

Implementing Attribute Grammars Using Conventional Compiler Construction Tools

Daniel Rodríguez-Cerezo Antonio Sarasa-Cabezuelo, José-Luis Sierra
Facultad de Informática. Universidad Complutense de Madrid. 28040 Madrid, Spain
Email: drodriguez@fdi.ucm.es, {asarasa, jlsierra}@fdi.ucm.es

Abstract—This article describes a straightforward and structure-preserving coding pattern to encode arbitrary non-circular attribute grammars as syntax-directed translation schemes for bottom-up parser generation tools. According to this pattern, a bottom-up oriented translation scheme is systematically derived from the original attribute grammar. Semantic actions attached to each syntax rule 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 it possible the direct implementation of attribute grammar-based specifications using widely-used translation scheme-driven tools for the development of bottom-up language translators (e.g. YACC, BISON, CUP, etc.). As a consequence, this initial coding can be subsequently refined to yield final efficient implementations. Since these implementations still preserve the ability of being extended with new features described at the attribute grammar level, the advantages from the point of view of development and maintenance become apparent.

I. INTRODUCTION

ATTRIBUTE grammars, which were introduced by Donald E. Knuth [8] as an extension of context-free grammars for describing the syntax and semantics of context-free languages, are widely-used as a high-level specification method for the first stages of the design and implementation of a computer language [1][11].

In order to make an attribute grammar – based specification executable, it is possible to use one of the many specialized tools supporting the formalism (see, for instance, [3] [10][11]). However, regardless the realized 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 parse generators (e.g., ANTLR, CUP, Flex, Bison...), general-purpose programming languages, or a suitable combination of both techniques [1]. The process of transforming the initial specification in a final implementa-

tion is usually ill-defined, and usually depends solely on the programmer's art, who many times discards formal specifications while directly hacks the final implementation. It seriously hinders systematic development and maintenance of language processors.

In order to bridge the gap between attribute grammar-based specifications and final implementations, we propose to articulate 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 encoding the attribute grammar in the input language of the development tool (usually, a parse generator like Bison or CUP). It will make it possible to yield an initial running implementation, which subsequently can 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 largely facilitated.

This paper is 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 parse generation tool. More precisely, we will focus on bottom-up parse generators of the YACC and CUP type. Unlike to works in LR-attributed grammars [2] and similar approaches (e.g., [6]), our approach will support the implementation of arbitrary non-circular attribute grammars. In addition, the encoding pattern will be independent of the final evaluation style chosen. Indeed, attribute grammars will be coded 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 finally used.

The structure of the rest of the paper is as follows: section II describes the encoding pattern itself. Sections III and IV show how to plug in different evaluation styles by providing suitable implementations of the attribution operations. Finally, section V concludes the paper and outlines some lines of future work.

II. ENCODING THE ATTRIBUTE GRAMMARS

In this section we introduce our coding pattern. In order to make it as general as possible, we will not compromise with any particular generation tool, and we will use pseudo-code comprising very simple and standard procedural interfaces and imperative constructs. In addition, we will use a YACC-like notation [1] to refer to semantic values of symbols in the parse stack. In subsection II.A we describe the basic attribution operations allowed in the syntax rules' semantic actions. In sections II.B and II.C we describe the coding pattern itself, and in section II.D we exemplify it.

TABLE I.
ATTRIBUTION OPERATIONS

Operation	Intended Meaning
mkCtx(n)	It creates and initializes a list of n attribute instances for a symbol in the parse stack.
mkDep(a_0, a_1)	It sets a dependency between two attribute instances. Indeed, it declares the attribute instance a_0 depends on the attribute instance a_1 .
inst(a, f)	It <i>instruments</i> 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 list of attribute instances 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 .

A. Attribution operations

Our coding pattern is largely based on the explicit description of the attribution structure of each grammar rule. For this purpose, we introduce the repertory of basic *attribution* operations outlined in Table 1. This table shows both the procedural interfaces of the operations and their intended meanings.

As such a description makes apparent, the purpose of these operations is to provide the developer with the necessary tools to describe how the *attribute dependency graph* associated with a sentence can be built conforming 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 overhead (see sections III and IV).

