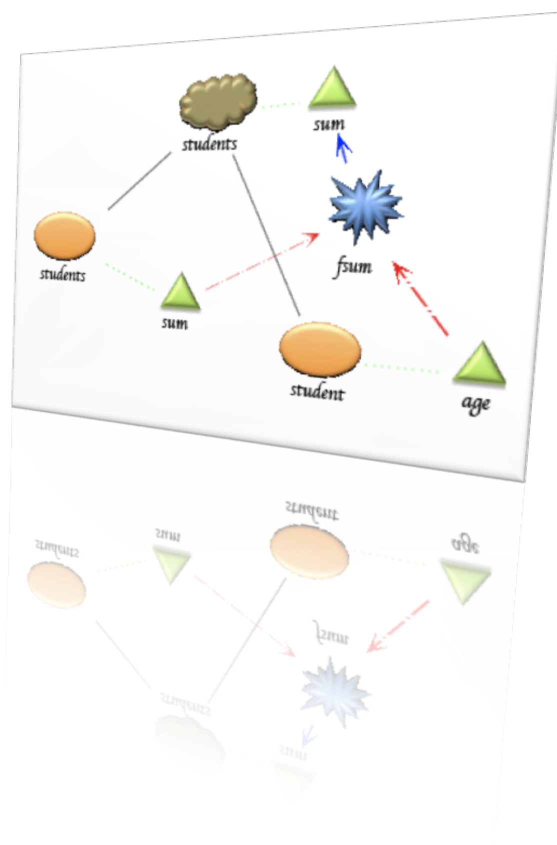
University of Minho, Informatics Department

Master in Informatics

Visual$\mathcal{LISA}$

A Visual Programming Environment for Attribute Grammars



Nuno Oliveira

Supervised by:

Pedro Rangel Henriques
Maria João Varanda Pereira
Daniela da Cruz

Braga, February 26, 2009

Abstract

LISA (Language Implementation System based on Attribute grammars) is a compiler generator tool based on textual attribute grammars.

This document reports the work concerned with the development of an environment for visual attribute grammars, named `Visual`*LISA*. The main purpose of this environment is to be used as a front-end for *LISA* tool, in order to ease and enrich the way language engineers design their attribute grammars.

This environment is generated from a specification of a visual language that ensures the possibility to draw, syntactically and semantically correct, attribute grammars, in an integrated editor. The visual specification of the attribute grammar is production-oriented and incremental. Semantic rules are drawn, together or separately, over the syntactic layout of the (respective) production. Attribute declarations are collected and gathered from tree node declarations. Moreover, the editor translates the drawn attribute grammar directly into *LISA* notation (generating *LISA* textual specifications) or alternatively into a universal `XML` representation designed to support different AG-based generators.

Special focus will be devoted to specification of visual languages and consequent automatic generation of a dedicated graphical editor integrated with a compiler. The tool generator `DEViL` will be introduced and its use explained.

Keywords: Attribute Grammars, *LISA*, Compiler Generator, Visual Programming Environment Generation, Visual Languages, Code Generation, Intermediate Representation, `DEViL`, `Visual`*LISA*.

Contents

1 Introduction **4**
1.1 Visual Language Definition . . . 6
1.2 Attribute Grammars . . . 7
1.3 Outline . . . 9

2 A Visual Language for Attribute Grammar specifications in LISA **10**
2.1 Problem Statement . . . 10
2.2 Objectives . . . 12
2.2.1 The Environment and its Language - Disambiguation . . . 13

3 Programming Environment Generators **14**
3.1 DEViL . . . 16
3.2 VLDesk . . . 17
3.3 TIGER Project . . . 18
3.4 Other Tested Not Tools . . . 18
3.4.1 $AToM^3$ . . . 18
3.4.2 MetaEdit+ . . . 18
3.4.3 DiaGen . . . 18
3.4.4 SIL-Icon . . . 19
3.4.5 VLG . . . 19
3.4.6 VPW . . . 19
3.4.7 Spargen . . . 19
3.4.8 CoCoViLa . . . 20
3.5 Why DEViL . . . 20

4 VisualLISA: The Syntax **21**
4.1 `PLG`: A Formalism to Specify Visual Languages . . . 21
4.2 Visual Language Requirements . . . 23
4.3 The Abstract Syntax . . . 25
4.4 The Visual Syntax . . . 28

5 VisualLISA: The Semantics **32**
5.1 Attributes Definition . . . 32
5.2 Contextual Conditions . . . 35

6 VisualLISA: The Translation **40**
6.1 LISA . . . 41
6.1.1 The Compiler Generator . . . 41
6.1.2 LISA's AG Specification . . . 42
6.2 Universality for Attribute Grammars . . . 45

6.2.1 XAGra - An XML dialect for Attribute Grammars 46

7 DEViL Implementation 53
7.1 Syntax Grammar 53
7.1.1 The Visual Language Specification 55
7.1.2 Replicating Structures 58
7.2 Environment Generation 60
7.2.1 The Interface Generation 61
7.2.2 Dock Buttons 64
7.2.3 Predefined Structures 66
7.3 Semantics Implementation 68
7.3.1 Semantics Implementation: Preview 68
7.3.2 Semantics Implementation: The Concretization 70
7.4 Tree-Grammar Traversing for Code Generation 73
7.4.1 Structuring the Output with Templates 73
7.4.2 The Problem of Being Visual 74
7.4.3 The Code Generation Process 77
7.5 Implementation Summary 80

8 User's Guide 81
8.1 Declaration and Edition of a Production 83
8.2 Start Symbol Definition 85
8.3 Defining the `LHS` 85
8.4 Defining the `RHS` and Associate Attributes 86
8.4.1 Insert a *NonTerminal* 86
8.4.2 Insert a Terminal 87
8.4.3 Insert an Inherited Attribute 88
8.4.4 Insert a Synthesized Attribute 90
8.4.5 Insert an *IntrinsicValueAttribute* 91
8.4.6 Reuse Attribute Attachments 93
8.4.7 Ordering the `RHS` symbols 93
8.4.8 Removing a Symbol in a Production 93
8.5 Specification of a Computation Rule 94
8.5.1 Declare and Edit a Computation Rule 94
8.5.2 The Identity Function 96
8.5.3 Specify a Function/Operation 96
8.5.4 Editing a Function/Operation 98
8.5.5 Initializing an *InhAttribute* 99
8.5.6 Printing the value of an Attribute 101
8.6 Removing a Production or a Computation Rule 102
8.7 Specification of Global Functions and Other Definitions 103
8.8 Perform Semantic Verification 106
8.9 Generate Code 107

9 Conclusion 109
9.1 Future Work 110
9.2 Further Usage Perspectives 110

A $\mathcal{X}$AGra Schema Definition 114

B Language Specification in DEViL **118**

C Code Generated For $\mathcal{X}$AGra- Example **124**

D Students Grammar Solution **126**

Chapter 1

Introduction

The crescent demand for programs with a stunning but easy-to-use visual interface has reached a level from where it is impossible to come back. Hence in the past few years, many programming editors have been improved with environments to visually create these interfaces in an easy way.

Some of these environments are fully created from the scratch, resorting to code blocks (sometimes) very hard to maintain and evolve. From that approach, result usually good and efficient visual programming environments, but there is no systematization on their development, and development costs are high. In addition, the visual language definition concept, always behind these environments, is lost.

In fact, any time that there is a language definition behind the hood (despite being visual rather then textual), the traditional language specification and processing techniques can be used to systematically develop the environment.

According to [ASU86] and many others, any language processing task is twofold:

- Analysis: to discover the meaning of the source program, i.e. its structure, and the precise contents;
- Synthesis: to create the output (for example, a target program) related with the source program.

Figure 1.1 shows the chain of tasks that pertain to these two phases.

The source program is converted into an abstract representation which embodies the essential properties of the source language. This abstract representation may be implemented in many ways, but it is usually conceptualized as a tree. The structure of the tree (nodes labelled by grammar symbols and productions, hierarchically linked to other nodes) represents the control and data flow aspects of the program; additional information (attributes) is attached to the nodes to describe other important values for the translation process.

The analysis phase of a language processor can be divided into three tasks: lexical, syntactical and semantical. The lexical analysis scans the source text and identifies terminal symbols; the syntactical analysis matches the input stream of terminal symbols against the grammar to identify the productions used and rebuild the derivation (syntactic) tree; the semantic analysis decorates tree nodes, computing the value of the attributes describing properties of the node, and additionally checks their consistency resorting to the symbol table, also known as identifier table, which is defined in this first phase, and consulted in both. Notice that the information collected in the attributes of a node is derived from the

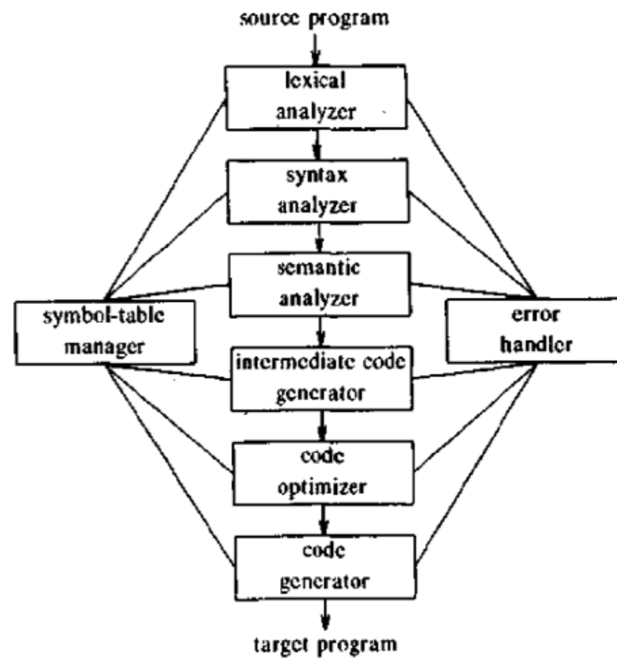


Figure 1.1: Common Compiler Phases

environment of that node; this allows to deal with *context sensitive information* based on a *context free grammar*.

Attribute Grammars (AG) have proven to be an useful aid in specifying the construction and decoration processes of the structure tree, because they constitute a formal definition of all syntactic (context-free) and semantics (context-dependent) language properties. Moreover, decorated tree transformation can also be specified through an AG.

So, to build a semantic-directed language processor in a systematic way, all the analyzers and the translator should be developed according to the attribute grammar. The semantic module can systematically derive from the AG, as it is possible for the lexical analyzer, and the parser. This enables the construction of tools that generate the semantic-directed language processor automatically from the AG. The main difficulty is concerned with the detection of cycles in the attribute definitions and the efficiency of the evaluation process.

AGs are the most used artifact to create language processors. But the different syntax notations of the meta-language imposed by some compiler-compilers and other difficulties associated to the AGs concept, make the developers avoid the usage of AGs and go through non systematic ways to achieve the same results.

Independent of the structural approach (monolithic, or modular), the meta-language chosen to write the AG is an important concern. The notation used to write the AG can be more abstract, or more concrete, depending on the purpose: if the interest is to have the AG as a specification document, it can be used a notation close to the mathematical formalism, as simple, abstract and generic as possible; otherwise, if it is intended to build a language processor based on the attribute grammar, it is needed a more concrete notation, not so far from the implementation language and strategies. This fact lead, some years ago, to purpose a visual language as the meta-language to write AGs.

The work reported in this document consist in the development of a visual programming

environment to work as a front-end for $\mathcal{LISA}$ [MLAv02, MKv95, MLAZ98].

$\mathcal{LISA}$[1] is a textual attribute grammar based compiler generator with a specific meta-language to define the AG.

With the development of such editor, is objective to ease the process of designing an AG, giving the user the opportunity to visually specify (using iconic symbols) both the grammar syntactic and semantic rules. And hence generate $\mathcal{LISA}$'s specification code from the drawings, in order to turn invisible the textual definition of an AG.

As can be seen, the development of this work addresses two important and well known subjects on the Language Processing area: Visual Languages and Attribute Grammars.

The following sections will briefly introduce important aspects on AGs. These aspects are necessary to understand the remainder of the document. Besides that, a simple definition about visual languages is exposed.

1.1 Visual Language Definition

Visual Languages (VL) are not easy to define, because there is not a consensual definition. The notion of VL is deeply connected with the notion of Visual Programming Language (VPL); by this reason, sometimes they are viewed as the same topic.

VLs or VPLs aim at offering the possibility of solving some complex problems by describing their properties or their behavior through to graphical/iconic definitions [dSD96]. Normally, these graphical definitions are (simple or complex) compositions of icons, which are intended to describe visually the flow and the transformations of data. The icons or figures are composed in a space with two or more dimensions, defining sentences that are formally accepted by parsers, where shape, color and relative position of the icons are relevant issues.

Thus, a good definition for VPL can be found in [Roc95]:

> *A Visual Programming Language defines a set of sentences formed by the spatial disposition of graphical objects with a very well defined semantics.*

The main difference, on a naif reckoning, between a VPL and the VL is indubitably that the first is a more concrete topic with a concrete application on the programming area. On the other hand, the latter is more abstract and includes several areas of computing (but not only). A relation between both of these subjects can be defined as:

$$\texttt{VPL} \subset \texttt{VL}$$

There are many types of VL, and they are neither equal nor fit in a same set of languages. Examples cover a large range from Musical Scores, Traffic Signals, Modeling Languages (DER, Class-Diagrams, Use-Cases Diagrams, State-Machines) until programming environments like graph transformations, DTS, Grafcet for digital equipment control and robotics Prograph, etc.

The literature about visual languages is also very large and distinct addressing several areas of VL. Regarding these differences, Margaret Burnett and Marla Baker developed a Classification System for Visual Languages [BB94] to classify the various articles and other documents, projects and prototypes.

[1] A more detailed presentation and study of this compiler-generator will be presented later in this document.

1.2 Attribute Grammars

A Context-free grammar (`CFG`) is formally defined by the following tuple:

$$G = (T, N, S, P)$$

where:
T is the set of terminal symbols that define the alphabet for the language;
N is the set of nonterminal symbols;
$S \in N$ is the start symbol of the grammar and
P is a set of productions.

A production, also known as derivation rule, is composed by a left-hand side (`LHS`), with `LHS` $\in N$; and by a right-hand side (`RHS`), with `RHS` $\subseteq N \cup T$.

A CFG can be visualized as a tree, where each production defines a level of it, that is, a sub-tree. In fact, the `LHS` of the production is the root of the sub-tree, and the `RHS` compounds the children knots of that tree. When a child of the sub-tree corresponds to a nonterminal symbol, then it induces the continuation of the overall tree, because this symbol must be the root of some other production. On the other hand when the child is a terminal symbol, then it corresponds to a leaf of that tree.

An Attribute Grammar (`AG`) is based on a CFG. It associates: a set, $A(X)$, of attributes with each symbol X in the vocabulary (V) of G (terminal (T) and non-terminal (N) symbols); a set, $R(p)$, of evaluation rules with each production $p \in P$; and a set, $C(p)$, of contextual conditions with each production $p \in P$.

So an attribute grammar is formally defined as the following tuple:

$$AG = (G, A, R, C, \tau)$$

where
$A = \bigcup_{X \in (N \cup T)} A(X)$ is the set of all the attributes;
$R = \bigcup_{p \in P} R(p)$ is the set of evaluation rules for all the productions;
$C = \bigcup_{p \in P} C(p)$ is the set of contextual conditions for all the productions and
$\tau = \bigcup_{p \in P} \tau(p)$ is the set of translation rules for all the productions.

Since an `AG` is based on a CFG, then it can also be seen as a tree. As Figure 1.2 depicts, the tree is now decorated with attributes associated to its knots, what reminds a tree with flowers. These flowers (the attributes) must turn into fruits by associating values to them.

Each attribute has a type, and represents a specific property of symbol X; we write $X.a$ to indicate that attribute a is an element of $A(X)$. For each $X \in (N \cup T)$, the set of attributes of X is splitted into two disjoint sets: $A(X) = Inh(X) \cup Syn(X)$, respectively the *inherited* and the *synthesized* attributes.

Each $R(p)$ is a set of formulas

$$X.a = func(..., Y.b, ...)$$

that define how to compute, in the precise context of production p, the value of each attribute a as a function of the value of other attributes b, where each defined attribute a should be a synthesized attribute associated with the nonterminal in the lefthand side or an inherited attribute associated with a nonterminal in the righthand side:

$$a \in (Syn(X_0) \cup Inh(X_i)), i \geq 1$$

and each used attribute b should be an inherited attribute associated with the nonterminal in the lefthand side or a synthesized attribute associated with a symbol in the righthand side:

$$b \in (Inh(X_0) \cup Syn(X_i)), i \geq 1$$

In fact, attributes like a are called *Out* attributes of a production $p \in P$, and used attributes, like b, are called *In* attributes of the same production.

As the name suggests, an *In* attribute is an attribute that brings a value computed in another production to the actual production p and the *Out* attribute is the opposite, i.e., it export a value from the actual production to another one. Figure 1.2 depicts a production highlighting the *in* and *out* attributes. *in* attributes are represented with a yellow line surrounding the triangle and the *out* attributes with a black one.

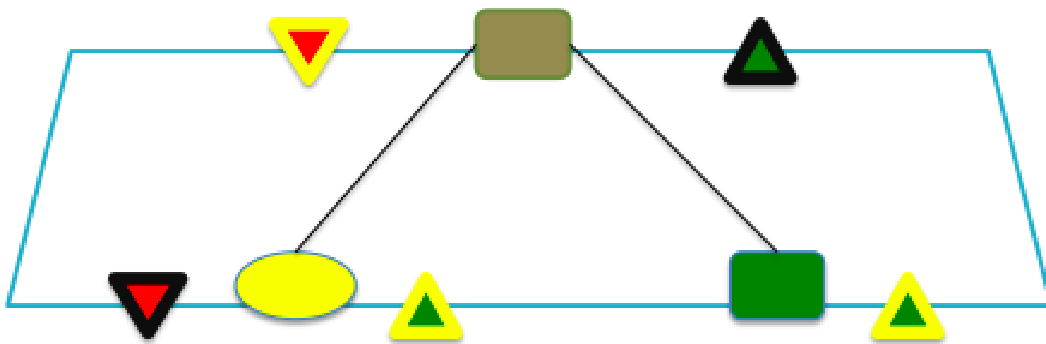


Figure 1.2: *In* and *Out* attributes of a production

Each $C(p)$ is a set of predicates

$$pred(..., X.a, ...)$$

describing the requirements that must be satisfied in the precise context of production p. Each predicate, checked for the actual value of the argument attributes (any synthesized or inherited attribute that occurs in that context can be an argument), must hold a `true` value, so that the production is meaningful (is valid from a semantic point of view).

Each $\tau(p)$ is a set of procedures

$$proc(..., X.a, X.b, ...)$$

that use the value of attributes available in the context of production p (preferably the synthesized attributes of production, but not restricted to) to produce, or generate, the output of the language processor.

Many times, and many authors, do not consider $C(p)$ and $\tau(p)$ separate from $R(p)$ — they consider an `AG` as a triplet

$$AG = (G, A, R)$$

In these cases, contextual conditions and translation rules are defined as boolean functions that associate a truth value with special boolean attributes and produce the desired action (contextual constraint verification or output building) as a side effect.

In summary, AGs are a formal and practical way to develop any kind of programming language. The possibility to use attributes to store and spread information through the processing phase, makes easier the derivation of all modules needed to compile a language, and hence, it is faster to get the desired output.

1.3 Outline

The reminder of this report will be structured as following: in Chapter 2 we present the problem, stating all the requirements the tool must hold. In Chapter 3 we describe a small project on visual languages. This simple project is used to present, review and test some tools used to generate visual language environments; yet in this chapter it is chosen one of these tools to base the work on, and are pointed some reasons to fundament this choice.

In Chapter 4 a formal definition of the meta-visual-grammar syntax is presented, along with the pictures that compound the concrete grammar drawings.

In Chapter 5 the semantic constraints for the visual language is defined both naturally and formally.

Chapter 6 presents $\mathcal{LISA}$ system. Then the structure of a $\mathcal{LISA}$ specification is analyzed to understand the notation to generate code. After this, we design and propose an XML dialect to support the intermediate representation of an attribute grammar.

Chapter 7 addresses the implementation steps using the chosen tool to automatically generate the visual programming environment.

Chapter 8 presents the developed tool, describing its features, and state the basics to handle it correctly.

Finally, on Chapter 9 we conclude the work briefing about what was done, and what else can be done in a future work.

Chapter 2

A Visual Language for Attribute Grammar specifications in LISA

The development of the visual editor here reported is based on the early general idea expressed in [PMdCH08]. Now the aim of this work is to realize the idea, what implies a more accurate definition and its implementation.

In this chapter, the problem underlying the work already introduced is defined, devoting attention to the objectives to attain when the work is finished.

2.1 Problem Statement

AG definitions are not as easy as people would desire. The difficulties of choosing the appropriate attributes and conceiving the attribute evaluation rules are enormous.

Normally, experienced language engineers sketch up on paper the complex dependencies of symbols, attributes and functions, in order to map the logical image (present in their minds) into a concrete drawing.

Moreover, teachers also use drawings of that dependencies aiming at helping the students to understand how the computations of attributes are done. This strategy simply alleviate the mental effort of beginners (someone looking to AGs for the first time) and in addition frees one's mind from the syntax awareness, that is, the developer or student can think abstract and syntax-independent.

In any case, there is not a standard method of creating such hasty drawings. Person A can think better on the semantic rules if the production is drawn in a unique line, but person B may find it better to draw the production in a tree format, or person C, exaggerating, is accustomed to think about productions with a not coherent format. With this work, it is intended to maintain the AG specification task the way it is, in order to *please both Greeks and Trojans* and not to impose a rigid visual notation — moreover a rigid notation is antagonistic to visual programming.

However, after being sketched the productions and the semantic dependencies between the attributes, they are not more than gibberish on paper. The person who draw it must explicitly go through a second effort correspondent to the translation of the (sometimes imperceptible) pencil strokes into the concrete syntax recognized by the compiler generator. Regarding the literature, a tool where both steps (sketch and translation) could be done together is missing; that tool also comply with strict efficiency requirements.

$\mathcal{LISA}$ as a textual AG-based compiler generator, pushes its users into the same problems and difficulties raised before.

It is intended to develop a new visual language that assures the possibility of specifying visually AGs for $\mathcal{LISA}$(in specific), and to generate a suitable programming environment, called Visual$\mathcal{LISA}$. The main objective is to diminish the difficulties presented for AGs-specification.

At a visual language design-time the language engineer must have in mind several aspects in order to create the visual language according with some rules, directives or, at least, cognitive dimensions [YB97, YDZ96]. It must be assured that a visual language has expressiveness, is intuitive and therefore easy to learn and to use, it avoids the ambiguity and syntactic or semantic errors.

A visual language implies the existence of a visual programming environment [KS02, CTODL95], because its absence makes the language useless. Commonly, a visual programming environment consists in a graphical front-end, that is, an editor, incremented by several tools for analyzing, processing and transforming the drawings resultant from the association of terminal symbols (characters of the iconic alphabet) of the visual language.

The definitions of a visual language and the automatic and systematic generation of a suitable programming environment are the foundations for this work.

The programming environment overview is depicted in Figure 2.1. The architecture and the main functionalities are exposed in Section 2.2.

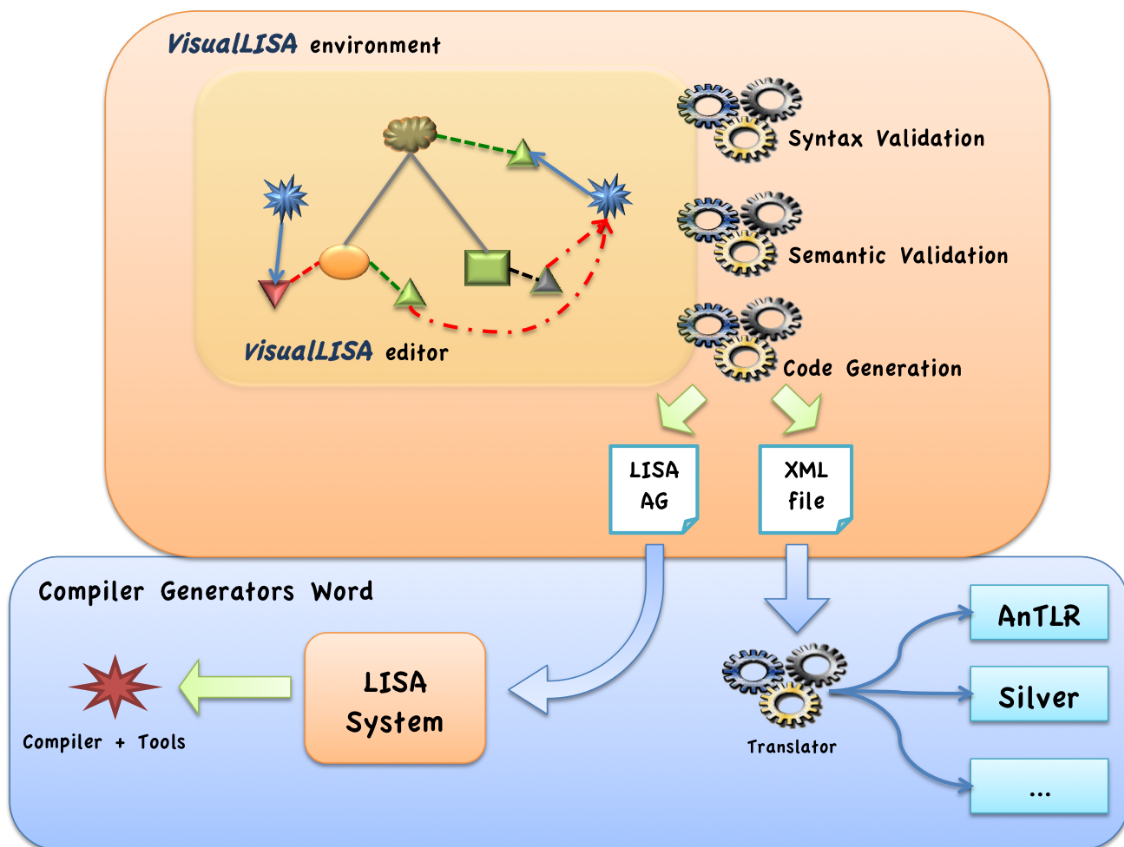


Figure 2.1: Architecture of Visual$\mathcal{LISA}$

As can be noticed on Figure 2.1 the original statement was incremented with a new challenge: to generate not only $\mathcal{LISA}$ meta-language, but also a generic representation that could rise up Visual$\mathcal{LISA}$ to a true *VisualAG* environment.

2.2 Objectives

Last section presented the problem that bases this work. It was also defined, in a broad perspective, the main objectives to attain. In the present section, the architecture sketched in Figure 2.1 will be explained and the concrete objectives (including the main functionalities) will be discussed.

Visual$\mathcal{LISA}$ has a simple architecture when compared with other systems. In fact, it is nothing more than a front-end for an AG-based compiler generator, so its architecture complements the architecture of the associated compiler generator.

In Figure 2.1 it is possible to see that the component on the top, labeled as Visual$\mathcal{LISA}$ environment, is composed by the visual editor, where the new visual language terminals — the basic images — combine each other spatially, to create the shape of the visual AG specification.

The environment proposed also offers tools or mechanisms to validate syntactically and semantically the drawings sketched up on the editor.

The syntax validation restricts some spatial combinations of the visual language terminals. This validation is a task to be performed at edit-time, originating a syntax-directed editor.

The semantic validation deals with syntactic restrictions that can not be expressed by the productions, and covers a set of constraints concerning the AG definitions.

Besides that, Visual$\mathcal{LISA}$ will generate code from the drawings. As told before, the target code will be $\mathcal{LISA}$ or XML.

The $\mathcal{LISA}$ specification generated can therefore be passed straightforward to $\mathcal{LISA}$ system, and finally be used to create the complier for the language defined with Visual$\mathcal{LISA}$. With this approach, the programming environment emulates the two-steps behavior of the language designer referred before: *i*) sketch up the attribute dependencies w.r.t the semantic rules and productions, and *ii*) mental effort to translate the hasty drawings into $\mathcal{LISA}$ notation; into a single step effort.

The use of XML as an output from the code generation mechanism gives the system more versatility because it allows a functional separation between the visual editor and the compiler generator tool. Besides that separation of concerns, it makes Visual$\mathcal{LISA}$ to become a *VisualAG* – visual programming environment for AGs independently of the target compiler generator. However, this only happens if two requirements were fulfilled:

1. The XML dialect is sufficient universal and abstract to accept any AG definition and
2. There is a translator which actuates as a midpoint between the XML file and the target compiler-compiler. Its obvious function is to transform the XML file content into a concrete syntax compatible with the syntax of the AG definition of the target compiler-compiler.

The first requirement must be assured, but the second one is out of the scope of the present work. The latter requirement must be faced as a righteous and important task for further works.

The XML translator must be developed, concerning the fact that the target compiler generators must be AG-based like $\mathcal{LISA}$, AntLR [PQ95], Silver [VWBGK08], JastAdd [EH07] and so forth. Despite of not being an AG-based compiler generator, the system Lex/Yacc is also a possible target for such translator, since exists the Ox system [Bis92] which resorts to an AG notation to create a Lex/Yacc specification.

The definition of the this visual language and creation of the associated programming environment must follow the traditional approach to compilers development, that is, the generation of the Visual$\mathcal{LISA}$ environment must be systematic. When the language to design is textual (not visual) the systematization of the process of creating language processors (compilers or parsers), is the usage of context-free or attribute grammars to feed compiler generators. For visual languages, the same systematization is already possible, and is to be used in this project. To accomplish this, a choice between several programming environment generators has to be done.

In Chapter 3 a simple visual language is used to briefly introduce, test and review some of the actual programming environment generators. After the experiments one of them is chosen and, therefore, used to systematically create the visual language processor.

2.2.1 The Environment and its Language - Disambiguation

Through out this document, references to the environment and to its underlying language will be used. From here on, in order to clarify them, the following words will be used:

Visual$\mathcal{LISA}$ to refer to the visual environment and

V$\mathcal{LISA}$ to identify the visual language underlying the environment.

Chapter 3

Programming Environment Generators

In the current chapter some visual programming environment generators are presented and experimented. The objective of these experiences is to choose, from all the tools, the most versatile which fulfills the project's necessities, in order to use it and systematically developVisual$\mathcal{LISA}$. The start point for the systematic development process should be the attribute grammar for the visual language desired (V$\mathcal{LISA}$ in that case).

In a fair political race, where the final objective is to elect the candidate that best serves a people, all the participants are submitted to equal conditions. They participate in the same political debates, answer to questions about the same social topic using their best knowledge and skills to please the people who must judge all of them and elect the best.

In the case presented along this chapter, there are no political candidates, but environment generator tools. For a fair decision about what tool is the best to serve the interests of this project, equal conditions of benchmarking must be prepared. With the purpose of preparing such conditions, a small project, comparable to Visual$\mathcal{LISA}$'s, was defined and is shortly described below.

Topic-Maps: A Benchmark Example A Topic-Map (TM) represents a base of knowledge about some subject. It captures the information about a topic and relates this information in a graph style. Additionally, concrete occurrences of a topic are connected to the respective topics; this creates two graph representations in a unique model.

XML is preferred to describe TMs, inasmuch as there are an ISO13250 standard dialect XTM to represent that information structure. However other easier languages, with associated tools to create navigable web-based graphs, exist and can be used to represent such information. The tool *Conceptual Map Compiler* (CMC), developed at University of Minho[1], is an example.

However, writing effective XTM or other TM-based language specifications is hard, because of the possible existence of enormous relations between the topics and the several concrete occurrences on them. So that it was decided to describe a visual language to be possible the drawing of such *bi-graph*.

The requirements that should be held by the generators are defined below. In fact, these proofs are the objectives for the simple TM project:

- ⋆ A visual language to represent the syntax of a TM should be defined;

[1] http://epl.di.uminho.pt/~gepl/CMC/

- ⋆ The terminals of the visual language should be: *i*) **topic** — simple circle where the name of the topic should figure; *ii*) **occurrence** — simple rectangle where the name of the occurrence should figure; *iii*) **topic-topic relation** — a simple connector where a label to express the type of relation should be present and *iv*) **topic-occurrence relation** — a simple connector without any label;
- ⋆ A visual programming environment should be generated;
- ⋆ Syntactic check should be effective and for best results the syntactic errors must be avoided on edition-time;
- ⋆ The drawings should be translated into `CMC` code and
- ⋆ Specifications should be possible to save and reload.

The syntax of the visual language must constraint the connections between topics and occurrences. A topic-topic relation should only connect two topics, and a topic-occurrence relation should only connect topics to occurrences.

As this project is too simple and small, the development of a semantic check module is set apart.

Besides what must be produced at the end, an important characteristic to compare these tools is the easiness of maintaining or evolving the original specification.

To assess this criterion, the first project statement, above described was incremented with a new requirement that implies the adaptation of the `TM` visual language specification. Changing symbol names and syntactic rules but keeping the same images, must be possible, in an easy way, to completely alter the `TM` specification in order to specify a visual language for `DRAW`.

`DRAW` is modeling language aiming at providing a simple way to develop websites. In a `DRAW` specification, pages are connected to each other either by a link labeled with the name of the link or by an operation/function that requires some information.

It is not difficult to identify the terminals on this new visual language and to relate them with the `TM` visual language: a **page** corresponds to a topic; a **link** corresponds to topic-topic relation, but with some differences, because it can connect pages to pages and pages to operations; **operation** corresponds to an occurrence.

Of course the purpose of the visual language `DRAW` is different from the one described before (`TM`). According to this, the code generation must also be different. However, the principal objective is to test the easiness of altering or maintaining a specification on the notation of the visual language environment generator. Thus, the code generation is not an objective for this second visual language.

Conditions are prepared to begin the experiments! Despite of being found many tools in the literature, the major part is not available. So that, not all the tools presented here were experimented. In the following sections, the tools that were found and experimented will be presented and shown some of their characteristics regarding the experiment requirements. Pros and cons that come from the experiment will be stated.

The not tested tools will be briefly described. Some aspects of these tools that make them useful or not for Visual$\mathcal{LISA}$ development will be emphasized.

In the last section of this chapter, it is picked one of the tools, and the reasons for that selection will be explained, regarding the other tools.

3.1 DEViL

DEViL [SKC06] is a visual programming environment based on an Object-Oriented-AG specification.

The visual language is specified by means of an attribute grammar that, in fact, is really similar to the classes of an Object-Oriented language. This eases the process of creating the abstract structure of the grammar, because one can rely on class diagrams. With this higher level vision of the language is easier to define syntactic constraints.

The specification of the language icons (the terminal symbols) is simple because an icon designer tool makes part of the DEViL system. In order to apply the drawn icons to the language symbols, one should use visual patterns. These patterns are already defined in the DEViL system and define the layout of the language.

The default visual programming environment generated by DEViL is very complete and easy to use. Features like undo and redo, save and load, export the models drawn into images, etc. are some examples. However it can be extended with new user-defined functionalities, or some default ones can be removed. The interaction with the visual language is nice and syntactic errors are avoided at edition time.

DEViL supports the translation of the drawing into a textual notation. This translation is based on the well-known AG approach to do the same task.

With DEViL, there are a very well defined separation of concerns at the development level. It is needed many files to create a new visual language. But the specifications inside each file are short and easy to maintain. So that using DEViL to specify the TM language and transform it into DRAW was a simple task.

In Figure 3.1 is shown an image of a part of the editor generated by DEViL.

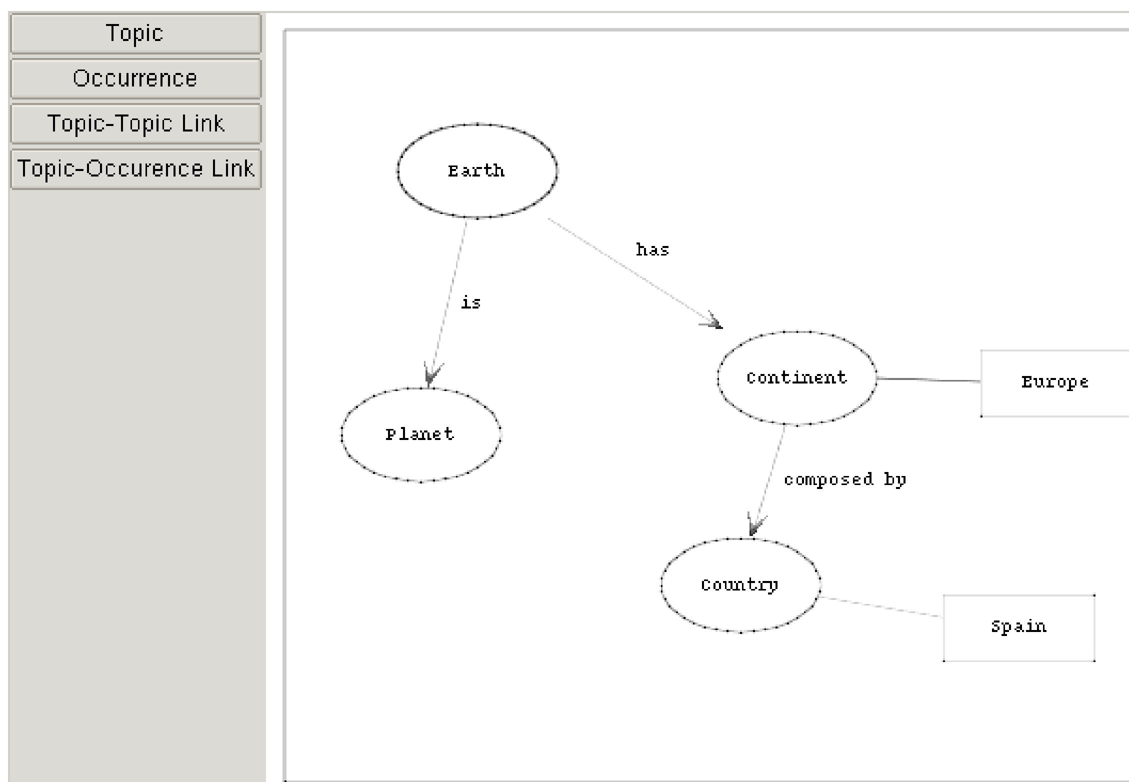


Figure 3.1: Generated Environment for TM using DEViL.

