies the coding pattern by showing the bottom-up translation scheme that implements the attribute grammar of Figure 1. Coded in the input language of a tool like YACC, Bison or CUP, and with a suitable implementation of the attribution functions and the basic attribution operations, it can be automatically turned onto a running implementation.

```
Sent ::= Exp where Decs      {$$ := init($1,$3)}
Exp ::= Exp + Opnd           {$$ := addition($1,$3)}
Exp ::= Opnd                 {$$ := chain($1)}
Opnd ::= num                 {$$ := num($1) }
Opnd ::= var                 {$$ := var($1) }
Opnd ::= (Exp)               {$$ := chain($2) }
Decs ::= Dec, Decs           {$$ := multiEnv($1,$3)}
Decs ::= Dec                 {$$ := singleEnv($1)}
Dec  ::= var = num           {$$ := entry($1,$3)}
```

Figure 8. Bottom-up translation scheme for the attribute grammar in Figure 1

4.3. Analysis of the method

The efficiency of the language processor generated will be manifested in the memory footprint of the recognition and evaluation process, which will in turn depend on the evaluation strategy used and on the kind of the initial attribute grammar:

- If the implementation uses the demand-driven evaluation style, it will incur in the highest amount of auxiliary memory required by the method. Indeed, the memory usage will be rather independent of the kind of the grammar, and proportional to the length of the input sentences. Indeed, the dependency graph will be almost entirely built before evaluation is initiated, and the process will be divided into two well differentiated phases: (i) a first one in which the input sentence is recognized and the dependency graph is built, and (ii) a second one in which the attribute values are computed.
- If the implementation uses the data-driven evaluation style, the performance will be optimal for s-attributed grammars. Indeed, the values of the attributes will be computed as soon as they are instrumented, and the amount of additional memory required for semantic evaluation will remain constant. However, in the presence of inherited information, the evaluation will be delayed until this information is *injected* into the process. The worst case happens when the overall evaluation process depends on inherited information to be set up in the grammar's initial symbol. In this case, most of the dependency graph must be built before initiating evaluation, and thus the method becomes equivalent to using a demand-driven strategy.

This analysis does not mean the method does not provide good (even nearly optimal) solutions for non s-attributed attribute grammars, since inheritance is not required to be global. For instance, for grammars like that of the example, the method, in combination with a data-driven evaluation style, yields not only nearly optimal, but also elegant implementations.

4.4. Refinements

Once the initial coding is available, the initial implementation can be systematically refined in an efficient implementation by using well-known techniques for dealing with inherited information during bottom-up parsing. In particular:

(a)

```
Sent ::= Mo Exp where Decs  {$$ := init($1,$2,$4) }
Mo ::=                      {$$ := mkEnv() }
Exp ::= Exp + Opnd          {$$ := addition($1,$3)}
Exp ::= Opnd                {$$ := chain($1)}
Opnd ::= num                {$$ := num($1) }
Opnd ::= var                {$$ := var($1,$0) }
Opnd ::= (M1 Exp)           {$$ := chain($3) }
M1 ::=                      {$$ = $-1}
Decs ::= Dec, Decs          {$$ := multiEnv($1,$3)}
Decs ::= Dec                {$$ := singleEnv($1)}
Dec ::= var = num           {$$ := entry($1,$3)}
```

(b)

```
...
function mkEnv() {
   return mkCtx(1)
}
...
function init(ExpEnv,Exp,Decs) {
   Sent := mkCtx(1)
   eq(Sent[vs], (Exp[val]),IDEN)
   eq(Exp ExpEnv[env],(Decs[env]),IDEN)
   release(ExpEnv)
   release(Exp)
   release(Decs)
   return Sent
}
...
```

```
function addition(Exp1,Opnd){
   Exp0 := mkCtx(2 1)
   eq(Exp1[env], (Exp0[env]),IDEN)
   eq(Opnd[env], (Exp1[env]),IDEN)
   eq(Exp0[val],
        (Exp1[val],Opnd[val]),ADD)
   release(Exp1)
   release(Opnd)
   return Exp0
}
...
function var(var, Env) {
   Opnd := mkCtx(2 1)
   eq(Opnd[val],
       (var[lex],Opnd Env[env]),VALOF)
   release(var)
   return Opnd
}
```

Figure 9. (a) Refinement of the translation scheme in Figure 8 by means of marker non-terminals; (b) modification of some attribution functions and the addition of a new one (erased code appears in strikethrough light-gray text, and new added coded appears shaded)

- Use of *marker non-terminals* (i.e., new non-terminal symbols defined by empty rules [2]) to mark the beginning of *left spines* (i.e., chains of elements generated by left-recursion). These non-terminals can store inherited attributes to which can be accessed from any point of the left spines without requiring explicit propagation. Using this technique, it is possible to deal with many l-attributed grammars with bounded memory footprint. The technique can be applied to the implementation exemplified before, yielding the translation scheme of Figure 9a. In this refinement it is possible to eliminate the inherited environment, since it can be remotely stored in the marker symbol **`Mo`** and referred from the marker symbol **`M1`**. In addition, the marker contexts can be passed on as an additional argument to the `var` attribution function. In Figure 9b we show the new attribution function `mkEnv` and how the old attribution functions `init`,

`addition` and `var` must be modified to fit in the new refinement. The other attribution functions can be modified in an analogous way, and therefore they will be omitted here.

