A Prolog Framework for the Rapid Prototyping of Language Processors with Attribute Grammars

José Luis Sierra [1] Alfredo Fernández-Valmayor [2,3]

Dpto. Sistemas Informáticos y Programación
Fac. Informática. Universidad Complutense
Madrid, Spain

Abstract

In this paper, we describe PAG (*Prototyping with Attribute Grammars*), a framework for building Prolog prototypes from specifications based on attribute grammars, which we have developed for supporting rapid prototyping activities in an introductory course on language processors. This framework works for general non-circular attribute grammars with arbitrary underlying context-free grammars, includes a specification language embedded in Prolog that strongly resembles the attribute grammar notations explained in the course cited, and lets students produce comprehensible prototypes from their specifications in a straightforward way.

Key words: attribute grammars, language prototyping framework, education in language processors, Prolog

1 Introduction

Attribute grammars have been recognized as very valuable artifacts to bring together the design of programming languages and the construction of their processors [13][15][20]. We have also adopted attribute grammar methodology as the basis for our pedagogical strategy in teaching a graduate level introductory course on language processors at the *Complutense* University of Madrid (Spain). In this course, we encourage a clear distinction between the specification of the source language and its translation, and the subsequent implementation of the processor. During specification, students are compelled to use attribute grammars, and subsequently to apply systematic techniques

[1] Email:jlsierra@sip.ucm.es
[2] Email:alfredo@sip.ucm.es
[3] The Spanish Committee of Science and Technology (TIC2002-04067-C03-02, TIN2004-08367-C02-02 and TIN2005-08788-C04-01) has partially supported this work.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

to move to a one-pass top-down or bottom-up based implementation. We also promote an intermediate prototyping stage, where students program a prototype in Prolog that closely mirrors the specification. This stage is founded both on a technical basis (e.g. to let the student validate and improve the quality of the specifications) and on a pedagogical one (e.g. to motivate the student to undertake an otherwise unpleasant activity). Our choice of Prolog instead of more specific environments [14][20] is because our students study logic programming as a core undergraduate topic, and therefore they are familiar with the use of this language for developing small- and mid-scale programming projects.

We have been promoting Prolog's definite clause grammars (DCGs) [1] as a prototyping technique for several years. While DCGs have been a very useful mechanism in letting students comprehend the main concepts behind syntax-directed translation techniques, and even many fundamental concepts behind logic programming, we have also detected several practical limitations: the lack of support for left-recursive specifications, and the need to be aware of the evaluation order for semantic equations. To overcome these limitations we have developed PAG (*Prototyping with Attribute Grammars*), a Prolog framework for the rapid prototyping of language processors from their attribute grammar based specifications. PAG includes a specification language that closely resembles the usual notation for attribute grammars that we use in our lessons. The combination of a general parsing algorithm with a simple technique for attribute evaluation lets PAG accept general non-circular attribute grammars with arbitrary (even ambiguous) underlying context-free grammars, a must for a successful prototyping activity.

The structure of this paper is as follows. In section 2 we describe the pedagogical context where PAG has arisen. In section 3 we overview PAG. In section 4 we describe its implementation. Finally, section 5 provides some conclusions and some lines of future work.

2 The Pedagogical Context

In this section we describe the pedagogical context of the present work, centered on our course on *Language Processors* at the *Complutense* University. In subsection 2.1 we briefly outline the main aspects of this course. In subsection 2.2 we describe how we have addressed rapid prototyping using Prolog's DCGs, as well as their pedagogical strengths and weaknesses. Finally, in subsection 2.3 we establish a set of initial requirements for a prototyping framework that preserves the advantages and overcomes the limitations of the DCG-based approach, and we justify the design and construction of PAG on the basis of these requirements.

2.1 The Course on Language Processors

Language Processors is a two-semester course of the Graduate Degree in Computer Science at the *Complutense* University. Our main pedagogical goal is to let students learn methods for the systematic description of computer languages and for the systematic development of their processors. Therefore, in this course we teach how systematically to *specify* computer languages and their processors, and how to systematically *implement* these processors using different standard techniques: hand-coded and automatically generated predictive-recursive top-down translators, automatically generated table-driven top-down translators, and also automatically generated LR bottom-up translators [2][8].

In our pedagogical method, we adopt a problem-based learning approach where students, working in groups, incrementally solve the problems posed by the specification and the construction of an interpretative compiler of a Pascal-like language. In order to facilitate the maintenance and evolution of the language and of its processor, we promote a clear distinction between specification and implementation. As said before, the central descriptive formalism used during specification is based on attribute grammars, although we also use other complementary resources (e.g. regular expressions and/or finite automata for describing lexical aspects, and semiformal algorithmic specifications for describing target machines and their supported object languages).

During the teaching of the course we have noted how students are reluctant to assimilate the convenience of separating specification and implementation. To convince them of the advantages of this separation of concerns we have adopted the following strategies:

- We make the incremental development process model usually followed in the construction of language processors explicit. Indeed, we start by proposing the implementation of a processor for a minimal language (two primitive types, an expression language involving basic operators with different precedence and association rules, declaration of variables and the assignment statement). Once students have constructed this processor, we propose successive extensions of the basic language: control statements, user-defined types, and recursive subprograms.
- We require several alternative implementations. The processor for the minimal language is initially hand-coded as a predictive-recursive descendent translator. Once we have introduced the students to more advanced implementation techniques, they must refactor the translator in terms of the techniques introduced , using suitable domain-specific supporting tools. In addition, they incorporate the successive extensions to the language proposed, either in the hand-coded processor or by using one of the tools tested.
- We introduce a rapid prototyping activity.