B. Writing the translation scheme

The actual encoding of the attribute grammar requires writing a translation scheme describing how to build the aforementioned attribute dependency graph for each processed sentence. It can be done in a straightforward way by applying the following guidelines to each rule of the attribute grammar:

- First at all, we need to create the semantic value for the rule left-hand side (LHS). It is done by using an `mkCtx` operation. We only need to indicate the number of semantic attributes for the LHS.
- Next, we need to describe the dependencies between the attribute instances. Such dependencies are directly determined by examining the semantic equations, and they must be stated using the `mkDep` operation.
- Once it 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 symbols in the rule's right-hand side (RHS). Once more, the code is straightforward: an `inst` operation for each equation. Notice we need to encode the semantic functions with integer identifiers, which can be interpreted by a *semantic function manager* (see subsection II.C).
- Finally, we need to release the attribute instance lists for the symbols in the rule's RHS.

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. Also, notice the underlying context-free grammar must belong to the kind of grammars supported by the parser generation tool. Since we are using bottom-up parse generation translators, which usually support LALR(1) grammars [1], in practice it does not suppose a serious limitation.

C. Writing the Semantic Function Manager

In addition to the translation scheme, we need to code another auxiliary component, the *semantic function manager*, supporting the execution of the semantic functions. This component 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.

D. Example

To illustrate the pattern we will consider the attribute grammar in Fig. 1. It models a very simple processor that makes it possible to evaluate simple arithmetic expressions involving addition and multiplication. To store the value we use a `val` synthesized attribute. Additionally, the processor can use a memory of predefined constants, which is propagated using an `env` inherited attribute.

Fig. 2 shows the translation scheme for this attribute grammar. In order to make the encoding more readable, we intro-

duce some constants for attribute instance indexes and for semantic function integer identifiers.

```

E ::= E + T
E1.env↓ = E0.env↓
T.env↓ = E0.env↓
E0.val↑ = E1.val↑ + T.val↑
E ::= T
T.env↓ = E.env↓
E.val↑ = T.val↑
T ::= T * F
T1.env↓ = T0.env↓
F.env↓ = T0.env↓
T0.val↑ = T1.val↑ * F.val↑
T ::= F
F.env↓ = T.env↓
T.val↑ = F.val↑
F ::= n
F.val↑ = toNum(n.lex↑)
F ::= id
F.val↑ = valOf(F.env↓, id.lex↑)
F ::= ( E )
E.env↓ = F.env↓
F.val↑ = E.val↑

```

Fig 1. Example attribute grammar. To improve readability, synthesized attribute occurrences are suffixed with ↑, and inherited occurrences are suffixed with ↓.

In this translation scheme, the first rule (which is not present in the original grammar) plays the role of initiating the processing. Indeed:

- It sets `env` in the root of the parse tree (we suppose the environment is returned by the `getEnv` external procedure).
- Then, it prints the value of `val` in such a root.
- Finally, it releases the root's attribute instance list.

The other rules are obtained by a step-by-step application of the guidelines described in the previous subsection. For instance, the encoding (shaded in Fig. 2) of the first rule of the attribute grammar (shaded in Fig. 1) is obtained as follows:

- Since `E`, 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. Notice that the resulting attribute instance list is assigned to `$$`, which in YACC-like notation is the pseudo-variable for the semantic value of the rule's LHS.
- From the first equation, we get $E_1.env$ depends on $E_0.env$. This dependency is declared by `mkDep($1[env], $$[env])`, since (i) `$1` refers, in YACC-like notation, to the semantic value of E_1 , and (ii) `$$` refers, as said before, to the semantic value of E_0 .
- In a similar way, the other three `mkDep` actions are derived from the other two equations. Notice that the third equation yields two `mkDep` actions, since, according to it, $E_0.val$ depends on two different attributes: $E_1.val$ and $T.val$.
- In their turn, each equation yields an `inst` action. For doing so, firstly we need to identify the semantic