- Use of global state. In order to integrate this global state in the evaluation machinery, it is possible to create *views* of this state as semantic attributes. The technique can be illustrated with the example discussed above, since the environment can be completely managed as a global variable. Thus, all the machinery concerning propagation of environments can be completely eliminated. Figure 10a shows the resulting translation scheme. Notice how the environment is managed as a global variable, and is also exposed as a globally accessible semantic attribute. With the exception of `init` (see Figure 10b), the attribution functions coincide with those used in the refinement sketched in Figure 9

(a)
```
global env = ∅
global aenv = mkCtx(1)
procedure addEntry(env,Var,Num) {
  env := extendWith({(val(var[lex]),
                      toNum(val(Num[lex])))},env)
}

Sent ::= Exp where Decs  {set(aenv[env],env); $$ := init($1); release(aenv); }
Exp ::= Exp + Opnd       {$$ := addition($1,$3)}
Exp ::= Opnd             {$$ := chain($1)}
Opnd ::= num             {$$ := num($1) }
Opnd ::= var             {$$ := var($1,aenv) }
Opnd ::= (Exp)           {$$ := chain($2) }
Decs ::= Dec, Decs       {}
Decs ::= Dec             {}
Dec  ::= var = num       {addEntry(env,$1,$3)}
```

(b)
```
function init(Exp) {
   Sent := mkCtx(1)
   eq(Sent[vs], (Exp[val]),IDEN)
   release(Exp)
   return Sent
}
```

Figure 10. (a) Use of a global environment to simplify the translation scheme in Figure 8; (b) the `init` attribution function in this refinement.

5. A Coding Pattern for Top-Down Parser Generation Tools

This section describes the coding pattern for top-down parser generation tools. For this purpose, it follows a similar structure to that of the previous one:

- Subsection 5.1 describes the structure of attribution functions in this pattern. In one sense, these attribution functions arose by *reversing* the bottom-up ones. Now, each attribution function takes the semantic context

of the LHS as argument, and it builds and returns the semantic contexts for each symbol in the RHS. As in the bottom-up cases, they also use the basic attribution operations to set up all the attribute evaluation machinery.
- Subsection 5.2 describes the general guidelines to code the translation scheme. As in the bottom-up case, it is carried out by placing attribution functions at strategic points in the syntax rules.
- Subsection 5.3 describes how to deal with underlying non-LL grammars. Indeed, bottom-up parser generation tools usually deal with predictive grammars of the LL-type, in which it is possible to determine which rule to expand by using a finite amount of input look-ahead. However, some grammatical features (e.g., left-recursion, common left-factors) destroy this capability to predict the rule to be applied. Fortunately, many of these grammars can be systematically transformed to forms suitable for top-down parsing. These transformations must be accompanied by the transformation of the semantic part, however. Thus, we researched how to perform these transformations for the case of our encoding scheme.
- As in the bottom-up case, subsection 5.4 briefly analyzes the method, and subsection 5.5 describes some subsequent refinements (the most prominent one deals with the systematic replacement of recursion by iteration in the resulting translation schemes).

5.1. The attribution functions

Although it is possible to undertake implementation by thinking of the bottom-up construction of the attribute dependency graph, as in the bottom-up case, it is possible to obtain more advantages if we think of the *top-down* construction of this graph. In particular, it will facilitate the propagation of inherited information during parsing.

```
function addition(Exp0){
   Exp1 := mkCtx(2)
   Opnd := mkCtx(2)
   eq(Exp1[env], (Exp0[env]),IDEN)
   eq(Opnd[env], (Exp0[env]),IDEN)
   eq(Exp0[val],
        (Exp1[val],Opnd[val]),ADD)
   release(Exp0)
   return (Exp1,Opnd)
 }
```

Figure 11. Top-down geared version of the attribution function `addition`

To enable the top-down construction of the dependency graph, we need to reverse the flow of semantic contexts in the attribution functions. Now, these functions will take the LHS context as input and it will return the RHS contexts as output. Thus, a typical attribution function begins by creating the RHSs contexts. Then it establishes the dependencies between attributes and instruments the attributes as in the bottom-up case. Finally, it releases the LHS context. Figure 11 exemplifies it by showing the top-down geared

version of the `addition` attribution function. The other attribution functions can be adapted in a similar way, and therefore they will be not detailed here.

5.2. The top-down translation scheme

As in the bottom-up case, the coding of the translation scheme is carried out in terms of the attribution functions. In addition, due to the inversion of the flow of semantic contexts in the attribution functions, it is necessary to connect the terminal contexts created in these functions to the contexts created by the scanner. This can be done by using the `conn` procedure sketched in Figure 12 (the name is an abbreviation for *connect*).

```
procedure conn(termCtx,lexCtx) {
   eq(termCtx[lex],(lexCtx[lex]),IDEN)
   release(termCtx); release(lexCtx)
}
```

Figure 12. Procedure for connecting terminal contexts.

