**Vrije Universiteit Brussel**
**Faculty of Sciences**
**In Collaboration with École des Mines de Nantes**



# Design and Implementation of a Reflective Hybrid Functional/Prototype-Based Language Kernel

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange
project funded by the European Community)

By: Sebastián González

Promotor: Prof. Theo D'Hondt (VUB)
Advisor: Wolfgang De Meuter (VUB)

August 19, 2002

**Design and Implementation of a Reflective Hybrid Functional/Prototype-Based Language Kernel**

---


## Abstract

A frequent dilemma in programming language design is the choice between a language with a rich set of notations and features, and a small, simple core language. In this work, the dilemma is addressed by proposing a syntactical and semantical extension mechanism based on the programming language Pic% developed at VUB. The result is a language whose kernel is basically a reduction of Pic%'s. This kernel defines a very simple functional language whose core syntax and semantics can be grown as needed. The extension process is performed from within the language itself, using a set of reflective facilities, and such process is dynamic (i.e. performed at run-time), thus enabling the programmer to use new constructs as soon as they are defined. The tools provided are: a small and well-defined core semantics, an evaluation environment playground, and a set of meta-level functions which conform the meta-level interface (MLI) of the language. The MLI contains utilities to extend the language grammar and assign a semantics to each extension. With these tools, it is shown that a complete and consistent object-oriented model can be defined by means of syntactic extension and assignment of the corresponding semantics. The OO layer is inspired on prototype-based programming, thus the language, called xPic% (*extensible Pic%* ), is transformed to be both a functional and a prototype-based programming language. As far as we know, an MLI for this kind of combination is unique.

Word count: 251

# Acknowledgements

I would like to gratefully acknowledge the supervision of Wolfgang De Meuter. He kindly accepted to propose additional subjects so that I could come to work at VUB. The subject he came up with fitted my personal interests and I can say that I not only enjoyed working on it but also learned a lot. Wolf came up with the right advises at the points of time where the work required them. Many of his ideas are present in this work. He also revised the text and gave some good writing tips which improved the overall readability of this dissertation.

The second big thank you is for Theo D'Hondt, a very open-minded person whose spirit gives impulse to the PROG lab and the EMOOSE program. Much of the work in this thesis is based on his work, more specifically, a lot of inspiration was drawn from Pic%, a language he designed and implemented. Finally, Theo obtained economical help for us, the three emoosers that came to the VUB this year, to go as student volunteers to ECOOP'02.

For the big effort of supporting the EMOOSE program I thank Theo D'Hondt, Annya Romanczuck and Jacques Noyé. The EMOOSE has an excellent academic level. Its organization (I suppose) is a difficult achievement given the international nature of the program and the collaboration of two different institutions.

I specially thank Isabel Michiels. Isabel (ex-emooser) embraced us as if we have known her from long time ago. She helped us with many administrative tasks regarding our accommodation in Brussels, economical help from the VUB and more.

Thanks to Johan Fabry for offering himself as a victim, erm... proof-reader of this dissertation. He commented out some parts of it. Thanks to all the other guys at the PROG lab for their friendly attitude which made the working environment very nice (lunch time was specially fun :-)

I shared this five months of hard work and lots of fun with Agus, Boris and Raül; so thanks guys for the nice moments together. For the same reason I thank also Isabelleke, Tielke, Sijke, Saartje and Miro.

I thank all the emoosers with whom I had an excellent time in Nantes, during the first half of the master. They know how a hard time we had but still how fun was to be together in such a nice context: people from many countries, an impressive board of lecturers coming from many universities, a nice university (École des Mines de Nantes) and a nice city (Nantes).

To thank my father, Sonia and my brother, words are not enough. My gratefulness is of a nature which I cannot describe but just feel.

# Contents


**1 Introduction** **1**
- 1.1 Objective . . . 2
- 1.2 Methodology . . . 2
- 1.3 Context . . . 2
  - 1.3.1 Advocation of prototype-based programming . . . 3
  - 1.3.2 Exploration of the design space of prototype-based languages . . . 3
  - 1.3.3 Research in the design of a language for educational purposes . . . 3
  - 1.3.4 Programming and meta-programming with multiple paradigms . . . 4
- 1.4 Scope of the work . . . 4
- 1.5 Outline of this dissertation . . . 5
  - 1.5.1 Roadmap for the busy reader . . . 5

**2 Background Concepts** **7**
- 2.1 Meta-programming and computational reflection . . . 7
  - 2.1.1 Basic concepts . . . 7
  - 2.1.2 Meta-programming . . . 8
  - 2.1.3 Computational reflection . . . 8
  - 2.1.4 Reflective architectures . . . 10
  - 2.1.5 Conclusion . . . 11
- 2.2 Prototype-based programming . . . 11
  - 2.2.1 Philosophical background . . . 12
  - 2.2.2 Limitations of class-based programming . . . 14
  - 2.2.3 Metaphors & mechanisms in prototype-based programming . . . 15
  - 2.2.4 Class-based vs. prototype-based languages . . . 18
  - 2.2.5 Conclusion . . . 20
- 2.3 Pico and Pic% . . . 21
  - 2.3.1 Basic concepts . . . 21
  - 2.3.2 Basic syntax and semantics . . . 23
  - 2.3.3 OO programming in Pic% . . . 26
- 2.4 Interpretation of programs . . . 28
  - 2.4.1 REP machines . . . 28
  - 2.4.2 Meta-circular interpreters . . . 29
- 2.5 A note about multi-paradigm programming . . . 31

**3 Syntactic Extensibility** **32**
- 3.1 Overall architecture . . . 32
- 3.2 Lexical analysis . . . 33
  - 3.2.1 Streams . . . 33
  - 3.2.2 The xPic% scanner . . . 34
- 3.3 Syntactic analysis . . . 35

3.3.1 A simple parser library . . . 36
3.4 Parser combinators . . . 37
3.4.1 A simple combinator library . . . 37
3.4.2 Using combinators to express EBNF rules and regular expressions . . . 39
3.4.3 Primitive parsers and combinators as reificators of the grammar . . . 41
3.4.4 Syntactic extensibility through the disjunction combinator . . . 41
3.4.5 Drawbacks of parser combinators . . . 42
3.5 Building ASTs . . . 42
3.5.1 Primitive parse trees . . . 43
3.5.2 Basic AST constructors . . . 44
3.5.3 More elaborated AST constructors . . . 45
3.6 The xPic% concrete grammar . . . 46
3.6.1 Grammar configuration spots . . . 46
3.6.2 Becoming minimalistic . . . 47
3.6.3 Reification of the concrete grammar & grammar management . . . 48
3.7 User-end tools for syntactic extensibility . . . 51
3.7.1 Defining new syntactic constructs . . . 51
3.7.2 Installing & accessing syntactic constructs . . . 53
3.7.3 Syntactic maps . . . 54
3.8 Conclusion . . . 54

**4 Semantic Extensibility** **56**
4.1 Overall architecture . . . 56
4.1.1 The top-level evaluator . . . 57
4.2 xPic% intrinsic semantics . . . 58
4.2.1 Value types . . . 58
4.2.2 Value binding . . . 60
4.2.3 Default actions . . . 62
4.3 The xPic% meta-level interface . . . 64
4.3.1 Evaluator registration . . . 64
4.3.2 Working with evaluation environments . . . 65
4.3.3 Expression manipulators . . . 67
4.3.4 Accessing basic mechanisms of the interpreter . . . 70
4.4 Conclusion . . . 70

**5 Validation of the Language Kernel** **72**
5.1 Bootstrapping the language . . . 72
5.1.1 Growing the minimal grammar . . . 73
5.1.2 Installing syntax to create functions . . . 74
5.2 Going from xPic% to Pico . . . 75
5.2.1 Basic constructs . . . 75
5.2.2 Standard definition, declaration, and assignment . . . 76
5.2.3 Table definition and table entry assignment . . . 77
5.2.4 The boolean system . . . 78
5.2.5 Execution control . . . 79
5.3 Object-oriented extensions . . . 79
5.3.1 *Ex-nihilo* creation of objects . . . 80
5.3.2 Message sending . . . 80
5.3.3 Agora's views and mixins . . . 83
5.4 Conclusion . . . 85

**6 Conclusion** **86**
6.1 The reflective language kernel . . . . . 86
6.2 Other achievements . . . . . 87
6.3 Limitations and future work . . . . . 88
6.3.1 Reasoning about the grammar . . . . . 88
6.3.2 Language safety, constrained extensibility . . . . . 88
6.3.3 Other parsing techniques . . . . . 89
6.3.4 Localized extensions . . . . . 89
6.3.5 User-defined value binding semantics . . . . . 89
6.3.6 Pure functional language semantics . . . . . 90
6.3.7 Minor work . . . . . 90
6.4 Finally... . . . . 90

**A Improving performance** **91**
A.1 Token stream caches . . . . . 91
A.2 Parse caches . . . . . 92

# Chapter 1

# Introduction

The design and implementation of programming languages is one of the oldest fields in computer science. It is also one of the most fundamental. If many find computer programming an exciting activity, the meta-activity of describing how to describe, i.e. of designing and implementing programming languages, cannot be less but fascinating. Now, through computational reflection, part of this fascination, and most importantly part of the power of a language's designer can be put to the service of the language's user. Reflection has the potential to permit the long deferred promise of truly *open programming languages and systems* to be realized.

Since more than a decade now, the area that have got the most attention is object-oriented programming. Object-oriented programming has settled down firm bases in the realm of computer science. The computational model it provides has proven to be useful in a large set of domains. Insight into the very nature of object computation has been gained due in part to its marriage with computational reflection. This union holds out the promise of dramatically changing the way we think about, organize, implement, and use programming languages and systems. Such an insight has not been gained in the combination of object-oriented and functional programming languages with reflective meta-level architectures. The benefits of this hybrid approach are a question that is lacking a sound answer. In this dissertation, we explore an hybrid functional/prototype-based language that allows its configuration through reflection, not only to tweak its semantics but also its syntax.

One particular area of application is *domain-specific languages* (DSLs). In the literature, they are also called *little*, *lightweight* or *micro* languages [1]. DSLs, tailored towards the specific needs of a particular domain, can significantly ease building software systems for that domain. They are less cryptic and easier to learn for domain experts, since they are usually small, offering only a restricted suite of notations and abstractions: while such languages provide a natural vocabulary for concepts that are fundamental to the problem domain, with general-purpose languages one is reduced to idiom.

By means of a highly reflective architecture, this dissertation aims at constructing a "language laboratory" that allows the exploration of language designs in the symbiosis of two paradigms, the functional and object-oriented, with the possibility of defining an appropriate set of notations for the problem domain at hand.

## 1.1 Objective

The main goal of the thesis is to design a meta-level interface for a reflective language, founded on an hybrid functional/prototype-based paradigm. The meta-level interface should allow the syntactic and semantic extension of the language with new, unpredicted constructs.

As a starting point, the design should be based in the programming language Pic%. The new language should maintain the existing qualities of Pic%, namely:

1. A very simple semantics.
2. An intuitive syntactic front end.
3. The natural symbiosis of the object-oriented and functional paradigms, e.g.:

   object $\longleftrightarrow$ evaluation environment
   method $\longleftrightarrow$ function
4. A clearly structured metacircular implementation that serves as a specification of the language and a testbed for its extension.

Furthermore, the extension mechanisms of the new language should support the exploration of concepts in object-based computational models, apart from those already present in Pic%.

## 1.2 Methodology

To achieve the goal of this thesis, a reflective language kernel was developed. The basic design of the kernel is inspired on Pic%'s kernel (syntactically and semantically). It supports the requirements stated in section 1.1 in the following way:

- It has a (default) semantics which is even simpler than Pic%'s, hence requirement 1 is met.
- The syntax of the language is extensible, to cope with requirement 2. Syntax extensions provide syntactic sugar for problem-specific abstraction.
- The model for object-orientation defined using the language kernel has the same foundation as Pic%'s model, thus meeting requirement 3.
- To support the exploration of new mechanisms in object-based programming, part of the language semantics is extensible in a reflective way, i.e. it is possible to modify the evaluator of the language at run-time via reflective facilities.
- The kernel was implemented in Pic%, and such implementation has approximately the same level of complexity than the previously existing implementation (requirement 4). This document contains, throughout chapters 3, 4 and 5, the most important parts of the metacircular code, with detailed explanations.

## 1.3 Context

The following main forces surround and give impulse to the work of this thesis:

### 1.3.1 Advocation of prototype-based programming

Object-oriented programming languages (OOPLs) are gaining acceptance, partly because they offer a useful perspective for designing computer programs. However, they do not all offer exactly the same perspective; there are many different ideas about the nature of *object-oriented computation*. The concepts of object, class, type, message, inheritance, polymorphism, encapsulation and reflection can be combined in different ways to produce a variety of definitions of object-oriented programming. This has lead to the development of a number of object-oriented programming languages including Simula, Smalltalk, CLOS, C++, Eiffel and Java. Although the predominant OOPLs are based on the concept of class, there is an interesting category of object-oriented languages in which there are no classes at all. In this *object-based* model, all programming is done in terms of concrete, directly manipulable objects that are often referred to as *exemplars* or *prototypes*. These prototypical objects resemble the instances in class-based languages, except that prototypical objects are more flexible in several regards. For instance, unlike class-based languages in which the structure of an instance is dictated by its class, in prototype-based languages it is usually possible to add or remove methods and variables at the level of individual objects. Other differences include that in prototype-based languages object creation usually takes place by copying, and that inheritance is replaced by some other, less class-centered sharing mechanism. Prototype-based languages are conceptually elegant and posses many characteristics that make them appealing. Moreover, these languages are seemingly closer to some of the cognitive theories presented by psychologists and philosophers in the last century. In general, when working with prototypes, one typically chooses not to classify, but to exploit alikeness. Rather than dealing with abstract descriptions of concepts, the designer is faced with concrete realizations of those concepts. For these reasons and more to come in section 2.2, this thesis work advocates the usage of prototype-based programming.

### 1.3.2 Exploration of the design space of prototype-based languages

If the intent is to explore the alternative models of object-oriented computation offered by prototype-based approaches, there is a need for languages to specify them conveniently. As in the case of class-based programming, the design space of prototype-based programming is rich enough to leave room for a plethora of such languages: Self [2], Kevo [3], Agora [4], Garnet [5], Moostrap [6], Omega [7], Obliq [8], NewtonScript [9] and more. Mixing prototype- and class-based concepts has also been proposed [10, 11]. A very general and informal characterization of these languages is rather simple [12]: they propose a programming model in which there is one kind of object equipped with attributes and methods, a primitive way to create objects (*ex-nihilo* creation, cloning or differential description) and one primitive computation mechanism (message sending). Beyond this general characterization, these languages exhibit slight differences in the definition of their fundamental mechanisms that turn out to have a profound impact on their programming models. This is one of the reasons that motivate this thesis work: the exploration of the design space of prototype-based programming languages and the need for a playground to test and validate their fundamental mechanisms.

### 1.3.3 Research in the design of a language for educational purposes

This thesis work is inspired on Pic% [13], a prototype-based language designed at VUB by Professor Theo D'Hondt for educational purposes. One of the main forces driving the design and implementation of Pic% is its application as an educative tool [14]. Many elements in

Pic% make it a very well suited device for teaching:

1. It is prototype-based, thus it seizes on some fundamental benefits of the paradigm: a simpler model of object-oriented computational systems, and thus less concepts to be exposed in a first introduction to OOP. Apart from simple, the prototype-based approach makes OOP a more intuitive and concrete experience, an interesting example of which can be found in [15].
2. It has a simple semantics with an intuitive syntactical front end. Both are specified in a metacircular implementation of the language which is small enough for students to grasp in a small period of time (one proof of it is the case of the author himself).
3. It introduces students to a number of fundamental concepts in the structure and interpretation of computer programs [16]; Pic% constitutes a self-contained technological framework for uniting all the relevant notions and concepts. Skirting any really formal approach to syntax and semantics of programming languages, students are exposed to a rigorous treatment of the matter of building a consistent language processor [14]. The methodology is based on the examination of the Pic% metacircular implementation.

The language kernel proposed in this thesis inherits the "pedagogical side" of Pic%. Thus one of the design lines is to maintain conceptual simplicity and ease of use in the language. For example, the meta-level interface does *not* provide at the base-level (i.e. from the user's perspective) all the functionality available at the meta-level (i.e. from the language implementor's perspective).

### 1.3.4 Programming and meta-programming with multiple paradigms

Reflection, understood as the construction of self-awared systems, is a persistent source of challenge. The mere feeling of touching the essence of computing, but also the tremendous potential for new applications insure a continuous quest for understanding its foundations. Object-oriented meta-programming in class-based systems is dominated by the class/metaclass approach that provides a highly satisfactory solution to the problem of structural reflection [17]. In the case of prototype-based programming, although there is not a broadly accepted approach, the development of many reflective models (take for instance [4]) confirms that reflection is possible and even easier as a consequence of the simpler underlying computational model. Indeed, there are cases [17] in which prototype-based programming has been chosen as a testbed for reflection since it avoids the unnecessary complexity of classes before a reflective facility is fully understood.

What has not been studied thoroughly is reflection in multi-paradigm environments, particularly in the symbiosis between the functional and object-based computational models. Thus a final and very important driving force of this thesis is the exploration of reflection in an hybrid functional/prototype-based setting.

## 1.4 Scope of the work

The work presented here is not about:

- An exhaustive exploration of prototype-based languages and their semantics. The aim of the kernel proposed in this thesis is to *allow* the exploration of language concepts, i.e. it is a research tool. One of the hypotheses of this work is that the design space of

prototype-based languages is big enough to justify the creation of such tool. To validate the language kernel, some concepts are explored (section 5.3) but to carry out a broad exploration of prototype-based programming mechanisms is out of our scope.

- Full behavioral reflection. There is no reification of the computational state at the base-level, thus the full requisites of behavioral reflection are not met (note though that the newest versions of Pic% *have* first-class computational state, but this work didn't include this feature since it is based on an older version).

- Reasoning about grammars. There is not a focus on either assessing properties of the language's grammar (e.g. ambiguity), nor on automatic transformation of grammars (e.g. grammar simplification). The language kernel is "passive" in the sense that in only manipulates the grammar on user request, but it never engages into finding facts about the grammar or modifying it automatically.

## 1.5 Outline of this dissertation

In this chapter, the objectives of this thesis and the way to achieve them have been established. The reasons that motivate the work and the main design lines were exposed. Finally, the limits in scope were established. A description of the contributions of this work will be delayed to the conclusions (chapter 6).

The rest of the dissertation is organized as follows:

In the next chapter, the background concepts necessary to understand the core part of the thesis are presented.

The interpreter of the language developed in this work is composed of three sub-machines: the lexical, syntactical and semantical analyzers. The first two are presented in chapter 3. Therein, a description of the techniques employed to achieve a syntactically extensible language is given. The last sub-machine, the semantical analyzer or evaluator of the language, is the subject of chapter 4.

A validation of the reflective language kernel developed in chapters 3 and 4 is given in chapter 5. This chapter assesses the ability of the kernel to emulate some constructs of the languages Pico [13], Pic% [13] and Agora [4], therefore providing a better understanding of what the extensible framework is capable of.

Finally, the conclusion in chapter 6 provides a unified view of the elements composing the language kernel and a few non-expected achievements of the work.

### 1.5.1 Roadmap for the busy reader

This introduction hopefully gave the reader an understanding of what this work is about. Now, to quickly grasp the core elements should the following guide be of any help:

1. If terms like 'reify', 'meta-circular', 'base-level' or 'meta-level' are unfamiliar, see section 2.1.

2. An initial description of prototype-based programming was presented in section 1.3; for a more complete background, see section 2.2.

3. The explanations of this document are based on metacircular code written in Pic%, therefore it is important to have a notion of its syntax and its semantics. To get acquainted with the basics of Pic%, read sections 2.3 and 2.4.1.

4. To see how syntactic extensions are achieved, first take a look at the beginning of section 3.6 to know how the grammar of the language looks like and in which points it can be extended. The base-level tools to perform such extensions are then described in section 3.7; these tools for syntax extension conform the first part of the meta-level interface of the language.

5. To know what the core semantics of the reflective kernel is, see section 4.2. Some of the contributions of this thesis concern this part; the semantics are described in a "neutral" manner in that section and then the contributions are highlighted in the conclusions, sections 4.4 and 6.2.

6. The tools provided at the base-level to allow the implementation of language extensions are described in section 4.3. These conform the second part of the meta-level interface of the language.

7. Examples of how the language can be extended using the meta-level interface can be found throughout section 5.2. This section also constitutes an example of how the user can explore the design of language mechanisms and the issues that arise (e.g. in the case of closure-based boolean systems, see section 5.2.4).

8. The symbiosis between the functional and object-based models is shown in section 5.3; the kind of symbiosis proposed is an idea inherited from Pic%; some differences are exposed and a few further prototype-based mechanisms are explored.

9. A final unified view of the language kernel is provided in section 6.1.

10. The achievements and contributions of this thesis can be seen in sections 3.8, 4.4 and 6.2.

What the reader would miss by following this roadmap is basically the internals of the interpreter (which allow syntactic and semantic extensibility); the internals include some interesting techniques like parser combinators.

# Chapter 2

# Background Concepts

This chapter presents all the concepts that are used without an explanation in the remainder of this document. Since xPic% is reflective, a general notion of meta-programming and reflection is necessary (section 2.1). The paradigm in which xPic% is circumscribed is an hybrid between functional programming and prototype-based programming. The notions of functional programming are assumed and thus not provided here (a good starting point is [16]). Regarding prototype-based programming, its basics are the subject of section 2.2. xPic% is derived from the programming language Pic%. Furthermore, Pic% is used to specify and implement xPic%, thus a general idea of what Pic% is and how it looks like is perhaps the most important background knowledge to have; an overview is available in section 2.3. Section 2.4 provides a global picture of how the xPic% interpreter is realized.

## 2.1 Meta-programming and computational reflection

In this section a crash introduction to meta-programming and computational reflection is presented. It synthesizes information from various sources [18, 19, 20] into a hopefully well-integrated view of the subject.

### 2.1.1 Basic concepts

This section establishes the terminology for the concepts in which meta-programming and reflection are based. In fact, this definitions will hold for the entire scope of this document.

#### Computational systems and their specification

To a big extent, computer science can be regarded as the study of complex processes by means of executable models. These models are called *computational systems*, or *systems* for short. Computational systems exhibit a specific behavior over time, which advances in discrete steps. Describing a system involves specifying how it evolves during its successive time steps. This description activity is called *programming*, and its product – a formal, executable specification of a computational system – is called a *program*. Programs are expressed in a formalism, the *programming language*, that can be interpreted automatically using some machine. By means of interpretation, the behavior of the computational system is obtained.

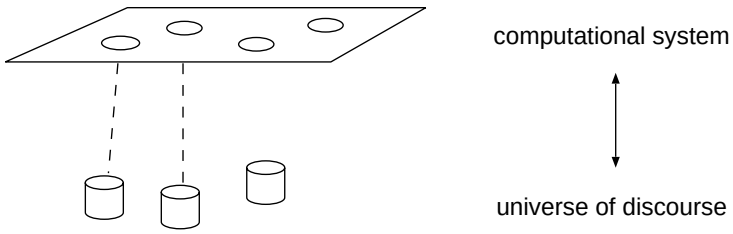



Figure 2.1: A computational system; two elements in the universe of discourse are reified, one is not. In the system, two structures are reifiers and the other two have a purpose other than representing domain elements.

#### Domain of a computational system

The *domain* or *universe of discourse* of a computational system is the collection of elements (entities or concepts) which the system can reason about. For example, an address book application is a system that reasons about a universe of discourse which contains (real) persons, names, phone numbers, days, addresses, etc. The computational system manipulates representations of these elements. However, not all the elements in the domain need to be represented explicitly in the system. Whenever an element has an explicit representation it is said to be *reified*, and such representation is called the *reifier*. Figure 2.1 illustrates the idea.

#### Relation between a computational system and its domain

A system is said to be *causally connected* to its domain if the internal structures and the domain elements they represent are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other. A system steering a robot-arm, for example, incorporates structures representing the position of the arm. These structures may be causally connected to the position of the robot's arm in such a way that, if the robot-arm is moved by an external force, the structures change accordingly and, if some of the structures are changed (by computation), the robot-arm moves to the corresponding position. So a causally connected system always has an accurate representation of its domain and it may actually cause changes in this domain as mere effect of computation.

### 2.1.2 Meta-programming

Programs can be constructed to reason about almost anything imaginable. In particular, programs that reason about other programs. Examples of such programs are compilers, interpreters and code generators. A program whose universe of discourse contains programs, is called a *meta-program* or *meta-level program*. The computational system it specifies is called a *meta-system*. The programs in the universe of discourse are called *base-programs* or *base-level programs*; they specify the *base-system* (figure 2.2).

### 2.1.3 Computational reflection

Reflection is a wide-ranging concept that has long been studied in philosophy and many different areas of science. It was introduced in computer science through artificial intelligence as it was considered as a property responsible, at least in part, for what is considered an

"intelligent behavior". In the area of programming languages it has been applied under the name of *computational reflection*. Computational reflection dates from Brian Smith's work in the early 80s [21].

A *reflective program* is a special kind of meta-program whose universe of discourse contains aspects of its own computational system (figure 2.3). A reflective program has access to data structures which reify its computational system or aspects thereof; the sum of these structures is called the *self-representation* of the system. This self-representation can be inspected or it can be acted upon. Because the self-representation is causally-connected to the aspects of the system it represents, the system always has an accurate representation of itself, and it can actually bring modifications to itself by virtue of its own computation. Computational systems specified by reflective programs are called *reflective systems*. Their main characteristic, as we have explained is to have a "causally connected self representation".

The difference between meta-programming and reflection is that, in the former, the base-system does not have access to its self-representation: such a representation is available from the meta-level only.

### Reflective computation

Reflective computation occurs when reflective code is executed, i.e. code performing computation about the system itself. Examples of reflective computation are to keep performance statistics, computation about which computation to pursue next (reasoning about control), self-modification and self-activation (through monitors or daemons).

A distinction is made on the nature of the reflective computation: observing the base level is distinguished from actually modifying it. Thus there exist two kinds of reflective operations [22]:

- *Introspection* is the ability of a program to observe and therefore reason about its own state.
- *Intercession* is the ability of a program to modify its own execution state or alter its own interpretation or meaning.

These operations require a reification mechanism that allows them to do such manipulations. As will be seen in chapters 3, 4 and 5, the language kernel proposed in this work allows reflective computation; both introspective and intercessive facilities will be defined.

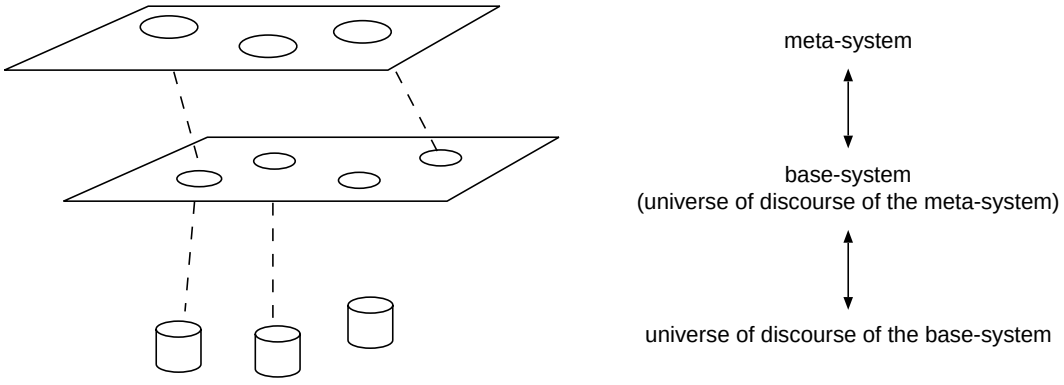



Figure 2.2: The idea of meta-programming: programs whose universes of discourse contains other programs. Causal connections among each level may exist.