function used in the equation. It can require some intermediate analysis. For instance, to make the semantic function apparent, $E_1.env↓ = E_0.env↓$ must be actually read as $E_1.env↓ = \lambda_v(v)E_0.env↓$. Thus, we can assign an integer code to this $\lambda_v(v)$ semantic function (in Fig. 2, this code is given by the `IDEN` constant). A similar technique can be used for equations sides involving more complex expressions. For instance, $E_0.val↑ = E_1.val↑ + T.val↑$ can be actually read as $E_0.val↑ = \lambda_{v_0}(\lambda_{v_1}(v_0+v_1))E_1.val↑ + T.val↑$, which leads to identify $\lambda_{v_0}(\lambda_{v_1}(v_0+v_1))$ as the semantic function (it is identified by the `ADD` constant in Fig. 2).

```

def env=0; def val=1;
def IDEN=0; def ADD=1; def MUL=2;
def TONUM=3; def VALOF=4;
S ::= E {
  set($1[env], getEnv());
  print(val($1[val]));
  release($1);
}
E ::= E + T {
  $$ := mkCtx(2);
  mkDep($1[env], $$[env]); mkDep($3[env], $$[env]);
  mkDep($$[val], $1[val]); mkDep($$[val], $3[val]);
  inst($1[env], IDEN);
  inst($3[env], IDEN);
  inst($$[val], ADD);
  release($1); release($3);
}
E ::= T {
  $$ := mkCtx(2);
  mkDep($1[env], $$[env]); mkDep($$[val], $1[val]);
  inst($1[env], IDEN);
  inst($$[val], IDEN);
  release($1);
}
T ::= T * F {
  $$ := mkCtx(2);
  mkDep($1[env], $$[env]); mkDep($3[env], $$[env]);
  mkDep($$[val], $1[val]); mkDep($$[val], $3[val]);
  inst($1[env], IDEN);
  inst($3[env], IDEN);
  inst($$[val], MUL);
  release($1); release($3);
}
T ::= F {
  $$ := mkCtx(2);
  mkDep($1[env], $$[env]); mkDep($$[val], $1[val]);
  inst($1[env], IDEN);
  inst($$[val], IDEN);
  release($1);
}
F ::= n {
  $$ := mkCtx(2);
  mkDep($$[val], $1[lex]);
  inst($$[val], TONUM);
  release($1);
}
F ::= id {
  $$ := mkCtx(2);
  mkDep($$[val], $$[env]); mkDep($$[val], $1[lex]);
  inst($$[val], VALOF);
  release($1);
}
F ::= ( E ) {
  $$ := mkCtx(2);
  mkDep($$[val], $2[val]);
  mkDep($2[env], $$[env]);
  inst($2[env], IDEN);
  inst($$[val], IDEN);
  release($2);
}

```

Fig 2. Encoding of the attribute grammar in Fig. 1.

- Finally, we include a `release` action for each symbol in the rule's RHS having semantic attributes.

Finally, in addition to the translation scheme, we need to provide a suitable semantic function manager. It is depicted by the pseudo-code in Fig. 3. Basically, it is a dispatcher that, according to the function's integer identifier, applies the actual function on the sequence of semantic attribute instances¹.

```

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

```

Fig 3. Semantic function manager for the attribute grammar in Fig. 1.

III. INCORPORATING A DEMAND-DRIVEN EVALUATION FRAMEWORK

In order to make possible the execution of the encodings proposed in the previous section, we need to implement the basic attribution operations. In this section we describe a straightforward implementation supporting a *demand-driven* evaluation style (see, for instance [5] [9]). In this implementation, semantic evaluation starts once the sentence has been completely parsed. In 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.

In the following subsections we describe the resulting framework explaining how attribute instances are represented (subsection III.A). Then, we specify how the attribution operations work (subsection III.B). Finally, we illustrate the complete framework with an example (subsection III.C). 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.

A. Representing the instances of the semantic attributes

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:

- Each attribute instance points to those attribute instances required to compute it (in a similar way to the *reversed* dependency graph used in [5]).
- In addition, it explicitly stores the identifier of the semantic function to be used in this computation.