3.2 VLDesk

VLDesk [CDP04] generates visual programming environments based on the eXtended Prositional Grammars (XPG) formalim.

It is an integrated development environment (IDE). All the steps on the definition of a visual language and its programming environment are concentrated in a single and complete interface. To draw the visual language icons, it offers an easy-to-use drawing tool. With these icons, is possible to define the productions of the grammar, which supports the visual language, in a visual way. However not every specification underlying the productions can be done using the visual editor. A big and complex part must be done resorting to YACC-based textual specifications.

The generated environment is not stand-alone[2]. However, it is very simple to draw a specification on that generated environment.

The code generation may be possible, but it seems to be very tricky and hard to achieve[3]. Being possible, the translation would follow the approach used in YACC, when a translation must be performed.

The code written to create the visual language is not easy to understand, unless the user is an YACC expert. So that, maintaining the code or altering it is an not so easy task. However, the translation of this `TM` language into `DRAW` was not so difficult.

In Figure 3.2 is shown an image of a part of the editor generated by VLDesk.

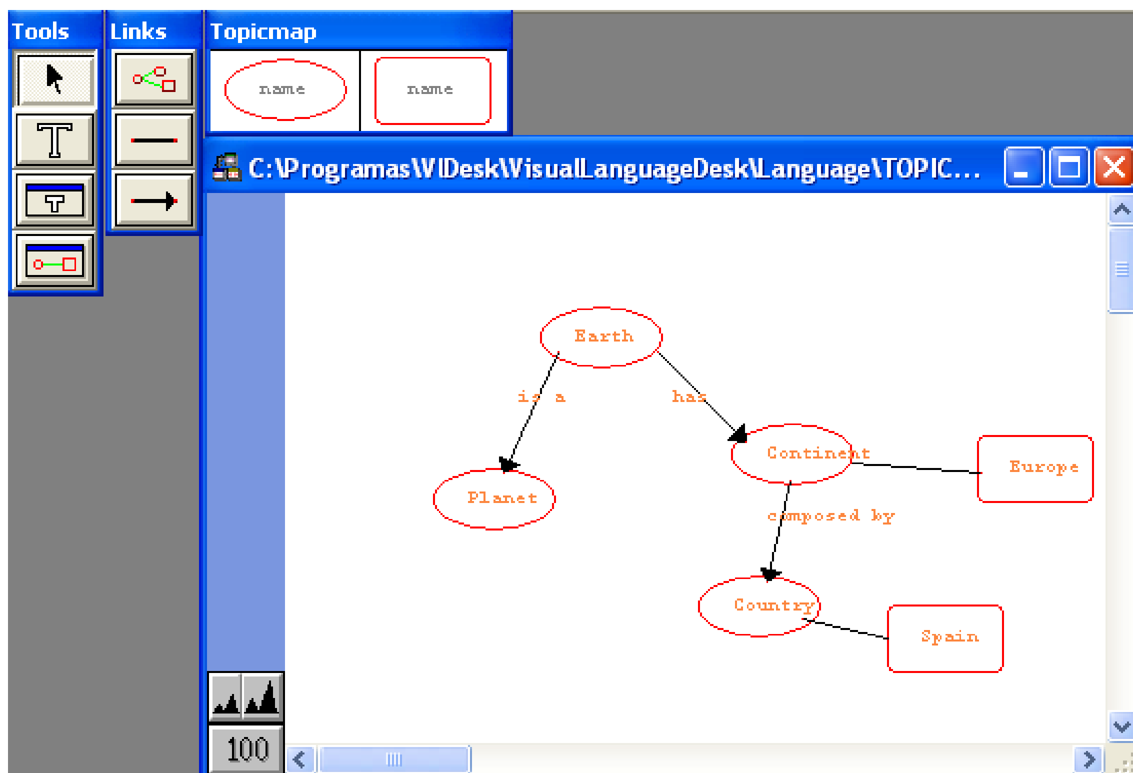


Figure 3.2: Generated Environment for `TM` using VLDesk.

[2] No documentation was found about the possibility of creating a stand-alone version of the editor.
[3] Once again, this was not deeply explored.

3.3 TIGER Project

TIGER [EEHT05], also known as TIGER Project, is a bit different from the tools presented before, because it is a rewrite-rules based system, that is, it is based on a formal graph transformation visual language specification. However, it also generates an editor, which in fact is an Eclipse Plugin. Also the language development is done in an integrated environment, which is itself a plugin for Eclipse.

To build the abstract structure of the language (the abstract grammar) TIGER offers a simple class diagram editor. This is useful, because the syntax of the language is achieved at a higher level. The icons that refer to the terminal symbols are created resorting to a visual environment integrated in TIGER. This icons are automatically associated to the symbols of the grammar.

The definition of transformation rules is a simple but tricky task. These rules must be defined to create the behavior of the language when inserting new symbols or removing one of the existent.

The generated environment, as highlighted before, is a plugin to use within the Eclipse system. So it is not stand alone environment nor it is very complete, in the sense that many traditional features are lacking.

The greatest fragility of this tool (at least for the kind of projects one defined) lies on the fact of being impossible to generate textual code.

3.4 Other Tested Not Tools

In this section are presented some tools that were found but not tested. Besides a brief presentation, it is stated why or why not these tools would fit to the development of `Visual`$\mathcal{LISA}$.

3.4.1 AToM^3

AToM^3 is a multi-formalism and meta-modeling tool. It is based on graph rewriting techniques and also on graph grammars to convert one formalism into another [dLV02]. The tool offers the possibility of code generation and also the specification of simulators.

Regarding the brief description of this tool, one can say that it could be used to develop `Visual`$\mathcal{LISA}$. The fact of being possible the code generation and the complete modeling of the user interface, are favorable points. However, several tests should be made in order to have a better knowledge of the features of this system.

3.4.2 MetaEdit+

MetaEdit+ [TR03] is an integrated development environment for definition of domain-specific modeling languages. It allows code generation from the model drawn, among many other functionalities. However, to use the designed language, it is needed some sort of base motor. So it is not as stand-alone as one would desire.

This tool could be perfect to develop `Visual`$\mathcal{LISA}$, but it does not serve the interests of the team, because it is a commercial tool.

3.4.3 DiaGen

DiaGen [MV95] is a system for the development of diagram editors. It is based on a formal definition of the diagram language to create the syntax of that language. It also allows

semantics constraints definition, ans the generated editor performs syntax verification at run-time, that is, the edition is syntax directed. Moreover, the editor can be completed and extended by using `Java` language.

At a first sight, this tool seems not to be good to develop `Visual`$\mathcal{LISA}$, because `Visual`$\mathcal{LISA}$ is not intended to be a diagrammatic language.

3.4.4 SIL-Icon

SIL-Icon (Syntactic Interactive Learning - Icon) [CTYY89] is used to specify, interpret and prototype visual languages. It is based on a context free grammar that uses spatial operators to define the language parser. In addition it allows the specification of semantic constraints.

From what was read, this tool seems to be not oriented to the definition of complex iconic languages like `Visual`$\mathcal{LISA}$ is intended to be. So that, SIL-Icon is not a good tool to use on `Visual`$\mathcal{LISA}$'s development. Moreover, this tool was not found on the internet.

3.4.5 VLG

VLG (Visual Language Generator) [CGP$^+$90] is an example-oriented visual language generator. That is, the system receives a set of visual sentences, and, through an inference mechanism, produces the grammar for the sentences given. Moreover, it is capable of inferring semantic information from the examples.

Despite of being a very interesting tool, from what is read in the literature, it seems that it would be not desirable to use for more complex languages where the syntax and the semantics must be very well defined. So that, this system is also discarded from the set of possible tools to use on the development of `Visual`$\mathcal{LISA}$. Moreover, this tool was not found on the internet.

3.4.6 VPW

VPW (Visual Prgrammers's Workbench)[RJG90] is a customizable environment generator for specification of visual programs. It presents a great set of tools to build visual languages. These languages can be defined resorting to the specification of its abstract structure, syntax, static and dynamic semantics. It resorts to the picture layout grammar formalism and to several disjoint specifications to define the language compiler.

In fact, without testing this tool, one can not reason whether this is desirable or not to develop `Visual`$\mathcal{LISA}$. At a superficial view, it seems a good tool, but experiences and a more deep research would be necessary to have a final opinion. However, this tool was not found available for download.

3.4.7 Spargen

Spargen [GM93] generates visual language compilers from an Object Oriented Picture Layout Grammar (OOPLG) specification. This tool was developed by the same authors of VPW, and like VPW it has not a graphical interface for definition of the grammars.

From that brief description, anything can be decided about its usage as the environment generator for `Visual`$\mathcal{LISA}$, and it was not found on the internet, so that, its use was not a possibility.

3.4.8 CoCoViLa

COCOVILA [GST05] is a visual programming environment generator. It translates schemas into textual representations or into programs that represent the meaning of the schemas. Semantics constraints can be implemented by using attribute grammars and apply them to the schema language. The dependencies between the attributes are `Java` methods. Code generation is achieved by using the values of such attributes.

Once again, at a superficial point o view, COCOVILA seems to be a good system to develop Visual$\mathcal{LISA}$. But experiences and more research about this tool and its features would be necessary to decide. However, the tool was not available to download on the its site.

3.5 Why DEViL

In the last sections, some tools were presented. Some of them were experimented against the problem described in the beginning of this chapter, the others were only researched and briefly described. Obviously, the choice for the tool which is to be used on Visual$\mathcal{LISA}$ development is one from `DEViL`, VLDesk and TIGER.

VLDesk is a little bit confuse despite of all steps for creating a new visual programming environment being accomplished inside the same program. VLDesk is distributed only for Windows operating systems, but its major handicap lies on the fact of the generated environment not being a stand-alone application.

TIGER is a graph-transformation tool. *A priori* it is already set apart, because it lacks a great set of features for defining a competitive visual programming environment for complex visual languages like Visual$\mathcal{LISA}$, but essentially because there are no facilities offered for code generation.

The systematization can be achieved in a very modular and

During the experiments, `DEViL` was the most convincing tool. Despite of the initial difficulties, it revealed to be the most complete compared to the others:

i) runs in the most important operating systems;

ii) generates stand-alone, intuitive and complete visual programming environments;

iii) a systematic and modular approach for the visual language definition can be followed;

iv) is extensible in the sense that is not limited to the base functionalities;

v) offers great flexibility on choosing the graphical design of the language icons;

vi) allows layout reutilization by coupling structures;

vii) all the environment is based on modular attribute grammars, symbol-oriented, rather then production-oriented;

vii) then the generation of customizable code is an easy AG-based translation task;

ix) the specifications are easy to maintain and evolve;

x) the support team is very attentive, unlike the teams of the other tools;

xi) etc.

For this reasons, and several others, `DEViL` was the tool chosen.

Chapter 4

VisualLISA: The Syntax

After describing the main objectives for this work, analyzed all the requirements that the system must hold, and choose a tool to help on the editor generation, it is needed to formally specify the visual language that will be the foundations for the system.

In this chapter is presented the visual language based on the PLG (Picture Layout Grammar) formalism, which is to be explained in section 4.1.

4.1 PLG: A Formalism to Specify Visual Languages

The PLG formalism, defined by Eric J. Golin on his PhD thesis [Gol91], is an attribute grammar to formally define visual languages. It assumes that each terminal and non-terminal symbol has associated a set of attributes.

PLG assumes, also, the existence of pre-defined terminal symbols like **text**, **circle**, **rectangle** and **line**. These geometric primitives are used to represent, in an abstract way, the drawings which will compose a visual language. All of these symbols (whether they are terminals or non-terminals) have, at least, four attributes which are used to compute the spatial location of each symbol. These four attributes are denominated by URx (UpperRightX), LLx (LowerLeftX), URy (UpperRightY) and LLy (LowerLeftY). For example, for terminals like **circle** or **rectangle** these spatial attributes define the rectangular space occupied by each one of these primitives. In contrast, **line** symbols use these attributes to define their start and end points. The **text** primitive has an intrinsic attribute which represents its content, and obviously, those four attributes compute the rectangular space occupied by the content, just like if the **text** primitive was inside a rectangle.

In order to obtain a nicer view over the visual language grammar, the PLG formalism introduces a set of spatial relation operators. These operators can be seen as black-boxes, since they hide the computation of the four spatial attributes presented before. In his master's thesis, Jorge G. Rocha [Roc95], opens these black-boxes and explains, for each one of the operators, the semantic rules that compute the values of those four attributes.

Table 4.1 presents a set of spatial relation operators, and describes their meaning, briefly. Note that not all the operators are present on the list. Only those which are intended to be the most used are showed.

The PLG's syntax is very simple, since it uses a notation similar to BNF's to define productions. The semantic rules are written below the production definition, and to specify semantic constraints it uses the **Where** clause.

Using the operators listed on Table 4.1, PLG's notation becomes even more simple, since these operators have built-in definitions for the spatial attributes. However, it is possible

Table 4.1: PLG Spatial Relation Operators

Operator	**Read**	**Summary**
over(A, B)	A is over B	Relates two symbols vertically, saying that the second symbol must be above the first one.
left_to(A, B)	A is left to B	Relates two symbols horizontally, saying that the first symbol must be left to the second one.
contains(A, B)	A contains B	Composes two symbols. The second symbol must be completely inside the first one.
adjacent_to(A, B)	A is adjacent to B	Relates two symbols in the space. It means that the first symbol must be positioned above, below, on the right or on the left of the second one.
group_of(A)	Area with A symbols	Composes an area with an arbitrary number of symbols of the same type.
tilling(A, B, ...)	Area with A, B, ... symbols	Composes an area with symbols of different types.
follow(L_1, L_2)	L_2 follows L_1	Relates two lines. The second line starts precisely where the first one ends.
join(L_1, L_2)	join L_1 and L_2	Relates two lines. The end point of both lines is the same.
fork(L_1, L_2)	fork L_1 and L_2	Relates two lines. The initial point of both lines is the same.
points_from(L_1, A)	L_1 leaves A	Relates a line with a symbol. It means that the line starts at the border of the symbol.
points_to(L_1,A)	L_1 arrives to A	Relates a line with a symbol. It means that the line ends on the border of the symbol.
labels(A, B)	A labels B	Relates two symbols. To be exact, the first argument must be **text** and the second another kind of symbol. It means that symbol B is labeled by A. In [Roc95], is explained when it is considered a symbol is labeling another

to redefine these computation rules and add attributes to terminals and non-terminals, anytime it is need.

Example in Listing 4.1 shows the definition of a production with the specification of the computation rules and the semantic constraints. Then in Listing 4.2 is showed the same production with the same value for the spatial attributes, but using the *contains* operator.

Listing 4.1: PLG specification of a semantic production, defining semantic rules.

```
1
2  NT → {rectangle, circle}
3        NT.LLx = rectangle.LLx
4        NT.LLy = rectangle.LLy
5        NT.URx = rectangle.URx
6        NT.URy = rectangle.URy
7      Where
8        rectangle.LLx ≤ circle.LLx
9        rectangle.LLy ≤ circle.LLy
10       rectangle.URx ≥ circle.URx
11       rectangle.URy ≥ circle.URy
```

Listing 4.2: PLG specification of a semantic production using operators.

```
1
2
3
4
5
6  NT → containts(rectangle, circle)
7
8
9
10
11 .
```

With PLG formalism is extremely simple to define a visual language. But besides PLG there are other formalisms that can be used to specify a visual language.

MASoViLa (Modular Attribute-based Specification of Visual Languages) [Roc95] is an example of another formalism based on attribute grammars. It is an extension of the MASLP [Hen92] for visual languages. MASoViLa differs from PLG because explores the reuse based on the modularity, that is, each symbol of the visual language is defined in a module divided into four parts (Uses, Structure, Semantics and Translation). These parts completely define a symbol of the visual language, since there we can explicitly say which are the symbols on the RHS; which attributes are associated to that symbol; specify the semantic rules with the semantic constraints; and specify the translation rules for code generation, for instance. It also uses the notion of spatial relation operators, reusing those defined on PLG to define the structure of the symbol being defined.

LGGs (Layered Graph Grammars) are also a possibility to specify visual languages, as can be read on [RS97]. The authors present the formalism and compare it with PLG pointing some deficiencies on the latter. They present also parsing algorithms for their formalism comparing them with specific graph algorithms.

4.2 Visual Language Requirements

A visual programming language must be simple to use when compared with textual programming languages. The visual aspect of a program specified with a visual language, that is, the drawing of this program must be easy to understand and must describe exactly what the programmer has in mind.

An attribute grammar has, as was seen in Introduction, some characteristics that separates it from other grammars or even from programming languages. Dissecting, logically, an attribute grammar, will originate two major parts: Productions and Computation Rules.

The drawing of a production in an visual AG must concern with the creation of connections between LHS and RHS symbols. Moreover attributes should be associated to each symbol of the production whether it is a terminal or a nonterminal, and pertains to LHS or RHS.

Concerning the visual environment to generate, users must be offered the possibility to:

1. drag the symbols used to define a production into the drawing area and
2. connect these symbols with different lines constraining the syntax;

Visually, terminal, nonterminal and attribute symbols should be different in what concerns the shape and the colors used to define its icons. Despite of being a nonterminal, the production's root symbol should be different from the other nonterminal symbols in the production, that is, its shape must be highlighted.

A Computation Rule is a mathematical operation that assigns a value to an attribute. Constants or other attribute values can be used as arguments for these operations.

The drawing of a computation rule is concerned with the association of functions to attributes whose values can be used either as arguments or as output for the transformation operations.

In spite of being a complementary part of a production, in visual aspects, it would be confuse if all the computations of a specific production were built on the same drawing. Therefore, the visual language must provide a way of drawing computation rules separately but over the same production.

When using the visual language, users must be able to:

1. drag the specific symbols for a computation rule into the drawing area;
2. reuse the drawing of productions as templates in order to increment them with the semantic rules and
3. connect the computation rules' specific symbols with the reused elements of the productions.

As for productions, the symbols used to draw the computation rules should be visually distinct.

Based on the fact that an `AG` can be seen as a decorated tree, and each production as a decorated sub-tree, then the image required for the representation of a production is a tree, but, as told in Chapter 2, it must not be a fixed representation.

Hence, somehow, the terminal and nonterminals of the `RHS` should be connected to the production's `LHS`.

Moreover the production tree should be decorated with attributes. Then connections between terminal or non-terminal symbols and attributes are mandatory to understand to which symbol the attribute belongs.

In the end, the attributes should be associated to computation rules in order to define their valuation.

With this small summary it was exposed what should be expected from the visual language in a visual point of view. But, besides this visual point of view about the language, syntactic constraints need also to be defined in order to make the language usage a safer and nicer experience.

Associated with an attribute grammar, there are some semantic constraints that must hold to consider it semantically correct. In Chapter 5 are defined all the semantic constraints that the visual language must comply with.

Next list defines (or reminds) some constraints that an `AG` description must obey and that can be enforced at a syntactic level:

SC.1 The production constructor must associate the production's root with only terminal and non-terminal symbols;

SC.2 An inherited or synthesized attribute must not be associated with a terminal symbol;

SC.3 An intrinsic attribute can only be associated with a terminal symbol;

SC.4 An intrinsic attribute must not be target of any function output (including the Identity function)

These constraints will be taken into account in the formal specification of V$\mathcal{LISA}$, in the next section.

4.3 The Abstract Syntax

In this section, the PLG formalism is used to define the abstract syntax for the visual language, which was detailed in section 4.2

The attributes for each symbol are not defined here, because it will be specified later when a bigger focus is given on the semantics for the visual language.

It was decided to present the grammar by parts, in order to explain interesting issues on each part, making the document easier and nicer to read.

In Listing 4.3 the main structure for our visual language is defined. Nonterminals are denoted by capitalized letters, and terminals are those pre-defined by PLG formalism. Since PLG has a small set of pre-defined terminals, it was decided to extend this set, adding some other geometric figures or line styles: "triangle", "dash_line", "arrow" and "dash_arrow". With these add-ons is possible to clearly specify what is wanted to get from the visual language.

Listing 4.3: Visual Attribute Grammar Syntax - General Definition

```
1
2   AG → contains(VIEW, ROOT)
3
4   VIEW → labels(text, rectangle)
5
6   ROOT → left_to(PRODS, SPECS)
7
8   SPECS → contains(VIEW, CONTENT_1)
9
10  CONTENT_1 → over(LEXEMES, over(TYPES, over(IMPORTS, USER_FUNCS)))
11
12  LEXEMES → group_of(LEXEME)
13
14  TYPES → group_of(TYPE)
15
16  IMPORTS → group_of(IMPORT)
17
18  USER_FUNCS → group_of(UFUNCTION)
19
20  LEXEME → left_to(text, text)
21
22  TYPE → left_to(text, text)
```

```
23
24  IMPORT → tilling(text)
25
26  UFUNCTION → left_to(text, text)
27
28  PRODS → group_of(SEMPROD)
29
30  SEMPROD → contains(VIEW, CONTENT_2)
31
32  CONTENT_2 → left_to(RULES, AG_ELEMS)
33
34  AG_ELEMS → group_of(AG_ELEM)
35
36  AG_ELEM → LEFT_SYMBOL
37          | NON_TERMINAL
38          | TERMINAL
39          | SYNT_ATTRIBUTE
40          | INH_ATTRIBUTE
41          | TREE_BRANCH
42          | INT_ATTRIBUTE
43          | SYNT_CONNECTION
44          | INH_CONNECTION
45          | INT_CONNECTION
46
47  RULES → group_of(RULE)
48
49  RULE → contains(VIEW, CONTENT_3)
50
51  CONTENT_3 → group_of(RULE_ELEM)
52
53  RULE_ELEM → FUNCTION
54            | IDENTITY
55            | FUNCTION_ARG
56            | FUNCTION_OUT
```

In Listing 4.4 the visual language syntax definition, continues. This second part presents the non-terminal symbols which are concerned with visual syntactic rules construction.

Listing 4.4: Visual Attribute Grammar Syntax - Symbols Concerning Productions

```
1
2   LEFT_SYMBOL → labels(text, rectangle)
3
4   NONTERMINAL → labels(text, circle)
5
6   TERMINAL → labels(text, rectangle)
7
8   SYNT_ATTRIBUTE → labels(text, triangle)
9
10  INH_ATTRIBUTE → labels(text, triangle)
11
12  INT_ATTRIBUTE → labels(text, triangle)
13
14  SC.1:
```

```
15  TREE_BRANCH → points_from(
16                        points_to(line, ~TERMINAL),
17                        ~LEFT_SYMBOL)
18                | points_from(
19                    points_to(line, ~NONTERMINAL),
20                    ~LEFT_SYMBOL)
21
22  SC.2:
23  SYNT_CONNECTION → points_from(
24                        points_to(dash_line, ~SYNT_ATTRIBUTE),
25                        ~LEFT_SYMBOL)
26                    | points_from(
27                        points_to(dash_line, ~SYNT_ATTRIBUTE),
28                        ~NON_TERMINAL)
29
30  SC.2:
31  INH_CONNECTION → points_from(
32                        points_to(dash_line, ~INH_ATTRIBUTE),
33                        ~LEFT_SYMBOL)
34                   | points_from(
35                        points_to(dash_line, ~INH_ATTRIBUTE),
36                        ~NON_TERMINAL)
37
38  SC.3:
39  INT_CONNECTION → points_from(
40                        points_to(dash_line, ~INT_ATTRIBUTE),
41                        ~TERMINAL)
```

In the second part of the grammar, a signal that have not yet been presented is used. The '∼' unary operator. It is used always before a symbol, and defines a context. That is, this operator is used to explicitly say that a production only can occur if the symbol marked by '∼' is present on the context of that production, i.e. it makes the symbol as compulsory.

Listing 4.5 defines the last part of our visual language grammar. The symbols defined here concern with visual computation rules construction.

Listing 4.5: Visual Attribute Grammar Syntax - Symbols Concerning Computation Rules

```
1
2   FUNCTION → over(rectangle, text)
3
4   SC.4:
5   IDENTITY → points_from(
6                  points_to(arrow, ~INH_ATTRIBUTE),
7                  ~INH_ATTRIBUTE)
8              | points_from(
9                  points_to(arrow, ~INH_ATTRIBUTE),
10                 ~SYNT_ATTRIBUTE)
11             | points_from(
12                 points_to(arrow, ~SYNT_ATTRIBUTE),
13                 ~SYNT_ATTRIBUTE)
14             | points_from(
15                 points_to(arrow, ~SYNT_ATTRIBUTE),
16                 ~INH_ATTRIBUTE)
```

```
17                | points_from(
18                        points_to(arrow, ~INH_ATTRIBUTE),
19                        ~INT_ATTRIBUTE)
20                | points_from(
21                        points_to(arrow, ~SYNT_ATTRIBUTE),
22                        ~INT_ATTRIBUTE)
23
24 FUNCTION_ARG → points_from(
25                        points_to(dash_arrow, ~FUNCTION),
26                        ~SYNT_ATTRIBUTE)
27                | points_from(
28                        points_to(dash_arrow, ~FUNCTION),
29                        ~INH_ATTRIBUTE)
30                | points_from(
31                        points_to(dash_arrow, ~INH_ATTRIBUTE),
32                        ~INT_ATTRIBUTE)
33
34 SC.4:
35 FUNCTION_OUT → points_from(
36                        points_to(arrow, ~INH_ATTRIBUTE),
37                        ~FUNCTION)
38                | points_from(
39                        points_to(arrow, ~SYNT_ATTRIBUTE),
40                        ~FUNCTION)
```

4.4 The Visual Syntax

In the previous section was defined, resorting to PLG formalism, the abstract syntax for the visual language.

In textual language grammars, characters are the basic piece to define terminal symbols. These characters are glued by Regular Expressions (RE) to represent valid words.

In PLG there is no notion of characters or REs. However, as told before, there are the geometric primitives that can play the role of such characters; and the spatial operators "*labels*" and "*points_X*", where X is *to* or *from*, can be seen as constructors for graphical symbols, like the REs do with the characters. The graphical symbols, i.e. the icons of the visual language, define terminals symbols in a visual language.

As seen in previous chapter, some nonterminal symbols were defined with these operators. They are not terminal symbols, but are used to define final derivation rules. This means that those nonterminal symbols derive in a terminal symbol which is, in fact, an icon of the visual language[1].

In the present section, a list with those symbols is defined, in order to associate them with the icons for the language. But not all of them are interesting, since not even all are that relevant for the grammar drawing and final visual syntactic aspect.

Some of these special nonterminal symbols are used to aid the user on specifying global definitions for the attribute grammar, like type definitions, lexeme declarations or even user functions code. So, for a visual syntactic aspect, they are not as important as the symbols that will represent the attribute or the nonterminal symbols on the visual attribute grammar specification.

[1]In fact, this realizes what happens in visual programming environments: when a symbol is dragged from the dock, it is transformed into an icon of the visual language underlying.

Table 4.2 defines a mapping between the nonterminals of the meta-grammar defined before and the names that will be used, from here on, to refer to them and to the visual symbols that they represent.

META-GRAMMAR NONTERMINAL	VISUAL SYMBOL NAME
VIEW	*View*
LEFT_SYMBOL	*LeftSymbol*
NON_TERMINAL	*NonTerminal*
TERMINAL	*Terminal*
SYNT_ATTRIBUTE	*SyntAttribute*
INH_ATTRIBUTE	*InhAttribute*
INT_ATTRIBUTE	*IntrinsicValueAttribute*
TREE_BRANCH	*TreeBranch*
SYNT_CONNECTION	*SyntConnection*
INH_CONNECTION	*InhConnection*
IV_CONNECTION	*IntrinsicValueConnection*
FUNCTION	*Function*
IDENTITY	*Identity*
FUNCTION_ARG	*FunctionArg*
FUNCTION_OUT	*FunctionOut*

Table 4.2: VLISA's terminal symbols

Symbol *View* is a special nonterminal that, *per se* is not so interesting. Several nonterminals use it to define their derivation rule, but for further explanations in future chapters, only two of them are relevant: SEMPROD and RULE. They will be referred to as *Semprod* and *ComputationRule*.

Semprod is the symbol that creates the container for one visual semantic production. Each semantic production is a combination of one decorated production tree and some semantic rules, which are given by the *ComputationRule* terminal.

The production tree is the tree structure constructed by the combination between the `LHS` symbol – given by the *LeftSymbol* – and the sequence of `RHS` symbols – given by *Terminal* and *NonTerminal*. That combination is done by associating the `RHS` symbols to the *LeftSymbol* using a *TreeBranch*.

The production tree is decorated with the attributes attached to each of the tree knots referred before. The attributes are given by *SyntAttribute*, *InhAttribute* and *IntrinsicValueAttribute*, and are attached to each symbol, regarding the syntactic constraints, by *SyntConnection*, *InhConnection* and *IntrinsicValueConnection*, respectively.

A *ComputationRule* adds value to the constructed tree. Each semantic rule can be defined by a generic operation or function – given by *Function* – or the often used identity function, which is given by the *Identity* symbol. The latter relates two attributes, regarding the syntactic constraints, with the semantic value of assigning the target attribute the value of the source one. In its turn, each *Function* assigns value to attributes – given by *FunctionOut* – using other attributes' value as arguments. The latter is done by relating *Function* to attributes with the *FunctionArg* symbol.

After this explanation of how the special nonterminals symbols are used to construct an attribute grammar, is possible to define the visual aspect, i.e., images for these symbols.

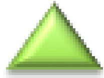
Figures 4.1 and 4.2 display the visual aspect of each symbol defined in Table 4.2, with exception for *ComputationRule* and *Semprod*, which are viewed as symbol containers and

not as concrete symbols.

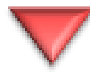
LeftSymbol

NonTerminal

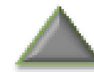
Terminal



SyntAttribute



InhAttribute



IntrinsicValueAttribute

Function

Figure 4.1: Defined figures for terminal symbols - Concrete symbols

Earlier in this chapter (Section 4.2) was presented the language requirements in order to have a visually attractive and easy-to-use visual language. The shape of the symbols was one of the explicit requirements insisted on, because it is very important, for a visual language, to have a nice and easy set of icons which together make a perceptible drawing.
So that were defined different shapes for each terminal icon, trying to ally both simplicity and clarity, adding colors to improve the readability of each one.

In this section was followed the *Icon Theory* [Cha90] to associate the meaning and the image to the most important symbols of V$\mathcal{LISA}$. From here on it is easier to map the name to the face, being also easier to understand what is being described and talked about in the sequel.

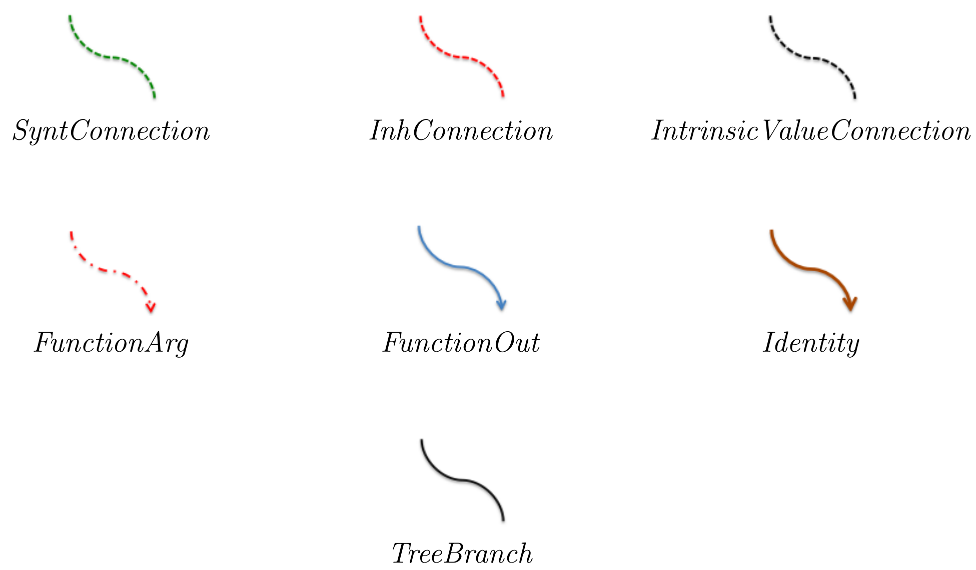


Figure 4.2: Defined figures for terminal symbols - Connection symbols

Chapter 5

VisualLISA: The Semantics

In this chapter is presented and described in natural language first, and then in a formal way, the semantic constraints that concern $\mathtt{V}\mathcal{LISA}$ defined by the grammar presented on Chapter 4.

Before specifying the semantic constraints, it is necessary to declare and associate attributes to the symbols defined and highlighted in Section 4.4. Then a formal description of these constraints can be made.

5.1 Attributes Definition

The semantic constraints of a programming language are directly related with attributes and their concrete values. These values can be evaluated or inferred from the specific context in which symbols occur in that sentence.

In Chapter 4 the VL syntax was defined with an attribute grammar formalism. However, no attributes, besides the PLG implicit ones, were declared or used in semantic rules. Because the attributes will be needed in the next section to describe the semantics, here they will be declared and associated to each terminal symbol.

Below are specified, for each relevant symbol X of the visual language (see table 4.2, second column) the set $A_{CC}(X) \subset A(X)$ of its attributes. $A_{CC}(X)$ is a not standard way of defining the set of the attributes used to define the contextual conditions.

The type of each attribute is declared along with a simple explanation of its purpose. The following syntax is used:

$$Attribute :: Type : Description$$

Semprod

- ⋆ **name** :: String : stores the name of the symbol;
- ⋆ **nLHS** :: Int : stores the number of *LeftSymbol* inside a production;
- ⋆ **lstTerm** :: List : stores all the *Terminal* in the production;
- ⋆ **lstNTerm** :: List : stores all the *NonTerminal* in the production;
- ⋆ **lstSyntAtt** :: List : stores all the *SyntAttribute* in the production;
- ⋆ **lstInhAtt** :: List : stores all the *InhAttribute* in the production;
- ⋆ **lstIVAtt** :: List : stores all the *IntrinsicValueAttribute* in the production.

- ⋆ **inAtt** :: List : stores all the *In* attributes in the production.
- ⋆ **outAtt** :: List : stores all the *Out* attributes in the production.
- ⋆ **nRule** :: Int : stores the number of *Function* and *Identity* associated to the production.

LeftSymbol

- ⋆ **symbolName** :: String : stores the name of the symbol;
- ⋆ **lstSyntAtt** :: List : stores all the *SyntAttribute* associated to the symbol;
- ⋆ **lstInhAtt** :: List : stores all the *InhAttribute* associated to the symbol;

Terminal* and *NonTerminal

- ⋆ **symbolName** :: String : stores the name of the symbol;
- ⋆ **nTBCon** :: Int : stores the number of *TreeBranch* connected to the symbol;
- ⋆ **regExp** :: String : stores a regular expression. This attribute is specific for *Terminal*;
- ⋆ **lstSyntAtt** :: List : stores all the *SyntAttribute* and *IntrinsicValueAttribute* associated to the symbol;
- ⋆ **lstInhAtt** :: List : stores all the *InhAttribute* associated to the symbol. Specific for *NonTerminal*;

SyntAttribute*, *InhAttribute* and *IntrinsicValueAttribute

- ⋆ **attributelName** :: String : stores the name of the attribute;
- ⋆ **type** :: TYPE : stores the type of the attribute;
- ⋆ **nSyntCon** :: Int : stores the number of *SyntConnection*connected to the attribute in a production. Specific for *SyntAttribute*.
- ⋆ **nInhCon** :: Int : stores the number of *InhConnection*s connected to the attribute in a production. Specific for *InhAttribute*.
- ⋆ **nIVCon** :: Int : stores the number of *IntrinsicValueConnection*s connected to the attribute in a production. Specific for *IntrinsicValueAttribute*.
- ⋆ **nIdsTarg** :: Int : stores the number of *Identity* symbols incoming into the attribute. Specific for *SyntAttribute* and *InhAttribute*.
- ⋆ **nIdsSrc** :: Int : stores the number of *Identity* symbols out-coming from the attribute. Specific for *SyntAttribute* and *InhAttribute*.
- ⋆ **nfOut** :: Int : stores the number of *FunctionOut* incoming into the attribute. Specific for *SyntAttribute* and *InhAttribute*.
- ⋆ **nfArg** :: Int : stores the number of *FunctionArg* out-coming from the attribute.

ComputationRule

⋆ **ruleName** :: String : stores the name of the rule;

Function

⋆ **functionName** :: String : stores the name of the function;

⋆ **returnType** :: TYPE : stores the function (return) type;

⋆ **operation** :: String : stores the mathematical operation used to calculate the value of an attribute;

⋆ **nfOut** :: Int : stores the number of *FunctionOut* symbols connected to *Function* in a *ComputationRule*;

⋆ **nfArg** :: Int : stores the number of *FunctionArg* symbols connected to *Function* in a *ComputationRule*;

⋆ **nOpArgs** :: Int : stores the number of effective arguments used in the operation declaration.

The other symbols defined on Table 4.2 which were not listed above, have attributes too. But those attributes are not relevant for the contextual conditions formalization. Later, in Chapter 7, those symbols will, finally, be associated to respective attributes.

