Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Notación Imperativa para Programación Lógica

Imperative Notation for Logic Programming

Autor: Paula Corral Rebollar
Tutor: José Francisco Morales Caballero
Cotutor: Manuel de Hermenegildo Salinas

Madrid, Junio 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Notación Imperativa para Programación Lógica

Imperative Notation for Logic Programming

Junio 2025

Autor: Paula Corral Rebollar

Tutor: José Francisco Morales Caballero
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Cotutor: Manuel de Hermenegildo Salinas
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

Las variables de estado y los bucles, entre otras características, se asocian típicamente con los lenguajes de programación imperativos y no se encuentran de forma nativa en los paradigmas de programación declarativa. Por ejemplo, muchos lenguajes basados en lógica dependen exclusivamente de la recursión para expresar la iteración. Sin embargo, las características imperativas pueden simplificar la implementación de ciertos algoritmos que, de otro modo, serían más engorrosos de expresar mediante recursión. Como resultado, algunos sistemas de programación lógica han incorporado diversos constructos imperativos.

`Prolog` es uno de los principales representantes del paradigma de programación lógica. Puede soportar notación funcional mediante un enfoque sintáctico, `FSyntax`, que aprovecha la sintaxis del lenguaje y las capacidades de expansión de términos. `Hiord` es otro enfoque sintáctico que permite la programación de orden superior en `Prolog`. Basándose en `call/n`, introduce características adicionales como los predicados anónimos. Tanto `FSyntax` como `Hiord` se utilizan ampliamente en el sistema `Ciao Prolog`, el cual se basa en la familia de `Prolog`, aunque es también un sistema de programación multiparadigma. `Ciao Prolog` dispone de un avanzado sistema de módulos que permite implementar fácilmente extensiones sintácticas y semánticas potentes.

En este trabajo proponemos un conjunto de constructores de estilo imperativo que extienden `FSyntax` y `Hiord`. Estas extensiones están diseñadas para integrarse de forma natural con la notación funcional básica, las facilidades de orden superior y otras extensiones del lenguaje como las restricciones. A diferencia de propuestas anteriores, nuestro enfoque ofrece tanto un conjunto de primitivas como un mecanismo de alto nivel que, en conjunto, permite a los usuarios extender fácilmente el lenguaje con características como notación de arrays, variables de estado, bucles, etc.

Ofrecemos una descripción detallada de cómo la compilación de módulos procesa estos constructores: desde la definición de la sintaxis hasta su traducción completa en predicados planos de `Prolog` y llamadas estáticas. Además, describimos el concepto de unificación unidireccional, su representación equivalente en `Prolog` y su implementación en el sistema `Ciao`.

A partir de estas extensiones, desarrollamos un paquete adicional que proporciona notación de acceso indexado a arrays. Este paquete consta de una interfaz y cuatro implementaciones diferentes, y utiliza variables de estado para realizar

la modificación del elemento n-ésimo de un array.

Evaluamos nuestro enfoque traduciendo problemas del *Proyecto Euler* de `Picat` a `Ciao`. Verificamos su corrección comprobando que todas las pruebas se superan, y medimos los tiempos de ejecución para evaluar el rendimiento. Estos resultados permiten comparaciones con implementaciones en otros lenguajes. Nuestros experimentos muestran que las extensiones propuestas ofrecen un rendimiento competitivo.

Finalmente, creemos que incorporar características sintácticas propias de la programación imperativa puede ayudar mucho al programador y, potencialmente, favorecer una adopción más amplia de los lenguajes declarativos. Además, dependiendo de su semántica, los constructores imperativos como los bucles pueden ayudar a los analizadores estáticos al proporcionar información implícita, como determinismo, no-fallo, modos, tipos y cotas del número de iteraciones de los bucles, lo cual resulta valioso para el análisis de coste y complejidad.

Abstract

State variables and loops, among other features, are typically associated with imperative programming languages and are not natively found in declarative programming paradigms. For example, many logic-based languages rely solely on recursion to express iteration. However, imperative features can simplify the implementation of certain algorithms that would otherwise be more cumbersome to express recursively. As a result, some logic programming systems have incorporated various imperative constructs.