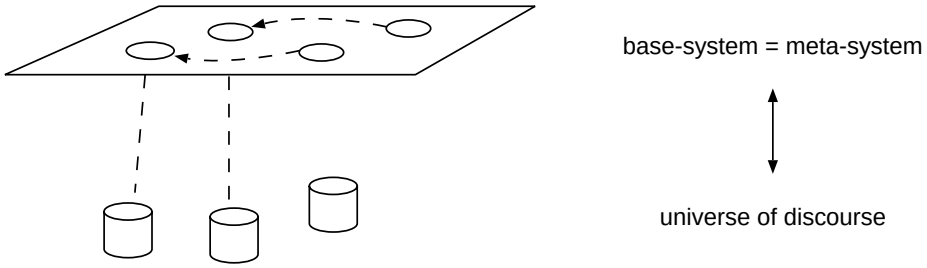



Figure 2.3: Reflection: parts of the computational system are about the system itself; some others are about the application's main domain

**Structural vs. behavioral reflection**

Two kinds of reflection are distinguished, *structural* and *behavioral* reflection [20]:

- Structural reflection is concerned with the ability of the language to provide a complete reification of both the program currently executed as well as a complete reification of its abstract data types.
- Behavioral reflection is concerned with the ability of the language to provide a complete reification of its own semantics and implementation (processor) as well as a complete reification of the run-time system. By definition, behavioral reflection allows a program to modify, even at run-time, its own code as well as the semantics and the implementation of its own programming language. This late-binding of the language semantics favors interpretive techniques. Interpreters ease modifications and react to them as soon as they occur.

Structural reflection requires the reification of entities used for building the system statically. Behavioral reflection goes further since it requires to reify entities used to perform the computation of the system (e.g. the execution stack); these entities belong to the dynamic part of the system.

The language kernel to be presented in the following chapters supports structural reflection but it does not allow full behavioral reflection. Although it is possible for a program to modify its own code at runtime, as well as part of the language semantics, there is no reification of the run-time system (the dynamic part of the evaluator).

### 2.1.4 Reflective architectures

A programming language is said to have a *reflective architecture* if it recognizes reflection as a fundamental programming concept and thus provides tools for handling reflective computation explicitly. Concretely, this means that:

- The interpreter of such a language has to give any system that is running access to data representing aspects of the system itself. Systems implemented in such a language then have the possibility to perform reflective computation by including code that prescribes how this data may be manipulated.
- The interpreter also has to guarantee that the causal connection between the data and the aspects of the system it represents is fulfilled. Consequently, the modifications

these systems make to their self-representation are reflected in their own status and computation.

Reflective architectures provide a fundamentally new paradigm for thinking about computational systems. In a reflective architecture, a computational system is viewed as incorporating an "object part" and a "reflective part". The task of the object computation is to solve problems and return information about an external domain, while the task of the reflective level is to solve problems and return information about the object computation.

The language developed in this work (xPic%) has a reflective architecture.

### 2.1.5 Conclusion

Reflection facilitates the design of specialized interpreters. Indeed, OO programming is a field in which experiments were extensively carried out to determine the best choices of design (see for example [23]). Some designs are appropriate for some applications but not for others. But reflective facilities make languages open-ended: in particular, reflection makes it possible to make local specialized interpreters of the language from within the language itself. This is one of the key points that motivate the creation of a reflective language kernel like xPic%.

## 2.2 Prototype-based programming

An Object-Oriented Computational System is nothing but objects sending messages to each other; every object has a state which is altered in convenient ways by means of message sending. Often a biological metaphor is used: messages are regarded as *stimulus* to which an object – like a cell – reacts, changing its state and provoking the emission of new messages to other objects that collaborate in the process; the exact way of transmitting the message is in principle irrelevant (it can be to 'neighboring' objects or to remote objects via 'conduits'), and the exact reaction to a message depends on the specific nature of the object. In this metaphor of objects as cells, the concepts of state encapsulation and object behavior are made clear.

The very basic concepts of object-oriented systems are thus *objects* and *messages* (with the associated concepts of state encapsulation and object behavior). *Object-Oriented Programming Languages* (OOPLs) are the formalisms in which the specification of an object-oriented computational system is expressed. Although this foundation is already operational, further concepts must be added in order to make it practical (i.e. to allow the construction of large object-oriented systems). Historically the first alternative introduced was the concept of *class*, based on the observation that in almost any system there are groups of similar objects, for instance in the Earth planet, there are animals, minerals and plants. Among animals there are mammals, fish, birds, and so on. A class describes the similarities among a group of objects, defining the state space and behavior of all its instances, whereas the instances hold the local data representing the particular state of objects. In broad terms, a class represents a concept, while an instance represents an individual, or a particular occurrence of a concept. This view of classes as concepts comes from the original Scandinavian school of object-oriented programming, started by Ole-Johan Dahl and Kristen Nygaard through their design of the programming languages Simula I and Simula 67 [24]. We will stick to this view; others regard classes as a sort of abstract data type, or a more implementation-oriented facility.

Here starts the divergence of two main branches in the OO paradigm, in one of which we are interested the most. We will call *class-based languages* those that have the concept of 'class', and *prototype-based languages* those that do not [25]. Most of the main-stream OOPLs in use nowadays are class-based. The question is, why should we worry about prototype-based languages?

### 2.2.1 Philosophical background

The material exposed in this section is mostly based on Taivalsaari's excellent introduction to the field of prototype-based programming [26]. As he explains,

> Some of the central concepts behind OOP –classes, instances and classification– have been of interest to human beings for centuries. The earliest characterization of classes versus instances was given by Plato over two thousand years ago. Plato made a clear distinction between *forms* – i.e. stable, immutable, "ideal" descriptions of things – and particular *instances* of these forms. He regarded the world of ideas as much more important than the world of instances, and contended that forms always have an existence that is more real than the concrete entities and beings in the real world.
>
> Research into classification (to be precise: biological classification) was continued by Plato's student Aristotle (384-322 b.C.) who had an endless interest in understanding and organizing the world to its smallest details. Whereas Plato was interested mainly in ideas and "eternal" concepts, Aristotle was the first philosopher interested especially in natural phenomena. In his works – over 170 in total – Aristotle aimed at providing a comprehensive, detailed taxonomy of all natural things – animals, plants, minerals, and so on. His classifications were based on the same idea that underlies object-oriented programming today. A group of objects belongs to the same category if the objects have the same properties. [...] New categories can be defined in terms of other categories if the new categories have at least the same properties as the defining ('genus') categories. The general rule for classification can be presented as follows:
>
> $$\text{essence} = \text{genus} + \text{differentia}$$
>
> In other words, categories are defined in terms of their *defining* properties and *distinguishing* properties. This corresponds precisely to the idea behind traditional class-based object-oriented programming, in which a class is defined in terms of its superclass (genus) and a set of additional variables and methods (differentia).

Class-based languages such as Smalltalk, C++ or Simula are platonic in their explicit use of classes to describe collections of similar objects. To dig up into the Aristotelian view of class-based programming, the reader is referred to [27].

The eastern world has been more holistic than the western. There seems to be a *need* in the western world to subdivide and classify things. Not only we classify but Aristote's work has lead to the idea that there is a single correct taxonomy of natural things, a universal classification. The Aristotelian 'classical' view stood unchallenged a long time. Categories were regarded as well-understood and unproblematic. It was the $19^{\text{th}}$ century when philosophers started to deny the existence of universal rules to determine what properties to use as a basis of classification of objects. They argued that classification is not a mechanical process but

it requires creative invention and evaluation. Consequently, there are no objectively 'right' classifications. People have many ways of making sense of things, depending on their cultural background, personal experience and their capacity to perceive, learn and remember, organize the things learned and communicate them efficiently. Thus taxonomies of all sorts abound.

Wittgenstein continued the criticism of classifications in the past century (1953), observing that it is difficult to say in advance what characteristics are essential for a concept. He gave several examples of seemingly simple concepts that are extremely difficult to define in terms of shared properties. A classical example is the concept of 'game'. Some games involve mere amusement, with no competition – winners or losers – though in other games there is competition. Some games involve luck, others involve skill, or a mixture of them to varying degrees. The number of players may also vary considerably from one, (solitaire), to hundreds, thousands, or even millions, as in lottery or race betting. There are even games in which no players are needed at all, such as the Game of Life (although some do not regard those as 'real' games). Another concept that is hard to define in terms of shared properties is 'work of art', since no one can really define clear boundaries for what is art and what is not.

In view of the mentioned flaws in the cognitive model of categorization, a new model was called for.

**A new cognitive model: the Theory of Prototypes**

How do humans better express concepts? By describing abstractions or remitting to significative examples? The expression of concepts by abstraction is the process of finding the defining properties of entities we observe in the real world. This is akin to the process of defining a set by comprehension in mathematics, with a predicate that describes the nature of its elements. It is argued that abstractions are usually discovered by generalizing from a number of concrete examples and that the abstraction process is likely to succeed much better if we have a lot of experience with the problem domain. In this direction, after criticizing the classical model of categorization, Wittgenstein defined what can be seen as the origin of prototype-based programming: the notion of 'family resemblance'. But it was Eleanor Rosch who introduced prototype theory in the mid-1970s. One of the central results of the prototype theory by Rosch and others is the observation that not all concepts and categories are equal. Rather, there are categories that are more 'basic' than others and objects that are 'better' representatives of categories than other objects. For instance, when one thinks of numbers, 1, 2, 3 pop to the mind, more than 5535, 1.666 or $\sqrt{2}$. This alternative point of view argues that knowledge may be hold in terms of the representatives which capture the main essence of the concept, the paradigms or *prototypes* of such concept [28]. In cognitive science, a prototype is a 'typical' member used to represent a family or a category of objects.

A (now) classical example was given by Lieberman in the mid-1980s [28], when prototype theory finally made its way through the world of OOP. If the elephant Clyde is our only experience with that kind of pachyderm, our concept of an elephant can really be no different than the concept of Clyde. We can consider Clyde to represent the concept of a *prototypical elephant*. After meeting other elephants, the analogies we make between them serve to pick out the important characteristics of elephants, allowing us to abstract better.

### 2.2.2 Limitations of class-based programming

The philosophical dilemmas exposed have in fact an impact in object-oriented programming. Until recently, most people have considered the notion of class an essential, inseparable part of OOP. However, the community have gradually become aware of the limitations and complexity of this approach. Taking into account the conceptual similarity of the Aristotelian model and the traditional OO paradigm, it is fairly obvious that the class-based OOPLs of today have pretty much the same shortcomings when it comes to modeling the real world.

**No optimum class hierarchies** There are no 'optimum' class hierarchies. As a consequence, designers of OO software should always be prepared for change. No matter how well designed a class library is, requirements may change in such a manner that substantial changes in the library are needed. It also happens that the point of view from which a hierarchy is designed doesn't fit well in other contexts, where the decomposition of the system (i.e. its classification) is done from a different point of view. There are no perfect taxonomies, thus problem domains usually admit more than one valid decompositions into classes; class-based programming allows to express only one of them in a given system.

**Unnatural implementation processes** An interesting observation is that when classes are organized into taxonomic hierarchies, the commonly used classes typically end up in the middle of the class hierarchy. In contrast, those classes that are at the top (root) or at the bottom (leaves) are typically of less interest, either because they are overly generic or overly specific for the purpose of examination. However, the implementation of an OO class hierarchy always proceeds (technically) from top to bottom, i.e. superclasses must exist before their subclasses. Therefore there is an inherent conflict between the classification process and the implementation of a hierarchy: the generic, more abstract classes can only be found when a substantial amount of expertise on the problem domain has been gathered, but those classes need to be implemented first. If the implementation is started before a sufficient level of expertise on the problem domain has been reached, substantial iteration in the development process is inevitable, since later knowledge is bound to reveal generalizations and new abstractions that will necessitate changes in the superclasses. But we cannot postpone implementation until the 'final' classification of the problem domain has been reached since we already know that a perfect classification is rarely possible. Thus the construction of a class library is inherently an iterative process: it will undergo a number of modifications till it becomes conceptually and technically satisfactory.

**Singletons** There are problem domains where unique objects exist. The GOF suggests the *singleton* design pattern to deal with this case [29]. It seems counterintuitive, however, to define a *whole* class of objects that contains in fact only one exemplar.

**Idiosyncratic Behavior** Aristotle realized himself that his model had problems and noted that many objects have 'accidental' properties, characteristic of the object under consideration but atypical for those kind of objects in general. Thus, the actual substance of concepts was defined in terms of two aspects: the *essence* and the *accidents* [27].

Since all objects of a class share the same behavior, it is unnatural to model this 'accidents' or *idiosyncratic behavior* of particular objects, which exhibit some particular properties but those do not set them apart from the family to which they belong.

**Too many roles** Classes usually have many roles in an OOPL:

- Abstraction of concepts.

- Instance descriptors and creators.
- Hierarchical categorization.
- Code sharing mechanism.
- Typing.

This role overloading makes classes conceptually less clear and difficult to manage. The proof of this is that everybody seems to agree in taking out the typing role from classes, like it happened in Java: the role was assigned to interfaces, i.e. pure typing constructs. This change made possible much cleaner designs in the Java API. The author doesn't know about studies that further simplify the role of classes in OOPLs.

**Meta-class regression** If classes are first-class (i.e. they are reified as objects), and since every object in class-based systems is an instance of a class, the well-known theoretical difficulty of infinite meta-regression occurs: a class, being an object, is an instance of a class, call it its meta-class. This meta-class is also an object, thus it must be an instance of a meta-meta-class, ... $\infty$

**Lack of flexibility** Class-based languages constrain objects too tightly. For example, they require every individual instance of a class to have the same behavior. They also forbid inheritance between objects, which could be used to share values of instance variables.

**Unnatural programming experience** The emphasis on classes in the programmer's interface is at odds with the goal of interacting with the computer in a concrete way (in systems like Smalltalk that promote exploratory programming). When designing a new object, one must first move to the abstract level of the class, write a class definition, then instantiate and test it, rather than remaining at one level, incrementally building an object.

The OO community has took sometimes ad hoc or unnatural solutions to these theoretical and technical difficulties, for instance the way that the meta-regression is prevented in Smalltalk (Metaclass class is both the class and an instance of Metaclass) or the singleton design pattern in class-based OO design. Sometimes, we also observe an impact in software engineering, with the need for CASE tools and complex methodologies.

### 2.2.3 Metaphors & mechanisms in prototype-based programming

The limitations presented in the previous section have lead a small group of researchers to the roots of the object-oriented paradigm (nothing but objects and messages) to look for alternative models. Some of the results they obtained are presented in this section: the set of metaphors, mechanisms and techniques they invented constitute the field of *prototype-based programming*. Hence, this section describes the concrete way in which the philosophical basement presented in section 2.2.1 is realized. The purpose of this section is not to provide a deep understanding of the issues raised by each mechanism but to give a general view of the subject. To organize the presentation, the different mechanisms have been split up in categories. This division is inspired in [12] and [23].

#### Object representation

Objects in prototype-based languages are defined by a set of properties, which are basically *bindings* of names to values. Some languages distinguish two types of properties, *attributes* and *methods*, whereas others amalgamate them into *slots*. Both approaches have advantages

and disadvantages. One argument in favor of slots is that they allow users to access attributes and methods the same way, and they permit overriding of an attribute with a method and conversely a method with an attribute.

### Object creation

Objects in prototype-based languages can be created in two ways: an object can be created *ex nihilo* (from scratch) or it can be created from an existing one (either by cloning or extension):

- *Ex nihilo* creation just produces new empty objects.
- Cloning consists in creating a new object and installing in it a copy of the bindings of the original object. This copy process has two variants: either *shallow copy*, where each value gets replicated, or *deep copy* in which not only the values are replicated, but pointer values are followed to replicate the pointed value. In contrast with deep copying, the bindings in a shallow copy point to the same values as the corresponding bindings of the original object. Most of the time, cloning performs a shallow copy of the object behavior, and a deep copy of the object state.
- In object extension, objects are obtained from an existing one by simply creating a new object and setting its *parent-link* to point to the original object.

### Object evolution

An important issue about object definition is whether this definition can evolve after the object has been created or not, i.e. if bindings can be added, modified or removed from an object. In particular, if objects are created *ex nihilo*, dynamic modification of structure is a must (otherwise they would remain uselessly empty). Another use of dynamic evolution is to allow idiosyncratic behavior: to configure specific objects with different method definitions, or to add new definitions.

### Sharing mechanisms

In reality almost *every* object has idiosyncratic behavior. As it is impractical to define by hand every object in the universe of discourse (in fact set definition by enumeration), every object-oriented system should provide a way of *sharing* behavior and state among objects. Organizing a large object-oriented system requires this ability [30]. In The Treaty of Orlando, Stein et al. [23] made explicit the alternatives regarding sharing:

**Static vs. dynamic sharing** depending on whether the system requires patterns of sharing to be fixed when an object is created or when it actually receives a message.

**Explicit vs. implicit sharing** depending on whether the language allows manipulation of the sharing patterns or not, for example selection of the specific method implementation to process a message.

**Group sharing vs. individual objects** depending on the way behavior is attached to objects: in mass or individually.

**Anticipated vs. unanticipated sharing** depending on whether the language permits anticipated structural description of the problem domain or is more suited to addition of behavior where it has not been foreseen.

There are many ways in which these sharing alternatives can be realized. In the following paragraphs, three sharing mechanisms will be presented.

**Inheritance** Inheritance is a well-known sharing mechanism. It should be noted that it is not exclusive of class-based programming; for instance, Pic% has normal inheritance. With respect to the criteria presented above (The Treaty of Orlando's terms), inheritance is a static mechanism to specify the implicit sharing of behavior for a group of objects.

Some languages allow an object to change its parent at run-time; this is sometimes called *dynamic inheritance*.

**Delegation** To explain delegation, suppose that an extension object is constructed from one or more prototypes. When it receives a message, an attempt is made to respond with its own 'personal' behavior; if it fails the message is forwarded to one of its prototypes, together with a reference to the object which originally received the message. This process of delegation goes on through the delegation chain until the message is handled. The key-point of delegation is that the `self` pseudo-variable inside a method refers to the original object that received the message, even if the method used to answer the message is found in one of its parents. So when messages are delegated, the original object has the opportunity to answer; that's why the metaphor between a delegator and a delegatee is used: "*I don't want to do all this job by myself, so I'll delegate you this particular part... if you have any questions just come back and ask me how to do it*". In The Treaty of Orlando's terms (that were presented above), delegation is a dynamic, implicit, individual sharing mechanism.

There are two alternatives in the semantics of delegation: a distinction is made between *implicit* and *explicit* delegation.

- In implicit delegation, when an object cannot answer a message, the interpreter automatically delegates it to another object; objects have a parent link to indicate to the interpreter to which object messages should be delegated. Some languages allow objects to have more than one parent. There is no conceptual problem with multiple delegation, although it raises similar problems to multiple inheritance.

- In explicit delegation, on the other hand, the delegation of messages is done explicitly for each message to be delegated; the delegating object names the object to which the message has to be delegated. Explicitly delegating a message resembles externally to normal message passing, except that the method invoked by explicit delegation is executed in the context of the delegating object (i.e. `self` still points to the delegator instead of the object to which the message is being delegated).

Both explicit and implicit delegation solve "the self problem" pointed out in [28]. An example of a possible usage of delegation is to implement the Delegation design pattern of the GOF book [29]. As we mentioned above, the metaphor of delegation says: "if you have any questions just *come back and ask me*". Without delegation, the "self problem" arises: a delegatee cannot refer any longer to the delegator. Hence, the Delegator pattern as stated in [29] (in the context of class-based languages) is not *true* delegation. In a language with delegation semantics, the problem is solved.

**Propagation** Languages like Kevo [3] do not support inheritance or delegation in the traditional way. Instead of these and other mechanisms that put a heavy emphasis on sharing, Kevo objects are logically stand-alone and typically have no shared properties with each other. New objects are created by cloning, and the essence of inheritance, incremental modification, is captured by providing a set of module operations that allow the objects to be manipulated flexibly. Indeed, when no delegation mechanism is provided, life-time sharing

between objects requires other "group-wide mechanisms" to automatically handle clone families in the system; the "clone family" of an object `obj` is defined as the transitive closure of the relation "is-a-clone-of(`obj`)". Kevo provides a mechanism called *propagation* allowing the application of operations like method addition to all objects of the same clone family. Regarding The Treaty of Orlando's terms, propagation is a dynamic, implicit, group and unanticipated sharing mechanism.

#### Split objects

Delegation raises an interesting issue, "the meaning of self" in a language [31]. More than message forwarding, delegation can also be interpreted as an extension mechanism. An object B that delegates to another object A, can be viewed as an extension of A. An extension object and its parent can be seen as different parts of the representation of the same domain entity. To split a representation in several objects in a delegation hierarchy is then a natural way of representing viewpoints. Taking an example from [12],

> Consider objects collectively representing a person – say "Joe" – in a delegation-based system. The object `JoePerson` holds the basic information about Joe (`address`, `age`, and so on), while the object `JoeSportsman`, an extension of `JoePerson`, holds information related to Joe as a sportsman. Creating `JoeSportsman` as an extension object instead of simply adding the slots `stamina` and `weight` to `JoePerson` also facilitates subsequent extensions, e.g. `JoeEmployee`. Any modifications to `JoePerson` are automatically seen by its extensions. Also, changes to Joe can be made through its extension objects: for example, if Joe the employee changes its personal address, the change will naturally be made at the person level and will be effective for all extensions of this person.

A split representation is a set of objects linked by delegation, representing a single entity of the domain such as the person Joe. There is no way to handle split representations as first-class objects, entities for which the term *split objects* has been coined, in today's prototype-based languages. Making split objects first-class would mean being able to create them, refer to them, clone them and otherwise deal with them as with other (atomic) objects. This is an open issue that is currently being investigated [12].

### 2.2.4 Class-based vs. prototype-based languages

The epistemological dichotomy between the two standpoints, classes vs. prototypes (rational vs. empirical) has direct implications in the way we express knowledge, and so does it in the design of OOPLs as our tools to represent reality and capture concepts in the digital domain. Here we make a comparison which puts in perspective the advantages and disadvantages of prototype-based programming using for this purpose a comparison with class-based programming.

**Flexibility** Prototype-based languages are intended to be more flexible than class-based languages; typically their origins are in the Lisp, Smalltalk and artificial intelligence communities where extreme flexibility is highly valued.

**Maturity** The main features of class-based languages begun with Simula in the mid-sixties [24]. In contrast prototype-based languages have emerged more gradually; their main principles were first gathered in the Treaty of Orlando. To a large extent, even their most basic notions are still evolving [25].

**Metaphors** Prototype-based languages are based on a biological metaphor: cloning and mutation of objects. Class-based languages are more mathematical-oriented, as they ask for classification which is set definition by comprehension.

**Concreteness** Prototype-based systems represent a view of the world in which one does not rely so much on advance categorization and classification, but rather tries to make the concepts in the problem domain as tangible and intuitive as possible, starting from experience. Some researchers [15] push to the limit this idea taking into account not only the language semantics but also the development process, the user interface and programming environment, to make the whole user experience enjoy of tangibility, simplicity and responsiveness.

**Natural apprehension process** A typical argument in favor of prototypes is that people seem to be a lot better at dealing with specific examples first, then generalizing from them, than they are at absorbing general abstract principles first (classifying) and later applying them in particular cases (instantiating).

**Simplicity** A goal of prototype-based programming is to offer a simpler and orthogonal programming model, with fewer concepts and primitives.

**Purity** State access, method invocation and other language elements tend to stick more closely to the OO paradigm than their class-based counterparts. Agora [4] illustrates this point. Another (somewhat extreme) example is Self [15], in which even method activation records are cloned objects of a prototype record.

**Language kernel complexity** No general comparison exists with respect to size of the language kernel. For instance Smalltalk has a very small kernel, yet it is class-based; C++ has a huge language kernel. Prototype-based languages tend to have small kernels.

**System construction process** In the class-based approach we have a common universe (the set of all objects modeled by the class `Object`) and our task is to define interesting subsets. The mechanism used is specialization or subclassing. Prototype-based systems go the other way around: the description of prototypes comes first and then we allow some aspects of the concept to vary. The prototype approach corresponds to generalization.

**Object creation** In class-based languages, object descriptions provide the templates from which object instances are generated; object construction may be a relatively complex procedure, involving message passing with possibly many collaborators. In prototype-based languages, instead, concrete and fully functional instances are built first; the process of instantiation (cloning) is semantically and computationally simpler.

Other advantage is that the client doesn't have to know the very specific nature of the object being created, it just has to send a 'clone' message: in a certain way a prototype follows the factory pattern, it can be used as a source of objects. The disadvantage of cloning with respect to class-based construction is that there is only one initial state for instantiated objects, i.e. the creation process is not parametrized. To solve this it would be possible to think of many prototypes for a family of objects, representing different initial states; those prototypes should be in close relation to the problem domain: if three typical objects are found in a certain application area then three prototypes could be maintained for object instantiation.

**Safety** Little attention has been devoted to designing statically typed prototype-based languages; as a consequence they are not as safe at runtime as some class-based languages. Class-based systems are less exposed to semantic inconsistencies (when they have a static type system).

**Efficiency** Static type systems allow efficient implementation of message dispatching. Many class-based languages have static typing thus they are more efficient than the majority of classless languages, which in general have a dynamic typing.

**Rapid prototyping** Classless environments are best suited to exploratory programming and rapid prototyping (proof of concept systems) than class-based environments, since making good designs from scratch is pretty time-consuming, and design errors, which are common in those application domains, are difficult to fix because of class coupling and lack of cohesion. Class-based environments are better in areas where the problem domain is well defined, making software models (class hierarchies) possible.

**Maintainability** One of the big problems with classless environments is that many structural aspects of the system are implicit due to the highly dynamic interaction between objects: there is a hidden network of collaboration. Class-based systems make explicit the relationship among instances through classes, so there is a considerable amount of information about system architecture available in the code.

### 2.2.5 Conclusion

The reader should note that there is much more to prototype-based programming than the material presented here; a good treatment is available in [32]. Section 2.2.2 shown why it is interesting to look at alternatives to class-based languages in object-oriented programming. Section 2.2.3 shown that the design space in prototype-based programming languages is big enough to make an extensible prototype-based language a useful tool. The different mechanisms presented in that section have deep implications in the programming models of prototype-based languages, a discussion which is out of scope in this dissertation. Section 2.2.4 aimed to putting in perspective the field of prototype-based programming by means of a comparison with class-based programming. But do prototype-based languages help overcome the limitations of the Aristotelian tradition that constrains the modeling capabilities of current class-based languages? (recall section 2.2.2) Unfortunately, as Taivalsaari replies [26], this is not really the case:

> Most prototype-based languages of today are motivated by relatively technical matters. For instance, prototypes are commonly used for reaching better reusability through increased sharing of properties and more dynamic bindings of objects, or for providing better support for exploratory programming. In contrast, they do not usually take into account the conceptual modeling side, let alone pay attention to the philosophical basis that underlies object-oriented programming. In a way, thus far the developers of prototype-based languages seem to have been even more ignorant to these underlying conceptual and philosophical issues than the Scandinavian inventors of the class-based paradigm.