SyntAttribute, *InhAttribute*, *IntrinsicValueAttribute* and *Function* terminals have one attribute which type is a reference for the symbol "TYPE".

To be possible the description of all contextual conditions, extra symbols must be assigned some more attributes. The symbols were already defined in the meta-grammar and were named: PRODS and TYPES.

The first symbol was defined as set of *Semprod*, so properties related with all the *Semprod* symbols must be stored within this symbol. Thus the definition of these attribute is the following:

PRODS

⋆ **lstRoot** :: List : stores all the elements that are production roots in the respective productions.

⋆ **lstNTerm** :: List : stores all the *NonTerminal*in the grammar.

⋆ **lstSymb** :: List : stores all the *LeftSymbol*, *Terminal* and *NonTerminal* symbols that occurs in the grammar, even when repeated.

The second symbol, was defined as a set of TYPE, which are used to store programming data types. So here can be stored the set of data types used to build the attribute grammar specification.

TYPES

⋆ **lstType** :: List : stores all the TYPE symbols declared.

5.2 Contextual Conditions

The constraints for $\mathtt{V}\mathcal{LISA}$ can be separated into two major groups. One concerning the syntactic rules and another concerning the respective computation rules. The former will be referred as *Production Constraints* (PC), and the latter will be referred to as *Computation Rules Constraints* (CRC).

In fact, some of the constraints are directly related with a generic definition of an attribute grammar, then these constraints are not specific for $\mathtt{V}\mathcal{LISA}$ instead they are well defined and known.

Attention should be paid to the fact that semantic rules to compute the attribute values will be stepped over. It is assumed that values were already computed, in order to focus on the explanation of the contextual conditions.

Besides the attributes, some boolean operations and other type-related functions are used to complete the formal description of the semantic constraints.

Concerning the *Semprod* symbol defined by the production

$$Semprod \rightarrow contains(VIEW, CONTENT_2)$$

the following constraints where elicited:

PC.1 *Semprod*'s `name` can not be the empty String:

$$\text{cc:} \quad Semprod.\mathtt{name} \neq \texttt{""}$$

PC.2 *Semprod*'s number of *LeftSymbol* must be one:

$$\text{cc:} \quad Semprod.\mathtt{nLHS} == 1$$

PC.3 Every *NonTerminal* and *Terminal* symbol on a *Semprod* must be connected only once to a *LeftSymbol* by a *TreeBranch*;

$$\text{cc:} \quad \begin{array}{c} \text{forall } S \in (Semprod.\mathtt{lstNTerm} \cup Semprod.\mathtt{lstTerm}), \\ S.\mathtt{nTBCon} == 1 \end{array}$$

PC.4 Every *InhAttribute* or *SyntAttribute* on a *Semprod* must be attached to a *NonTerminal* or to a *LeftSymbol* by a unique *InhConnection* or *SyntConnection*, respectively.

$$\begin{array}{ll} \text{cc1:} & \begin{array}{c} \text{forall SA} \in Semprod.\mathtt{lstSyntAtt}, \\ \text{SA}.\mathtt{nSyntCon} == 1 \end{array} \\ \text{cc2:} & \begin{array}{c} \text{forall IA} \in Semprod.\mathtt{lstInhAtt}, \\ \text{IA}.\mathtt{nInhtCon} == 1 \end{array} \end{array}$$

PC.5 Every *IntrinsicValueAttribute* on a *Semprod* must be attached to a *Terminal* by a unique *IntrinsicValueConnection*.

$$\text{cc:} \quad \begin{array}{c} \text{forall IVA} \in Semprod.\mathtt{lstIVAtt}, \\ \text{IVA}.\mathtt{nIVCon} == 1 \end{array}$$

Concerning the *LeftSymbol*, *NonTerminal* and *Terminal* symbols the following constraints were drawn:

PC.6 *LeftSymbol*'s `symbolName` can not be the empty String.

$$\begin{array}{ll} \text{p :} & \textit{LeftSymbol} \rightarrow \text{labels(text , rectangle)} \\ \text{cc:} & \textit{LeftSymbol}.\texttt{symbolName} \neq \texttt{""} \end{array}$$

PC.7 *NonTerminal*'s `symbolName` can not be the empty String;

$$\begin{array}{ll} \text{p :} & \textit{NonTerminal} \rightarrow \text{labels(text , circle)} \\ \text{cc:} & \textit{NonTerminal}.\texttt{symbolName} \neq \texttt{""} \end{array}$$

PC.8 Every *NonTerminal* specified on the grammar must be root of one production.

$$\begin{array}{ll} \text{p :} & \text{PRODS} \rightarrow \text{group_of}(\textit{Semprod}) \\ \text{cc:} & \begin{array}{c} \text{forall NT} \in \text{PRODS}.\texttt{lstNTerm}, \\ \text{NT} \in \text{PRODS}.\texttt{lstRoot} \end{array} \end{array}$$

PC.9 *Terminal*'s `symbolName` can not be the empty String.

$$\begin{array}{ll} \text{p :} & \textit{Terminal} \rightarrow \text{labels(text , rectangle)} \\ \text{cc:} & \textit{Terminal}.\texttt{symbolName} \neq \texttt{""} \end{array}$$

Concerning the *SyntAttribute* the *InhAttribute* and the *IntrinsicValueAttribute* symbols the following constraints must hold:

PC.10 The `attributeName` of an attribute can not be the empty String;

$$\begin{array}{ll} \text{p1 :} & \textit{SyntAttribute} \rightarrow \text{labels(text , triangle)} \\ \text{cc:} & \textit{SyntAttribute}.\texttt{attributeName} \neq \texttt{""} \\ & \\ \text{p2 :} & \textit{InhAttribute} \rightarrow \text{labels(text , triangle)} \\ \text{cc:} & \textit{InhAttribute}.\texttt{attributeName} \neq \texttt{""} \\ & \\ \text{p3 :} & \textit{IntrinsicValueAttribute} \rightarrow \text{labels(text , triangle)} \\ \text{cc:} & \textit{IntrinsicValueAttribute}.\texttt{attributeName} \neq \texttt{""} \end{array}$$

PC.11 The `type` of an attribute can not be the empty String;

$$\begin{array}{ll} \text{p1 :} & \textit{SyntAttribute} \rightarrow \text{labels(text , triangle)} \\ \text{cc:} & \textit{SyntAttribute}.\texttt{type} \neq \texttt{""} \\ & \\ \text{p2 :} & \textit{InhAttribute} \rightarrow \text{labels(text , triangle)} \\ \text{cc:} & \textit{InhAttribute}.\texttt{type} \neq \texttt{""} \\ & \\ \text{p3 :} & \textit{IntrinsicValueAttribute} \rightarrow \text{labels(text , triangle)} \\ \text{cc:} & \textit{IntrinsicValueAttribute}.\texttt{type} \neq \texttt{""} \end{array}$$

PC.12 Every occurrence of symbol X in the grammar must be associated to a set of attributes equal to $A(X) = AS(X) \cup AI(X)$

PC.12.1 One occurrence of $X.s$, where $s \in AS(X)$ in a production must be coherent in the sense that $s \in AS(X)$ in any occurrence of $X.s$ in other productions.

$$\begin{array}{ll} \text{p :} & \text{PRODS} \rightarrow \text{group_of}(\textit{Semprod}) \\ & \text{forall } S_1,\ S_2 \in \text{PRODS}.\texttt{lstSymb}, \\ \text{cc:}^1 & \text{if } S_1.\texttt{symbolName} == S_2.\texttt{symbolName} \text{ then} \\ & S_1.\texttt{lstSyntAtt} == S_2.\texttt{lstSyntAtt} \end{array}$$

PC.12.2 One occurrence of $X.i$, where $i \in AI(X)$ in a production must be coherent in the sense that $i \in AI(X)$ in any occurrence of $X.i$ in other productions.

$$\begin{array}{ll} \text{p :} & \text{PRODS} \rightarrow \text{group_of}(\textit{Semprod}) \\ & \text{forall } S_1,\ S_2 \in \text{PRODS}.\texttt{lstSymb}, \\ \text{cc:}^2 & \text{if } S_1.\texttt{symbolName} == S_2.\texttt{symbolName} \text{ then} \\ & S_1.\texttt{lstInhAtt} == S_2.\texttt{lstInhAtt} \end{array}$$

PC.12.3 The data type of an attribute $X.a$ in a production, must be the same in any production that $X.a$ occurs.

$$\begin{array}{ll} \text{p :} & \text{PRODS} \rightarrow \text{group_of}(\textit{Semprod}) \\ & \text{forall } S_1,\ S_2 \in \text{PRODS}.\texttt{lstSymb}, \\ & \text{if } S_1.\texttt{symbolName} == S_2.\texttt{symbolName} \text{ then} \\ \text{cc:} & \text{forall } A \in (S_1.\texttt{lstSyntAtt} \cup S_1.\texttt{lstInhAtt}) \text{ exists} \\ & A' \in (S_2.\texttt{lstSyntAtt} \cup S_2.\texttt{lstInhAtt}), \\ & (A'.\texttt{attributeName} == A'.\texttt{attributeName}) \wedge (A.\texttt{type} == A'.\texttt{type}) \end{array}$$

The constraints referred above were specific for a production or a set of productions. The following list concerns the constraints for the computation rules.

In attribute grammars, computation rules are directly related with the assignment of values to the inherited or synthesized attributes of the production's symbols. As formally defined in Section 1.2, there are a few rules to write successfully these computations. In order to formalize the semantic constraints concerned with the correctness of a computation rule, it is needed to recall what was told, in Section 1.2, about *in* and *Out* attributes of a production.

CRC.1 The Rule's `ruleName` can not be the empty String:

$$\begin{array}{ll} \text{p :} & \textit{ComputationRule} \rightarrow \text{contains(VIEW , CONTENT_3)} \\ \text{cc:} & \textit{ComputationRule}.\texttt{ruleName} \neq \texttt{""} \end{array}$$

CRC.2 Only *Out* attributes can be the target of an *Identity* or a *FunctionOut* connection;

$$\begin{array}{ll} \text{p :} & \textit{Semprod} \rightarrow \text{contains(VIEW , CONTENT_2)} \\ \text{cc:} & \text{A} \in \textit{Semprod}.\texttt{outAtt} \Rightarrow \\ & \text{A}.\texttt{nIdsTarg} == 1 \vee \text{A}.\texttt{nfOut} == 1 \end{array}$$

[1] S_1 and S_2 is a *Terminal*, a *NonTerminal* or a *LeftSymbol*. For this context condition, it is assumed the abstraction that an *IntrinsicValueAttribute* is also a synthesized attribute.

[2] S_1 and S_2 is considered to be a *NonTerminal* or a *LeftSymbol*.

CRC.3 Only *In* attributes can be the source of an *Identity* or a *FunctionArg* connections;

$$\begin{array}{ll} p: & \textit{Semprod} \rightarrow \text{contains(VIEW , CONTENT_2)} \\ cc: & A \in \textit{Semprod}.\texttt{inAtt} \Rightarrow \\ & A.\texttt{nIdsSrc} \geq 0 \vee A.\texttt{nfArg} \geq 0 \end{array}$$

CRC.4 If there are some *out* attributes declared in a *Semprod*, then, at least one rule must exist for that production;

$$\begin{array}{ll} p: & \textit{Semprod} \rightarrow \text{contains(VIEW , CONTENT_2)} \\ cc: & \textit{Semprod}.\texttt{outAtt}.\text{size} == \textit{Semprod}.\texttt{nRule} \end{array}$$

CRC.5 The type of the target attribute and the return type of a function, when they are connected by a *FunctionOut* symbol, must match;

$$\begin{array}{ll} p1: & \textit{FunctionOut} \rightarrow \text{point_from(points_to(arrow, } \sim\textit{InhAttribute}),\sim\textit{Function}) \\ cc: & \textit{InhAttribute}.\texttt{type} == \textit{Function}.\texttt{returnType} \end{array}$$

$$\begin{array}{ll} p2: & \textit{FunctionOut} \rightarrow \text{point_from(points_to(arrow, } \sim\textit{SyntAttribute}),\sim\textit{Function}) \\ cc: & \textit{SyntAttribute}.\texttt{type} == \textit{Function}.\texttt{returnType} \end{array}$$

CRC.6 The type of the target and the source attribute of an *Identity* connection, must match.

$$\begin{array}{ll} p1: & \textit{Identity} \rightarrow \text{point_from(points_to(arrow, } \sim\textit{InhAttribute}),\sim\textit{InhAttribute}) \\ cc: & \textit{InhAttribute}[1].\texttt{type} == \textit{InhAttribute}[2].\texttt{type} \end{array}$$

$$\begin{array}{ll} p2: & \textit{Identity} \rightarrow \text{point_from(points_to(arrow, } \sim\textit{InhAttribute}),\sim\textit{SyntAttribute}) \\ cc: & \textit{InhAttribute}.\texttt{type} == \textit{SyntAttribute}.\texttt{type} \end{array}$$

$$\begin{array}{ll} p3: & \textit{Identity} \rightarrow \text{point_from(points_to(arrow, } \sim\textit{SyntAttribute}),\sim\textit{SyntAttribute}) \\ cc: & \textit{SyntAttribute}[1].\texttt{type} == \textit{SyntAttribute}[2].\texttt{type} \end{array}$$

$$\begin{array}{ll} p4: & \textit{Identity} \rightarrow \text{point_from(points_to(arrow, } \sim\textit{SyntAttribute}),\sim\textit{InhAttribute}) \\ cc: & \textit{SyntAttribute}.\texttt{type} == \textit{InhAttribute}.\texttt{type} \end{array}$$

$$\begin{array}{ll} p5: & \textit{Identity} \rightarrow \text{point_from(points_to(arrow, } \sim\textit{InhAttribute}),\sim\textit{IntrinsicValueAttribute}) \\ cc: & \textit{InhAttribute}.\texttt{type} == \textit{IntrinsicValueAttribute}.\texttt{type} \end{array}$$

$$\begin{array}{ll} p6: & \textit{Identity} \rightarrow \text{point_from(points_to(arrow, } \sim\textit{SyntAttribute}),\sim\textit{IntrinsicValueAttribute}) \\ cc: & \textit{SyntAttribute}.\texttt{type} == \textit{IntrinsicValueAttribute}.\texttt{type} \end{array}$$

CRC.7 The data type of an attribute and the return data type of a function must be a valid data type.

$$\begin{array}{ll}
\text{p1}: & \mathit{InhAttribute} \rightarrow \text{labels(text, triangle)} \\
\text{cc:} & \mathit{InhAttribute}.\texttt{type} \in \text{TYPES}.\texttt{lstType} \\
\\
\text{p2}: & \mathit{SyntAttribute} \rightarrow \text{labels(text, triangle)} \\
\text{cc:} & \mathit{SyntAttribute}.\texttt{type} \in \text{TYPES}.\texttt{lstType} \\
\\
\text{p3}: & \mathit{IntrinsicValueAttribute} \rightarrow \text{labels(text, triangle)} \\
\text{cc:} & \mathit{IntrinsicValueAttribute}.\texttt{type} \in \text{TYPES}.\texttt{lstType} \\
\\
\text{p4}: & \mathit{Function} \rightarrow \text{over(rectangle, text)} \\
\text{cc:} & \mathit{Function}.\texttt{returnType} \in \text{TYPES}.\texttt{lstType}
\end{array}$$

The next contextual constraints are still concerned with computation rules, but are specific for *Function* symbol, and defined by the production

$$Function \rightarrow over(rectangle, text)$$

CRC.8 *Function*'s `returnType` can not be the empty String;

$$\text{cc:} \quad \mathit{Function}.\texttt{returnType} \neq \texttt{""}$$

CRC.9 `functionName` can not be the empty String;

$$\text{cc:} \quad \mathit{Function}.\texttt{functionName} \neq \texttt{""}$$

CRC.10 *Function*'s `operation` can not be the empty String;

$$\text{cc:} \quad \mathit{Function}.\texttt{operation} \neq \texttt{""}$$

CRC.11 A *Function* symbol must be the source one and exactly one *FunctionOut* connection symbol;

$$\text{cc:} \quad \mathit{Function}.\texttt{nfOut} == 1$$

CRC.12 The number of arguments of a *Function* must match the number of arguments used on *Function*'s `operation`

$$\text{cc:} \quad \mathit{Function}.\texttt{nfArg} == \mathit{Function}.\texttt{nOpArgs}$$

Chapter 6

VisualLISA: The Translation

According to what was told in the Introduction (see Figure 1.1 for a faster understanding), a compiler realizes six tasks. Thus there should be defined the modules to make possible the execution of such tasks.

The first and second modules, concerning the lexical, syntax and semantic analyzers, are defined based on what was told in chapter 4 and 5, respectively. These modules will integrate the environment for specifications which, in fact, is `Visual`$\mathcal{LISA}$.

A specification or programming environment is known to be an integration of several tools that aid the programmer in its work. The common tools in such environment are the source code editor, the compiler or interpreter, the linker tool and finally the debugger. A visual programing environment is a little different since the textual source code editor disappears (or partially disappear, since the visual expressions may be used as a graphical interface for the textual source code editor as happens with *Microsoft Visual Studio* [SF03]). Nonetheless, an integrated compiler with its code generator is commonly used.

The last module for the construction of a compiler is concerned to the code generation, and it is an important task. Compilers, that parse textual specifications, often translate the specifications into machine code, in order to be possible their execution in a real computer environment. Some visual programming environments also generate machine code, but others just translate the drawings into textual source code, leaving the other tasks for a classical compiler. At the end, the main requirement of all the code generators is to generate correct and high quality code [ASU86], forgetting its visual elegance when both issues can not be allied.

The objective of `Visual`$\mathcal{LISA}$ is, mainly, to generate source code for $\mathcal{LISA}$ compiler, from the visual attribute grammar specification. But this could diminish the usage of such programming environment. Thus, besides the facility of generating specific $\mathcal{LISA}$ source code, an `XML` representation of the specified model will be generated at users' choice.

In this chapter, a brief explanation about $\mathcal{LISA}$ is done, and the concrete structure of one $\mathcal{LISA}$-compliant `AG`-specification is presented. The second part of the chapter explains the `XML` dialect designed to describe attribute grammars in an abstract notation, assuring the universality of `Visual`$\mathcal{LISA}$.

The aim of this chapter is to gather information about $\mathcal{LISA}$ textual specifications and design the `XML` dialect in order to translate the visual attribute grammar drawings into one of these target text languages.

6.1 LISA

In this section $\mathcal{LISA}$ system is introduced, and the structure of its attribute grammar specification language is explored, in order to explain and extract concrete information to be possible the correct translation of the visual `AG` language.

6.1.1 The Compiler Generator

$\mathcal{LISA}$ is a compiler generator based on attribute grammars, developed at University of Maribor. The main output of this system is a compiler; it is guaranteed from the input, an `AG`. However, much more *tools* are produced. Editors, inspectors or animators are some of the tools also derived from the `AG` given as input; a complete list and description can be seen in *Table 1* of [HPM+05].

$\mathcal{LISA}$ system was developed in `Java`, and so the underlying environment of each $\mathcal{LISA}$ specification is `Java`. Thus the system is multi-plataform and it is easy to extend a specification reusing and declaring new `Java` methods.

$\mathcal{LISA}$ is a complete, extensible and well featured compiler generator with the capacity to generate LR(0),LR(1), LL(1), LALR(1) and SLR(1) parsers.

Unlike other similar systems (p.e. `Lex` & `Yacc`), $\mathcal{LISA}$ is an Integrated Development Environment (IDE) where the syntactic (productions) and semantic (semantic rules) aspects of a language and the respective lexical aspects (`RE`) are written in the same file. This helps the user to write a complete language specification in just one file.

`RE`s, in $\mathcal{LISA}$, are defined according with some rules. The major part of these rules follow the `POSIX ERE` (`POSIX Extended Regular Expressions`) standard, unless some meta-symbols used in $\mathcal{LISA}$ specifications are also used in this standard. In case of symbols overlapping, a '\' character must be placed before the symbol, to interpret it as it is.

Productions are usually defined using `BNF` or `EBNF` notation. $\mathcal{LISA}$ is not an exception. It uses standard `BNF` conventions to write the set of the grammar productions. So that, it is not possible to use extended notation to write a production.

Since $\mathcal{LISA}$ is a compiler-compiler `AG`-based it is, obviously, possible to use synthesized and inherited attributes. Their computations, that is, the semantic rules associated to a production, are basically seen as a set of `Java` assignments. The Attributes are always accessed using the grammar symbol to which it is *attached*, as the next syntax example shows:

$$Symbol.attribute \tag{6.1}$$

Besides this generic aspects on attribute grammar specifications, $\mathcal{LISA}$ also offers the possibility to reuse previous defined grammars by extending them (the traditional OO extension mechanism), and to reuse semantic rules using Aspect Oriented Programming (`AOP`). The latter facility is known as AspectLISA [RMH+06].

The introduction given above is very short compared with what can be said about $\mathcal{LISA}$. However, more information can be retrieved from the system's website[1].

Next section will devote attention to the structure of a $\mathcal{LISA}$ input document.

[1] `http://marcel.uni-mb.si/lisa/`

6.1.2 LISA's AG Specification

As told in the brief introduction about $\mathcal{LISA}$ this system produces the output from an AG specification. This specification is defined in just one document in the sense that everything needed to define such AG is integrated in only one textual document. Well, this is not completely true if the specifications are extended using others already defined. In spite of being an interesting topic, Visual$\mathcal{LISA}$ does not take it into account nor does it for aspects (as presented in AspectLISA). Therefore Visual$\mathcal{LISA}$ is intended to be a visual environment for sober AGdefinitions.

So that, under this work, a $\mathcal{LISA}$ specification is centered in one document. This document follows a predefined scheme from which the syntax must be studied, since the semantics of such specification is, actually, the semantic aspects of any generic AG, and were already addressed in Chapter 1 and also in Chapter 5.

Listing 6.1 shows the template for $\mathcal{LISA}$ specifications needed to retrieve information for the code generated with Visual$\mathcal{LISA}$.

Listing 6.1: Skeleton of a $\mathcal{LISA}$ specification

```
1
2  language L {
3
4     lexicon {
5
6        RegName   RegExp
7        ...
8
9     }
10
11    attributes
12       Type Symbol.attName;
13       ...
14
15    rule ruleName {
16
17       X ::= X_1  ...  X_i  ...  X_n compute {
18
19          semantic_operations
20
21       }
22
23    };
24
25    method methodName {
26
27       java_declarations
28
29    }
30
31 }
```

Along with the explanation of the template presented, a (semi-abstract) CFG will be used to formally define the syntactic aspects of $\mathcal{LISA}$ specifications. The initial production is defined below:

$$p_1 : LisaML \rightarrow \textbf{language}\ id\ \{\ Body\ \}$$

It is easy to note that such a structure can be separated into four major parts:

⋆ Lexicon definition;

⋆ Attributes declaration;

⋆ Productions definition and

⋆ Methods declaration.

Then a second production of the CFG can be:

$$p_2 : Body \rightarrow Lexicon\ Attributes\ Productions\ Methods$$

In fact, these four parts are encapsulated in a bigger one that identifies the language being defined, giving it a name. In line 2 of Listing 6.1 the reserved word **language** determines where the specification for the language, with name `L`, starts. Notice that the specification body lies in a block delimited by '{' and '}'.

The parts that really define the language and the attribute grammar itself are discussed below with a finer syntactic detail, together with the continuation of the CFG.

Lexicon

$$p_3 : Lexicon \rightarrow \textbf{lexicon}\ \{\ LexBody\ \}$$

$$p_4 : LexBody \rightarrow (regName\ regExp)\star$$

The lexicon part of the language is where the programmer should declare the lexemes that are used to perform the lexical analysis creating a sequence of tokens from the input text.

In $\mathcal{LISA}$ the lexicon part is seen as a simple set of lexemes. Lexemes are pairs "`RegName RegExp`" where `RegName` is an identifier that maps the `RE` defined by `RegExp`.

`RegExp` is a `RE` as presented in the introduction of this chapater.

A lexeme can reuse a previous defined one. The syntax to reuse a lexeme is very simple, it is only needed to add a ♯ before the identifier, i.e., the `RegName`.

Attributes

$$p_5 : Attributes \rightarrow \textbf{attributes}\ (type\ symbol\ .\ attName\ ;)\star$$

The attributes of a symbol are referred as described in the Syntax Example 6.1.

The declaration of attributes in a $\mathcal{LISA}$ document starts with the reserved word **attributes**. Unlike the lexicon part, this declaration is not contained between curly-braces. However a set of "`Type Symbol.attName`" pairs is declared, but this time, each declaration is separated by a semicolon.

Since $\mathcal{LISA}$ is a `Java`-based tool and also generates `Java`-based environments, then it is legitimate to take advantage from the synergy between $\mathcal{LISA}$ and `Java`. So that `Type` must be a `Java` valid type.

Productions

$$p_6 : Productions \rightarrow \textbf{rule}\ id\ \{\ Derivation\ \}\ ;$$

$$p_7 : Derivation \rightarrow symbol\ ::=\ Symbs\textbf{compute}\ \{\ SemOperations\ \}$$

$$p_8 : Symbs \rightarrow symbol+$$

$$p_9 : Symbs \rightarrow \textbf{epsilon}$$

$$p_{10} : SemOperations \rightarrow symbol\ .\ attName\ =\ Oper\ ;$$

$$p_{11} : Oper \rightarrow \ldots$$

The productions definition part is where the user defines the syntax and the semantics for the language in development.

In $\mathcal{LISA}$ specifications, that part is constituted by a set of rules, separated by a semicolon. In turn, each rule is composed by three main parts which appears after the reserved word **rule**:

1. **Name**, presented in Listing 6.1 as `rulename` is an identifier for the actual rule.
2. **Production** appears after the `ruleName` inside a block delimited by curly-brackets.

   The production must be declared using the `BNF` notation. The symbol "::=" is used to separate the `LHS` from the `RHS` of the production. After that declaration, the reserved word **compute** must appear in order to determine where the semantic definitions start.

   Every nonterminal symbols in the grammar are written with capitular letters, p.e. STUDENT. The terminal symbols are written following three syntactic constraints, regarding the nature of the symbol:

   ⋆ If the terminal is a reserved word, then it must be written in lower letters.
   ⋆ If the terminal is the identifier of a lexeme defined in the lexicon part, then the identifier must be preceded by ♯.
   ⋆ If the terminal is a $\mathcal{LISA}$ special character, then it must be preceded by \.

   When the `RHS` of the production is an empty sequence, then $\mathcal{LISA}$ uses the reserved word **epsilon** to denote that the symbol on the `LHS` derives into the null or empty string.

3. **Semantics** is contained in a block { } that starts after the reserver word **compute**. In Listing 6.1, this part is referred as `semantic_operations`.

   In practice it corresponds to a set of `Java`-based assignment operations. Hence all the operations follows this syntax:

   $$X.a = func(..., X_i.b, ...)$$

   Where attributes a are *out* attributes and b are *in* attributes[2].

   In the operations, almost all the symbols represented by X are nonterminals, therefore they must be written in capitular letters. Nonetheless, some symbols may be terminals and the syntax to write them was already described above. There are

[2] Remember the *in* and *out* attributes definition presented in Chapter 1

no attributes for terminal symbols in $\mathcal{LISA}$, however the intrinsic values can be retrieved using functions like `value()`.

When in the same production co-exixt symbols with the same name, whether they appear repeated on the `LHS` or in the `RHS`, a number, regarding the number of times it appears repeated on the list of symbols, must be added, in order to be possible to understand which one is being used. Listing 6.2 shows an example of this situation:

Listing 6.2: Production with repeated symbols

```
1
2 rule xpto {
3     X := X A compute {
4         X[0].a = A.a + X[1].a;
5         X[1].b = X.b;
6     }
7 }
```

Note that is not necessary the usage of [0] to indicate the usage of the first symbol with name X.

Methods

$$p_{12} : Methods \rightarrow \textbf{method}\ id\ \{javaDeclarations\ \}$$

The last part is a zone on the specification where the user can declare additional functions or additional packages.

Each method starts with the reserved word **method** and requires an identifier (`methodName`). Inside the block delimited by curly-brackets, it must be written valid native `Java` code, since:

Legal `Java` *code = Legal* $\mathcal{LISA}$ *method code.*

With the information retrieved from the study of some $\mathcal{LISA}$ documents, it was possible to reconstruct the base model for a generic specification. In every document, new syntactic aspects were found and presented here, in order to gather all the information for the implementation steps on the $\mathcal{LISA}$ code generation from the attribute grammar specified in `Visual`$\mathcal{LISA}$.

6.2 Universality for Attribute Grammars

In the late 1990's a group of people invented `XML`. `XML` is often regarded as a markup language. In fact, it is wrong, since this *buzzword* is only a specification to define markup languages. In other words, `XML` is a meta-markup-language. But what it is or it is not, does not removes from it the great advance it brought to computing and in particular to the information storage.

`XML` documents are completely platform-independent. Despite the variety of structures they can have, these documents are text containers, hence any text-reader software is able to open it. But what really makes them being platform-independent is the fact that they are never executed like `Java` programs, for instance. Their content is just structured (annotated) data that can be used and transformed by programs that can read and interpret it.

XML's famous portability is always faced as an admirable quality. In fact it is, because the information from a program can be uploaded in another if they recognize the same XML structure, that is, if the first program saves the information in an XML structure and the second is able to read that structure. To achieve this interoperability between software tools, there were created XML standard dialects.

Nowadays, there are XML standards for almost everything and in almost any area where informatics gives a hand:

Healthcare: HL7 – Health Level 7;

Finances: XBRL – eXtensible Business Reporting Language;

Mathematics: MathML – Mathematics Markup Language;

Astronomy: FITSML – Flexible Image Transport System Markup Language (by NASA);

Web Services: WSDL – Web Services Distributed Management;

Telecommunications: TIM – Telecommunications Interchange Markup;

Multimedia: SMIL – Synchronized Multimedia Integration Language;

Many standards exist on the referred areas and in a bunch of several others. A more complete list can be seen in [Ass08].

But sometimes, despite of the dialect being a standard it does not mean anything. *XML Metadata Interchange* (XMI) is an example. Although *Object Management Group* (OMG) has defined it as a standard, each software modeling tool generates a different structure for this standard, raising up the impossibility to transport information from a tool to another. Rare are the cases where the structure is compatible.

When there is no standard dialect, the portability is in risk because the target software needs an intermediate filter tool capable of transforming the inputed XML document into a readable one.

However, it is nicer to store the output of a program in an XML document rather then in other textual format if the intention is to, sooner or later, create the referred intermediate filter. The answer lays on the fact that processing an XML file is easier than processing another type of file, because almost every programming language has an efficient way of dealing with XML rather then other annotation languages.

The last declarations create the belief that XML opens opportunities and creates new applications for a tool. This belief, allied to the fact that there is no standard notation to represent attribute grammars, led to the will of designing an XML dialect which represents a universal structure to support attribute grammar specifications.

The aim of this approach is to generate the XML specification from the drawings of the visual attribute grammar, and then be possible to translate it into any AG-based compiler generator specification.

6.2.1 XAGra - An XML dialect for Attribute Grammars

All the XML dialects already standardized, including those listed before in this section, have a unique name to identify itself. To the XML being defined in this sub-section was given the name $\mathcal{X}$AGra which stands for XML dialect for Attribute Grammars. From here on, this new XML notation will be referred to as $\mathcal{X}$AGra.

$\mathcal{X}$AGra should denote an abstract representation of an AG, in order to be syntax-free. From the definition of AG presented in the Introduction, and from the experience acquired

with the study about $\mathcal{LISA}$'s specifications structure (presented in Sub-section 6.1.2), was possible to define the main structure for $\mathcal{X}$AGra.

A correct way to define a new XML dialect is creating a schema, using the standard XML schema definition (XSD) language. The integral textual definition of the $\mathcal{X}$AGra's schema is presented in Appendix A, accompanied by a complete drawing of it.

For reasons of visibility and readability, the referred drawing of the schema is broken into several important sub-parts, and explained in this Sub-section. Figures 6.1 to 6.7 represent these sub-parts, and are used to explain the dialect.

$\mathcal{X}$AGra's root element was defined as `attributeGrammar`. This element has a unique attribute – `name`, whose objective is to keep the name of the grammar or the language that the grammar defines – and is a sequence of several elements. In fact, these elements represent components of an AG's formal definition, complemented by extra parts related with the usage of AG-based compiler generators. These elements are the following:

* symbols;
* attributesDecl;
* semanticProds;
* importations and
* functions.

Below an explanation of what each element represent in an AG will be given.

The concrete diagram of element `attributeGrammar` can be seen in Figure A.1, which represents the overall view of $\mathcal{X}$AGra's underlying schema.

symbols Figure 6.1 presents the schema for the element `symbols`. As the name suggests, this element contains the declaration of the symbols of the grammar.

It is composed by a sequence of three elements: `terminals`, `nonterminals` and `start`.

The element `terminals` is a sequence of zero or more elements named `terminal` which, in its turn, has an attribute – `id`, used to store the name of a terminal symbol of the grammar. This attribute is an identifier, hence any instance of it, must be different from the others, and must be always instantiated. Besides the information kept on the attribute, this element has a textual content where the respective RE can be declared.

The same can be told about element `nonterminals`, since its structure is similar to the one presented for `terminals`. The unique difference lays on the fact that it represents a sequence of zero or more `nonterminal` elements which have not textual content. The attribute `id` has the same purpose as the attribute with the same name in the `terminal` element.

Finally, the element `start` has a single attribute named `nt`. This attribute is used to refer the nonterminal symbol of the grammar (already defined in the $\mathcal{X}$AGra specification), that is, the start symbol (or Axiom) of the AG.

attributesDecl This element is composed by a sequence of zero or more `declaration` elements. For the sake of readability, Figure 6.2 depicts only the structure of element `declaration` that is a sequence of one or more elements `attribute`. The latter element has three mandatory attributes:

Figure 6.1: $\mathcal{X}$`AGra` Schema - **Symbols** sub-part definition

- ⋆ `id` – stores the name of the attribute being declared. Any kind of text can be used to define it, but it is always better to use the syntax reported in the Subsection 6.1.2, Syntax Example 6.1. As it is an identifier, it must be different from all other identifiers on the specification.
- ⋆ `type` – stores the data type of the current attribute value.
- ⋆ `class` – defines the class of the attribute. It must be one of: InhAttribute, SyntAttribute and IntrinsicValueAttribute.

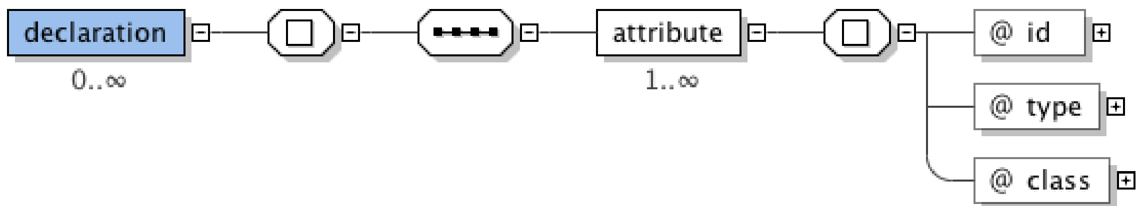


Figure 6.2: 𝒳`AGra` Schema - Attribute Declarations sub-part definition

semanticProds The element `semanticProds` represents the structure to define productions and associated semantic rules in 𝒳`AGra` specifications. This structure is composed by a sequence of zero or more `semanticProd` elements. Each `semanticProd` element has one single attribute – `name` – used to store the name of the production, which is mandatory and is an identifier.

`semanticProd` elements have three direct descendants, whose structure is depicted by Figures 6.3, 6.4 and 6.5. These elements are respectively: `lhs`, `rhs` and `computation`.

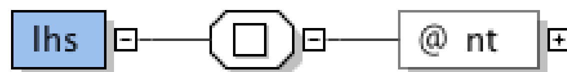


Figure 6.3: 𝒳`AGra` Schema - Semantic Productions sub-part definition: `LHS`

`lhs`, as the name suggests and Figure 6.3 shows, has a very simple structure used to declare the nonterminal symbol on the `LHS` of the production. This element has one only attribute – `nt` – to hold a reference to an existent nonterminal identifier must be created.

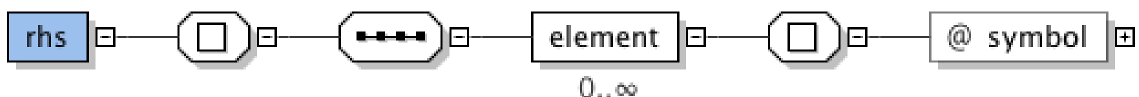


Figure 6.4: 𝒳`AGra` Schema - Semantic Productions sub-part definition: `RHS`

`rhs` element, depicted in Figure 6.4, has a structure appropriate to store the nonterminals on the `RHS` of a production. Like several other elements presented before, this one is composed by a sequence of zero or more `element` elements. For this purpose each `element`

has a single attribute – `symbol` – which is mandatory and represents a reference to one symbol, whether it is terminal or nonterminal, already instantiated in the initial `symbols` structure.

`computation` (Figure 6.5) is the last child of the element `semanticProds`. It is a little bit more complex than the two children of `semanticProd` presented before because it also represents an harder concept of `AG`s: the definition of the semantic rules.

The `computation` element has one attribute – `name` – used to give a name to the computation being declared. This attribute, despite being mandatory, is not a unique identifier, because more than one computation can have the same name, unlike `semanticProd`, for instance.

The structure of `computation` represents a pure abstraction of what is a semantic rule in an `AG` definition: the attribute to which a value is assigned, and the operation that computes this value. Thus, the element `computation` has two children: the `assignedAttribute` and the `operation` elements.