`Prolog` is one of the major representatives of the logic programming paradigm. It can support functional notation through a syntactic approach, `FSyntax`, which leverages the language's syntax and term expansion capabilities. `Hiord` is another syntactic approach to supporting higher-order programming in `Prolog`. Building on `call/n`, it introduces additional features such as anonymous predicates. Both `FSyntax` and `Hiord` are extensively used in the `Ciao Prolog` system. `Ciao Prolog` is a multi-paradigm programming system based on the `Prolog` family. It features an advanced module system that enables easily implementing powerful syntactic and semantic extensions.

In this work, we propose a set of imperative-style constructs that extend `FSyntax` and `Hiord`. These extensions are designed to integrate smoothly with the basic functional notation, higher-order facilities, and other language extensions such as constraints. Unlike previous proposals, our approach offers both a collection of primitives and a high-level mechanism that together enable users to easily extend the language with features such as array notation, state variables, and loop constructs, etc.

We provide a detailed account of how the compilation module processes these constructs—from syntax definition to their full translation into flat `Prolog` predicates and static calls. Additionally, we describe the concept of one-sided unification, its equivalent representation in `Prolog`, and its implementation in the `Ciao` system.

Building on these extensions, we have developed an additional package that provides indexed array access notation. This package consists of an interface and four different implementations and uses state variables to support modification of the nth element of an array.

We evaluate our approach by translating *Euler Project* problems from `Picat` to `Ciao`. We verify correctness by checking that all tests pass, and we measure

execution times to assess performance. These benchmarks allow comparison with implementations in other languages. Our experimental results show that the proposed extensions offer competitive performance.

Finally, we argue that incorporating syntactic features from imperative programming can improve programmer convenience and potentially support wider adoption of declarative languages. Moreover, depending on their semantics, imperative constructs like loops can assist static analyzers by providing implicit properties such as determinism, non-failure, modes, types, and bounds on loop iterations, information that is valuable for cost and complexity analyses.

Contents

1 Introduction **1**
1.1 Motivation . . . 2
1.2 Objectives . . . 4
1.3 Structure . . . 4

2 Background **5**
2.1 The `Ciao Prolog` System . . . 5
2.1.1 The Ciao Module System . . . 6
2.1.2 Functional Approach in `Ciao Prolog` . . . 7
2.1.3 Higher order . . . 9
2.1.4 Notation Predicate . . . 10
2.2 Imperative Approaches in Logic Programming . . . 11
2.2.1 Single-sided Unification . . . 11
2.2.2 State Variables . . . 12
2.2.3 Loops . . . 13
2.2.4 Array-Like Index Notation . . . 14

3 Single-sided Unification **15**
3.1 Pattern-Matching Rules . . . 15
3.2 Syntax . . . 15
3.3 Semantics . . . 15
3.4 Compilation Rules . . . 17
3.5 Compatibility With Other Modules . . . 19

4 State Variables **21**
4.1 Syntax . . . 21
4.2 Compilation Rules . . . 22
4.2.1 First Phase . . . 22
4.2.2 Second Phase . . . 24

5 Implementation of Array Index Notation **29**
5.1 Design of the Package . . . 29
5.2 Syntax . . . 30
5.3 Semantics . . . 30
5.3.1 Mutable Arrays . . . 31
5.3.2 Fixed Arrays . . . 32

5.3.3 Extendable Arrays 32
5.3.4 Lists 33
5.4 Results of the Array Implementations 34

6 Loops **35**
6.1 Syntax 35
6.2 Loop Definition 36
6.2.1 While Loop 37
6.2.2 For Loop 37
6.3 Compilation Rules 39
6.3.1 First Phase 39
6.3.2 Second Phase 41
6.3.3 Third Phase 45
6.4 Other Implemented Loop Types 50

7 Experimental Results **51**

8 Impact Analysis **55**

9 Conclusions and Future Work **57**

Bibliography **59**

Appendices **65**

A *Sieve of Eratosthenes*: Implementations in (Ciao) Prolog **65**

B Arrays Testing Code **67**

C Interface of Arrays Package **69**

D Plagiarism Results **71**

Chapter 1

Introduction

Declarative programming allows for the efficient development of complex software systems while also helping in achieving correctness and safety. *Logic programming* is a declarative programming paradigm based on formal logic. It allows the definition of sets of facts and rules (i.e., logic programs) that are used to find answers or solutions to queries posed by the user. The logic program expresses *what* the problem is (i.e., the *logic*), independently of *how* to solve it (i.e., the *control*), as the latter is handled to a great extent by the system. The main representative of this paradigm is the `Prolog` language. To find all possible solutions, `Prolog` relies on a predefined search mechanism based on unification, SLD-resolution inference, and backtracking.