Since prototype-based programming is not new, there is no imminent reason to see a switch of paradigms in the following years. As the Treaty of Orlando states [23], and as happens in almost every computer science discussion there is no "winner"; different contexts call for different OO sub-paradigms, and there will be always strong arguments from both sides of the OO field.

## 2.3 Pico and Pic%

Pic% is an object-oriented extension of Pico[1]. Pico is a functional language designed by Professor Theo D'Hondt at the Programming Technology Lab of Vrije Universiteit Brussel. From the Pico web site [13]:

> The fundamental aim of Pico is to introduce the essentials of computer programming to undergraduate students in the basic sciences other than computer science. In conceiving Pico we were strongly inspired by the approach used in Abelson and Sussman's book "Structure and Interpretation of Computer Programs" [16] and equally strongly repulsed by the various standard efforts to teach computer programming at the high-school and academic level. Pico can actually be viewed as an effort to render Scheme palatable and even enjoyable to people unable or unwilling to make the intellectual effort necessary to grasp its elegance and power. We do so by adapting Scheme's syntax (significantly) and semantics (subtly) in order to use what (little) understanding the novice science student has of specification languages.
>
> The word Pico should be interpreted as synonymous with very small (according to Webster's). The idea was indeed to have a very small language with a very general impact.
>
> We were also strongly driven by the ambition to return to the original attraction exerted by computer programming on young people; we strongly deplore the current situation where most of these regard programming as a mind-numbing chore. Pico is essentially the result of sugar-coating the hard essentials of computer programming in such a way that students cannot help but enjoy it.

Pic% is a simple yet conceptually-rich language. In the experience of learning it, the student gets exposed to many concepts of language theory and implementation. It is quite *sui generis* in the sense that it has all this teaching philosophy behind, and an hybrid functional/prototype-based paradigm.

The whole work of this thesis is explained using Pic% concepts and syntactic constructs, hence it is necessary to become familiar with it before proceeding to chapters 3, 4 and 5. A presentation of Pic% will be given in the following subsections, but the reader should notice that it is not a complete manual: only the parts of Pic% that are necessary to make this document self-contained are presented. Furthermore, some definitions have been slightly changed to suit better our context.

### 2.3.1 Basic concepts

Almost all the concepts in Pic% are "basic" in the sense that they are pretty fundamental; Pic% hardly has non-essential features. Here we present the minimal set of concepts needed to understand how Pic% works.

#### Primitive value types

The following seven value types are available:

[1] The percent symbol in the name stands for object/oriented → o/o → %

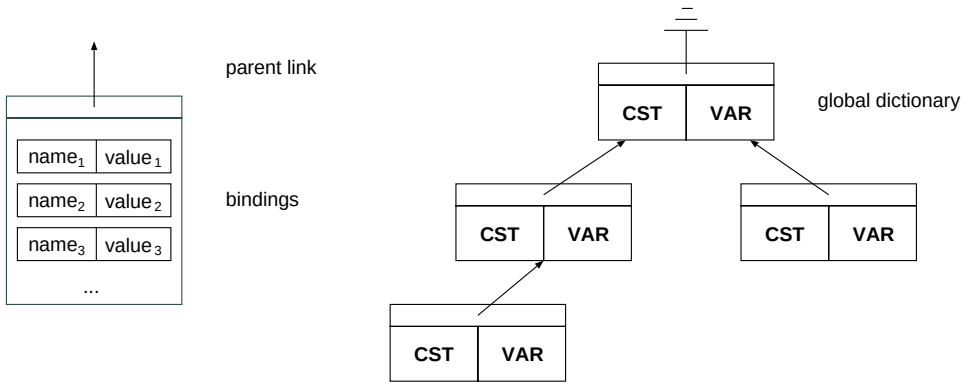



Figure 2.4: conceptual schema of a single dictionary (left) and a dictionary hierarchy (right). In the right, the distinction between the variable and constant parts of a dictionary has been illustrated.

| | |
|---|---|
| void | `void` – no value, nothing, |
| numbers | `89`, `0`, `-44` – i.e. integers, |
| fractions | `4.56`, `-8.76`, `10e-12` – i.e. floating-point numbers, |
| text | `"hola"` – i.e. strings of characters, |
| tables | `[ 1, "hola", 5.5, void ]` – analog to Scheme lists, |
| functions | `f(x):  x + 1` (first-class citizens like in Scheme), and |
| dictionaries | explained below. |

The functions `is_void()`, `is_number()`, `is_fraction()`, `is_text()`, `is_table()`, `is_function()`, and `is_dictionary()` can be used to query the type of a value.

Note that booleans are not primitive value types; the boolean system of Pic% (Church Booleans) is cleverly defined in terms of other basic language constructs. Section 5.2.4 gives the idea of such definition. Anyhow, it is not a required concept in this document.

### Dictionaries

The notion of *dictionary* is essential in Pic%. The importance of dictionaries is due to their use as evaluation environments for expressions.

A dictionary is in charge of maintaining *bindings* between names and values associated to those names. There can be *constant* bindings and *variable* bindings; constant bindings cannot be modified once entered into the dictionary. A dictionary can have a *parent* dictionary, thus hierarchical arrangements of dictionaries are possible. The topmost dictionary is called the *global* dictionary. Figure 2.4 illustrates the idea. The following dictionary operations are available:

`lookup_all(name)` tries to find a binding (either constant or variable) corresponding to `name`; failing to do so will result in a similar lookup in the parent dictionary, if any. The search proceeds upwards the chain until the binding is found or `void` is returned by the global dictionary.

`lookup_const(name)` the same as `lookup_all()` but the search process is performed considering constant bindings only.

`lookup(name)` is the default lookup operation; it can be `lookup_const(name)` or `lookup_all(name)`, depending on whether the dictionary is *protected* or not. This functionality will be used in section 4.3.2.

`define(name, value)` enters a new variable binding `[ name, value ]` into the dictionary. It will overwrite a previously variable binding with the same name, but it is not possible to override constant bindings.

`declare(name, value)` enters a new constant binding `[ name, value ]` into the dictionary. It is not possible to override a previously existing binding, either constant or variable, with a constant binding.

`assign(name, value)` Changes the value of the binding corresponding to `name`, with the given new `value`. An attempt to modify a constant binding results in an error.

Many implementation strategies may be used for dictionaries, e.g. the metacircular implementation of Pic% employs linked lists of bindings, and xPic%'s implementation uses objects that encapsulate variable-size tables.

### 2.3.2 Basic syntax and semantics

We will review in this section the main syntactical constructs of Pic% and their associated semantics.

#### References

A *reference* is just an alphanumeric name used to identify a binding `[ name, value ]` in a certain dictionary. The following operations can be applied with references:

`name : exp` Defines a *variable* binding `[ name, eval(exp) ]` in the current dictionary.
≡ `define(name, eval(exp))`.

`name :: exp` Declares a *constant* binding `[ name, eval(exp) ]` in the current dictionary. This implies that the value bound to `name` cannot be changed afterwards.
≡ `declare(name, eval(exp))`.

`name := exp` Modifies the value of the binding associated with `name`.
≡ `assign(name, eval(exp))`.

`name` Accesses the value associated with `name` in the current dictionary,
≡ `lookup(name)`.

Some examples are:

```
name :  "Pic%"
pi   :: 3.1415
i    := i+1
```

The : operator is called the *definition* operator, :: is the *declaration* operator, and := is the *assignment* operator.

### Tables and tabulations

Tables are the analog of 'arrays' in other languages. The operation of accessing a table entry by its positional index in the table is called a tabulation.

`[exp`$_1$`, ... , exp`$_n$`]` Denotes a table of size $n$, whose entries are the result of evaluating the given expressions.

`name[exp]` This is a tabulation, it retrieves the entry with index `eval(exp)` from the table bound to the reference `name`.

`name[exp`$_1$`]: exp`$_2$ Defines a new table of size `eval(exp`$_1$`)` in the current dictionary. Each entry in the table will be initialized (in order) by evaluating `exp`$_2$; if `exp`$_2$ has side effects, it is possible to have entries initialized with different values, e.g.

```
idx: 0;
tbl[5]: idx:= idx+1;
```

will create the table `[ 1, 2, 3, 4, 5 ]`.

`name[exp`$_1$`]:: exp`$_2$ Analogous, but the table bound to `name` cannot be changed afterwards; note however that it is possible to modify the table *entries* – what is constant is the binding of the table and its name, not the table content.

`name[exp`$_1$`]:= exp`$_2$ Sets the value of the entry at position `eval(exp`$_1$`)` to `eval(exp`$_2$`)` in the table bound to the reference `name`.

One important function related to tables that will be used is `size(t)`: it returns the number of entries of the table `t`.

### Functions and function application

A *function definition* consists of a name, a list of formal parameters and a body (the code of the function):

`name(x`$_1$`, ... , x`$_n$`): exp`$_B$ Defines a new function whose body `exp`$_B$ is parametrized by `x`$_1$`, ... ,x`$_n$ and enters it in the current dictionary under the given `name`. The arity of the function is fixed and equal to $n$. Example,

```
prepend(elem, list): [elem, list]
```

defines the `prepend()` function which returns a new table `[elem, list]` when applied — in analogy with Scheme's `cons` function.

`name@x: exp`$_B$ Defines a new function whose body `exp`$_B$ is parametrized by the sole parameter `x` and binds it to `name`; when the function is invoked, the arguments of the invocation are passed as a table bound to the parameter `x`, so the body can access each one by means of tabulation. The arity of the function is variable (it is `size(x)` for each invocation). Example,

```
head@list: list[1]
tail@list: list[2]
```

define the function `head` which returns the first element of a list (like Scheme's `car`) and `tail` which returns the 'rest' of a list, discarding the head (like Scheme's `cdr`).

`name(`$x_1$`, ... ,`$x_n$`):: `$exp_B$ , `name@x:: `$exp_B$ The same as the colon operator (`:`) except that the binding produced between the function and the name is constant.

`name(`$x_1$`, ... , `$x_n$`):= `$exp_B$ , `name@x:= `$exp_B$ The same as the colon operator (`:`) except that an existing binding (corresponding to `name`) is modified instead of entering a new one into the current dictionary.

A function *application* has two syntactic forms:

`name(`$exp_1$`, ... , `$exp_n$`)` Application with fixed arity = $n$. This is the most common notation in programming languages. Examples have already been seen in this document:

```
define(name, eval(exp))
size(t)
```

`name@exp` Application with variable arity = size(`exp`); the result of evaluating `exp` must be a table. Two examples of this kind of function application are:

`tail@prepend(1, prepend(2, prepend(3, [])))` evaluates to `[ 2, [ 3, [] ] ]`, `head@[ 1, [ 2, [ 3, [] ] ] ]` evaluates to `1`.

When a function is applied, a new evaluation environment is created, whose parent link is the the current environment. In this new environment, the actual arguments of the invocation are *bound* to the formal parameters of the function, a process that will be described in detail in section 4.2.2. Then the body of the function is evaluated using the extended dictionary, which is holding the bound parameter values. Function application is detailedly explained in section 4.2.3.

The 'name' of a function needs not be alphanumeric, it can also be a sequence of one or more operator characters: `< = > # ~ $ % + - | & * / \ ! ? ^`. For example `-#-(x, y, z)` is a valid function application, the name of the function is `-#-`. Functions named in this way are called *operators*. In the case of unary or binary operators, Pic% allows to have prefix and infix notations respectively,

• `exp` is in fact the function application with arity 1: `•(exp)`.

`exp`$_1$ `•` `exp`$_2$ is the function application with arity 2: `•(exp`$_1$`, exp`$_2$`)`.

In the code of chapters 3 and 4, an operator will sometimes be used, `>--`. This operator allows dynamic inheritance. It is invoked using prefix syntax:

```
>-- parent
```

which means "set `parent` as the parent of the current dictionary". Hence in Pic% it is possible to change at run-time the hierarchical structure of dictionaries that was illustrated in figure 2.4. The implementation of the `>--` operator will be shown in section 4.4, although it is not necessary to know its internals; only the fact that it is used for inheritance should be kept in mind.

### Begin blocks

To implement an algorithm, it is often necessary to evaluate a sequence of expressions. With the tools we have now this is already possible to achieve, by evaluating a table with the desired expressions: `[`$exp_1$`, ... ,`$exp_n$`]`; Pic% semantics ensures that all the expressions will be evaluated, from left to right. But this has two drawbacks: it is a 'hack' (i.e. not conceptually clean) and the result of evaluating the group of expressions is a table, whereas we would like, when defining function bodies, to return other kind of values as well. The `begin()` function serves this purpose,

`begin@t:: t[size(t)]`

It comes predefined in Pic%. Thus we can use

`begin(`$exp_1$`, ... , `$exp_n$`)`

The expressions are evaluated in order and the table of result values is passed to `begin()`, which will take the last entry (i.e. `t[size(t)]`) as the resulting value of the function.

To make programs more readable, syntactic sugar for begin blocks is available:

`{ `$exp_1$`; ... ; `$exp_n$` }` ≡ `begin( `$exp_1$`, ... , `$exp_n$` )`

Begin blocks are the last basic construct we needed to explain. Next, the object-oriented part of Pic% will be presented.

### 2.3.3 OO programming in Pic%

Object-orientedness in Pic% is minimalistic: Pic% doesn't provide a rich set of constructs to work with objects, yet it is pretty much what can be done with them. We will see now how the principal concepts of prototype-based programming are mapped in Pic%.

#### Objects, methods and message sending

As we said in the introduction of this chapter,

$$object = encapsulated\ state + behavior$$

In Pic%, an object is represented by a dictionary. The object's state can be seen as the values currently present in the dictionary, and the particular values which are functions as the behavior. Thus object methods in Pic% are in fact functions being held in the object. Sending a message to an object is done through function application over a receiver:

`receiver.name(`$exp_1$`, ... , `$exp_n$`)` – or – `receiver.name@exp`

`name` is looked up in the *constant part* of `receiver`, and the resulting value, which should be a function, is applied to the given arguments using `receiver` as evaluation environment, instead of the current dictionary. In this way encapsulation comes into play: a method is never looked for in the variable part of the object. So the designer of the object should declare (with the double-colon operator `::`) the public methods and define (with the colon operator `:`) those that are private to the object. All non-functional values should be kept private to achieve state encapsulation (a public variable cannot be modified since it is a constant binding but it would allow anyway to peek into the object's state).

Within a method, the way to send messages to the parent object (i.e. to the parent dictionary) is to omit the receiver,

`.name(`$exp_1$`, ... , `$exp_n$`)` – or – `.name@exp`.

A useful function when working with objects is `this()`: it returns the current dictionary. There is no pseudo-variable pointing to the current receiver like `this` in Java or self in Smalltalk.

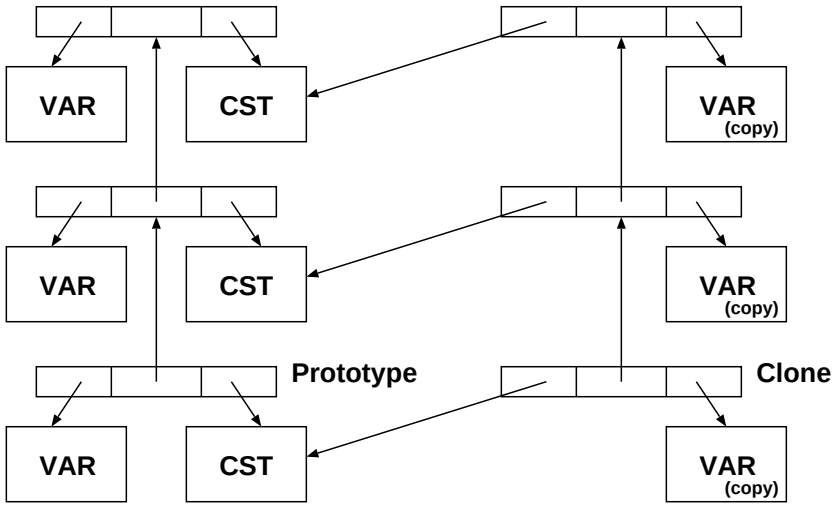



Figure 2.5: object cloning in Pic%

### Object creation

The native function `clone()` provides a way to create objects. When invoked, it deep-copies the variable part of the current dictionary, and shallow copies the constant part. Recall figure 2.4; now in figure 2.5 (with dictionaries illustrated in a sightly different manner) the process of cloning is shown. To clone an object, a new object is created. The constant part is set to point to the constant part of the prototype. The variable part is copied completely and the parent of the clone is set to a clone of the parent of the prototype. The following pseudo-code illustrates this mechanism:

```
clone(prototype): {
  constant: get_constant_part(prototype);
  variable: get_variable_part(prototype);
  parent: get_parent(prototype);
  new_dictionary(constant, copy(variable), clone(parent)) }
```

Cloning is the only tool needed, apart from objects and messages, to start programming in an object-oriented way in Pic%. For example:

```
make_dictionary(): {
  keys[10]: void;
  values[10]: void;
  get(key):: ...
  put(key, value):: ...
  clone() };
dictionary_proto: make_dictionary()
```

`dictionary_proto` will be a dictionary with two constant bindings '`put`' and '`get`', and two variable bindings '`keys`' and '`values`'. Only the two methods will be visible. The structure of objects is fixed once they are created: no bindings (of methods or variables) can be added or removed afterwards.

There is a second way of creating objects, using the function `clone()` seen before: the object to be cloned can be passed as an argument. The same cloning process takes place, but using the given object instead of the current dictionary. In fact `clone()` $\equiv$ `clone(this())`. Once

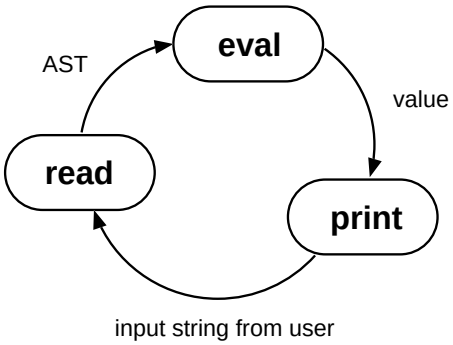



Figure 2.6: A read-eval-print (REP) machine.

we have a prototypical instance, we can copy it using this second form of `clone()`. Continuing the example,

```
moons: clone(dictionary_proto);
moons.put("Phobos", 22);
moons.put("Deimos", 8);
moons.get("Ganymedes");
 ...
```

## 2.4 Interpretation of programs

Like Pic%, xPic% is an interpreted language. In this section some concepts about interpretation are given. Surely enough, a whole treatment of the subject requires a book [16]. Hence we review only a few ideas that give a global picture of how xPic% works. In particular section 2.4.1 helps to understand better the role of the syntactical and semantical analysis machines in the interpretation process; such machines will be the subject of chapters 3 and 4. The following discussion applies equally to Pic% and xPic%, thus for brevity only the latter will be mentioned. The reader should keep in mind, though, that these ideas were originally applied in Pic%; the interpreter of xPic% inherits its global architecture from Pic%'s interpreter.

### 2.4.1 REP machines

The xPic% interpreter is a traditional *read-eval-print* (REP) machine. This means that the interpretation process is divided in three phases:

**read** When a program is entered by the user, this program is just a string, a sequence of characters. Before the core of the evaluator can do something reasonable with it, it has to be checked whether that sequence of characters indeed constitutes a valid program. If it does, the read phase transforms the text into a tree data structure, the so-called abstract syntax tree (AST). This tree represents the structure of the program and no longer contains irrelevant information such as whitespace and comments. Furthermore, such representation is much more appropriate for implementation of algorithms that depend on the structure of the program (like evaluation, the next phase).

The main tools for the realization of this phase are the *lexical* and *syntactical analyzers* of xPic%. They are described in chapter 3.

**eval** Analyzes an AST and executes the required actions according to the language semantics; the result of this phase is a value. In almost every case this analysis boils down to performing a recursive process on the tree. Hence, the evaluator is a tree recursive process that consumes an AST and returns a value.

The main tool used here is the *semantical analyzer* or *evaluator* of xPic%, described in chapter 4.

**print** Presents in the user interface, a legible representation of the value produced by the evaluation phase.

Although Abelson and Sussman [16] take as synonyms *evaluator* and *interpreter*, we will refer with the former to the semantical analyzer exclusively, and the latter will denote the whole interpretation system (lexical + syntactical + semantical analysis machines + printer).

The read and print phases are stateless: their execution depends only on the input. The eval phase is stateful, this is, the execution of an expression depends on the history of previously executed expressions. The only reason for which the evaluator is stateful, is that it holds a dictionary, the *current dictionary*. Each time an expression is evaluated, the current dictionary is used to resolve variable references and modify bindings as needed. This implies that expressions are able to have side effects, therefore xPic% is not purely functional (just like Scheme).

This is all what we need to know about the general structure of interpreters. Now we proceed to describe a commonly used technique to describe them.

### 2.4.2 Meta-circular interpreters

An interpreter that is written in the same language that it interprets is said to be meta-circular. In the chapters to come, the semantics of the language xPic% will be formally defined using programs capable of executing it; these programs will be written in Pic%. As xPic% is a subset of Pic%, we still consider the interpreter of xPic% (which is written in Pic%) to be meta-circular. Studying a metacircular evaluator is a useful activity because:

- By studying this amount of Pic% code, the reader gets to know Pic% much better than by studying small code examples or the general description presented in section 2.3.
- By studying the code, the reader gets a precise definition of the semantics of xPic%. In fact, the code for the evaluator is the only precise definition of the semantics.
- Although the evaluator studied is written specifically for xPic%, it contains the essential structure of an evaluator for any computer language. Thus by studying the metacircular evaluator, the user gets acquainted to the discipline of interpreter building in a very accessible way.

#### Meta-circular interpreters and reflection

We will use Pic% as an example to illustrate some aspects of meta-circular interpreters. Suppose the only definition we have of a Pic% interpreter is written in Pic%, i.e. suppose we only have a meta-circular implementation of the interpreter. If Pic% is evaluated using an interpreter written in Pic%, how runs in its turn the code of the interpreter? Another instance of the meta-circular interpreter is the only possible answer. This other instance is

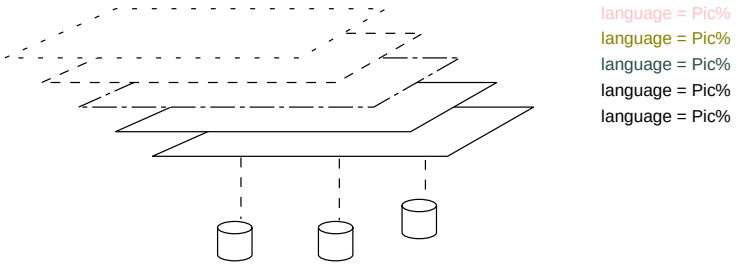



Figure 2.7: An infinite tower of Pic% interpreters.

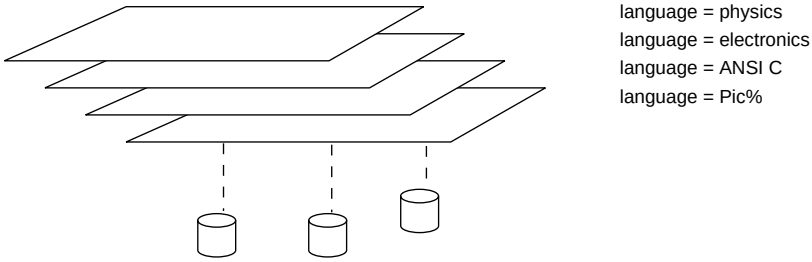



Figure 2.8: The tower of interpreters used to run Pic% programs.

also written in Pic%, so an interpreter of Pic% must run it, and the only one we could use is meta-circular... $\infty$. Virtually, the interpretation of Pic% consists of an infinite tower of meta-circular interpreters, each one interpreting the meta-circular interpreter below (figure 2.7). Technically, this infinity is realized by the presence of a second interpreter written in another language, which is able to interpret the circular interpreter (and which should be guaranteed to generate the same behavior as the circular one). Figure 2.8 illustrates the idea in the case of Pic%; the infinite tower is cut by the interpreter written in C. The C interpreter (which has been translated to machine language) relies on the circuitry of the computer (which uses the language of digital logic); the electronic interpreter is written in the language of physics (electricity, conductors, etc.). In this way, we have a finite chain of interpreters, making the interpretation process possible.

Meta-circular interpreters constitute a simple technique to achieve reflection (recall section 2.1.3). For instance, they present an easy way to fulfill the causal connection requirement. The self-representation that is given to a system is exactly the meta-circular interpretation-process that is running the system. The consistency between the self-representation and the system itself is automatically guaranteed because the self-representation is actually used to implement the system. So there is not really a causal connection problem. There only exists one representation which is both used to implement the system and to reason about the system. Note that a necessary condition for a meta-circular interpreter is that the language provides one common format for programs in the language and data, or more precisely, that programs can be viewed as data-structures of the language.

One problem with meta-circular interpreters is that a self-representation has to serve two purposes. Since it serves as the data for reflective computation, it has to be designed in such a way that it provides a good basis to reason about the system. But at the same time it is used to implement the system, which means that it has to be effective and efficient. These are often contradicting requirements.

## 2.5 A note about multi-paradigm programming

It is widely accepted that different types of tasks can be best implemented in different paradigms. As an example the logic programming paradigm is particularly well suited to experts systems, the functional programming paradigm allows to elegantly express algorithms, and the object-oriented paradigm is good for modeling well-structured systems.

A *multi-paradigm language* is a programming language designed to support different paradigms with equal ease: logic, functional, imperative, constraint, object-oriented, sequential, concurrent, etc. Some of the problems in achieving multi-paradigm programming in a language are:

- accommodation of different syntactic notations,
- accommodation of diverse execution models, and
- support for different implementation strategies.

Pic% is both a functional language (since it is a superset of Pico) and a prototype-based language. Multi-paradigm programming in Pic% can allow each part of a system to be implemented in the most suitable of the two paradigms. The issues mentioned above were dealt with in the particular case of Pic%. The symbiosis between the functional and prototype-based paradigms happens to be quite clean: there is not much interference from one into the other from the syntactical, semantical and implementation points of view. Thus the mix of paradigms is not forced. It should be taken into account that Pic% is not purely functional, which makes the symbiosis easier regarding semantics. The imperative nature of object-orientedness could enter in conflict with a purely functional language semantics.

# Chapter 3

# Syntactic Extensibility

One of the two core elements of the extensibility framework of xPic% is syntactic extensibility. In this chapter we will explain all that is necessary to understand it; this knowledge will be used later, in chapters 4 and 5.

The end product of this chapter is a tool set made available at the base-level to the xPic% programmer. This tool set consists of meta-functions that enable the modification of the language syntax at run-time; they belong to the meta-level interface (MLI) of the language. Section 3.7 (the last) presents this end product. The sections before give the necessary preliminar concepts (for instance the definition of the xPic% grammar) and show how the meta-level manages to support run-time syntax extensibility.