While incremental development and alternative implementations are useful

in order to appreciate how carefully prepared specifications can pay off during the development process, we have realized that they are not enough to convince students of their benefits. Indeed, it is not unusual to see how excellent students, overwhelmed by the work to be done, concentrate on the programming tasks while abandoning specification activities. This situation has been largely alleviated by rapid prototyping. Indeed, this activity is highly motivating for students, since they are able to get a running processor at a very early stage of the development process. Therefore, students feel that they are doing worthwhile work during specification, and they concentrate on this activity. This effort has very positive repercussions in the rest of the process. The next subsection concentrates on rapid prototyping in our pedagogical method.

2.2 Rapid Prototyping with Definite Clause Grammars

Prolog DCGs have been largely recognized as valuable artifacts for prototyping language translators [5][24][25]. We have also realized this fact as part of our teaching experience. Indeed, as said before, we have adopted DCGs as a basic prototyping technique for several years, since our students have a good working experience with Prolog as part of their undergraduate education. This enables us to introduce the technique as syntactic sugar for the direct Prolog encoding of a translation schema in one or two one-hour classroom sessions.

As we have realized during the use of the technique, our students have found the use of DCGs very valuable for understanding the main concepts behind language translation. More concrete than attribute grammars, DCGs have also helped many of them to better understand the operational mechanisms behind the more abstract attribute grammar-based specifications. In addition, we have been pleasantly surprised to discover how DCGs have helped some of our students to better understand some of the more important features of logic programming: non deterministic execution and the use of unification to deal with incomplete structures [24].

Regardless of these advantages, the approach also exhibits several limitations, as exposed in the introduction. Prolog's DCGs do not work with left-recursive underlying context-free grammars (with the exception of specialized implementations that make use of *tabling*, like [26]), therefore hindering many otherwise natural specifications (e.g. left-recursive syntax for expressions with left-associative operators). Also the usual DCG style promotes attribute evaluation during parsing, which forces students to be aware of the evaluation order for semantic equations. Thus, the primary spirit of an attribute grammar-based specification is broken.

Example 2.1 In Fig. 1 we illustrate the use of DCGs in the construction of prototypes based on attribute grammars. Notice that, in order to transform the specification into a suitable form for prototyping, the student must make an effort comparable to transforming the attribute grammar into a syntax-directed translation schema oriented to a top-down recursive-descent imple-

(a)

```
exp ::= exp + term
   exp0.v = exp1.v + term.v
exp ::= term
   exp.v = term.v
term ::= num
    term.v = num.v
term ::= ( exp )
    term.v = exp.v
```

(b)

```
exp(Vo) --> term(V1),rexp(V1,Vo).
rexp(Vho,Vo) --> [+],term(V1),{Vh1 is Vho+V1}, rexp(Vh1,Vo).
rexp(V,V)  --> [].
term(V) --> [num(V)].
term(V) --> ['('],exp(V),[')'].
```

Fig. 1. (a) A very simple attribute grammar; (b) DCG-based prototype resulting from (a). Left-recursion has been eliminated and an explicit evaluation order for the semantic equations has been chosen.

mentation.

The limitations exposed can be frustrating for the average student, thus defeating his/her acceptance of general attribute grammars as a good way to think about programming language design and implementation. Indeed, we have realized that many students concentrate on producing specifications that avoid left-recursion in the underlying grammars, which can be readily translated onto DCG-based prototypes, but which in some cases are rather unnatural and not suitable for producing certain types of implementations (e.g. based on LR translators).

2.3 A better Prototyping Alternative

The limitations detected with the use of DCGs during prototyping have led us to consider alternative approaches. Among the initial requirements for a suitable alternative we established the following:

- *Simplicity* requirement. The selected approach should maintain the simplicity of DCGs. It should be easily assimilated by our students and the impact on the current course schedule should be minimized. Ideally, the approach should provide a very simple formalism, close to the notation used for attribute grammars in our lessons (see Fig. 1a for an example of such a notation).
- *Syntactic freedom* requirement. The approach should be able to deal with arbitrary context-free syntax.
- *Semantic freedom* requirement. The approach should deal with general non

circular attribute grammars. Indeed, in our introductory course we do not deal with the possibility of circular attribute grammars, and we identify circularity as an erroneous condition.

- *Comprehensibility* requirement. The generated prototypes should be easily understood by students, who should be able to trace their behavior when required.
- *Deployment* requirement. The supporting tool should be portable and easy to install. In addition, it should be modular and easy to integrate into web-based learning scenarios like [22], and those based on the learning object paradigm [21][23], since we are making intensive use of e-learning solutions in order to accommodate a smooth migration of our pedagogical methods to the forthcoming European Space of Higher Education [7].