In contrast, a program in the *imperative programming* paradigm expresses essentially the *control*, through sequences of instructions or statements that describe *how* the program should execute to achieve specific results. These instructions modify the state of the program, and include control flow structures such as loops, which some users find more intuitive or easier to understand [1]. `Python` and `Java` are examples of widely used programming languages that follow this paradigm.

There has been interest in making `Prolog` multi-paradigm and some steps have taken in this direction; some examples are `Ciao` [2] and `Janus` [3], or, in the more general area of logic-based languages, `Picat` [4, 5]. The motivation is that, for certain problems, it can sometimes be easier or more convenient for the programmer to use syntactic constructs and features borrowed from other programming paradigms. For example, `FSyntax` [6] is a syntactic approach to supporting functional notation in `Prolog` systems which is based on the use of the syntax and term expansion facilities of the language. `Hiord` [7] is also a syntactic approach to supporting higher-order in `Prolog`, building on `call/n`. Together they bring in the syntactic convenience of functional and higher-order constructs and both are used profusely by the `Ciao Prolog` system [2].

Mar 2025	Mar 2024	Change	Programming Language	Ratings	Change
1	1		Python	23.85%	+8.22%
2	3	^	C++	11.08%	+0.37%
3	4	^	Java	10.36%	+1.41%
4	2	v	C	9.53%	-1.64%
5	5		C#	4.87%	-2.67%
6	6		JavaScript	3.46%	+0.08%
7	8	^	Go	2.78%	+1.22%
8	7	v	SQL	2.57%	+0.65%
9	10	^	Visual Basic	2.52%	+1.09%
10	15	^^	Delphi/Object Pascal	2.15%	+0.94%

Figure 1.1: TIOBE index ranking [8].

1.1 Motivation

The TIOBE index serves as an indicator of programming language popularity [8]. As of March 2025, Prolog ranks as the twenty-second most popular programming language. This puts it above the other declarative languages such as Lisp, Scala, Haskell, etc. However, as shown in Figure 1.1, essentially all the top languages are imperative in nature. This suggests that most beginners are introduced to programming through imperative languages, familiarizing them only with imperative paradigms. Consequently, learning `Prolog` can be challenging, as its lack of basic imperative structures, such as loops, might feel unfamiliar.

Also, descriptions of algorithms in papers and textbooks are most often presented using imperative pseudocode, containing loops and other imperative constructs. Such algorithms can be implemented in logic languages using recursion, frequently rather elegantly, but this encoding can also in some cases be awkward and/or blur the correspondence with the original algorithm description. Consider, for example, the *Sieve of Eratosthenes* for computing primes. It is a classic example in both functional and logic programming, where it is typically presented in an elegant recursive form. Figure A.1 in Appendix A shows a version of this classic encoding, using the `FSyntax` package in Ciao Prolog.[1] The program is relatively simple, and illustrates the elegance of declarative programming. Unfortunately, this classic encoding does *not* implement the algorithm by Eratosthenes, but rather a naïve algorithm known as *trial division* (see O'Neill [9]).

If we want to implement the actual algorithm by Eratosthenes, we can perhaps turn to the Wikipedia[10], which provides a description of the algorithm in imperative pseudocode, shown in Figure 1.2.

This algorithm can be coded in standard Prolog (see Figure A.2 in Appendix A), but the result is not very compact and can feel a bit awkward. A version coded

[1] We show an eager version for simplicity; the lazy version is essentially as the one used in lazy functional languages, see the code in Figure A.3 (in Appendix A) and [6].

```
input: integer N > 1
output: list of prime numbers from 2 to N

A = array with index from 2 to N with all the values set to True

for i = 2,3, ..., sqrt(N) do
    if A[i] == True
        for j = i*i, i*i + i, i*i + 2i, ... below or equal to N do
            A[j] = False

return list with all k where A[k] is True
```