Thus, for each syntax rule $A::=X_0 .. X_n$ in the grammar, we need to add a rule $A(\downarrow ctxA)::=\{(ctx_0,\ldots,ctx_n):=\phi(ctxA)\ \}\ I_0\ \ldots\ I_n$ where: (i) ϕ is the rule's attribution function, (ii) $(ctx_0,\ldots,ctx_n)$ collects the RHS contexts (this assignment is optional; it can be omitted if the attribution function does not return any context), and (iii) each I_i is $X_i(ctx_i)$ if X_i is a non-terminal, $X_i(lexctx_i)$ {`conn(`ctx_i`,`$lexctx_i$`)`} if it is a terminal with semantic charge, or X_i if it is a terminal without semantic charge (a keyword, a punctuation symbol, etc.). These guidelines are illustrated in Figure 13, which shows the top-down translation scheme for the grammar in Figure 1.

```
Sent(↓co) ::= {(c1,c2) := init(co)} Exp(c1) where Decs(c2)
Exp(↓co)  ::= {(c1,c2) := addition(co)} Exp(c1) + Opnd(c2)
Exp(↓co)  ::= {c1 := chain(co)} Opnd(c1)
Opnd(↓co) ::= {c1 := num(co) } num(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := var(co) } var(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := chain(co) } (Exp(c1))
Decs(↓co)  ::= {(c1,c2) := multiEnv(co)} Dec(c1) , Decs(c2)
Decs(↓co) ::= {c1 := singleEnv(co)} Dec(c1)
Dec(↓co) ::= {(c1,c2) := entry(co)} var(lc1) {conn(c1,lc1)} = num(lc2) {conn(c2,lc2)}
```

Figure 13. Top-down translation scheme for the attribute grammar in Figure 1 (warning: this translation scheme is not yet implementable with a top-down parser generator!)

Unfortunately, since top-down translators usually require LL underlying context-free grammars, translation schemes obtained according to the stated guidelines can require further transformation before allowing their implementation during parsing. In particular, the context-free grammar of the translation scheme in Figure 1 exhibits left-recursion, which make this coding

unsuitable for top-down parser generation. Next subsection deals with this problem.

5.3. Factoring and immediate left-recursion elimination

In many cases the problematic top-down translation schemes and the associated attribution functions can be systematically tuned by applying similar patterns to the well-known factoring and left-recursion elimination transformations presented in any compiler construction textbook [2]. In particular:

- Figure 14a sketches a transformation pattern for removing common factors in a rule-set. Notice this transformation supposes the explicit construction of the common factor's semantic context. It will be carried out by a *context-construction* function (denoted by $\texttt{mkCtx}_{\alpha}$ in Figure 14a). In addition, it is necessary to keep this context alive, regardless whether it will be released in the common factor. For this purpose, we need to create another twin context ($c_{\alpha p}$ in Figure 14a), and to connect it to the actual common factor's semantic context. This connection is achieved with a *context connection procedure*, denoted by $\texttt{conn}_{\alpha}$ in Figure 14a. Finally, it will require explicitly modifying the attribution functions for each rule affected. The modified attribution functions (denoted by ϕ'_i in Figure 14a) do not need to create the semantic context for the common factor; instead, they will take it as a parameter.
- Figure 14b shows a transformation pattern for removing immediate left-recursion. The pattern requires the explicit construction of the context for the recursive non-terminal, which is achieved by using a context-construction function ($\texttt{mkCtx}_{A}$ in Figure 14b). As usual, the chain generated by left-recursion in the original grammar is generated by using right-recursion in the transformed one. Each stage of this right-recursive process can be associated with a stage in the bottom-up construction of the parse tree in the original grammar. Therefore, it is possible to take the context associated to the root of the already constructed sub-tree as input, and then to modify the corresponding attribution function to take this as an additional argument instead of creating it (the modified functions are noted ϕ'_i in Figure 14b, and they must take care of releasing the semantic context once they are not necessary). In addition, it is necessary to provide a context connection procedure for performing the connection between the input and the last context created once the right-recursion is finished (it is denoted by $\texttt{conn}_{A}$ in Figure 14b).

Figure 15 illustrates the application of these patterns to the translation scheme of Figure 13. The grammar of the transformed scheme is LL(1) and, therefore, suitable for its implementation in any of the top-down parser generation tools mentioned.