When looking for a suitable solution meeting all these requirements, we considered the following alternatives:

- Using programming languages that, like Elegant [11] or ALE [4], are derived from or closely related to the attribute grammar formalism. Nevertheless, this alternative clearly violates the *simplicity* requirement, since we must spend a considerable amount of time teaching the new language to our students.
- Using an existing attribute-grammar based environment, like FNC-2 [12], Eli [9] or Cocktail [10]. Some of them, such as LISA [17] are recognized as especially well-suited for educational purposes [18]. Nevertheless, this kind of environments is usually conceived as development tools instead as prototyping ones. They usually integrate parser generators for deterministic classes of context-free grammars (e.g. LL(1), LALR(1), etc.), which violates the *syntactic freedom* requirement. Also they are oriented to generating efficient static attribute evaluators, which violates the *semantic freedom* requirement, and, more important, the *comprehensibility* one (although comprehensibility can be enhanced by using appropriate GUI support, as in LISA [18]).Finally, these systems integrate specifications languages with powerful features (e.g. attribution patterns, multiple inheritance, template rules, etc). While these features are very valuable during development, they violate the *simplicity* requirement in our educational context.

In addition, all these third-party alternatives also could make deployment in a web-based learning scenario more difficult than our own tool. Once we concluded this exploration without finding the *perfect* candidate, we decided to undertake the design and construction of PAG. The rest of the paper describes the technical details of the resulting framework.

3 The Prototyping Framework

PAG produces working prototypes from attribute grammar specifications and suitable Prolog implementations of the semantic functions required. In this section, we survey the framework. In subsection 3.1 we describe the structure of specifications in PAG. In subsection 3.2 we describe how the framework is used during prototyping.

3.1 Specifications in PAG

A specification in PAG is formed by two parts:

- The specification of the attribute grammar. This specification is given in a Prolog-embedded domain-specific language that strongly resembles the basic notation for attribute grammars used in a typical introductory course on language processor construction.
- The definition of the semantic functions. This definition can be kept independent of the attribute grammar, and relates the signatures of the semantic functions with the Prolog goals used to compute them.

Specification ::= *Symbols Axiom* (*Rule*)+
Symbols ::= (nt‘(’**symbol**,**inh-attr-list**,**syn-attr-list**‘)’. |
t‘(’**symbol**,**attribute-list**‘)’.)+
Axiom ::= axion‘(’**symbol**‘)’.
Rule ::= **head-nt** ‘::=’ *Body* (, *Equations*)?.
Body ::= [ ] | **symbol** (, **symbol**)*
Equations ::= *Equation* (,*Equation*)*
Equation ::= *Attribute* = **definition**
Attribute ::= **att-name** of **symbol**

Fig. 2. Syntax of the specification language.

The syntax for the PAG attribute grammar specification language is outlined in Fig. 2. This syntax, which is embedded in Prolog with the usual facilities to introduce user-defined operators, is featured as follows:

- Non-terminals must be declared using the `nt/3` predicate. The first argument is the symbol itself. The second argument represents the inherited attributes, while the third declares the synthesized attributes.
- Terminals can be declared using the `t/2` predicate. The first argument is the terminal name, while the second one is a list with the lexical attributes. Notice that a terminal without lexical attributes does not need to be declared.
- The axiom of the grammar is distinguished using the `axiom/1` predicate.

- Attributes attached to syntactic symbols are referred to using the `of` operator. When there is more than one occurrence of a symbol in a production, the occurrence number can be indicated (by default it is the first one).
- Grammar rules are built with the `::=` operator. λ is specified with the empty list `[]`, as in DCGs. With these rules it is also possible to attach a set of semantic equations, which are specified using the `=` operator.
- The left-hand side of a semantic equation must be an attribute reference. The right side of a semantic equation can be an arbitrary Prolog term, which will usually contain references to other attributes. This term will be interpreted as the expression for computing the attribute value.

Example 3.1 In Fig. 3 we show an attribute grammar for a simple calculator language based on DESK, the example language introduced in [20]. This language also enables us to bind constants to values and to use these constants in the binding scopes. The attribute grammar associates a suitable value to each expression.

In addition to the specification of the attribute grammar, definitions need to be provided for the semantic functions used in computing the attribute values. PAG establishes the `defun/2` hook for this purpose. In this predicate, the first argument must be a term whose functor identifies the function name, and its arguments are associated with the function inputs. The second argument of `defun/2` is associated with the function result. In its definition, the body of the corresponding clause will link the function with a Prolog computation of the result. By default all the functions declared are *strict* (i.e. their arguments in semantic equations will be evaluated before applying the function). This default behavior can be altered by distinguishing the function signature with the `nonstrict/1` hook. In this case, the evaluation strategy must be customized in the definition. Finally, any undeclared semantic function will be interpreted as declared as `defun(F,F)`, and therefore as a term constructor. PAG defines several utility functions in a prelude file, which can be loaded with each specification.

Example 3.2 In Fig. 4 we include the definitions of the semantic functions used in the grammar of Fig. 3. The `+` arithmetic function is already defined in the prelude, and we only include it for the purpose of illustration. The `emptyEnv`, `mkEnv` and `valueOf` functions are used to manage a simple environment binding variables to their values.