Figure 1.2: *Sieve of Eratosthenes* algorithm in pseudocode (based on Wikipedia [10]).

```
:- module(_, _, [functional]).
:- use_module(library(logarrays)).

primes(N) := Res :-
    complete_sieve(2,floor(sqrt(N)),N,~new_array,CompleteSieve),
    take_primes(2,N,CompleteSieve,Res).

complete_sieve(Curr,To,N,Sieve) :=
    ( Curr > To            ? Sieve
    | aref(Curr,Sieve,_El) ? ~complete_sieve(Curr+1,To,N,Sieve)
    | ~complete_sieve(Curr+1,To,N,~set_multiples(Curr**2,Curr,N,Sieve)) ).

set_multiples(Curr,Step,To,Sieve) :=
    ( Curr > To ?  Sieve
    | aset(Curr,Sieve,0,Sieve1) ? ~set_multiples(Curr+Step,Step,To,Sieve1) ).

take_primes(Curr,N,Sieve) :=
    ( Curr > N             ? []
    | aref(Curr,Sieve,_El) ? ~take_primes(Curr+1,N,Sieve)
    | [Curr| ~take_primes(Curr+1,N,Sieve)] ).
```

Figure 1.3: *Sieve of Eratosthenes* algorithm in Prolog + `FSyntax`.

again using `FSyntax` is shown in Figure 1.3. This version is more compact and elegant, but it still suffers from some of the previously mentioned issues, and in particular the correspondence with the original pseudocode is not obvious.[2]

The Python implementation of the algorithm, shown in Figure 1.4, is relatively simple and close to the algorithm represented by the pseudocode, making it easy to see their correspondence. Some of the simplicity of the python version comes from the characteristics also shared by Prolog, such as being a dynamic language, the correspondence issue brings us back to the idea that there are cases where incorporating features from imperative programming can bring programmer convenience. In general, this can also potentially also contributes to the wider adoption of declarative languages.

[2]In addition, coding the algorithm in a (lazy) functional language is not trivial, as shown also by O'Neill [9], and suffers from the same drawbacks.

```python
import math

def primes(n):
    # Initialize the array with n+1 values to access until index n
    a = [True] * (n+1)

    # The loop goes until the square root of n
    to = math.floor(math.sqrt(n))

    # We cross out the non primes
    for i in range(2,to+1):
        if a[i]: # If a[i] is prime we cross out its multiples
            for j in range(i*i,n+1,i):
                a[j] = False

    # We take the primes from the sieve and put them into a list
    result = []
    for k in range(2,n+1):
        if a[k]:
            result.append(k)

    return result
```

Figure 1.4: The Sieve of Eratosthenes algorithm, in Python.

1.2 Objectives

Motivated by this, this project aims to explore and develop imperative extensions for Ciao-Prolog, and evaluate them through using and benchmarkking them. A list of more specific objectives are the following:

1. Familiarize ourselves with logic programming systems, and specifically Ciao-Prolog.
2. Explore single-sided unification, state variables, array index notation, and loops for those systems.
3. Study and develop an implementation of these imperative constructs in Ciao-Prolog.
4. Test and benchmark these extensions through a set of examples.

1.3 Structure

The rest of the project is structured as follows: Chapter 2 provides background and an overview of the state of the art, focusing on the Ciao-Prolog system and approaches to imperative constructs in logic programming. Then Chapter 3 covers single-sided unification, while Chapter 4 focuses on state variables and Chapter 5 introduces array index notation. Continuing, Chapter 6 discusses loops. Chapter 7 contains the study of practical examples and the experimental results, while Chapter 8 gives an impact analysis. Finally, Chapter 9 ends the report with conclusions and future research directions. Each of the chapters devoted to the extensions includes both a description of the extension and an account of the compilation process for that extension.

Chapter 2

Background

A number of multiparadigm programming languages exist that integrate features from logic-, functional-, constraint-, or imperative programming.

Some of these are entirely new languages rather than extensions of `Prolog`, in contrast to our approach. For example, `Picat`, in addition to state variables, also incorporates two types of loops and array access notation [5]. `MiniZinc` [11, 12] is a popular modeling language for combinatorial problems. It provides a loop construct which can be used to place constraints that depend on the loop bounds in a natural and regular way.

Returning to approaches based on extending `Prolog` with new syntactic constructs, as in our proposal, apart from the `Hiord` and `fsyntax` extensions in `Ciao Prolog`, for instance `ECLiPSe` introduces *logical loops* as a language extension [13], and array notation, although it does not include state variable support. SICStus [14] has traditionally supported `setarg/3` and later incorporated *mutables*. XSB [15] also supports `setarg/3` and has an array facility. SWI [16] has global variables (also supported by Yap [17]) and some other extensions. François Fages has developed a mathematical modeling library for `Prolog` [18], inspired by `MiniZinc`. This library brings constraint modeling capabilities to `Prolog` with a mathematical focus. It includes the forall loop notation, which, like `MiniZinc`, is limited to verifying that a constraint holds across all iterations and does not support state variables.