¹Remark that, by the sake of generality, we are intentionally using a minimal set of programming language features. Indeed, by using a more sophisticated programming paradigm (e.g., a language equipped with higher-order features), it could be possible to give more elegant solutions to these basic conceptualizations.

TABLE II.

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

B. Implementing the attribution operations

Table 3 outlines, using pseudo-code, the implementation of the attribution operations. In this pseudo-code, references are intended to work like 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 many fresh attribute instances as needed.
- `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, decreases in 1 its reference count. If this count becomes 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.

Thus, the demand-driven evaluation process arises from the interplay of the `val` attribution operation and the semantic function manager. Notice that, in our minimalistic conceptualization, we assume this manager has the pre-estab-

lished `exec` name, and its implementation is changed from encoding to encoding².

TABLE III.

IMPLEMENTATION OF THE ATTRIBUTION OPERATIONS FOR ALLOWING A DEMAND-DRIVEN EVALUATION STYLE.

Operation	Implementation
<code>mkCtx(n)</code>	<code>as := new list</code> <code>for i := 0 to n-1 do</code> <code>add(as, new attribute)</code> <code>end for</code> <code>return as</code>
<code>mkDep(a₀, a₁)</code>	<code>add(a₀.deps, a₁)</code> <code>a₁.refcount := a₁.refcount + 1</code>
<code>inst(a,f)</code>	<code>a.semFun := f</code>
<code>release(as)</code>	<code>foreach a in as do</code> <code>release(a)</code> <code>end foreach</code> <code>delete as</code>
<code>release(a)</code>	<code>a.refcount := a.refcount - 1</code> <code>if a.refcount = 0 then</code> <code>foreach a' in a.deps do</code> <code>release(a')</code> <code>end foreach</code> <code>delete a.deps</code> <code>end if</code> <code>delete a</code>
<code>set(a,val)</code>	<code>a.value := val</code> <code>a.available := true</code>
<code>val(a)</code>	<code>if ¬ a.available then</code> <code>set(a, exec(a.semFun, a.deps))</code> <code>release(a.deps)</code> <code>end if</code> <code>return a.value</code>

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 deallocating 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 becomes unreachable. It can be due to attribute instances that are not finally required in any computation (e.g., `F.env` in a `F ::= id` context), 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, ...).

C. Example

In order to illustrate the internal functioning of the framework, we will use the example developed in subsection II.D, together with the input sentence `5 * (6 + x)`.

²Again it is possible to achieve more elegant solutions by using a programming language with a minimal higher-order support (e.g., a conventional object-oriented language). Nevertheless, our conceptualization keeps the essence of this evaluation approach.

Action	Input	Parse Stack
1. init	<code>*(6+x)\$</code>	
2. shift	<code>(6+x)\$</code>	<code>n^[1]</code>
3. reduce <code>F ::= n</code>	<code>(6+x)\$</code>	<code>F^[2,3]</code>
4. reduce <code>T ::= F</code>	<code>(6+x)\$</code>	<code>T^[4,5]</code>
5. shift	<code>6+x)\$</code>	<code>T^[4,5]*</code>
6. shift	<code>+x)\$</code>	<code>T^[4,5]*(</code>
7. shift	<code>x)\$</code>	<code>T^[4,5]*(n^[6]</code>
8. reduce <code>F ::= n</code>	<code>x)\$</code>	<code>T^[4,5]*(F^[7,8]</code>
9. reduce <code>T ::= F</code>	<code>x)\$</code>	<code>T^[4,5]*(T^[9,10]</code>
10. reduce <code>E ::= T</code>	<code>x)\$</code>	<code>T^[4,5]*(E^[11,12]</code>
11. shift	<code>)\$</code>	<code>T^[4,5]*(E^[11,12] +</code>
12. shift	<code>\$</code>	<code>T^[4,5]*(E^[11,12]+id^[13]</code>
13. reduce <code>F ::= id</code>	<code>\$</code>	<code>T^[4,5]*(E^[11,12]+F^[14,15]</code>
14. reduce <code>T ::= F</code>	<code>\$</code>	<code>T^[4,5]*(E^[11,12]+T^[16,17]</code>
15. reduce <code>E ::= E+T</code>	<code>\$</code>	<code>T^[4,5]*(E^[18,19]</code>
16. shift	<code>\$</code>	<code>T^[4,5]*(E^[18,19])</code>
17. reduce <code>F := (E)</code>	<code>\$</code>	<code>T^[4,5]*(F^[20,21]</code>
18. reduce <code>T := T*F</code>	<code>\$</code>	<code>T^[22,23]</code>
19. reduce <code>E := T</code>	<code>\$</code>	<code>E^[24,25]</code>
20. reduce <code>S := E</code>	<code>\$</code>	<code>S</code>