## 3.1 Overall architecture

The syntax extension architecture is depicted in the following figure:

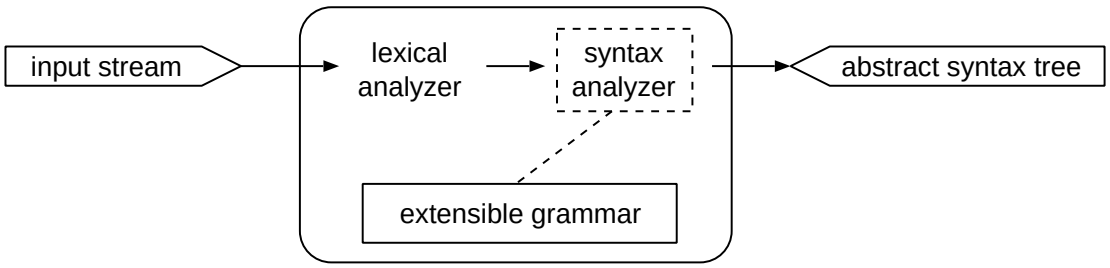



Figure 3.1: The xPic% parser

We can think of this architecture as an abstract machine that takes as input a character stream representing a program in xPic% and produces as output an abstract syntax tree (AST) – the same program but represented in the language understood by the abstract evaluator machine, which we will describe in chapter 4. Character input is transparently read from the user interface or from a file.

The architecture depicted in figure 3.1 is pretty standard in interpreters and compilers. What makes it different is the extensible grammar part: that the grammar encoded in the parser

is not fixed but can vary at run-time. We will briefly describe the xPic% lexical analyzer in section 3.2 and then devote the rest of the chapter to the syntax analyzer and its extensibility mechanisms.

## 3.2 Lexical analysis

The principal job of a *lexical analyzer* or *scanner* is to place a first level of abstraction over a raw stream of characters given as input, in order to make it semantically richer for client code (in our case, the parser). To achieve this, there are two abstraction sub-layers. The first consists in representing the input and output as *streams*, thus enabling the scanner to be independent from the concrete source of input characters and the client of the produced output. This simple layer will be described in section 3.2.1. The second and most important sub-layer consists in representing, as a unit, a logically cohesive sequence of atomic symbols; this unit is called *token* [33]. For example in the string "year:= 1492", the individual characters 1, 4, 9, 2 represent a same entity, the number 1492. The job of the scanner is to find this groupings and present them as tokens to client code.

**Example 3.2.1** Consider the character input stream as seen by the scanner:

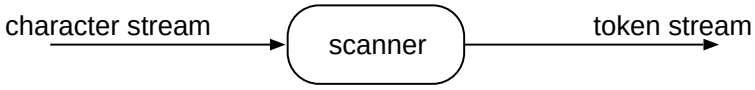
... → d → i → s → p → l → a → y → ( → ' → x → P → i → c → % → ' → ) → ...

It would yield the output token stream:

... → [ NAME, "display" ] → [ CHAR, "(" ] → [ STR, "xPic%" ] → [ CHAR, ")" ] → ...

The first element of each token is a tag (normally an integer) that identifies the type of entity found, the second is its representation. This stream is semantically richer for the parser than the pure characters alone. In addition, characters that are not semantically relevant are wiped from the input, such as white space and comments; they don't get a representation as tokens in the output stream ■

The figure illustrating the lexical analysis sub-machine of figure 3.1 is thus very simple:




### 3.2.1 Streams

The stream abstraction is useful to make transparent to the client code the specific source of elements it consumes:

```
make_abstract_istream(): {
  peek()::          <abstract>
  skip()::          <abstract>
  mark()::          <abstract>
  next()::          { item: peek(); skip(); item };
  exhausted()::     is_void(peek());
  rollback(mark)::   ...
   ...
  clone() }
```

**peek()** Returns the next character available or void when the input is exhausted.

**skip()** Advances the stream position one character.

**mark()** Returns the current position of the stream.

**rollback(mark)** Sets the current position of the stream to the given mark.

Because of the parsing technique we will be using (described in section 3.3), `mark()` and `rollback()` will prove to be useful when a parser needs to go back in the parsing process in order to try out more alternatives at a given point of the input.

In terms of the abstract input stream it is possible to define two types of concrete streams, a *character input stream*,

`make_char_istream(string)` — like the first stream shown in example 3.2.1

which encapsulates a string of characters and is used by the scanner, and a *token input stream*,

`make_token_istream(scanner, char_istream)` — like the second stream of example 3.2.1

which encapsulates a scanner and a character stream to yield a sequence of tokens useful to parsers. The implementation of `make_token_istream()` can be seen in section A.1 (of appendix A), where a technique to avoid scanning the input more than once is described.

### 3.2.2 The xPic% scanner

The xPic% scanner is based on an abstract scanner,

```
make_abstract_scanner(): {
   ...
  next(char_istream)::        <abstract>
  register_token_tag(name)::  ...
  get_token_tag(name)::       ...
   ...
  clone() }
```

**next()** Returns the next available token found in the stream given as argument. Note thus that the scanner is stateless: a different stream can be passed each time.

**register_token_tag(), get_token_tag()** An abstract xPic% scanner maintains a dictionary of the numeric tags it uses to identify tokens. When a scanner is attached to a parser, the parser can query which tags correspond to the tokens it needs; the tokens are identified by strings. It can happen that the scanner doesn't support a specific token type, in which case the scanner and parser are incompatible and cannot be plugged together[1]. This architecture makes scanners and parsers independent from each other and thus reusable in different interpreters (or compilers).

The concrete xPic% scanner does two things: it registers the specific tokens it supports and implements the `next()` method:

[1] The message `"token name not supported by scanner"` appears in the user interface.

```
xpicoo_make_scanner(): {
  >-- make_abstract_scanner();
   ...
  token_NAME: register_token_tag('name');
  token_STRING: register_token_tag('string');
  token_NUMBER: register_token_tag('number');
   ...
  next(char_istream):: {
    state: state_START;
    token_tag: void;
    scanned_data: '';
    while(is_void(token_tag), {
       ...
    });
    [ token_tag, scanned_data ] };
  clone() };

xpicoo_scanner: xpicoo_make_scanner()
```

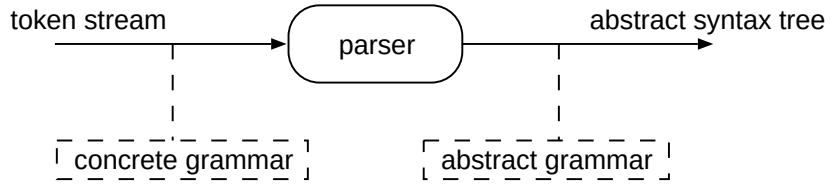
The `>--` operator means "inherits from", it is a (reflective) extension of Pic% to support dynamic inheritance (recall section 2.3.2).

The scanner is implemented as a finite state automaton using state transition tables (which are normal, hard-coded Pic% tables). From the `next()` method we can see that the automaton is initialized and then executed in a `while` loop. In the last line of `next()` it can be seen that the result token is made of an identifying tag and its associated *lexical value* (i.e the scanned data: it can be empty, a number, a string, etc.).

The possibility of making the scanner extensible was not explored in this work. However, being based on state transition tables, the architecture wouldn't be difficult to modify in order to support extensibility.

## 3.3 Syntactic analysis

Having explained the first layer of abstraction provided by the scanner, we can focus our attention now on the syntactic analysis sub-machine. The key tools to build the syntactic analysis framework are parsers. A parser expects as input a stream of tokens conforming to a *concrete grammar*, and imposes a hierarchical structure on the token stream to produce as a result an *abstract syntax tree* (AST), conforming to an *abstract grammar*. Whereas the concrete grammar encodes programs as understood by the programmer, the abstract grammar is the language understood by the evaluator machine. The syntactic analysis phase is in charge of performing the translation between the two representations:




This illustration corresponds to the syntax analysis sub-machine of figure 3.1.

In our case, the concrete grammar is the grammar of the xPic% programming language, used to write valid xPic% programs, and the abstract grammar is the specification of valid ASTs for the xPic% evaluator. Both the concrete and abstract grammars are extensible. We will see how this is achieved in the following sections.

### 3.3.1 A simple parser library

The parser of xPic% is not one big, complex unit but rather a combination of many simple parsers. All of them are based on a generic parser:

```
abstract():: {
   ...
  parse(token_stream)::          <abstract>
  set_ast_cons(new_ast_cons)::   ...
  get_ast_cons()::               ...
   ...
  clone() }
```

**parse()** Returns the abstract syntax tree it finds in the given token stream, or `void` in case the input doesn't match any pattern recognized by the parser.

**set_ast_cons()** Changes the function used by the parser to build the resulting AST in each invocation of `parse()`. This will be discussed further in sections 3.5.2 and 3.5.3.

**get_ast_cons()** Returns the constructor currently in use by the parser.

Deriving from this generic parser, some concrete, *primitive* parsers are provided. As the syntax is extensible, there aren't many predefined parsers: complex parsers are supposed to be built up from the simple ones, using the technique described in section 3.4. The set of parsers is:

```
make_parser_library(scanner): {
  abstract()::  ...
  failure()::   ...
  success()::   ...
  terminal(terminal_name)::  ...
  char_satisfy(predicate):: ...
  char_literal(char)::  ...
  char_literal_string(_string)::  ...
   ...  };

parselib: make_parser_library(xpicoo_scanner)
```

We already described the `abstract()` parser. Regarding the others,

**parselib.failure()** Is a parser that always fails without consuming any input; it always returns `void`.

**parselib.success()** Is a parser that always succeeds without consuming any input; it returns an empty AST.

**parselib.terminal(terminal_name)** Recognizes a terminal of the grammar, i.e. a single token, identified by the string `terminal_name`. This string is used to ask the scanner for the numeric tag it uses to identify tokens of that type (recall the method `get_token_tag()` in section 3.2.2). This is the way of fitting together lexical and syntax analyzers: the scanner must support all the terminals required by the sub-parsers of type `terminal()` that conform the main parser of a grammar.

These three primitives are all that is required to build the framework. Other three parsers prove to be useful, however:

**parselib.char_satisfy(predicate)** Expects a 'character' terminal whose associated data (i.e. the character itself) satisfies the given predicate; the argument `predicate` must be a boolean function.

**parselib.char_literal(char)** Matches literal occurrences of characters, for instance "`(`" and "`)`" which are used to delimit argument lists in function application. This characters cannot be special (such as digits or double quotes) because those are consumed in other type of tokens (numbers, strings), not in tokens of type 'character'.

**parselib.char_literal_string(string)** Like `char_literal()` but matches a whole string instead of a single character.

These three parsers can be defined in terms of the former three ones. In the implementation, the former three are in fact manually in-lined in the latter three for efficiency reasons.

With this simple set of six parsers, we can proceed to build more complex ones, using the technique described in the following section.

## 3.4 Parser combinators

In functional programming, a popular approach to building recursive-descent parsers is to model them as functions. To implement grammar constructions such as sequencing, choice, and repetition, higher order functions (i.e. functions whose parameters are functions) are used. These higher-order functions are called *parser combinators* [34]. The technique can be easily expressed in object-oriented terms. We already defined what an abstract parser is (section 3.3.1); a combinator will be an object conforming to this abstract specification, i.e. it is a parser, whose job is to compose together other parsers in different ways (to be explained next). Since a combinator is also a parser, combinators of combinators are also allowed, like in the Composite design pattern [29]. Parser combinators are the tool that will enable us to build and manipulate dynamic grammars easily.

### 3.4.1 A simple combinator library

We will explain combinators using the small yet complete combinator library available in the implementation of xPic%:

```
make_combinator_library(parselib): {
  all@parsers::  ...
  any@parsers::  ...
  repeat(parser)::  ...
```

```
  optional(parser)::  ...
  strict_repeat(parser)::  ...
   ...  };

comblib: make_combinator_library(parselib)
```

Each one of these combinators inherits (by means of the operator `>--`) from the abstract parser and redefines the `parse()` method, for example:

```
all@parsers:: {
  >-- parselib.abstract();
  parse(istream)::  ...
  clone() }
```

For the other combinators the structure is similar. Now, what is the job of each combinator?

**comblib.any@parsers** Takes an arbitrary list of parsers and applies each one in order to the input stream. As soon as one of them succeeds, the combinator succeeds with the resulting AST of that parser. Between each try, the combinator marks and rolls back the stream pointer (recall section 3.2.1) so that each parser tries to parse the input starting from the same point. Example:

`comblib.any(parselib.char_literal("+"), parselib.char_literal("*"))`

is a parser that tries firstly to match the character "`+`"; if it succeeds, the corresponding parse tree is returned (which actually is just the Pic% table `["+"]` as will be seen in section 3.5); if doesn't succeed, an attempt is made to match the character "`*`", and the parse tree is returned on success or `void` otherwise.

The `comblib.any@parsers` combinator is like an existential qualifier ∃ that asserts (constructively, since the result is obtained): 'in the given set of parsers, there exists at least one that matches the input stream'. An empty list of parsers will produce an always-failing parser, equivalent to `parselib.failure()`, which is in concordance with the semantics of the existential qualifier: in absence of a solution the predicate is false.

**comblib.all@parsers** Takes an arbitrary list of parsers and applies each one in order. If one of them fails, the whole combinator fails (and the input stream is rolled back), otherwise an AST composed of all the sub-trees produced by the parsers is returned. Example:

`comblib.all(parselib.char_literal("+"), parselib.char_literal("*"))`

is a parser that matches a "`+`" followed by a "`*`"; the produced parse tree, in case of success, is `[ ["+"], ["*"] ]`, or `void` otherwise.

The `comblib.all@parsers` combinator is the analogous of an universal qualifier ∀, testing that all the parsers in the given set match the input stream; an empty set of parsers produces an always succeeding parser, equivalent to `parselib.success()`. This conforms with the semantics of the universal qualifier.

**comblib.repeat(parser)** Applies the given parser as far as it doesn't fail, thus consuming many times the same syntactic construct if it is available in the input stream. Failing to apply the parser at least once results in a trivial success (i.e. it behaves like `parselib.success()`) . For example,

`comblib.repeat(parselib.char_literal_string("=>"))`

is a parser that matches the string "`=>`" as many times as it is present in the token stream.

Even though these three combinators are expressive enough to implement all the grammatical constructs of xPic%, two other combinators are included to make the library closer to a regular expression library (more on this in section 3.4.2 below):

**comblib.strict_repeat(parser)** The same as the `comblib.repeat()` combinator except that failing to apply the parser at least once results in a failure. This combinator can be implemented in terms of previously existing combinators:
`comblib.strict_repeat(parser)` ≡ `comblib.all(parser, comblib.repeat(parser))`.

**comblib.optional(parser)** Tries to apply `parser` and, if it succeeds, returns the resulting parse tree; otherwise, the combinator succeeds trivially. This combinator can be implemented in terms of previously existing combinators and primitive parsers:
`comblib.optional(parser)` ≡ `comblib.any(parser, parselib.success())`.

Although it is not a formal restriction, notice that all the combinators shown are constructed exclusively in terms of other parsers given as arguments, making the library design conceptually clean.

### 3.4.2 Using combinators to express EBNF rules and regular expressions

Combinators allow to express *regular expressions* [33] on parsers:

**union operation**
The analog of `comblib.any@parsers` is the operator |, that is to say
$\texttt{comblib.any(parser}_1\texttt{, ... , parser}_n\texttt{)} \equiv \texttt{parser}_1 \mid \texttt{parser}_2 \mid \ldots \mid \texttt{parser}_n$

**concatenation operation**
The analog of `comblib.all@parsers` is the default concatenation operator, which doesn't have a symbolic representation because it is the default operation in regular expressions:
$\texttt{comblib.all(parser}_1\texttt{, ... , parser}_n\texttt{)} \equiv \texttt{parser}_1\ \texttt{parser}_2\ \ldots\ \texttt{parser}_n$

**Kleene closure**
$\texttt{comblib.repeat(parser)} \equiv \texttt{parser}*$

**positive closure**
$\texttt{comblib.strict\_repeat(parser)} \equiv \texttt{parser}+$

**zero or one instance**
$\texttt{comblib.optional(parser)} \equiv \texttt{parser}?$

On the other hand, for specifying the syntax of a language, there is a widely used notation called the *Extended Backus-Naur Form* (EBNF [33]). A sample EBNF rule used in the Pic% grammar is:

<operation> ::= <comparand> | <comparand> <comparator> <operation>

The combinators `comblib.any@parsers` and `comblib.all@parsers` can be used to translate this rule into Pic% code:

```
operation:
  comblib.any(comparand, comblib.all(comparand, comparator, operation)))
```

It can be seen that the correspondence with EBNF notation, as in the case of regular expressions, is straightforward.

### RegExp/EBNF notation

In EBNF notation, some common constructs like repetition or optional elements need two rules or two options in a rule to be expressed. We can think of a mix between EBNF and regular expressions to ease the expression of these routine constructs in a grammar. For example the rule shown before can be written as:

<operation> ::= <comparand> (<comparator> <comparand>)*

We will call this mix of both EBNF and regular expression notations, *RegExp/EBNF notation*. Because of the correspondences

1. combinators $\longleftrightarrow$ regular expressions, and
2. combinators $\longleftrightarrow$ EBNF rules

shown before, it is very easy to translate RegExp/EBNF grammars into Pic% code and vice versa. A more complex example used in the grammar of xPic% is function application:

<application> ::= ≺reference≻ " ( " (<expression> " , ")* <expression> | ≺success≻ " ) "

The notation ≺*terminal*≻ will be used to differentiate terminals and non-terminals in the grammar. The above rule maps to:

```
entry_comma_parser:
  comblib.all(expression_parser, parselib.char_literal(","));
mult_entries_parser:
  comblib.all(comblib.repeat(entry_comma_parser), expression_parser);
args_parser:
  comblib.all(parselib.char_literal("("),
  comblib.any(mult_entries_parser, parselib.success()),
  parselib.char_literal(")"));
application_parser:
  comblib.all(reference_parser, args_parser);
```

The point in favor here is that RegExp/EBNF notation is much easier to read when designing and documenting the grammar — it can be better than pure EBNF notation since a lot of routine grammatical constructs can be expressed concisely and intuitively. The second advantage is that a very direct translation is available to make such an specification a reality (i.e. to obtain an implementation), which makes the code clear and the translation of grammar into code less prone to errors, and the inverse process as well: the grammar rules of xPic% presented in this document were reverse-engineered from the implementation made in terms of combinators. In the remainder of this thesis, we will specify xPic% grammars using RegExp/EBNF grammar notation.

### 3.4.3 Primitive parsers and combinators as reificators of the grammar

From the meta-programming point of view (recall section 2.1), primitive parsers are a reification of the concept of terminal at the base-level grammar, and combinators reify the rules used to build non-terminals in that grammar. Consider for instance the `application_parser` shown in the previous section. It is able to recognize function applications such as `f()`, `f(x)` and `f(x,y,z)`; it uses primitive parsers to reify the terminals "`(`" "`,`" "`)`" and ≺success≻, and `comblib.all()`, `comblib.any()` to reify the different parts of the given grammar rule. The resulting parser, `application_parser`, which is in fact the outermost `comblib.all` combinator, reifies the non-terminal <application> as a whole.

### 3.4.4 Syntactic extensibility through the disjunction combinator

Now that we know what combinators are, we will see how to use them to build extensible grammars. The only way of extending a grammar is by adding rules, or by adding options to a rule by means of the operator |. The approach we present uses the second alternative. As will be shown, this will enable us to extend grammars in a *controlled* manner: the grammar will be configurable only at certain spots and in certain ways.

If we are going to add options to rules with the disjunction operator | of RegExp/EBNF, our main tool for extensibility will be its corresponding combinator, `comblib.any@parsers` (recall the beginning of section 3.4.2). Creating the combinator

`rule: comblib.any(parser`$_1$`, parser`$_2$`, ... , parser`$_n$`)`

yields the rule in the grammar:

rule ||= parser$_1$ | parser$_2$ | ... | parser$_n$

We will call these *configurable rules* because, as we will see next, they can be modified after they have been created. Notice that we use the notation ||= instead of ::= to highlight the fact that these rules are a special kind of disjunction in a grammar: an extensible one.

As said in section 3.4.1, an empty disjunction is equivalent to an always failing parser (correspondingly an always non-matching rule), thus the combinator

`rule: comblib.any()`

yields the rule

rule ||= ⊥

which matches no stream of tokens whatsoever. Empty rules will be seen in the default grammar of xPic%, in section 3.6.

Now that configurable rules have been defined, a way to modify them is needed in order to allow syntactic extensibility. To this end, the disjunction combinator `comblib.any@parsers` has two methods more than the other combinators:

```
any@parsers:: {
  >-- parselib.abstract();
  parse(istream):: ...
  add_before(after_parser, new_parser):: ...
  append(new_parser):: ...
  clone() }
```

The tool to extend disjunction rules is those two methods,

**add_before(after_parser, new_parser)** Inserts `new_parser` as a new option of the rule, with precedence over the already-existing option `after_parser`. When the combinator is invoked, it will try to match new_parser before it tries to match `after_parser`. The rule option `after_parser` must exist for `add_before()` to succeed.

In terms of grammar rules,

rule ||= $\text{parser}_1$ | $\text{parser}_2$ | ... | $\text{parser}_n$

will be transformed after a call to `rule.add_before(`$\text{parser}_j$`, new_parser)` into

rule ||= $\text{parser}_1$ | $\text{parser}_2$ | ... | new_parser | $\text{parser}_j$ | ... | $\text{parser}_n$

**append(new_parser)** Similar to `add_before()`, but the given parser will be added as the last option of the rule.

A method `add_after()` could also have been defined but it didn't turned out to be necessary.

In section 3.6 we will see how configurable disjunctions are used to make the xPic% grammar extensible at certain predefined spots, therefore achieving controlled grammar extensibility. With the ||= notation, it will be easy to locate them at a glance in the grammar definition, for instance in the grammar:

```
<operand>     ||=  ≺string≻ | ≺number≻ | ≺reference≻ | <table> | <application>
<table>       ::=  "[" <expression> * "]"
<application> ::=  ≺reference≻ "@" <operand>
<expression>  ||=  <operand>
```

### 3.4.5 Drawbacks of parser combinators

Parser combinators have two drawbacks:

1. The parsers built are recursive-descent with unlimited lookahead, thus the kind of grammars that can be parsed is restricted to the family LL(n).
2. The constructed parsers can be quite slow.

A solution to problem 2 is discussed in appendix A; regarding item 1, it constitutes a more fundamental limitation and cannot be overcome unless other parsing techniques are used. One challenging direction for future work is to study extensible syntax mechanisms in LR parsers: how to modify the transition tables of the stack automaton that encodes the grammar, when a change in the syntax of the language is requested by the user at runtime. Note however that, in spite of limitation 1, LL(n) grammars are expressive enough to describe most syntactic constructs in programming languages [33].

## 3.5 Building ASTs

Until now, we know how to obtain parsers for LL grammars using primitive parsers and parser combinators. There is still an open issue: what kind of trees those parsers return? The answer to this question is important if the produced ASTs must conform to the abstract grammar expected by the evaluator machine (recall section 3.1).

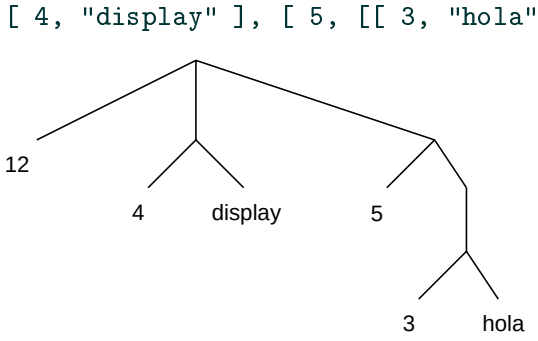



Figure 3.2: representation of trees using Pic% tables

ASTs are represented using Pic% tables, as illustrated in figure 3.2. In this section we will see how basic parse trees are obtained and how to build valid abstract syntax trees starting from these basic trees.

### 3.5.1 Primitive parse trees

The most basic trees are returned by primitive parsers:

**parselib.failure()** → `void`
: The always failing parser never returns a tree.

**parselib.success()** → `[ ]`
: The always succeeding parser returns an empty parse tree, i.e. an empty table.

**parselib.terminal(terminal_name)** → `[ data ]`
: Returns a tree containing the value associated with the recognized terminal. Thus a parser for terminals of type "number" returns trees containing single numbers, a parser for "string" terminals returns trees containing single strings, and so on.

**parselib.char_satisfy(predicate), parselib.char_literal(char)** → `[ char ]`
: For both parsers the result is a tree containing the literal character found.

**parselib.char_literal_string(string)** → `[ string ]`
: Always returns the tree containing the literal string matched in the input.

Recall that parsers return a tree only if they succeed, otherwise they return `void`. The trees returned do not comply with the abstract grammar of the evaluator, for instance their first element is not a numeric tag identifying their type, which is a requirement in valid abstract syntax trees (as will be shown in chapter 4). In the next section, the way to obtain valid ASTs from primitive parse trees is explained.

#### Remark

As we saw in section 3.2.2, the scanner produces tokens `[ token_tag, scanned_data ]` annotated with a tag and the scanned data; the trees returned in the last four parsers above contain in fact the `scanned_data` taken from the tokens found in the input stream. On the contrary, note that the `token_tag`, although may be accessed inside the `parse()` method of some of the parsers, is never used in the final trees produced. This implies that the resulting trees are completely independent from the particular scanner used, since they never

contain scanner-specific information. The tags are an "internal protocol" for the scanner to communicate with the parsers. This remark is made because, as explained in the following section, ASTs are also identified with a numeric tag; however, this tag is not related in any way with token tags.

### 3.5.2 Basic AST constructors

Recall, from section 3.3.1, the method `set_ast_cons()` available in every parser. This method receives as argument a function, and installs it as the new AST constructor of the parser. An AST constructor is in charge of transforming trees so that they become valid with respect to the abstract grammar. Consider the parser for terminals of type "string" in xPic%,

```
string_parser: parselib.terminal('string')
```

If we have the expression `"hola"`, its parse tree will be

```
[ "hola" ]
```

which doesn't comply with the abstract grammar of the evaluator. Here AST constructors become very important: we can change the way string_parser produces its trees. To fix string_parser we can do:

```
string_parser.set_ast_cons( lambda(string): [ tag_STR, string ] )
```

where tag_STR is a predefined numeric tag that identifies string expressions. Now, for the same example expression `"hola"`, and with with `tag_STR = 3`, the produced tree would be:

```
[ 3, "hola" ]
```

which is a valid AST, representing the string. In this way string_parser produces output that can be used to compose more complex ASTs, or it could be passed directly to the evaluator machine for evaluation. For other types of primitive parsers the AST constructors are analog:

```
number_parser: parselib.terminal('number');
number_parser.set_ast_cons( lambda(number): [ tag_NUM, number ] );
ref_parser: parselib.terminal('name');
ref_parser.set_ast_cons( lambda(ref): [ tag_REF, ref ] );
```

and so on.

#### A note about meta-programming

The tree `[ 3, "hola" ]` is in fact a representation of the base-level string `"hola"` at the meta-level (recall section 2.1). Suppose that `tag_REF = 4`, `tag_NUM = 2` and `tag_TAB = 5`; then some examples would be:

- `[ 4, "hola" ]` is the meta-level representation of the base-level variable called "hola",
- `[ 2, 1492 ]` is the meta-level representation of the base-level number `1492`, and
- `[ 5, [ [ 2, 1], [ 2, 2 ] ] ]` is the meta-level representation of the base-level table `[1, 2]`.