We provide more detailed background in the following, organized by different features, topics, or systems.

2.1 The `Ciao Prolog` System

`Ciao` [2] is a multi-paradigm, general-purpose programming language in the `Prolog` family. It allows programming with a fully declarative syntax but also supports extensions that provide different syntaxes and semantics, such as functional syntax or higher-order logic programming with predicate abstractions. It also provides constraint extensions that allow setting up constraints

over various domains. These extensions are based on its module system, where modules can coexist and allow different levels of analysis using the Ciao Program Preprocessor, CiaoPP [19].

2.1.1 The Ciao Module System

Modularity is a basic feature in programming which consists on dividing programs or systems into several independent modules, each acting as a functional unit. The Ciao system has been designed to be modular, having a single core and multiple syntactic and semantic extensions implemented as modules. Concretely these extensions can contain operators definitions to define a syntax, compilation options or rules among others.

One of the main objectives of this modular system design is enabling modular compilation, such that only a modified module has to be recompiled and not the rest of related modules. This constitutes an important feature to increment efficiency and take advantage of the benefits of CiaoPP. Another key objective is the ability to locally extend or restrict features or syntax, the point is to be able to allow specific features in a specific module without automatically propagating them in other modules.

In Ciao [20], modules are typically contained within a single file and have to be declared at the beginning of the file with the directive

```
:- module(Name,Exports,Packages)
```

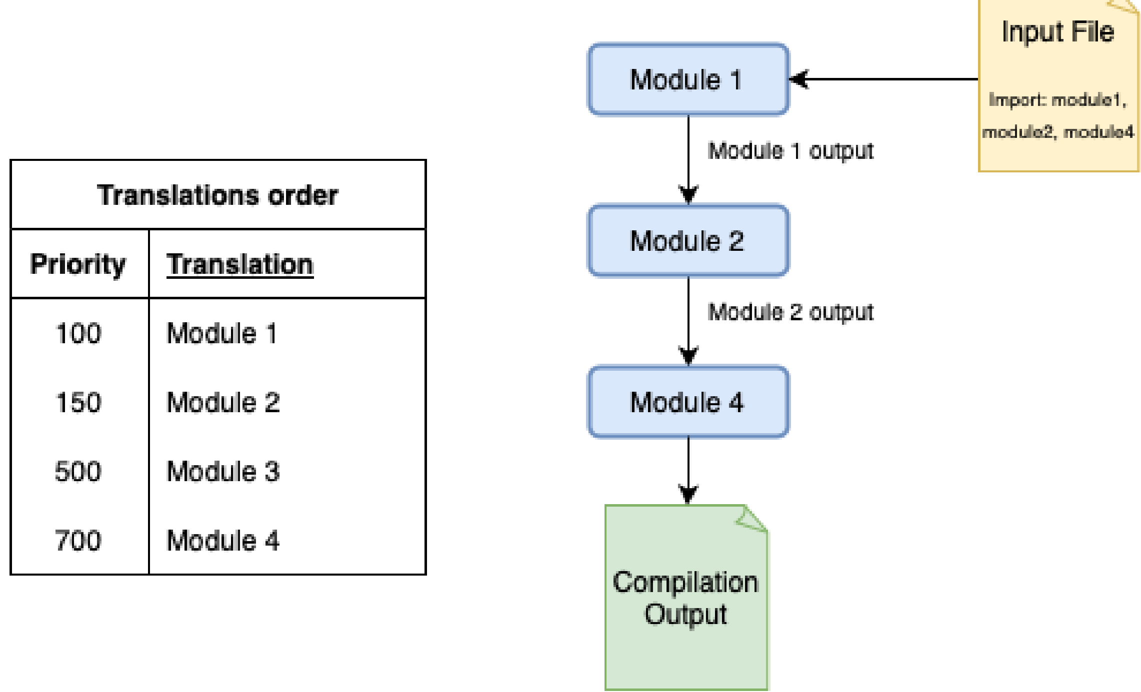
where Name is the name of the module, Exports is a list of predicate names to be exported and Packages are the packages to import. Usually a predicate is completely defined in a single module but some predicates can be declared as multifile, which means that the predicates can be defined across multiple modules. In addition to importing packages, single modules can also be included using the directive

```
:- use_module(Module)
```