Fig 4. Evolution of the translator generated from Figure 2 while analyzing `5 * (6 + x)`

Fig. 4 illustrates the evolution of the parser. Each symbol in the parse stack is superscripted by the list of the references to their attribute instances. Fig. 5 outlines the dependency structure created in the heap. In this structure, nodes correspond to attribute instances, while dependencies are indicated by mean of arrows. Each instance is accompanied by a numeric identifier (it is also used to indicate references in the parse stack), and by its *duration* (i.e., the parse action in which the instance was created, followed by the parse action in which it was deleted). For example, the instance 16 was created in the action 14 (reduction of the `T ::= F` rule), and it was destroyed in action 20 (once the parsing concluded and semantic evaluation was activated as a consequence of consulting `val` in the parse tree root).

Finally, it is important to remark several points. On one hand, it should be notice the parse tree is never explicitly built, since the process only requires the underlying dependency graph. Additionally, dependencies in Fig. 5 are reverted with respect to the usual convention, according to which, when `a` is used to compute `b`, an arc starting in `a` and finishing in `b` is used [8]. Indeed, dependencies actually represent the contents of the `deps` field. Additionally, the fact they appear reversed with respect to the usual convention em-

phasized the demand-driven nature of this evaluation strategy. Also, while most of the instances exists until the end of the process, this example makes apparent how those instances that become unreachable are readily de-allocated. For example, consider action 4: when rule $T ::= F$ is reduced, the instance for $F.env$ (i.e., the instance 3) is no longer needed, and therefore it can be de-allocated. Actually, this instance is de-allocated as consequence of releasing the F attribute instance list. By last, notice how lexical attributes for terminal symbols are created in the *shift* action that precedes the actual shift of such a symbol, or during parse initialization, in the case of the first shift. It is due to lexical attributes are actually created by the scanner, and we suppose 1-lookahead parsers, as those generated by LALR(1) parser generators.

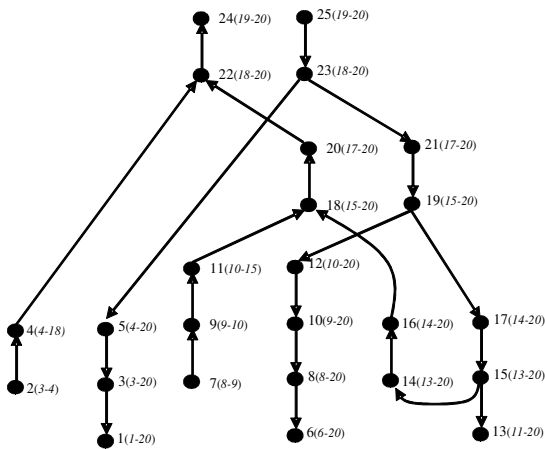


Fig 5. Dependency structure created in the heap as by the process outlined in Fig. 4.

IV. INCORPORATING A DATA-DRIVEN EVALUATION FRAMEWORK

In this section we describe an alternative implementation of the attribution operations, which leads to a *data-driven* evaluation style (see, for instance, [7]). In this evaluation style, attribute instances are scheduled for being evaluated as soon as the values for all the instances on which it depends are available. Thus, this method can shorten the durations of attribute instances. Additionally, it can interleave evaluation with parsing. These features can result of interest to process 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.

