From Diagram to Code via Attribute Grammar

Isabel Cafezeiro
isabel@lmf-di.inf.puc-rio.br

Edward Hermann Häusler
hermann@inf.puc-rio.br

Armando Haeberer
armando@inf.puc-rio.br

PUC-RioInf. MCC03/97 March, 1997

Abstract
This report describes within a formal approach how to generate code from a graphical description based on diagrams. The method is general enough not to consider a specific methodology nor a specific target programming language. The essence of the method consists of combining Object Oriented ideas to capture the generality of diagrams and Attributed Grammars to solve the context-sensitive feature. A case study is presented.

Resumo
Este relatório descreve, em uma abordagem formal, como gerar código a partir de uma descrição gráfica baseada em diagramas. O método é suficientemente geral de modo a não fixar nem uma metodologia específica, nem uma linguagem de programção específica. A essência do método consiste em combinar idéias de Orientação a Objetos para alcançar a generalidade dos diagramas e Gramáticas de Atributos para resolver a sensibilidade ao contexto. Um estudo de casos é apresentado.

From Diagram to Code via Attribute Grammar

Isabel Cafezeiro[1]
Edward Hermann Heausler[2]
Armando Haeberer[3]

[1] isabe@lmf-di.inf.puc-rio.br
[2] hermann@inf.puc-rio.br
[3] Armando@inf.puc-rio.br

Introduction

It is well known that people spend more efforts to understand written information than the ones that are organised in pictures. This fact is largely explored in computers, and particularly in descriptions of systems. Since the seventies graphs were used in software development. But at that time, the visual information played only the passive role of documenting the system. Nowadays, in addition of presenting a useful graphic notation to describe a system, the named *CASE Tools* (*C*omputer *A*dded *S*ystem *E*ngineering) are expected to play the active role of generating some code. So, there must be a correspondence between the entities that are put in the graph and mechanisms of the programming language in which the system will be implemented. Of course, this correspondence is attached to the level of abstraction been described: the more abstract the description, the more incomplete the code to generate. The purpose of this text is to use *attribute grammars* to describe how to make such a connection.

The approach to be followed here is formalise the correspondence between the diagram and the programming language used, in order to put the results general enough so that most of the existing *CASE Tools* can be formalised in this way.

The next section is a brief explanation of the *CASE Tool* selected. After this section, we begin with the formal description. The last section is the general formal description.

Plaina: An overview

PLAINA is a CASE tool designed to support object oriented methodologies[4]. It has as underlying concept the idea that there are two important viewpoints when developing a system: the users viewpoint, and the systems viewpoint. The later concerns implementational aspects: how reality is represented in a computer. The former concerns the aspects that the user has to manipulate when defining the logical structure of the system, i.e. concepts that are not related with files, directories, etc. This report will feature the users viewpoint to present a formal description of Plaina environment. So, let us briefly describe the PLAINA elements considering the users viewpoint.

In order to develop a system, the user has to write a **project** of it. The project is a set of **diagrams** that describes the system in its various levels of abstraction. We have five different kinds of diagrams in PLAINA: **entity**, **class**, **object**, **state** and **time diagrams**. All these diagrams are related, and, in some way, each one of them contributes to the generation of **DDL** class codes. DDL (Design Description Language) is an object oriented programming language created to represent OO designs. The complete set of DDL classes will be the implementation of the system. PLAINA also has a **library** which contains the class codes, and a module that translates DDL code in C++ code.

The construction of an OO system involves activities as defining the **system domain**; the **analysis phase**, the **design phase**, and the **programming phase**. The first phase consists in identifying the subject of the system, that is, to delimit its scope. The second is concerned with identifying the entities and their relationships. In the design phase, the system gets the form to be implemented in.

---

[4] See 2GOOD:
A Tool for Second Generation Object Oriented Development

Finally, the programming phase is when code is written. In some way, we can reflect this activities in the diagram construction. For example, the analysis phase can be associated with the construction of entity diagrams. In the entity diagram, the entities are represented by boxes and their relations are established by means of directed and labelled arrows - the label shows the nature of the relation, i.e., whether it is an inheritance relation, or composition relation, etc. In the entity diagram, entities are described in a great degree of abstraction, as been part of the application domain. The state diagram can represent the states of this entities and state changes. Passing from analysis to design, the class diagram is the description of these entities, but in a detailed view, as requires the implementation domain. Besides, the object diagram reflects the instanciation and service requests, that will be necessary to compose the final code. Time diagrams represent object life cycles.

As we do not identify a strong frontier separating these activities, we claim that there is not a predefined order which has to be followed to construct the system. Instead, the system is a result of a sequence of reviews and refinements in each activity. For this reason, PLAINA does not determine an order of making the diagrams. The user can draw pieces of incomplete and disconnected diagrams, and later, connect them providing a sense to the full diagram. This possibility is assured by two features that lead us to think of PLAINA as a reactive environment:

- Each action the user makes has an effect in the class codes,
- At every moment, the generated code reflects the exact movements of the user.

PLAINA is able to generate code even in the non-sense steps. Having this feature PLAINA provides the user with the ability of dealing with incomplete diagrams.

When this report was beeing written PLAINA was not completed yet. Because of this, we are considering here only a small part of the environment.

The attribute grammar

Basic Concepts

Because of its natural characteristic of describing proper nesting[5], context free grammars (CFG) have been successfully used as a mechanism for describing structured languages[6]. A common notation for CFG is the Backus-Naur form (BNF), where productions are expressed as

< E > ::= < E > **+** < E >
< E > ::= **(** < E > **)**
< E > ::= **id**

The symbols in bold face are the symbols of the language (terminals). The others between "<" and ">" are called "non terminal". If they appear at the right side of each production they are to be replaced by the right side of a production having the same symbol at the left side. The symbol "::= " is to indicate this rewriting. Thus, we can say that an occurrence of < E > can be replaced by < E > **+** < E >, following the first rewritting rule. By the same mechanism, and the third rewritting rule, we can say taht < E > can be replaced by **id**, generating **id + id**. This is the derivation of a simple addition:

[5] By "proper nesting" we are referring to some kind of sentences that begins with a pattern and ends with this pattern but in reversed order. Examples are "abcdcba" or (((id))).
[6] By "structured languages" we mean the ones which sentences can be described by a tree structure.

$$< E > \Rightarrow < E > + < E > \Rightarrow \mathbf{id} + < E > \Rightarrow \mathbf{id} + \mathbf{id}$$

BNF is a creation of _____ in 19_____ and is even nowadays largely used in programming language description. In the above example we have a grammar to generate expressions like “a”, “a+a”, “(a+a)”, ...

Despite of presenting very clear notations, and of being powerful enough to describe syntactical features of structured languages, context free formalisms are not able to express some possible dependence among parts of a structure that are not related in the same production. This kind of relation we call “context dependent”.

To express context dependent features we have to add context dependent productions like

$$\alpha_1 A \alpha_2 ::= \alpha_1 \beta \alpha_2$$

with α_1, α_2 , β, being a non empty string of grammar symbols and *A* a non terminal. According to this production, *A* can be rewritten by β only in the context of α_1 and α_2. Context dependent features can not be represented in a BNF notation.

Context free notations like BNF are very useful in the implementation of programming language because they have an efficient model of implementation. So, there must be a solution on how to add context dependent features to context free notations, or, in other words, how to transform BNF-like descriptions in context dependent formalisms without loosing its clarity, and without causing big changes on the implementation model.

The solution can be achieved by **attribute grammars**.

Attribute grammar is a context free grammar augmented with attribute rules. Consider that each node of the grammar has an associated set of attributes. The rules will compute the value of these attributes.

An attribute can be **synthesised** or **inherited**. The synthesised are the ones that belong to a non terminal at the right side of the production and whose value is used to compute an attribute of the non terminal at left side of the rule. For example, consider the grammar production

$$(\ i\ ) < E_1 > ::= < E_2 > + < E_3 >$$

that has the attribute rule

$$< E_1 >.\text{value} := < E_2 >.\text{value} + < E_3 >.\text{value}.$$

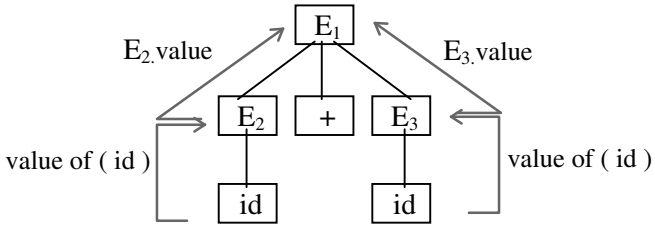
The subscript numbers are to distinguish the non terminal with the same names and allow references of each one in the attribute rule.

The attribute *value* of non terminal < E > is synthesised because it depends on attributes of non terminals at the right side of the rule. Do not matter if the of non terminals at the right side of the rule are also < E >. These occurrences of < E > will also be computed by the same rule, or, to make recursion ends, by a rule like

$$< E_1 >.\text{value} := \textit{value of} (\ \mathbf{id}\ )$$

of the production < E > ::= **id**.

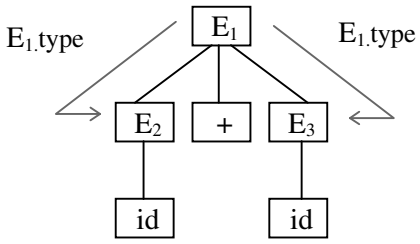
Looking to a syntactic tree, we can see that synthesised attributes goes up the tree.



The attributes that are not synthesised are inherited. These one have their value computed on the value of attributes of the left side of the rule, or on the right side of the rule. Thus, considering the production (*i*), we can have the followings rules:

$< E_2 >.type := < E_1 >.type$
$< E_3 >.type := < E_2 >.type.$

Inherited attribute goes down the syntactic tree.



Two restriction are needed to an attributed grammar:

1) An attribute of a non terminal must be *or* inherited *or* synthesised (exclusive *or*).
2) A rule of a production must not reference attributes of a symbol out of the production.