These packages can constitute a semantic or syntactic extension which may contain translations, operator definitions or compilation options among other functionalities. A package usually contains several modules, where some contain code that applies for runt-time and others should be part of the compiler. This compilation modules act as ‘plug-ins’ for the compiler and are incorporated when they are declared in `:- load_compilation_module/1` directive, so that the code is used only when required by the module. Furthermore, the package should specify the type of translation it performs (sentence, term, clause, or goal) and include a static entry point in the file to carry out the translation during compile time. [21]

Ciao integrates extensions into its compilation model through compilation modules, which are essentially a set of translation rules [22]. When a module depends on several compilations modules, a specific order must be established for loading and executing them. This is necessary because the output of a compilation module usually is the input of the next one as we can see in Figure D.1.

This is important because translation rules might have dependencies on previous modules, such us the recognition of patterns that are the result of a translation of the previous modules, specially in compilation processes with multiple imports.



Translations order	
Priority	**Translation**
100	Module 1
150	Module 2
500	Module 3
700	Module 4

Figure 2.1: Module order in the compilation process

To address this ordering challenge, `Ciao Prolog` provides with an incremental compilation model [22]. This compilation model is complemented with a list that includes existing translations contained in compilation modules with their respective priorities, similar to operators definitions. The priority assigned to each translation determines the sequence in which it is applied if imported. However, some high-level experimental compilation modules are not included in this list, and a reserved priority space is allocated for them. An example of this ordering can be seen in the translations order list in Figure D.1.

2.1.2 Functional Approach in `Ciao Prolog`

A previous approach of `Ciao Prolog` was the introduction of functional notation [6], making a step towards a syntax more alike to imperative programming. This notation allows the use of functions instead of traditional predicates, which is a feature not available in standard `Prolog`. The inclusion of this extension is relevant to the topic as it shortens the gap between logic and imperative programming, highlighting its usefulness in updating a state variable with the result of a function.

The functional notation extension provides with syntactic sugar that allows the use of functions. At compile time, it transforms function patterns into predicate patterns by adding the output argument of the function as the last argument of the predicate. This transformation, performed the way explained in Section 2.1.1, allow the use of any predicate as function and vice versa, just by changing the position of the last argument.

The function `foo(A,B):=Res :-... .` and the predicate `foo(A,B,Res):-... .` are equivalent, as the functional notation is internally translated into the predicate format during the compilation process. Furthermore, both versions can be called in the two equivalent ways, using functional syntax `R =~foo(X,Y).` or using the predicate form `foo(X,Y,R)` . As seen, this extension allows to call a predicate as if it was a function, ensuring full compatibility of the extension with previously existing libraries and predicates.

The following example shows how this extension works:

```
:- module(_,_,[fsyntax]).

:- use_module(library(write)).
:- use_module(library(streams)).

increment(A,B,Res) :- Res is A + B.

multiply(A,B) := Res :- Res is A * B.

main :-
    increment(1,2,R1), write(r1=R1), nl,

    R2 = ~increment(3,7), write(r2=R2), nl,

    multiply(2,3,R3), write(r3=R3), nl,

    R4 = ~multiply(2,10), write(r4=R4), nl.
```

```
?- main.
  r1=3
  r2=10
  r3=6
  r4=20

  yes
```

As we have mentioned above, `FSyntax` translates patterns of the form `foo(A,B):=Res` into standard predicates of the form `foo(A,B,Res)` . More specifically, we can define `FSyntax` as a package that gives a meaning to terms by translating patterns into other terms that can be later recognized by the next phases of the compiler. Therefore, this functionality has been further extended to recognize other patterns than functional ones, like the curly brackets that will be explained in

Section 2.1.3. Specifically, lately it has been generalized or flexibilized into a package called xsyntax that performs a fine grained control of the extensions. This control is made through a set of flags that can be easily activated or deactivated. The objective is to be able to use one extension, like higher order negative closures, without having to use other, like functional notation.

2.1.3 Higher order

The system includes a package, Hiord, which facilitates higher-order untyped logic programming and combined with hiordx allows the syntax described below. This package supports predicate abstractions, enabling logic programming translations of lambda expressions commonly used in functional programming and ensuring its compatibility with functions from the FSyntax package [23]. The predicate abstractions are anonymous predicates of type:

```
P = {''(X) :- X = 1}
```

Anonymous predicates are predicates which do not have their own name, but they can be referenced by a variable. In the code above P is the variable referencing the anonymous predicate, whose limits are defined between curly brackets. Inside the predicate, the variables inside ''() are the arguments of the predicate, X in the example and the terms at the right side of the neck are the body, X = 1 in this case.