**Management of tags in the abstract grammar**

The concrete value of tags like `tag_STR`, `tag_NUM`, `tag_REF` or any other tag is never encoded 'by hand' in AST constructors; in fact the actual value of tags is not predefined, but established at runtime. In this section, we have shown AST constructors with hard-coded tags to make the explanations easy to understand. But, as an example, the true AST constructor used for strings is in reality:

```
string_parser.set_ast_cons(lambda(string): meta.deify_string(string))
```

and there are analog constructors for all the other value types. The `meta` object is in charge of keeping track of the available tags in the abstract grammar of xPic%.

### 3.5.3 More elaborated AST constructors

Now that we have seen AST constructors, the aim of this section is to show more elaborate examples using them to configure combinators, instead of primitive parsers. AST constructors are specially useful when used with combinators. Let's take as an example the quoted code parser of xPic%,

```
quoexp_parser: comblib.all(parselib.char_literal("'"), expression_parser);
```

Recall from section 3.4.1 that the `comblib.all@parsers` combinator produces a tree that is just an ordered grouping of the trees returned by the sub-parsers. In the example, the first subcomponent `parselib.char_literal("'")` always returns the primitive tree `[ "'" ]`, and the second subcomponent returns a whole AST corresponding to the arbitrary expression that is going to be quoted. Suppose we want to parse the expression:

```
'display("hola")
```

The AST corresponding to the subexpression `display("hola")` (i.e. without the apostrophe) is shown in figure 3.2; thus the resulting AST produced by `quoexp_parser` would be:

```
[ ["'"], [ 12, [ 4, display ], [ 5, [ [ 3, hola ] ]]] ]        (3.1)
```

which is not a valid AST because its first component is not a numeric tag. We can tweak the parser, as in section 3.5.2, using the AST constructor:

```
quoexp_parser.set_ast_cons( lambda@tree: [ tag_QUO, tree[2] ] )
```

In the function `lambda@tree`, the component `tree[2]` corresponds to the tree of the expression to be quoted, `[ 12, [ 4, display ], [ 5, [[ 3, hola ]]]]`. Notice that the newly installed AST constructor never uses `tree[1]`, whose value is always `["'"]`, so it can be discarded safely since it provides no new information. The second component `tree[2]` is used to build the final AST. Supposing that `tag_QUO = 13`, the result would be:

```
[ 13, [ 12, [ 4, display ], [ 5, [ [ 3, hola ] ]]] ]
```

One final example for parsing function applications with @-syntax:

```
at_application_parser:
  comblib.all(ref_parser, parselib.char_literal('@'), operand_parser);

at_application_parser.set_ast_cons(lambda@apl: [ tag_APL, apl[1], apl[3] ]);
```

```
<operand>        ||=   ≺string≻ | ≺number≻ | ≺reference≻ | <table> | <application>
<application>    ::=   ≺reference≻ "@" <operand>
<table>          ::=  "[" (<expression> ",")* <expression> | ≺success≻ "]"
<power_op>       ||=   ⊥
<power>          ::=   <operand> ( <power_op> <operand> )*
<product_op>     ||=   ⊥
<product>        ::=   <power> ( <product_op> <power> )*
<addition_op>    ||=   ⊥
<addition>       ::=   <product> ( <addition_op> <product> )*
<comparison_op>  ||=   ⊥
<comparison>     ::=   <addition> ( <comparison_op> <addition> )*
<operation>      ::=   <comparison>
<expression>     ||=   <operation>
```

Figure 3.3: The bare-bones xPic% concrete grammar

It can be seen in the AST constructor that only the first and third components of the parse tree produced by the combinator are used to build the final AST, and the second subtree, corresponding to the "`@`" symbol, is discarded since it doesn't give any new information.

## 3.6 The xPic% concrete grammar

Now that we have set up the base of the extensibility framework, we will proceed to describe the bare-bones xPic% concrete grammar: the syntactic base needed to have a language which is usable. What remains to be explained of the syntactic extension mechanism will start from this basic grammar definition. The extension mechanism will enable us to evolve the basic grammar into the full grammar of standard Pic% (section 5.3). Further extensions will also be possible.

xPic%'s *basic* or *bare-bones* grammar is depicted in figure 3.3. A typical xPic% program that uses this grammar looks like:

```
f@[ g@[x, "example"], y ]
```

We will see a complete example of a program with this same syntax in the bootstrap code of xPic% (section 5.1). Basically, the bare-bones grammar defines a language of infix operations with 4 different levels of precedence, on top of a simple functional syntax, as seen in the example expression above. The different configuration spots are commented in the next section, and a discussion of how could the initial grammar be yet smaller is the subject of section 3.6.2.

### 3.6.1 Grammar configuration spots

As can be observed in figure 3.3, the following configuration spots are available:

**<expression>** This is the goal of the grammar. Every program is an expression, thus being able to configure the <expression> rule means being able to configure what a program is in xPic%. In particular, this configuration point will be used to install the syntax for definition (`:`), declaration (`::`) and assignment (`:=`) of Pico (section 5.2.2).

**<comparison_op>, <addition_op>, <product_op>, <power_op>** The grammar defines binary operators with 4 different levels of precedence (in ascending order): comparison → addition → product → power. As initially there are no operators present

in the grammar, the addition operator + (for instance) won't be recognized. If for example we add "+" as an option of the rule <addition_op> and "*" as an option of the rule <product_op>,

<product_op> ||= "*"
<addition_op> ||= "+"

then expressions, such as `1+2*3` would be recognized as valid. Because outer rules (e.g. <addition>) are defined in terms of deeper rules (e.g. <product>), the latter will be recognized before the former (e.g. a try to match multiplicative operations will be done before a try to match additive ones). This is the standard way of enforcing operator precedence in a grammar: `1+2*3` will be recognized as `1+(2*3)` instead of `(1+2)*3` as would happen if we didn't have precedence in operators.

After adding an operator to one of the rules, it becomes available, with:

- infix notation,
- the level of precedence corresponding to the point where it was added (comparison, addition, etc.), and
- left associativity; notice that the grammar doesn't say anything about operator associativity. To define associativity within a level of precedence, AST constructors are used (see section 3.6.3).

**Note 1** We considered the possibility of having arbitrary levels of precedence. This is not included because such amount of flexibility is not necessary for practical purposes. Nevertheless, it would not be difficult since, as can be observed in figure 3.3, all four types of operations (i.e. all the precedence levels) have the same syntactic structure, suggesting the kind of generalization that could be used. In this case, the initial xPic% grammar would be minimal (see section 3.6.2) since it wouldn't have any rule concerning infix operations.

**Note 2** Although not used in the extensible framework presented here, it is possible to configure operator associativity within a given level of precedence to have right associativity; how is this achieved will be a subject of section 3.6.3.

<**operand**> Here, expressions that are the target of infix operators can be defined. Some options come already installed in the default grammar of xPic%: numbers, strings, function applications, etc. (see figure 3.3). Other operand types that could be added are, for example, tables (denoted in Pic% with "[" and "]") and begin blocks (denoted with "{" and "}").

### 3.6.2 Becoming minimalistic

The concrete grammar presented in figure 3.3 can be simplified. Initially, there are no predefined operators, i.e. all the operator rules are empty ($\perp$). The 4 rules for operations can be reduced, consider <addition> for instance:

<addition> ::= <product> ( $\perp$ <product> )$*$

since the empty rule never matches the input, this is the same as

<addition> ::= <product>

```
<operand>     ||=  ≺string≻ | ≺number≻ | ≺reference≻ | <table> | <application>
<table>       ::=  "[" (<expression> ",")* <expression> | ≺success≻ "]"
<application> ::=  ≺reference≻ "@" <operand>
<expression>  ||=  <operand>
```

Figure 3.4: The minimal xPic% grammar

that is, an addition is initially the same as a product. All the operation rules can be reduced in this way, making in the end <expression> equivalent by reduction to <operand>. We obtain the grammar shown in figure 3.4. This grammar is the simplest possible, it is called the *minimal grammar*. It defines the absolutely least number of syntactic constructs needed to have a language which is usable at all. In this document, a different terminology is used to distinguish the grammars of figure 3.4 and 3.3: one is the *minimal* grammar and the other is the *basic* or *bare-bones* grammar. To be able to grow the minimal grammar into the bare-bones grammar of xPic%, the only element missing in the extensibility framework is the definition of arbitrary levels of precedence; an idea of how this could be achieved was mentioned in section 3.6.1.

The only non-trivial constructs in the minimal grammar are function application and table definition by enumeration of elements. Hence it defines a very simple, pure functional syntax, analog to Scheme's: nothing more than either primitive values (strings, numbers, references, tables) and function applications. Although the minimal grammar is the simplest Pico-compatible grammar possible, note that it isn't the simplest grammar ever: Scheme's is simpler, since it has no special rule for function application. But, as was mentioned in the introduction of section 2.3, one of the objectives of Pico is precisely to be a bit syntactically-friendlier than Scheme; function application is one of the points where this is achieved.

### 3.6.3 Reification of the concrete grammar & grammar management

Now that we have presented the xPic% grammar, how can it be represented at the meta-level? One requisite to enable an extensible syntax is to reify the grammar of the language as an object, in order to be able to manipulate it. Firstly we define a generic grammar object:

```
make_abstract_grammar(): {
   ...
  register(name, parser):: ...
  get(name):: ...
  clone() }
```

This grammar object is in charge of registering the grammar terms by name and later fetching them on demand. Note that, conceptually, this grammar object is not the only object that conforms the reification of the xPic% grammar, but, as we mentioned in section 3.4.3, also the primitive parsers and combinators that reify individual terminals and non-terminals.

Secondly, we need a set of tools to enable grammar manipulation. A refinement of the generic grammar object is made to support the particular extensibility framework of xPic%:

```
xpicoo_make_grammar(scanner, meta): {
  >-- make_abstract_grammar(scanner);
  install_disjunction(name): ...
  install_disjunction_alternative(disjunction, name, priority, parser):: ...
  install_userdef_alternative(level, name, priority, parser):: ...
```

```
  infix_operator_category(operand, operator): ...
   ...
  clone() };

xpicoo_grammar: xpicoo_make_grammar(xpicoo_scanner, xpicoo_meta);
```

These tools to manipulate the grammar on the meta-level will be described next. They are used in section 3.7 to allow the manipulation of the grammar from the base-level.

### Setting up disjunction points and adding options

The function `install_disjunction(name)` is used to install a new disjunction point in the grammar. It creates a new `comblib.any` combinator and registers it under the given name, so that it can be fetched later. For instance, the configurable rule <expression> is installed in the xPic% grammar using:

```
expression_parser: install_disjunction('expression');
```

A new `comblib.any` parser is installed under the name "expression"; it is ready for receiving sub-parsers as discussed in section 3.4.4. The other configuration spots are installed in a similar manner:

```
operand_parser: install_disjunction('operand');
power_operator: install_disjunction('power operator');
product_operator: install_disjunction('product operator');
addition_operator: install_disjunction('addition operator');
comparison_operator: install_disjunction('comparison operator');
```

Once disjunction points are created, new alternatives can be added to them with the function `install_disjunction_alternative(disjunction, name, priority, parser)`, which uses the functionality of the `comblib.any` combinator described in section 3.4.4. If the `priority` argument is `void`, the alternative is installed using `disjunction.append(parser)`, otherwise `disjunction.add_before(priority, parser)` is used. The new alternative is registered under the given name. The alternatives that come installed by default in the basic grammar of xPic% are set up in this way. We give three examples:

```
install_disjunction_alternative
  (operand_parser, 'string', void, string_parser);

install_disjunction_alternative
  (operand_parser, 'application', ref_parser, application_parser);

install_disjunction_alternative
  (expression_parser, 'operation', void, operation_parser);
```

### Creating operator categories

To create each level of precedence in the grammar, the following combinator is used:

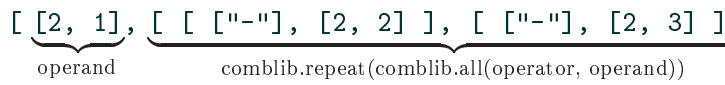
```
left_operation_parser:
  comblib.all(operand, comblib.repeat(comblib.all(operator, operand)));
```

Note that it is a direct reification of the RegExp/EBNF rule

<operation> ::= <operand> (<operator> <operand>)*

which is a generalization of the operations <comparison>, <addition>, <product> and <power> found in figure 3.3.

Supposing that "`-`" is installed as an option of the rule <addition_op>, `left_operation_parser` would produce the following parse tree for the expression `1-2-3`:




We need to install an AST constructor to convert this parse tree into a valid AST. The following constructor will do the job:

```
left_operation_parser.set_ast_cons(
  lambda@tree: fold_left(left_ast_term, tree[1], tree[2]));
```

It will create left-associative ASTs. The function `fold_left(operation, zero, operands)` takes a binary operation, the neutral element $a_0$ for that operation, and a set $[a_1, a_2, a_3, \ldots, a_n]$ of operands; it then applies to the left the binary operation over the operands:

$$\texttt{fold\_left(operation, } a_0\texttt{, [ } a_1\texttt{, } a_2\texttt{, } \ldots\texttt{, } a_n\texttt{ ])}$$

yields the value

$$\texttt{operation(operation(operation(}a_0\texttt{, } a_1\texttt{), } a_2\texttt{), } \ldots\texttt{ ), } a_n\texttt{)}$$

or just the neutral element $a_0$ if the set of operands is empty (n=0).

It would be too much detail to describe the function `left_ast_term(`$op_1$`, `$op_2$`)` passed as an argument to `fold_left()`; it suffices to say that it constructs valid left-associative ASTs for binary operations. For instance if the tag for the "`-`" operator was `25`, `5` for tables and `2` for numbers, the produced tree would be:

```
[ 25, [ 5, [ [ 25, [ 5, [ [2, 1], [2, 2] ] ] ], [2, 3] ] ] ]
```

The subtraction operator doesn't receive just two numbers, but a base-level table containing two numbers. This is because base-level functions implementing user-defined operations do not necessarily need to be binary (more on this in chapter 4). We can see how left associativity is encoded in the above tree (subtraction is applied first to `1` and `2`, and afterwards to `3`). Right associativity can be achieved in a very similar way, using the combinator

```
right_operation_parser:
  comblib.all(comblib.repeat(comblib.all(operand, operator)), operand);
```

which encodes the rule

<operation> ::= ( <operand> <operator> )* <operand>

and then installing the AST constructor

```
right_operation_parser.set_ast_cons(
  lambda@tree: fold_right(right_ast_term, tree[1], tree[2]));
```

where `fold_right(operation, operands, zero)` applies the operation to the right over the set of operands.

In this section we have seen the way in which the meta-level representation of the grammar allows extensibility. Now our aim is to put such tools at the disposition of the xPic% user in the base-level.

## 3.7 User-end tools for syntactic extensibility

Until now, we have seen how the internals of the syntax extension mechanism work. In this section we present the tools made available at the base-level to the end users of xPic%. These tools are a set of meta-functions that enable the modification of the language's grammar at runtime; they are a part of the meta-level interface (MLI) of xPic%.

### 3.7.1 Defining new syntactic constructs

The possible ways in which syntax can be extended are divided by operator arity:

**Fixed-arity operators.** Within this category, we have:

| | | |
|---|---|---|
| prefix operators, | $\bullet\ x$ | like $-$x in Pic% |
| infix operators, | $x \bullet y$ | like x+y in Pic% |
| postfix operators, | $x \bullet$ | like x++ in C |
| fixed mixfix operators, | $x\ \bullet_1\ y\ \bullet_2\ \ldots\ \bullet_n\ z$ | like C's conditional `x ? y : z` |

In general any operator that is not either prefix, infix or postfix, is called a *mixfix operator*. We call the last operator type a *fixed* mixfix operator because it has a fixed number of arguments.

**Arbitrary-arity operators.** There is only one type:

variable mixfix operators, $\bullet_{\text{start}}\ x_1\ \bullet_{\text{sep}}\ x_2\ \bullet_{\text{sep}}\ \ldots\ \bullet_{\text{sep}}\ x_n\ \bullet_{\text{end}}$

like Pic%'s begin blocks, { $\text{exp}_1$, $\text{exp}_2$, ... , $\text{exp}_n$ }. We carry on with the "mixfix" nomenclature, but adding the adjective *variable* to specify that the number of operands is not predefined.

This division of operators by arity inspires the definition of two base-level functions, `var_mixfix_operator()` and `fix_mixfix_operator()`. These allow the user to create parsers in a controlled way, without all the freedom that would give the deification into the base-level of the tools available at the meta-level (primitive parsers and combinators).

| var_mixfix_operator(block_start, block_sep, block_end) |
|---|

This function creates a parser for the RegExp/EBNF rule

$$\prec \text{block\_start} \succ (\text{<expression>} \prec \text{block\_sep} \succ) * \ \text{<expression>} \prec \text{block\_end} \succ$$

this is, a list of expressions of arbitrary length delimited by ≺block_start≻, ≺block_end≻ and separated by ≺block_sep≻. For instance the way to create a parser for Pic% begin blocks would be:
`var_mixfix_operator("{", ";", "}")`, and for Pic% tables,
`var_mixfix_operator("[", ",", "]")`.

| fix_mixfix_operator(arg$_1$, arg$_2$, ... , arg$_n$) |
|---|

`fix_mixfix_operator()` creates a parser for the RegExp/EBNF rule

<arg$_1$> <arg$_2$> ... <arg$_n$>

which means that in principle it just concatenates its arguments, but, to make its usage easier, not all the arg$_i$ need to be parsers: some of them can be strings.

Each argument arg$_i$ can be a string or a parser. If it is a string, the parser

`parser`$_i$ : `parselib.char_literal_string(arg`$_i$`)`

is used instead (recall section 3.3.1). If it is a parser,

`parser`$_i$ : `arg`$_i$

is used (i.e. the parser arg$_i$ without modifications). In the end a list of parsers

`[ parser`$_1$`, parser`$_2$`, ... , parser`$_n$ `]`

is obtained. The fix-mixfix parser is then `comblib.all@parsers`.

**Example 3.7.1** The following examples illustrate some fix-mixfix operator definitions:

`fix_mixfix_operator("(", expression, ")")`
defines a parser for subexpressions within parenthesis,

`fix_mixfix_operator(reference, "[", expression, "]")`
is a parser for standard Pic% tabulations,

`fix_mixfix_operator(".", operand)`
is a parser for standard Pic% super sends, and

`fix_mixfix_operator`
`(reference, ".", reference, var_mixfix_operator("(", ",", ")"))`
is a parser for standard Pic% message sends. A variable-mixfix operator is used to recognize the list of arguments of the invoked method.

The combined use of `fix_mixfix_operator()` and `var_mixfix_operator()` can be quite powerful. The last example shows that, even though syntactic extension is performed in a controlled way, the framework is expressive enough to allow simple specifications for relatively complex constructs ■

One last parser constructor that turns to be useful at the base-level is:

| literal_operator(representation) |
|---|

It just returns a deification of `parselib.char_literal_string(representation)`, which makes it possible to define parsers for literal occurrences of operator characters:

```
literal_operator(">")
literal_operator("+")
literal_operator("=")
```

and so on.

### 3.7.2 Installing & accessing syntactic constructs

New syntactic constructs can be defined using the tools shown in the previous section. Now a way of adding them in the configuration spots of the grammar is needed. The meta-function

| install_syntax(disjunction, name, priority, parser) |
|---|

is used to install new syntactic constructs in the grammar. Basically, it is the base-level equivalent of the meta-level function `install_disjunction_alternative()` seen in section 3.6.3. The only slight difference is that, for each new parser installed, a new value type, with a corresponding freshly assigned abstract grammar tag, is created, and the AST constructor of the user-provided parser is modified so that is constructs ASTs identified with this new tag. The produced trees look like:

```
[ tag, subtree ]
```

where `subtree` is the the tree that would originally be returned by the user-supplied parser if its AST constructor was not modified. The job of the new AST constructor is just to tag the otherwise untagged tree returned by the given parser. In this way the abstract grammar of xPic% gets extended: by installation of new AST types.

**Example 3.7.2** An example of installing a new syntactic construct from the base-level is

```
install_syntax(operand_parser, "braces block", void,
  var_mixfix_operator("{", ";", "}"))
```

which defines syntax for Pico begin blocks. The <operand> configuration spot (recall section 3.6.1) is the target of the addition. Since the third argument is `void`, the new grammar term <braces block> is appended as the last option of the disjunction (recall figure 3.3):

<operand> ||= ≺string≻ | ≺number≻ | ... | <application> | <braces block> ■

**Example 3.7.3** The user can execute

```
install_syntax(expression_parser, "standard definition", operation_parser,
  fix_mixfix_operator(operation_parser, ":", expression_parser)),
```

to install <standard definition> as an option of the rule <expression>; the option is added with priority over (i.e. as immediate predecessor of) <operation>:

<expression> ||= <standard definition> <operation>

After this syntax has been installed, expressions like `id(x):x` (the definition of the identity function) are recognized as valid. Many more examples will be seen in chapter 5 ■

To access an installed syntactic construct at the base-level, the function

| grammar_term(name) |
|---|

is available. For instance `operand_parser: grammar_term("operand")` fetches the parser object that reifies the rule <operand>.

### 3.7.3 Syntactic maps

It is possible, using `parser.get_ast_cons()` (section 3.3.1) to get the AST constructor of a parser. This makes it possible to define another AST constructor in terms of the original. The "wrapping" constructor can invoke the original constructor to obtain the tree it produces, and then transform it as needed. This technique is used, for instance, to tag parse trees in the function `install_syntax()` of section 3.7.2. Another example of post-processing would be partial evaluation of abstract syntax trees.

There is yet another usage of post-processing:

| install_syntactic_map(parser, function) |
|---|

replaces the AST constructor of the given parser by one that firstly invokes the original constructor to obtain an AST, and subsequently applies the given user-defined function to it. The original AST constructor is "wrapped" with a new constructor that post-processes the produced trees in a user-defined way. These user-defined AST transformers are called *syntactic maps* in xPic%.

**Example 3.7.4** An example of the usage of syntactic maps is to define syntax for "weird function applications":

```
install_syntax(operand, "weird application", reference,
  fix_mixfix_operator(reference, var_mixfix_operator("-<", ",", ">-")));

install_syntactic_map(grammar_term("weird application"), application)
```

The function `application(functor, args)`, as will be seen in section 4.3.3, builds the AST of a function application given a target function and the arguments to which it should be applied. With this syntactic map installed, the code

```
display-<"this is a weird application">-
```

would be transformed at parse time into the AST of a normal function application, as if the original code would had been

```
display("this is a weird application")
```

■

Regarding efficiency, the penalty in execution time of syntactic maps occurs at parse-time, since the syntactic map has to be applied whenever the corresponding syntactic construct is found. Syntactic maps do not affect evaluation speed, though.

## 3.8 Conclusion

This chapter has shown how the syntactic analysis machine works and the tools it offers for syntax extensibility at the meta-level. These tools are used to implement a set of meta-functions that allow the user to extend the syntax of the language within the language itself, at the base-level. These functions are part of the meta-level interface (MLI) of xPic%. The intention is not to have a 1-1 correspondence between meta-level and base-level functionality; instead, expressible enough tools are provided so that the end-user can install syntactic constructs easily, without the burden of specifying individual parsers and combinations of

parsers. To this end, the framework is based on a very general notion of "operator", and the functions in the MLI allow to define and install new operators. This meta-level interface happens to be enough for almost any syntax extension, a result that will be shown in chapter 5. In that chapter, the minimal syntax of xPic% will be grown into the full syntax of Pic% and beyond.

# Chapter 4

# Semantic Extensibility

Semantic extensibility is the second core element of the extensibility framework of xPic%. It enables the user to assign a meaning to any syntax extension. As in the case of the previous chapter, the end product of this chapter is a tool set of meta-functions made available at the base-level. This tool set is the second part of the meta-level interface (MLI) of the language; it will be exposed in section 4.3. The sections before provide the necessary background concepts and an understanding of how this part of the MLI is realized.

## 4.1 Overall architecture

The AST produced by the xPic% parser (recall figure 3.1) is the description of a computational system, i.e. it is a *program*. The behavior of such system can be obtained using a semantic analysis machine to *interpret* the program. The xPic% semantic analysis machine, also called the xPic% *evaluator*, is sketched in figure 4.1.

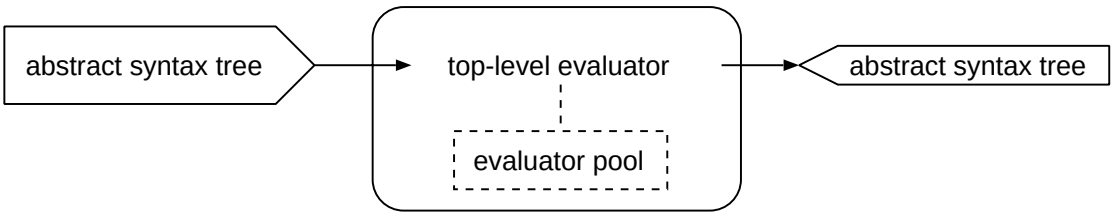



Figure 4.1: The xPic% semantic analysis machine.

The input AST passed to the evaluator must be valid; the output AST will be valid also. We say that a given AST is *valid* if it complies with the abstract grammar of the language. Note that the machine is completely independent from the source code, hence it would be possible to execute programs written in languages other than xPic%, if they are first translated by an appropriate parser to a valid xPic% AST, and this AST is passed as input to the xPic% evaluator.

As seen in figure 4.1, the entry point of the evaluation process is the *top-level evaluator*: an algorithm that, when applied to a valid AST, performs the actions required by the language semantics, and returns the result of such computation. We proceed now to describe the top-level evaluator.

### 4.1.1 The top-level evaluator

The main job of the top-level evaluator is to delegate the responsibility of evaluating an AST to other evaluators maintained in a pool. These evaluators are called *semantic actions*. For each AST type, the top-level evaluator keeps an associated action. When an AST is passed to the top-level evaluator, the tag that identifies the AST type is used as an index to look for the corresponding action in the evaluator pool, and this action is executed. A small protocol is enough to maintain the pool and perform evaluation:

```
make_abstract_evaluator(): {
   ...
  register(exp_type_name, action):  ...
  evaluate(exp)::  ...
  clone() }
```

**register(exp_type_name, action)** Looks for the abstract grammar tag corresponding to `exp_type_name` and associates the given `action` with it. Each time a subtree of type `exp_type_name` is to be evaluated, the `action` will be executed. If there was a previous action associated to `exp_type_name`, it gets overwritten. Note that client code doesn't need to know the particular tag value but a human-readable string which is in a 1-1 correspondence with it.

**evaluate(exp)** Invokes the appropriate action for the given expression, using the information that has been installed up to that moment using `register()`. The top-level evaluator is just a *dispatcher*, all the "real" work of evaluating an expression is delegated to semantic actions. The expression `exp` must be a valid AST.

It is possible to extend or modify behavior, but not to remove it. Although there is no *a priori* reason to forbid behavior removal, that possibility was not considered in this work. It would be enough with a method `remove()` in the top-level evaluator that unregisters an action for a given value type.

**Example 4.1.1 (action definition and registration)** The action associated with numbers in xPic% is

```
register('number', id(x): x)
```

i.e. it is the identity function, hence the result of evaluating a number is just the number itself. More examples will be given in section 4.2.3 ■

As has been explained in this section, for each AST given as input the evaluator is in charge of performing the actions required by the language semantics. Now we proceed to describe such semantics. Firstly, the part of the semantics that is built into the language will be explained. This is called the *intrinsic* semantics. Afterwards, the tools to extend the semantics with user-defined actions will be shown. This extended part is called the *extrinsic* semantics of the language. The intrinsic semantics will be explained in section 4.2, and the tools to extend it are shown in section 4.3.