Observing these two restriction we can ensure that values will walk up or down the tree, level by level, without performing "jumps". This fact is very important because it ensures the simplicity of the implementation.

As a last remark, we have to say that a good attribute definition do not have cycles, i.e., there is not an attribute whose value has to be computed on values that depends directly or indirectly of itself.

To finish our brief exposition of attribute grammars, we have to say that it is a creation of _____ knuth, 1968, on _____ .
More about this topic can be find on references _____

Grammar Nodes: their attributes and their behaviour

Let us begin with the description of the class diagram. For each visual entity in PLAINA's environment there will be a terminal node in the syntactic tree. In class diagrams we have classes and arrows. Basically, the role of the terminal nodes is to generate code in the related class, and to be displayed on the screen. To provide this last issue, the terminal nodes that represent classes will have as attribute a frame record, that is, a record containing left upper and right bottom co-ordinates, and colour.

Frame for class:

left upper
right bottom
colour

As a class is always represented by boxes, thus two co-ordinates are sufficient to describe its view on the screen. The attribute colour is due to the fact that classes marked to be deleted get red colour. To be able of generating code, we have to put as classes attributes the name of the class and a kind of pattern that contains the almost empty code of the class: at the first moment the code contains only the class name and its nature. Thus, name and pattern are classes attribute

Arrow in the diagram will represent relationship between classes. So, let us name the terminal to represent this node relationship.

In case of relationship, we will also have as attribute the frame record, but at this time it will be composed only by source point and target point (the visual points of connection with the arrow and the boxes) and colour (again, unmarked are black, marked are red).

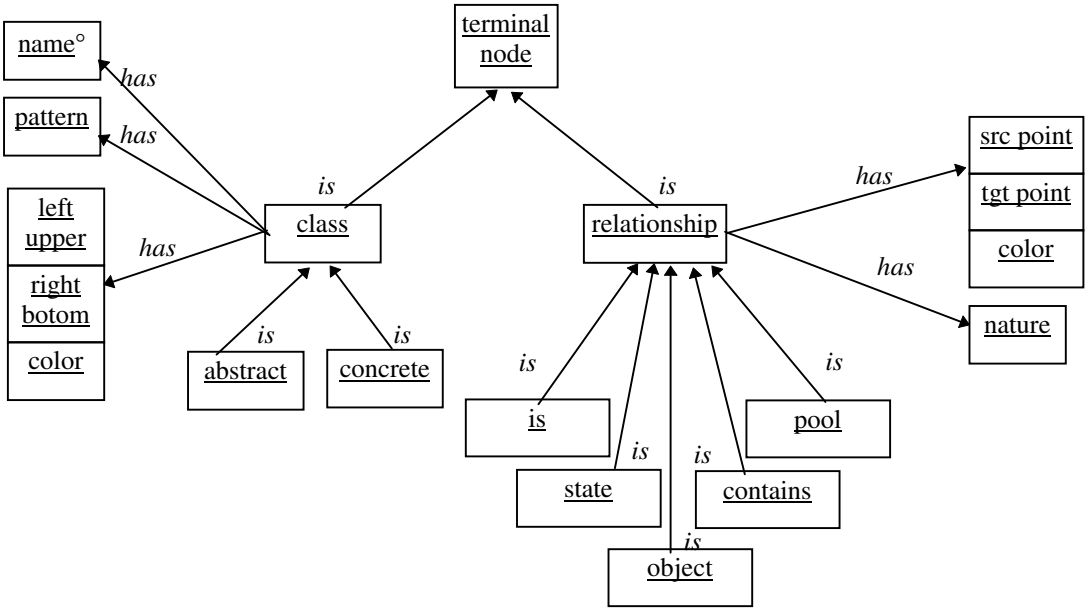
Frame for relationship:

src point
tgt point
color

Another attribute called nature will be necessary to complete the class code with the reserved word that this relationship represents.

We mark with an empty bullet the attributes whose contents are filled by means of an auxiliary dialogue window.

Using PLAINA notation to describe the class diagrams, we have



The above hierarchy shows some of PLAINA classes and possible relationships between them. As we can see, the node class is an abstract description of a class. Abstract and concrete nodes inherit all the features that are described in class. They also have some particular features, as they describe the actual implementations of classes. The same is valid to relationship: the node relationship describes the abstraction, and the lower nodes describe the actual implementation.
Considering, now, the reaction of the system to user's actions, we have to define two basic behaviours for class and relationship, that correspond to the action of placing those entities on the screen. So, beginning by class, we define create as virtual[7] behaviour of classes:

```
create;          { for abstract }
        BEGIN
        name :=  Input by dialogue box
        frame :=  Result of placing the mouse on the screen
        pattern := "ABSTRACT CLASS" || name || [inheritance clause] ||
                        [contains clause] || [pool structure] ||
                        [object structure] || [class state]  || "END CLASS"
        END

create;          { for concrete }
        BEGIN
        name :=  Input by dialogue box
        frame :=  Result of placing the mouse on the screen
        pattern := "CLASS" || name || [inheritance clause] ||
                        [object structure] || [class state]  ||
                        [contains clause] || [class body]  || "END CLASS"
```

[7]Virtual, in the sense that it exist in all inheritance hierarchy, but are only described in the leafs. The purpose of been virtual is the assurance that whenever we add a new node, we will have to define this behaviour for it.

```
        END
```

The symbol || indicates concatenation of strings.

In the case of relationship we will also have a virtual behaviour:

```
connect;                { for is }
        BEGIN
        frame := Result of placing the mouse on the screen
        nature := "is"
        END

connect;                { for state }
        BEGIN
        END

connect;                { for object }
        BEGIN
        frame := Result of placing the mouse on the screen
        nature := "object"
        END


connect;                { for contains }
        BEGIN
        frame := Result of placing the mouse on the screen
        nature := "contains"
        END


connect;                { for pool }
        BEGIN
        frame := Result of placing the mouse on the screen
        nature := "pool"
        END
```

Before showing the grammar, we will explain the role of non terminal nodes that will be used.

The non terminal class diagram is responsible for grouping relationships. Below a class diagram node in a syntactic tree, we can find the set of relationship that this node has, or in other words, all the arrows that leave it. Class diagram has a synthesised attribute class name, which contents is inputted by user and comes to their node via terminal class. An inherited attribute called target of will keep the name of the source of the relation at which this node is connected. If the class is not target of any relationship, the content of this attribute will remain empty. Remember that a node class diagram can be target of a unique relation, as we are using trees for representing relations. If a class is target of more then one relation, it's corresponding node class diagram will be replicated.

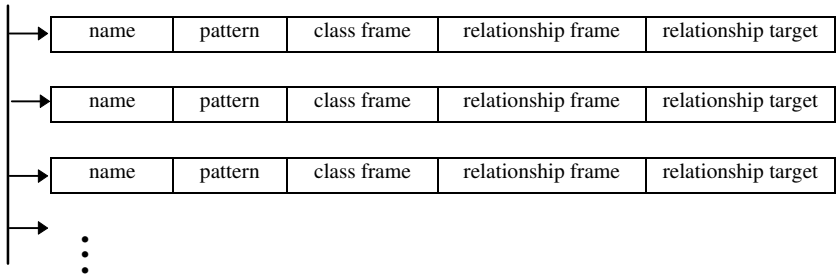
Set of relations is the non terminal responsible for grouping relationships. It has the attribute source that will keep the name of the class that is source of the relation. Its contents is inherited from class diagram.

Finally, we have relation, the non terminal that associates to the source class a relationship and another class. The inherited attribute source will permit the lower node class diagram to complete its target of information. We have also target, a synthesised attribute that will keep the name of the class below this node. It is the non terminal that collects all existent information about relations of a class. Note that any relation belonging to this set has the same source: the class connected to the class diagram above this relation. The set of attributes inheritance clause, contains clause, object structure and pool structure are to keep in separate each structure of the class. This attributes are filled by synthesised information.

Each non terminal will have attributes entry list, class view and code table. The first one, an inherited attribute, will collect a set of entries to update the others. Thus, each entry in entry list will have the following format:

name	pattern	class frame	relationship frame	relationship target

where pattern and relationship target are to fill in code table, and class frame and relationship frame are to fill in class view. Entry list is a list of the above records:



We will use the word *insert* to denote the action of inserting in entry list a new record containing only the information about classes: class name, class pattern and class frame.

entry list ← *insert* (name, pattern, frame).

As information about relationships comes later, we use the word *update target* and *update frame* to fill in an entry the relationship frame, and the relationship target. In case of pattern, it is necessary to add the newer structures or clauses to the code in entry list. So, we have *update pattern* to perform this task. The name of the class is needed to do the search in entry list.

entry list ← *update target* (name, target)
inserts in the entry of the class named "name"
entry it's corresponding relationship target name.

entry list ← *update frame* (name, target)
inserts in the entry of the class named "name"
entry it's corresponding relationship frame.

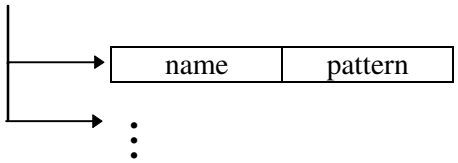
entry list ← *update pattern* (name, nature, target)
In the component pattern of entry list there are special places (underscored in the following code) to put clauses or structures. Clauses are unique, so having inserted one, there will not be any place to insert another of the

same type. Structures can be composed by a set of components, thus after inserting a structure, an empty place is created to put another, like is showed in the following code:

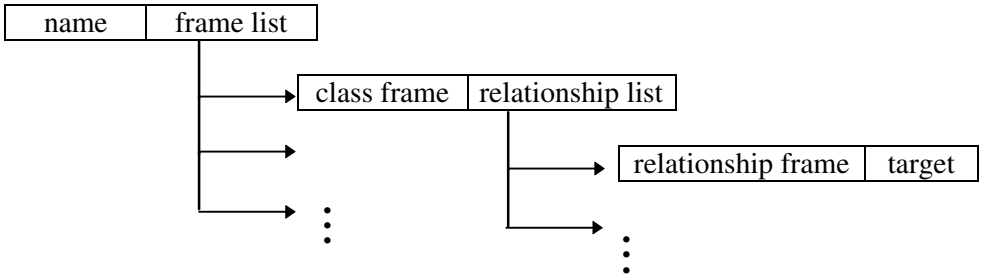
```
if nature is "is" then inheritance clause := "is " || target
if nature is "state" then
if nature is "object" then object struct := object struct  || target
if nature is "contains" then contains clause :=  "contains " || target
if nature is "pool" then pool struct := pool struct || target
```

In *update target*, let us consider that if class name is empty, no action occurs. This is a simple way of handling with the fact that the first class (the roof of the tree) is not target of any class.