As in the previous section, we outline the representation of attribute instances (subsection IV.A), the implementation of attribute operations (subsection IV.B), and we illustrate how the method works with an example (subsection IV.C).

A. Representing the instances of the semantic attributes

Table 4 outlines the representation of attribute instances in the data-driven style. Notice that, in addition to the list of in-

stances on which an instance depends, it is needed to maintain the reverse relationship (i.e., each attribute instance must refer to those instances which depend on it). Indeed, this representation is similar to the used by networks of *observables-observers* in the *observer* object-oriented pattern [4]³.

TABLE IV.

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.	\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
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 <i>deps</i> whose values have not yet been determined.	0
semFun	It stores the integer code of the semantic function required to compute the value.	\perp
instrumented	<i>True</i> if <i>semFun</i> was set, <i>false</i> otherwise.	false
refcount	A counter of references to this attribute instance (used to enable garbage collection).	1

B. Implementing the attribution operations

Table 5 outlines the pseudo-code of the attribution operations whose implementation differs from those in the demand-driven style. In 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 was already computed. If it is not available, the first instance must be added to its *obs* list, since such an instance depends on its value, a value which is not yet available.
- On its hand, *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 establishment of all the required dependencies before instrumentation, which is ensured by our encoding pattern.
- *Set* must take care of decrementing the *required* counters in all the instances depending

³As with the demand-driven style, this representation could be simplified, 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.

of the current one. In addition, if a counter becomes 0, it must enforce the evaluation of the corresponding instance.

- Finally, `val` immediately computes the value, unless the instance has not been yet instrumented.

TABLE V.

IMPLEMENTATION OF THE ATTRIBUTION OPERATIONS FOR ALLOWING A DATA-DRIVEN EVALUATION STYLE (ONLY THOSE IMPLEMENTATIONS DIFFERING FROM TABLE III ARE PRESENTED).

Operation	Implementation
<code>mkDep(a_0, a_1)</code>	<pre> add (a_0.deps, a_1) a_1.refcount := a_1.refcount + 1 if $\neg a_1$.available then add (a_1.obs, a_0) a_0.required := a_0.required + 1 a_0.refcount := a_0.refcount + 1 end if </pre>
<code>inst(a, f)</code>	<pre> a.semFun := f a.instrumented := true if a.required = 0 then $val(a)$ end if </pre>
<code>set(a, val)</code>	<pre> 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) </pre>
<code>val(a)</code>	<pre> if a.instrumented then set(a, exec(a.semFun, a.deps)) a.available := true release(a.deps) end if </pre>

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. As a consequence, garbage collection also interplays with parsing, and, therefore, this method can incur in less heap overhead. It can be realized by considering the implementation of an *s-attributed* grammar (i.e., a grammar with only synthesized attributes) [1]. In this case, LHS attribute instances are computed when they are instrumented, and RHS attribute instances are garbage collected immediately before the reduction of the corresponding rules. On other cases, the behavior strongly depends on the nature of inherited attributes. In the extreme case (e.g., the example developed in subsection II.D), the dependency structure will be fully constructed, and evaluation will be delayed until the end of parsing, as in the demand-driven style. Still in these cases, it is possible to apply some straightforward optimizations on the resulting encoding, based on the use of *marker* non-terminals [1], in order to improve performance.

```

def env=0; def val=0;
def IDEN=0; def ADD=1; def MUL=2;
def TONUM=3; def VALOF=4;
S ::= M0 E {
  print (val ($2[val]));
  release ($1); release ($2); }
M0 ::=  $\lambda$  {
  $$ := mkCtx (1);
  set ($$[env], getEnv ()); }
E ::= E + M1 T {
  $$ := mkCtx (1);
  mkDep ($$[val], $1[val]); mkDep ($$[val], $4[val]);
  inst ($$[val], ADD);
  release ($1); release ($3); release ($4); }
M1 ::=  $\lambda$  {
  $$ := mkCtx (1);
  mkDep ($$[env], $-2[env]);
  inst ($$[env], IDEN) }
E ::= T {
  $$ := mkCtx (1);
  mkDep ($$[val], $1[val]);
  inst ($$[val], IDEN);
  release ($1); }
T ::= T * M1 F {
  $$ := mkCtx (1);
  mkDep ($$[val], $1[val]); mkDep ($$[val], $4[val]);
  inst ($$[val], MUL);
  release ($1); release ($3); release ($4); }
T ::= F {
  $$ := mkCtx (1);
  mkDep ($$[val], $1[val]);
  inst ($$[val], IDEN);
  release ($1); }
F ::= n {
  $$ := mkCtx (1);
  mkDep ($$[val], $1[lex]);
  inst ($$[val], TONUM);
  release ($1); }
F ::= id {
  $$ := mkCtx (1);
  mkDep ($$[val], $0[env]); mkDep ($$[val], $1[lex]);
  inst ($$[val], VALOF);
  release ($1); }
F ::= ( M2 E ) {
  $$ := mkCtx (1);
  mkDep ($$[val], $3[val]);
  inst ($$[val], IDEN);
  release ($2); release ($3); }
M2 ::=  $\lambda$  {
  $$ := mkCtx (1);
  mkDep ($$[env], $-1[env]);
  inst ($$[env], IDEN) }