The `assignedAttribute` is a simple reference to a grammar attribute previously declared in the `attributesDecl` part. The element has two mandatory attributes:

⋆ `att` – Is used to reference an attribute;

⋆ `position` – Is a number that identifies the position of the symbol associated to the attribute in the list of elements of the production. That is, If the attribute is connected to the `LHS`, then the value for `position` must be 0. If the associated symbol belongs to the `RHS`, then its value should correspond exactly to the position that the symbol occupies in the `RHS` sequence of symbols, starting with 1.

The `operation` structure aggregates a sequence of zero or more `argument` elements and a single `modus` element. In addition to the elements, it has an attribute – `returnType` – used to store the data type of the value returned by the operation.

`argument` elements are, in all aspects, equal to the `assignedAttribute` element. Each one has two attributes with the same name and the same semantic value underlaying, therefore its structure is used for the same thing, that is, to refer to previous declared grammar attributes. The difference is on the fact that this time, the attributes referenced are those used to compute the value in the operation.

The last element – `modus`, which is a latin expression for *way* (of computing something) – is a simple text field to write the expession used to compute the value. Somehow, in this element's text, a reference to the argument attributes should be made.

This child of `attributeGrammar` finishes the parts that generically and formally define an `AG`. The next two simple parts extend the mathematical definition of `AG`s to the abstract language of any compiler generator based on `AG`s.

importations Figure 6.6 presents the structure to declare the importation of packages or programming language modules that can be necessary for the computation of all attributes. This element, named `importations`, is a sequence of zero or more `import` elements, which in its turn is a simple text container, where the name of the package or module should be written.

functions In a very similar way, `functions` element is a sequence of zero or more `function` elements.

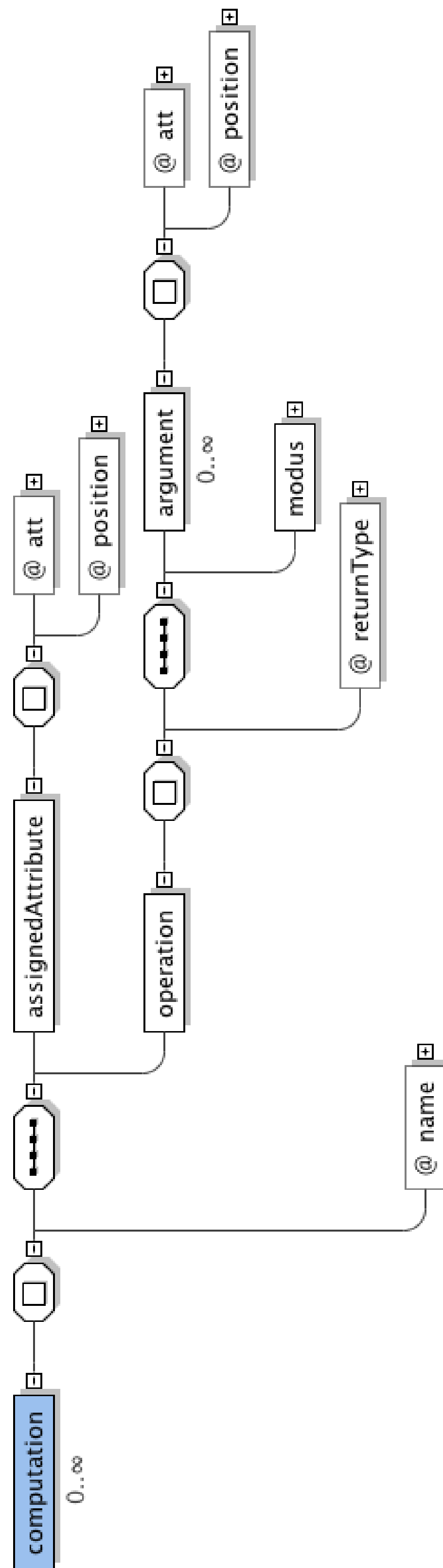


Figure 6.5: $\mathcal{X}$`AGra` Schema - Computations sub-part definition

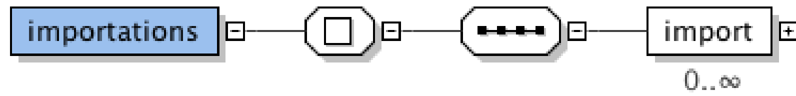


Figure 6.6: $\mathcal{X}$AGra Schema - Importations sub-part definition

Each `function` element has a mandatory attribute – `name` – used to store the name of the function. This element is defined as a text container, in order to be possible the definition of a concrete function. The code of the function must be written in the target programming language like `Java` or other.

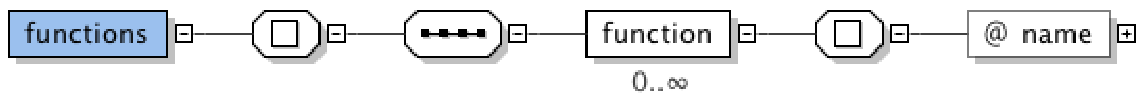


Figure 6.7: $\mathcal{X}$AGra Schema - Functions sub-part definition

$\mathcal{X}$AGra schema is now completely defined and explained, revealing the universality needed to store any AG for any AG-based compiler generator.

A complete diagram or the textual definition of the schema is available in Appendix A.

Chapter 7

DEViL Implementation

Through out this report until this part, the problem underlying the project was defined and the most important tasks were analyzed and explained with a detailed theoretic support.

A good analysis is always half way to have a better and easier implementation.

In this chapter, it will be presented the steps to implement the visual language and the programming environment associated resorting to `DEViL`, selected on Chapter 3. ahhh

7.1 Syntax Grammar

In Chapter 4 it was presented a formal definition of the attribute grammar which specifies V$\mathcal{LISA}$. In this section, that definition will be translated into `DEViL` specific notation, so that the language and a syntactic validator can be generated.

`DEViL` uses a specific textual notation to write its language specifications, as every compiler generators do. Despite of using `AG`s as notation, they are not common `AG`s. `DEViL` gives an object-oriented view over them, making easier the way a language is specified.

Briefly, in `DEViL`'s notation each nonterminal symbol is regarded as a class. That class can be either concrete or abstract whether the nonterminal has only one or more than one option, respectively. The class attributes define terminal symbols if they are defined with built-in data-types like `VLString` or `VLInt`. Otherwise if the data-type of an attribute is not primitive, then it refers to a nonterminal symbol that must be declared somewhere in the grammar specification.

`DEViL`'s syntax is not presented in this document, because it could draw attention into topics out of this document's scope.

In fact, the strategy used by `DEViL` is not completely new because some traditional compiler generators, which rely upon object-oriented platforms, usually translate grammar grammar elements into classes of the underlying programming language, creating the language processor.

But in addition to that simple object-vision, `DEViL` offers the possibility of grouping symbols in abstract classes (by inheritance). This way, it is easier to join classes with similar behavior. Hence the definition of finer constraints for the visual language syntax is an easy task.

Considering the grammar presented on Chapter 4, the effort to translate it into `DEViL`'s notation is almost none, regarding the two simple rules invoked before: *i*) Each `LHS` leads to a concrete or abstract class depending on the number of options and *ii*) The `RHS` symbols become attributes of the class.

For a better understanding about the visual language and its syntactic constraints, it was created a meta-model from the grammar defined through Listings 4.3 to 4.5. Figure 7.1 depicts the class diagram of the visual language.

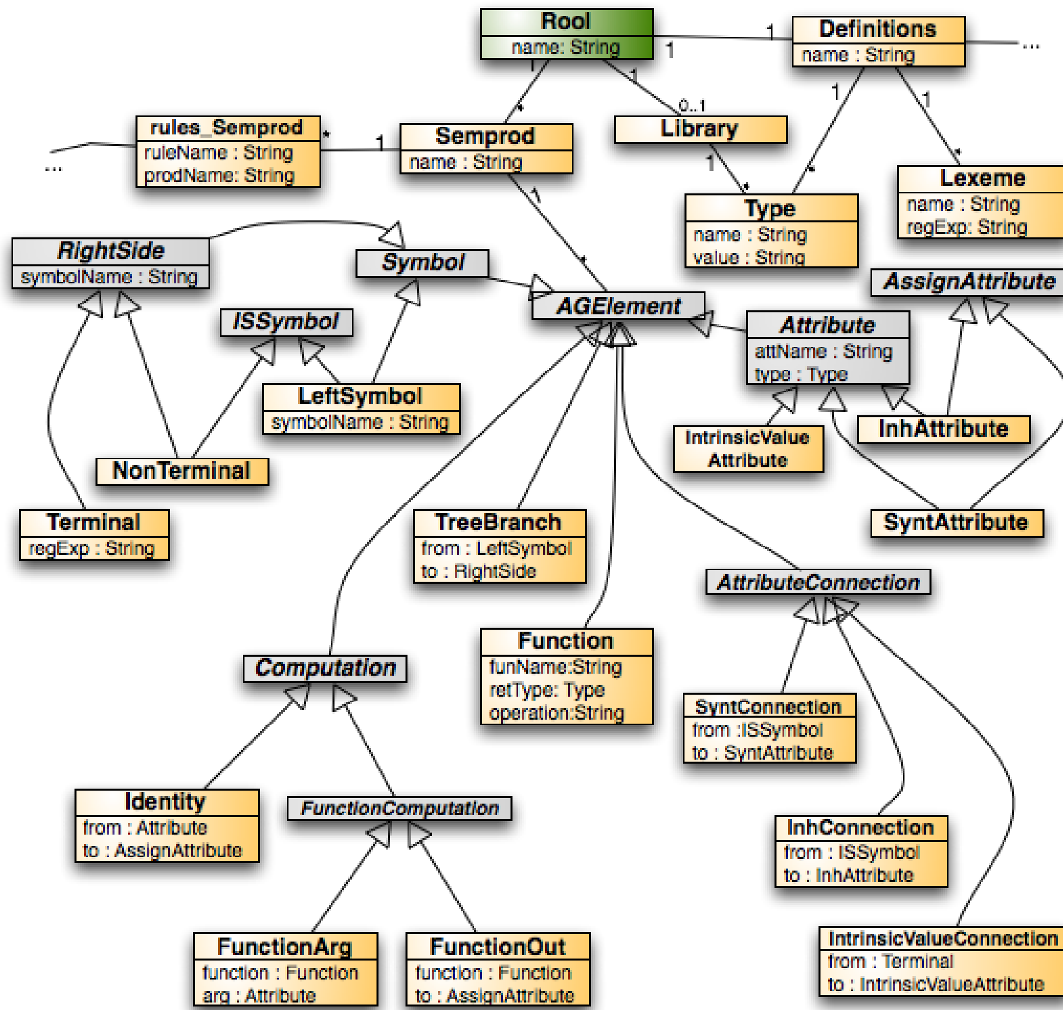


Figure 7.1: Meta-Model for V$\mathcal{LISA}$

Notice that some symbols used in the grammar definition in Chapter 4 have disappeared in this class diagram, and others were added. This means that some of the nonterminals defined before are not as important as others w.r.t the visual language specification in `DEViL`.

The new symbols added (painted in gray, in Figure 7.1) correspond to abstract classes. As told before, these classes are used to group some concrete symbols (painted in yellow) that have similar behavior. For instance, the abstract class *AssignAttribute* is used in the concrete class *Identity* as an attribute. This means that only *InhAttribute* and *SyntAttribute* can be referenced by an object of the class *Identity*. This is true, because both *InhAttribute* and *SyntAttribute* inherit the behavior and the static structure of the class *AssignAttribute*. This implements (in some part) the constraint defined as **SC.4** in Section 4.2, because every attribute type but the *IntrinsicValueAttribute* can be used as a

target of any (including the identity) function.

The class diagram presented is not complete, as can be noticed by the usage of reticences. In the case of the *Definitions* class, it only misses two other classes, as simple as *Lexeme* or *Type* — the *UFunction* and *Import* classes.

The case of *rules_Semprod* is a bit different and will be discussed later with detail.

With a class diagram which, in fact, represents V$\mathcal{LISA}$, its translation into DEViL's notation is a straightforward task. The next subsection shows some parts of the visual language specification[1], and explains some basic things about the notation and what are its repercussions in the visual environment.

7.1.1 The Visual Language Specification

Listings 7.1 to 7.9 show the source code of the visual language specification, using DEViL's notation. Well it is not completely here. For the sake of space and readability, the complete specification was dragged to Appendix B. Some attributes like those used to store the position and the size of a visual symbol were also omitted here for the same reasons.

DEViL needs always a start symbol with a prefixed designation: *Root*. Listing 7.1 shows the structure of that symbol.

As can be seen, *Root* class has four attributes, each one of them with different notations and semantics.

The attribute `name` will be used to store the designation of the new language being defined with the visual attribute grammar specification. It represents a valuable attribute what means that its value is to be filled, by the user, in edition-time. In a textual grammar definition it would be a terminal.

In fact, each concrete class on the DEViL's notation can (but is not mandatory) result into a visual icon. In case of being a base icon for the language, and as long as it is declared with these kind of attributes, it will result into a form with text fields or other usual dialogue fields to be filled by the user, in order to be characterized.

Attribute `semprods` refers to a symbol named *Semprod* and the $\star$ means that it can appear zero or more times in the drawing. The ! after the reference to *Definitions* means that ate least one symbol of that type can appear on the drawing. In its turn, the ? after the *Library* reference says that a symbol of that type can exist, or not, in the final drawing.

Listing 7.1: Class Root

```
1
2
3 CLASS Root {
4   name: VAL VLString;
5   semprods: SUB Semprod*;
6   definitions: SUB Definitions!;
7   library: SUB Library?;
8 }
```

Listing 7.2: EBNF production for symbol Root

```
1
2
3 Root -> name
4          Semprod*
5            Definitions
6              Library?
7
8 .
```

The three last attributes refer to classes in the grammar definition. In other words, these symbols can be regarded as nonterminals which appear in the RHS of that production.

Listing 7.2 presents an EBNF notation to denote what could be a possible textual definition for the same production: `name` is a terminal symbol, *Semprod*, *Definitions* and *Library* are nonterminal symbols with a special cardinality.

[1]The complete specification can be seen in Appendix B.

Definitions and *Library* classes have a simple structure analogous to the *Root*'s, but referencing different symbols.

Semprod has, equally, a simple structure. However it has an important role in V$\mathcal{LISA}$, because it represents a fundamental componente of an AG— the semantic productions. Each semantic production groups both the derivation rule (i.e. the production in BNF notation) and the semantic rules which assigns values to the different attributes of the production.

Listing 7.3 shows the structure of the symbol *Semprod*. It begins with the definition of the attribute `name` which will store the name of the production. Attribute `grammarElements` represents a reference to a list of symbols *AGElement*.

Listing 7.3: Class Semprod

```
1
2
3
4 CLASS Semprod {
5   name:   VAL VLString;
6   grammarElements: SUB AGElement*;
7   computationRules:SUB rules::Semprod*;
8 }
9
10 .
```

Listing 7.4: Some Abstract Classes

```
1
2 ABSTRACT CLASS AGElement {
3 }
4
5 ABSTRACT CLASS Symbol INHERITS
6 AGElement{
7 }
8
9 ABSTRACT CLASS ISSymbol {
10 }
```

Attribute `computationRules` is also defined as a list of symbols *rules::Semprod*. This symbol is marked in a different way. Its notation is divided in two parts separated by '::'. The first part refers to a namespace, and the second refers to a symbol inside that namespace. In Sub-section 7.1.2 this notation and the purpose of its usage will be explained with more detail.

Listing 7.4 presents three examples of abstract classes. Each one of them is different from the others in what concerns both the structure and the semantics.

As told before, an abstract class in DEViL's notation can have two different connotations: *i*) it is used to group symbols that have a similar behavior, in order to define syntactic constraints easily, or *ii*) to define the several options for a symbol.

The symbol *AGElement* bases its usage on point *ii*) since in line 36 of Listing 4.3 its correspondent symbol (AG_ELEM) is formally defined with several alternatives, that it, as derivable in a huge set of possible nonterminals.

Each symbol that derives from *AGElement* must inherit it. An example of an inheritance of this symbol can be seen in Listing 7.4, line 5. Notice that symbol *Symbol* does not make part of the options presented in line 36 of Listing 4.3. *Symbol* is an abstract class which follows the above item *i*) just like the abstract class *ISSymbol* does. The former groups arbitrary symbols of the grammar; the latter is more specific: it is used to group those symbols which can be connected to *InhAttribute* or *SyntAttribute* symbols. As Listing 7.5 shows, only the *LeftSymbol* and the *NonTerminal* symbols inherit it; this prevents a *Terminal* to connect to an *InhAttribute* or a *SyntAttribute*. It is worthwhile to notice the base of multiple-inheritance in DEViL.

Listings 7.5 and 7.6 present the core-symbol for the AG meta-language. At the left are defined the production symbols like *NonTerminal*, *Terminal* and *LeftSymbol*; and at the right are defined the several type of attributes: *InhAttribute*, *SyntAttribute* ans *IntrinsicValueAttribute*.

Notice the usage of the inheritance of structure in symbol *Terminal*, for instance. Besides its specific attribute `regExp`, since it inherits from the abstract class *RightSide*, it has the attribute `symbolName`. The class *Terminal*, although it may be difficult to see in this textual specification, is also a sub-class of *AGElement*. Figure 7.1 is a better way to

see these dependencies among symbols. Each arrow points from a class to the super-class, then following the arrows, it is possible to see the hierarchy of inheritances.

The reserved word EDITWITH is used to explicitly inform `DEViL` of what type of dialogue is to be created to input the information for the attribute. When the type is "None" then no dialogue is created; when EDITWITH is not declared, then the dialogue type created by default is of type "Entry", that is, a text input field.

Listing 7.5: Production's symbols

```
1
2
3
4 CLASS LeftSymbol INHERITS
5 ISSymbol, Symbol {
6   symbolName: VAL VLString;
7 }
8
9 ABSTRACT CLASS RightSide INHERITS
10 Symbol{
11   symbolName: VAL VLString;
12 }
13
14 CLASS NonTerminal INHERITS
15 RightSide, ISSymbol {
16 }
17
18 CLASS Terminal INHERITS
19 RightSide{
20   regExp: VAL VLString?
21             EDITWITH "Entry";
22 }
23
24
25
26 .
```

Listing 7.6: Attribute's symbols

```
1
2 ABSTRACT CLASS Attribute INHERITS
3 AGElement{
4 }
5
6 ABSTRACT CLASS AssignAttribute {
7 }
8
9 CLASS SyntAttribute INHERITS
10 Attribute, AssignAttribute {
11   attributeName: VAL VLString;
12   type : REF Type;
13 }
14
15 CLASS IntrinsicValueAttribute
      INHERITS
16 Attribute {
17   attributeName: VAL VLString;
18   type : REF Type;
19 }
20
21 CLASS InhAttribute INHERITS
22 Attribute, AssignAttribute {
23   attributeName: VAL VLString;
24   type : REF Type;
25 }
```

For disambiguation, attributes (the symbols of V$\mathcal{LISA}$) will be noted as *Attribute*, when generally speaking of *InhAttribute*, *SyntAttribute* and *IntrinsicValueAttribute*; and attributes (the class attributes used to characterize the symbol) will be noted simply as "attributes".

As told before, each *Attribute* must have a type and a name in order to be properly characterized. That is what happens with the classes in Listing 7.6. Notice that the structure for all of the three type of *Attributes* is the same, and all of them have in common the inheritance of the class *Attribute*.

The attribute `type`, on each *Attribute* class, is declared using the reserved word REF. This means that the type of an *Attribute* is a reference for an existent element of class *Type*. Different from the reserved word SUB, which is used to denote that the symbol after it is a child of the correspondent class, the REF is used to refer to a symbol that must exist in any part of the whole derivation tree.

The specification of connections between the grammar symbols use extensively the REF mechanism. Listing 7.7 shows an example of one of the symbols used to connect other symbols. As can be observed, it has two attributes that refer to elements that must exist in the whole derivation tree, but not necessarily in the sub-tree from which it is root.

In class *InhConnection* it is stated that the attribute `from` (which stores the symbol that is source of this connection) is a reference to a symbol of the class *ISSymbol*. And the attribute `to` (which stores the symbol that is target of this connection) is a reference to a symbol of the class *InhAttribute*. This means that only symbols like *LeftSymbol* or *NonTerminal* can be connected to an *InhAttribute* via an *InhConnection*.

Listing 7.7: Attributes connection

```
1
2 CLASS InhConnection INHERITS
3 AttConnection {
4   from: REF ISSymbol;
5   to : REF InhAttribute;
6   anchorPoints: VAL VLList?
7               EDITWITH "None";
8 }
```

Observe, in Listing 7.7, the use of an abstract class to constraint the syntax of V$\mathcal{LISA}$. This constraint was introduced as **SC.2** in Chapter 4, and formally defined in Listing 4.4. Using the abstract class *ISSymbol* to group two classes, prevents the necessity of creating more than one option for the *InhConnection*. Otherwise it would create some non-determinism when `DEViL` was generating the visual language.

In `DEViL` specifications, each symbol must appear just once.

7.1.2 Replicating Structures

The last sentence may have a dual connotation.

In fact, the same symbol with the same name and structure may occur multiple times in the same specification. However, each declaration of the same symbol must occur in different namespaces.

In `DEViL`, namespaces are used to separate the base model (which represents the base abstract structure of the visual language) from sub-models. These sub-models replicate the base structure, but not necessarily reusing all the symbols defined in the base model.

When a sub-model is created, it is said that it derives from the base model. Then the constructor DERIVED is used to create such namespace [SKC07]. Its syntax is simple and can be found in line 2 of Listing 7.8: first is identified the name for the namespace and then is specified which symbol is the root for that derived sub-model; in this case, rules and *Semprod*, respectively. The derived model is a block delimited by curly-brackets, where the base model's symbols are redefined.

Inside the derived block, it can not be declared new symbols, that is, symbols that were not declared in the base model. However the symbols' structure can be modified.

Listing 7.8 shows the symbols *Semprod* and *Function* in the derived model which namespace was defined as `rules`. As can be observed and compared with the class in Listing 7.9, *Function* suffered no alteration to its base structure (besides the addition of the attribute `baseRef` used to refer the class from which it derives). However, class *Semprod* was incremented with a new attribute: `ruleName`; and the previous declared attribute `name` is now named `prodName`.

The advantages to derive a model is the possibility to replicate structures and maintain all of them continuously synchronized: any change that may occur on the base model, occurs immediately in the derived structure.

Visually, it reflects in the reutilization of drawings that can be more or less complicated.

Listing 7.8: Coupling of the model

```
1
2  DERIVED rules ROOT Semprod{
3
4  CLASS Semprod {
5   prodName: VAL VLString?
6             EDITWITH "None";
7   ruleName:  VAL VLString;
8   grammarElements: SUB AGElement*;
9   baseRef: REF ::Semprod
10            EDITWITH "None";
11 }
12
13 ...
14
15 CLASS Function INHERITS
16 AGElement {
17  returnType: REF ::Type;
18  functionName: VAL VLString
19                EDITWITH "Entry";
20  operation: VAL VLString
21                EDITWITH "Entry";
22  baseRef: REF ::Function?
23                EDITWITH "None";
24 }
25
26
27 }
```

Listing 7.9: Class Function in the base model

```
1
2
3
4
5
6
7
8
9
10
11 CLASS Function INHERITS
12 AGElement {
13   returnType: REF Type;
14   functionName: VAL VLString
15                 EDITWITH "Entry";
16   operation : VAL VLString
17                 EDITWITH "Entry";
18 }
19
20
21
22
23
24
25
26
27 .
```

In V$\mathcal{LISA}$case, the possibility for such replication of drawing structures is very important.

Imagine the drawing of a production with 4 elements in the `RHS`. Imagine that each symbol was a nonterminal and each one had associated one inherited and one synthesized attribute. Imagine that the semantic rules were all specified over the same drawing. As the number of attributes in the drawing is big, the number of computation rules would be also big. Then it would lead into a messy drawing even worst as if it was drawn in a paper.

An improvement, to avoid this clumsy mess, is to draw the production with the associated attributes, and then reuse this drawing as a template to draw each semantic rule in a separate copy of the template.

Since the manual redefinition of the production is completely out of question, the best solution is to derive the sub-structure from symbol *Semprod*.

And so it was done. Listings 7.3 and 7.8 depicts how it was obtained. The class *Semprod* in the base model has the attribute `computationRules` which refers to a list of *rules::Semprod* symbols. This means that in the sub-tree of each *Semprod* symbol, there may exist an undetermined number of *Semprod* symbols under the namespace *rules*.

As long as class *rules::Semprod* is the root of the derived structure, and the attribute `baseRef` points to symbol *Semprod* in the base model, then the whole structure of the latter will be replicated. Notice the necessity of removing the attribute `computationRules` from the *rules::Semprod*. Otherwise it could origin in infinite nested structures.

Special Computation Rule Symbols

But a problem remains. It is known that after deriving structures they maintain a tight synchronization. In some cases it is an advantage, but sometimes is a disadvantage. For V$\mathcal{LISA}$ it is a blend of both.

As symbol *Semprod* can group in its sub-tree any sub-class of *AGElements*, it can group specific symbols for the computation rules like *Function*, *Identity* and others. Thus, once a *Function* is drawn in the derived model, the same symbol will be created at the base model. This would lead to that initial messy drawing which must be avoided.

In case of removing the computation symbol from the base model, the same symbol will be removed from the derived model.

To solve the problem it is necessary to adjust the derived model. `DEViL` offers a list of standard adjustment schemes which can help fixing this problem.

The scheme needed to solve this problem is called *RemoveAbandoned*. It is applicable to object nodes and removes the object if there is no counterpart in the base model. Well, the default operation of this scheme is what is already happening: if the base-model object is removed, then the derived object is removed because the attribute `baseRef` points to nothing, i.e. is abandoned.

Listing 7.10 shows the declarations for the adjustment of the derived model. For example, "*Function*.removeAbandoned = 0" means that when the symbol on the base model, referred by the *rules::Function* symbol, is removed, it causes no deletion of the derived symbol.

Listing 7.10: Adjustment of the derived structure

```
1
2   Identity.removeAbandoned = 0;
3   FunctionArg.removeAbandoned = 0;
4   FunctionOut.removeAbandoned = 0;
5   Function.removeAbandoned = 0;
```

Listing 7.11: Event to delete a Function

```
1
2 proc edithook::create_Function {obj} {
3   edit::delete $obj
4 }
5 .
```

This way it is possible to associate an event to the action of creating a computation-specific symbol on the derived model in order to delete the counterpart symbol that is created in the base model. Listing 7.11 shows how simple the code for this event is. As long as an object is created (whether it is created on the base or on the derived model), it rises a *creation* event, so that it is possible to use the object that rose the event to delete it immediately.

The presented code is working at the creation of a *Function* symbol in the base model.

7.2 Environment Generation

V$\mathcal{LISA}$ was defined in the last section. That specification defines the syntactic constraints to avoid bad composition of the language visual elements. Besides these structural rules, the addition of attributes to the concrete symbols define the information to characterize each one.

However, that specification is only the basis for every processor to generate the core of the programming environment. It is an abstract structure that defines the syntax of the language. Nothing in that specification tells `DEViL` which is the visual aspect of the environment, or how can the symbols turn into the images defined in Figure 4.1 and 4.2.

The present section shows how to use the core specification of the visual language in order to instruct `DEViL`'s processor to create the interface and the visual icons for the visual `AG`'s symbols. Moreover it explains how predefined structures were created in order to create default templates and items to reuse, freeing the user of defining them.

7.2.1 The Interface Generation

Despite of its unfamiliar aspect, the notation of the specification defined for V$\mathcal{LISA}$ is the notation for an AG. Hence some attributes were attached to the symbols, but all of them are used to store information that characterizes the symbol and which is *synthesized* at edition-time. If they only get a value at edition-time, then they can be used for semantic check or even for code generation.

But these attributes are not enough to instruct DEViL to generate a visual programming environment. It is said that this environment is created automatically, but *automatically* is not the most correct word. Some work must be done under the hood in order to create such environment.

DEViL generates, automatically, many functionalities like save and load a specification or cut, copy and paste a symbol. However all of these are worthless functionalities for Visual$\mathcal{LISA}$ if there is not a space to draw the visual AG.

In fact, this space is offered by DEViL, but, by default, it is completely white with neither behavior nor interaction.

The attributes recalled moments ago are not used to define the behavior and interaction for the drawing area. But, regarding what was said in the last section, there are attributes used to store the position and size of the symbols[2].

These attributes are not created just because they seem to be needed. There are a major force that implies their presence in the abstract structure of the visual language: the Visual Pattern.

Visual Patterns, in DEViL, can be seen as a mask that enfolds a symbol and increments it with a visual behavior or layout. That is:

> *Visual patterns are reusable implementations of common representation concepts like lists, sets, line connections and forms* [SKC06].

All the patterns are already implemented by DEViL and are offered to ease the process of defining the layout of the visual language.

For this work it was used a set of basic patterns, very easy to understand, and easy to know when to apply. A summary of their behavior is shown below:

VPRootElement The element gets its position shifted to the upper left corner of the drawing area;

VPForm The element presents an image which can be seen as its layout for presenting the information associated to its attributes and so on;

VPFormElement The element must be child of a symbol which inherits the VPFrom. It means that the actual element is information that is presented in the form;

VPIdTextPrimitive The element (that must be text) is shown as text that identifies something. With this, two different elements that inherit this pattern can not have the same textual value;

VPTextPrimitive The element (that must be text) is shown as text, without the constraints of the previous pattern;

VPSimpleList The element behaves like a container for various elements. It is used when the order of the child elements is relevant;

[2]Notice that they were not present in the code excerpts shown in the last section, but all of them can be seen at Appendix B.

VPSimpleListElement The element must be child of a symbol that inherits the VP-SimpleList. This means that the actual element is an element that occurs inside the list implementation;

VPSet The behavior of the element is like the list pattern, but here the order is not relevant. So the element that inherits this pattern contains symbols not regarding to the order;

VPSetElement Like the VPFrom or the VPSimpleList, the element that inherits this pattern must be child of a symbol that inherits the VPSet. It means that the actual element is part of the set, and can be placed wherever it is wanted, but not overlapping other symbol;

VPConnectionArea The element behaves like a container for connection symbols;

VPConnectionEndPoint The element gains the capacity of being the start or end point of one connection line;

VPPolyConnection The element (that must be a line) is able to have multiple points that attach to the drawing area, allowing the possibility to create curves on the line;

VPConnectionFrom The element must be child of a symbol that inherits the VPPoly-Connection implementation. It means that the actual symbol behaves as the start point of the connection and

VPConnectionTo Is equal to the previous pattern, but the element plays the role of the end point of the connection.

Notice that only a simple description is given. The attributes that must be evaluated or those which are optional are not listed above, because such explanation is not on the scope of this work.

One or more visual patterns can be associated to a symbol of the grammar.

The grammar symbols that accept these visual patterns are the concrete ones. This happens because after the processing of the abstract structure, `DEViL` tailors a tree-grammar whose nodes are the concrete symbols of the base model.

Once the tree-grammar is available, the effort to create the visual behavior for the environment is equal to the effort of knowing the visual patterns and what is wanted to be visualized in each moment.

In fact, the first big effort is to choose how the language is viewed. It depends on what are the requirements for the language's visual aspect. Almost all requirements for this topic derive from the agreeable way of working with the language and the easiness of understanding what is being drawn.

For V$\mathcal{LISA}$, it was decided to create four distinct views in order to be nicer the usage of the environment and underlying language. These views separate the visual `AG` edition into four logical areas that approximate the visual to the textual edition. The four views are listed below:

Root View - *rootView* In this view, as the name suggests, are presented the foundations for the `AG` being drawn. Here is where the user can have access to the global definitions, and is able to declare productions.

Definitions View - *defsView* This view is where the definitions are developed. That is, this view offers the possibility of creating new lexemes, functions and data-types, and importing required modules or packages.

Productions View - *prodsView* This view is where a production is rigged up. That is, is where the icons for terminal, nonterminal and attribute symbols of the `AG` are composed with each other in order to create one production. This view is also where the computation rules can be declared[3].

Rules View - *ruleView* The last view, as the name suggests, is where the rules are defined. This view reuses the drawing of the production (developed in the latter view), as was seen in Section 7.1, and allows the language engineer to increment it with the operations to calculate the attributes value.

To define each of these views, there must be defined two distinct files. The first file is out of the scope of present Sub-section (will be exposed in Sub-section 7.2.2), the second file is where the semantic rules and the visual patterns for the symbols of the tree grammar are combined, in order to create the behavior of the language.

For each view, there must be chosen a root symbol and a set of symbols (based on that symbol structure) to be visible on that view. All the symbols enrolled must have visual patterns associated and some semantic rules that help to evaluate the patterns' attributes (when they exist).

Figure 7.2 shows the root symbol for the *rootView*. In fact, this element is exactly the *Root* symbol of the grammar. In the source code presented it is possible to see two symbol computations. The first refers to the *Root* symbol in general. The second refers to its attribute `name`.

```
1
2  SYMBOL rootView_Root INHERITS
3  VPRootElement,
4  VPForm
5  COMPUTE
6     SYNT.drawing =
7          ADDROF(rootViewDrawing);
8  END;
9
10 SYMBOL rootView_Root_name INHERITS
11 VPFormElement,
12 VPIdTextPrimitive
13 COMPUTE
14    SYNT.formElementName =
15                        "grammarName";
16 END;
```

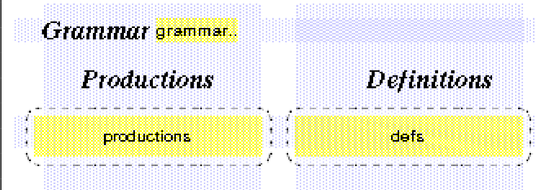


Figure 7.2: The RootView form declaration

As can be seen, both symbol definitions inherit some classes. These classes are the visual patterns.

The first symbol inherits the VPRootElement and the VPForm patterns. This indicates that this symbol will behave as the view's root element and in addiction will act as a form. Since it inherits the behavior of the form pattern, then it is necessary to compute the attribute `drawing` of the VPForm class. This attribute is used to assign an image to the form, using the constructor ADDROF.

Along with `DEViL`, the developers offer a tool that aids on the drawing of images or forms which can be associated to grammar symbols. In Figure 7.2, at the right, can be

[3]Notice that the declaration of a computation rule is different from its definition.

seen the image that supports the form for the *Root* symbol. In yellow are represented the containers which are used to display information about the symbol.

The second symbol inherits the VPFormElement and the VPIdTextPrimitive patterns. Since `name` is an attribute of symbol *Root*, then it makes sense that the value of such attribute could be displayed as information in its form. Inheriting the VPFormElement pattern, makes the attribute `name` a part of that form. This visual pattern implies the existence of a computation for the `formElementName` attribute. The value given for this attribute must be the name of a valid container in the form.

The first container in the image of the form in Figure 7.2 is labeled with `grammarName`[4], which was the same name given for the `formElementName` value.

The symbols used to build a view, can be reused in another views, with a completely different behavior. Listings 7.12 and 7.13 illustrate exactly this. The first refers to the *rootView* and the second refers to the *prodsView*. Both share the symbol *Semprod*. In the first, that symbol plays the role of a list element, but in the second, it acts like a form and is the root symbol of that view.

Listing 7.12: Use of *Semprod* at Root View

```
1
2
3
4 SYMBOL rootView_Semprod INHERITS
5 VPSimpleListElement ,
6 VPIdTextPrimitive
7 END;
8
9 .
```

Listing 7.13: Use of *Semprod* at Productions View

```
1
2 SYMBOL prodView_Semprod INHERITS
3 VPRootElement ,
4 VPForm ,
5 VPConnectionArea
6 COMPUTE
7   SYNT. drawing =
8         ADDROF( productionDrawing ) ;
9 END;
```

From what was shown above it is possible to realize that a view can not be defined without the visual patterns. In fact, it is very hard to rig up a view on `DEViL` if the visual patterns and their implications are not well studied and known. Because, as can be seen in the figures and listings presented above, there is nothing besides the visual patterns that instruct `DEViL`'s processor in order to create a visual environment with an easy-to-use layout.

7.2.2 Dock Buttons

In the last sub-section it was said that to define a view for the visual language two specification files are needed. The second file was already presented in that sub-section to specify used to develop V$\mathcal{LISA}$views. This sub-section will present the first file specification, and what are the repercussions in the visual programming environment.

Since V$\mathcal{LISA}$ has four distinct views, then four files to declare them were created. Listing 7.14 encodes the declaration of the *rootView* for V$\mathcal{LISA}$.

Listing 7.14: Declaration of the *rootView*.

```
1
2 VIEW rootView ROOT Root {
3   BUTTON IMAGE "img::btnSemprod"
4                  INSERTS Semprod
5                          INFO "Inserts a new Grammar Production";
6 }
```

[4] In the image part of the name is hidden because it did not fit into the container.

The content of these files can be more complex than the one presented. However, for `Visual`$\mathcal{LISA}$ was tried to keep it simple, with the basic specifications.

In these files, are declared the name and the root symbol for the view. The root symbol must exist in the tree-grammar tailored by `DEViL`, when processing the abstract structure. Line 2 of the code above shows exactly it. The constructor VIEW declares the name for the view, and the constructor ROOT declares the name of the symbol used as root of the view.

Moreover, these files are used to specify the interaction with the language icons, by means of creating buttons that are used to *drag* the icons into the drawing area.

These buttons are disposed in a palette (also known as dock) on the left side of the drawing area. They appear in the same order as they are created in the specification file. A button can be defined as a simple clickable area with text on it, or it can be used an image to define a visually attractive clickable area. In Listing 7.14, lines 3, 4 and 5 show the declaration of a button using the latter format. The constructors BUTTON and IMAGE and the file name preceded by "`img::`" declare that the clickable area in the dock (i.e. the button) is the image in the file "*btnSemprod.png*".