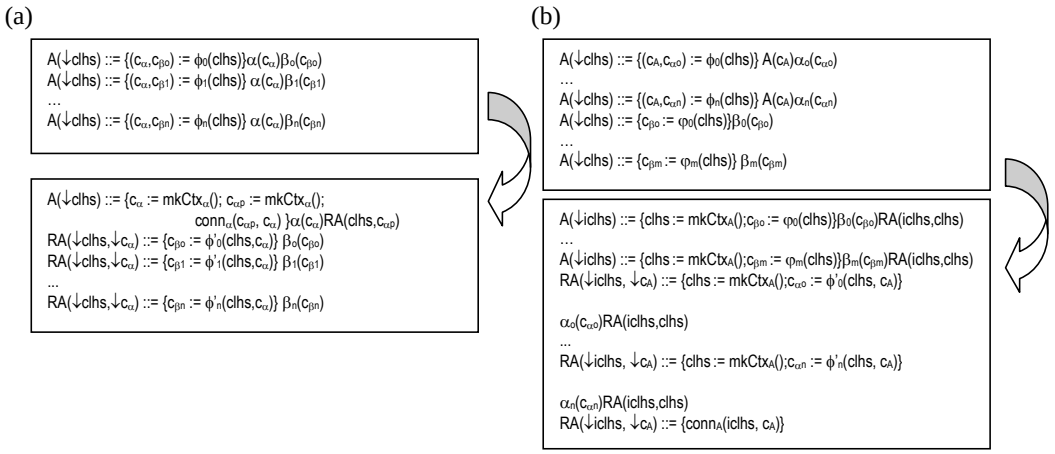

Figure 14. (a) Factoring pattern; (b) Immediate left-recursion elimination pattern

```
function mkCtxExp() {return mkCtx(2)}
function mkCtxDec() {return mkCtx(1)}
procedure connExp(ic,c) {eq(c[env],(ic[env),IDEN); eq(ic[val],(c[val]),IDEN) }
procedure connDecs(cp,c) {eq(cp[env],(c[env),IDEN);}

Sent(↓co) ::= {(c1,c2) := init(co)} Exp(c1) where Decs(c2)
Exp(↓ic)  ::= {co := mkCtxExp(); c1 := opnd(co)} Opnd(c1) RExp(ic,co)
RExp(↓ic, ↓c1) ::= {co := mkCtxExp(); c2 := addition(co,c1)} + Opnd(c2) RExp(ic,co)
RExp(↓ic, ↓co) ::= {connExp(ic,co)}
Opnd(↓co) ::= {c1 := num(co) } num(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := var(co) } var(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := chain(co) } (Exp(c1))
Decs(↓co)  ::= {c1 := mkCtxDec(); c1p := mkCtxDec(); connDecs(c1p,c1) }
                                                    Dec(c1) RDecs(co,c1p)
RDecs(↓co, ↓c1) ::= {c2 := multiEnv(co,c1)} , Decs(c2)
RDecs(↓co, ↓c1) ::= {singleEnv(co,c1)}
Dec(↑co) ::= {(c1,c2) := entry(co)} var(lc1) {conn(c1,lc1)} = num(lc2) {conn(c2,lc2)}
```

Figure 15. Result of eliminating common factors and immediate left-recursion in the top-down translation scheme of Figure 13 (the transformed parts are shadowed) in order to obtain an artifact implementable with a top-down parser generator.

5.4. Analysis of the method

As in the bottom-up case, the use of a demand-driven evaluation style will imply explicitly constructing dependency graphs, and therefore the highest memory overhead. As in bottom-up implementations, it can be alleviated by using data-driven evaluation. In this case, the method will incur in the lowest auxiliary evaluation memory overhead for l-attributed grammars. Indeed, for these grammars, data-driven evaluation will yield a behavior equivalent to a one-pass, on-the-fly translation process.

Finally, since the initial coding encourages the explicit coding of the plain, BNF grammar, the resulting translators will be highly recursive, which should be taken into account if the final implementation language does not support tail recursion optimization. Fortunately, as will be indicated in the next section, by using EBNF notation in the underlying context-free grammars, it will be possible to easily turn many right-recursions into iteration.

5.5. Refinements

As in the bottom-up case, it is possible to use global state to simplify the propagation of context. Nevertheless, due to the nature of top-down translators, this refinement is less critical from a performance perspective. Concerning the use of marker non-terminals, it is nonsense in this scenario.

However, as indicated in the previous subsection, an interesting refinement would be to exploit the support of EBNF notation provided by typical predictive recursive parser generation tools in order to overcome the potential stack overflow problem associated with the recursive implementation of genuinely iterative processes[5]. Indeed, it is equivalent to performing a tail-recursion optimization process by hand[6].

In addition, it is possible to carry out several simplifications oriented to minimizing the use of temporary variables (e.g., by passing complex expressions as parameters to non-terminal symbols).

```
Sent(↓co) ::= {(c1,c2) := init(co)} Exp(c1) where Decs(c2)
Exp(↓ico)  ::= {co := mkCtxExp()} Opnd(chain(co)) RExp(ico,co)
RExp(↓ic, ↓c1) ::= ({co := mkCtxExp()} + Opnd(addition(co,c1)) {c1:=co})*
                                                        {connExp(ic,c1)}
Opnd(↓co) ::= num(lc1) {conn(num(co),lc1)}
Opnd(↓co) ::= var(lc1) {conn(var(co),lc1)}
Opnd(↓co) ::= ( Exp(chain(co)) )
Decs(↓co)  ::= {c1 := mkCtxDec(); c1p := mkCtxDec(); connDecs(c1p,c1) }
                                                    Dec(c1) RDecs(co,c1p)
RDecs(↓co, ↓c1) ::= ({co := multiEnv(co,c1)} ,
                     {c2 := mkCtxDec(); c1 := mkCtxDec(); connDecs(c1,c2) }
                      Dec(c2))* {singleEnv(co,c1)}
Dec(↑co) ::= {(c1,c2) := entry(co)} var(lc1) {conn(c1,lc1)} = num(lc2) {conn(c2,lc2)}
```

Figure 16. Refinement of the translation scheme in Figure 15

Figure 16 exemplifies the result of applying these refinements on the translation scheme of Figure 15. The resulting scheme can be readily implemented on any typical recursive predictive parser generation tool (e.g., JavaCC or ANTLR), or directly by hand in a general-purpose programming language. As this example makes apparent, after applying this refinement,

[5] Notice this problem does not affect bottom-up parsers, provided sequences are represented by means of left-recursion.

[6] Indeed, it could be possible to directly formulate the immediate left-recursion elimination pattern in iterative terms.

recursion will only be used to express nesting (in the example, it is due to the use of parenthesis in expressions), which constitutes the most natural use of this grammar feature.

6. Related Work

As indicated in the introduction, the standard way of implementing an attribute grammar is to use one of the tools that directly supports the formalism. Indeed, as [35] makes apparent, since its invention by Knuth at the end of the sixties of the past century, the computer language community has proposed many of these tools, starting with classical systems like GAG [22], FNC-2 [20], ELI [15] or Elegant [7], and ending with recent proposals like LISA [17][31][33], Silver [51] or JastAdd [29]. These tools take attribute grammars as input, and generate operative language processors as output. In addition, they support metalanguages by adding many extensions to the basic formalism (e.g., modules [21], generics [42], higher-order [48], object [16] and aspect orientation [39][40], etc.), which facilitate the production and maintenance of complex specifications.