## 4.2 xPic% intrinsic semantics

In this section the inherent or *intrinsic* semantics of xPic% is defined, i.e. the kernel of concepts and rules that conform the operational definition of the language. This is a cornerstone of the interpretation process.

In xPic%, there is no difference between programs and data, they are both ASTs at the meta-level. Said in another way, "expression" is synonym of "value" in xPic%. Handling values (equivalently, expressions) is all what the xPic% intrinsic semantics is about. The three main concerns are:

- what is the set of value types provided,
- what is the binding semantics of value types (that allows for instance to pass values as arguments of functions), and
- how values should be evaluated, i.e. what the semantic actions associated with each value type.

We will explain these three points in the following sections: 4.2.1, 4.2.2 and 4.2.3.

### 4.2.1 Value types

The value types that come installed in a fresh interpreter of xPic% are shown in this section, together with their respective ASTs. It helps in acquiring a precise idea of how base-level values are represented and the kind of information required for each one. As xPic% is dynamically typed, every single value comes annotated with its type.

***void*** ≡ `[ tag_VOI ]`
: a trivial value, represents the absence of information.

***number*** ≡ `[ tag_NUM, num ]`
: a base-level number; `num` is a meta-level number.

***string*** ≡ `[ tag_STR, str ]`
: a string of characters, called "text" in Pic%; `str` is a meta-level string.

***table*** ≡ `[ tag_TAB, tab ]`
: a base-level table; `tab` is a meta-level table.

***reference*** ≡ `[ tag_REF, ref ]`
: a variable identifier, given by the string `ref`.

***quotedexp*** ≡ `[ tag_QUO, exp ]`
: a 'frozen' expression; `exp` can be any valid AST.

***function*** ≡ `[ tag_FUN, param, body ]`
: a base-level function. Only the formal parameters (`param`) and the body of the function (`body`) are kept. The function name is not relevant, since the same function can be stored in many dictionaries with different names. The type of the parameters determine the way arguments are bound when the function is applied, according to the semantics of value binding that will be defined in section 4.2.2. The body is an expression (i.e. an AST) whose evaluation process is explained in section 4.2.3.

***nativefun*** ≡ `[ tag_NFN, param, type, impl ]`

a base-level representation of a meta-level function. Native functions are similar to functions defined in the base-level: the `param` component is the same, and the `impl` component is equivalent to `body` but is consists of native code instead of an AST. The `type` entry is used for parameter type checking: it avoids having to implement the checks by hand for each native function defined. It is a table `[pred`$_1$`, ... , pred`$_n$`]` of boolean functions (i.e. predicates) that check the type of the value passed as argument. There is a type checker for each value type: `is_base_str()`, `is_base_num()`, and so on. When more than one type is allowed for a parameter, a function of the style `lambda(x): or(is_base_dct(x), is_base_str(x))` can be used (in general any predicate is valid).

***closure*** ≡ `[ tag_CLO, fun, dct ]`

a function `fun`, which can be user-defined or native, together with the environment `dct` where it should be evaluated.

***application*** ≡ `[ tag_APL, fun, args ]`

represents the application of a function `fun` to the actual arguments `args`, as explained in section 4.2.3.

***dictionary*** ≡ `[ tag_DCT, dct ]`

a base-level dictionary; every evaluation environment in xPic% is a value of type this type, hence its importance. The entry `dct` is a meta-level object with the protocol presented in section 2.3.1 for Pic% dictionaries. One way of creating dictionaries is by cloning (shallow copying a previously existing dictionary). A second way is the meta-level function:

```
xpicoo_build_dictionary(var_env, const_env, parent): {
  define(name, value)::   ...
  declare(name, value)::  ...
  assign(name, value)::   ...
  lookup_all(name)::      ...
  lookup_const(name)::    ...
  lookup::                ...
   ...
  clone() }
```

There are more ways of creating dictionaries, for instance there are also "protected" dictionaries, which will be explained later in this chapter. Any object conforming to the required protocol could be used.

***nativedict*** ≡ `[ tag_NDC, dct ]`

a base-level representation of a meta-level object; `dct` can be *any* Pic% object present at the meta-level, including but not limited to dictionaries. For instance, the function `grammar_term()` from section 3.7.2 returns meta-level parsers, which are represented at the base-level as values of type *nativedict*.

Note that six of these value types were introduced in section 2.3.1 from the user's perspective (i.e. base-level perspective). The presentation in this section has been carried out from the language's implementor perspective (i.e. the meta-level perspective).

#### User-defined value types

The set of value types is not fixed, but it grows as the language gets extended. The format of user-defined value types is `[ tag, exp ]`, where `tag` is an abstract grammar tag (an integer) created when the user installs a grammar rule, and `exp` is any valid AST; this process was explained in section 3.7.2.

### 4.2.2 Value binding

Value binding is a general process that allows the evaluator to perform some fundamental tasks, such as variable definition or function application.

To *bind* a pair of values x $\xleftarrow{\text{assoc}}$ y means to associate them using a given operation *assoc*. The left-hand side value will be called the *parameter* and the right hand side value the *argument*. Value binding is performed by the function:

```
bind_param(param, arg, assoc)
```

The `assoc` parameter is a binary function which associates the parameter with the argument: the most commonly used associations are the methods `define()`, `declare()` and `assign()` of dictionaries (recall section 2.3.1).

**Example 4.2.1** The meta-level expressions

```
bind_param(meta.deify_ref("x"), meta.deify_number(1), cur_dct.define)
bind_param(meta.deify_ref("y"), meta.deify_number(2), cur_dct.declare)
bind_param(meta.deify_ref("x"), meta.deify_number(3), cur_dct.assign)
```

are equivalent (using standard Pic% syntax) to the base-level expressions

```
x:  1          - defines x as a variable initially holding the number 1
y:: 2          - declares y as the constant number 2
x:= 3          - assigns x the value 3
```

respectively. Another example will be seen in section 4.2.3, where function application is explained: `bind_param()` is invoked with the function's formal parameters, the actual arguments, and `paramdct.define` (the environment in which the parameters should be bound to the arguments). Thus function parameter binding follows the exact same semantics as variable definition; in fact, the terminology "parameter/argument" is inspired on function applications. Nevertheless, it is important to note that value binding is a general process, not exclusively related to function parameter binding ■

The way in which the binding process occurs depends on the type of the parameter: based on the type of `param`, the function `bind_param()` delegates its job to more specific binding routines. Now we proceed to describe these routines.

#### Binding to a reference

The simplest case is binding to a reference:

```
bind_ref_param(ref, arg, assoc):
  assoc(meta.get_ref_name(ref), evaluate(arg))
```

`arg` is evaluated and associated to the reference. If (for instance) `assoc` = `dct.declare`, then `dct.declare(meta.get_ref_name(ref), evaluate(arg))` is executed.

This is the only binding procedure that applies `assoc`. As will be shown, all the others delegate their work back to `bind_param()`, so they just pass around `assoc` but they never use it. Hence the point where associations are finally performed has been isolated. Currently, xPic% does not benefit from this fact, although it might be useful in future evolutions of the language's design.

### Binding to a quoted parameter

This binding semantics in new, it does not exist in Pic%. When it is invoked, the parameter gets unquoted, the argument gets quoted, and both results are bound:

```
bind_quo_param(quoparam, value, assoc):
  bind_param(meta.unquote_exp(quoparam), meta.quote_exp(value), assoc)
```

Examples:

| | |
|---|---|
| `('x): 1` | Defines `x` as a quoted expression of type *number*: `'1` |
| `('y): this()` | Defines `y` as a quoted expression of type *application*: `'this()` |
| `quotation('x): x` | Defines a function that returns a quoted version of its argument. |

In the first two examples, the parentheses are needed to force the binding of a quoted reference (`'x` and `'y`) to an expression (`1` and `this()` respectively); if the parentheses were omitted, evaluating for instance the expression `'x: 1` would just quote the whole definition `x: 1`, and no binding process would take place. The same applies for `y`.

### Binding to a function application

Suppose that a parameter `apl` is a function application with the form: `functor(`$x_1$`,...,`$x_n$`)`. Then binding a given argument `body` to `apl` constructs a closure. Firstly, a new function with formal parameters $x_1, \ldots, x_n$ and the given `body` is created (recall the *function* value type from section 4.2.1). Secondly, a new closure is built using this function and the current environment (recall the *closure* value type from the same section). Finally, this closure is bound to the application's `functor`:

```
bind_apl_param(apl, body, assoc): {
  functor: meta.get_apl_functor(apl);
  params: meta.get_apl_arg(apl);
  function: meta.deify_function(params, body);
  closure: meta.deify_closure(function, get_cur_env());
  bind_param(functor, closure, assoc) }
```

**Example 4.2.2** `functor` usually is a reference and `assoc` is the definition operation, so the whole effect of the binding process is to have a new closure defined in the current environment with the name given by `functor`:

| | |
|---|---|
| `a(): 1` | the variable `a` holds a closure whose body is the number 1. |
| `f(x): x+a()` | the variable `f` holds a closure that increments a given argument by 1. |
| `id(x): x` | the identity closure (same as in example 4.1.1). |

#### Binding to a table

If the target of a binding operation is a table, then the argument must be a table of the same size. The entries of the target table are considered a parameter each, and similarly the entries of the source table are considered arguments. A 1-1 binding process is then applied over such sets of parameters and arguments:

```
bind_tab_param(base_params, base_args, assoc): {
  if(meta.is_base_table(base_args), {
    params: meta.reify_table(base_params);
    args: meta.reify_table(base_args);
    size_params: size(params);
    size_args: size(args);
    if(size_params = size_args, {
      for(i: 1, i:= i+1, not(i > size_params),
        bind_param(params[i], args[i], assoc));
      base_args },
      eval_error('wrong number of arguments')) },
    eval_error('argument must be a table')) }
```

This kind of binding is used when a fixed-arity function is applied: to bind the table of formal parameters to the table of actual arguments.

**Example 4.2.3** The following definitions illustrate table binding in xPic%:

| | |
|---|---|
| `[a, b]: [1, 2]` | defines two variables `a` and `b` initialized to 1 and 2 respectively. |
| `['x, id(x)]: [1, x]` | defines the variables `x` as `'1` and `id` as the identity function. |

In Pic%, although the semantics of table binding are programmed into the evaluator, it is only used implicitly during function application, but it cannot be invoked explicitly as shown here: the two expressions above are invalid in Pic%. The example shows the benefits of regarding binding as a general process, that transcends the frontiers of function parameter binding ■

### 4.2.3 Default actions

As was seen in section 3.6.2, bare-bones xPic% is a very simple functional language, thus its main semantics is straightforward to define. An important part was already specified in section 4.2.2. What remains to explain is so small that it takes just a few paragraphs. There are only three non-trivial cases; the most important is function application, which will be shown as the last one. The trivial cases are mentioned first.

#### Trivial cases

The easiest cases are:

```
register('void',       id);
register('number',     id);
register('string',     id);
register('quotedexp',  id);
register('function',   id);
register('closure',    id);
```

```
register('nativefun',  id);
register('dictionary', id);
register('nativedict', id);
```

i.e. no action is performed for these value types; the result of evaluation is the value itself (`id` is the identity function as in example 4.1.1).

### References

To evaluate a reference implies to look for its value in the current dictionary; this value is the result of evaluation:

```
register('reference', eval_ref(base_ref): {
  dct: get_cur_env();
  value: dct.lookup(meta.get_ref_name(base_ref));
  if(is_void(value), meta.deify_void(), value) })
```

### Tables

To evaluate a table implies to evaluate each entry in order and to put the results in a new table which is the result of evaluation:

```
register('table', eval_table(base_table): {
  table: meta.reify_table(base_table);
  i: 0;
  eval_table[size(table)]: evaluate(table[i:= i+1]);
  meta.deify_table(eval_table) })
```

### Function application

To evaluate a function application means to obtain the function value, the table of actual arguments, and to apply the function to them:

```
register('application', eval_application(base_apl): {
  base_function: evaluate(meta.get_apl_function(base_apl));
  base_args: meta.get_apl_args(base_apl);
  base_args:= if(meta.is_base_table(base_args),
    base_args, evaluate(base_args));
  apply(base_function, base_args) })
```

When "`func(`$\texttt{arg}_1$`, ..., `$\texttt{arg}_n$`)`" syntax is used, `[ `$\texttt{arg}_1$`, ..., `$\texttt{arg}_n$ `]` is the table of arguments, whereas in "`func@args`" syntax there is just one function argument, the reference `args`, which needs to be evaluated to obtain the final table of arguments (that's the purpose of the `if` statement in the code above).

Whether the function is of type *nativefun*, *closure* or *function* (recall section 4.2.1) will make `apply()` decide for a different algorithm. The application algorithm for native functions is not of interest in this discussion. For plain functions and closures we have:

```
apply_function(base_function, base_arg, env): {
  origdct: get_cur_env();
  paramdct: xpicoo_make_dictionary(env);
  base_param: meta.get_function_parameter(base_function);
  if(is_void(bind_param(base_param, base_arg, paramdct.define)), void, {
      set_cur_env(paramdct);
```

```
result: evaluate(meta.get_function_body(base_function));
set_cur_env(origdct);
result }) }
```

where `env` can be the current environment (for plain function application) or the closure's environment. As can be seen, to apply a function basically means to evaluate the function's body in an environment which is the augmentation of `env` with the *bound parameters* of the function. Parameter binding was described in section 4.2.2.

That is all about the intrinsic semantics of xPic%; it only remains to see the tools that are made available at the base-level to allow the definition of the extrinsic semantics of the language.

## 4.3 The xPic% meta-level interface

The intrinsic semantics of xPic% has been specified in section 4.2. Now we present the tool set to build the *extended* or *extrinsic* semantics, this is, behavior that can be specified from the base-level instead of being embedded in the interpreter. This tool set contains the meta-functions which conform the second part of the meta-level interface of xPic%, complementary to the first part presented in section 3.7.

**Note**

The extrinsic semantics is not defined in terms of meta-functions exclusively, non-reflective functions like

- `tabulate(tables, index)` to access the table's entry at position `index`,
- `size(table)` to get the number of entries in a table,
- `binary_sum(op1, op2)` to sum two numbers,
- `binary_eq(op1, op2)` to test for equality,
- ...

are used also. It would not be worth however to describe a complete API of functions in this document. What interests us the most is the set of functions that provides access and allows manipulation of interpreter structures otherwise unavailable to the programmer.

### 4.3.1 Evaluator registration

There is a base-level equivalent of the meta-level function `register()` which was used in section 4.2.3 to associate semantic actions with each expression type:

| assign_semantics(exp_type_name, function) |
|---|

registers a user-defined `function` as the evaluator for expressions of type `exp_type_name`. Any previously existing evaluator is overwritten. Each time an abstract syntax tree (say `ast`) of type `exp_type_name` is passed to the top-level evaluator, the given function will be applied passing the tree as argument, i.e. `function@ast`.

**Example 4.3.1** The principal usage of `assign_semantics()` is to assign a semantics to user-defined syntax extensions defined with `install_syntax()` (recall section 3.7.2). An example is to give a meaning to the `^` operator:

```
install_syntax(power_operator_parser, "standard power", void,
  literal_operator("^"));

assign_semantics("standard power", binary_pow)
```

The natively-defined function `binary_pow(a, b)` raises `a` to the power of `b` ■

### 4.3.2 Working with evaluation environments

Functions to work with evaluation environments are shown in this section. Some related concepts were presented in section 2.3.1. One of the most basic functions is:

this()

which returns the current evaluation environment (i.e. the dictionary currently in use by the evaluator).

define(dct, target, value) declare(dct, target, value) assign(dct, target, value)

These 3 functions give access to the value binding semantics of xPic%. Recall from section 4.2.2 the function `bind_param(param, arg, assoc)`; we have:

```
define(dct, target, value)   ≡  bind_param(target, value, dct.define)
declare(dct, target, value)  ≡  bind_param(target, value, dct.declare)
assign(dct, target, value)   ≡  bind_param(target, value, dct.assign)
```

The expressions to the left are base-level code and to the right is the meta-level code that gets actually executed. As can be seen, normal variable definition follows the same semantics as parameter binding in function application. This makes the intrinsic semantics of xPic% smaller and homogeneous.

The dictionary functions `dct.define()`, `dct.declare()` and `dct.assign()` were explained in section 2.3.1; do not confuse these meta-level methods with the base-level functions presented here. The parameters `target` and `value` must be quoted expressions, the `dct` parameter can be omitted, in which case it is assumed to be `this()`.

**Example 4.3.2** The following three examples illustrate `define()`.

`define(this(), 'a, '1)` in xPic% has the same effect as `a:1` in Pic%,
`define(this(), 'id(x), 'x)` is equivalent to `id(x):x`, and
`define(this(), '[a, id(x)], '[1, x])` has the same effect as the previous two expressions together.

Similar examples hold if `define()` is substituted by `declare()` and `assign()` ■

lookup(dct, sym)

`lookup()` searches the symbol `sym` (which should be a quoted reference) in the dictionary `dct`; returns the bound value if found, or `void` otherwise.

parent(dct) setparent(dct, parent)

`parent(dct)` returns the parent dictionary of `dct` (recall section 2.3.1); `setparent()` changes the parent dictionary of `dct` to `parent` (which must be another dictionary or `void`).

clone(dct) extend(dct)

Cloning was explained in section 2.3.3; `extend(dct)` creates a new empty dictionary, whose parent link is set to `dct`. Thus initially all the symbols available in `dct` are also seen from the extension. New symbols introduced in the extension do not affect `dct`, so the only way of affecting the parent dictionary of an extension is through `assign()`.

protect(dct) unprotect(dct)

`protect()` returns a *protected* version of the given dictionary. A protected dictionary hides its variables, leaving visible only the constants. To this end, the `lookup()` method of the dictionary is redefined to be `lookup_const()` (recall section 2.3.1). The implementation clearly illustrates it:

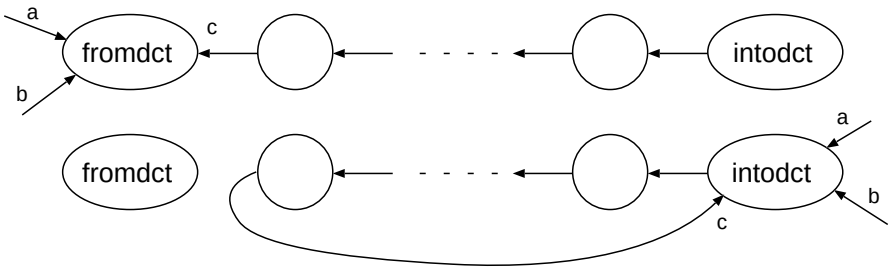
```
make_protected_dictionary(dct): {
  lookup:: dct.lookup_const;
  obj: clone();          (get an object obj with the redefined lookup method)
  obj -->-- dct;         (change the parent of obj to dct)
  obj }                  (return the protected version of the dictionary)
```

`unprotect()` is similar but it redefines the `lookup()` method as `lookup_all()` instead of `lookup_const()`. Thus in an unprotected dictionary both variables and constants are accessible.

become(fromdct, intodct) ancestorbecome(fromdct, intodct)

`become()` transforms all the references to the dictionary `fromdct` present in the system into references to the dictionary `intodct`. The function `become()` is inspired in actor languages.

Like `become()`, `ancestorbecome()` transforms all the references to the first argument into references to the second argument, but with the extra knowledge that the former is an ancestor of the latter in the hierarchy of dictionaries (recall figure 2.4). If `become()` were used to convert all the references to a dictionary into a reference to one of its descendants, the following (wrong) transformation would be performed (`c` is a parent-link reference and `a`, `b` are other references in the system):




i.e. a loop in the inheritance chain; `ancestorbecome()` avoids this problem by changing all the references in the system *excepting* the parent links (if any) that point to `fromdct`.

Hence after a call to `ancestorbecome()`, `fromdct` is still reachable, by navigation of the dictionary hierarchy (using for example the function `parent()` presented above). In the case of `become()`, all references to `fromdct` are lost.

**Remark**

A final note before finishing the explanation about environment manipulation functions: the argument `dct` in many of the presented functions can be omitted, whenever it makes sense to do so; in that case the current environment is used. For instance:

```
define(x, y)  ≡  define(this(), x, y)
parent()      ≡  parent(this())
clone()       ≡  clone(this())
```

### 4.3.3 Expression manipulators

The xPic% evaluator manipulates nothing but *values*. We use interchangeably the terms "expression" and "value" much in the spirit of Scheme: programs and data should not be treated differently. Thus the ability to create or modify *any* type of value at run-time allows programs not only to perform computation about the domain of the application, but also about themselves, which is one of the corner stones of reflection (recall section 2.1.3). To this end, not only numbers, strings or functions are first-class citizens, but also programming constructs like references and function applications are manipulable at run-time. We refer to the functions in this part of the meta-level interface as *value manipulators* or *expression manipulators*.

Values – equivalently, expressions – are always represented by abstract syntax trees. The reader will notice that the signatures of all the value manipulators presented here are related to their respective ASTs in section 4.2.1.

reference(name)

Creates a reference to a variable identified by the string `name`, for instance

```
reference("x")
```

is equivalent in Pic% syntax to just writing down the reference:

```
x
```

quote(value)  unquote(value)  quotation(value)

`quote()` evaluates the `value` passed as argument and returns the quoted result; `unquote()` expects a quoted expression as argument, and it returns the unquoted version; `quotation()` literally quotes the value passed as argument, without evaluating it: `quotation(exp)` is equivalent to `'exp`. The difference between `quote()` and `quotation()` is illustrated with the following example:

```
quotation(1+1)  gives as a result: '1+1
quote(1+1)      gives as a result: '2
```

Note that `quotation()` can be implemented in the base-level, using the semantics for quoted parameter binding: `quotation('exp): exp`. However, this makes use of the '-syntax for quoting expressions, and the semantics for this syntax is defined with `quotation()` (as will be seen in section 5.1.1), so we have a *chicken-and-egg* problem. In xPic%, `quotation()` is defined as a native function: the aim is to have as less syntax as possible.

table(size, entryexp)

Creates a table of the given `size`; the entries are initialized in order with the result of evaluation of `entryexp`, which should be passed as a quoted expression. The Pic% syntax

```
name[size]: entryexp
```

is equivalent to

```
define(this(), name, table(size, entryexp))
```

As an example, if the variable `i` is defined as `0`,

```
x: table(5, i:=i+1)   or   x[5]: i:=i+1
```

are equivalent to

```
x: [1, 2, 3, 4, 5]
```

function(param, body)

Constructs a function with a formal parameter `param` and a given `body`, both to be passed as quoted expressions. Thus the standard Pic% syntax

```
name@param: body
```

is equivalent to

```
define(this(), name, function(param, body))
```

If `param` happens to be a table of parameters `[`$\texttt{param}_1$`, ... , `$\texttt{param}_n$`]`, we would have

$$\texttt{name@[param}_1\texttt{, ... , param}_n\texttt{]: body}$$

which is the same as

$$\texttt{name(param}_1\texttt{, ... , param}_n\texttt{): body}$$

This means that `function()` can create both variable-arity and fixed-arity functions, depending on whether `param` is a single parameter or a table of parameters (respectively). In many functional languages the `function()` constructor is called `lambda` (for instance in Scheme). In $\lambda$-calculus it is the main operation, $\lambda$.

closure(function, env)

Constructs the closure of a given `function`, with a given evaluation environment. If the environment is omitted, the current environment is used. Recall from section 4.2.1 that `function` can be either a plain or native function.

application(functor, arg)

Receives a quoted `functor` and the argument `arg` to which it should be applied, and returns a value of type *application* (recall section 4.2.1) equivalent to

`functor@arg`

in Pic% syntax. As in the case of the `function()` constructor, if `arg` happens to be a table of arguments `[arg`$_1$`, ... , arg`$_n$`]`, then the constructed application would be equivalent to

`functor@[arg`$_1$`, ... , arg`$_n$`]`

which is

`functor(arg`$_1$`, ... , arg`$_n$`)`

Thus `application()` can create both variable-arity and fixed-arity invocations. Example 3.7.4 already illustrated a possible usage of this value constructor.

dictionary(dct)

Creates a new empty dictionary. The optional argument `dct`, if provided, is the parent dictionary. After a dictionary is created, it can be manipulated with the functions presented in section 4.3.2.

**Remarks**

Some remarks are necessary before finishing the presentation about expression manipulators:

- It makes no sense to define constructors for the value types *nativefun* and *nativedict*.
- For *void*, *number* and *string* it would be possible to have constructors `void()`, `number()` and `string()` returning `void`, `0` and `""` (an empty string), respectively, but it is easier to use the built-in syntax for numbers and strings, and the pseudo-variable `void` comes predefined in the global environment.
- Most expression manipulators presented in this section are *value constructors*. Note however that, for each value type, there are also *value accessors*. For instance for values of type *dictionaries*, section 4.3.2 presented the function `parent()` to access the parent of a dictionary. Although they are not mentioned here, there are similar accessors for other value types. For instance, for function applications, the accessors `application_functor()` and `application_arg()` retrieve the application's functor and argument respectively; for closures, `closure_function()` and `closure_environment()` retrieve the encapsulated function and its evaluation environment. Listing all the accessors would make the presentation too lengthy; they can be inferred naturally from the value type definitions given in section 4.2.1.
- Finally, there are also predicates (boolean functions) that enable a program to find out the type of a value at hand. These predicates are `is_void()`, `is_number()`, `is_string()`, and so on for each value type.

### 4.3.4 Accessing basic mechanisms of the interpreter

The functions `read()`, `eval()` and `print()` give access to the three main phases of the REP machine that is interpreting the program (recall section 2.4.1):


`read()` receives as argument a string and, by means of the xPic% syntactical analysis machine (explained in chapter 3), returns the corresponding AST. This AST is a normal xPic% value, with one of the types defined in section 4.2.1.

`eval()` receives as argument an expression and an environment in which the expression should be evaluated; it evaluates the expression by invoking the top-level evaluator (described in section 4.1.1). If the argument `env` is omitted, the current environment (`this()`) is used.

`print()` receives as argument an expression and shows a human-readable representation in the user interface of the interpreter.

With these three functions, the REP-evaluation process of an xPic% program can be implemented at the base-level:

```
print(eval(read(program)))
```

where `program` is a string containing the source code of a program. The syntax of this program must comply with the concrete grammar of xPic% as defined at that moment.

Finally, there is another commonly used function that gives access to the fundamental mechanisms of the interpreter:


applies a function `function` to a given argument `arg`, using as application environment the dictionary `env`. The function application process is defined by the meta-level function `apply_function()` described in section 4.2.3.

## 4.4 Conclusion

We have seen in this chapter the core elements of the xPic% semantical evaluator machine. The semantics was divided into the *intrinsic* semantics of the language and the extended, user-defined or *extrinsic* semantics.