```

Fig 6. Result of optimizing the translation scheme of Figure 2 with the help of markers to get the most of the data-driven evaluation style.

C. Example

In order to illustrate the potential advantages of the data-driven method with respect to heap requirements, we will slightly modify the encoding of Fig. 2 by introducing marker non-terminals (i.e., new non-terminal symbols defined by empty rules) in strategic places⁴.

These marker non-terminals will allocate references to the `env` attribute instance for their immediate successors in the parse stack. It lets us discard equations to propagate the environment along the parse tree left-spines. The resulting encoding is shown in Fig. 6.

Fig. 7 illustrate the evolution of the parser while analyzing the sentence $5 * (6 + x)$. Fig. 8 shows the dependency structure created in the heap. Notice how the marker-based optimization performed on the encoding makes it possible to get a behavior equivalent to a *one-pass, on-the-fly*, evaluation of the semantic attributes, as the durations indicated in Fig. 8 make apparent.

⁴It must be carefully done, as the resulting context-free grammar can lose its desired character -e.g., LALR(1).

Action	Input	Parse Stack
1. init	*(6+x)\$	
2. reduce $M0 ::= \lambda$	*(6+x)\$	$M0^{[2]}$
3. shift	(6+x)\$	$M0^{[2]}n^{[1]}$
4. reduce $F ::= n$	(6+x)\$	$M0^{[2]}F^{[3]}$
5. reduce $T ::= F$	(6+x)\$	$M0^{[2]}T^{[4]}$
6. shift	6+x)\$	$M0^{[2]}T^{[4]}*$
7. reduce $M1 ::= \lambda$	6+x)\$	$M0^{[2]}T^{[4]}*M1^{[5]}$
8. shift	+x)\$	$M0^{[2]}T^{[4]}*M1^{[5]}($
9. reduce $M2 ::= \lambda$	+x)\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}$
10. shift	x)\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}n^{[6]}$
11. reduce $F ::= n$	x)\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}F^{[8]}$
12. reduce $T ::= F$	x)\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}T^{[9]}$
13. reduce $E ::= T$	x)\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}E^{[10]}$
14. shift)\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}E^{[10]}+$
15. reduce $M1 ::= \lambda$)\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}E^{[10]}+M1^{[12]}$
16. shift	\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}E^{[10]}+M1^{[12]}id^{[11]}$
17. reduce $F ::= id$	\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}E^{[10]}+M1^{[12]}F^{[13]}$
18. reduce $T ::= F$	\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}E^{[10]}+M1^{[12]}T^{[14]}$
19. reduce $E ::= E+MT$	\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}E^{[15]}$
20. shift	\$	$M0^{[2]}T^{[4]}*M1^{[5]}(M2^{[7]}E^{[15]})$
21. reduce $F ::= (M2E)$	\$	$M0^{[2]}T^{[4]}*M1^{[5]}F^{[16]}$
22. reduce $T ::= T*M1F$	\$	$M0^{[2]}T^{[17]}$
23. reduce $E ::= T$	\$	$M0^{[2]}E^{[18]}$
24. reduce $S ::= M0 E$	\$	S