3.2 Prototyping with PAG

PAG lets students automatically process the specifications introduced in previous subsections to generate prototypes. PAG is able to deal with general non-circular attribute grammars with an arbitrary underlying context-free syntax in a simple way. The structure of the prototypes generated is sketched in Fig. 5, and it is featured as follows:

```
nt(prog, [], [val]).
nt(exp, [envh], [val]).
nt(fact, [envh], [val]).
nt(constPart, [], [env]).
nt(constDefs, [], [env]).
nt(constDef, [envh], [env]).
t(num, [val]).
t(id,  [lex]).
axiom(prog).

prog ::= exp, constPart,
  val of prog = val of exp,
  envh of exp = env of constPart.
exp ::= exp, +, fact,
  val of exp(1) = val of exp(2) + val of fact,
  envh of exp(2) = envh of exp(1),
  envh of fact = envh of exp(1).
exp ::= fact,
  val of exp = val of fact,
  envh of fact = envh of exp.
fact ::= id,
  val of fact = valueOf(lex of id, envh of fact).
fact ::= num,
  val of fact = val of num.
constPart ::= where, constDefs,
  env of constPart = env of constDefs.
constPart ::= [],
  env of constPart = emptyEnv.
constDefs ::= constDefs, ',', constDef,
  env of constDefs(1) = env of constDef,
  envh of constDef = env of constDefs(2).
constDefs ::= constDef,
  env of constDefs = env of constDef,
  envh of constDef = emptyEnv.
constDef ::= id, =, exp,
  env of constDef =
     makeEnv(envh of constDef, lex of id, val of exp),
  envh of exp = envh of constDef.
```

Fig. 3. A PAG Attribute Grammar.

```
defun(X+Y,R) :- R is X+Y.
defun(emptyEnv,[]).
defun(makeEnv(Env,Id,Val),[(Id,Val)|Env]).
defun(valueOf(Id,Env),Val) :-
   member((Id,Val),Env),!.
```

Fig. 4. Definition of semantic functions for the attribute grammar of Fig. 3.

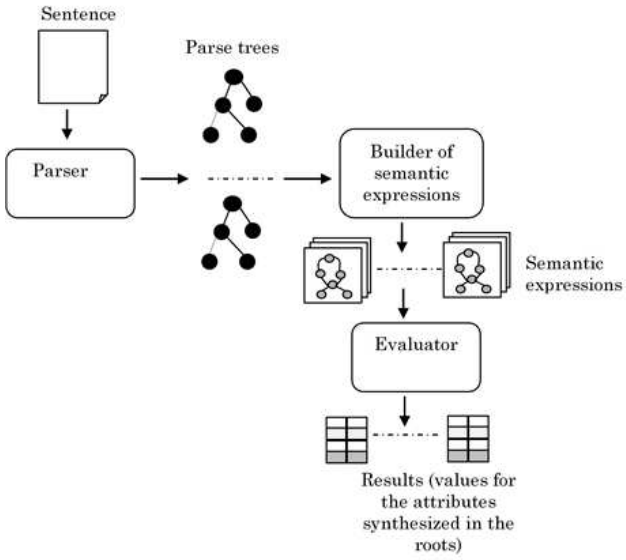


Fig. 5. Structure of prototypes built with PAG.

- The *parser* parses sequences of tokens into parse trees.
- The *builder of semantic expressions* traverses these trees and associates a suitable *semantic expression* with each attribute in each node. Semantic expressions are ground terms on the signature of semantic functions, and they will be used to compute the attribute values.
- The *evaluator* component performs the evaluation of the semantic expressions. Actually, it is only needed to evaluate the expressions attached to the synthesized attributes of the parse tree's root.

Notice that the evaluation of attributes is further split into two independent stages. The first one, which can be thought of as a *substitution* step in solving semantic equations, is performed by the builder of semantic expressions. The resulting expressions are actually evaluated by the evaluator during the second stage. Also, notice that a sentence can be parsed into several parse trees (this will be the case with ambiguous syntax). In these cases the prototype will non-deterministically yield several results.

These prototypes can be automatically generated from the specifications by using the following PAG predefined components [4]:

- The *parsing kernel*. This component contains the machinery required to parse sentences into parse trees.
- The *evaluation kernel*. This component is used to evaluate the expressions attached with the semantic attributes.
- The *generator*. This component translates the attribute grammar specification into several working components required to produce the final pro-

[4] From an implementation viewpoint, in the context of this paper *components* are constituted by a set of clauses and optionally, directives to the underlying Prolog engine.

totype.

- The *prelude*. As said before, this component defines several semantic functions that can be reused in different specifications.

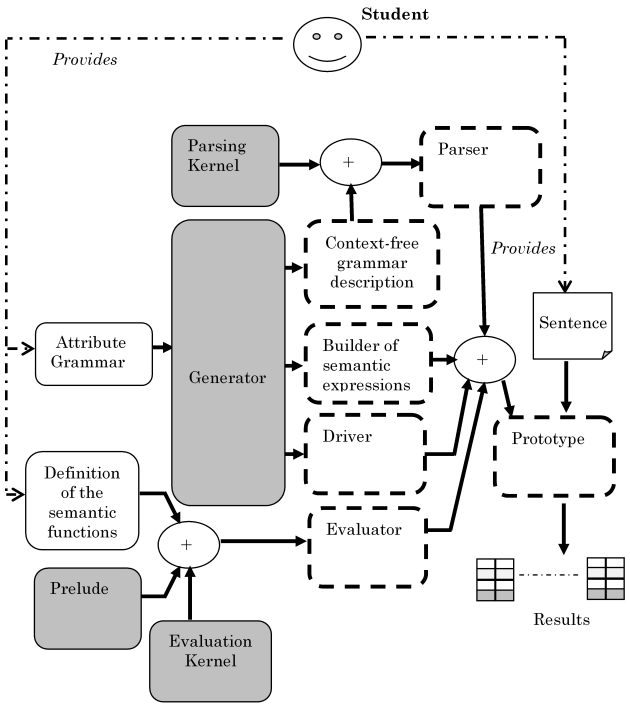


Fig. 6. Prototyping workflow in PAG. Predefined components are shadowed. Generated components are dash-lined. With + we denote unions of clause sets.