Attribute grammar-based systems as the abovementioned promote orchestrating the development entirely in terms of attribute grammars, and, in particular, in terms of the metalanguages supported. On the contrary, the goal of our approach is not to provide yet another attribute grammar system, but to propose systematic ways of integrating attribute grammars in conventional language implementation processes, by using conventional parser generation tools. In this way, in our approach attribute grammars are used at the initial stages of the development process, as a formal specification tool. In addition, our work promotes an initial design-preserving coding in a conventional parser generation tool, in the form of a suitable translation scheme. Beyond this point, the development process proceeds through several refinements, making use of the parser generation tool facilities and the tool's target implementation language.

In consequence, our approach promotes straightforward coding patterns, which can be applied by hand to get initial codings, and which make it possible to identify the different pieces of the original attribute grammar in these codings. On the other hand, the code generated by an attribute grammar-based tool is usually a highly optimized artifact, usually generated following a *static* approach in which evaluation and storage strategies are determined as the result of a static analysis of the input grammar [1], and which is not intended to be inspected and modified by humans.

In addition, our approach is oriented to converge with conventional development processes. Because of it, on one hand we encourage the use of semantic evaluation methods that can be easily coupled with parsing. This is not necessarily true for attribute grammar-based tools, many of which promote final implementations that operate on (concrete or abstract) syntax trees. Of course the patterns described in this paper could be automated in

the form of attribute-grammar based tools. Indeed, tools for the processing of XML based on attribute grammars like those described in [43] are inspired by these patterns (in particular, these tools use the data-driven evaluation strategy to make the stream-oriented, asynchronous, processing of very wide XML documents possible). These tools could be used as a sort of CASE support during the development process model promoted in this paper, which in turn could imply the provision of some roundtrip support (see the future work description in the next section).

The coupling of attribute evaluation and parsing has been extensively addressed as a way of implementing restricted classes of attribute grammars (see, for instance, [3] for a tutorial introduction). The works in [2][3] show how l-attributed grammars with underlying LL grammars can be implemented during top-down predictive descent parsing. In addition, different classes of LR-attributed grammars have been identified, which allow semantic evaluation to be implemented using straightforward extensions of LR parsers [4]. In the marriage of attribute grammars and logic programming, the class of *logical one-pass logical* attribute grammars shows how some kinds of right dependencies can also be managed during conventional top-down parsing [34][36]. Contrary to the work presented in this paper, all these approaches constrain the classes of allowed grammars to strict subclasses of non-circular attribute grammars. In contrast, our approach is able to deal with arbitrary non-circular attribute grammars. If the grammars are of certain types (e.g., l-attributed grammars with an LL(1) underlying context-free grammar), and a suitable semantic evaluation approach is used (e.g., a data-driven strategy), our implementations produce artifacts comparable in performance and memory footprint to those promoted by the abovementioned works. In other cases, the approach is still able to produce running implementations, which can adapt the memory footprint to that required for performing semantic evaluation.

The development of some attribute grammar-based systems has exploited the marriage between attribute grammars and parser generation tools. A common strategy is to build a preprocessor by translating an attribute grammar-based specification language into a running implementation written in terms of a parser generator. In [23] one of these systems is described, which takes an attribute grammar-like specification as input, and it turns it into a YACC implementation. However, since the resulting implementation evaluates attributes during parsing, the class of supported grammars is restricted to a subset of the LR-attributed ones. The Ox system [8] follows a similar approach, but it supports arbitrary non-circular attribute grammars. For this purpose, the processors generated decouple parsing and semantic evaluation by using an optimized implementation of the processing models behind attribute grammars (i.e., to build the parse tree, to arrange attribute instances in topological order, and then to perform evaluation according to this order). XLOP [43], a system developed by us to describe XML processing tasks as attribute grammars, also translates attribute grammar specifications into inputs to a parser generation tool (in this case, CUP). RIE [44], a system that supports a very general class of LR-attributed grammars (ECLR-

attributed grammars [4]) adopts a different implementation approach, by basing the metagenerator on an explicit modification of the Bison parser generation tool. Regardless of the implementation strategy followed (in these examples, based on preprocessors for / extensions to parser generation tools), they ultimately fall in the category of attribute grammar-based tools. Therefore, the general considerations made above concerning the relationships between our approach and attribute grammar – based tools also applies here.