The code table is indexed by class name, and keeps some information about classes code. Think in code table as a list as:



The class view, also indexed by the name of the class, will have a more complicated format:



As a class can appear more then one time at the diagram, we can have more than one frame for it. As an occurrence of a class on the diagram can have one ore more relationships leaving it, we will have a component relationship list that is a list of relationship frame (a record containing all the relevant information to draw the relationship), and target, the name of the class at the end of the relationship.

Let us use the name *insert* to denote the action of putting all entries in a new entry list in a table. The reverse operation, i.e., getting all the information from the table, and putting them in an entry list we will call *get*. Thus,

code table ← *insert* (entry list)
class view ← *insert* (entry list)
entry list ← *get* (code table)
entry list ← *get* (class view)

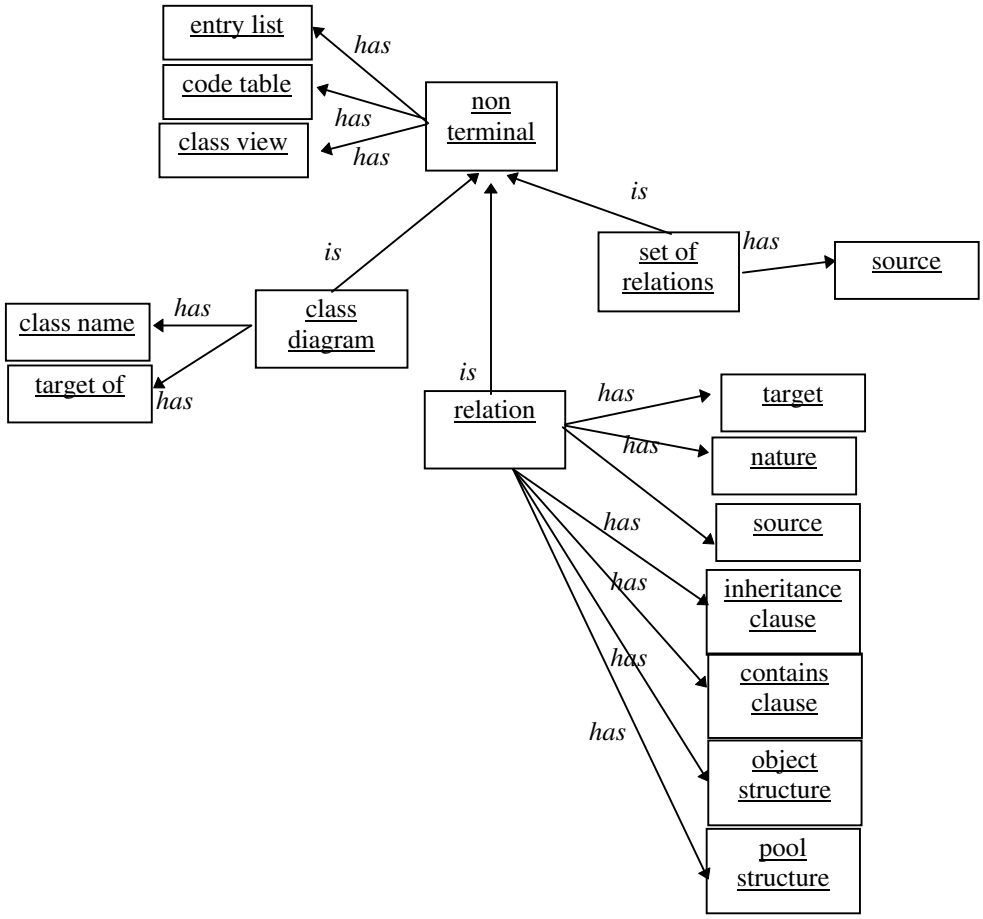
It is also necessary to define an operation to complete the pattern of a class with the key words indicating the beginning and ending of structures. This operation will act directly on the component pattern of an entry indexed by "name" of code table:

code table ← *finish pattern* (name);
    if object struct is not empty then
        object struct := "object"|| object struct || "end object" ;
    if contains clause is not empty then
        contains clause := "contains"|| contains clause ;
    if pool struct is not empty then
        pool struct := "pool"|| pool struct ;

Finally, we have an operation to join structure of the same nature:

code table ← *join* (code table)
class view ← *join* (class view)

We can think in the hole code table structure as a global environment, and the entry attribute from each non terminal as subtables that will compose the code table and the class view.



A Few words about dialogue boxes and visual information

It is necessary to consider that there are a lot of information that must be passed to the environment via dialogue boxes instead of graphical tools[8]. As simple examples, we can mention the names of classes, or components of classes. But it is also possible that, in order to avoid making incomprehensible diagrams, the user prefers to define class components by means of the *inspector* box. In spite of the fact that some of this information do not appear in the graphical interface, they are important to the code generation. As, our main goal in this formal description is to retract how to pass from diagrams to code, we will ignore the these constructions that do not came from the graphical tool. In section 0 we will see that there is a global structure that is responsible for keeping the code of classes. This structure will be the unique one affected by this kind of action. So, consider that there is a way of relating parts of the environment (like the *inspector*) with this structure, that ensures the consistence of the code.

[8] In the last section we used the empty bullet to emphasise them.

The grammar

Class Diagram

Abstract Syntax

We list below the abstract syntax of class diagrams.

< class diagram>	::= **Class** < set of relations >
< set of relations >	::= < relation > < set of relations >
< set of relations >	::= λ
< relation >	::= **Relationship** < class diagram >

The figure below shows na example of a tree and its corresponding diagram.

_____

Description of the attribute grammar

Now, we describe the attribute grammar. Considering that all the names we will use above are names of the attribute grammar, we will forget the underscores used to remark this names in the text, and pass to a usual notation. The symbol ":=" means attribution and "←"mean application of an operation.

< class diagram > ::= **Class** < set of relations >

< class diagram >.class name	←	**Class**.name
< class diagram >.entry list	←	insert (**Class**.name, **Class**.pattern, **Class**.frame)
< set of relations >.source	:=	< class diagram >.class name
< set of relations >.entry list	:=	< class diagram >.entry list
< class diagram >.entry list	←	update target (< class diagram >.target of, **Class**.name)
< class diagram >.code table	:=	< set of relations >.code table
< class diagram >.class view	:=	< set of relations >.class view
< class diagram > .code table	←	finish pattern(< class diagram >.class name)

< set of relations^1 > ::= < relation > < set of relations^2 >

< relation >.entry list	:=	< set of relations^1 >.entry list
< relation >.source	:=	< set of relations^1 >.source
< set of relations^2 >.source	:=	< set of relations^1 >.source
< set of relations^2 >.entry list	←	get (< relation >.code table)
< set of relations^2 >.entry list	←	get (< relation >.class view)
< set of relations^1 >.code table	←	join (< set of relations^2 >.code table)
< set of relations^1 >.class view	←	join (< set of relations^2 >.class view)

< set of relations > ::= λ

< set of relations >.code table	←	insert (< set of relations >.entry list)
< set of relations >.class view	←	insert (< set of relations >.entry list)

< relation > ::= **Relationship** < class diagram >

< relation >.nature	:=	**Relationship**.nature
< class diagram >.target of	:=	< relation >.source
< relation >.entry list	←	update frame(< relation >.source, **Relationship**.frame)
< relation >.entry list	←	update pattern(< relation >.source, **Relationship**.nature, **Relationship**.target)
< class diagram >.entry list	:=	< relation >.entry list
< relation >.code table	:=	< class diagram >.code table
< relation >.class view	:=	< class diagram >.class view

Example

Object Diagram

Briefly, we resume what is the object diagram.

Object diagrams consists of ellipses representing objects connected by arrows representing operations. The ellipse at the origin of the arrow is the object that requests the operation, and the ellipse at the end is the object that has to respond to the invocation. So, when the user connect two ellipses by an arrow, we have to create in the class of target object the code of the applied operation. The code consists only of the header, and possibly, the body of the operation. This information are passed via dialogue windows. The class of the source object will remains the same because the diagram do not offer sufficient information to decide where to put the request instruction, and where to define the created object (it can be local to an operation, local to the class representation, shared, etc.).

Abstract syntax and attribute description

Now, we return for the task of describing the attribute grammar. Beginning by the abstract syntax, we can see that the structure of the diagram is the same as the class diagram. There, we had boxes connected by arrows, and here, we have ellipses connected by arrows. Thus, let us use:

< object diagram >	::= **Object** < set of relations >
< set of relations >	::= < relation > < set of relations >
< set of relations >	::= λ
< relation >	::= **Operation** < object diagram >

The object terminal node has the attribute frame that is a record containing centre point, big ray, little ray and colour.

Frame for object:

centre point
big ray
little ray
colour

We have to keep in this terminal the name of the class this object belongs. This is informed by the user through a dialogue box at the moment the object is placed at the screen. So, let us add an attribute class. Finally, we will keep the name of the object in the attribute name.