Fig 7. Evolution of the translator generated from Figure 6 while analyzing $5 * (6+x)$

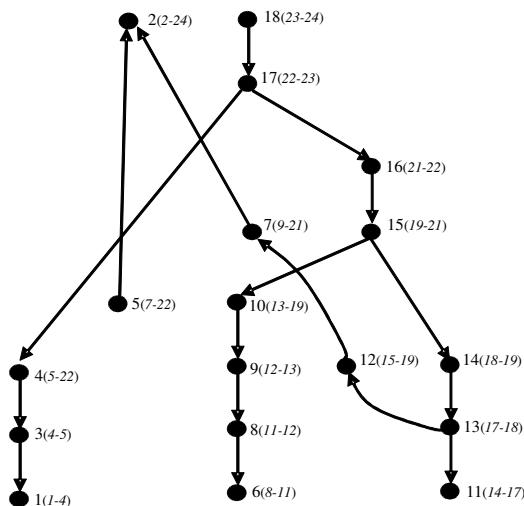


Fig 8. Dependency structure created in the heap as by the process outlined in Fig. 7

V. CONCLUSIONS AND FUTURE WORK

This paper has shown how to systematically encode arbitrary non-circular attribute grammars in the input languages

of bottom-up, LALR(1) parse generation tools like YACC, BISON or CUP. It is done using a small set of attribution operations. These operations, in their turn, can be implemented of 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 one supporting a data-driven one. The results of this work can be useful to promote a systematic method of using conventional bottom-up parse generation tools to yield final implementations. This method starts with the initial encoding of an attribute grammar-based specification, and then it evolves it in a final implementation by applying systematic implementation patterns and techniques. Besides, the method facilitates the incremental introduction of new language features, since they can be described according to attribute grammar conventions, then readily encoded in the implementation, and finally optimized according to implementation-dependent criteria. Therefore, the method transports the attribute grammar amenability for doing modular and extensible specifications incrementally to an implementation process based on parse 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. As future work, we plan to apply the method to descent parser generation tools (e.g., JavaCC or ANTLR).

ACKNOWLEDGMENT

Thanks are due to the project grants TIN2010-21288-C02-01.

REFERENCES

- [1] Aho A.V, Lam M.S, Sethi R, Ullman J.D. 2006. Compilers: principles, techniques and tools (2nd Edition). Addison-Wesley.
- [2] Akker, R; Melichar, B.; Tarhio, J. The Hierarchy of LR-attributed grammars. WAGA'90, Paris, France, September 19-21. 1990
- [3] Ekman, T., Hedin, G. The JastAdd system - modular extensible compiler construction. Sc. of Comp. Prog. 69(1-3), 14-26. 2007
- [4] Gamma,E; Helm,R; Jhonson,R; Vlissides,J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995
- [5] Jalili, F. A general linear-time evaluator for attribute grammars. ACM SIGPLAN Notices 18(9), 35-44. 1983
- [6] Katwijk, J. A preprocessor for YACC or a poor man's approach to parsing attributed grammar. ACM SIGPLAN Notices 18(10), 12-15. 1983
- [7] Kennedy, K.; Ramanathan, J. A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing. ACM Transaction of Programming Languages and Systems 1(1), 142-160. 1979
- [8] Knuth, D. E. Semantics of Context-free Languages. Math. System Theory 2(2), 127-145. 1968. See also the correction published in Math. System Theory 5, 1, 95-96
- [9] Magnusson, E.; Hedin, G. Circular reference attributed grammars—their evaluation and applications. Sc. of Comp. Prog. 68(1), 21-37. 2007
- [10] Memik, M.,Lenic, M., Acdicausevic, E., Zumer, V. LISA: An Interactive Environment for Programming Language Development. CC 2002, Grenoble, France, April 8-12, 2002
- [11] Paaki, J. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Comp. Surveys, 27, 2, 196-255. 1995