Concerning parser generators, there is a plethora of systems available that can be used during the development of a language processor. A basic feature differentiating them is whether they generate top-down parsers (e.g., the aforementioned tools JavaCC [26] and ANTLR [38], as well as classic tools like COCO/R [32]), or bottom-up ones (e.g., the aformentioned YACC [45],Bison [27] and CUP [5], as well as tools like Tatoo [11], SableCC [13], Beaver[7], Copper [49] or YaJco[8]). Also, these tools differ in the class of grammars allowed (e.g., JavaCC supports LL(k) grammars, while ANTLR supports the aforementioned LL(*) parsing method, able to deal with unbounded look-ahead; additionally tools like Elkhound [30], SDF [10] or, under certain settings, Bison, provide support to arbitrary context-free grammars via the GLR parsing method [46]), by the expressiveness of its specification language (e.g., ANTLR or Tatoo support very sophisticated features, like grammar modularization, rule inheritance, etc.), by whether they include support for lexical specification (e.g., JavaCC, ANTLR) or whether it must be made by using a separating tool (e.g., CUP), and by many other features whose detailed analysis is beyond the scope of the present work. As was indicated, the patterns presented in this paper are applicable to most of these parser generators (in particular in those tools that support deterministic grammars; in tools like SDF, whose outcome is parse forests that must be subsequently disambiguated, the applicability of these patterns vanishes). Also, it is important to notice that, while many of these parser generation tools support the concept of *semantic attribute*, like attribute grammars (e.g., this terminology is explicitly included in ANTLR), it does not mean that these tools give direct support for attribute grammars. Indeed, in addition to managing semantic attributes, the essential aspect of attribute grammars is the support for a dependency-driven execution style: semantic evaluation is not necessarily coupled with parsing, but emerges as a consequence of the dependencies among attributes. In this way, the patterns introduced in this work make it possible to incorporate this computation style into specifications for parser generation tools, and, in consequence, to facilitate the subsequent refinement into more efficient implementations.

Finally, as the implementations of our attribution operations make apparent, we avoid the explicit construction of the parse tree. While this construction is necessary in order to support more sophisticated evaluation

[7] http://beaver.sourceforge.net/

[8] http://code.google.com/p/yajco/

strategies (see, for instance [1]), our simple coding patterns make it unnecessary, since it is centered directly on the construction of dependency graph-like structures. A similar technique is followed in [6], an implementation of circular attribute grammars in Prolog whose semantic equations are described by using λ-expressions. The execution model of the resulting artifact works in two stages: (i) construction of λ-expressions for the root's synthetized attributes, and (ii) interpretation of these expressions according to a least fixpoint semantics to yield the final values. Thus, the resulting approach resembles our demand-driven implementation. In [50], Prolog is also used to implement attribute grammars, and two evaluation strategies are proposed. The first one supposes building terms representing semantic expressions for the root's synthetized attributes, which are subsequently interpreted with a separate interpreter. The second one promotes the use of Prolog co-routine facilities to delay evaluation of arguments until they are instantiated. Thus, the first strategy is analogous to our demand-driven implementation (nevertheless, our implementation is optimized to avoid duplicated evaluations; see [47] for a similar implementation in Prolog that also avoids redundant evaluations). The second one is a Prolog implementation of a data-driven strategy.

7. Conclusions and future work

This paper has shown how to systematically code arbitrary non-circular attribute grammars in the input languages of bottom-up, LALR(1) parser generation tools like YACC, BISON or CUP, as well as top-down, LL parser generation tools like JavaCC or ANTLR. It is done by using a small set of attribution operations. These operations, in turn, can be implemented in different ways in order to enable different semantic evaluation styles. In particular, this paper has illustrated two alternative implementations: one supporting a demand-driven style, and another supporting a data-driven one. The results of this work can be useful to promote a systematic method of using conventional parser generation tools to yield final implementations. This method starts with the initial coding of an attribute grammar-based specification, and then it evolves it in a final implementation by applying systematic implementation patterns and techniques. Thus, by applying and documenting systematic refinements, it is possible, on one hand, to yield efficient implementations and, on the other hand, to track the refinement chain from these final implementations to the original attribute grammar-based specifications. Besides, the method facilitates the incremental introduction of new language features, since they can be described according to attribute grammar conventions, then readily coded in the implementation, and finally optimized according to implementation-dependent criteria. Therefore, the method transports the attribute grammar amenability to doing modular and extensible specifications incrementally to an implementation process based on parser generation tools.

Currently we have successfully tested our method with several small examples, and we are applying it to the development of a non-trivial translator for a Pascal-like language. From these experiences, we have realized how the encoding patterns are simple enough to being applied without specific tooling support (although, of course, this support could be a very valuable facility in our methodology). Also, we have gained further evidence on the feasibility and usefulness of our method with its application in an introductory compiler construction course during the first period of the 2011-2012 academic year at the Complutense University. Indeed, we proposed that our students produce initial implementations of language processors by taking attribute grammar specifications as a guide, and using the method described in this paper. We observed that they didn't find it more difficult to apply than students of previous courses found while hand-coding conventional recursive descent translators. In addition, the quality of the final programs was substantially better than in previous years, since the method encouraged rigorous adherence to the original specification. Thus, we plan to further apply it as a systematic learning method in future editions of the course. Also, as future work, we plan to provide the aforementioned tooling support in order to facilitate the application of the method: automatic application of the coding patterns to produce the initial translation schemes, support for some of the transformations and refinements described in this paper, roundtrip support and support for tracking successive refinements, and support for profiling and debugging the semantic evaluation processes.

Acknowledgements. Thanks are due to project grants TIN2010-21288-C02-01 and Santander-UCM GR 42/10, group reference 962022. Also, Daniel Rodriguez-Cerezo was supported by the Spanish University Teacher Training Program (EDU/3445/2011).