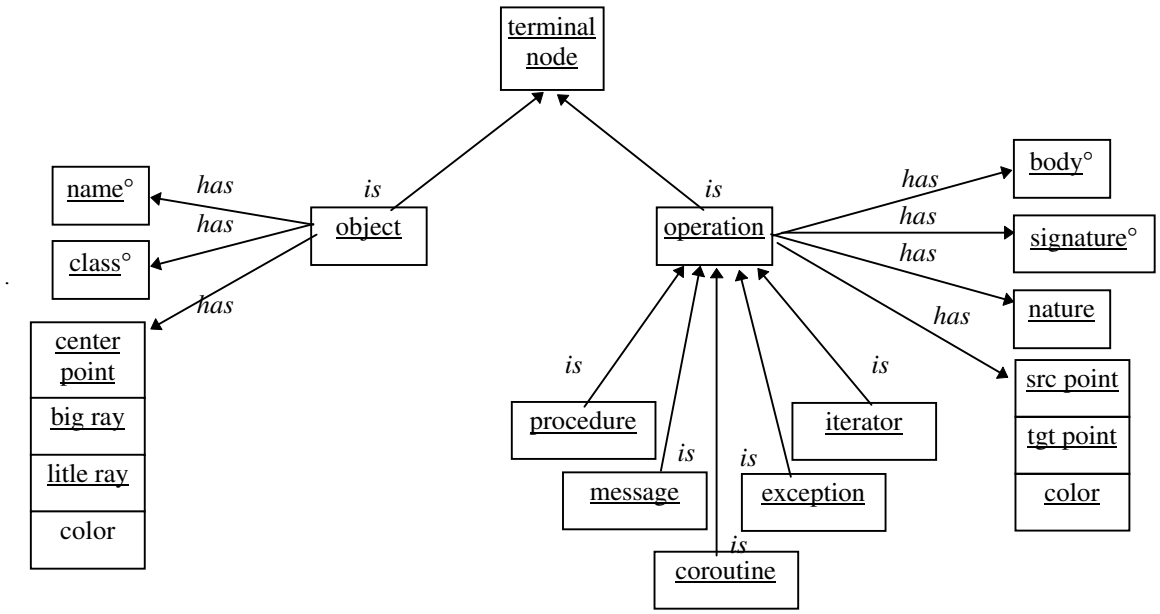
The terminal operation is almost the same as the class relationship. It has the same attribute frame, a record that contains source point, target point and colour, as shows the picture below.

Frame for operation:

src point
tgt point
colour

As operations can be procedure, message, exception, coroutine, or iterator, the terminal operation has an attribute nature. The attribute signature keeps the signature of each operation, and body keeps the code.

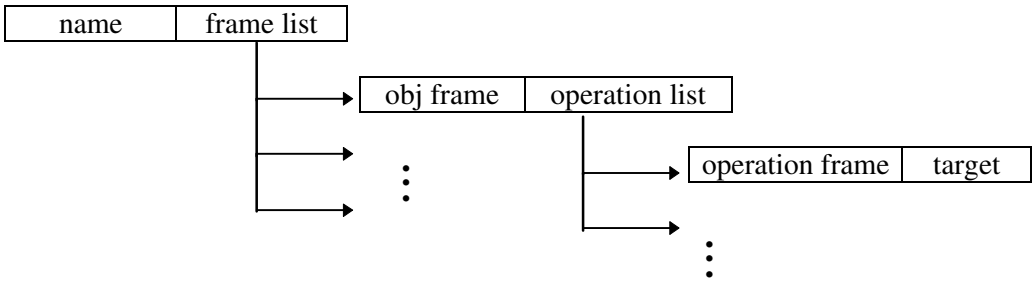
Following, we have the hierarchical description:



The virtual behaviours that describes the effect of placing one of this entities on the screen are create, for objects, and connect for operations:

```
create;          { for object }
        BEGIN
        name :=  Input by dialogue box
        frame :=  Result of placing the mouse on the screen
        class := Input by dialogue box
        END

connect;                 { for operation }
        BEGIN
        frame := Result of placing the mouse on the screen
        nature := "is"
        signature := Input by dialogue box
        body := Input by dialogue box
        END
```

Passing to the non terminals nodes, the non terminal object diagram has, as attributes class, name, code table, object view, entry list, nature, signature, body, target class and target of. The first, synthesised, is the name of the class of the object. The second is the name of the object. The third is the same table of the class diagram. This is an inherited attribute. Later, when defining the connection of class and object diagrams, we will clarify how it is passed from class diagram to the object diagram. The fourth attribute, object view, also synthesised, is similar to class view from the class diagram. It keeps information about how the object are drawn:



Entry list is an inherited list of records like the one below, where object frame and operation frame will be used in object view, and nature, signature and body will be used in code table.

class	name	object frame	nature	signature	body	operation frame

The component class is a key to find the correct class entry in the code table, and name is a key to find the correct object entry in object view. Having found the entry, its pattern will be completed with the information about the operation. We will use *insert operation* to denote these actions[9]:

[9] In *The Design Description Language* we can see that [class body] can have the following optional components: [pool structure], [object structure], [class state], [class messages], [class exceptions], [class constants] and [object behaviour]. This last one is composed by [class operation], which is by [procedure declaration] or [message handler declaration] or [coroutine declaration] or [iterator declaration].

code table ← *insert operation* (entry list) means:
For all entries in entry list, search the class in code table
in the component pattern:
if nature = "procedure" then
replace [procedure declaration]
by signature || "begin" || body || "end" || [procedure declaration]
else if nature = "message" then
.
.
.

The same will be done to put information of entry list in the object view table

object view ← *insert* (entry list).

Also for these two structures, it is necessary to define how to make the union of two structures of the same kind. So, let us define a *join* operation:

code table ← *join* (table)
object view ← *join* (view)

For the attribute entry list it will be necessary to describe how to put an entry, or entries information in the list. We have to define three operations, the first, using the object class and the object name, is responsible for creating the entry. The second, to update the entry. We are using two separated operations because in case of the root of the tree, that is not target of any operation, the *update* will have no effect (the arguments will be empty). Finally, the third operation is to update the frame part of entry list of an object that has an operation defined: as its target will be defined in lower nodes, it is necessary to search in entry list the incomplete entry with class named "class" and object name "incomplete obj" and complete its target object component name with "name".

entry list ← *insert* (class, name)
entry list ← *update* (class, name, nature, signature, body)
entry list ← *update target* (class, incomplete obj, name).

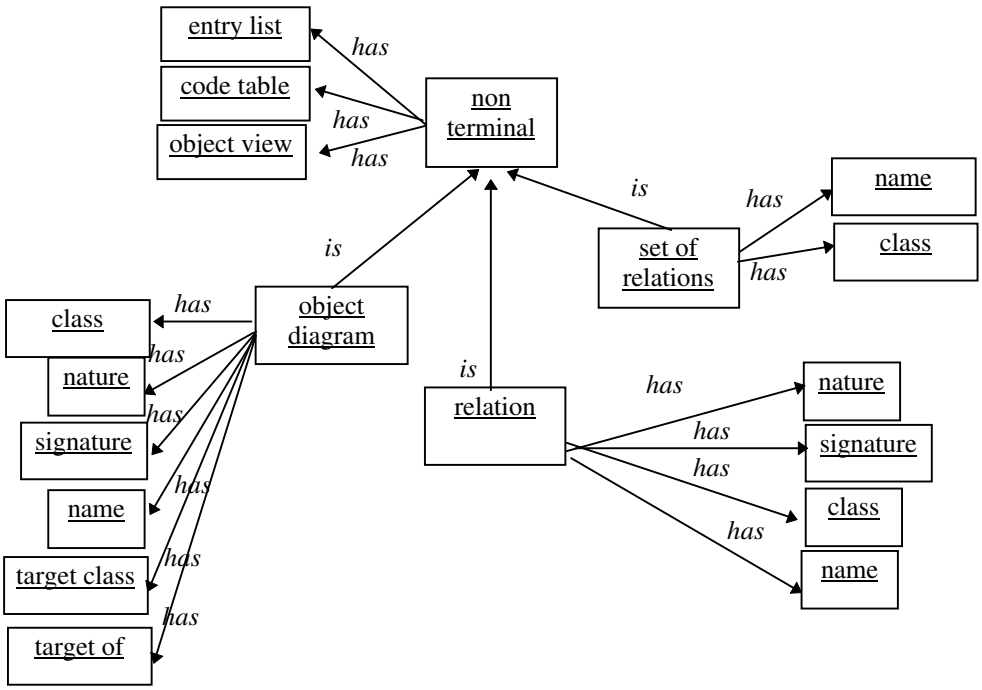
When it is necessary to recover information from code table and object view for entry list we will use:

entry list ← *get* (code table).
entry list ← *get* (object view).

To finish the description of the attributes of non terminal object diagram, the attributes nature, signature and body are to be inherited from relation, when this object diagram is the target of an operation. Target of is the name of the object of which this object diagram is the target of an operation. Target class is its class name.

In the non terminal set of relations keeps the name of the class in the inherited class, and the name of the object in the inherited name. Other attributes are object view, code table and entry list that make possible the arrival of these information in all the nodes .

In relation we have nature, signature, class and frame. All of them are passed from the terminal operation and will be inherited for the non terminal object diagram below this node. We have also object view, code table and entry list.



The figure below shows a three and its corresponding diagram.