The entire process is depicted in Fig. 6, which also highlights the different components in the framework. That way, the process begins when the student specifies the prototype, providing an attribute grammar and defining the semantic functions used. The union of these semantic functions, the prelude and the evaluation kernel yields the prototype's evaluator. In turn, the attribute grammar specification is processed by the generator to produce:

- A description of the underlying context-free grammar that, added to the parsing kernel, will yield the parser component.
- A builder of semantic expressions for the final prototype.
- A driver that will glue all the prototype components together. This driver will be used by the student to run the prototype.

4 Implementing the Prototyping Framework

PAG is based on two simple principles to provide students with the expressive freedom required during prototyping:

- On one hand, the framework is able to process arbitrary context-free grammars. This is carried out by using a suitable implementation of Earley's al-

gorithm [6] in the parsing kernel, a general parsing method able to perform a reasonably efficient parsing of sentences regarding any (even ambiguous) context-free grammar.

- On the other hand, the system uses a simple technique for dealing with any non-circular attribute grammar. By interpreting semantic functions in the Herbrand domain (i.e. by interpreting them as term constructors) a logical one-pass attribute grammar (in the sense of [19]) is obtained. Indeed, the builder of semantic expressions can be considered a one-pass evaluator for such a grammar. The semantic expressions produced are subsequently evaluated by the evaluator. Therefore, instead of interleaving parsing, tree traversal, and evaluation, they are kept as separated processes. Separation of parsing and evaluation has been also proposed in [3], where lambda calculus is taken as a notation for semantic expressions and circular dependencies are transformed into lambda calculus fixpoint computations. It has also been proposed in the context of definite clause translation grammars (DCTGs) [1], where parsing yields parse trees decorated with Horn-like semantic rules. PAG also separates parsing and tree traversal to enable arbitrary context-free syntax. In addition, a simple technique is used to avoid reevaluation of common subexpressions in the semantic expressions produced.

The following subsections explore the implementation details. Subsection 4.1 presents the parsing kernel. Subsection 4.2 describes how this kernel is specialized in particular grammars to yield parsers for these grammars. Subsection 4.3 describes the pattern followed by the builders of semantic expressions. Subsection 4.4 describes the implementation of the evaluation kernel. Finally, subsection 4.5 outlines the implementation of the generator.

4.1 The Parsing Kernel

As aforementioned, the parsing kernel is based on Earley's algorithm [6]. The algorithm works for arbitrary (even ambiguous) context-free grammars and sentences of length n with a worst-case time complexity in $\mathcal{O}(n^3)$ and space complexity in $\mathcal{O}(n^2)$. Since test sentences are usually small, these overheads are acceptable. In addition, the algorithm is simple and intuitive enough to be easily comprehended and traced by the students, therefore letting them debug the syntax. In the following, we briefly summarize the main aspects of the algorithm and of our implementation.

The central concept in Earley's algorithm is that of an *item*. An item is an object of the form $< i, j, X ::= \alpha.\beta >$, indicating an intended situation where:

(i) The parser is at position i on the input.

(ii) The production $X ::= \alpha\beta$ is being used to analyze an input fragment starting at position j.

(iii) An α structure has already been recognized, and the parser is waiting for

a β structure.

---

init: $< 1, 1, S' ::= .S, [\ ] > \in \mathcal{I}_G$

closure: $$\frac{<i,j,X::=\alpha.Y\beta,\tau> \in \mathcal{I}_G \ ;; \ Y::=\gamma \in \mathcal{P}_G}{<i,i,Y::=.\gamma,[\ ]> \in \mathcal{I}_G}$$

shift: $$\frac{<i,j,X::=\alpha.a\beta,\tau> \in \mathcal{I}_G \ ;; \ w_i=a}{<i+1,j,X::=\alpha a.\beta,append(\tau,[a])> \in \mathcal{I}_G}$$

reduce: $$\frac{<i,j,X::=\gamma.,\tau> \in \mathcal{I}_G \ ;; \ <j,k,Y::=\alpha.X\beta,\tau'> \in \mathcal{I}_G}{<i,k,Y::=\alpha X.\beta,append(\tau',[t(X,\tau)])> \in \mathcal{I}_G}$$

Fig. 7. Rules characterizing the set of Earley's items $\mathcal{I}_G$ for a context-free grammar G with a set of productions $\mathcal{P}_G$ and for a sentence w.

The rules in Fig. 7 characterize all the possible items for a context-free grammar and a sentence. In this characterization we have also enriched items with a fourth component to yield objects with the form $< i, j, X ::= \alpha.\beta, \tau >$. Here τ is the sequence of the parse trees corresponding to the parsed symbols α. Parse trees are represented as terms with the form `t(`*root*`,[`*Child*$_1$`, ..., `*Child*$_k$`])`.

The intended meanings of the rules are:

- The *init* rule establishes the parser initialization. In this rule S denotes the original grammar's axiom, while S' is the new axiom of the grammar expanded with a new production $S' ::= S$. Therefore, the item $< 1, 1, S' ::= .S, [\ ] >$ means that the parser is waiting to recognize the entire input according to the grammar given.
- When waiting for a non-terminal, the *closure* rule makes it possible to *activate* all the productions for this non-terminal.
- The *shift* rule allows the recognition of a terminal on the input.
- The *reduce* rule enables all the rules waiting for a non-terminal to advance when a production for this non-terminal has been finished.