Regarding the intrinsic semantics, the default value types supported by the machine were described, and a meaning (i.e. default semantic action) was defined for each one of them. Furthermore, a way of associating values was described for a subset of the default value types. This *binding semantics* is used during variable definition, declaration, assignment and during function application; in xPic%, function parameter binding has the exact same semantics as variable definition. Some benefits of viewing value binding as a general process were shown, like the ability to bind tables to tables: `[x,y]: [1,2]`, a feature which is not present in Pic%. The extra functionality is obtained at no cost as a consequence of the very simple and orthogonal intrinsic semantics. A second binding semantics which is a novelty is the binding of quoted parameters. It allows, for example, to quote function arguments from the function definition itself, without requiring the client of the function to do so. For instance,

the function `alternativedef('name, exp): define(this(), name, quote(exp))` allows to define variables in the current environment. An invocation like `alternativedef(a, 1+1)` would bind `a` to the value `2`. It is not necessary to quote the reference `a` before it is passed to `alternativedef()`: the function itself takes care of this. In Scheme, a special form would be needed to achieve a similar result.

Regarding the extrinsic semantics, an MLI was defined to support the extension of the default language semantics with evaluators defined at the base-level. The function `assign_semantics()` can be used to install this user-defined evaluators. The evaluators can be implemented using the other functions available in the MLI, apart from standard functions in the API of xPic%. The functions in the MLI are a complement to the standard API to better support reflective computation: xPic% programs can find out information about their execution environment, and inspect and modify any kind of value found in the environment (including parts of the program itself). This means that the MLI enables both introspection and intercession (recall section 2.1.3) in xPic% programs. Note, though, that Pic% is also a reflective language. The contribution of xPic% in this area is to extend the set of reflective facilities and to provide a standardized way of accessing and modifying values. The way in which reflective computation is performed in Pic% was not explained; currently, the only problem is that it is implementation-specific. As an example, the definition of the `>--` operator used throughout chapter 3 is:

```
>-- proto(): {
  curenv: proto[2];
  super: proto();
  curenv[3]:= super;
  super }
```

The hard-coded indices `2` and `3` depend on the C implementation of the Pic% interpreter. They are used, respectively, to access the environment of the closure `proto` and the parent-link of that environment. In Borg [35], a language derived from Pic%, the indices are different.

# Chapter 5

# Validation of the Language Kernel

A language extensibility framework has been explained in chapters 3 and 4. Here we give a "test-drive" of the possibilities that it offers, thereby validating our work. Both syntactic and semantic matters are discussed. The chapter starts by showing how to perform an initial extension of xPic% that provides some basic constructs (section 5.1); from this version of the language, it is easier to express further extensions. In particular the way to transform xPic% into Pico will be shown in section 5.2. After a Pico-equivalent language is obtained, the basic object-oriented layer of Pic% is defined, followed by some further constructs inspired on prototype-based languages (section 5.3).

All the code shown in this chapter is base-level code: we will not be talking anymore about the internals of the xPic% interpreter, which was defined in chapters 3 and 4 using a meta-circular implementation. The examples in the first sections are harder to read than later examples, but, as new syntax is defined, things get easier and easier to express.

## 5.1 Bootstrapping the language

Each time a bare-bones interpreter is started, it executes a program whose purpose is to install a few basic syntactic constructs and function definitions. This *bootstrap* process leaves the language into a more user-friendly state. When the interpreter starts using the minimal syntax of xPic% (recall section 3.6.2), the main jobs of the bootstrap process are:

1. To provide syntax for quotations (e.g. `'exp`).
2. To provide syntax for function application using "parentheses syntax" (e.g. `f(x)`), since the basic grammar includes syntax for `@`-applications only (e.g. `f@[x]`). The `@`-syntax is the more general but it is less intuitive and common than parentheses syntax.
3. To provide syntax for plain function definition (e.g. `f(x):- x`). Recall that the syntax `f(x): x` defines a closure, not a plain function.

We proceed to show how this extensions are performed.

```
[ `- declare needed variables -`

  declare@[quotation@[operand_parser],
    quotation@[grammar_term@["operand"]]],

  declare@[quotation@[expression_parser],
    quotation@[grammar_term@["expression"]]],

  declare@[quotation@[reference_parser],
    quotation@[grammar_term@["reference"]]],

  `- install function application and quotation syntax -`

  install_syntactic_map@[
    install_syntax@[operand_parser, "standard parentheses application",
      reference_parser, fix_mixfix_operator@[
        reference_parser,  var_mixfix_operator@["(", ",", ")"]]],
    application],

  install_syntactic_map@[
    install_syntax@[operand_parser, "standard quotation", void,
      fix_mixfix_operator@["'", expression_parser]],
    quotation],

  display@["language bootstrapped"] ]
```

Figure 5.1: Growing the basic grammar of xPic% to obtain quotations and parentheses applications

### 5.1.1 Growing the minimal grammar

Code written using the basic syntax of xPic% can be very difficult to understand, since *everything* has to be performed by means of function applications (figure 5.1 shows an example). Some "syntactic sugar" would render the language syntax more readable. First of all, syntax for parentheses applications and quotations – two of the most commonly used constructs – will be defined. The program shown in figure 5.1 does this job; it is (of course) written using minimal constructs only. Looking at the first part of the code, three variables are first declared to refer the natively-installed parsers that reify the grammar terms <operand>, <expression> and <reference> (recall figure 3.4). These variables are used in the second part of the code to install two syntactic maps, that transform the ASTs produced by the user-defined fix-mixfix parsers into ASTs corresponding to a quotation and an application, respectively. To this end, the value constructors `quotation()` and `application()` presented in section 4.3.3 are passed as the second argument of `install_syntactic_map()`. Note that the term <standard parentheses application> is installed with priority over <reference>, otherwise it would never be invoked. For <standard quotation> there are no priority problems, thus the third argument of `install_syntactic_map()` is `void` in this case.

#### Note

There is no difference in evaluation speed between an interpreter that comes with the basic grammar and is extended with the code shown in figure 5.1, and one that comes with the extended grammar natively installed. The reason is that, for every expression, the evaluator receives the exact same ASTs in both cases (the two syntactic maps are responsible for this

in the former case). In contrast, parsing efficiency is affected. The penalty in execution time of the parsing process comes from applying the functions `quotation()` and `application()` each time a quoted expression or a `()`-application are found in the source code. The current implementation of xPic% actually uses a native grammar which is a bit bigger than the basic grammar. The native implementation is equivalent to an interpreter that comes with the basic xPic% grammar and is extended with the code of figure 5.1. This is however just a matter of parse-time efficiency, there are not fundamental restrictions in starting up from the basic grammar. In fact, the code of figure 5.1 was tested in a version of the interpreter that starts up with (strictly) the basic grammar installed.

Now we proceed to describe the second important extension performed by the bootstrap process: syntax to make plain function definition concise and easy to read.

### 5.1.2 Installing syntax to create functions

Recall that plain functions are not the same as closures (section 4.2.1). Even though standard Pico has the concept of functions, it provides no syntax to define them; only closure definition is possible. In xPic% it is possible, since we can define any arbitrary syntax for it, and the tools to create functions are available at the base level. The following syntax was chosen (recall the meaning of definition, declaration and assignment from section 2.3.1):

`name(`$\mathtt{arg}_1$`, ... , `$\mathtt{arg}_n$`):- body` for function definition,

`name(`$\mathtt{arg}_1$`, ... , `$\mathtt{arg}_n$`)::- body` for function declaration, and

`name(`$\mathtt{arg}_1$`, ... , `$\mathtt{arg}_n$`):=- body` for function assignment.

and analog syntax for @-definitions. We show how the extension is achieved for the ":-" case; declaration and assignment are practically identical:

```
install_syntax(expression_parser, "standard function definition",
  operation_parser,
  fix_mixfix_operator(application_parser, ":-", expression_parser))

assign_semantics("standard function definition",
  function('['apl, 'body],
    'define(parent(),
      quote(application_functor(unquote(apl))),
      quote(function(quote(application_arg(unquote(apl))), body)))))
```

The semantics are defined by a function that receives the application (`apl`) and the body (`body`) as quoted parameters, then it applies `define()` to bind the function name, obtained with `application_functor()`, to a new function created on the fly by means of the value constructor `function()`. The function formal parameters are obtained from the arguments of `apl` using the accessor `application_arg()` and the body is readily available in the parameter `body`. The reason to use `parent()` as the environment of definition is given in section 5.2.2. This example illustrates many aspects of the framework: creation of syntactic constructs, usage of quoted function parameters and creation and manipulation of values (functions, applications, and quotations). The code is a bit complex to read since we are at a low-level stage of the extension process.

This is all for the bootstrap process, in the following sections further extensions will be performed using the base we have established here.

## 5.2 Going from xPic% to Pico

This section shows how xPic% can be extended to become Pico-compatible (syntactically and semantically). This is neither an unexpected nor a hard result to obtain since the core syntax and semantics of xPic% were distilled from Pico's. Nevertheless, the exploration of the Pico language definition raises some interesting issues that are presented in the sections to follow, in particular 5.2.3, 5.2.4 and 5.2.5. Also, the sample extensions give insight into the way the extensibility framework can be used and the aspects that have to be taken into consideration.

### 5.2.1 Basic constructs

In this section the easiest extensions are shown:

#### Begin blocks

One of the most heavily used syntactic sugarings in Pico is "begin blocks" (recall section 2.3.2):

$$\texttt{\{ exp}_1\texttt{; exp}_2\texttt{; ... ; exp}_n\texttt{ \}} \equiv \texttt{begin(exp}_1\texttt{, ... , exp}_n\texttt{)}$$

Not the syntax nor the semantics (the function `begin()`) exist natively in xPic%, but they can be defined:

```
install_syntax(operand_parser, "braces block", void,
    var_mixfix_operator("{", ";", "}"))

begin@arg:- tabulate(arg, size(arg))
assign_semantics("braces block", begin)
```

The `tabulate()` and `size()` accessors where presented in section 4.3. Note that the ":-" syntax for function definition of section 5.1.2 makes the code easier to read, otherwise the `function()` constructor and `define()` would have to be used.

When the construct $\texttt{\{ exp}_1\texttt{; exp}_2\texttt{; ... ; exp}_n\texttt{ \}}$ is found, $\texttt{begin@[exp}_1\texttt{, exp}_2\texttt{, ... , exp}_n\texttt{]}$ will be invoked. Upon invocation, the table of arguments will be bound to the parameter `arg` of `begin`. According to the binding semantics presented in section 4.2.2, this means that the table will be evaluated and the result bound to `arg`. Evaluating the table means to evaluate in order each of its entries $\texttt{exp}_i$ and to store the results in a new table, thus the actual value of `arg` will be this table of results. The body of `begin` simply returns the last of such results, i.e. the result of evaluation of $\texttt{exp}_n$. The overall effect of the assigned semantics is that all the expressions in the group $\texttt{\{ exp}_1\texttt{; exp}_2\texttt{; ... ; exp}_n\texttt{ \}}$ are evaluated from left to right and that the result of the last one is returned.

This example aimed at showing the detailed semantics of how a user-defined function handles the ASTs produced by a user-defined parser. Even though the `begin` function looks simple, many processes are triggered when it is invoked. It can be seen how important the binding semantics is; for other syntactic constructs, like definitions (section 5.2.2), value binding is central.

**Arithmetic and boolean operators**

Most of the extensions presented in previous chapters have used the <expression> and <operand> configurable rules of the basic xPic% grammar (defined in section 3.6.1), but there are not many examples of operator definition using the configuration spots <comparison_op>, <addition_op>, <product_op> and <power_op>; here we give some. The standard Pico boolean operators are defined as:

```
install_syntax(comparison_operator_parser, "standard less than", void,
  literal_operator("<"))
install_syntax(comparison_operator_parser, "standard greater than", void,
  literal_operator(">"))
install_syntax(comparison_operator_parser, "standard equality", void,
  literal_operator("="))

assign_semantics("standard less than", binary_lt)
assign_semantics("standard greater than", binary_gt)
assign_semantics("standard equality", binary_eq)
```

The relational operations `binary_lt()`, `binary_gt()` and `binary_eq()` are natively defined functions. All the relational operators are installed in the same level of precedence (<comparison>), thus left-associativity applies for chains of operations, e.g. `a < b = c > d` is treated as `((a < b) = c) > d` (recall sections 3.6.1 and 3.6.3). For arithmetic operations the extension is very similar:

```
install_syntax(addition_operator_parser, "standard addition", void,
  literal_operator("+"))
install_syntax(product_operator_parser, "standard multiplication", void,
  literal_operator("*"))
 ...
assign_semantics("standard addition", binary_sum)
assign_semantics("standard multiplication", binary_mul)
 ...
```

Arithmetic operators in the same level of precedence are also left-associative.

### 5.2.2 Standard definition, declaration, and assignment

As we know from section 2.3.2, Pico comes with syntax to define new variables, new constants and to change the value of a variable (with the syntaxes `:` , `::` and `:=` respectively). We show here how the ":"-syntax for variable definition can be obtained; for declaration and assignments the constructs are similar.

```
install_syntax(expression_parser, "standard definition", operation_parser,
  fix_mixfix_operator(operation_parser, ":", expression_parser))

assign_semantics("standard definition",
  lambda('target, 'value):- define(parent(), target, value))
```

An explanation of why the `parent()` environment is used follows. When `lambda()` is invoked, its body is evaluated in an extension of the current environment. The purpose of this

extension is to hold the bound parameters `target` and `value` of the function (recall the semantics of function application in section 4.2.3). Suppose that the definition being evaluated is `a:this()`; as `target` and `value` are quoted parameters, the expressions `a` and `this()` are quoted and passed to `define()`, which unquotes them and performs the binding between the two (using `bind_param()` from section 4.3.2). In this moment the expression `this()` is evaluated. Normally, `this()` would return the environment of execution of the `define()` function, but `define()` ensures that the binding is always performed in the parent environment, which in this case is `lambda()`'s evaluation environment; `this()` would then return that environment, which is not the original environment where the ":" syntactic construct was used. Passing `parent()` to `define()` makes the binding to occur in the environment where the syntactic construct was used.

The aim of this section was to give an example of how the semantics of function application and parameter binding enter into play when defining syntactic constructs and their evaluators. This same explanation holds for section 5.1.2, where the ":-" syntax was defined. The explanation was delayed to this section because at that time we couldn't easily talk about the evaluator `lambda`, since an anonymous function was used.

### 5.2.3 Table definition and table entry assignment

In Pico, the `:` , `::` and `:=` operators can be used to define or declare tables, or to assign table entries respectively; some examples to have the idea from section 2.3.2 fresh in mind:

```
a[2]: void            yields a = [ void, void ]
a[1]:= 2              yields a = [ 2, void ]
i:0; b[5]:: i:=i+1    yields b = [ 1, 2, 3, 4, 5 ]
```

In xPic%, the semantics defined in the previous section for the operators `:` , `::` and `:=` do not work in the case of tables, simply because there is no predefined semantics to bind a tabulation (e.g. `b[5]`) to an expression (e.g. `i:=i+1`). Thus separate syntax and semantics have to be defined to deal with table definition and table entry assignment.

To start with, syntax and semantics for standard Pico tabulations can be defined:

```
install_syntax(operand_parser, "standard tabulation", reference_parser,
  fix_mixfix_operator(reference_parser, "[", expression_parser, "]"))

assign_semantics("standard tabulation", tabulate)
```

Then, syntax for standard table definitions is easy to achieve:

```
tabulation_parser: grammar_term("standard tabulation")
standard_definition: grammar_term("standard definition")
install_syntax(expression_parser, "standard table definition",
  standard_definition,
    fix_mixfix_operator(tabulation_parser, ":", expression_parser))
```

The grammar term <standard definition> was defined in section 5.2.2. Here, the <standard table definition> rule is defined to have priority over it; thus a try to recognize a table definition will always be made before a try to recognize a standard definition. If this were not the case (i.e. if `void` is passed as the third argument of `install_syntax()`), a standard definition would always be recognized first and the interpreter would reply after trying to perform the normal binding process:

```
evaluator: cannot bind a value of type <standard tabulation>
```

since binding is defined for just a few predefined value types (recall section 4.2.2), but never for user-defined types.

To define semantics, standard table definition makes use of the `table()` value constructor (section 4.3.3):

```
assign_semantics("standard table definition",
  lambda('name, size, 'entryexp):-
    define(parent(), name, quote(table(size, entryexp))))
```

A similar extension can be used for table declarations (replacing "`:`" by "`::`" in the operator definition and `define()` by `declare()` in the semantics) and for table entry assignment, replacing "`:`" by "`:=`" and using the semantics:

```
assign_semantics("standard table entry assignment",
  lambda(table, index, entryval):- set_table_entry(table, index, entryval))
```

`set_table_entry()` is one of the value manipulators presented in section 4.3.3.

The three operators `:` , `::` and `:=` deal with dictionary operations (define, declare and assign). The same happens with table definition and declaration. However, table entry assignment is slightly different: its semantics has nothing to do with dictionaries, it just modifies an entry of a table.

### 5.2.4 The boolean system

The boolean system of xPic% is the same as Pico's (Church booleans) with one exception that will be mentioned below. The idea is to define booleans as functions that choose between two options given as arguments (ignore the "`^`" syntax for a moment):

```
true('t, 'f):-   ^t
false('t, 'f):-  ^f
and(p, 'q):-     p(^q, false)
or(p, 'q):-      p(true, ^q)
not(p):-         p(false, true)
```

so for instance `true("hola","adios")` ≡ `"hola"` and `false(1,2)` ≡ `2`; also the standard boolean logic rules hold: `and(true,false)` ≡ `false`, `not(false)` ≡ `true`, etc.

The difference between this boolean system and Pico's boolean system is that the latter uses closures instead of quotation. As an example, `true` is defined in Pico as

```
true(t(),f()): t()
```

A problem similar to that shown in section 5.2.2 arises. Suppose `true(this(),void)` is evaluated; then the closures `t():this()` and `f():void` are passed as arguments to `true()`. When it applies `t()` (its first argument), the closure's environment, which is the environment where the original expression `true(this(),void)` was evaluated, is obtained *and extended* with the environment supposed to hold the bound parameters of `t()` (even though `t()` doesn't have parameters, such an environment is always created). Thus the call to `this()`

returns this extended environment, which probably is not what the user expects[1]. With the quotation-based boolean system shown above, this problem is avoided.

The "`^`" syntax used above is a shorthand for the function

```
eqp(exp):- eval(unquote(exp), parent(parent()));
```

which evaluates the given expression in the parent of the parent of its execution environment. One `parent()` call is made to get out of `eqp`'s environment and one more to get out of `true()`'s evaluation environment (or whatever function that uses the `^` operator). The acronym eqp stands for "(e)valuate (q)uotation in (p)arent environment".

### 5.2.5 Execution control

The execution control functions rely on the way the boolean system is defined. They mimic their counterparts of Pico, except that (again) they are based on quotation rather than closures. One of the fundamental forms of execution control is conditional execution:

```
if(cond, 'then, 'else):- cond(^then, ^else)
```

Evaluating for example `if(true, x:1, x:2)` will leave a variable `x` defined in the current environment with value `1`. For the same reasons of section 5.2.4 a definition like

```
if(c, t(), e()):- c(t(), e())
```

would not work in the way users expect.

The other fundamental form of execution control is iteration. In Pico, it is based on recursivity. The following example is Pico's `while` translated to the quotation system:

```
while('cond, 'exp):- {
  loop(value, pred):- pred(loop(^exp, ^cond), value);
  loop(void, ^cond) }
```

Notice how easy is to express constructs now that we have a more complete grammar.

## 5.3 Object-oriented extensions

This is the last part of the extension tests: to build up an object-oriented system on top of the functional system provided by xPic%. There is not a single object-oriented concept present in the intrinsic semantics of the language. To define the OO layer, the tools that will be used are syntactic extensibility, the meta-level interface, and the evaluation environment playground provided by xPic%.

As was mentioned in the introduction of section 2.2, an object-oriented computational system consists of nothing but objects sending messages to each other. A way to create objects will be shown in section 5.3.1; message sending is described in section 5.3.2. The addition of this two concepts transforms xPic% a complete and consistent object-oriented model. After this minimal OO system is established, some more mechanisms inspired on prototype-based languages are defined as a way of showing the potentialities of the extensibility framework.

[1] Actually, the C implementation of Pico doesn't use a closure-based boolean system: it is implemented natively with a slightly different semantics, so the example expression works. However, the metacircular implementation of Pico uses closures and standard semantics for function application; in this case the example fails.

### 5.3.1 *Ex-nihilo* creation of objects

First of all we need objects. An object will be defined as an evaluation environment (a dictionary), like in Pic%. So we know what objects are. Next, a mechanism to create them is necessary. The reader already knows one mechanism, *Pic%'s way* of object creation, in which an object is a clone of the evaluation environment of a function; thus no special syntax or semantics has to be introduced into xPic% to achieve this mechanism. As an example, the following code creates an object representing a person:

```
make_person(name, age): {
  getname():: name;
  getage():: age;
  clone() };
john: make_person("John Coltraine", 40)
```

To create a concrete object, the function `make_person()` has to be defined and invoked. This extra steps go against the philosophy of prototype-based programming, which is to be as direct and concrete as possible (recall section 2.2.1). To bring the language closer to this philosophy, we will introduce in the language an alternative mechanism for object creation commonly used in prototype-based programming, *ex-nihilo* creation of objects. Creating an object will be as simple as listing its slots in a block delimited by `{-` and `-}`, for example:

```
john: {-
  getname():: "John Coltraine";
  getage():: 40 -}
```

The syntax for ex-nihilo object creation is obtained with:

```
install_syntax(operand_parser, "ex-nihilo object block",
  void, var_mixfix_operator("{-", ";", "-}"));
```

and the associated semantics are:

```
ex_nihilo_creation@'objdef:- {
  object: dictionary(parent());
  eval(unquote(objdef), object);
  protect(object) };

assign_semantics("ex-nihilo object block", ex_nihilo_creation)
```

The `dictionary()` value constructor (from section 4.3.3) is used to create a new dictionary. The parent of the ex-nihilo created object is the environment where the `{- -}` block is being evaluated (which is obtained with `parent()` to skip `ex_nihilo_creation()`'s environment). The object definition found in the block, which is frozen thanks to the quoted parameter `objdef`, is unquoted and evaluated using the freshly created dictionary as evaluation environment, thus installing all the definitions in it. Finally, the object is protected and returned. In this way state encapsulation is achieved (so for instance method lookup doesn't reveal private methods, as will be seen in next section).

### 5.3.2 Message sending

After having objects, we need messages. First we define the syntax:

```
install_syntax(operand_parser, "standard send", reference_parser,
  fix_mixfix_operator(reference_parser, ".", application_parser));
```

which corresponds to the classical "dot" notation of object oriented languages; it looks like:

`receiverobj.message(`$\texttt{arg}_1$`, ... , `$\texttt{arg}_n$`)`; also `@`-syntax can be used:

`receiverobj.message@arg`.

The following function defines the corresponding semantics:

```
standard_send(object, 'apl):- {
  method: lookup(object,
    quote(application_functor(unquote(apl))));
  if(is_void(method),
    display("message not understood"),
    {
      npobj: unprotect(object);
      extenv: extend(npobj);
      declare(extenv, 'self, quote(npobj));
      declare(extenv, 'super, quote(parent(npobj)));
      apply(method, application_arg(unquote(apl)), extenv)
    }) };

assign_semantics("standard send", standard_send)
```

So to send a message means to look up its selector in the the object (which is protected so private slots will not be searched), and, if the method is found, to apply its body. The environment of evaluation of the body is an extension of the object which is unprotected in order to allow the method to see the private state of its host object. Also, the constant binding `self` is defined to refer to the receiver object. This extra "hidden" argument passed to object methods is present in most OO languages: for instance it is also called "self" in Smalltalk and Self, and it is called `this` in Java and C++. Similarly, a binding `super` is declared to point to the parent of the receiver so that the method can refer to its parent; "super" is also present in some languages (e.g. Smalltalk and Java). The advantage of using the `self` and `super` bindings is that the message passing mechanism is very homogeneous: there is only one syntax and one semantics for all possible types of messages.

This is all about message sending in xPic%. A final remark will be made which can be skipped if the reader is not interested in knowing how the semantics of message sending that has been defined slightly differs from that of Pic%.

### Remark

Objects created with the Pic%'s way of object creation (instead of ex-nihilo creation) work fine with the message sending semantics defined above, although one remark has to be mentioned. Retake the example from section 5.3.1; it should be modified slightly:

```
make_person(name, age): {
  getname():: name;
  getage(): age;
  protect(clone()) };
```

The difference is the call to `protect()`. With the message sending semantics defined in this section, returning an unprotected clone would render visible the private methods of the object being created.

In order to have the exact same object creation mechanism as Pic% (i.e. without the `protect()` call), we could modify the message sending semantics to use `lookup_const()` (which restricts the search to the constant part of the dictionary) instead of `lookup()` when searching for message selectors:

```
picoo_send(object, 'apl):- {
  method: lookup_const(object,
    quote(application_functor(unquote(apl))));
  if(is_void(method),
    display("method not found"),
    apply(method, application_arg(unquote(apl)), object)) }
```

This is the exact same semantics of message sending of Pic%. Note that the bindings `self` and `super` are no longer defined. "Self sends" as understood in Pic% (unqualified method invocations like `msg()`) will work thanks to the default lookup mechanism of the interpreter. To allow super sends, Pic% introduces special syntax. In Pic%, the expression `.msg()` means the same as `super.msg()` would mean with our message passing semantics. We can extend xPic% with this syntax:

```
install_syntax(operand_parser, "Pic% super send", void,
  fix_mixfix_operator(".", application_parser))
```

and assign it an appropriate semantics:

```
picoo_super_send('apl):- {
  parentobj: parent(parent(parent()));
  method: lookup_const(parentobj, quote(application_functor(unquote(apl))));
  if(is_void(method),
    display("method not found"),
    apply(method, application_arg(unquote(apl)), parentobj)) }
```

Note that three `parent()` calls are required, the first to get out of the environment of `standard_super_send()`, the second one to get out of the method where the super message send is being performed (i.e. where the `.msg()` syntax was used), and a final one to get out from the current object to finally reach its parent, which is the addressee of the message.

To install Pic%'s semantics, it suffices to do:

```
assign_semantics("standard send", picoo_send);
assign_semantics("Pic% super send", picoo_super_send)
```

The "standard send" semantics, `standard_send()`, gets overwritten by `picoo_send()`. The semantics for "Pic% super send" is new.

Whether to use a single message sending syntax and semantics like we saw at the beginning of this section, or to use special syntax for special kinds of messages, is a user choice. In the author's opinion, homogeneity (the former) is the best of the options. Although the object-oriented layer of Pic% can be emulated in xPic% with the same syntax and semantics, we prefer to develop our OO layer using ex-nihilo creation of objects and the message sending mechanism presented at the beginning of this section.

### 5.3.3 Agora's views and mixins

In this section some mechanisms of the prototype-based language Agora will be installed into xPic%. The reader is referred to [4] for a good overview of what Agora is and the mechanisms we are going to present here. Consider the following example (Agora's syntax is Smalltalk-like):

```
listnode VARIABLE:
  [ next VARIABLE: null;
    getnext METHOD: SELF next;
    setnext:n METHOD: SELF next: n ]
```

It defines a `listnode` variable which refers to an ex-nihilo created object. Ex-nihilo creation of objects in Agora is achieved with the delimiters `[ ... ; ... ; ... ]`. The object has one variable `next` and two methods `getnext` and `setnext:`. Up to this point, we have enough mechanisms in xPic% to emulate the same behavior as the code above. The equivalent code is:

```
listnode: {-
  next: void;
  getnext():: next;
  setnext(n):: next:= n -}
```

But `METHOD:` is not the only way to create a method in Agora. There are three other method types we are interested in: cloning, view and mixin methods. They can be installed with the `CLONING:`, `VIEW:` and `MIXIN:` keyword messages respectively. All these method creators receive an expression, called the body, which they use in different ways:

**Cloning methods** Upon invocation of a cloning method, its body is executed in the context of a clone of the receiver instead of the context of the receiver itself. Example:

```
point VARIABLE:
  [ x VARIABLE: 0;
    y VARIABLE: 0;
    newx:coordx y:coordy CLONING:
      { SELF x: coorx;
        SELF y: coory } ]
```

A new cloning method `newx: y:` is installed in the ex-nihilo created object `point`. Upon invocation, the `x` and `y` variables in the copy are initialized with the given arguments.