Description of the attribute grammar

Following, we can see the attribute grammar

< object diagram > ::= **Object** < set of relations >

< object diagram >.class	:=	**Object**.class
< object diagram >.name	:=	**Object**.name
< set of relations >.class	←	< object diagram >.class
< object diagram >.entry list	←	insert (< object diagram >.class, < object diagram >.name)
< object diagram >.entry list	←	update (< object diagram >.class, < object diagram >.name, < object diagram >.nature, < object diagram >.signature, < object diagram >.body)
< object diagram >.entry list	←	update target (< object diagram >.target class, < object diagram >.target of, < object diagram >.name)

< set of relations >.entry list := < object diagram >.entry list
< object diagram >.code table := < set of relations >.code table
< object diagram >.object view := < set of relations >.object view

< set of relations^1 > ::= < relation > < set of relations^2 >
< relation >.entry list := < set of relations^1 >.entry list
< relation >.class := < set of relations^1 >.class
< relation >.name := < set of relations^1 >.name
< set of relations^2 >.class := < set of relations^1 >.class
< set of relations^2 >.name := < set of relations^1 >.name
< set of relations^2 >.entry list ← get (< relation >.code table)
< set of relations^2 >.entry list ← get (< relation >.object view)
< set of relations^1 >.code table ← join (< set of relations^2 >.code table)
< set of relations^1 >.object view ← join (< set of relations^2 >.object view)

< set of relations > ::= λ
< set of relations >.code table ← insert operation(< set of relations >.entry list)
< set of relations >.code view ← insert (< set of relations >.entry list)

< relation > ::= **Operation** < object diagram >
< relation >.nature := **Operation**.nature
< relation >.signature := **Operation**.signature
< relation >.body := **Operation**.body
< object diagram >.nature := < relation >.nature
< object diagram >.signature := < relation >.signature
< object diagram >.body := < relation >.body
< object diagram >.target of := < relation >.name
< object diagram >.target class := < relation >.class
< relation >.entry list ← update frame (< relation >.class,
< relation >.nature,
Operation.frame)
< object diagram >.entry list := < relation >.entry list
< relation >.code table := < object diagram >.code table
< relation >.object view := < object diagram >.object view

Example

No exemplo, chamar atenção para o fato de que as informaçoes que entram na classe entram no nível do object diagram. Elas são sintetizadas até este nó. Aí e´que se sabe, pelo objeto, qual é a classe em que isto vai entrar.

Connecting the diagrams

In the last sections we defined two levels of abstraction of PLAINA: the class diagram and the object diagram. These two diagrams are completely independent as visual entities: if the user chooses the option to construct object diagram he will not be able to add visual features in the class

diagram. The same occurs if he selects the option to construct class diagram: object diagram remains the same. But if the matter is code generation, it is clear that the two diagrams must be related: when we add an object to a class, it is obvious that we want this object to belongs to the code of that class. Thus, we can think in the code table as the actual global environment that has to be visible to all abstraction level. We can even say that the code table transcend this attribute grammar, as it must incorporate information that comes from different media, not only from the diagrams (see section 0 - A Few words about dialogue boxes and visual information).

Thinking in code table as the actual global environment, we have to ensure that it is inherited by object diagram in a consistent way. If there exist some classes defined, they have to be already in code table. If there is not any class defined yet, the action of defining one object must be preceded by a class creation. As we are not interested in information that comes from outside of diagrams, let us describe only the way of passing the code table constructed by class diagram to object diagram. We are, thus, considering that the class diagram is to be constructed first, although it is not real in PLAINA environment.

At the end of the process of computing attributes of class diagram, the synthesised attribute code table is completed, and available at the root. We have to pass this attribute to object diagram. To be more precise, code table has to be passed to the root of object diagram: an object diagram non terminal. But as even in object diagram, the attribute code table is synthesised, it is necessary to extract information of it and transfer to an inherited attribute, so that it can go down the syntactic tree and arrive at the leafs, where, with other information coming from different parts of the tree, they will compose the updated code table. To finish this process, the new code table will be synthesised to at the root.

Let us define a new non terminal called diagrams to connect the two diagrams and make possible to transfer information. Below, we list the new abstract syntax (the unique new line is the first):

< diagrams>	::= < class diagram> < object diagram >
< class diagram>	::= **Class** < set of relations >
< set of relations >	::= < relation > < set of relations >
< set of relations >	::= λ
< relation >	::= **Relationship** < class diagram >
< object diagram >	::= **Object** < set of relations >
< set of relations >	::= < relation > < set of relations >
< set of relations >	::= λ
< relation >	::= **Operation** < object diagram >

Following, we have the attributed rules related to the new line:

< diagrams > ::= < class diagram > < object diagram >
< object diagram >.entry list ← *get* (< class diagram >.code table)

Connecting the examples

_____

A degree of abstraction: from a diagram to a code

Let us consider now that we have only a diagram. A diagram is composed by nodes that have its semantics determined by its visual form. From each node we can have a set of arrows leaving it and connecting it to other nodes. The semantics of arrows is determined by its nature. We do not know at this time which is the semantic to be related to each symbol, i.e., which is the code to be generated for each entry. We do not even know how is the visual representation of each node. So, our aim now is only to construct the more abstract structure that can describe how to generate code from a diagram (or a collection of diagrams).

Abstract syntax and attribute definition

Looking at a diagram as a set of nodes, each one having a set of arrows, connecting it to another node, the abstract grammar to describe the diagram is showed below:

< diagram >	::= **Node** < set of relations >
< set of relations >	::= < relation > < set of relations >
< set of relations >	::= λ
< relation >	::= **Arrow** < diagram >

If it is the case of describing levels of abstractions within a language, we can use a set of different diagrams, each one of a level, all of them having the same structure. It may be necessary to communicate some attributes between them.

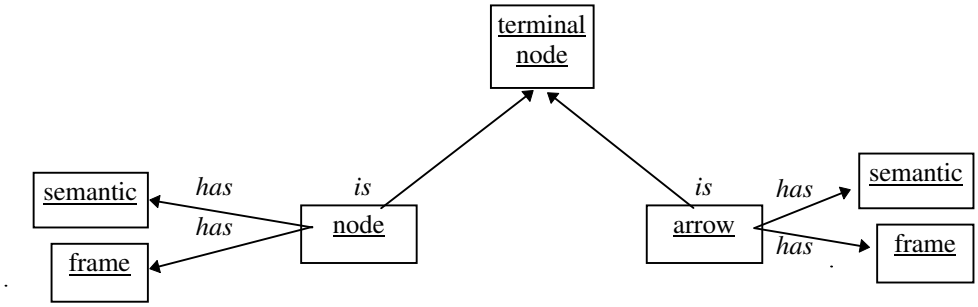
We want this description the more abstract as possible, so in the following sections we will say nothing about particularities of nodes or arrows. The description will be complete enough to describe general entities that may be node of a diagram or that may connect nodes.

Using the same approach as before, let us start the description of this diagram by the terminal nodes node and arrow. We can group the attributes of the terminal nodes into two different domains: the visual domain, which contains attributes related to visual representation, and semantic domain, that keeps attributes that will participate in the semantic of the entity. The former we will call frame. Thus, frame is a record containing such visual information as co-ordinates, colour, kind of geometric figure, etc. Let us call semantic the record that contains information coming as a result of an users action, as positioning an entity on the screen, inserting a name via keyboard. Information that comes from the keyboard can be the name of an entity, some indicator of the kind of action that this entity performs, an item to explicitly compose the entity's code, like the body of an operation.

Despite of the fact that the attributes mentioned above have the same name for each terminal (semantic for node and for arrow, frame for node and for arrow) and that they play the same role (a semantic role or a visual role), they are essentially different in their internal structure. This is expected, as the information needed, for example, to draw an arrow can not be of the same kind of the ones to draw a box. To emphasise the semantic role it would be a good solution to place these attributes at the more abstract level terminal node, as "virtual attributes"[10] and let them be inherited by node and arrow, were they would be specialised. But as we want to give our structure the capacity of been infinitely specialised, we wont say that node and arrow are leafs of this hierarchy. Instead of this, we want the user to define entities that will be specialisation of node and arrow. If we add specialised nodes of node and arrow, maintaining semantic and frame as virtual, we will

[10] Virtual entities are specified at the leafs.

delegate to this new leafs the task of defining semantic and frame. Let us define semantic and frame at the level of node and arrow as virtual nodes, and let the leafs specify details. The attribute grammar will be written in term of the abstract role of semantic and frame. Particular features will appear only in the body of the operations that deal with these attributes.



As we did before, it is necessary to define some operations that must be performed at the moment that a terminal node is placed on the screen. These operations will set frame co-ordinates, colour, etc., and will also make the request of user inputting information. Consider, thus,

```
create;      { for node }
connect;     { for arrow }
```

two behaviours, whose code depends on the application considered.

The non terminal node and arrow will have three common attributes named entry list, code table and view. The first attribute is an inherited one and has the role of collecting information from the upper parts of the syntactic tree, making them available in the lower nodes. This information can belong to the semantic domain, or visual domain. Entry list acts like a bus of information, and it's internal structure is completely dependent on the application.

The two other attributes code table and view, are a kind of environment attributes. But, as code table is the one that belongs to the semantic domain, it is the actual global environment attribute. As a consequence it has to be passed to the other diagrams, if they exist. Information belonging to visual domain are local to a diagram and do not have to be passed. Code table and view are filled with information extracted form entry list and are synthesised attributes.