Items of the form $< n + 1, 1, S' ::= S., [t] >$ represent complete parses of the input, with t the corresponding parse tree. Notice that items can be grouped by the input position to yield *parser lists*. For an input of length n, there will be $n + 1$ such lists. Earley's algorithm proceeds by:

- Initializing the first list by applying the *init* rule.
- Applying the *closure* and *reduce* rules to the items in each list until reaching

an equilibrium.

- Moving to the next list by applying the shift rule.

In addition, by fusing items with a common core (i.e. with the same three first elements) into a single one and by making smart use of *pointers* to kept track of the parse trees, it is possible to overcome the potential exponential complexity of a naïve implementation based on the rules of Fig. 7.[5] Also, we use lookahead information to achieve further improvements in efficiency.

4.2 Specializing the Parsing Kernel

As mentioned in the previous section, to produce a parser for a concrete grammar, a suitable description of this grammar must be added to the parsing kernel. This description includes:

- The grammar's axiom. This is indicated with an `axiom/1` predicate.
- A description of each production. This is indicated with a `prod/3` predicate. The first argument of `prod/3` is a unique identifier for the production. The second argument is the production head. The third argument is the sequence of symbols in the production's body. These symbols are encapsulated in a term with a `body` functor to make access to the arguments in constant time possible. The empty phrase λ is represented with a `body` constant.

The resulting parsers operate on lists of tokens. While it is possible to attach lexical attributes to these tokens, representing them as terms, only their functors are considered during shifting.

Example 4.1 Fig. 8a shows a representation of the underlying context-free grammar in Fig. 3 to be used with the parsing kernel. The combination of these facts and the parsing kernel yields the parser, as depicted in Fig. 8. This parser can be applied to sentences, as represented in Fig. 8b, in order to yield parse trees with the format illustrated in Fig. 8c.

4.3 Pattern for the Builders of Semantic Expressions

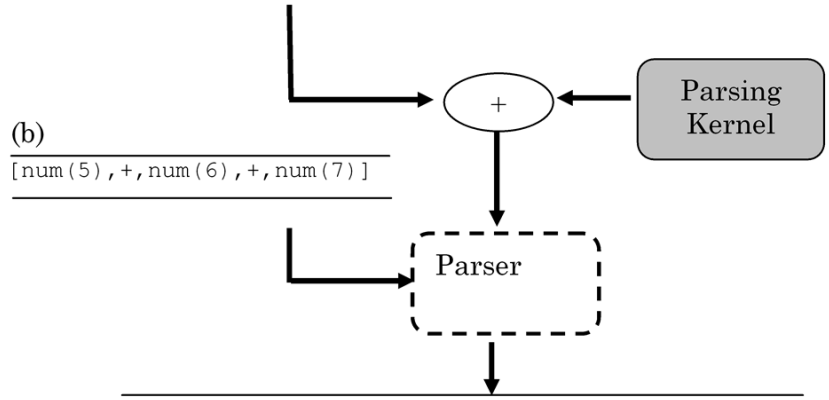
Builders of semantic expressions operate on the parse trees and take full advantage of the unification mechanism in logic programming to automatically solve the dependencies between attributes during a single top-down, left-to-right traversal. As mentioned before, they can be conceived as straightforward implementations of evaluators for non-circular attribute grammars where all the semantic functions have been interpreted as term constructors. These components are structured according to the following common pattern:

- Each non-terminal yields a predicate. The last argument for this predicate corresponds to the parse tree. The other arguments correspond to the

[5] Consider a grammar like `A ::=` λ`| aA | Aa` with a naïve implementation.

```
prod(prog0, prog, body(exp, constPart)).
prod(exp0, exp, body(exp, +, fact)).
prod(exp1, exp, body(fact)).
prod(fact0, fact, body(id)).
prod(fact1, fact, body(num)).
prod(constPart0, constPart, body(where, constDefs)).
prod(constPart1, constPart, body).
prod(constDefs0, constDefs, body(constDefs, (','), constDef)).
prod(constDefs1, constDefs, body(constDef)).
prod(constDef0, constDef, body(id, =, exp)).
axiom(prog).
```

(a)



(b)

```
[num(5),+,num(6),+,num(7)]
```

(c)

```
[t(prog, [t(exp,
        [t(exp,
     [t(exp, [t(fact, [num(5)])]),
      +,
      t(fact, [num(6)])]),
         +,
         t(fact, [num(7)])]),
           t(constPart, [])])]
```

Fig. 8. (a) Description of the underlying context-free grammar in Fig. 3 for the parsing kernel; (b) a sentence to be processed by the resulting parser; (c) list with the single parse tree associated with this sentence.

semantic attributes.