After the button is created, some action when it is clicked must be executed. The natural action is the insertion of one symbol in the drawing area. So that, the reserved word INSERTS followed by the name of a symbol in the tree-grammar (in the example the symbol is *Semprod*) is used to execute such action.

Other optional elements can be used in this definition. In Listing 7.14 it was used the reserved word INFO followed by a descriptive text, in order to create a tool-tip to appear whenever the mouse is over the button (for some instants of time).

Figure 7.3 depicts the *rootView*button.

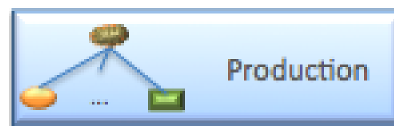
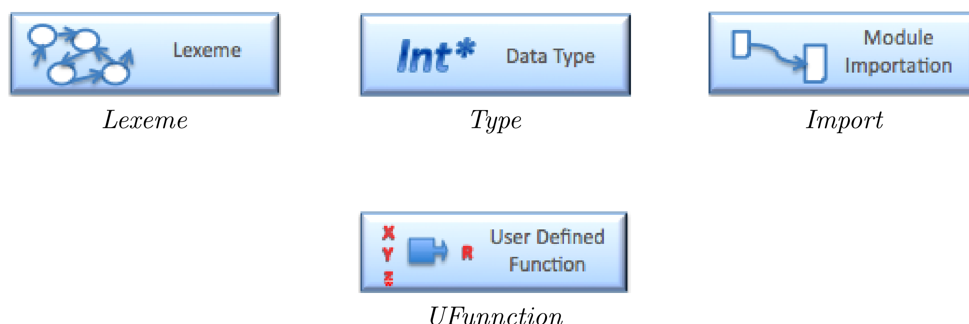


Figure 7.3: Dock button for *rootView*.

Figures 7.4, 7.5 and 7.6 show the buttons to appear in the dock for *defsView*, *prodsView* and *ruleView*docks, respectively. Below the image of each button, appears the name of the tree-grammar symbol that is created when the button is clicked and dragged into the drawing area.



Lexeme *Type* *Import*

UFunnction

Figure 7.4: Dock buttons for *defsView*

Notice the precaution of drawing intuitive images for the buttons along with intuitive labels. It is said that an image worth more than a thousand of words, but an image with a

label is always better. The images for the buttons are composed by figures with the same shape of the language icons, with the purpose of making the interaction more intuitive.

For instance, the button to create a *ComputationRule* (Fig. 7.5) take advantage from the shapes of symbols *Function*, *SyntAttribute*, *InhAttribute*, *FunctionArg* and *FunctionOut*, to design a simple computation of a value, like it should be done in the drawing area.

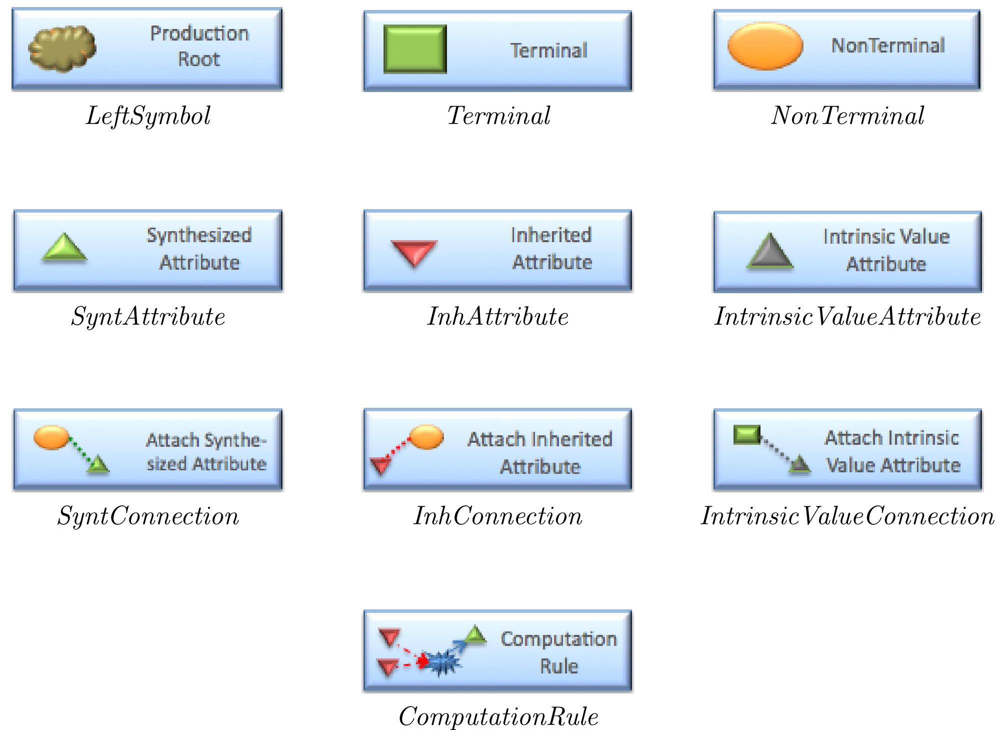


Figure 7.5: Dock buttons for *prodsView*

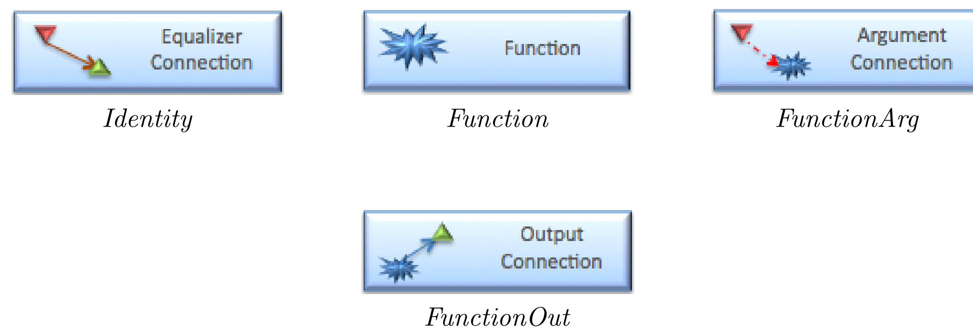


Figure 7.6: Dock buttons for *ruleView*

7.2.3 Predefined Structures

When a visual specification created in `Visual`$\mathcal{LISA}$ (or any other environment generated by `DEViL`) is saved, it is used an `XML` dialect to store the internal representation of the drawing. `DEViL` offers this simple way of saving specifications, rather than saving them in a binary format.

In fact, the load time of such XML file can be bigger than the one used to load a binary file. However, a binary file is neither human-readable nor human-writable. The characteristics of such binary files make harder the possibility of posterior manipulation or extensibility. With an XML file the possibilities to manipulate, extend, or do whatever is needed, is a complete open road. It is a backdoor to enter in the drawing specifications.

So that, DEViL lets users to define XML files (with the proper notation) that can be glued with the initial representation of a drawing, with the purpose of extending and completing it or even of creating default values. Obviously, it is only possible to do if the XML structure (after the combination of all the components with the initial representation) and the visual language defined match.

Listing 7.15 shows an example used in Visual$\mathcal{LISA}$ to create a predefined list of valid data types, which can be used in the declaration of an attribute.

Listing 7.15: An XML library of data types for Visual$\mathcal{LISA}$

```
1
2  <?xml version="1.0" encoding="UTF-8" ?>
3
4  <VLX editor="VisualLISA Editor" grammarVersion="1.0" editorVersion
       ="0.1.0" >
5    <Library id="::types" >
6        <types>
7            <Type id="::all::int" >
8                <name>all::int</name>
9                <value>int</value>
10           </Type>
11
12           ...
13
14       </types>
15   </Library>
16 </VLX>
```

The root element for these, lets call them *components*, is VLX. This element and its attributes is the gluing point of all the components and the main structure. They must match.

Then, the following element, is the symbol of the visual language that is being defined in the component. In the case of Listing 7.15, the symbol is *Library.* Regarding the visual language specification (Listing 7.1), this symbol is part of the *Root*'s RHS. So that this component glues with the structure defined for the *Root* symbol.

Inside the element *Library*, the several types are declared, always following the abstract structure defined in the visual language.

As told before, this component defines a list of valid data-types that can be used to declare the attributes for the visual AG.

However, these kind of components can be used to predefine structures with the purpose of creating a skeleton or a template to be completed.

In Visual$\mathcal{LISA}$, besides *Library*, *Definitions* symbol was created with that components facility. Unlike the component for the *Library*, the latter defines a skeleton with some predefined values. That is why, when a new AG is being created in Visual$\mathcal{LISA}$, the main window has already one occurrence of the *Definitions* symbol, and inside it, can be found one occurrence of a *Lexeme*.

7.3 Semantics Implementation

In the previous two sections was defined the abstract structure of V$\mathcal{LISA}$, and was given life to that structure by rigging up an environment, defining the visual aspect of the language's icons, and setting up button docks for user interaction.

With the definitions told above, the specification of an AG is possible, because the grammar symbols can be dragged and composed with each other. However, the composition is, at most, syntactically correct.

It was seen, in Chapter 5, that are many constraints (at semantic level) that must be hold to assure the complete correctness of the AG. So that, this section, reports the implementation of the module for semantics verification, taking advantage from DEViL's extensibility.

7.3.1 Semantics Implementation: Preview

DEViL gives a diversity of ways to perform semantics verification on the visual AG specification. Three of them are discussed here, aiming at finding the better way to implement such module.

DEViL uses the abstract grammar structure to perform syntactic and some semantic analysis at edition-time, without any complementary specification. Nonetheless this semantic analysis is, most of the times, not enough to correct the entire model. Fortunately, DEViL opens some doors to come in and extend the base semantics verification mechanism.

Editor Constraint Approach

It is possible to know which symbol is created, whenever a button is dragged to the drawing area of the editor. This happens because the mapping between a button and a symbol is deterministic. Thus DEViL offers edition events (also known as "edithooks") that are risen whenever an edition in the model occurs, depending on the context[5].

These events cause a change on the normal behavior of the editor, namely, on dock buttons. Since the events are triggered whenever, for instance a *LeftSymbol* is created or a *NonTerminal* is deleted, this approach is perfect to perform a semantic verification after an edition and, based on its results, execute some actions to revalidate the model.

The advantage of using this approach, which will be referred to as "*Editor Constraint Approach*" is that the verifications are performed as earlier as possible, avoiding the user to specify their AGs with semantic errors from the beginning.

Attribute Grammar Approach

Another way to perform semantic verification is to use the AG approach. The abstract structure of the language, enriched with some attributes, can be used to write the contextual conditions for V$\mathcal{LISA}$' specifications.

Besides attributes, built-in functions and predefined entities, like INCLUDING, CONSTITUENTS, IF or ORDER, can be used in order to combine them with properties, elegantly defined in *.pdl* files, available for each node of the tree-grammar structure.

With this approach, which will be referred to as "*Attribute Grammar Approach*", is guaranteed that every contextual condition can be implemented (with more or less effort). However there are some cons.

[5] An example of such events can be seen in Listing 7.11.

Despite of being two well separated concerns, the contextual conditions have to be specified in the same file where the code generation is specified. This augments the difficulty of code maintenance in the future.

But the truth is that in a textual AG specification, both contextual conditions and transformation rules are written in the same file as a mono-block piece of code.

Other problem beneath this approach is that the semantic verification can only be performed in the end, when the user asks the tool to generate code. Until there, the model can be specified with semantic errors.

Contextual Verification Approach

During the edition of the AG specification in Visual$\mathcal{LISA}$, an internal representation of that model is built. That representation is an Abstract Syntax Tree (AST).

As DEViL generates a tree and adds to it a sort of visitor pattern based on a contextual traverse function, it is possible to use the model's AST, in order to perform a traverse to that tree.

This traverse function, called *addCheck*, is the real extension for the basic semantic verification that DEViL offers for free. *addCheck* requires the specification of the context where it will be executed. The context refers to places in the tree. So a context is, for example, a *Semprod* symbol, or even the *Semprod*'s `name` attribute.

The returning of such function is a string. If the string is empty, then nothing occurs, otherwise a semantic error (with the text in the string) is risen for the context in which it was executed.

Differently from the *Attribute Grammar Approach*, a fine separation of concerns is attained, because the implementation of semantic analyzer module is completely separated from the code generation.

Moreover, with this approach, which will be referred to as "*Contextual Verification Approach*" the visual AG specification can be semantically verified not only before generating code but whenever it is desired.

Explained all the pros and cons about the three implementations, the contextual conditions defined in Chaper 5 will be grouped to see where their implementation fit better. Table 7.1 shows that distribution. To refer to each constraint, their numbers (e.g **CRC.1**) will be used.

As can be seen in Table 7.1, the first column lists the constraints that are already implemented by DEViL. The other columns have some constraints in common. This means that the constraint, for instance, **PC.2** can be implemented using all the approaches.

The second and third columns don't list all the constraints because its implementation is hard, and DEViL does not offer structures capable of supporting all the necessary computations.

But relegate responsibilities for other systems rather then Visual$\mathcal{LISA}$ is not a good policy because this way users are being stimulated either to abandon Visual$\mathcal{LISA}$ or to perform some necessary changes in the code which Visual$\mathcal{LISA}$ will generate.

On account of that, all the constraints defined must be implemented; Looking at the table is easy to see that the fourth column lists all the constraints (except those that are implemented by DEViL). So with this approach every constraints can be implemented, even the more complex ones.

Based on Table 7.1, the choice of the approach is obvious. The *Contetual Verification Approach* will be used to implement all the contextual conditions.

Implemented by DEViL	**Editor Constraint Approach**	**Attribute Grammar Approach**	**Contextual Verification Approach**
PC.1 **PC.6** **PC.7** **PC.9** **PC.10** **PC.11** **CRC.1** **CRC.8** **CRC.9** **CRC.10**	**PC.2** **PC.3** **PC.4** **PC.5** **PC.8** **PC.12**	**PC.2** **PC.3** **PC.4** **PC.5** **PC.8** **CRC.2** **CRC.3** **CRC.4** **CRC.5** **CRC.6** **CRC.7** **CRC.11**	**PC.2** **PC.3** **PC.4** **PC.5** **PC.8** **PC.12** **CRC.2** **CRC.3** **CRC.4** **CRC.5** **CRC.6** **CRC.7** **CRC.11** **CRC.12**

Table 7.1: Distribution of Contextual Conditions for Possible Implementation Approaches

Next sub-section discusses, in more detail, some topics concerning the implementation using such approach.

7.3.2 Semantics Implementation: The Concretization

In the last section was chosen an approach for implementing the contextual conditions defined at Chapter 5. The approach, as explained before, is based on a function that traverses the `AST`, that `DEViL` builds at edition-time, and executes some code when recognizing a context of the tree.

The traverse function, *addCheck*, is implemented under `tcl` programming language. Therefore, is possible to use the `tcl` data-types structures, like lists, or hash-tables, in order to ease the process.

Using this approach, may appear that there is no notion of attributes and semantic rules to calculate them; and the formal specification of the constraints done, in Section 5.2, are worthless. But that is not true at all. Instead of attributes there are variables, instead of semantic rules there are operations and instead of formal specifications there are concrete test conditions. At the end, there is a mapping between all of these entities.

Listing 7.16 shows the code written to implement the contextual condition **PC.2**.

That code actuates at *Semprod* context; and, as can be seen in the referred listing, the *Semprod* symbol is being added a check.

This piece of code illustrates the above referred mapping: the attribute `nLHS` of the symbol *Semprod* was converted into a variable with the same name; the computation (which was not presented when defined the attribute) is a direct query to the `AST` based on the context, which is given by the $obj variable; the formal specification (i.e. the invariant that must hold) was transformed into its negation in order to rise appropriated semantic

error messages.

Listing 7.16: Implementation of Contextual Condition **PC.2**

```
1
2  checkutil::addCheck Semprod {
3   set nLHS [llength [c::getList {$obj.grammarElements.CHILDREN[LeftSymbol]}]]
4   set name [c::get {$obj.name.VALUE}]
5
6   if { $nLHS == 0 } {
7    return "Production '$name' must have one Root symbol!"
8   } elseif {$nLHS > 1} {
9    return "Production '$name' must have only one Root symbol!"
10  }
11
12  return ""
13 }
```

The constraint implemented above is really basic and easy to understand. The actual difficulty of implementing the entire set of constraints lies on the assignment of values to the variables (attributes). This happens because it is necessary to know the internal structure of the model, i.e., the `AST`, to specify the query path. However, even that difficulty vanishes, because `DEViL` shows the tree of the model being specified, which is very good to learn the structure. Figure 7.7 shows an example of that tree.

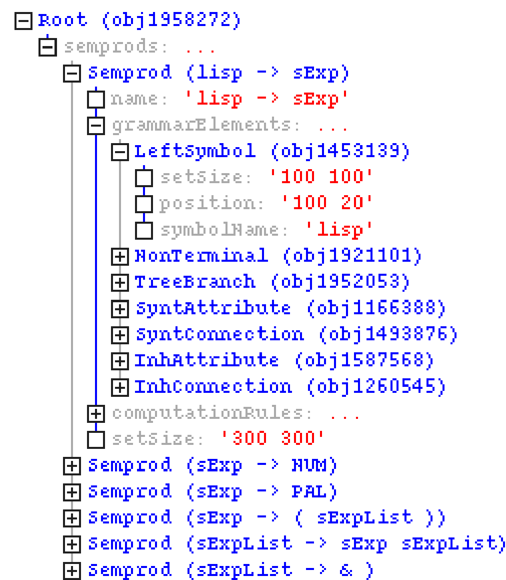


Figure 7.7: An example of a model's tree, produced by `DEViL`

Besides the queries, it is necessary to gather information about every symbol that appears on the model being specified, in order to check the consistency of the symbols. This means that an identifier table (as presented in the Introduction) must be defined and populated.

Hash-tables are usually used to implement these Identifier Tables, because of the easiness of associating values to keys and the fast lookup strategies. Moreover, the identifier tables are constructed and used in the symbol's context, therefore the usage of *addCheck* to implement such table is obvious.

Not all the symbols in VLISA are relevant to store their information in an identifier table, because some of them are just connectors. Only those directly related with the se-

mantic constraints defined in Chapter 5 are relevant: *LeftSymbol*, *NonTerminal*, *Terminal*, *InhAttribute*, *SyntAttribute*, *IntrinsicValueAttribute* and *Function*.

The relevant symbols of VLISA were separated into four classes. For each class, different data needs to be stored.

Below, the data items that are stored in the Identifier Table, are explained.

id is the name of the symbol.

class is the class of the symbol. It can be one of Terminal, Nonterminal, Attribute or Function.

type is the data-type of the symbol. For *Function* it is the data-type of the return value.

args corresponds to the number of arguments of a *Function*.

Table 7.2 defines, for each identifier class, the information that is stored in the Identifier Table along with the symbols pertaining to that class.

Table 7.2: Information in the Identifier Table by Class of Symbol

	Symbols	id	class	type	args
Terminal	*Terminal*	√	√		
Nonterminal	*NonTerminal* *LeftSymbol*	√	√		
Attribute	*InhAttribute* *SyntAttribute* *IntrinsicValueAttribute*	√	√	√	
Function	*Function*	√	√	√	√

Listing 7.17 shows the code for the construction and the usage of the Identifier Table, in the context of a *LeftSymbol*.

Listing 7.17: Example of the Implementation of the Identifier Table

```
1  checkutil::addCheck LeftSymbol {
2   global idTable
3   set id [c::get {$obj.symbolName.VALUE}]
4   set class "NonTerminal"
5
6   if {[lsearch [array names idTable] $id] != -1} {
7    if {$idTable($id,class) != $class} {
8     return "Symbol '$id' was previously defined as '$idTable($id,class)'"
9    }
10  } else {
11   set idTable($id) $id
12   set idTable($id,class) $class
13  }
14
15  return ""
16 }
```

The identifier table is defined as a global variable because it needs to be seen in every context of the `AST`. *LeftSymbol* is a simple symbol in the sense that there is not many information associated to it relevant for posterior calculus. Only its name (defined as variable `id`, in Listing 7.17) and its class are relevant to verify its semantics.

In the referred code, before inserting the new item in the table, some verifications are performed. The algorithm is simple: *i*) Is the symbol with that name already in the table? *ii*) If no, then the symbol's information must be inserted; otherwise *iii*) Is the class of the existing symbol different from the class of the new symbol? *iv*) If yes, then the symbol is the same and the model is correct; otherwise a semantic error must be thrown, because a symbol with the same name and different class is being used.

The implementation of the other contextual conditions follows the same scheme presented in Listing 7.16; also the construction of the Identifier Table at the context of each one of the other symbols is based on the example of Listing 7.17, for *LeftSymbol*.

7.4 Tree-Grammar Traversing for Code Generation

With the modules implemented in the last sections, the modeling of an AG is completely defined and its semantics can be verified with the purpose of achieving a correct specification.

In this section, the implementation of the last module for Visual$\mathcal{LISA}$, that is, the translation of the iconic specification into a textual one, is documented.

In traditional AG approaches, the translation of the information in the several attributes into any kind of output, is done by traversing the intermediate structure and applying the translation rules in the precise context.

In DEViL, as it uses AGs to specify visual languages and generate environments for them, the process of generating code from the drawings is similar to the related before. Unlike what happens with the semantics, DEViLdoes not offer a function to traverse the AST and to generate text as output. For this task, DEViL encourages the use of the *Attribute Grammar Approach* defined in the section before.

The code generation task, is always done after the semantic verification of the drawing AG. For that reason, the *Attribute Grammar Approach* can be applied. Remember that the biggest difficulty on using it was the definition of structures to store the information. For the code generation task, the structures used are simple, and can be implemented, for instance, using auxiliary functions.

Although it is not mandatory, the development of this task is twofold: first there can be specified templates in order to structure out the output — this step is not mandatory; the second corresponds to the specification of semantic computations, in order to evaluate the value to the attributes used in the translation.

7.4.1 Structuring the Output with Templates

As seen in Chapter 6, the structures of $\mathcal{LISA}$ and $\mathcal{X}$AGra are very well defined. There are some parts that are fixed, and others that change depending on the case that is being specified.

These static parts are easily identified. For $\mathcal{LISA}$, the template can contemplate the parts that are shown in Listing 6.1. On the other hand, for $\mathcal{X}$AGra, the skeleton is obviously defined by the opening and closure of the XML tags that define the notation.

DEViL facilitates the construction of templates because it offers the possibility to use a template language denominated *ptg* or its improved version *iptg*.

Listings 7.18 and 7.19 show a part of the template defined for $\mathcal{LISA}$ and for $\mathcal{X}$AGra, respectively. The template language used was the *iptg*, because it is simpler to define,

read and understand when comparing with its previous version. Therefore *iptg* is easy to maintain.

Listing 7.18: Part of the Template Definition for $\mathcal{LISA}$

```
1
2 lisaRule(pName, lhs, rhs, comps):
3   rule [pName] {
4     [lhs] ::= [rhs] compute {
5       [comps]
6     };
7   }
8
9 .
```

Listing 7.19: Part of the Template Definition for $\mathcal{X}$AGra

```
1
2 xmlProcessSymbols(t, nt, st):
3   <terminals>
4     [t]
5   </terminals>
6   <nonterminals>
7     [nt]
8   </nonterminals>
9   <start nt="[st]" />
```

In fact, the templates are function-based. Each function defines a little part of the overall structure, and not the complete one. This happens because these functions are translated into the `C` language for later integration in the second step of the translation task. For example, the declaration of **`lisaRule`**, in Listing 7.18, is transformed into the function with name `PTGlisaRule` and same number of parameters.

The parameters of the template functions are used in the body in between brackets ([and]). In the functions showed above, the parameters have the type `PTGNode`. However any other type, as long as `DEViL` recognizes it, can be used.

There are many important rules when specifying these templates. But their explanation is not part of this document.

Besides $\mathcal{LISA}$ and $\mathcal{X}$`AGra`, it was decided to translate the visual specification into a different output, a `BNF` specification in order to transform the pictographic representation into the traditional one.

The `BNF` notation is very simple and the need for a template is almost none. However a simple template was defined; two functions are sufficient to define all the template. Listing 7.20 presents the code for such template.

Listing 7.20: Template Definition for `BNF` notation

```
1
2           bnfMain(listProds):
3                   [listProds]
4
5           bnfProd(lhs, rhs):
6                   [lhs] -> [rhs]
```

7.4.2 The Problem of Being Visual

In visual specifications there is not a pre-established notion of order. The figures can be placed in any position in the drawing area; as long they respect the syntactic imposed order, the position they are placed in is not relevant.

Regarding `V`$\mathcal{LISA}$, specifying a production by placing the icon *LeftSymbol* below or above the icons correspondent to the `RHS`, is irrelevant, because the sentence will still represent a production. The same happens with the attributes of a symbol; the *InhAttribute* may appear at the right or at the left of the correspondent symbol. As long they are connected by an *InhConnection* the model is correct.

This means that, in visual languages, the syntax and the position of the icons are two well separated issues, and in many cases, the latter is completely irrelevant.

But notice that the order of the symbols in the RHS of a production is really relevant. So that in V$\mathcal{LISA}$ the position of the symbols connected to the *LeftSymbol* is also important; their order should be retrieved from the drawings.

Aiming at solving this problem, two approaches were taken into account, but only one was reasonable. Both the approaches are presented below.

Temporal Order

As told before, DEViL builds a tree at edition-time. Then it stores the symbols, regarding a temporal order. So it is possible to know what symbol was created before or after another.

A possible way of obtaining an order for the RHS of a production, is to pick the symbols in the correct order from the tree that DEViL offers. The approach is simple and the cost of development is really low.

But the time is something abstract, and therefore, impossible to see. Visual languages are all about seeing icon compositions and understanding it exclusively by looking at it. In a simple drawing, the time is not perceptible; it is not possible to apprehend the temporal order of symbol creation with a simple visualization of the drawing.

Figure 7.8 *a*) depicts a production where the LHS is A, and the RHS is the set with the symbols B, C, D, E, F. These symbols were created (temporally) in the same order as the alphabetic order of their names. But, as can be seen, their position does not follow a logical order capable of being apprehended just by looking at the specification. If the order of their creation did not match the alfabetic-order of their names, then it would be impossible to acquire such order.

If the production was

$$A \rightarrow BCEDF$$

then the RHS symbols would be created by the following order: B, C, E, D, F. But a different user would not understand why was that the order, because the figure can not transmit the order of creation.

Positional Order

Rather than having an order given by something that is not perceptible from the visual specification, it is desirable to obtain such order directly from that specification.

The majority of the people in the world is used to read a text from left to right, top to bottom. This is a logical way of reading texts and also images. That is, if the order of something is given by this simple rule, it is possible to apprehend the implicit order just by looking at that something. Then if this something is a set of RHS symbols in a production specified with V$\mathcal{LISA}$, then the set of symbols would easily be transformed in an ordered sequence.

Visual$\mathcal{LISA}$ establishes the drawing area as a bi-dimensional space. Then, implicitly, exists two oriented axes which determine the height (Y-axe) and the width (X-axe) of the space, converting it into a Cartesian Coordinate System. All the symbols of V$\mathcal{LISA}$ have implicitly a coordinate (position), given by a pair (x, y). Therefore, regarding these coordinates, it is possible to define an order for the symbols.

Using the conventional *reading rule*, that was referred above, and applying it to the specification in Figure 7.8 *a*), taking into account the symbol coordinates, the production would be read as following

$$A \rightarrow BCFDE$$

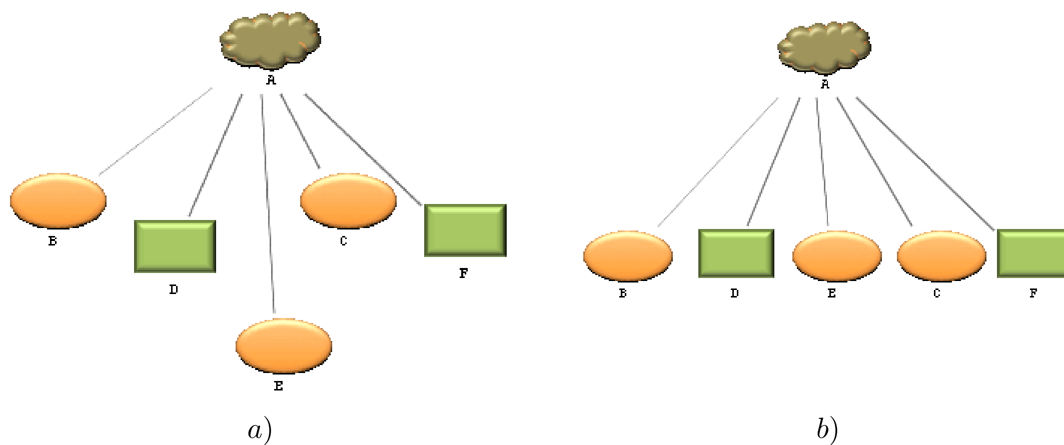
because symbols B and C are in the first *line* but B occurs first; F is placed a little bit above the D but in the *line* below the first one; and E is in the last *line*.

Obviously that this is a better way of reading the production when comparing with the temporal order, but it is not yet perfect. Instead of computing the order by looking to the X and Y axes, it is possible to have a better and faster ordering by looking only to the X-axe. This point of view corresponds to morphing the second part of the coordinate of a symbol (the y part) into a constant λ. Figure 7.8 *b*) shows the result of such ordering.

With the last approach, the production would be read as

$$A \rightarrow BDECF$$

This way, the user can specify the productions only with the notion that the `RHS` symbols are ordered from left to right, no matter their height.



a) *b*)

Figure 7.8: Production Specification. *a*) *had-oc* position of `RHS` symbols; *b*) `RHS` symbols aligned through imaginary X-axe.

A simplification of the algorithm used to implement this ordering is in Listing 7.21.

Listing 7.21: Ordering Algorithm

```
1
2  L ← SELECT RHS FROM production;
3  foreach symbol ∈ L
4    Lx_i ← (symbol.position.x, symbol);
5
6  Lxo ← ORDER Lx::(x,s) BY x ASC;
7
8  foreach pair ∈ Lxo
9    Lo_i ← pair.symbol;
10
11 return Lo;
```

In line 2, L stores the list of all the `RHS` symbols that pertain to the production. These symbols can be retrieved from the AST that devil builds. In line 3 and 4, a pair composed by the x part of the coordinate of each symbol, and the symbol itself is associated to each position of Lx. In line 6, Lxo stores the result of ordering the list Lx (of the form (x, s) where x is the position on the X-axe and s the symbol) by the value of x and in an ascendent form, that is, from the lowest to the higher value. In line 8 and 9, Lo stores the list of all the symbols from Lxo, and line 11 return this list. Lo represents the list of symbols already ordered by the position in the X-axe.

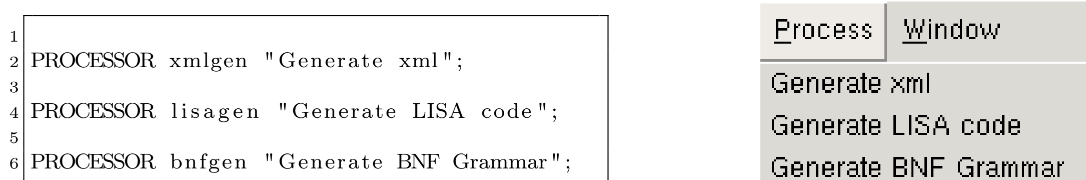
Besides the problem addressed in the last sub-section, others were found; some approaches to solve them were considered for reasoning about the one to follow. Some of these problems were only sub-problems of the one discussed here; for instance, the problem of numbering the repetition of symbols in a production. See Listing 6.2 to remember the motivation for this problem.

7.4.3 The Code Generation Process

By default, `DEViL` does not generate any kind of output in textual format, therefore no button for that purpose is created. However it reserves a space, in the graphical environment, for such button.

In order to have a button (or more), which action is to generate code, it is necessary to specify a model comparable to the models specified to rig up the views of the language (see Section 7.2).

Figure 7.9 presents the code used by `DEViL` to create three buttons and to associate them to the module of the code generation, and shows the generated buttons in the graphical interface.



```
1
2 PROCESSOR xmlgen "Generate xml";
3
4 PROCESSOR lisagen "Generate LISA code";
5
6 PROCESSOR bnfgen "Generate BNF Grammar";
```


Figure 7.9: Processing Buttons

Regarding the code in Fig. 7.9, it is possible to see how easy is to create buttons and to associate them to a code generation module. `DEViL` constructor PROCESSOR determines that a button is to be declared. The first word, after the constructor, is the name of the module where is specified the code generation. Finally, the third chunk is only the name to appear to identify the button. As can be seen in the drop-down menu, at the right in Figure 7.9, there are a direct mapping between the order and the names of the buttons on the menu and the order and the names declared in the source code.

The definition of the processing buttons is a crucial step, because it associates the declaration of the interface aspects to the logical actions of the code generation.

As told in the preamble of the present section, the code generation module can be defined resorting to the *Attribute Grammars Approach*. In this approach, needed attributes can be associated to the symbols of the tree-grammar, in which context the semantic rules to compute the value for the attributes are declared. Using this approach, is still possible to resort to auxiliary functions and to those originated from the template declarations.

Listing 7.22 presents the complete `AG` specification to the translation module to generate `BNF` notation from the visual `AG` drawn. This is a simple example, but addresses all the points needed to understand the strategy used to develop the translations for $\mathcal{LISA}$ and for $\mathcal{X}$`AGra` notations, which are much more complex than the presented here.

The algorithm for translating the `AG` modeled in `Visual`$\mathcal{LISA}$ into `BNF` notation can be described succinctly:

1. collect all the productions. This is done by, in each symbol *Semprod*:

   (a) collect the `LHS` symbol;

(b) collect and order the RHS symbols;

(c) transform these collections in the BNF notation, using the template;

2. show the code generated;

Listing 7.22: Specification of the BNF notation translator

```
1
2  SYMBOL bnfgen_Root: bnfCode: PTGNode;
3  SYMBOL bnfgen_Root
4  COMPUTE
5    SYNT.bnfCode = PTGbnfMain(CONSTITUENTS bnfgen_Semprod.bnfCode
6                      WITH(PTGNode, PTGNewLineSeq, IDENTICAL, PTGNull));
7
8    PTGOutWindow("Name", THIS.bnfCode);
9  END;
10
11 SYMBOL bnfgen_Semprod: bnfLHS : PTGNode;
12 SYMBOL bnfgen_Semprod: bnfRHS : PTGNode;
13 SYMBOL bnfgen_Semprod: bnfCode : PTGNode;
14 SYMBOL bnfgen_Semprod
15 COMPUTE
16   SYNT.bnfLHS = CONSTITUENTS bnfgen_LeftSymbol.pers_symbolName
17                  WITH(PTGNode, PTGNewLineSeq, PTGAsIs, PTGNull);
18
19   SYNT.bnfRHS = PTGAsIs(VLString(SELECT(vlList("printBNFOrderedRHSElements",
20                                     THIS.objId),
21                               eval()
22                         )));
23
24   SYNT.bnfCode = PTGbnfProd(THIS.bnfLHS, THIS.bnfRHS);
25
26 END;
```

As can be seen, the name of the module **bnfgen**, used as namespace in this specification, is the same name that was used when declaring the association button-module in Figure 7.9.

Lines 2 and 11 to 13, are used to associate attributes to the symbols, defining their names and their type. The attributes used here are all synthesized. In line 24, is a sample of the usage of templates. The template function **bnfProd** declared in Listing 7.20 was converted into a C language function called **PTGbnfProd**, and is used directly to compute the value for the attribute **bnfCode**. In line 19, is used an auxiliary function, named **printBNFOrderedRHSElements**, in order to compute the value for the attribute **bnfRHS**. This function was declared with **tcl** language, so its use is not direct as the use of C functions. That function is used to order all the symbols on the RHS of the production, resorting to the algorithm sketched in Listing 7.21.

To show the result to the users, **DEViL** allows both the use of a window, or the creation of files in the file system. But the latter is not as directly as the former. Visual*LISA* shows the output in a window. In the code above, the instruction in line 8 explains how this is done. Notice that it is a pre-defined template function that receives two parameters: the name of the window, and the text to be displayed in such window.

Figure 7.10 and 7.11 show an example of the code generated for BNF notation and for *LISA* specification, respectively.

An example of the generated code for *X*AGra is not presented here, for the sake of space and readability. In Appendix C this example is shown.

```
school -> students
students -> student students
students -> student
student -> name age
```

Figure 7.10: BNF Code Generated

```
language schoolGra {

   lexicon{
      Name    [A-Z][a-z]+
      Age    [0-9]+
      ignore    [\0x09\0x0A\0x0D\ ]+
   }

   attributes
      int  SCHOOL.sum;
      int  STUDENTS.sum;
      int  STUDENT.age;



   rule school {
      SCHOOL ::=  STUDENTS compute {
         SCHOOL.sum = STUDENTS.sum;
      };
   }

   rule students_1 {
      STUDENTS ::=  STUDENT STUDENTS compute {
         STUDENTS.sum = STUDENT.age + STUDENTS[1].sum;
      };
   }

   rule students_2 {
      STUDENTS ::=  STUDENT compute {
         STUDENTS.sum = STUDENT.age;
      };
   }

   rule student {
      STUDENT ::=  #Name #Age compute {
         STUDENT.age = Integer.parseInt(#Age.value());
      };
   }
```

Figure 7.11: $\mathcal{LISA}$ Code Generated

7.5 Implementation Summary

At this stage, the implementation of Visual$\mathcal{LISA}$ is completely over. The bigger problems were discussed here and the pros and cons were presented. Through out this chapter, was devoted some importance to the files that must be created to have a complete DEViL specification. The dependency relations between some of these files were addressed, however a global overview of such dependencies was not shown.

Figure 7.12 presents a complete web of dependencies between the files that make part of the Visual$\mathcal{LISA}$ specification.