For these three attributes, we can define a set of operations that will make clear the description of the hole attribute grammar:

entry list ← *insert* (name)
creates a new record in the list with a key "name"

entry list ← *update component* (name, component)
update a specific component named *component* of all the entries having "name" as key. This is actually a set of operations.

entry list ← *get* (code table)
for all entries in code table extract the components that are used in entry list.

entry list ← *get* (view)
for all entries in view extract the components that are used in entry list.

code table ← *update component* (name , component)
update a specific component named *component* of all the entries having "name" as key.
As in entry list, this is a set of operations.

code table ← *join* (table)
makes the union of two code tables

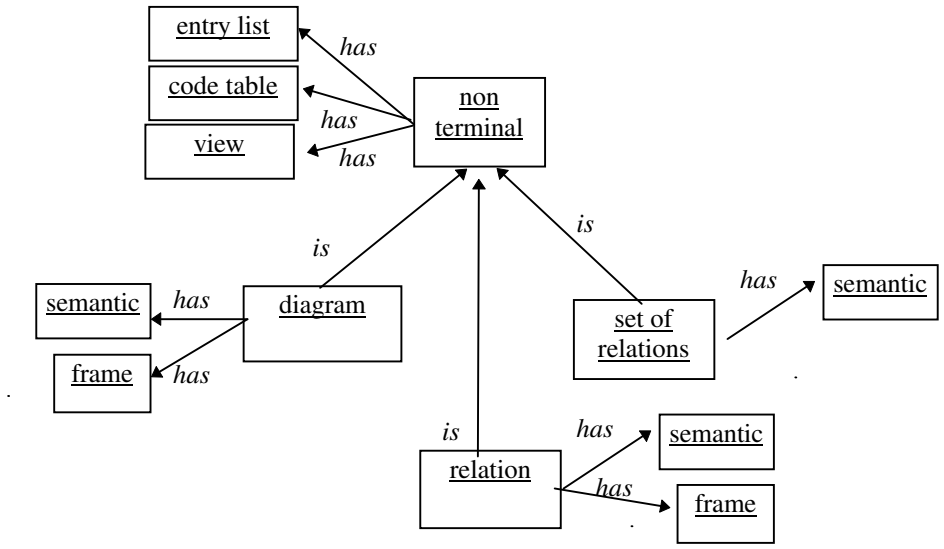
code table ← *insert* (entry list)
for all entries in entry list extract the components that are used in code table.

code table ← *finish* (name)
make the last adjustments in the code of the class named "name".

view ← *insert* (entry list)
for all entries in entry list extract the components that are used in view.

view ← *join* (view)
makes the union of two views

Particular non terminals have particular attributes according to their physical connection to terminals. For example, diagram has a closely connection with node, and as so, it has to have some attributes to receive semantic and visual information from this terminal and pass it to entry list. The attributes frame and semantic of the non terminal are defined in a very similar way of the same ones of the terminal nodes. There is no need to maintain the attribute frame in the non terminal set of relation because visual information is not to be transformed during the walking in the tree. They begin and finish at the same way as they entered by the terminal node.



Description of the attribute grammar

< diagram > ::= **node** < set of relations >

1.	< diagram >.semantic	←	collect (**node**.semantic)
2.	< diagram >.frame	←	collect (**node**.frame)
3.	< set of relations >.semantic	:=	< diagram >.semantic
4.	< diagram >.entry list	←	insert (< diagram >.semantic ← get component)
5.	< diagram >.entry list	←	update component (< diagram >.semantic ← get component, < diagram >.semantic ← get component)
6.	< diagram >.entry list	←	update component (< diagram >. semantic ← get component, < diagram >.frame ← get component)
7.	< set of relations >.entry list	:=	< diagram >.entry list
8.	< diagram >.code table	:=	< set of relations >.code table
9.	< diagram >. view	:=	< set of relations >.view

1,2) The semantic/visual information coming from node will be necessary to fill entry list. The operation *collect* make them available in diagram, where they can be passed to entry list. Note that semantic is a set of attributes. At this step, part of this attribute is filled by node information. The other part is left empty, or preserve inherited contents.

3) The semantic information must also be passed to set of relation to be available to lower parts of the tree.

4) Among semantic component of diagram.semantic, there must be a specific one that serve as key to all the structures that walks in the three. An specific operation *get component*, where the

component will be one that act as a key, will return a name and the first entry for it will be created in entry list.

5,6) A set of operation (one for each component) is performed. Some components of entry list are filled with specific components of diagram.semantic and other are filled with specific components of diagram.frame. The first argument of *update component* must be a key for indexing entry list.

7) The inherited entry list is passed to lower parts of the tree.

8,9) Having been already completed with synthesised information, code table and view arrive in diagram. As this non terminal represents a complete class diagram it may be necessary to make the last adjustments in the code. This is the task of *finish*.

< set of relations^1 > ::= < relation > < set of relations^2 >

1.	< relation >.entry list	:=	< set of relations^1 >.entry list
2.	< relation >.semantic	:=	< set of relations^1 >.semantic
3.	< set of relation^2 >.semantic	:=	< set of relations^1 >.semantic
4.	< set of relations^2 >.entry list	←	get (< relations >.code table)
5.	< set of relations^2 >.entry list	←	get (< relations >.class view)
6.	< set of relations^1 >.code table	←	join (< set of relations^2 >.code table)
7.	< set of relations^1 >.view	←	join (< set of relations^2 >.view)

1,2,3) The attributes entry list and semantic are passed down to relation. The lower set of relation also receives semantic, but not entry list. This last attribute will be modified by the relations and classes below relation before be passed to the right set of relation.

4,5) Having computed all operations below relation, code table and view will be ready to be passed to the right side of the tree, the node set of relation. The right side of the tree is beginning to be computed.

8,9) Finally, the right side is already computed, and the newer information arriving in code table and view by the right side are joined with the ones coming from the left side.

< set of relations > ::= λ

1.	< set of relations >.code table	←	insert (< set of relations >.entry list)
2.	< set of relations >. view	←	insert (< set of relations >.entry list)

1,2) This is the and of the way down the tree. Attributes will begin to be inherited. It is necessary to extract information from entry list and pass them to the correct structures.

< relation > ::= **arrow** < diagram >

1.	< relation >.semantic	←	collect (**arrow**.semantic)
2.	< relation >.frame	←	collect (**arrow**.frame)
3.	< diagram >.semantic	:=	< relation >.semantic
4.	< relation >.entry list	←	update component(
			< relation >.semantic ← get component,
			< relation >.semantic ← get component)
5.	< relation >.entry list	←	update component(
			< relation >.semantic ← get component,

< relation >.frame ← get component)

6.	< diagram >.entry list	:=	< relation >.entry list
7.	< relation >.code table	:=	< diagram >.code table
8.	< relation >. view	:=	< diagram >.view

1,2,3) Another time, *collect* will make available for the non terminals the information coming from terminal nodes.

4,5) The operation *update component* will transfer the semantic and visual information to entry list. The first *get component* will return a component of semantic to serve as key for indexing entry list. The second *get component* is to return the desired component.

6) The inherited entry list continue its way down the tree.

7,8) The synthesised code table and view continue its way up the tree.

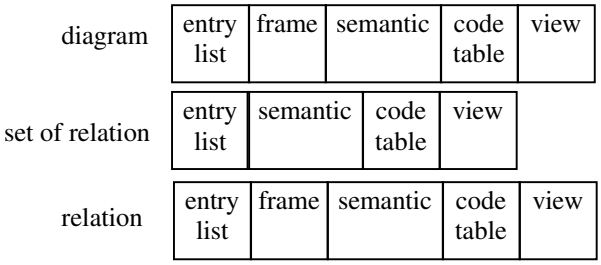
Proof of strong non-circularity

Following we will construct the proof of strong non-circularity that ensures that we can evaluate attributes according to some partial order. Having defined the order, a recursive procedure can be constructed to compute the attribute grammar.

For each non terminal of the grammar we will define a graph, **dependence graph of the symbol**, that will show the dependence among attributes of the same symbol.

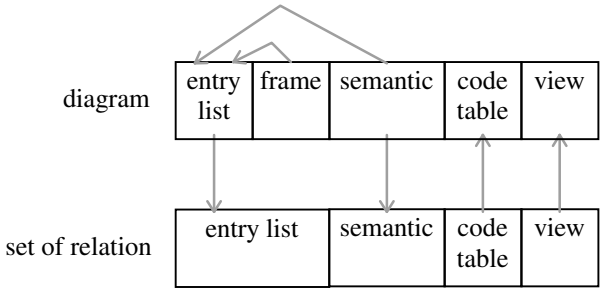
For each rule of the abstract syntax we will construct another graph, the **dependence graph of the rule**. This graph has as nodes the non terminals of the rule and directed arrows from linking two nodes if an attribute of the target node depends on an attribute of the source node to be computed. The **dependence graph of the rule** will be used to show the dependence among attributes of the non terminal at the left side of the rule and attributes of non terminals at the right side of same rule. This graph will be modified to form the **augmented dependence graph of the rule**, that will show the dependence among attributes of the non terminal at left side of the rule and the attributes of non terminals at the right side of same rule, but involving the others rules of the grammar.

Beginning by the **dependence graph of the symbol**, we have, initially none arrows connecting the attributes of the symbols:

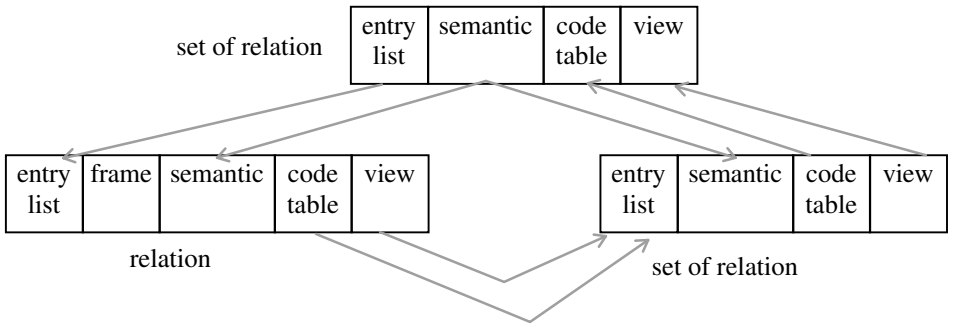


Considering, now, the **dependence graph of the rule**, we have for each rule:

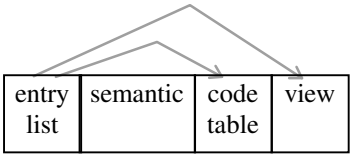
< diagram > ::= **Node** < set of relations >



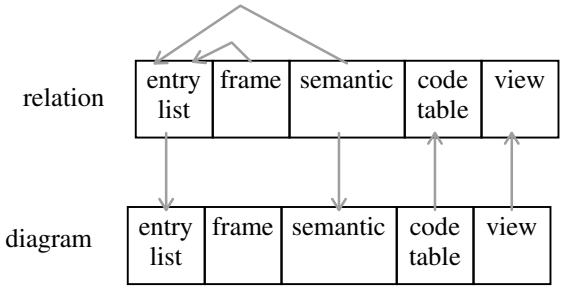
< set of relations > ::= < relation > < set of relations >



< set of relations > ::= **λ**



< relation > ::= **Arrow** < diagram >



Now, we can match the two graphs using the following steps:

i) For each **dependence graph of the rule**:

Considering *A*, the non terminal at the left side of the rule, add an arrow from *x* to *y* in **dependence graph of the symbol** *A* if *x* and *y* are attributes of *A*, and there is a path from *x* to *y* in **dependence graph of the rule**.

ii) For each **dependence graph of the symbol**:

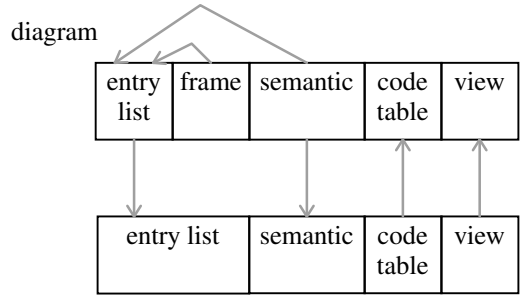
If an arrow from *x* to *y* was added in **dependence graph of the symbol** for the non terminal *A*, add an arrow from *x* to *y* in **dependence graph of the rule** for each rule having *A* in the right side.