- The predicate for a non-terminal is defined with a clause for each grammar rule. The body is formed by following the right side of the production.
- Each non-terminal in the body is translated into an invocation to the corresponding predicate. In this invocation, the last argument is bound to the corresponding child in the parse tree. In addition, fresh variables are introduced for each semantic attribute.
- Each terminal is translated by binding the corresponding child to a suitable term, with a fresh variable for each lexical attribute.
- Each copy equation $a\ of\ s = a'\ of\ s'$ is translated as $X = Y$, where X is the variable for $a\ of\ s$, and Y that for $a'\ of\ s'$.
- Any other equation $a\ of\ s = t$ is translated as $X = \#(\_, t')$. X is the variable associated with $a\ of\ s$, and in t all the references to attributes are substituted by the corresponding variables to yield t'. Furthermore a fresh variable is associated with the resulting term. This variable is called

a backup variable, and it will be used to avoid redundant re-evaluations, as indicated in the next subsection.

The resulting builders traverse the parse trees in a depth-first, left-to-right way, binding the attribute variables to their possibly incomplete semantic expressions. Indeed, when a variable associated to an attribute with right-dependencies of other attributes is bound, variables associated with such attributes remain unbound until they are reached. Then unification will fill the holes for free. The pattern works for general non-circular attribute grammars.

```
prog(A, t(prog, [B, C])) :-
        exp(D, E, B),constPart(F, C),A=E,D=F.
exp(A, B, t(exp, [C, D, E])) :-
        exp(F, G, C),D= +,fact(H, I, E),B= #(J, G+I),F=A,H=A.
exp(A, B, t(exp, [C])) :-
        fact(D, E, C),B=E,D=A.
fact(A, B, t(fact, [C])) :-
        C=id(D),B= #(E, valueOf(D, A)).
fact(A, B, t(fact, [C])) :-
        C=num(D),B=D.
constPart(A, t(constPart, [B, C])) :-
        B=where,constDefs(D, C),A=D.
constPart(A, t(constPart, [])) :-
        A= #(B, emptyEnv).
constDefs(A, t(constDefs, [B, C, D])) :-
        constDefs(E, B),C= (',',),constDef(F, G, D),A=G,F=E.
constDefs(A, t(constDefs, [B])) :-
        constDef(C, D, B),A=D,C= #(E, emptyEnv).
constDef(A, B, t(constDef, [C, D, E])) :-
        C=id(F),D= (=),exp(G, H, E),
                  B= #(I, makeEnv(A, F, H)),G=A.
```

Fig. 9. The builder of semantic expressions for the grammar in Fig. 3 such as it is automatically generated in PAG.

Example 4.2 Fig. 9 shows the builder of semantic expressions generated by PAG from the specification in Fig. 3.

4.4 The Evaluation Kernel

The evaluation kernel, whose code is shown in Fig. 10, evaluates semantic expressions in an applicative order, with the exception of those affecting non strict functions (for them, the arguments are passed to the function without being evaluated, allowing the customization of any other suitable evaluation strategy). The kernel invokes the semantic functions when defined, or otherwise uses the functors as term constructors. The only tricky aspect of this process is the use of backup variables to avoid the reevaluation of terms duplicated in the expression. Indeed, when an expression of the form $\#(V, E)$ is

evaluated:

- If V is free, the expression E is actually evaluated and V is bound to the resulting value.
- If not, the backed up value is used instead.

```
eval(#(Val,Exp),Val) :-
  var(Val),!,
  eval(Exp,Val).
eval(#(Val,_),Val) :- !.
eval(Exp,Val) :-
  nonstrict(Exp),!,
  doResult(Exp,Val).
eval(Exp,Val) :-
  Exp =.. [F|Args],
  evalArgs(Args,VArgs),
  Funcall =.. [F|VArgs],
  doResult(Funcall,Val).
evalArgs([],[]).
evalArgs([Exp|Exps],[Val|Vals]) :-
  eval(Exp,Val),
  evalArgs(Exps,Vals).
doResult(Funcall,Val) :-
  defun(Funcall,Val),!.
doResult(Funcall,Funcall).
```

Fig. 10. The evaluation kernel.

```
#(A,
   #(B, valueOf(x,
     #(C,
        makeEnv(#(D, emptyEnv),
                      x, 5))))
     +
   #(E, valueOf(x,
     #(C,
        makeEnv(#(D, emptyEnv),
                      x, 5)))))
```

Fig. 11. A semantic expression with duplicated subexpressions.

Example 4.3 In Fig. 11 the semantic expression for the `val` attribute of `prog` and for the input `[ id(x), +, id(x), where, id(x), =, num(5) ]` is shown. Notice that in this expression the subexpression for looking up the value of `x` is duplicated. Nevertheless, each duplicated expression is only evaluated once, since all the duplicates share the same backup variable. Also

notice that the term shown in Fig. 11 is an externalization of the corresponding structure, which can be stored efficiently by sharing common substructures.

4.5 The Generator

The generator processes the attribute grammar specification to produce a context-free grammar description, a builder of semantic expressions, and a driver. The grammar description and the parsing kernel yield the parser, which is connected to the builder and the evaluator in the cited driver.

All the elements in the specifications, including the rules, are treated as Prolog facts by the generator. Indeed, the syntax of the specification language is easily embedded in Prolog by properly defining the `::=` and the `of` operators. The generator can be integrated with the Prolog system by using the static metaprogramming facilities found in many Prolog implementations. This lets students directly load PAG specifications into the Prolog engine.