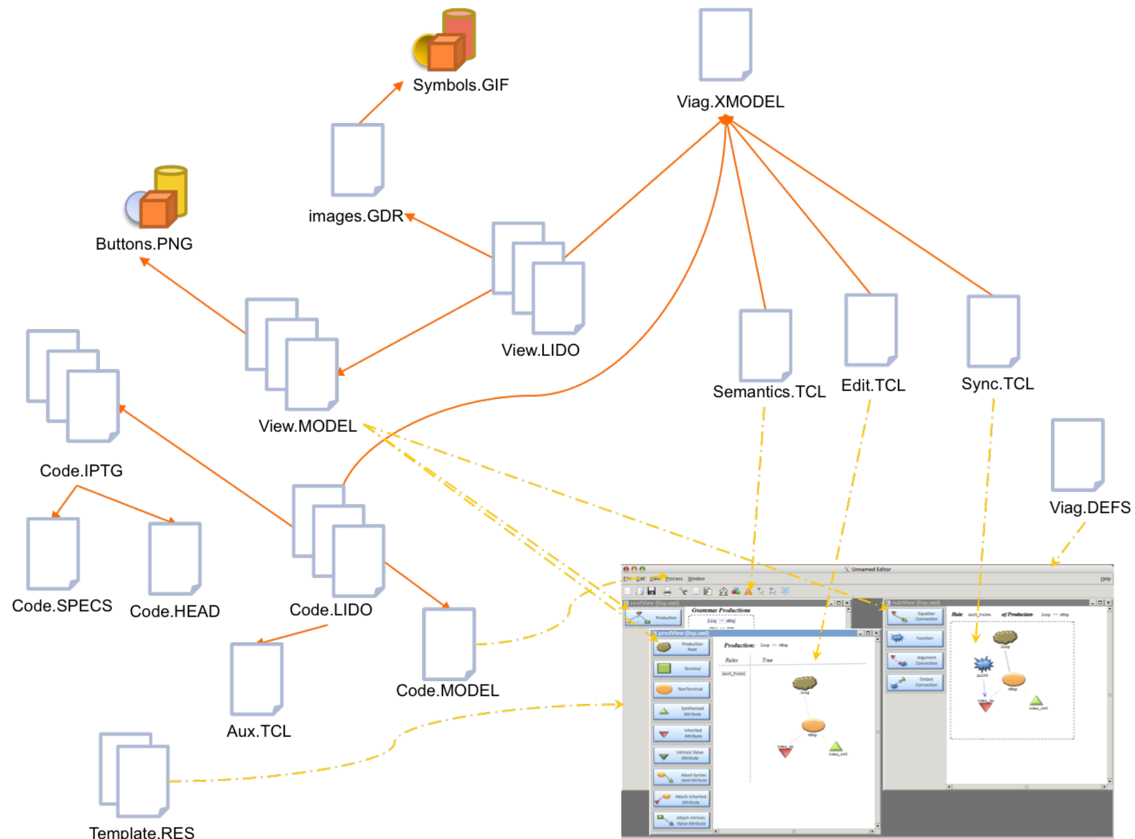


Figure 7.12: Complete Overview of the File Dependencies and Relation with the Environment Generated

The orange full-arrows determine the dependencies between files. For example, the files to specify the views of the language depend on the Viag.XMODEL file (as almost every files do), because the latter is where the abstract syntax of the language is declared. In yellow dashed-arrows, are represented the relations between the files and what is generated. For instance, the arrow from Semantics.TCL points to a little triangular button, in the menu-bar of the graphical interface, because this is the button associated with the semantics module.

Chapter 8

User's Guide

In this chapter a detailed explanation of how to use Visual$\mathcal{LISA}$ is presented.

An example will be followed to illustrate this explanation[1]. The example consists in the specification of a language to describe a list of students ($\mathcal{L}_{students}$), where for each student is provided his name and age. This language should be specified by an AG (henceforth called Students Grammar). The AG should be specified in a visual language (V$\mathcal{LISA}$). Moreover, it must specify a processor to accept sentences of $\mathcal{L}_{students}$ and to sum up the ages of the students. For instance, taking a concrete sentence like the one shown in Listing 8.1, the output of the processor will be 63.

Listing 8.1: Sentence of $\mathcal{L}_{students}$

```
1 Peter   12
2 John    13
3 Maria   12
4 Mark    12
5 Ann     14
```

Listing 8.2 shows Students Grammar. The context-free grammar is written in BNF notation; and the semantic rules associated to each production are written in an abstract way.

Listing 8.2: The Students Grammar

```
1 P1: Students → Student Students
2          {Students.sum = Students.sum + Student.age}
3 P2: Students → Student
4          {Students.sum = Student.age}
5 P3: Student → name age
6          {Student.age = age.value}
```

The lexical definition of the terminal symbols is:
name: [a-zA-Z]+ and age: [0-9]+.

The outcome from this exercise, that is, the AG specification in Visual$\mathcal{LISA}$, appears in Appendix D.

The chapter is organized in sections to make it a true User's Guide, clear and simple to consult. In Figure 8.1 is presented a graph that induces the dependencies between the several sections, subsections and so on. With this graph, the user can easily find a reasonable way of reading what he is looking for without loosing the context.

[1] The proposed example may not address every aspects that need to be explained. Whenever these cases occur, a generic approach will be used.

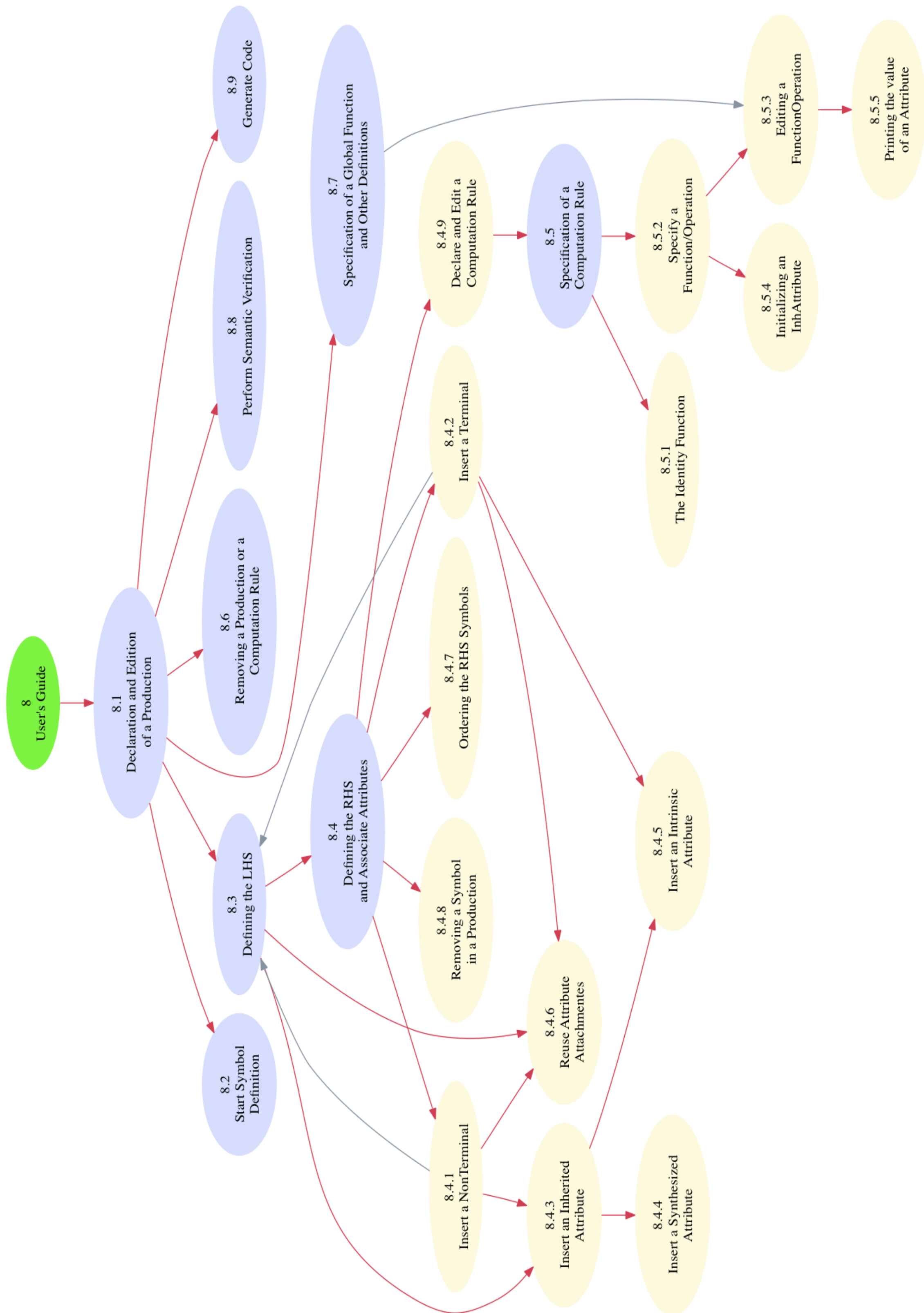


Figure 8.1: User Manual's Dependency Graph.

8.1 Declaration and Edition of a Production

Visual$\mathcal{LISA}$ is production oriented. That is, the basic-piece of an AG in Visual$\mathcal{LISA}$ is the specification of a production. Each production is independent from the others.

The main view (Figure 8.2) presents two sections. One for declaring productions, and another to specify global definitions.

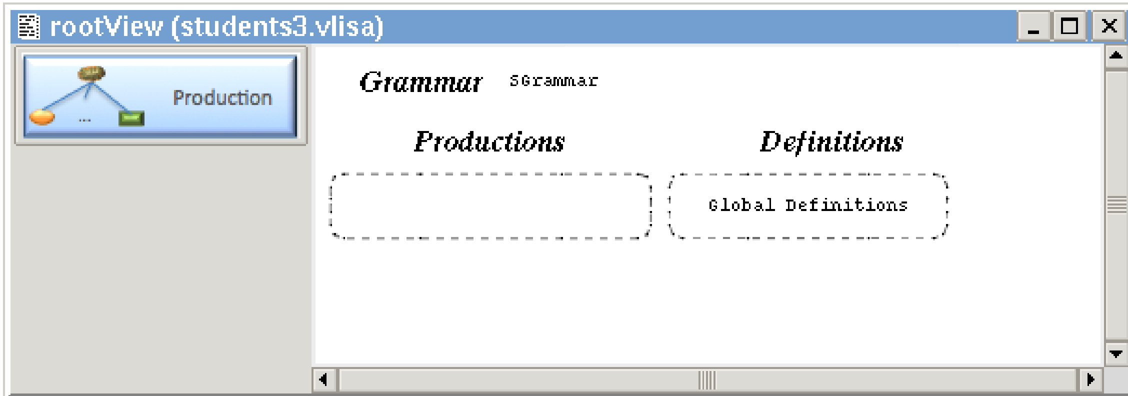


Figure 8.2: New Grammar Specification View: *rootView*.

The first step on defining the AG is declaring the productions. They can be declared all at once without specifying their layout, or one at a time by specifying the layout before declaring a new production. The user is free to choose the approach to follow.

To declare a new production *The user must toggle the button at the left, and drag the mouse to the productions section. When the mouse is over that section, a black bar appears to show where the production will be declared (Figure 8.3.a). When that bar is in the wanted position, a left-click is sufficient to create the new production (Figure 8.3.b).*

Figure 8.3: Declaring a Production: (*a*) Setting the Position in the List; (*b*) Production Declared

Tip It is not necessary to have the mouse right over the productions section. In this case, Visual$\mathcal{LISA}$ is deterministic and knows exactly where to place the production. The user must only specify its position in the list, by dragging the mouse up and down.

To edit a Production *The user should click twice over the desired production declaration. It will open a new window where the layout of the production must be sketched (Figure 8.4).*

The window opened is also divided into two sections. The thinest part is to list the computation rules associated to a production. The largest is used to support the layout

definition of the production. Moreover it has a title with the fixed text `Production` where the name of the production goes.

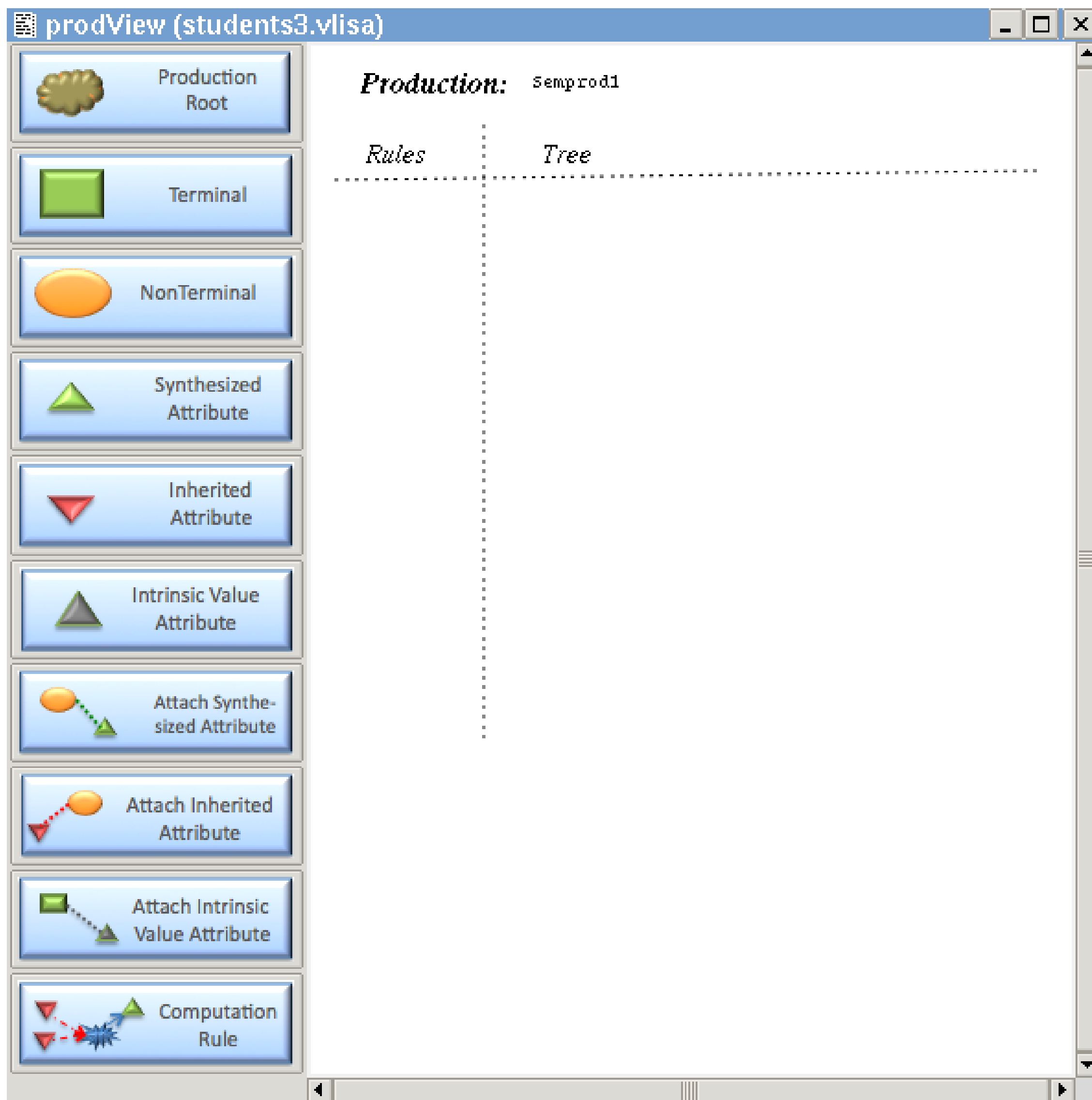


Figure 8.4: Production Specification View: *prodsView*

To change the name of a production *The user should click twice over the title of the production, as referred above. A box like the one in Figure 8.5 will appear. The name of the production should be written in the text box.*

Tip It is suggested that the name of a production follow a syntax close to the `BNF` notation. It would bring more legibility to the list of productions, in the *rootView*, since the name given to the production is the name that appears in the list.

The definition of the layout of a production requires the specification of the `LHS`, the `RHS`, the association of attributes and the definition of the start symbol. The next sections will address these topics.

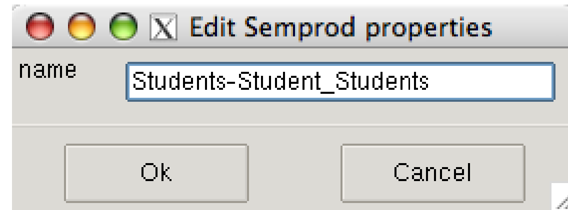


Figure 8.5: Form to Edit the Properties of a Production

8.2 Start Symbol Definition

In Visual*LISA* there is not a nice way to define the *start symbol* of the grammar. In fact this is hidden from the user. So that, this section is important to understand the mechanism of declaring the start symbol.

Regarding the list of productions in Figure 8.3.*b* the user can notice that a notion of order can be retrieved from there.

To define the start symbol *The user should place, on the top of that list, the production which* **LHS** *symbol is the desired start symbol of the grammar.*

Tip When the production on the top of that list is not the desired one, then it can be dragged down, so that the list is rearranged. Of course this is not only applied to the first production. The rearrangement of the list is possible in any direction and using any production.

8.3 Defining the LHS

There are two different ways of defining the LHS of a production. The first one is a simple drag-and-drop task, the second one is an user-friendly action that occurs in a determined context.

To add the LHS symbol (1) *The user should toggle the button in Figure 8.6.a and drag the mouse to the area used to define the layout (labeled with* **Tree** *in Figure 8.4).*

Figure 8.6: (*a*) Button to insert a new Production Root Symbol; (*b*) The Production Root Icon Inserted.

To add the LHS symbol (2) *The user should insert a Terminal or a NonTerminal symbol before inserting the* **LHS***, like was referred in the approach (1). This action creates a root symbol for the production and connects automatically it with the Terminal or NonTerminal that raised that action.*

To change the name of LHS symbol *The user should click twice on the icon. A box similar to the one shown in Figure 8.5 will pop-up. The name of the symbol should be written in the text box.*

Tip The name of the *NonTerminal* symbol does not have any syntax constraint. But the user is encouraged to write the first letter as a capital.

Example The example, running in background, will addresses the first approach. Figure 8.7 presents the definition of the LHS of the production P_1 from the Students Grammar.

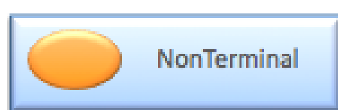
Figure 8.7: Definition of the LHS of production P_1 from the Students Grammar.

8.4 Defining the RHS and Associate Attributes

The definition of a RHS of a production is, in Visual$\mathcal{LISA}$, the construction of a tree, where the LHS of the production is the root of the tree. The association of attributes is the decoration of that tree.

8.4.1 Insert a *NonTerminal*

To insert a *NonTerminal* *The user should toggle the button in Figure 8.8.a and drag the mouse to the layout specification area of prodsView, inserting the icon in Figure 8.8.b.*



(*a*)

(*b*)

Figure 8.8: (*a*) Button to insert a new *NonTerminal* Symbol; (*b*) The *NonTerminal* Icon Inserted.

To change the name of *NonTerminal* symbol *The user should click twice on the icon. A box similar to the one shown in Figure 8.5 will pop-up. The name of the symbol should be written in the text box.*

Tip The name of the *NonTerminal* symbol does not have any syntax constraint. But the user is encouraged to write the first letter as a capital.

Example Figure 8.9 presents the complete layout of the BNF notation for production P_1 from the Students Grammar.

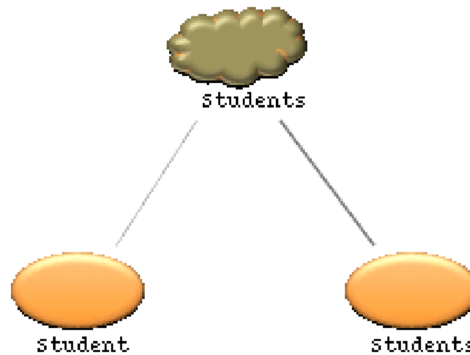


Figure 8.9: Definition of the LHS and RHS of production P_1 from the Students Grammar.

8.4.2 Insert a Terminal

To insert a *Terminal* *The user should toggle the button in Figure 8.10.a and drag the mouse to the layout specification area of prodsView, inserting the icon in Figure 8.10.b.*

Figure 8.10: (*a*) Button to insert a new *Terminal* Symbol; (*b*) The *Terminal* Icon Inserted.

To change the name of *Terminal* symbol *The user should click twice on the icon. A box like the one in Figure 8.11 will pop-up. The name of the symbol should be written in upper text box.*

Tip The name of the *Terminal* symbol does not have any syntax constraint. But the user is encouraged to write it with a lower first letter.

To change the RE of *Terminal* symbol *The user should click twice on the icon. A box like the one in Figure 8.11 will pop-up. The RE for that symbol should be written in the lower text box.*

Tip (1) The RE should follow the syntax used for in the target AG meta-language. In $\mathcal{LISA}$ is used the POSIX ERE standard to write the REs.

Tip (2) When the *Terminal* is a single character like {, for instance, the user should specify a name like *lBrace*, and the RE with the symbol that it will represent; in this case the symbol {.

Edit Terminal properties
symbolName age
regExp [0-9]+
Ok Cancel

Figure 8.11: Form to Edit the Properties of a *Terminal*.

Tip (3) When the *Terminal* refers to a reserved word `BEGIN`, for instance, the user is encouraged to write the name of the *Terminal* exactly as the reserved word (forgetting about the conventions set to write a *Terminal* name). However the user may choose between this approach and the natural one: write the name in lower letters, and use the `RE` field to write the reserved word `BEGIN`, in this case.

Tip (4) When the *Terminal* is repeated (the name is equal to other *Terminal* already defined), it is not necessary to reproduce the `RE`. Only the first occurrence (regarding temporal order) is obliged to have the `RE`. Although this, the user is encouraged to write the same `RE` in every occurrence, for a better comprehension of the grammar.

Example Figure 8.12 presents the complete layout of the `BNF` notation for production P_3 from the Students Grammar.

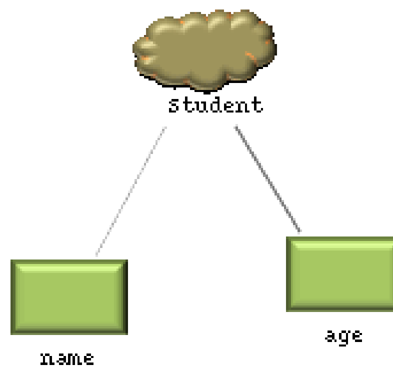


Figure 8.12: Definition of the `LHS` and `RHS` of production P_3 from the Students Grammar.

8.4.3 Insert an Inherited Attribute

To insert an *InhAttribute* *The user should toggle the button in Figure 8.13.a and drag the mouse to the layout specification area of prodsView, inserting the icon in Figure 8.13.b.*

To attach an *InhAttribute*to a symbol *The user should toggle the button in Figure 8.14.a and drag the mouse to the layout specification area of prodsView. A red pointed line (Figure 8.14.b.) will appear in the drawing area. When selecting this line, the user*

Figure 8.13: (*a*) Button to insert a new *InhAttribute* Symbol; (*b*) The *InhAttribute* Icon Inserted.

will see two handlers (the blue circles on the edges of the line in Figure 8.14.b) in the start (leftmost) and ending (rightmost) points of the line.

The leftmost point should be dragged up to a NonTerminal or a **LHS** *symbol; the rightmost point should be dragged up to an InhAttribute created before.*

Figure 8.14: (*a*) Button to attach an *InhAttribute*symbol to a *NonTerminal* or to the production's root; (*b*) The *InhConnection* Icon Inserted.

Tip The user is encouraged to place the *InhAttribute*symbols on the left of the associated symbol, in order to maintain a logic readability of the grammar.

To change the name of *InhAttribute* symbol *The user should click twice on the icon. A box like the one in Figure 8.15 will pop-up. The name of the symbol should be written in upper text box.*

To set the data-type of the attribute *The user should click twice on the icon. A box like the one in Figure 8.15 will pop-up. The data-type of the attribute should be chosen from the list presented.*

Edit InhAttribute properties
attributeName
type
all::byte
all::char
all::double
all::float
all::int
Ok
Cancel

Figure 8.15: Form to Edit the Properties of an *InhAttribute*.

Tip The list presented is not limited. It can be extended with new user-defined data-types (See Section 8.7 for more details).

The running example does not address the use of inherited attributes. But the attachment of an inherited attribute is similar to the attachment of a synthesized attribute. The latter will be exposed in next section.

8.4.4 Insert a Synthesized Attribute

To insert an *SyntAttribute* *The user should toggle the button in Figure 8.16.a and drag the mouse to the layout specification area of prodsView, inserting the icon in Figure 8.16.b.*

Figure 8.16: (*a*) Button to insert a new *SyntAttribute* Symbol; (*b*) The *SyntAttribute* Icon Inserted.

To attach an *SyntAttribute* to a symbol *The user should toggle the button in Figure 8.17.a and drag the mouse to the layout specification area of prodsView. A green pointed line (Figure 8.17.b.) will appear in the drawing area. When selecting this line, the user will see two handlers (the blue circles on the edges of the line in Figure 8.17.b) in the start (leftmost) and ending (rightmost) points of the line.*

The leftmost point should be dragged up to a NonTerminal or a `LHS` *symbol; the rightmost point should be dragged up to a SyntAttribute created before.*

Figure 8.17: (*a*) Button to attach an *SyntAttribute*symbol to a *NonTerminal* or to the production's root; (*b*) The *SyntConnection* Icon Inserted.

Tip The user is encouraged to place the *SyntAttribute* symbols on the right of the associated symbol, in order to maintain a logic readability of the grammar.

To change the name of *SyntAttribute* symbol *The user should click twice on the icon. A box similar to the one in Figure 8.15 will pop-up. The name of the symbol should be written in upper text box.*

To set the data-type of the attribute *The user should click twice on the icon. A box similar to the one in Figure 8.15 will pop-up. The data-type of the attribute should be chosen from the list presented.*

Example The base layout of production P_1 was already defined. It misses the association of synthesized attributes. From the formal specification in Listing 8.2, the symbols on P_1 decorated with attributes are: *Students* (both `LHS` and *NonTerminal* on the `RHS`) and *Student*. The attribute `sum` is associated to symbols *Students* (both `LHS` and *NonTerminal* on the `RHS`); the attribute `age` is associated to symbol *Student*. Figure 8.18 shows the result of decorating the production tree.

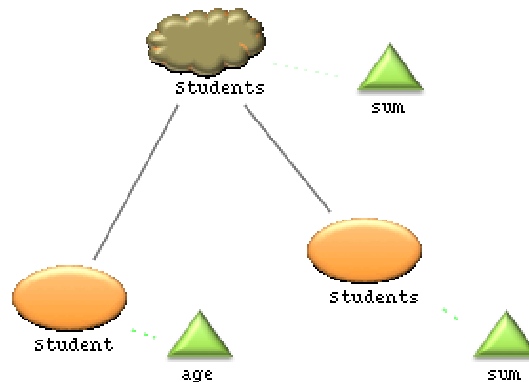


Figure 8.18: Decoration of production P_1 with Synthesized Attributes.

8.4.5 Insert an *IntrinsicValueAttribute*

To insert an *IntrinsicValueAttribute* *The user should toggle the button in Figure 8.19.a and drag the mouse to the layout specification area of prodsView, inserting the icon in Figure 8.19.b.*

(*a*) (*b*)

Figure 8.19: (*a*) Button to insert a new *IntrinsicValueAttribute* Symbol; (*b*) The *IntrinsicValueAttribute* Icon Inserted.

To attach an *IntrinsicValueAttribute* to a symbol *The user should toggle the button in Figure 8.20.a and drag the mouse to the layout specification area of prodsView. A black pointed line (Figure 8.20.b.) will appear in the drawing area. When selecting this line, the user will see two handlers (the blue circles on the edges of the line in Figure 8.20.b) in the start (leftmost) and ending (rightmost) points of the line.*

The leftmost point should be dragged up to a Terminal symbol; the rightmost point should be dragged up to an IntrinsicValueAttribute created before.

Tip The user is encouraged to place the *IntrinsicValueAttribute* symbols on the right of the associated *Terminal* symbol, in order to maintain a logic readability of the grammar.

(a) (b)

Figure 8.20: (a) Button to attach an *IntrinsicValueAttribute*symbol to a *Terminal*; (b) The *IntrinsicValueConnection* Icon Inserted.

To change the name of *IntrinsicValueAttribute* symbol *The user should click twice on the icon. A box similar to the one in Figure 8.15 will pop-up. The name of the symbol should be written in the upper text box.*

Tip The name of the intrinsic attribute is deeply related with the syntax of the target AG meta-laguage. The name of such attribute must be the name of the selector or method used in the target meta-language to retrieve the property of the terminal symbol.

In the example (Figure 8.21), it was used the name *value()*, because it is the method used in $\mathcal{LISA}$ to access the intrinsic textual value of an attribute.

To set the data-type of the attribute *The user should click twice on the icon. A box similar to the one in Figure 8.15 will pop-up. The data-type of the attribute should be chosen from the list presented.*

Tip As happens for the name, the data-type of the intrinsic attribute depends also from the target meta-language.

In the example below, the data-type used is a *String*, because in $\mathcal{LISA}$ that method to retrieve the intrinsic textual value returns a *String*.

Example For this example is used the production P_3. The symbol *age* has an intrinsic value that stores the string with the number that is parsed. This is the value synthesized from that *Terminal* symbol. Figure 8.21 shows the result of decorating the production P_3 tree.

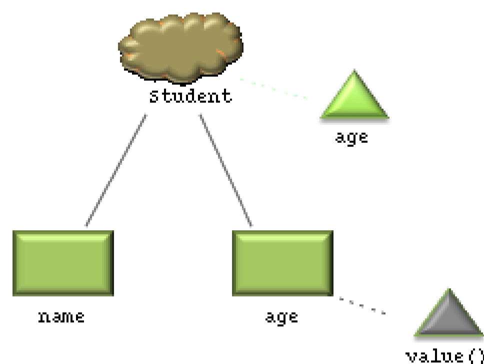


Figure 8.21: Decoration of production P_3 with an Intrinsic Attribute.

8.4.6 Reuse Attribute Attachments

When a grammar has multiple choices for a symbol (*NonTerminal* or *LeftSymbol*) or a symbol is used several times (*NonTerminal*, *LeftSymbol* or *Terminal*), and that symbol has one or more attributes, it is boring the task of defining, in every production, the same attributes to the same symbols. So that, it was thought a semi-automatic way to reduce this effort.

To automatically reuse the attributes of a symbol *The user should write, in the symbol properties box, the same name of the symbol from where he desires to copy the attributes.*

Tip 1 To synchronize the associated attributes to the several symbols with the same name, it is required to open the symbol's properties box and, without doing anything else, click in the `OK` button. Notice, though, that this task should be performed over the symbol that is desired to re-sync.

Tip 2 When performing the re-synchronization of the attributes of a symbol, the user should pay attention to the fact that new *objects* are created. This means that if the attribute was referenced in a computation rule, and the symbol to which it is attached was refreshed, then the reference in the computation rule is lost.

8.4.7 Ordering the RHS symbols

The order of the symbols in the RHS of a production is an important issue. In Visual$\mathcal{LISA}$ it was thought a method and constraint to visually inform the ordering of the symbols in the RHS of a production.

To order the RHS symbols *The user should drag the several symbols and place them from left to right in an horizontal alignment (does not matter their vertical position).*

Example In Figure 8.9 the symbols of the RHS are ordered from left to right: *Student* and *Students*; The symbols are exactly over the same imaginary horizontal line, but in Figure 8.12 the symbols are slightly unaligned, however their vertical position is not relevant for this ordering mechanism; so they are ordered as follows: *name* and *age*.

8.4.8 Removing a Symbol in a Production

In a production there is the notion of an hierarchy regarding the importance of a symbol.

The *LeftSymbol* (LHS) is the most important symbol. So that deleting it, forced the deletion of all its sub-elements.

Next in the hierarchy come the *Terminal* and *NonTerminal* symbols. When removing one of these symbols, all of its sub-elements (the attributes) are forced to be deleted. In addition, the *TreeBranch* symbol used to connect the symbol to the LHSis also removed.

Last standing in the hierarchy are the attributes (the three types). When removing them, since they have no sub-elements, only themselves and the edges that connect them to a symbol are removed.

To remove a symbol *The user can select it and either delete it with the* `delete` *button on its keyboard, or right-click above the icon and select the delete operation. A dialogue box like the one presented in Figure 8.22 will appear. Then the user only have to confirm or cancel the deletion.*

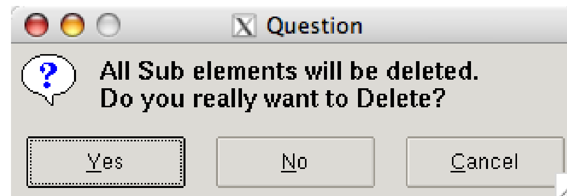


Figure 8.22: Dialogue Box to Confirm/Refuse the Deletion of a Symbol.

8.5 Specification of a Computation Rule

The specification of a computation rule is easy. However there are some issues that the user must be aware of. The next sub-sections will address them and explain how to specify the computation rules taking advantage from the reuse of the layout of the production to which the rule is associated.

8.5.1 Declare and Edit a Computation Rule

The declaration of a Computation rule is an operation similar to that performed to declare a production.

To declare a Computation Rule *The user should toggle the button in Figure 8.23.a and drag the mouse to the* rules area *(the thinest section in Figure 8.4) of prodsView, inserting a new element in that list of rules (Figure 8.23.b).*

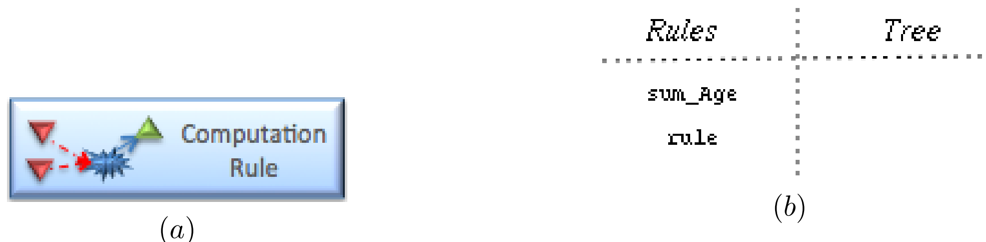


Figure 8.23: (*a*) Button to insert a new Computation Rule; (*b*) The List of Rules Associated to a Production.

To Edit a Computation Rule *The user should click twice over a rule title in the list of rules. A new window (Figure 8.24), named ruleView, is opened. There, the semantic rule can be specified over the syntactic layout of the production, which is automatically drawn*[2].

[2]Notice that in the context of Figure 8.24, it was not yet defined the production layout, so in the presented window there is not a base structure.

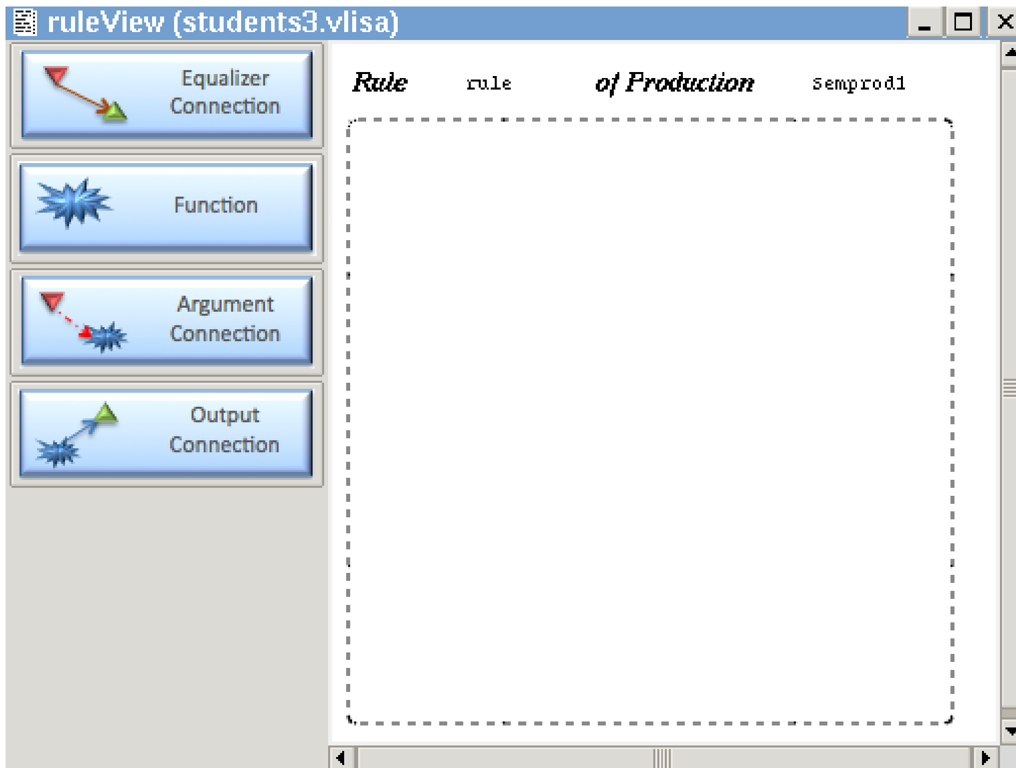


Figure 8.24: A New Computation Rule Window: *ruleView*.

Tip The user is encouraged to use one *ruleView* window to write a single semantic rule, in order to avoid visual confusion. However the user is free to specify n semantic rules in a single window (where n is the number of *Out* attributes of a production).