**View methods** A view evaluates the body in a new object that has the receiver as parent, i.e. in an extension of the receiver. For example in

```
point VARIABLE:
  [ x VARIABLE: 0;
    y VARIABLE: 0;
    circle:r VIEW:
      { radius VARIABLE: r;
        getradius METHOD: SELF radius } ]
```

circle views can be laid down onto the point by sending the message `circle:`. A view does not destructively change the receiving object. When a `circle:` message is sent to the point, a new object is created with the point as parent. The slots `radius` and `getradius` are installed by evaluating the body of the view in the context of the extension.

**Mixin methods** In contrast with views, mixin methods destructively change the receiving object. In the following example, sending `circle:` to `point` really adds a `radius` variable and a `getradius` method to the original point. All the objects in the system that can access the point, can now access `radius` a `getradius`:

```
point VARIABLE:
  [ x VARIABLE: 0;
    y VARIABLE: 0;
    circle:r MIXIN:
      { radius VARIABLE: r;
        getradius METHOD: SELF radius; } ]
```

While views put an extra inheriting layer (i.e. a new object) around an object, mixins change the object (i.e. the very object).

In xPic%, we can define three functions that implement the semantics of Agora's `CLONING:`, `VIEW:` and `MIXIN:` methods. They will be explained one by one:

`cloning()` Let's translate the example used when `CLONING:` was explained to xPic%:

```
point: {-
  x: 0;
  y: 0;
  new(coorx, coory):: cloning(
    '{ x:= coorx;
       y:= coory }) -}
```

As can be seen, a quoted group of expressions is passed to `cloning()` for evaluation; `cloning()` will "unfreeze" the block (i.e. unquote it) and evaluate it in the context of a clone of `point`; in other words:

```
cloning(def):- {
  cloneobj: clone(parent());
  eval(unquote(def), cloneobj);
  cloneobj };
```

The `parent()` call is necessary to skip `cloning()`'s own execution environment. The environment obtained is the execution environment of `new(coorx, coory)`. Thus the references to `coorx` and `coory` in the block will be resolved without problem. Finally, the clone is returned as the result of calling `cloning()`.

`view()` Consider the example used when `VIEW:` was defined, now translated into xPic%:

```
point: {-
  x: 0;
  y: 0;
  circle(r):: view(
    '{ radius: r;
       getradius():: radius }) -}
```

Like in `cloning()`, a quoted expression is passed as argument; `view()` will unquote and evaluate it in the execution environment of `circle()`, which is already an extension of `point`:

```
view(def):- {
  viewobj: parent();
  eval(unquote(def), viewobj);
  viewobj };
```

As in the case of `cloning()`, the variable references within the block will be resolved appropriately. The extension is returned as the result of `view()`.

`mixin()` The translated example is very similar:

```
point: {-
  x: 0;
  y: 0;
  circle(r):: mixin(
    '{ radius: r;
       getradius():: radius }) -}
```

and the implementation as well:

```
mixin(def):- {
  mixinobj: parent();
  eval(unquote(def), mixinobj);
  ancestorbecome(self, mixinobj);
  mixinobj }
```

Like `view()`, unquotes and evaluates the expression `def` in the context of an extension, in this case `circle()`'s execution environment. The only difference is the call to `ancestorbecome()`, which transforms all the existing references to the receiver object `self` into references to the created extension, preserving the inheritance relationship. The function `ancestorbecome()` was presented in section 4.3.2.

A final remark to end the discussion about implementing Agora's cloning, view and mixin methods, is that the quoted expressions passed as the argument of `cloning()`, `view()` and `mixin()` do not necessarily have to be defined *in-situ* as we did in the examples above; quoted expressions can be stored in variables and passed around before they are finally used as the argument of `cloning()`, `view()` or `mixin()`.

## 5.4 Conclusion

This chapter was about assessing the expressiveness of the extension framework and giving further examples of its usage. The computational model provided by xPic% – functions and their evaluation environment playground – proved to be sufficient to implement, using base-level tools only (i.e. in a reflective way), a complete and consistent object-oriented model. After a minimal OO system was at hand, some further extensions inspired on prototype-based programming were developed. The object model of xPic% differs slightly from that of Pic%, although it would be possible to implement the latter with the exact same syntax and semantics.

One of the good points about building up a language from within the language itself, is that the user is not distracted with many implementation details. As we could see, the user-defined evaluators (that give a meaning to syntax extensions) have no more than 10 lines of code. What requires a bit more attention are the subtleties of parameter binding, function application and the relationships among evaluation environments. It is very important to have a precisely defined semantics (from chapter 4) to perfectly understand what is going on at every point of execution.

# Chapter 6

# Conclusion

The goal of this thesis was to build a "programming language laboratory" in which fundamental concepts of a multi-paradigm (functional/object-oriented) programming environment can be explored. The result is the design of a language kernel which is basically a reduction, and at the same time a generalization, of Pic%'s kernel, a language designed by professor Theo D'Hondt at VUB. In chapter 5 this kernel was validated by extending it to equal Pic%'s and then with some further constructs inspired on prototype-based programming. Section 6.1 will put together the elements of the kernel, which is the major outcome of this work. This elements have been presented in chapters 3 and 4. Some other achievements are presented in section 6.2. Section 6.3 states the limitations of our approach, and some areas that we think could be interesting given the experience of this thesis work.

## 6.1 The reflective language kernel

From the experience gathered in this work, it is concluded that to build up a full-fledged hybrid functional/prototype-based programming language the necessary elements are:

1. The minimal grammar depicted in figure 3.4.
2. The language intrinsic semantics seen in section 4.2.
3. The following parts of the xPic% meta-level interface:
   - a syntax extension tool set: `var_mixfix_operator()`, `fix_mixfix_operator()`, `grammar_term()`, `install_syntax()`, `install_syntactic_map()` and `assign_semantics()` (from section 3.7).
   - an evaluation environment playground, i.e. first-class evaluation environments and the ability to manipulate them through `this()`, `parent()` `clone()`, `extend()` and the other functions of section 4.3.2. First-class evaluation environments were fundamental to develop the prototype-based layer of xPic%.
   - a value manipulation tool set: `table()`, `function()`, `application()` `quotation()`, and the other functions of section 4.3.3. This set is particularly useful to implement syntactic maps and base-level evaluators of user-defined syntactic constructs (recall `install_syntactic_map()` and `assign_semantics()`); thus it serves as a "glue" between syntax and semantics.

The language kernel thus obtained is extensible both syntactically and semantically. As was mentioned in section 2.5, the symbiosis between the functional and prototype-based paradigms happens to be quite clean, due in part to the fact that the functional part of the kernel is not pure (like Scheme); this goes well along with the imperative nature of object-orientedness. The kernel proved to be flexible enough to allow the emulation of language constructs found in Pico, Pic%, and Agora. Although available, some mechanisms were not explored, like the function `become()` inspired in actor languages, or dynamic inheritance through `setparent()`.

The so-called "little languages" or domain-specific languages were proposed as an application area of the language kernel, which, by means of a highly reflective architecture, allows its configuration to suit particular domains. A second application area is the exploration of the language design space that results from the symbiosis of the functional and prototype-based paradigms. Such exploration can be performed within the language itself, i.e. in a reflective way, thus allowing the user to stay at one level, the base-level; it is not necessary to know the internals of the language. The code needed in the base level for performing extensions is usually very short, as was shown in chapter 5. Yet another application area, inherited from Pic%, is education: the kernel helps in understanding fundamental elements of programming languages; notably, a sharp division is established between the language "skin" and its "guts", i.e. between its syntax and semantics. Students can explore both of them at will, and understand their interaction.

## 6.2 Other achievements

The following are non-expected outcomes of the thesis:

**Quoted parameters** Functions can have quoted parameters, take for instance:

```
quotation('exp): exp
```

As was explained in section 4.2.2, the semantics is that, on invocation, the formal parameter gets unquoted, the actual argument gets quoted, and the binding process is invoked again with this two new values. For example, if `quotation(a)` is applied, then the parameter `exp` will be bound to the value `'a` and this quoted reference is the result of the application. Another example is available in section 4.4 (the conclusions of chapter 4). The idea of quoted parameters is new up to the author's knowledge.

From this parameter binding mechanism emerges the idea of generalizing parameter binding with user-defined semantics for user-defined constructs, a line of future work which is exposed in section 6.3.5.

**Generalization of the notion of binding** In xPic%, there is no difference between function parameter binding and the process of defining variables (with the `:` syntax or using the function `define()`): both are performed using the simple value binding semantics of section 4.2.2. As an example, if a function `f(x,y): ...` is defined, then upon the invocation `f("a","b")`, the binding `[x,y]` $\xleftarrow{\text{define}}$ `["a","b"]` is performed in the function's environment. This same process occurs if the definition `[x,y]: ["a","b"]` is evaluated. An application of this sample case (binding of tables) is to easily handle multiple values returned by a function in a table.

The binding process has a recursive nature: all the `bind_*` procedures "bounce" the responsibility back to `bind_param()` excepting `bind_ref_param()` which finally invokes

the association `assoc` (recall section 4.2.2). Due to recursivity, a kind of *pattern-matching* binding semantics is obtained; for instance, the following expression is valid in xPic%: `[x,[f(t),'r]]: [1,[t+1,h*2]]`; it leaves the variables `x`, `f` and `r` bound in the current environment to a number, a closure and a quoted expression respectively. Note that the table to the left could be the parameter list of a function, i.e. `f(x,[f(t),'r]): ...` If it is applied with the arguments of the previous example: `f(1,[t+1,h*2])`, the same binding process would take place, and the variables `x`, `f` and `r` would be available in the environment of evaluation of the function's body. This general notion of binding is not present in Pic%, thus parameter definitions like in `f(x,[f(t),'r])` are not possible.

## 6.3 Limitations and future work

The areas in which future work is devised have been split up into sections. The first ones are more "fundamental" and the final section (6.3.7) presents work which is not far from being achieved starting from the base presented in this document.

### 6.3.1 Reasoning about the grammar

This thesis made no emphasis in finding out grammar properties such as ambiguity or left recursion (as the constructed parsers are recursive-descent, left-recursion in the grammar is an issue). Although the user can extend the grammar in a controlled way, i.e. at certain spots and with a set of tools which is not as general as combinators, it is still possible to introduce left-recursive or ambiguous constructs. The user should receive, tough, as much assistance as possible from the system during the language extension processes. The following are some areas of future work in this direction:

**Grammar navigation** No tools were defined at the meta-level to navigate the grammar structure, for example to iterate over its rules and each disjunction option. Under the framework presented, navigability is not difficult to achieve, since the grammar elements are already reified at the meta-level; it only remains to define accessor methods for each reifier object. Navigation is a requisite for grammar reasoning.

**Usage of techniques from language theory** Once the grammar is navigable, the adoption of existing techniques from classical language theory to detect and manage grammar properties would help the user in appropriately extending it.

**Seizing on meta-information** As there is plenty of information available at the meta-level about the grammar, including human-readable names for every token type, grammar terminal and non-terminal, it would be interesting to seize on that information, for instance, to give good error reports.

### 6.3.2 Language safety, constrained extensibility

Reasoning about the grammar is to provide safety at the syntactic level. In the current interpreter of xPic%, it is possible to override the fundamental semantics of the language, like function application (which renders the language unusable). This could be improved by restricting language extension to a set of constraints.

### 6.3.3 Other parsing techniques

One challenging area of work is to study reflective grammar extensibility for grammar types other than the LL(n) family, for instance LR(n) grammars. Normally the parsers for this kind of grammars consist of a stack automaton; the idea would be then to modify the transition tables of the automaton as the user performs syntax extensions at run-time.

### 6.3.4 Localized extensions

Usually a big system is split up into parts, each one dedicated to a specific purpose. Within each part some extensions to the language may be handy to deal with the specific domain of the part, i.e. domain-specific sub-languages can be useful. In the framework presented in this work, extensions have a global scope: semantical and syntactical changes globally affect the interpreter. One area of future research would be to study the possibility of having *localized* extensions, i.e. extensions which have an effect in a certain environment, but not in the whole interpreter.

### 6.3.5 User-defined value binding semantics

One of the core elements in the language semantics is value binding: how to associate two given expressions in a dictionary. The definition of such semantics for xPic% was provided in section 4.2.2; it is inspired on Pico's. The addition of quoting in the core (intrinsic) semantics of the language inspired one of the contributions of this work: to define a semantics for binding of quoted parameters. It would be possible, as was done for quoting, to define binding semantics for new value types introduced in the language. Given that the set of value types in xPic% is extensible (recall section 4.2.1), the interesting part would be to design a mechanism in which *user-defined binding semantics* are assigned, from the base-level, to user-defined value types. For instance, one application of such a possibility is to introduce type checks in function parameters. Recall from section 5.2.2 the introduction of the following syntax in the language:

```
install_syntax(expression_parser, "standard definition", operation_parser,
  fix_mixfix_operator(operation_parser, ":", expression_parser)),
```

We could think of a function which uses <standard definition> in its parameters:

```
func(a: is_number, b: lambda(x): or(is_dictionary(x), is_void(x))):
  { ... do something with a and b ... }
```

With the current version of xPic%, such definition of `func()` is in fact accepted by the interpreter, but upon invocation, say with `func(1,2)`, the interpreter replies:

```
evaluator: cannot bind a value of type <standard definition>
evaluator: cannot bind a value of type <standard definition>
```

The intrinsic semantics of the language defines value binding for just a few value types, and such semantics is not extensible. The future work is to design a mechanism to allow the extension from the base-level of binding semantics, in which users can provide their own `assoc(target, value)` functions (recall section 4.2.2) to perform binding. In the example above, `assoc()` could be defined such that it would invoke `is_number(1)` and `lambda(2)` upon the invocation `func(1,2)` to verify that the arguments have correct types (and it would fail since the second doesn't). Up to author's knowledge, a configurable value binding semantics has never been explored.

### 6.3.6 Pure functional language semantics

Like Scheme, it is possible to use xPic% (or Pic% as well) in a purely-functional way. This dissertation didn't explore the consequences of a symbiosis between prototype-based programming and a *pure* functional paradigm.

### 6.3.7 Minor work

The following work is either easy to implement or does not involve very fundamental issues:

**Absorption** The *nativedict* value type from section 4.2.1 is a good point to start thinking about the incorporation of absorption mechanisms into xPic%, as it is done for example in Agora [4].

**Extensible scanner** Study the possibility of making the scanner extensible (as was suggested in section 3.2.2).

**Ability to shrink syntax** Explore the possibility of not only adding but removing extensions, for instance to allow option removal in disjunctions (recall section 3.4.4).

**Arbitrary levels of precedence** Having arbitrary levels of precedence in operators, an idea of which a rough draft was given in section 3.6.1. If this is achieved, the basic xPic% grammar would be as simple as figure 3.4 and the extension framework would be more flexible.

**Improving parsing performance** In section 3.4.5 it was mentioned that intensively using "raw" parser combinators makes syntax analysis inefficient, as was the case in the first versions of xPic%. To solve the problem, a caching technique was developed, which is described in appendix A (particularly important are parse caches, section A.2). Although the technique significantly improves performance, it could be better; this is one possible direction of future work.

## 6.4 Finally...

Before entering the EMOOSE, the author didn't have a clue about the existence of prototype-based programming. As the alternatives never were presented to him, he didn't even think about separating classes and OOP. Prototype-based programming has enriched a lot his view of object-oriented programming, which is now regarded from a different perspective. The overwhelming success of class-based languages like C++ or Java has turned their particular flavor of object-oriented programming into the most widely accepted paradigm in use today. Some fundamental ideas are lacking attention or are loosing it, as universities increasingly accommodate their curricula to supply the demand for professionals who dominate the technologies currently in use. A "language lab" like xPic% looks towards putting alternative paradigms on a sound footing.

# Appendix A

# Improving performance

Techniques to improve the performance of both lexical and syntax analyzers are commented in this section. As was mentioned in section 3.4.5, using parser combinators hampers performance. In fact the xPic% interpreter was performing so poorly that it was useless after some grammar extensions. The techniques presented here solved the problem making the interpreter usable again.

## A.1 Token stream caches

A stream can be marked and rolled back later. How can this be implemented in the case of a token stream? One possibility is simply to rollback the underlying character input stream that is passed to the scanner's `next()` method (recall section 3.2.2). As the scanner is immutable, rescanning this stream would yield the exact same sequence of tokens that was found before, i.e. the same work would be done probably many times. In the implementation of the token stream, a 'cache' of tokens is maintained to solve this problem; it is in fact very simple and efficient thanks to the fact that Pic% is garbage-collected. As an image speaks more than a thousand words, an excerpt of code will in this case transmit the idea better than a general description:

```
make_token_istream(scanner, char_istream): {
  >-- make_abstract_istream();
  pointer: [ scanner.next(char_istream), void ];
   ...
  mark()::
      pointer;

  rollback(mark)::
    pointer:= mark;

  peek()::
    pointer[1];

  skip():: {
    is_void(pointer[2]) =>
      (pointer[2]:= [ scanner.next(char_istream), void ]);
    pointer:= pointer[2] };

  next()::
    { item: peek(); skip(); item } }
```

This is, a new token will be asked to the scanner (in `skip()`) only in case the cache is empty. This cache is implemented as a simply linked list. Each newly retrieved token is appended to the end of the list. If the user never uses `rollback()` to go back in the stream, the pointer will be always at the end of the list. When `mark()` is used, the returned value is in fact a pointer to the current element of the list. The garbage collector will not reclaim the elements from that point on because all of them will be referenced (starting by the mark and following the links of the list), so the cache will start to grow. If `rollback()` is called later to restore a mark (i.e. a pointer to a certain point of the list), `skip()` will start using cached tokens instead of asking the scanner for them. When marks are dropped, parts of the cache (from the head of the list to the element before the first mark that is still being retained) become candidates for garbage-collection.

This caching mechanism never uses more memory than necessary and avoids a lot of book-keeping thanks to garbage collection; algorithms without it could be much more complicated. The aim of this small discussion was, apart from presenting the technique, to give a good example in favor of garbage-collected languages.

## A.2 Parse caches

There is one combinator that was not introduced in section 3.4.1,

comblib.cache(parser)

It wraps a given parser so that all its methods are still available in the wrapped object. To achieve this, the newly created cache object dynamically inherits from the given parser, hence every method not found in the cache object is looked up automatically in the wrapped parser. This design follows the Decorator pattern of [29]. The only behavior modification introduced by the cache object is in the method `parse()` (recall section 3.3.1). Each time it is invoked with a given token stream as argument, it will ask an object, called the `cache_manager`, to get a cached AST for the given stream, if available. If it is found, the cached version is returned immediately, saving all the work of parsing the input at the current position. If there is no cached ASTs, the overridden `parse()` is invoked to parse the input, and, if the parsing succeeds, the returned AST is added to the cache. The `cache_manager` is in fact not complicated,

```
make_cache_manager(): {
   ...
  add(stream, start_mark, ast):: ...
  get(stream):: ...
  clone() };
```

**add(stream, start_mark, ast)** Cached ASTs are maintained for every scan position of the stream (e.g. the first token in the stream has position 1, the second token is at position 2, etc.). Apart from saving the AST in the cache at the position specified by `start_mark`, the region of the stream that yielded such an AST is stored (i.e. the start and end positions, just before parsing and after parsing has been performed).

**get(stream)** It will lookup the cache to see if there is a stored AST for the current position of the `stream`. If this is the case, the cached AST is returned and the position of the stream is advanced as if the AST would have been parsed (this is the end position that `add()` stored together with the AST).

Thanks to dynamic inheritance, the cache is transparent and can be installed in any parser object, including those like `comblib.any` (recall section 3.4.4) that define other methods apart from `parse()`.

Too much caching degrades performance. By means of tests it was found that it suffices to install only one cache,

```
operation_parser: comblib.cache(comparison_parser)
```

This is the reification of the penultimate rule of figure 3.3. It is a critical point of the grammar when syntactic constructs that use `operation_parser` are installed, like the standard definition, declaration and assignment operators `:` `::` and `:=` of Pico. Even though caching this spot of the grammar significatively improved performance, the strategy to place caches (i.e. which parsers to wrap) should be studied more thoroughly. This could be an area for future work, as well as the development of better caching algorithms.

# Bibliography


[1] J. Bentley, "Programming pearls: Little Languages," *Communications of the ACM*, vol. 29, no. 8, pp. 711–721, 1986.

[2] D. Ungar and R. B. Smith, "Self: The Power of Simplicity," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (N. Meyrowitz, ed.), vol. 22, (New York, NY), pp. 227–242, ACM Press, 1987.

[3] A. Taivalsaari, *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. PhD thesis, University of Jyväskylä, 1993.

[4] W. D. Meuter, "Agora: The Scheme of Object-Orientation, or, The Simplest MOP in the World," in *Prototype-Based Programming: Concepts, Languages and Applications* (J. Noble, A. Taivalsaari, and I. Moore, eds.), ch. 12, pp. 247–272, Springer-Verlag, 1999.

[5] B. A. Myers, R. McDaniel, R. Miller, B. V. Zanden, D. Giuse, D. Kosbie, and A. Mickish, "The Prototype-Instance Object Systems in Amulet and Garnet," in *Prototype-Based Programming: Concepts, Languages and Applications* (J. Noble, A. Taivalsaari, and I. Moore, eds.), ch. 7, pp. 141–176, Springer-Verlag, 1999.

[6] P. Mulet and P. Cointe, "Definition of a Reflective Kernel for a Prototype-Based Language," in *Proceedings of the 1st JSSST International Symposium on Object Technologies for Advanced Software* (S. Nishio and A. Yonezawa, eds.), vol. 742, pp. 128–144, Springer-Velag, 1993.

[7] G. Blaschek, "Omega: Statically Typed Prototypes," in *Prototype-Based Programming: Concepts, Languages and Applications* (J. Noble, A. Taivalsaari, and I. Moore, eds.), ch. 8, pp. 177–196, Springer-Verlag, 1999.

[8] L. Cardelli, "A Language with Distributed Scope," *Computing Systems*, vol. 8, pp. 27–59, January 1995.

[9] W. Smith, "NewtonScript: Prototypes on the Palm," in *Prototype-Based Programming: Concepts, Languages and Applications* (J. Noble, A. Taivalsaari, and I. Moore, eds.), ch. 6, pp. 109–139, Springer-Verlag, 1999.

[10] P. Steyaert and W. D. Meuter, "A Marriage of Class- and Object-Based Inheritance Without Unwanted Children," Tech. Rep. vub-prog-tr-95-02, Programming Technology Lab, Vrije Universiteit Brussel, 1995.

[11] W. R. LaLonde, D. A. Thomas, and J. R. Pugh, "An exemplar based Smalltalk," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 322–330, ACM Press, 1986.

[12] C. Dony, J. Malenfant, and D. Bardou, "Classifying Prototype-based Programming Languages," in *Prototype-based programming: Concepts, Languages and Applications* (J. Noble, A. Taivalsaari, and I. Moore, eds.), ch. 2, pp. 17–45, Springer-Verlag, 1999.

[13] Pico home site – Programming Technology Lab, Vrije Universiteit Brussel. (online) <http://pico.vub.ac.be> [visited: august 2002].

[14] T. D'Hondt and I. Michiels, "Combating the paucity of paradigms in current OOP teaching." Programming Technology Lab, Vrije Universiteit Brussel, 2000.

[15] R. B. Smith and D. Ungar, "Programming as an Experience: The Inspiration for Self," in *Prototype-Based Programming: Concepts, Languages and Applications* (J. Noble, A. Taivalsaari, and I. Moore, eds.), ch. 5, pp. 77–107, Springer-Verlag, 1999.

[16] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*. Electrical Engineering and Computer Science Series, Cambridge, MA: The MIT Press, 2nd ed., 1996.

[17] J. Malenfant, C. Dony, and P. Cointe, "Behavioral Reflection in a Prototype-Based Language," in *Proceedings of International Workshop on Reflection and Meta-Level Architectures* (A. Yonezawa and B. Smith, eds.), pp. 143–153, 1992.

[18] K. D. Volder, *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[19] P. Maes, "Concepts and Experiments in Computational Reflection," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 22, pp. 147–155, ACM Press, December 1987.

[20] F.-N. Demers, J. Malenfant, and M. Jacques, "A Tutorial on Behavioral Reflection and its Implementation," in *Proceedings of the 1st International Conference on Computational Reflection* (G. Kiczales, ed.), pp. 1–20, April 1996.

[21] B. C. Smith, "Reflection and Semantics in a Procedural Language," Tech. Rep. 272, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[22] D. G. Bobrow, R. P. Gabriel, and J. L. White, "CLOS in Context – The Shape of the Design Space," in *Object-Oriented Programming* (A. Paepcke, ed.), ch. 2, MIT Press, 1993.

[23] L. A. Stein, H. Lieberman, and D. Ungar, "A Shared View of Sharing: The Treaty of Orlando," in *Object-Oriented Concepts, Databases and Applications* (W. Kim and F. Lochovsky, eds.), pp. 31–48, Reading (MA), USA: ACM Press/Addison-Wesley, 1989.

[24] O.-J. Dahl and K. Nygaard, "SIMULA — an ALGOL-Based Simulation Language," *Communications of the ACM*, vol. 9, no. 9, pp. 671–678, 1966.

[25] M. Abadi and L. Cardelli, *A Theory of Objects*. New York, NY: Springer-Verlag, 1996.

[26] A. Taivalsaari, "Classes vs. Prototypes: Some Philosophical and Historical Observations," in *Prototype-Based Programming: Concepts, Languages and Applications* (J. Noble, A. Taivalsaari, and I. Moore, eds.), ch. 1, pp. 3–16, Springer-Verlag, 1999.

[27] D. Rayside and G. T. Campbell, "An Aristotelian Understanding of Object-Oriented Programming," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 337–353, ACM Press, 2000.

[28] H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (N. Meyrowitz, ed.), vol. 21, (New York, NY), pp. 214–223, ACM Press, 1986.

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series, Reading, MA: Addison-Wesley, 1995.

[30] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle, "Organizing Programs Without Classes," *Lisp and Symbolic Computation*, vol. 4, no. 3, pp. 223–242, 1991.

[31] "Prototype-based languages: from a new taxonomy to constructive proposals and their validation."

[32] J. Noble, A. Taivalsaari, and I. Moore, eds., *Prototype-Based Programming: Concepts, Languages and Applications.* Springer-Verlag, 1999.

[33] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools.* Computer Science series, Addison-Wesley, 1986.

[34] G. Hutton and E. Meijer, "Monadic Parser Combinators," *Journal of Functional Programming*, vol. 8, pp. 437–444, July 1998.

[35] Borg home site – Programming Technology Lab, Vrije Universiteit Brussel. (online) <http://borg.rave.org> [visited: august 2002].