This type of predicate can be called using call/n, which is provided by the Hiord package, as in $call(Pred, Arg_1, Arg_2, ..., Arg_n)$. It can also be called by treating the variable referencing it as a callable entity, like $Pred(Arg_1, Arg_2, ..., Arg_n)$. Our example can be called as `call(P,Y)` or `P(Y)` and the result will be `Y = 1` .

On a step forward, there is a newer extension called hiordx that provides new higher-order notation and semantics. We can define closures as predicates or functions that are able to capture variables from the parent environment. On a first approach to combine closures and anonymous predicates, all the variables instantiated outside the definition of the anonymous predicate are shared by default, except for the arguments of the closure (useful if a variable with the same name appears outside the closure). Such that if we have `foo(Y):- A = 1, P ={ ''(X):- X = A }, P(Y).` the call `foo(Y)` would return `Y = 1` .

In order to explicitly define which variables are to be shared among the environment and the predicate abstraction, hiordx provides with positive sharing closures. This type of closures allows define a list of sharing variables of the form [ ] -> such that if we put `foo(Y):- A = 1, P ={ [A]-> ''(X):- X = A}, P(Y).` , the call would would return `Y = 1` . If we put `foo(Y):- A = 1, P = [] -> ''(X):- X = A, P(Y).` , the call will return `Y = _` as we did not state that A should be a shared variable in the list. As we can see, including this syntax makes that none on the variables found inside the predicate abstraction are captured from the environment by default and they would only be captured if they are explicitly included in the list.

Furthermore, *hiordx* also provides with negative sharing closures, that allows to explicitly define a list of non sharing variables with the form `-[ ] ->` where all the variables are captured by default from the parent environment except if they are explicitly included in the negative list. If we put `foo(X):- A = 1, P = -[] -> ''(X):- X = A, P(Y).` , the call would return `Y = 1` but if we put `foo(X):- A = 1, P = -[A] -> ''(X):- X = A, P(Y).` the call returns `Y = _` as A is not captured from the parent environment.

As we mentioned, the semantics of the notation used above is provided by *hiordx* package. This package has introduced both, positive and negative sharing closures, and the curly brackets notation. More specifically, it is in charge of transforming negative sharing closures into positive sharing closures and transforming both into the internal representation recognized by the *hiord* package. On the other hand, the syntax of both types of closures is provided by the *xsyntax* package explained in Section 2.1.2, with the flags provided by *hiordx*.

2.1.4 Notation Predicate

In order to perform high-level translations, the `Ciao Prolog` system provides the `notation/2` predicate. This predicate defines translations as macros used by the compiler, specifically by the *xsyntax* extension. Whenever *xsyntax* finds an expression matching a pattern defined in *notation*, it substitutes it with the corresponding translation established by the predicate. When it checks if the expression matches any pattern defined, the search is done from the last to the first notation instantiations found. The following example shows how it can be used:

```
:- use_package(fsyntax).
:- use_module(library(between)).
:- use_module(library(aggregates)).

:- op(550, yfx, ..).
:- notation(A .. B, ~range(A,B)).

range(A,B) := Range :-
    findall(X,between(A,B,X),Range).

notation_example(X) :- X = 1..10.
```

```
?- notation_example(X).
   X = [1,2,3,4,5,6,7,8,9,10]
```

We can see that whenever it finds an expression of the type `A ..B`, it substitutes it by `~range(A,B)` such that the rest of the compilation and the execution just take into account its translation.

2.2 Imperative Approaches in Logic Programming

In this extension of logic programming it is interesting to introduce new characteristics like loops or state variables, inspired in other programming languages, to Prolog. Taking steps towards a more imperative syntax and state representation is relevant in logic with approaches like LPS [24].

Talking about imperative approaches we should mention Picat [5], which highlights with its imperative notation, and has been an inspiration for this imperative notation extension. Picat is a versatile, logic-based multi-paradigm programming language designed for general-purpose applications. It integrates features from logic programming, functional programming and various different modules. Some of the its most important characteristics are that it works using single-sided unification instead of standard unification and that most of the common modules are automatically included in every file.