The new window is characterized by a small description on the top, where the title (or name) of the rule is between the fixed bolded-words `Rule` and `of Production`. The latter identifies the production to which the rule is associated. Below that description figures a drawing area. In this area, the base structure of the production, is automatically drawn, so that the user only has to increment it with symbols to define semantic rules.

To change the name of a rule *The user should click twice on the rule title. A box, like the one presented in Figure 8.25, opens. The name of the rule must be written in the text box field.*

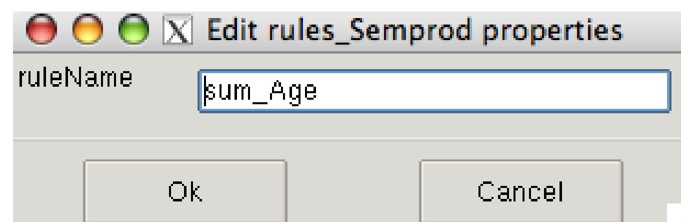


Figure 8.25: Form to Edit the Properties of a Computation Rule.

Tip Despite the default name *rule* being already associated to the title, the user is forced to write a new one. However the new name can be *rule* again.

8.5.2 The Identity Function

For those not used with this nomenclature, the identity is the function which the output is equal to the input. In mathematics the '=' operator is an example of an identity function. In Visual$\mathcal{LISA}$, the identity is regarded as an operation that assigns to an attribute the value of another attribute, without any extra transformation.

To insert and connect an identity function *The user should toggle the button in Figure 8.26.a and drag the mouse to the drawing area in the ruleView, inserting the brown full-arrow in Figure 8.26.b. This arrow his handled like the attribute connecters shown above. The leftmost handler must be connected to the source attribute and the rightmost to the assigned attribute.*

Figure 8.26: (*a*) Button to insert a new Identity Function; (*b*) The Icon that Represents the Identity Function.

Tip Despite the possibility of connecting the identity arrow to any attribute, there are an important constraint that must be followed: *the source attribute must belong to the In attributes of the production, and the target attribute must pertain to the Out attributes of the production*[3].

Example In the running example, production P_2 was formalized with a semantic computation that uses the identity function, that is, the value of the attribute *Studentes.sum* is equal to the attribute *Student.age*. Figure 8.27 shows the reuse of the template of the production P_2 to increment it with the identity function, specifying the semantic rule.

The visual semantic representation depicted in Figure 8.27 can be readed or formally translated as follows:

$$Students.sum = Student.age$$

8.5.3 Specify a Function/Operation

The usage of functions, in Visual$\mathcal{LISA}$, is probably the most complex concept, because it has many restrictions and tricks. In this sub-section the insertion of a new *Function* will be addressed, and in the next three sections other specific aspects will be explained.

To insert a *Function* *The user should toggle the button in Figure 8.28.a and drag the mouse to the drawing area in the ruleView, inserting the blue star-shaped icon in Figure 8.28.b.*

[3]See the Introductory chapter of this document for more about *In* and *Out* attributes, and see Chapter 5 to see the formal definitions of these constraints.

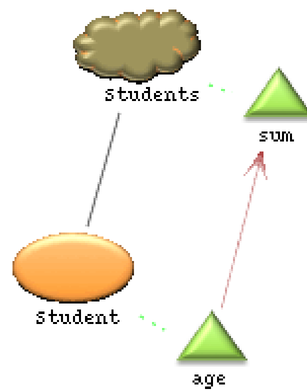
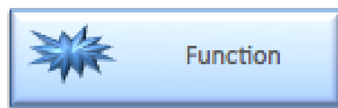


Figure 8.27: Computation Rule Using the Identity Function.



(*a*)

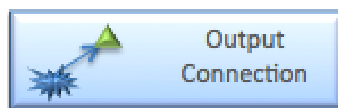
(*b*)

Figure 8.28: (*a*) Button to Insert a *Function*; (*b*) The Icon that Represents the *Function*.

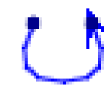
Tip A *Function*, in Visual$\mathcal{LISA}$, must always have an output value. That is, its return type is never void. However, it may have zero or more arguments.

To specify an output *The user should toggle the button in Figure 8.29.a and drag the mouse to the drawing area in the ruleView, inserting the blue full-arrow icon (FunctionOutsymbol) in Figure 8.29.b.*

This arrow has two handlers. The leftmost handler should be dragged and dropped on the Function symbol; the rightmost handler should be dragged and droped on the Out attribute.



(*a*)



(*b*)

Figure 8.29: (*a*) Button to Insert an Output Connector; (*b*) The Icon that Represents the Output Connector.

To specify an argument *The user should toggle the button in Figure 8.30.a and drag the mouse to the drawing area in the ruleView, inserting the red dashed-arrow icon (FunctionArgsymbol) in Figure 8.30.b.*

This arrow has two handlers. The leftmost handler should be dragged and dropped on the In attribute; the rightmost handler should be dragged and droped on the Function symbol.

(a) (b)

Figure 8.30: (a) Button to Insert an Argument Connector; (b) The Icon that Represents the Argument Connector.

8.5.4 Editing a Function/Operation

The tricks of the *Function* symbol are all hidden in its properties. This section explains the several relations between the properties of the *Function* symbol and the connections with the attributes.

To edit a *Function* *The user should click twice on the Function symbol. A window with its properties (like the one in Figure 8.31) will open.*

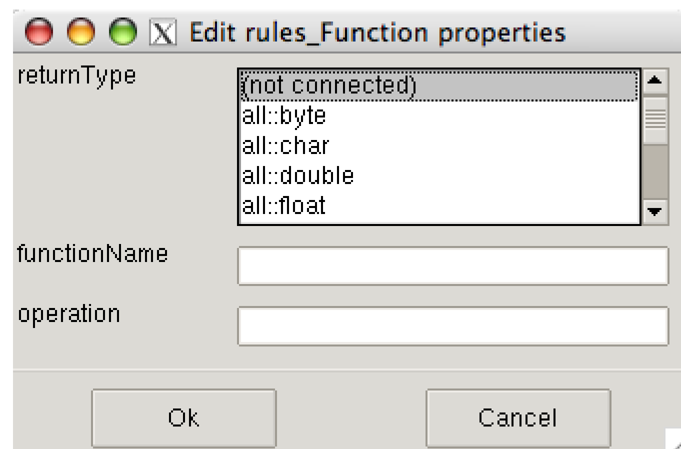


Figure 8.31: Window with the Properties of a *Function*.

This window has three fields. The first `returnType` is used to choose the type of the output value. The second `functionName` is used to give a name to the function — this name should be clear to understand what the function does. The third, and the most important one, is used to specify the mathematical operation that transforms the parameters into the final output.

The next words are devoted to the last field. Instead of generic approaches, examples will be used to show the relations between the code that should be written and visual aspects related with the arguments of the function.

Example (1) - Specifying Basic Operations Figure 8.32 relates the properties window of the *Function* symbol used to define the computation rule for production P_1 from the Students Grammar, and its visual aspect.

Notice how the operation code was written: *$1 + $2*. The $i symbols refer to the arguments of the function, where i is the number of the argument ordered by the creation of the *FunctionArg* symbol[4]. The value of i must be higher than 0.

[4]The arguments order is not visually perceptible, because it relies on the temporal order. It is planed to

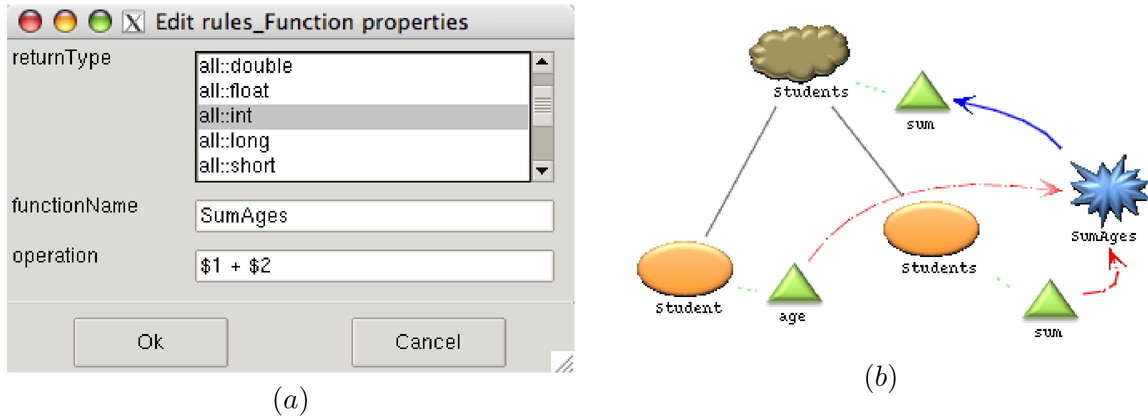


Figure 8.32: (*a*) Properties of the Sum Function; (*b*) The Visual Syntax of the Sum Operation of the production P_1 of the Students Grammar.

In this case, the $1 refers to the attribute *Students.age*. The $2 refers to the *Students.sum*. But the order here is irrelevant because the sum operation is commutative.

Tip (1) With this visual schema, it is possible to have operations like

$$\$1 + \$2 - (\$2 * \$1) \ldots$$

This happens because only two arguments are necessary to perform this calculation.

If a $3 was specified, then an error would occur.

If a third argument was specified, but a $3 was not included in the operation field, an error would also occur.

Tip (2) The user is encouraged to first write the operation code, regarding the number of the arguments, and only after it create the connections by the correct order. This way the number of mistakes decrease.

Example (2) - Using User Defined Functions In this example (Figure 8.33) it is not shown the properties window. Instead, it is presented a feature that enables the visualization of the function properties without opening that window.

The tooltip presented in the figure appears when the mouse is over the *Function* symbol. Similar tooltips will appear when the mouse is over the attributes, since the attributes are frozen in the *ruleView*, that is, their properties can not be changed, and without the tooltips, the data-type of the attribute would not be seen, unless the user change to the *prodsView* window.

In this example, an user defined function (see Section 8.7 for more details) was used to transform the input value. The operation entry in the tooltip, *toInt($1)*, shows the code in the operation field of the properties window. The fact of invoking $1, implies the specification of an argument.

8.5.5 Initializing an *InhAttribute*

The initialization of an inherited attribute is a tricky task. Some users may be used to initialize them on textual definitions before the parsing start, that is, in the preamble of

implement a new ordering algorithm for these connections, so that the user can have a more real perception of the semantic rule.

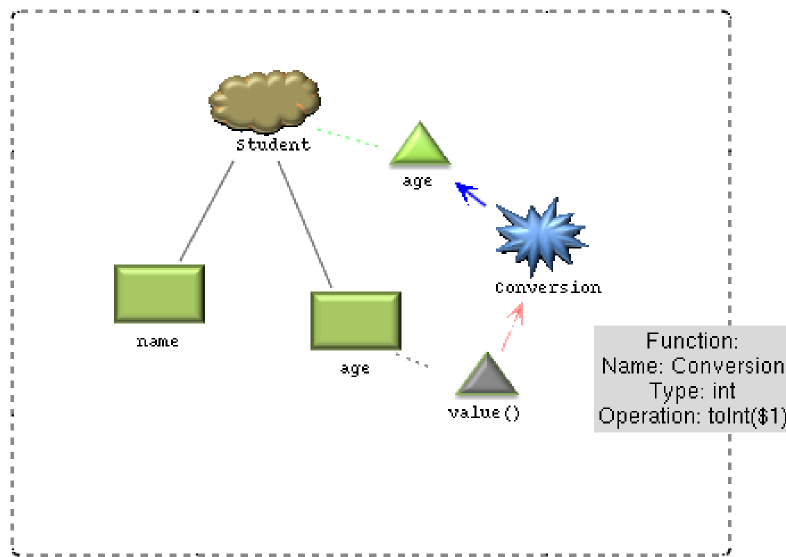


Figure 8.33: The Visual Syntax of the Computation Rule for production P_3 of Students Grammar.

the root production, and so that they are able to associate inherited attribute to the root symbol.

However, in `Visual`$\mathcal{LISA}$, it is not like that. The root symbol can not be associated to an inherited attribute. This happens because of the correctness of the attribute grammar that is built with this tool. In the Introduction it was recalled that a production has *In* and *Out* attributes, and in Section 5 it was constrained that an *In* element of a production can not be the output attribute of a *Function*. Since the initialization of an inherited attribute is done by using the *Function* symbol, a workaround to cope with these constraints must be explained.

In Figure 8.34 are presented two approaches for the initialization of an inherited attribute. In both approaches the `LHS` symbol of the production are the start symbol of the grammar.

In the first approach, Figure 8.34.a, the `LHS` symbol is As and it is associated to an inherited attribute named *in_ a*, which is an *In* attribute of the production. So, regarding what was referred above, this is a wrong specification of an inherited attribute initialization.

In Figure 8.34.b, in order to solve the problem, it was added a new production:

$$Root \rightarrow As$$

With this approach, the *NonTerminal* ‘As’ is still associated to the inherited attribute *in_ a*. But in this production, it is an *Out* attribute, so that its initialization is correct.

To summarize the section, is left the instruction on how to achieve the correct approach.

To correctly specify the initialization of an inherited attribute associated to the start symbol *The user should add a new production in order to make its* `LHS` *become the new start symbol*[5]*. The* `RHS` *of the new production must then be the older start symbol with the same attributes associated. At the end, the computation rule to initialize can be added without breaking any constraint.*

[5]See Section 8.2 for more details on how to define the start symbol of a grammar in `Visual`$\mathcal{LISA}$.

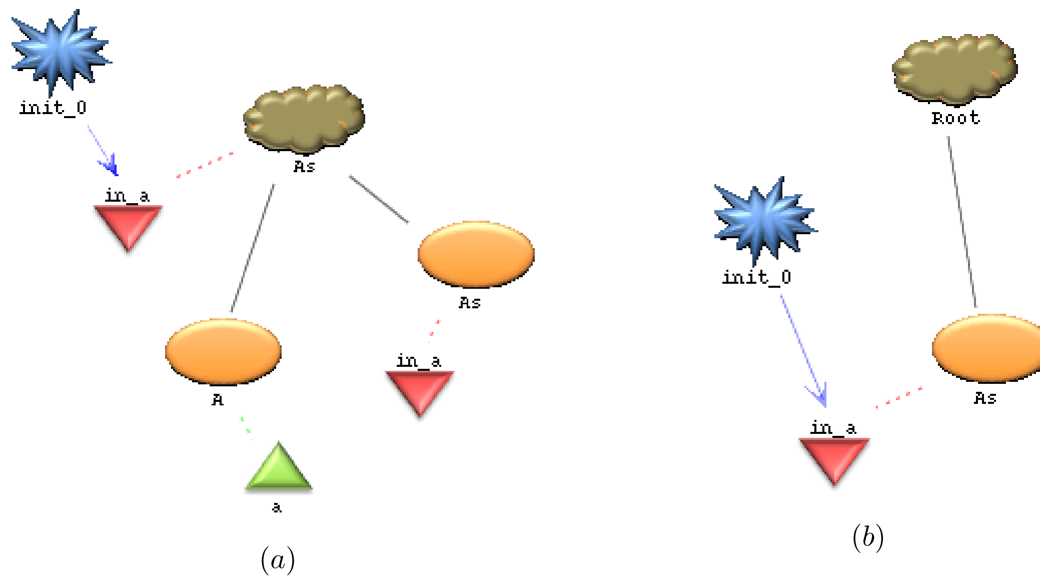


Figure 8.34: Initialization of an Inherited Attribute: (*a*) Wrong Approach; (*b*) Correct Approach.

8.5.6 Printing the value of an Attribute

Some compiler generator tools based in `AG`, like $\mathcal{LISA}$, do not accept explicit operations to print out the value of an attribute. This happens because each operation must be associated to an attribute.

However there are some cases where the processing of void functions is allowed in the `AG`. In spite of `Visual`$\mathcal{LISA}$ being prepared only to the former cases, that is, the functions must not be void, there is always a workaround to attain the objectives.

Bellow is shown a simple approach to solve the problem.

To print the value of an attribute named X *The user can associate a new synthesized attribute, let's call it A, to the* `LHS` *symbol of its production. This attribute,* `A` *should have a type, for instance, boolean. Then it must be added a Function* `F` *where* `A` *is the output. The operation in* `F` *must return a boolean value, but at the same time print the value of* `X`*.*

Normally a print function is void, so that the user must define a new function in the global definitions section (see Section 8.7), in order to wrap the print function in a boolean one.

An example of a user-defined function, in `Java` language, that receives an integer as parameter, can be specified as follows:

```
boolean printWrapper(int val){
    System.out.println(val);
    return true;
}
```

This approach is not one of a kind! There are many solutions to attain the same result. Sometimes it may be also necessary to add a new production defining a new start symbol.

Example Figure 8.35 addresses the approach described above. It pretends to print the value of the attribute *Students.sum* of the start symbol. To do this it was incremented a

new start production:

$$School \rightarrow Students$$

Now *School* is the start symbol. It has associated to it a synthesized attribute *ages* that was declared as a boolean.

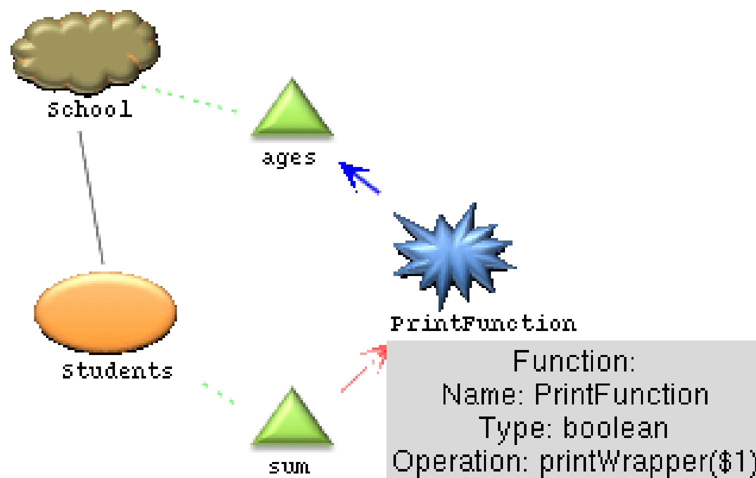


Figure 8.35: Computation Rule to Print the Value of the Attribute *Students.sum*.

8.6 Removing a Production or a Computation Rule

The remotion of a production or a computation rule is a not so easy task, because the natural way of doing it can cause an irreversible error in the working session.

Here will be explained the way of removing a production. To remove a computation rule the steps are similar.

To remove a production (1) *The user should go to rootView (Figure 8.2). In the list of productions right-click on the production to remove. A contextual menu similar to the one presented in Figure 8.36.a will pop up. Click on the* `Delete` *operation to perform that task.*

Notice, however, that this operation causes an error similar to that presented in Figure 8.36.*b*. When clicking `OK`, or `Skip Messages` the message disappears. But every time the cursor of the mouse is over the specification window, the message appears again.

Tip To get rid of this message, the user is forced to close `Visual`$\mathcal{LISA}$ and start it again. When restarting, the production (respectively the computation rule) stills remaining in the list, unless the user managed to save the specification before abort the program.

To remove a production (2) *The user should click on the button presented in Figura 8.37. A window with a tree view like the one presented in Figure 7.7 will appear. The user should then browse the tree to find the desired production (respectively computation rule).*

The next steps are similar to the approach (1), that is, right-click on the desired production entrance and choose the `Delete` *operation from the context menu.*

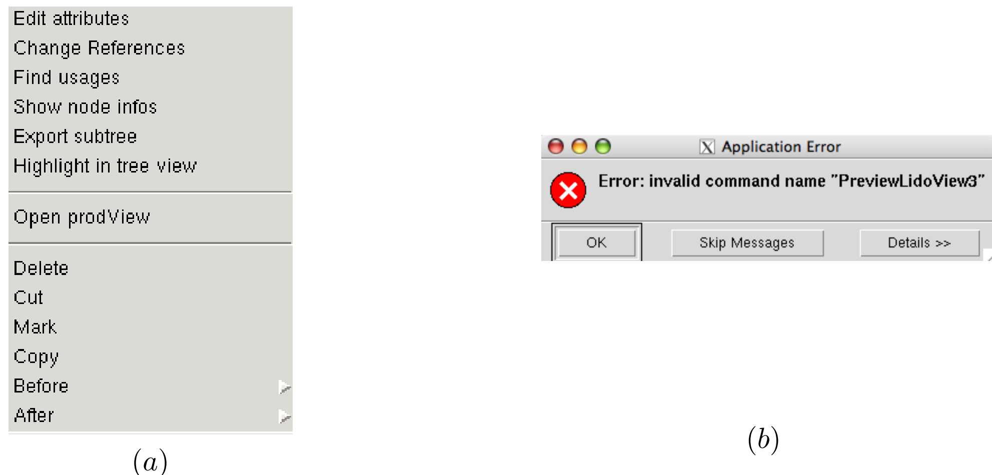


Figure 8.36: (*a*) Contextual Menu where the Delete Operation is Available; (*b*) Fatal Error Caused by Deleting a Production (respectively Computation Rule).

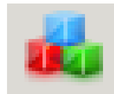


Figure 8.37: Button to Open Tree View Window.

Tip Is important to know some parts of the tree structure to an easy search for the symbol to remove (or inspect).

The root of that tree is named `Root`. The productions are children of the root's child named `semprods`, so, expanding that child, a list of *Semprod*symbols appear. Between parenthesis, is placed the name of the production. Following the approach (2) of removing a production, these are the entrances that can be selected to be removed.

Expanding a *Semprod*, its attributes will be listed. One of its children is named `computationRules`. If this child is expanded, there will appear the list of *Computation-Rule* symbols associated to the production. These are the elements that must be selected in order to perform the remotion of a computation rule following the approach (2).

8.7 Specification of Global Functions and Other Definitions

Visual*LISA* offers a semi-visual and simple way to make global definitions for the `AG`. User-defined functions, new lexemes, new data-types or importation of packages or modules necessary to the correct work of the compiler that will be generated from the `AG`.

To open the Global Definitions window *The user should go to the rootView window (Figure 8.2) and click twice on the "Global Definitions" entrance on the section at the right of the productions list. The window in Figure 8.38, named defsView, opens to show the definitions.*

That window is divided into four sections, each one used to define a different aspect from those recalled later: Lexemes, Types, Modules and User Functions.

Notice that in the Lexemes section is defined, by default, a lexeme. That lexeme, named *ignore* is used to ignore the white spaces and line breaks that exist in the source texts.

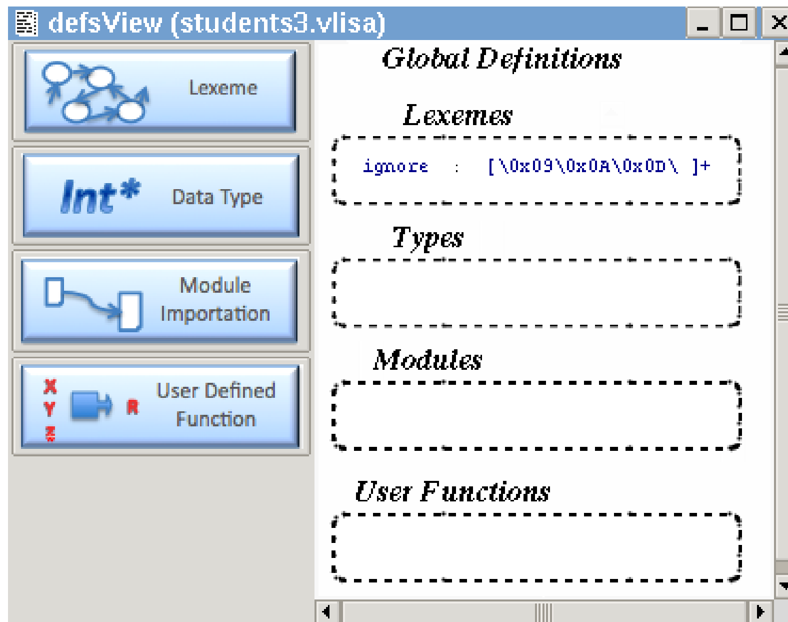


Figure 8.38: Global Definitions Window: *defsView*.

To insert and edit new Lexeme *The user should toggle the button in Figure 8.39.a and drag the mouse to the* Lexemes *section of defsView, inserting a default name in that area. Then the user should click twice on that name to open the Lexeme properties form. Figure 8.39.b. shows that form.*

The name of the new Lexeme should be written in the first field. In the second field, the user should write the **RE** *that defines the lexeme.*

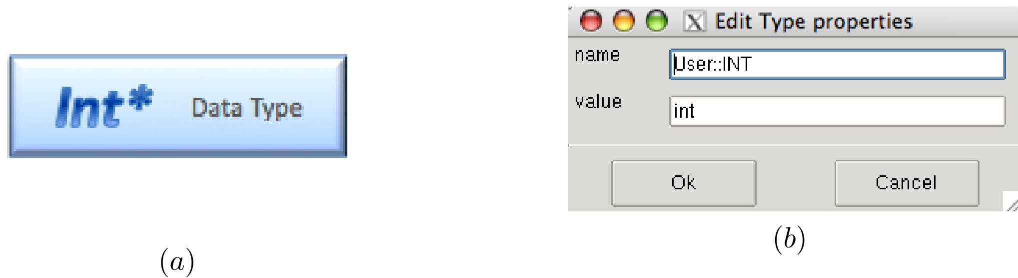
(*a*) (*b*)

Figure 8.39: (*a*) Button to insert a new Lexeme; (*b*) Form to Edit the Properties of a Lexeme.

To insert and edit new data-type *The user should toggle the button in Figure 8.40.a and drag the mouse to the* Types *section of defsView, inserting a default text in that area. Then the user should click twice on that text to open the types properties form. Figure 8.40.b. shows that form.*

The name of the new type should be written in the first field. In the second field, the user should write the real name that defines the type.

Tip It is encouraged to use the prefix 'User::' on the name of the new type, however it is not mandatory. This name will appear in the list of types, presented for example in



(*a*) (*b*)

Figure 8.40: (*a*) Button to insert a new Data-Type; (*b*) Form to Edit the Properties of a Data-Type.

Figure 8.15. Remember that this name is only an alias for the type that is written in the field *value*.

In Figure 8.40.*b*, it was defined the type *int* in the field *value*. For that, it was written one alias with the prefix 'User::' and a remainder — in this case, the word *INT* was used.

To insert a new module *The user should toggle the button in Figure 8.41.a and drag the mouse to the* Modules *section of defsView, inserting a default text in that area. Then the user should click twice on that text to open the properties window. Figure 8.41.b. shows that form.*

In the field named code*, the user should write only the name of the module. For example, to import the java.util.Calendar package, the user should write only java.util.Calendar.*

(*a*) (*b*)

Figure 8.41: (*a*) Button to insert a new Importation; (*b*) Form to Edit the Properties of an Importation.

To insert a new user function *The user should toggle the button in Figure 8.42.a and drag the mouse to the* User Functions *section of defsView, inserting a default text in that area. Then the user should click twice on that text to open the properties window. Figure 8.42.b. shows that form.*

In this form there are three properties to identify the function. The first is used to write the name for that function; the second is used to identify the number of parameters of the function; the third is used to write a complete function, including the signature of the function.

Tip (1) The code of the function must follow the correct syntax allowed by the target `AG` meta-language.

Tip (2) Actually, the values in the first and second fields are not used for any purpose. And they are not used to validate anything (yet). However, the user is encouraged to write

(*a*) (*b*)

Figure 8.42: (*a*) Button to insert a new Importation; (*b*) Form to Edit the Properties of an Importation.

the name and number of parameters of the function matching with the textual specification in the field *body*.

Example In the context of the running example, it was used a user-defined function to translate the textual value of the *age* into an integer value. In Figure 8.43 is shown how this function was created using the global definitions.

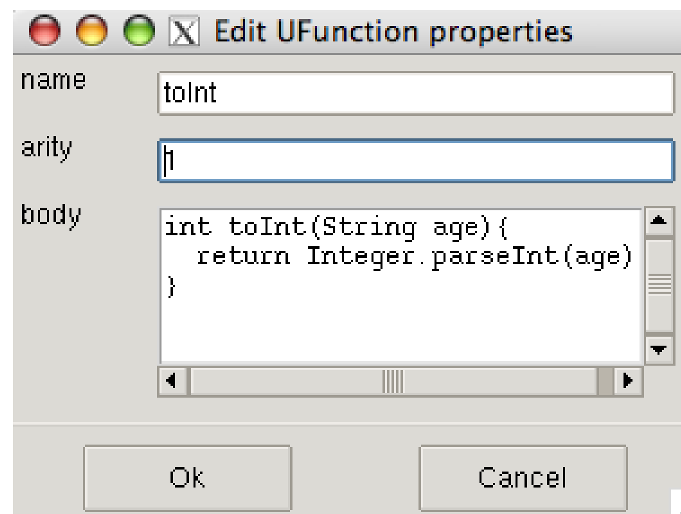


Figure 8.43: Example of Definition of a User Function.

8.8 Perform Semantic Verification

The semantics verification in VisualLISA is a task performed every time an edition action occurs. In spite of that, it is done very fast and do not compromise the edition speed. However the errors are not outputted, unless the user directly ask for them or tries to generate code.

To perform semantics verification and visualize the errors *The user should click on the button in Figure 8.44.*

The errors (if there were any) will appear in a new window like the one presented in Figure 8.45.

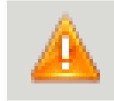


Figure 8.44: Button to Perform Semantics Verification.

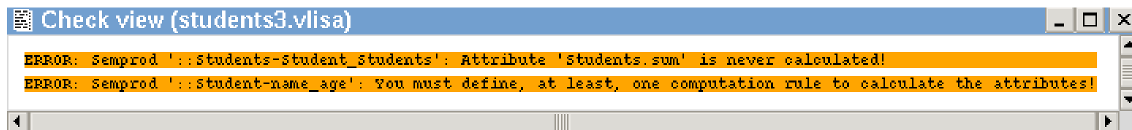


Figure 8.45: Window with the List of Semantic Errors.

Tip Each orange line reports a semantic error in a given context. If the user clicks on these lines, the window where the error occurs will open.

In Figure 8.45 there are presented two errors. Let's decipher the first:

Error: Semprod'::Students-Student_Students' : Attribute 'Students.sum' is never calculated!

The first part: *Semprod*'::Students-Student_Students' identifies the context where the error occurred. So that, an error in the context of the production with the name 'Students-Student_Students' occurred.

The second part: *Attribute 'Students.sum' is never calculated!* identifies the error. This example means that a computation rule was associated to the production, but the attribute 'Students.sum' have never been calculated.

Errors like the second one occur when there exist attributes that must be calculated, in a production, but a computation rule has not yet been associated to the production.

The major part of the errors point to the correct symbols where the error occurs. Sometimes it is given only an approximation.

8.9 Generate Code

The code generation can be performed at any time the user requires. But this only occurs if the drawn specification of the visual `AG` is semantically correct.

The semantics verification is performed always and automatically before the generation of code. If errors occur, a window, similar to the one presented in Figure 8.45 will pop up.

`Visual`*LISA*, as referred before, offers the possibility of generating *LISA*, `BNF` and *X*`AGra` notations.

To Generate Code *The user should click on the menu item named* `Process`*. A drop-down menu, like shown in Figure 8.46 will appear. Then the user is free to choose what is the code to generate from the three options offered.*

After generating the target code, the textual notation is flushed to a new window (presented in Figure 8.47).

This window has a single menu option: to save the code into a file. *LISA* code should be saved with .lisa extension and *X*`AGra` with .xml extension. For `BNF`, there is not a standard extension. If the user decides to save it, the .txt extension is a good option.

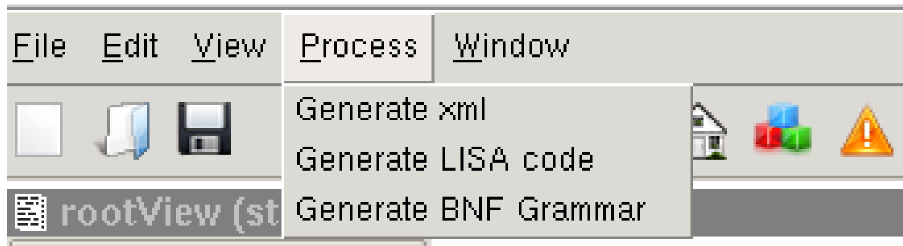


Figure 8.46: Menu to Generate Code

Example The Students Grammar is now ready to be translated into any required notation from the three offered by `Visual`*LISA*. In Figure 8.47, the Students Grammar was translated into *LISA*.

```
File
language SGrammar {

   lexicon{
      Name     [a-zA-Z]+
      Age     [0-9]+
      ignore     [\0x09\0x0A\0x0D\ ]+
   }

   attributes
      boolean  SCHOOL.ages;
      int  STUDENTS.sum;
      int  STUDENT.age;



   rule School_Students {
      SCHOOL ::=  STUDENTS compute {
         SCHOOL.ages = printWrapper(STUDENTS.sum);
      };
   }

   rule Students_Student_Students {
      STUDENTS ::=  STUDENT STUDENTS compute {
         STUDENTS.sum = STUDENTS[1].sum + STUDENT.age
      };
   }

   rule Students_Student {
      STUDENTS ::=  STUDENT compute {
```

Figure 8.47: Window with the Code Generated - *LISA* in this case.

Chapter 9

Conclusion

Textual programming languages are normally used to define huge computer programs. This happens because hard work was done in this area since the creation of the first compiler until now. Many languages were developed, environments to support the development of programs with these languages were built, efficiency was brought to the code generated by the compilers, etc. Because of this set of artifacts and their characteristics, the users feel comfortable to use textual languages.

In the field of Attribute Grammars (AG) many compiler generator tools were created based on language processing meta-languages defined resorting to several approaches, but all text-based. It is known that textual languages have some cons, besides all the pros. So that Visual Programming languages (VPLs) were created to overcome the drawbacks of textual languages.

In the project reported in this document a completely new concept on the specification of attribute grammars was developed: a new visual language, named V$\mathcal{LISA}$ to specify attribute grammars was defined, and a visual programming environment (Visual$\mathcal{LISA}$) was automatically generated taking advantage from the usage of DEViL, a tool for the automatic generation of visual language editors and compilers.

A new XML dialect, called $\mathcal{X}$AGra, was defined to make possible the translation of the visual AG specification into an abstract representation of an AG.

Moreover, from this work, some lessons were learned. Firstly it was confirmed that using automatic Visual Programming Environment generator tools, a complete and usable visual environment can be developed, resorting to small and maintainable specifications separated by several files. Secondly, regarding the fact that developing a visual environment has always underlying the specification of a visual language, it is possible to resort to a systematic approach based on compilers construction to specify and develop the complete environment. The approach, proposed and followed, is split in four main steps: Abstract Syntax Specification; Interaction and Layout Definition; Semantics Implementation and Code Generation.

At the end, the desired visual language for the specification of AGs (V$\mathcal{LISA}$) was defined and the environment Visual$\mathcal{LISA}$ was completely developed meeting all the requirements elicited. Visual$\mathcal{LISA}$ allows the visual specification of attribute grammars and its translation into $\mathcal{LISA}$ textual notation. Optionally, it allows the translation into $\mathcal{X}$AGra, what opens, in different ways, the purposes of Visual$\mathcal{LISA}$'s usage, and therefore originates new work around this tool.

During the development of Visual$\mathcal{LISA}$, several talks were given to different audiences. An intermediate and simple usability test, resorting to a group of students, was made, in order to gather information to improve Visual$\mathcal{LISA}$. Two papers, one for the

assessment of the author in the context of the UCE-15 master degree, another for an international conference (INTERACT 2009) were written. A third one, to submit to the conference on visual languages (VL-HCC 2009) is underwork at the moment of writing this document. Also a web-site[1] was created to spread widely the ideas and to distribute the software versions as well as documentation published and the present document that reports the work done.

9.1 Future Work

A complete usability test is lacking for this tool. In the future is important to submit Visual$\mathcal{LISA}$ through an usability test, to see how well it does with cognitive dimensions. Depending on the results of these tests, improvements should be made.

Despite the improvements that would come from this usability test, some other tasks should be accomplished. There is not a semantic verification module for the user-defined functions and their correct usage on the definition of computation rules. Namely, it is not checked the number of arguments and their types when using the user-defined functions.

Related with the specification of computation rules, the algorithm of ordering the arguments of a function should be reviewed. So that the order of the parameters can be visually perceptible. At this moment, the ordering of the parameters is based on the order of the parameter connectors creation. From this perspective, the user can not understand the order just by looking at the drawing.

9.2 Further Usage Perspectives

The fact of generating $\mathcal{X}$AGra, allows the use of Visual$\mathcal{LISA}$ to specify AGs for other compiler generators rather than $\mathcal{LISA}$. This implies the creation of translators that take $\mathcal{X}$AGra as input, and transform it into the target compiler notation.

The other way around, that is, the conversion of $\mathcal{X}$AGra into Visual$\mathcal{LISA}$ compliant notation, is also a possibility. This opens the perpectives of using Visual$\mathcal{LISA}$ as a visualization tool to help on AG comprehension.

[1] www.di.uminho.pt/~gepl/VisualLISA/

Bibliography

[Ass08] Barry & Associates. Xml vocabularies. `http://www.service-architecture.com/xml/articles/xml_vocabularies.html`, 2008.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. aw, 1986.

[BB94] Margaret Burnett and Marla Baker. A classification system for visual programming languages. *Journal of Visual Languages and Computing*, 5:287–300, 1994.

[Bis92] Kurt M. Bischoff. User manual for ox: An attribute-grammar compiling system based on yacc, lex, and C. Technical Report IASTATECS//TR92-30, Iowa State University, December 1992.