References

1. Ablas, H. Attribute Evaluation Methods. In Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science Vol. 545, Springer, 48-113. (1991)
2. Aho A.V, Lam M.S, Sethi R, Ullman J.D.: Compilers: principles, techniques and tools (2nd Edition). Addison-Wesley. (2006)
3. Akker, R., Melichar, B., Tarhio, J. Attribute Evaluation and Parsing. In Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 187-214. (1991)
4. Akker, R., Melichar, B., Tarhio, J.: The Hierarchy of LR-attributed grammars. In Deransart, P., Jourdan, M (eds.): Attribute Grammars and their Applications – Proceedings of the International Workshop on Attribute Grammars and their Applications (WAGA'90), Paris, France, Lecture Notes in Computer Science 461, Springer, 13-28. (1990)

5. Appel, A.W. Modern Compiler Implementation in Java. Cambridge University Press. (2002)
6. Arbab, B. Compiling Circular Attribute Grammars into Prolog. IBM Journal of Research and Development, Vol. 30, No. 3, 294-309. 1986
7. Augusteijn, L. The Elegant Compiler Generator System. In Deransart, P., Jourdan, M (eds.): Attribute Grammars and their Applications – Proceedings of the International Workshop on Attribute Grammars and their Applications (WAGA'90), Paris, France, Lecture Notes in Computer Science 461, Springer, 238-254. (1990)
8. Bischoff, K.M. Design, Implementation, Use and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex and C. TR #92-31, Dp. Of Computer Science, Iowa State University, (1992)
9. Bochmann, G.V.: Semantic Evaluation from Left to Right. Communications of the ACM, Vol. 19, No. 2, 55-62. (1976)
10. Brand, M.G.J v.d., Deursen, A, v., Heering, J., Jong, H.A.d., Jonge, M.d., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, JJ., Visser, E., Visser, J. The Asf +Sdf Meta-environment: A Component-Based Language Development Environment. In Wilhelm, R (ed.): Compiler Construction - Proceedings of the 10th International Conference on Compiler Construction CC'01, Genova, Italy, Lecture Notes in Computer Science, 2027, Springer, 365-370. (2001)
11. Cervelle, J., Forax, R., Roussel, G. Tatoo: an innovative parser generator. 4th International Symposium on Principles and Practice of Programming in Java PPPJ'06, Mannheim, Germany, ACM, 13-20. (2006)
12. Ekman, T., Hedin, G. The JastAdd system - modular extensible compiler construction. Science of Computer Programming, Vol. 69, No. 1-3, 14-26. (2007)
13. Gagnon, E.M., Hendren, L.J. SableCC, an Object-Oriented Compiler Framework. International Conference on Technology of Object-Oriented Languages TOOLS'98, Sta Barbara, CA, USA, IEEE, 140-154. (1998)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. (1995)
15. Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.M., Waite, W.M.: Eli: A Complete, Flexible Compiler Construction System. Communications of the ACM, Vol. 35, 121-131. (1992)
16. Hedin, G. An Object-Oriented Notation for Attribute Grammars. 3rd European Conference on Object-Oriented Programming, Nottingham, UK, Cambridge University Press, 329-345. (1989)
17. Henriques, P.R., Varanda-Pereira, M.J., Mernik, M., Lenic, M., Gray, J.G., Wu, H. Automatic Generation of Language-Based Tools using the LISA System. IEE Proceedings – Software, Vol. 152, No. 2, 54-69. (2005)
18. Jalili, F.: A general linear-time evaluator for attribute grammars. ACM SIGPLAN Notices, Vol. 18, No. 9, 35-44. (1983)
19. Jones, L.G.: Efficient Evaluation of Circular Attribute Grammars. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, 429-462. (1990)
20. Jourdan, M., Parigot, D.: Internals and Externals of the FNC-2 Attribute Grammar System. In Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 485-504. (1991)
21. Kastens, U., Waite, W.M.: Modularity and Reusability in Attribute Grammars. Acta Informatica, Vol. 31, No. 7, 601-627. (1994)

22. Kastens, U.: GAG: A Practical Compiler Generator. Lecture Notes in Computer Science 141, Springer. (1982)
23. Katwijk, J.: A preprocessor for YACC or a poor man's approach to parsing attributed grammar. ACM SIGPLAN Notices, Vol. 18, No. 10, 12-15. (1983)
24. Kennedy, K., Ramanathan, J.: A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing. ACM Transaction of Programming Languages and Systems, Vol. 1, No. 1, 142-160. (1979)
25. Knuth, D. E.: Semantics of Context-free Languages. Mathematical System Theory, Vol. 2, No. 2, 127–145. (1968). See also the correction published in Mathematical System Theory, Vol. 5, No. 1, 95–96.
26. Kodaganallur, V. Incorporating language processing into Java applications: a JavaCC tutorial. IEEE Software, Vol. 21, No. 4, 70-77. (2004)
27. Levine, J. Flex & Bison: Text Processing Tools. O'Reilly Media. (2009)
28. Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E.: Attributed Translations. Journal of Computer and System Sciences, Vol. 9, No. 3, 279-307. (1974)
29. Magnusson, E. Hedin, G.: Circular Reference Attributed Grammars—Their Evaluation and Applications. Science of Computer Programming, Vol. 68, No. 1, 21-37. (2007)
30. McPeak, S., Necula, G.C. Elkhound: A Fast, Practical GLR Parser Generator. International Conference on Compiler Construction (CC'04), Barcelona, Spain, Lecture Notes in Computer Science, Vol. 2985, 73-88. (2005)
31. Mernik, M., Lenic, M., Acdicausevic, E., Zumer, V.: LISA: An Interactive Environment for Programming Language Development. 11th International Conference on Compiler Construction (CC'02), Grenoble, France, Lecture Notes in Computer Science, Vol. 2304, Springer, 1-4. (2002)
32. Mössenböck, H. A Generator for Production Quality Compilers. 3rd intl. workshop on Compiler Compilers (CC'90), Schwerin, Lecture Notes in Computer Science Vol. 477, 42–55. (1990)
33. Oliveira, N., Varanda-Pereira, M.J., Henriques, P.R., da Cruz, D., Cramer, B.: VisualLISA: A Visual Environment to Develop Attribute Grammars. Computer Science and Information Systems Journal, Vol. 7, No. 2, 266-289. (2010)
34. Paakki, J. Prolog in Practical Compiler Writing. Computer Journal, Vol. 34, No. 1, 64-72. (1991)
35. Paakki, J.: Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computing Surveys, Vol. 27, No. 2, 196-255. (1995)
36. Paakki, J.: PROFIT: A System Integrating Logic Programming and Attribute Grammars. 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP'91), Passau, Germany, Lecture Notes in Computer Science Vol. 528, 243-254. (1991)
37. Parr, T., Fisher, K. LL(*): the Foundation of the ANTLR Parser Generator. ACM SIGPLAN Notices - PLDI '11, Vol. 46, No. 6, 425-436. (2011)
38. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf. (2007)
39. Rebernak, D., Mernik, M., Henriques, P.R., Carneiro, D., Varanda-Pereira, M.J. Specifying Languages Using Aspect-oriented Approach: AspectLISA. Journal of Computing and Information Technology, Vol. 4, 343-350. (2006)
40. Rebernak, D., Mernik, M., Henriques, P.R., Varanda-Pereira, M.J.: AspectLISA: An Aspect-oriented Compiler Construction System Based on Attribute Grammars. Electronics Notes in Theoretical Computer Science – LDTA'06, Vol. 164, 37-53. (2006)