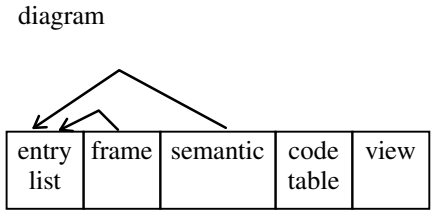
iii) Repeat the above steps until none arrow is added.

This process is showed bellow. The first column is the **dependence graph of the rule**, and the second is the **dependence graph of the symbol**.

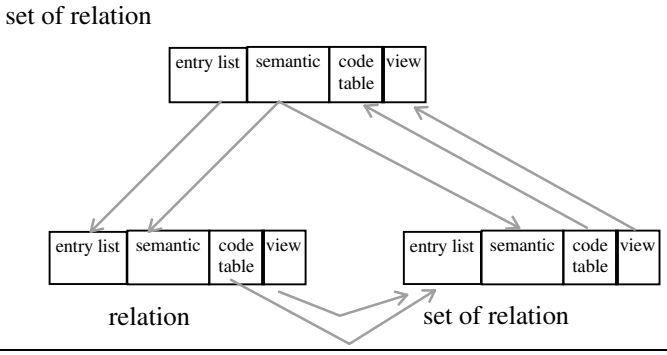
(*i*) Each black arrow in the **dependence graph of the symbol** represents a path in the **dependence graph of the rule** that has this symbol as left non terminal. The path has as source an attribute of the symbol and as target another attribute of the same symbol.

< diagram > ::= **Node** < set of relations >

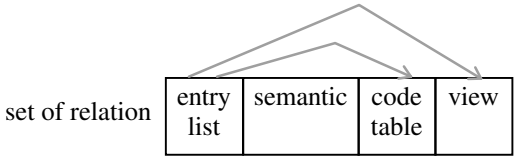


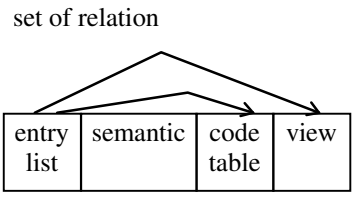


< set of relations > ::=
< relation > < set of relations >

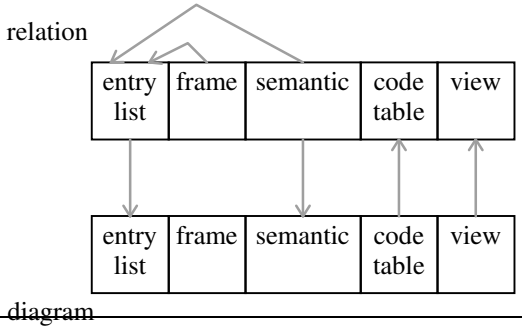


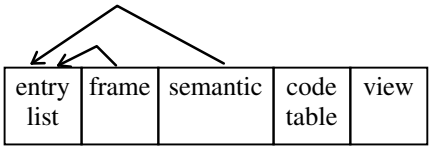
< set of relations > ::= **λ**





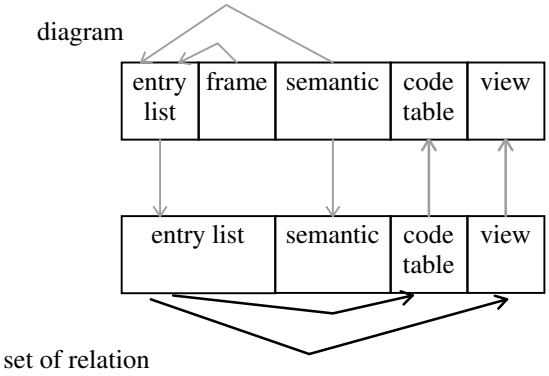
< relation > ::= **Arrow** < diagram >

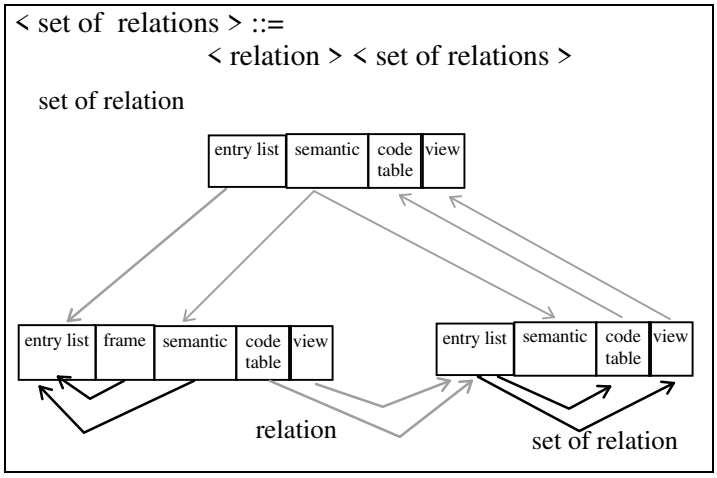


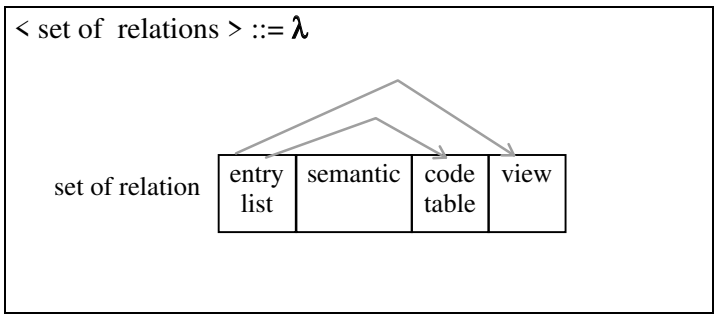


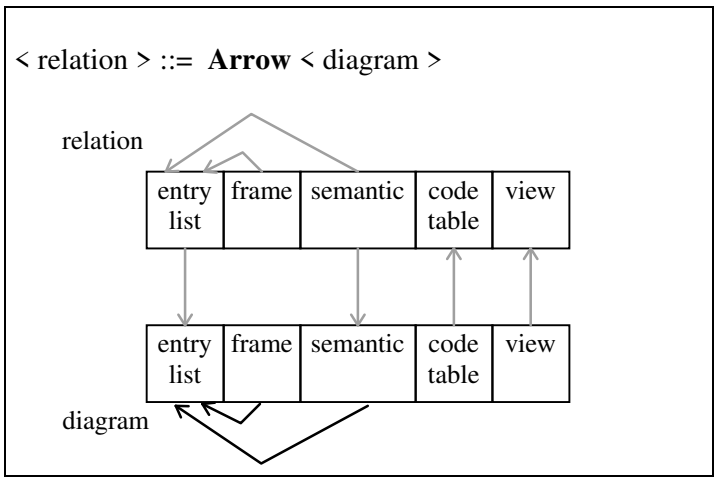
(*ii*) Each **dependence graph of the symbol** is reverted to each **dependence graph of the rule** that has this symbol as right non terminal.

< diagram > ::= **Node** < set of relations >









(*iii*) Considering the **dependence graph of the rule**, there is not any different path from an attribute of a left non terminal to another attribute of the same non terminal. Thus there is not any node to add on the graphs. The process is concluded.

As we can not find a cycle in any graph, we can say that the attribute grammar is strong noncircular. An order of evaluation for the attributes for each node can be directly deduced from the possible paths in each graph. Below we show one possible order:

< diagram > ::= **Node** < set of relations >
< diagram >.semantic
< diagram >.frame
< set of relations >.semantic
< diagram >.entry list
< set of relations >.entry list
< set of relations >.code table
< set of relations >.view
< diagram >.code table
< diagram >.view

< set of relations^1 > ::= < relation > < set of relations^2 >
< set of relations^1 >.entry list
< set of relations^1 >.semantic
< relation >.entry list
< relation >. semantic
< set of relations^2 >.semantic
< relation >.frame
< relation >.code table
< relation >.view
< set of relations^2 >.entry list
< set of relations^2 >.code table
< set of relations^2 >.view
< set of relations^1 >.code table
< set of relations^1 >.view

< set of relations > ::= λ
< set of relations >.semantic
< set of relations >.entry list
< set of relations >.code table
< set of relations >.view

< relation > ::= **Arrow** < diagram >
< relation >.semantic
< relation >.frame
< diagram >.semantic
< relation >.entry list
< diagram >.frame
< diagram >.entry list
< diagram >.code table
< diagram >.view
< relations >.code table
< relations >.view

The recursive evaluator

In section 0, we showed how to extract a partial order to evaluate attributes from the dependence graph. Considering this order, it is easy to construct a procedure to evaluate the grammar: a recursive evaluator. This procedure is a walking in the syntactic tree, where the path is determined

by the order. In some case it may be necessary to visit a node more than one time, each time a set of attributes will be ready to be calculated. The recursive evaluator must have an extra parameter to indicate the current visit. In the grammar presented in the above sections, this is not needed.