[CDP04] Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese. A framework for modeling and implementing visual notations with applications to software engineering. *ACM Trans. Softw. Eng. Methodol.*, 13(4):431–487, 2004.

[CGP+90] Claudia Crimi, Angela Guercio, Giuliano Pacini, Genoveffa Tortora, and Maurizio Tucci. Automating visual language generation. *IEEE Trans. Softw. Eng.*, 16(10):1122–1135, 1990.

[Cha90] Shi K. Chang, editor. *Principles of visual programming systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[CTODL95] Gennaro Costagliola, Genoveffa Tortora, Sergio Orefice, and Andrea De Lucia. Automatic generation of visual programming environments. *Computer*, 28(3):56–66, 1995.

[CTYY89] Shi K. Chang, Michael J. Tauber, Bing Yu, and Jing S. Yu. A visual language compiler. *IEEE Trans. Softw. Eng.*, 15(5):506–525, 1989.

[dLV02] Juan de Lara and Hans Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 174–188, London, UK, 2002. Springer-Verlag.

[dSD96] Luis Miguel da Sivla Dias. Linguagens visuais de programação - paradigmas e ambientes. Master's thesis, Universidade do Minho, Escola de Engenharia, December 1996.

[EEHT05] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Generation of visual editors as eclipse plug-ins. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 134–143, New York, NY, USA, 2005. ACM Press.

[EH07] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.

[GM93] Eric. J. Golin and T. Magliery. A compiler generator for visual languages. In *Visual Languages, 1993., Proceedings 1993 IEEE Symposium on*, pages 314–321, 1993.

[Gol91] Eric J. Golin. *A Method for the Specification and Parsing of Visual Languages*. PhD thesis, Brown University, Department of Computer Science, Providence, RI, USA, May 1991.

[GST05] Pavel Grigorenko, Ando Saabas, and Enn Tyugu. Cocovila - compiler-compiler for visual languages. *Electr. Notes Theor. Comput. Sci.*, 141:137–142, 2005.

[Hen92] Pedro R. Henriques. *Atributos e Modularidade na Especificação de Linguagens Formais*. PhD thesis, Universidade do Minho, December 1992.

[HPM+05] P. R. Henriques, M. J. V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu. Automatic generation of language-based tools using the lisa system. *Software, IEE Proceedings* -, 152(2):54–69, 2005.

[KS02] Uwe Kastens and Carsten Schmidt. Vl-eli: A generator for visual languages - system demonstration. *Electr. Notes Theor. Comput. Sci.*, 65(3), 2002.

[MKv95] Marjan Mernik, Nikolaj Korbar, and Viljem Žumer. Lisa: a tool for automatic language implementation. *SIGPLAN Not.*, 30(4):71–79, 1995.

[MLAv02] Marjan Mernik, Mitja Lenič, Enis Avdičaušević, and Viljem Žumer. Lisa: An interactive environment for programming language development. *Compiler Construction*, pages 1–4, 2002.

[MLAZ98] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. A reusable object-oriented approach to formal specifications of programming languages. *L'Object*, 4:273–306, 1998.

[MV95] M. Minas and G. Viehstaedt. Diagen: a generator for diagram editors providing direct manipulation and execution of diagrams. In *VL '95: Proceedings of the 11th International IEEE Symposium on Visual Languages*, Washington, DC, USA, 1995. IEEE Computer Society.

[PMdCH08] Maria João Pereira, Marjan Mernik, Daniela da Cruz, and Pedro R. Henriques. VisualLISA: a visual interface for an attribute grammar based compiler-compiler (short paper). In *CoRTA08 — Compilers, Related Technologies and Applications, Bragança, Portugal*, July 2008.

[PQ95] Terence Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995.

[RJG90] Robert V. Rubin, James, and Eric J. Golin. Early experience with the visual programmer's workbench. *IEEE Trans. Softw. Eng.*, 16(10):1107–1121, 1990.

[RMH+06] D. Rebernak, M. Mernik, P. R. Henriques, D. da Cruz, and Varanda M. J. Pereira. Specifying languages using aspect-oriented approach: Aspectlisa. In *Information Technology Interfaces, 2006. 28th International Conference on*, pages 695–700, 2006.

[Roc95] Jorge Gustavo Rocha. Especificação de linguagens visuais de programação. Master's thesis, Universidade do Minho, Departamento de Informática, June 1995.

[RS97] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages & Computing*, 8(1):27–55, February 1997.

[SF03] Chris Sells and Jon Flanders. *Mastering Visual Studio .Net.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[SKC06] Carsten Schmidt, Uwe Kastens, and Bastian Cramer. Using devil for implementation of domain-specific visual languages, 2006.

[SKC07] Carsten Schmidt, Uwe Kastens, and Bastian Cramer. Specifying coupled structures for implementation of visual languages, 2007.

[TR03] Juha P. Tolvanen and Matti Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93, New York, NY, USA, 2003. ACM.

[VWBGK08] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: an extensible attribute grammar system. *Electron. Notes Theor. Comput. Sci.*, 203(2):103–116, 2008.

[YB97] Sherry Yang and Margaret M. Burnett. Representation design benchmarks: a design-time aid for vpl navigable static representations. *Journal of Visual Languages and Computing*, 8:563–599, 1997.

[YDZ96] S. Yang, E. Dekoven, and M. Zloof. Design benchmarks for vpl static representations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 263–264, 1996.

Appendix A

$\mathcal{X}$AGra Schema Definition

This appendix presents, in Listing A.1, the complete textual schema used to define the XML universal notation for AGs, $\mathcal{X}$AGra.

At the end, Figure A.1 is a (complete) diagrammatic representation of the schema, composed by all the sub-parts presented in Chapter 7.

Listing A.1: Schema for $\mathcal{X}$AGra

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3     <xs:element name="attributeGrammar"><xs:complexType><xs:sequence><
          xs:element name="symbols">
4         <xs:complexType>
5             <xs:sequence><xs:element name="terminals"><xs:complexType><
                  xs:sequence maxOccurs="1" minOccurs="1"><xs:element
                  name="terminal" maxOccurs="unbounded" minOccurs="0" >
6                 <xs:complexType mixed="true"><xs:attribute name="id"
                      use="required" type="xs:ID"></xs:attribute>
7                 </xs:complexType>
8             </xs:element>
9             </xs:sequence>
10            </xs:complexType>
11            </xs:element>
12                <xs:element name="nonterminals"><xs:complexType><xs:
                      sequence><xs:element maxOccurs="unbounded"
                      minOccurs="0" name="nonterminal"><xs:complexType><
                      xs:attribute name="id" use="required" type="xs:ID
                      "></xs:attribute>
13                </xs:complexType>
14                </xs:element>
15                </xs:sequence>
16                </xs:complexType>
17                </xs:element>
18                <xs:element name="start">
19                    <xs:complexType>
20                        <xs:attribute name="nt" type="xs:IDREF" use="
                              required"></xs:attribute>
21                    </xs:complexType>
22                </xs:element>
23            </xs:sequence>
24        </xs:complexType>
25    </xs:element>
```

```
26        <xs:element name="attributesDecl"><xs:complexType><xs:sequence><xs:element name="declaration" maxOccurs="unbounded" minOccurs="0"><xs:complexType><xs:sequence><xs:element name="attribute" maxOccurs="unbounded" minOccurs="1"><xs:complexType><xs:attribute name="id" type="xs:ID" use="required"></xs:attribute>
27          <xs:attribute name="type" type="xs:string" use="required"></xs:attribute>
28          <xs:attribute name="class" use="required" type="xs:string"></xs:attribute>
29        </xs:complexType>
30        </xs:element>
31        </xs:sequence>
32        </xs:complexType>
33        </xs:element>
34        </xs:sequence>
35        </xs:complexType>
36        </xs:element>
37        <xs:element name="semanticProds"><xs:complexType><xs:sequence><xs:element name="semanticProd" maxOccurs="unbounded" minOccurs="0"><xs:complexType><xs:sequence><xs:element name="lhs"><xs:complexType><xs:attribute name="nt" type="xs:IDREF" use="required"></xs:attribute>
38        </xs:complexType>
39        </xs:element>
40          <xs:element name="rhs"><xs:complexType><xs:sequence><xs:element name="element" maxOccurs="unbounded" minOccurs="0"><xs:complexType><xs:attribute use="required" type="xs:IDREF" name="symbol"></xs:attribute>
41          </xs:complexType>
42          </xs:element>
43          </xs:sequence>
44          </xs:complexType>
45          </xs:element>
46          <xs:element name="computations" maxOccurs="1" minOccurs="1"><xs:complexType><xs:sequence><xs:element name="computation" maxOccurs="unbounded" minOccurs="0"><xs:complexType><xs:sequence><xs:element name="assignedAttribute"><xs:complexType><xs:attribute name="att" type="xs:IDREF" use="required"></xs:attribute>
47            <xs:attribute use="required" type="xs:integer" name="position"></xs:attribute>
48          </xs:complexType>
49          </xs:element>
50            <xs:element name="operation"><xs:complexType><xs:sequence><xs:element name="argument" maxOccurs="unbounded" minOccurs="0"><xs:complexType><xs:attribute name="att" type="xs:IDREF" use="required"></xs:attribute>
51              <xs:attribute use="required" type="xs:integer" name="position"></xs:attribute>
52            </xs:complexType>
53            </xs:element>
54              <xs:element name="modus" type="xs:string"></xs:element>
55            </xs:sequence>
```

```
56                          <xs:attribute name="returnType" type="xs:string"
                                use="required"></xs:attribute>
57                      </xs:complexType>
58                      </xs:element>
59                  </xs:sequence>
60                      <xs:attribute use="required" type="xs:string" name="
                            name"></xs:attribute>
61                  </xs:complexType>
62                  </xs:element>
63                  </xs:sequence>
64                  </xs:complexType>
65                  </xs:element>
66              </xs:sequence>
67                  <xs:attribute name="name" type="xs:string" use="required
                        "></xs:attribute>
68              </xs:complexType>
69              </xs:element>
70              </xs:sequence>
71              </xs:complexType>
72              </xs:element>
73              <xs:element name="importations"><xs:complexType><xs:sequence><
                    xs:element name="import" type="xs:string" maxOccurs="
                    unbounded" minOccurs="0"></xs:element>
74              </xs:sequence>
75              </xs:complexType>
76              </xs:element>
77              <xs:element name="functions"><xs:complexType><xs:sequence><xs:
                    element name="function" maxOccurs="unbounded" minOccurs
                    ="0"><xs:complexType mixed="true"><xs:attribute use="
                    required" type="xs:ID" name="name"></xs:attribute>
78              </xs:complexType>
79              </xs:element>
80              </xs:sequence>
81              </xs:complexType>
82              </xs:element>
83          </xs:sequence>
84              <xs:attribute name="name"></xs:attribute>
85          </xs:complexType>
86          </xs:element>
87  </xs:schema>
```

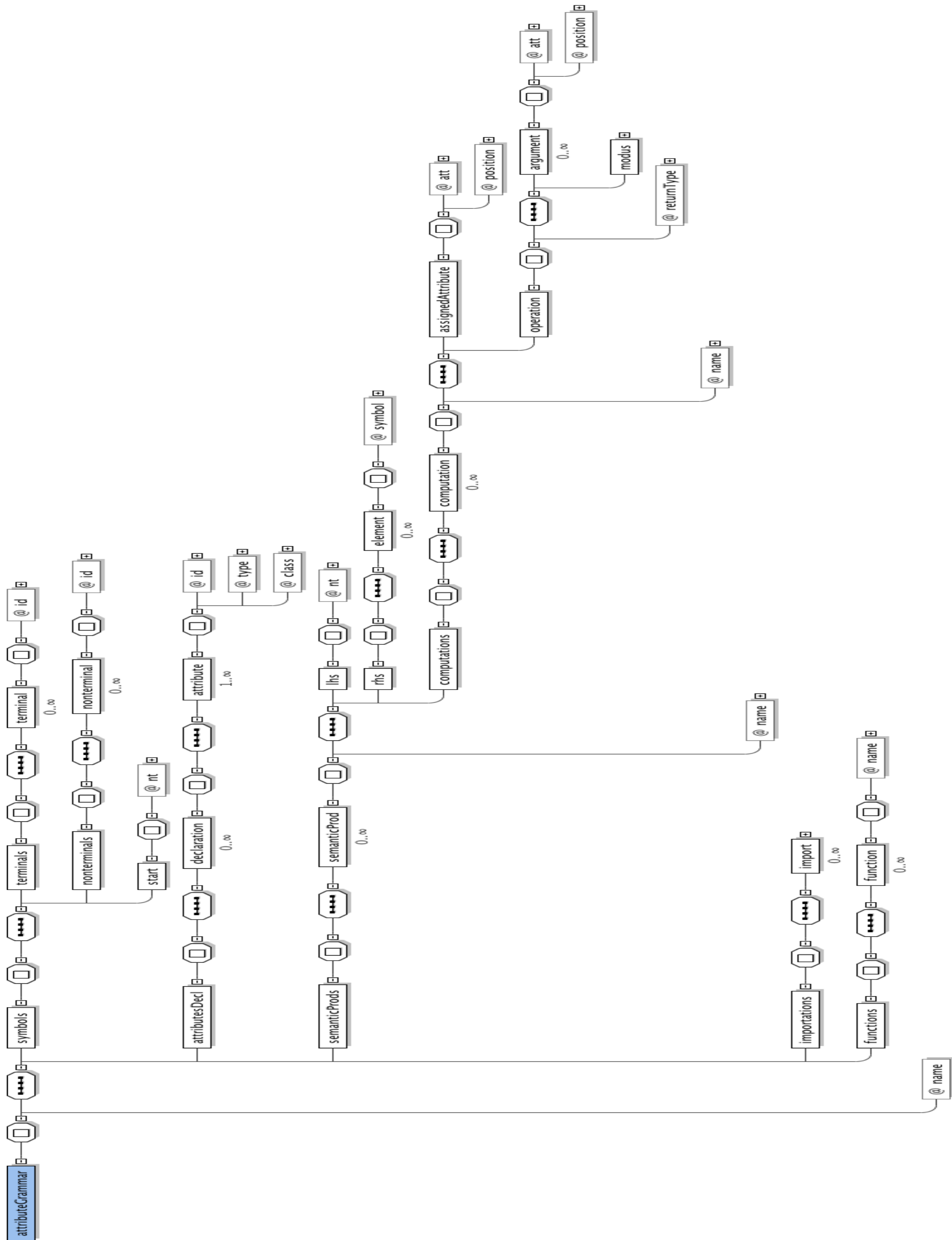


Figure A.1: Complete diagram for $\mathcal{X}$AGra scheme

Appendix B

Language Specification in DEViL

In this appendix the complete abstract structure of V$\mathcal{LISA}$ is presented (Listing B.1). This abstract syntax is specified using DEViL's notation.

Comment lines were removed from this specification for the sake of conciseness. The most important explanations or hints on V$\mathcal{LISA}$ description were already presented in chapter 7.

Listing B.1: Complete V$\mathcal{LISA}$specification

```
1
2  CLASS Root {
3            name: VAL VLString;
4        semprods: SUB Semprod*;
5            definitions: SUB Definitions!;
6            library: SUB Library?;
7  }
8
9
10 CLASS Definitions {
11          name: VAL VLString EDITWITH "None";
12          lexicon: SUB Lexeme*;
13          imports: SUB Import*;
14          userTypes: SUB Type*;
15          userFunctions : SUB UFunction*;
16 }
17
18 CLASS Library {
19          types : SUB Type*;
20 }
21
22 CLASS Type {
23          name: VAL VLString;
24          value : VAL VLString;
25 }
26
27 CLASS Import {
28          code: VAL VLString;
29 }
30
31 CLASS UFunction {
32          name : VAL VLString;
33          arity: VAL VLString;
```

```
34          body: VAL VLString EDITWITH "Text";
35 }
36
37
38
39
40 CLASS Lexeme {
41          name : VAL VLString;
42          regExp: VAL VLString;
43
44          setSize: VAL VLPoint INIT "300 300" EDITWITH "None";
45 }
46
47 CLASS Semprod {
48            name:  VAL VLString;
49            grammarElements: SUB AGElement*;
50            computationRules: SUB rules::Semprod*;
51            setSize: VAL VLPoint INIT "300 300" EDITWITH "None";
52 }
53
54 ABSTRACT CLASS AGElement {
55 }
56
57 ABSTRACT CLASS Symbol INHERITS AGElement{
58          setSize: VAL VLPoint INIT "100 100" EDITWITH "None";
59          position: VAL VLPoint EDITWITH "None";
60 }
61
62 ABSTRACT CLASS ISSymbol {
63 }
64
65 CLASS LeftSymbol INHERITS ISSymbol, Symbol {
66          symbolName: VAL VLString;
67 }
68
69 ABSTRACT CLASS RightSide INHERITS Symbol{
70          symbolName: VAL VLString;
71 }
72
73 CLASS NonTerminal INHERITS RightSide, ISSymbol {
74 }
75
76 CLASS Terminal INHERITS RightSide{
77          regExp: VAL VLString? EDITWITH "Entry";
78 }
79
80 ABSTRACT CLASS Attribute INHERITS AGElement{
81          setSize: VAL VLPoint INIT "100 100" EDITWITH "None";
82          position: VAL VLPoint EDITWITH "None";
83 }
84
85 ABSTRACT CLASS AssignAttribute {}
86
87 CLASS SyntAttribute INHERITS Attribute, AssignAttribute {
88          attributeName: VAL VLString;
89          type : REF Type;
```

```
90 }
91
92 CLASS IntrinsicValueAttribute INHERITS Attribute {
93         attributeName: VAL VLString;
94         type : REF Type;
95 }
96
97 CLASS InhAttribute INHERITS Attribute, AssignAttribute {
98         attributeName: VAL VLString;
99         type : REF Type;
100 }
101
102 CLASS TreeBranch INHERITS AGElement{
103         from : REF LeftSymbol;
104         to : REF RightSide;
105
106         anchorPoints: VAL VLList? EDITWITH "None";
107         position: VAL VLPoint EDITWITH "None";
108 }
109
110 ABSTRACT CLASS AttConnection INHERITS AGElement {
111 }
112
113 CLASS InhConnection INHERITS AttConnection {
114         from: REF ISSymbol;
115         to : REF InhAttribute;
116
117         anchorPoints: VAL VLList? EDITWITH "None";
118         position: VAL VLPoint EDITWITH "None";
119 }
120
121 CLASS SyntConnection INHERITS AttConnection {
122         from : REF ISSymbol;
123         to : REF SyntAttribute;
124
125         anchorPoints: VAL VLList? EDITWITH "None";
126         position: VAL VLPoint EDITWITH "None";
127 }
128
129 CLASS IntrinsicValueConnection INHERITS AttConnection {
130         from : REF Terminal;
131         to : REF IntrinsicValueAttribute;
132
133         anchorPoints: VAL VLList? EDITWITH "None";
134         position: VAL VLPoint EDITWITH "None";
135 }
136
137 ABSTRACT CLASS Computation INHERITS AGElement {
138         position: VAL VLPoint EDITWITH "None";
139         setsize: VAL VLPoint INIT "140 80" EDITWITH "None";
140 }
141
142 CLASS Identity INHERITS Computation{
143         from: REF Attribute;
144         to: REF AssignAttribute;
145         anchorPoints : VAL VLList? EDITWITH "None";
```

```
146 }
147
148 ABSTRACT CLASS FunctionComputation INHERITS Computation {
149 }
150
151
152 CLASS FunctionArg INHERITS FunctionComputation {
153         function : REF Function;
154         arg : REF Attribute;
155         anchorPoints : VAL VLList? EDITWITH "None";
156 }
157
158 CLASS FunctionOut INHERITS FunctionComputation {
159         function : REF Function;
160         to : REF AssignAttribute;
161
162         anchorPoints : VAL VLList? EDITWITH "None";
163 }
164
165 CLASS Function INHERITS AGElement {
166         returnType: REF Type;
167         functionName: VAL VLString EDITWITH "Entry";
168         operation : VAL VLString EDITWITH "Entry";
169         setSize: VAL VLPoint INIT "100 100" EDITWITH "None";
170         position: VAL VLPoint EDITWITH "None";
171
172 }
173
174
175 DERIVED rules ROOT Semprod{
176
177         Identity.removeAbandoned = 0;
178         FunctionArg.removeAbandoned = 0;
179         FunctionOut.removeAbandoned = 0;
180         Function.removeAbandoned = 0;
181
182 CLASS Semprod {
183             prodName: VAL VLString? EDITWITH "None";
184             ruleName:  VAL VLString;
185             grammarElements: SUB AGElement*;
186             baseRef: REF ::Semprod EDITWITH "None";
187             setSize: VAL VLPoint INIT "300 300" EDITWITH "None";
188 }
189
190 CLASS LeftSymbol INHERITS AGElement, ISSymbol, Symbol {
191         baseRef: REF ::LeftSymbol EDITWITH "None";
192         symbolName: VAL VLString EDITWITH "None";
193 }
194
195 ABSTRACT CLASS RightSide INHERITS Symbol{
196         symbolName: VAL VLString EDITWITH "None";
197 }
198
199 CLASS NonTerminal INHERITS RightSide, ISSymbol {
200         baseRef: REF ::NonTerminal EDITWITH "None";
201 }
```

```
202
203 CLASS Terminal INHERITS RightSide{
204         baseRef: REF ::Terminal EDITWITH "None";
205         regExp: VAL VLString? EDITWITH "None";
206 }
207
208 CLASS SyntAttribute INHERITS Attribute, AssignAttribute {
209         attributeName: VAL VLString EDITWITH "None";
210         type: REF Type? EDITWITH "None";
211         baseRef: REF ::SyntAttribute EDITWITH "None";
212 }
213
214 CLASS InhAttribute INHERITS Attribute, AssignAttribute {
215         attributeName: VAL VLString EDITWITH "None";
216         type: REF Type? EDITWITH "None";
217         baseRef: REF ::InhAttribute EDITWITH "None";
218 }
219
220 CLASS IntrinsicValueAttribute INHERITS Attribute {
221         attributeName: VAL VLString EDITWITH "None";
222         type: REF Type? EDITWITH "None";
223         baseRef: REF ::IntrinsicValueAttribute EDITWITH "None";
224 }
225
226 CLASS TreeBranch INHERITS AGElement{
227         from : REF LeftSymbol;
228         to : REF RightSide;
229         baseRef: REF ::TreeBranch EDITWITH "None";
230         anchorPoints: VAL VLList? EDITWITH "None";
231         position: VAL VLPoint EDITWITH "None";
232 }
233
234
235 CLASS InhConnection INHERITS AttConnection {
236         from: REF ISSymbol;
237         to : REF InhAttribute;
238         baseRef: REF ::InhConnection? EDITWITH "None";
239         anchorPoints: VAL VLList? EDITWITH "None";
240         position: VAL VLPoint EDITWITH "None";
241 }
242
243
244 CLASS SyntConnection INHERITS AttConnection {
245         from : REF Symbol;
246         to : REF SyntAttribute;
247         baseRef : REF ::SyntConnection? EDITWITH "None";
248         anchorPoints: VAL VLList? EDITWITH "None";
249         position: VAL VLPoint EDITWITH "None";
250 }
251
252 CLASS IntrinsicValueConnection INHERITS AttConnection {
253         from : REF Terminal;
254         to : REF IntrinsicValueAttribute;
255         baseRef : REF ::IntrinsicValueConnection? EDITWITH "None";
256         anchorPoints: VAL VLList? EDITWITH "None";
257         position: VAL VLPoint EDITWITH "None";
```

```
258
259 }
260
261
262
263
264 CLASS Identity INHERITS Computation{
265         from: REF Attribute;
266         to: REF AssignAttribute;
267         baseRef: REF ::Identity? EDITWITH "None";
268         anchorPoints : VAL VLList? EDITWITH "None";
269 }
270
271
272 CLASS FunctionArg INHERITS FunctionComputation {
273         function : REF Function;
274         arg : REF Attribute;
275         baseRef: REF ::FunctionArg? EDITWITH "None";
276         anchorPoints : VAL VLList? EDITWITH "None";
277 }
278
279
280 CLASS FunctionOut INHERITS FunctionComputation {
281         function : REF Function;
282         to : REF AssignAttribute;
283         baseRef: REF ::FunctionOut? EDITWITH "None";
284         anchorPoints : VAL VLList? EDITWITH "None";
285 }
286
287 CLASS Function INHERITS AGElement {
288         returnType: REF ::Type;
289         functionName: VAL VLString EDITWITH "Entry";
290         operation: VAL VLString EDITWITH "Entry";
291         baseRef: REF ::Function? EDITWITH "None";
292         setSize: VAL VLPoint INIT "100 100" EDITWITH "None";
293         position: VAL VLPoint EDITWITH "None";
294
295 }
296
297 }
```

Appendix C

Code Generated For $\mathcal{X}$AGra- Example

In this appendix is shown the result, $\mathcal{X}$AGra notation, of the translation of an AG specified in Visual$\mathcal{LISA}$. For reasons of space, the image with $\mathcal{X}$AGra notation was divided in three parts, resulting in Figures C.1, C.2 and C.3

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
   <attributeGrammar name="schoolGra">
      <symbols>
         <terminals>
            <terminal id="name">[A-Z][a-z]+</terminal>
            <terminal id="age">[0-9]+</terminal>
            <terminal id="ignore">[\0x09\0x0A\0x0D\ ]+</terminal>
         </terminals>
         <nonterminals>
            <nonterminal id="school" />
            <nonterminal id="students" />
            <nonterminal id="student" />
         </nonterminals>
         <start nt="school">
      </symbols>
      <attributesDecl>
         <declaration symbol="school">
            <attribute id="sum" type="int" class="SyntAttribute" />
         </declaration>
         <declaration symbol="students">
            <attribute id="sum" type="int" class="SyntAttribute" />
         </declaration>
         <declaration symbol="student">
            <attribute id="age" type="int" class="SyntAttribute" />
         </declaration>
         <declaration symbol="age">
            <attribute id="value()" type="String" class="IntrinsicValueAttribute" />
         </declaration>
      </attributesDecl>
      <semanticProds>
         <semanticProd name="school">
            <lhs nt="school" />
            <rhs>
               <element symbol="students" />
            </rhs>
            <computations>
               <computation name="copy_value">
                  <assignedAttribute att="sum" assocSymbol="school" position="0" />
```

Figure C.1: $\mathcal{X}$AGra Code Generated - Part 1

```
        <operation>
           <argument att="sum" assocSymbol="students" position="1" />
           <modus> $1 </modus>
        </operation>
     </computation>
  </computations>
</semanticProd>
<semanticProd name="students_1">
  <lhs nt="students" />
  <rhs>
     <element symbol="student" />
     <element symbol="students" />
  </rhs>
  <computations>
     <computation name="getTheSum">
        <assignedAttribute att="sum" assocSymbol="students" position="" />
        <operation>
           <argument att="age" assocSymbol="student" position="1" />
           <argument att="sum" assocSymbol="students" position="2" />
           <modus> $1 + $2 </modus>
        </operation>
     </computation>
  </computations>
</semanticProd>
<semanticProd name="students_2">
  <lhs nt="students" />
  <rhs>
     <element symbol="student" />
  </rhs>
  <computations>
     <computation name="copy_value">
        <assignedAttribute att="sum" assocSymbol="students" position="0" />
        <operation>
           <argument att="age" assocSymbol="student" position="1" />
           <modus> $1 </modus>
        </operation>
     </computation>
  </computations>
```

Figure C.2: $\mathcal{X}$AGra Code Generated - Part 2

```
     </semanticProd>
     <semanticProd name="student">
        <lhs nt="student" />
        <rhs>
           <element symbol="name" />
           <element symbol="age" />
        </rhs>
        <computations>
           <computation name="toInt_cpy">
              <assignedAttribute att="age" assocSymbol="student" position="" />
              <operation>
                 <argument att="value()" assocSymbol="age" position="2" />
                 <modus> Integer.parseInt($1) </modus>
              </operation>
           </computation>
        </computations>
     </semanticProd>
  </semanticProds>

  <functions>

  </functions
</attributeGrammar>
```

Figure C.3: $\mathcal{X}$AGra Code Generated - Part 3

Appendix D

Students Grammar Solution

This chapter presents the final state of the Students Grammar used as running example to aid the explanation of Visual$\mathcal{LISA}$ in Chapter 8.

Figure D.1 shows the list of productions and the global definitions used to specify the attribute grammar.

Figure D.2 shows the layout of the production, that defined the start symbol, and the associated semantic rule used to print the sum of the ages.

Figure D.3 to D.5 shows the layout of production P_1, P_2 and P_3 respectively, and the associated semantic rules used to compute the attributes value.

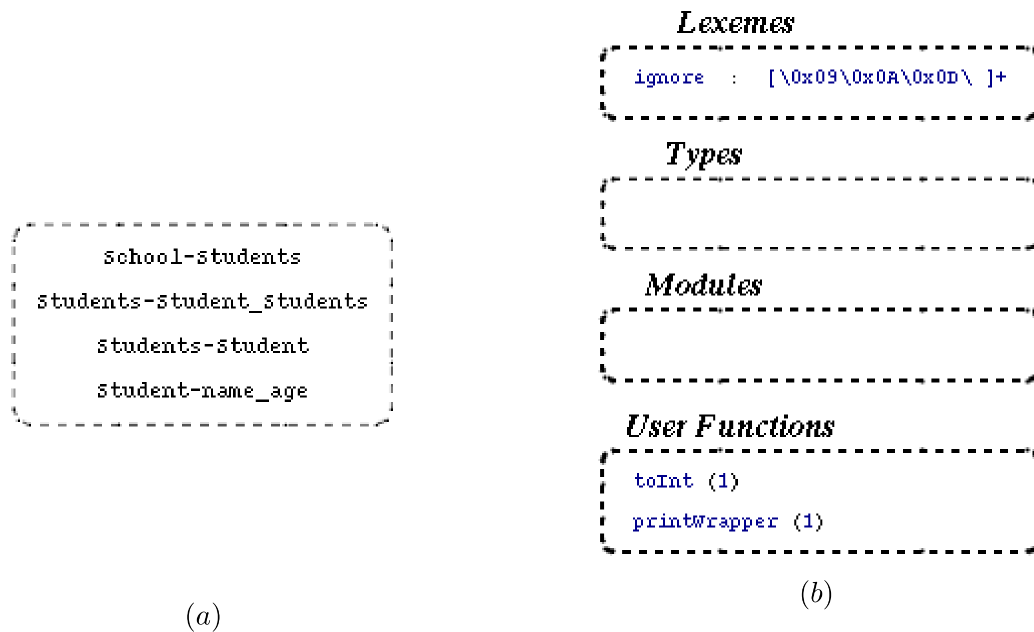


Figure D.1: *StudentsGrammar* : (a) List of Production; (b) Global Definitions.

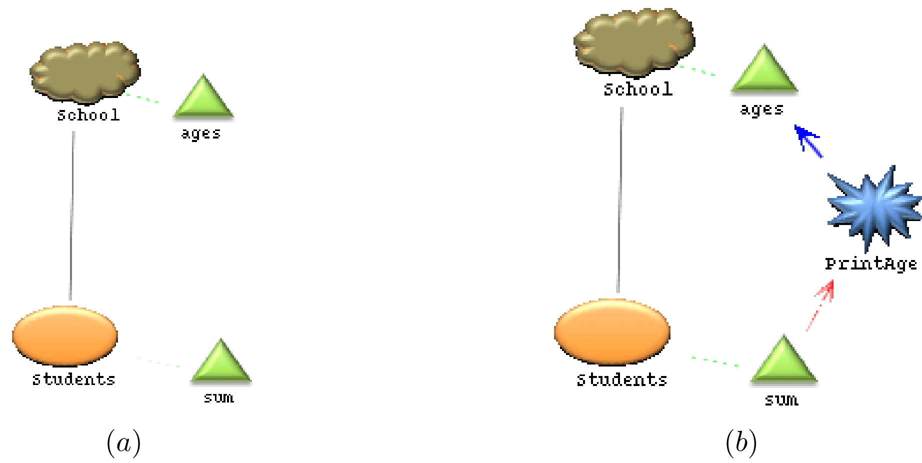


Figure D.2: Students Grammar: (*a*) Initial Production; (*b*) Semantic Rule to Print the Ages.

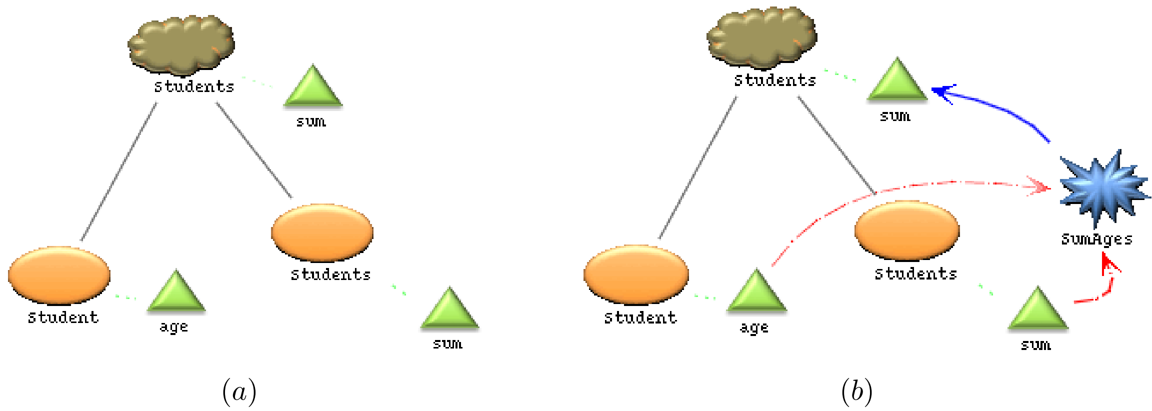


Figure D.3: Students Grammar: (*a*) Layout of Production P_1; (*b*) Rule to Compute the Value of the *Students.sum* Attribute.

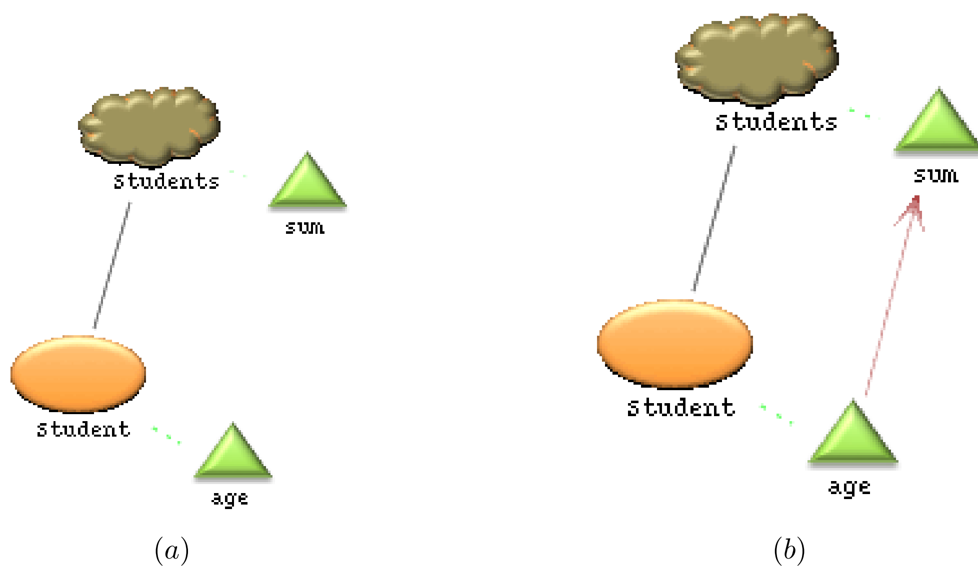


Figure D.4: Students Grammar: (*a*) Layout of Production P_2; (*b*) Rule to Compute the Value of the *Students.sum* Attribute.

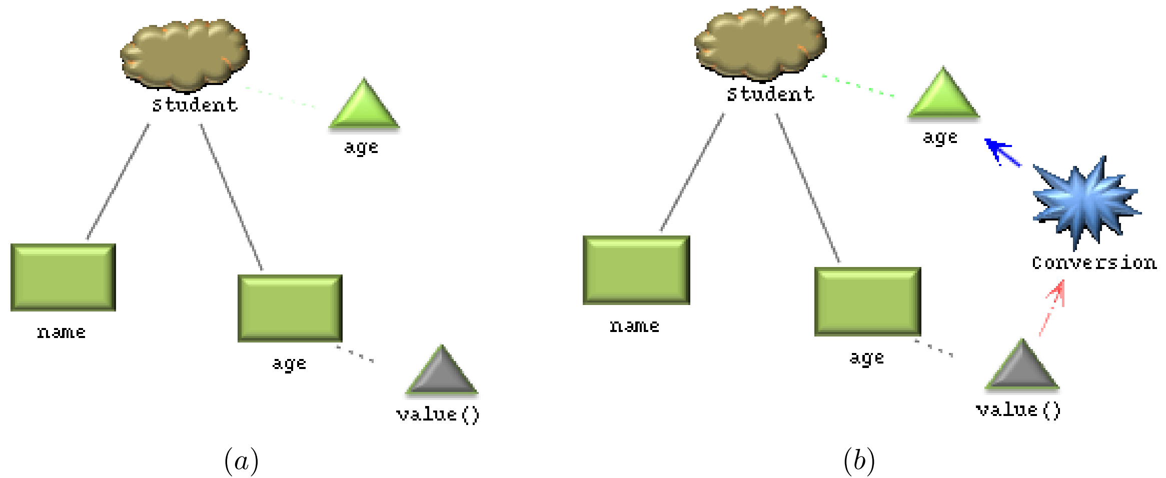


Figure D.5: Students Grammar: (*a*) Layout of Production P_3; (*b*) Rule to Compute the Value of the *Student.age* Attribute.