41. Rodriguez-Cerezo, D., Sarasa, A., Sierra, J.L.: Implementing Attribute Grammars Using Conventional Compiler Construction Tools. 3rd Workshop on Advances in Programming Languages (WAPL'11), Szczezin, Poland, IEEE, 855-862. (2011)
42. Saraiva, J., Swiestra, D.: Generic Attribute Grammars. 2nd Workshop on Attribute Grammars and Their Applications (WAGA'99), Amsterdam, The Netherlands. (1999)
43. Sarasa, A., Temprado-Battad, B., Sierra, J.L, Fernández-Valmayor, A.: XML Language-Oriented Processing with XLOP. 5th International Symposium on Web and Mobile Information Services, Bradford, UK, Proceedings of AINA'09 Workshops, IEEE, 322-327. (2009)
44. Sassa, M., Ishizuka, H., Nakata, I. Rie, a compiler generator based on a one-pass-type attribute grammar. Software – Practice & Experience, Vol. 25, No. 3, 229-250, (1995)
45. Schreiner, A.T., Friedman, H.G. Introduction to Compiler Construction with Unix. Prentice-Hall. (1985)
46. Scott, E., Johnstone, A. Right nulled GLR parsers. ACM Transactions on Programming Languages and Systems, Vol. 28, No. 4, 577-618. (2006)
47. Sierra, J.L., Fernández-Valmayor, A. A Prolog Framework for the Rapid Prototyping of Language Processors with Attribute Grammars. Electronics Notes in Theoretical Computer Science – LDTA'06, Vol. 164, 19-36. (2006)
48. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher-Order Attribute Grammars. ACM SIGPLAN Notices Vol. 24, No. 7. (1989)
49. Vyk, E.R.v., Schwerdfeger, A.C. Context-aware scanning for Parsing Extensible Languages. 6th International Conference on Generative Programming and Component Engineering GPCE'06, Portland, Oregon, USA, ACM, 63-72. (2006)
50. Walsteijn, M.J., Kuiper, M.F.: Attribute Grammars in Prolog. Technical Report, RU-CS-86-14, Utrecht University. (1986)
51. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: An Extensible Attribute Grammar System. Science of Computer Programming, Vol. 75, No. 1-2, 39-54. (2010)

Daniel Rodríguez-Cerezo is a PhD student in the Computer Science School at UCM, and a member of the research group ILSA (Implementation of Language-Driven Software and Applications: http://ilsa.fdi.ucm.es). His research is focused on the use of several e-Learning techniques (simulations, interactive prototyping tools, recommendation systems for learning object repositories, etc.) to improve teaching and learning of the Software Language Engineering discipline. Besides, he is interested in the development and improvement of software language engineering techniques.

Antonio Sarasa-Cabezuelo is a full-time Lecturer in the Computer Science School at Complutense University of Madrid, Spain (UCM). His research is focused on the language-oriented development of XML-processing applications, and on the development of applications in the fields of digital humanities and e-Learning. He was one of the developers of the Agrega project on digital repositories (a pioneer project in this field in Spain). He is a member of ILSA. He has participated in several research projects in the fields

of software language engineering, digital humanities and e-learning, and he has published over 50 research papers in national and international conferences.

José-Luis Sierra is an Associate Professor at the UCM's Computer Science School, where he leads the ILSA Research Group. His research is focused on the development and practical uses of computer language description tools and on the language-oriented development of interactive and web applications in the fields of digital humanities and e-Learning. Prof. Sierra has leaded and participated in several research projects in the fields of digital humanities, e-learning and software language engineering, the results of which have been published in over 100 research papers in international journals, conferences and book chapters. He serves regularly as reviewer / PC Member for several international reputed journals and conferences.

Received: December 23, 2011 Accepted: June 1, 2012.