5 Conclusions and Future Work

In this paper we have presented PAG, a framework for the rapid prototyping of language processors in Prolog. This framework is oriented to supporting the learning process of students enrolled in an introductory course in language processors by letting them test their specifications. The framework is able to deal with general non-circular attribute grammars on arbitrary (maybe ambiguous) context-free syntax in a comprehensible way, which is a primary requirement in the application context mentioned. To deal with arbitrary syntax, Earley's parsing algorithm is used. General non-circular attribute grammars are managed by first interpreting semantic functions as term constructors. The semantic expressions yielded are then definitively evaluated considering the actual definitions for the semantic functions. The overhead incurred by the cited separation of concerns is acceptable in a prototyping context, where the simplicity and comprehensibility of the techniques for the average student come before considerations of efficiency.

Currently we are extending PAG with simple modularization facilities based in the composition of *semantic aspects*, as suggested in [13]. This is in accordance with our pedagogical method, since we introduce different *views* of the complete attribute grammar: one for the construction of the symbol table, another for checking the contextual constraints on the source language, and a third one for dealing with the translation concerns. We are also considering the extension of PAG to deal with circular attribute grammars. This extension is oriented to our students of a Ph.D. course on e-learning, where we promote the use of language processor technologies in the processing of the markup languages proposed by the different e-learning specifications (see, for instance, [16]). The basic idea is to work with circular Prolog terms in managing circular definitions. With this we hope to provide students, who

belongs to several disciplines, with a less knowledge-demanding alternative than the one based on fixpoint computations [3]. We are also planning to include domain-specific visual tracing capabilities in the system. As future work we want to use PAG in an introductory course on computational linguistics. We also want to take advantage of the modularity of the approach to build a complete learning scenario based on the learning object paradigm and supported by the web-based e-learning systems deployed at our university.

References

[1] Abramson, H. Dahl, V, "Logic Grammar", Springer, 1989.

[2] Aho, A. Sethi, R. Ullman, J. D, "Compilers: Principles, Techniques and Tools", Adisson-Wesley, 1986.

[3] Arbab, B, *Compiling Circular Attribute Grammars into Prolog*, IBM Journal of Research and Development **30**(3) (1986), 294–309.

[4] Carpenter, B. Penn, G, "The Attribute Logic Engine User's Guide Version 3.2.1", University of Toronto, 2001.

[5] Clocksin, W. F. Mellish, C. S, "Programming in Prolog", Springer, 1987.

[6] Earley, J, *An Efficient Context-free Parsing Algorithm*, Communications of the ACM **13**(2) (1970), 94–102.

[7] "Focus on the Structure of Higher Education in Europe 2004/05. National Trends in the Bologna Process", Eurydice, 2005.

[8] Fisher, C. N. LeBlanc, R. J. J, "Crafting a Compiler", The Benjamin/Cummings Publishing Company, 1988.

[9] 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 **35** (1992), 121–131.

[10] Grosch, J. Emmelmann, H, "A Tool Box for Compiler Construction", CoCoLab White Paper, 1990.

[11] Jansen, P. Augusteijn, L. Munk, H, "An Introduction to Elegant. Second Edition", Philips Research Laboratories, 1993.

[12] 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, 1991.

[13] Kastens, U, *Attribute Grammars as an Specification Method*, in Ablas, H. Melichar, B. (eds), "Attribute Grammars, Applications and Systems", Lecture Notes in Computer Science **545**, Springer, 1991.

[14] Klint, P. Lämmel, R. Verhoef, C, *Toward an Engineering Discipline for Grammarware*, ACM Transactions on Software Engineering and Methodology **14**(3) (2005), 331–380.

[15] Knuth, D. E, *Semantics of Context-free Languages*, Mathematical Systems Theory **2**(2) (1968), 127–145. See also the correction published in Mathermatical System Theory **5** (1) (1971), 95–96.

[16] Koper, R. Tatersall, C (eds.), " Learning Design, a Handbook on Modelling and Delivering Networked Education and Training", Springer, 2005.

[17] Mernik, M. Lenič, M. Avdičaušević E. Žumer, V, *Compiler/Interpreter Generator System LISA*, Proc. of the 33rd Hawaii International Conference on Systems Science, 2000.

[18] Mernik, M. Žumer, V, *An Educational Tool for Teaching Compiler Construction*, IEEE Transactions on Education **46**(1) (2003), 61–68.

[19] Paakki, J, *A Logic Based Modification of Attribute Grammars for Practical Compiler Writing*, Proc. of the 7h International Conference on Logic Programming, 1990.

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

[21] Polsani, P. R, *Use and Abuse of Reusable Learning Objects*, Journal of Digital Information, **3**(4), 2003.

[22] Rehberg, S. Ferguson, D. McQuillan, J. Eneman, S. Stanton, L, "The Ultimate WebCT Handbook, a Practical and Pedagogical Guide to WebCT 4.x", Ultimate Handbooks, 2004

[23] Sierra, J. L. Fernández-Valmayor, A. Guinea, M. Hernanz, H. Navarro, A, *Building Repositories of Learning Objects in Specialized Domains: The Chasqui Approach*, Proc. of the 5th IEEE International Conference on Advanced Learning Technologies ICALT05, 2005.

[24] Sterling, L. Shapiro, E, "The Art of Prolog", The MIT Press, 1994.

[25] Šveda, M. Jankovský,M, *Prototyping with Attribute Grammars and Prolog*, Proc. of the 23rd EUROMICRO Conference'97 New Frontiers of Information Technology, 1997.

[26] Warren, D. S, "Programming in Tabled Prolog (Draft)", 1999, Available online at http://www.cs.sunysb.edu/∼warren/xsbbook/book.html (last visited on Febraury 9, 2006).