Consider a recursive procedure wich has as parameters the node being visited and the number mentioned above. The body of the procedure will be a selection structure that, depending on the current node, computes the local attributes - the ones whose attributes are already disponible in the node or in its children nodes - and makes recursive calls to it´s children to compute values that have to go up from lower nodes to the current one. The procedure will be called having the root of syntactic tree as first node. When going down the tree, inherited attibutes can be evaluated and when going up the tree, synthesised attributes are evaluated. Attributes that depends on others attributes that are not yet computed are delayed to a next visit.

Below, we list the code of a recursive evaluator for the grammar described above. Let us use the name "pt" to denote a pointer to a node of the syntactic tree. Each node of the syntact tree has the form

left child	rule	entry list	frame	semantic	code table	view	right child

where *rule* is the number of the rule that expands a nonterminal. If the rule has two symbols at the right side, the first one will be pointed by *left child*, and the second one by *right sid*e. If the rule has a unique child, *rigth child* will be null. For simplicity, we are considering the same representation for all the grammars nodes, althougt some of them do not have all these attributes.

The procedure will be first called at the root of syntactic tree.

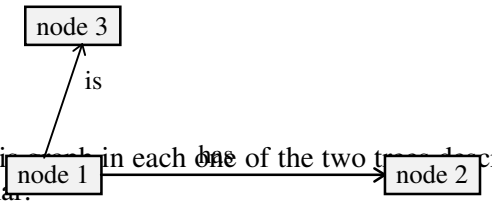
```
Procedure evaluate( pt )
case pt.rule = 1 then      /* pt points to a diagram node */
        pt.semantic ← collect( pt.left_child.semantic )
        pt.frame ← collect( pt.left_child.semantic )
        pt.right_child.semantic := pt.semantic
        pt.entry_list ← insert ( pt.semantic ← get component )
        pt.entry_ist ← update component ( pt.semantic ← get component,
                                          pt.semantic ← get component )
        pt.entry_list ← update component ( pt.semantic ← get component,
                                          pt.frame ← get component )
        pt.entry_list ← insert ( pt.semantic ← get component )
        pt.right_child.entry_list := pt.entry_list
        evaluate( pt.right_child )     /* to compute synthesised attributes */
        pt.code_table := pt.right_child.code_table
        pt.view := pt.right_child.view

case pt.rule = 2 then      /* pt points to a set of relation node */
        pt.left_child.entry_list := pt.entry_list
        pt.left_child.semantic := pt.semantic
        pt.right_child.semantic := pt.semantic
        evaluate( pt.left_child )     /* to compute synthesiseded attributes from relation*/
        pt.right_child.entry_list ← get ( pt.left_child.code_table )
```

```
        pt.right_child.entry list ← get ( pt.left_child.class view )
        evaluate( pt.right_child )    /* synthesiseded attributes from set of  relation */
        pt.code_table ← join ( pt.right_child.code_table )
        pt.view ← join ( pt.right_child.view )

case pt.rule = 3 then     /* pt points to a set of relation node */
        pt.code_table ← insert ( pt.entry_list )
        pt.view           ← insert ( pt.entry list )

case pt.rule = 4 then     /* pt points to a relation node */
        pt.semantic ← collect ( pt.left_child.semantic )
        pt.frame ← collect ( pt.left_child.frame )
        pt.right_child.semantic := pt.semantic
        pt.entry_list ← update component( pt.semantic ← get component,
                                          pt.semantic ← get component )
        pt.entry_list ← update component( pt.semantic ← get component,
                                          pt.frame ← get component )
        pt.right_child.entry_list :=  pt.entry_list
        evaluate( pt.right_child )    /* synthesiseded attributes from diagram */
        pt.code_table := pt.right_child.code_table
        pt.view := pt.right_child.view
```

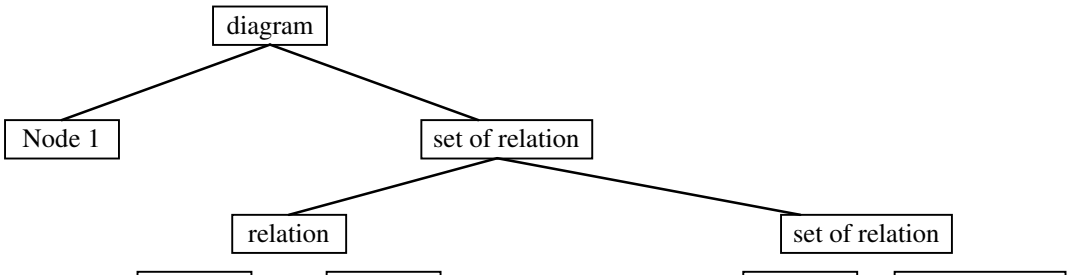
Recursive evaluators are to be applied to a constructed syntactic tree. Because of this, we can say that the attribute grammar is independent of the parser. This independence is very important to our case, where the entity beeing parsed is not a text, but a graph. It is easy to see that the grammar specified above is ambiguous, and as so, can not be parsed by the usual techniques. An special way of doing the parsing is explained on the next section, where the order of construction will be considered to eliminate the ambiguity.

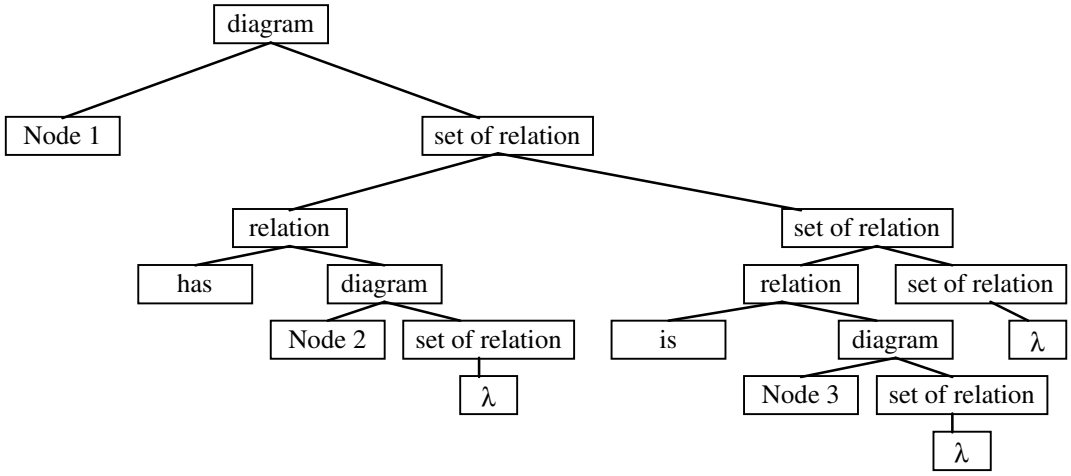
How to construct the parser

The grammar described above is able to generate sequences of altenating nodes and arrows. The output generated by this grammar does not gives the idea of a diagram. But if we analise the syntactic tree that is the result of expanding rule by rule we can see that this tree is able to describe a graph. The problem is that, given a graph, there is more than one tree that can represent it. For example, consider the graph



We represent this graph in each one of the two trees described below, and both of them are derived from the grammar:





Note that these two trees are very simple because node 2 and node 3 do not have any relationship. But when others relationships are considered, we can have very different trees representing the same graph.

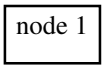
As we are giving semantics to an interative constructed diagram, and as it is not possible for the user to insert two entities of a graph at the same time, the obivious way of selecting the tree that will represent the diagram is to consider the order of construction. We can also think about the possibility of having two incomplete diagrams that can be connected by an arrow. Also in this case, each diagram will have its order of construction, so, the tree to be generated by the connected diagram is a junction of their trees.

Bellow, we represent the process of generating the tree for a graph. The actions of the user are considering on the right box, and on the left we represent the effect of these actions on the tree that is being created.

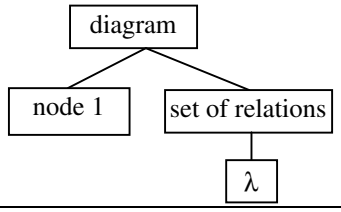
First, there is a screen providing all the tools to construct diagrams, but without any diagram: the user will define the first one.	The tree is empty.

He begins by inserting a node picture (to begin with an arrow, he would have to insert the source and target that do not exist yet). He selects a node picture, specifying its semantics attributes, say the name, and place it on the screen.

In the tree we have the creation of a terminal node.

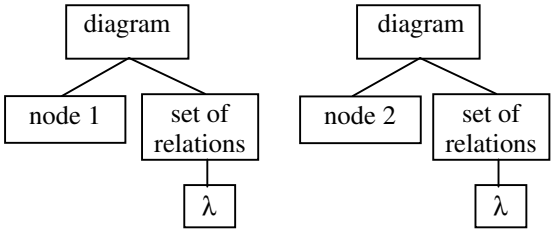


But, as the trees begin by a non terminal < diagram >, this sigle node is transformed in:



The user wants, now, to make a relationship. But for doing this, he will have to define another node.

Now we have two trees:



At this time, the user can stablishes a relationship: he puts an arrow from node 1 to node 2.

After this step, if it is the case of inserting one more relationship, another diagram will be creater as effect of placing the target of the relationship on the screen. If the new node has to have as source the node 1, the tree will be expanded bellow the dotted < set of relation >. But if the source is node 2, the tree will grow bellow the lower < set of relation >.

The part of the tree created as an effect of putting the new arrow is remarked with dotted line.