In order to review the logic programming approaches in imperative notation we are going to divide the section into some representative concepts of imperative programming:

2.2.1 Single-sided Unification

Single-sided unification is a type of unification that performs pattern matching. This is based on the existing predicate *subsumes_term/2* , which is part of the International Standard for Prolog [25]. The predicate *subsumes_term(Term1,Term2)* succeeds if *Term2* is an instance of *Term1* . Similarly, this type of unification succeeds if the call is an instantiation of the head of a predicate and fails if it does not match any clause.

This type of unification is widely used in the logic programming language Picat, where it is the default unification mechanism. Picat refers to it as pattern matching rather than unification, arguing that it simplifies indexing of rules [5]. Other logic programming languages, like SWI-Prolog, provide this feature as an option or extension [26]. Now, we can see the Picat implementation of fibonacci predicate that uses single-sided unification:

```
fib(0,F) => F = 0.
fib(1,F) => F = 1.
fib(N,F), N > 1 => fib(N-1,F1), fib(N-2,F2), F = F1+F2.
```

In imperative programming, pattern matching simplifies data filtering and provides a more readable approach for executing specific instructions based on data characteristics, although it is more common in functional paradigms. For example, in Python, the match statement performs pattern matching, where only the first matching case is executed. This avoids the use of multiple 'if-else' structures, which can make the program harder to read . Below, we present the translation of the previous fibonacci predicate into Python, where the match statement performs the pattern matching analogous to the single-sided unification feature in Picat, giving the same results in both languages:

```
def fib(n):
    match n:
        case 0:
            return 0
        case 1:
            return 1
        case _ if n > 1:
            return fib(n-1) + fib(n-2)
```

2.2.2 State Variables

A *state variable* is a variable which keeps its current state, meaning its value can change throughout the execution of a program. In imperative programming languages, like Python, all standard variables are considered state variables as they allow assignment operations that modify their values and consequently their state. For example, if we first execute `x = 1` the value of `x` is set to 1. Then, if we execute `x = 4`, the value of `x` will be set to 4, overwritting the previous value 1.

A declarative representation of state can be achieved by using Definite Clause Grammars (DCGs), which, thanks to their implicit arguments, can be used to represent both the previous and next state of a variable. For example, DCGs automatically perform a translation:

```
foo(X) --> a(X), b(X)
⇓
foo(X,OrigSt,FinSt) :- a(X,OrigSt,MidSt), b(X,MidSt,FinSt)
```

where the initial and final states of variable `OrigSt` are kept in variables `OrigSt` and `FinSt` respectively. We can clearly see the representation of state included as arguments of each predicate call along the `foo` predicate body.

DCGs can be extended to track multiple variables by encapsulating them in a single structure, but this can be difficult to manage, as it should be done with one structure containing all of the variables.

For example, if we had the variables `St1` and `St2` to change of state a predicate of the type

```
foo(X) --> a(X), b(X)
⇓
foo(X,(OrigSt1,OrigSt2),(FinalSt1,FinalSt2)) :-
a(X,(OrigSt1,OrigSt2),(MidSt1,MidSt2)),b(X,(MidSt1,MidSt2),(FinSt1,FinSt2))
```

such that even if there are more state variables all predicate stay of $Arity = Arguments + 2$. In this case the arity is 3, as the dcg has just one argument `X` and the two arguments created by the translation represent respectively, the original state and the final state.

To address this limitation, extended DCGs were introduced by Peter Van Roy [27], which allow simultaneous updates to multiple variables. However, EDGCs are still a comparatively less intuitive method for state management.

A more direct approach is offered by languages like Picat [5], which uses the := operator for updating state variables. Mercury uses the ! notation to denote state variables and automatically expands variables into two to represent the state before and after a state change [28].

In addition to state variables, logic programming systems often support some form of mutable variables and/or the setarg/3 primitive, which allows destructive updates to term arguments. This is typically used for localized updates to large data structures, such as array assignments, rather than for tracking the state of individual variables. However, this kind of destructive assignment is at odds with the declarative nature of logic programming, as it complicates the semantics significantly.

2.2.3 Loops

Loops are sets of instructions that execute repeatedly until a specific condition is met [29]. The most common type of loops, for and while, can be found in imperative programming languages, like Java or Python. There are multiple approaches to include this notation into other programming paradigms. For example, Oz, a multi-paradigm programming language, has introduced loop syntax [30].

As already mentioned at the beginning of this chapter, initial approaches towards introducing loop notation into other programming paradigms are provided by MiniZinc, and, inspired on it, the mathematical modeling library for Prolog proposed by François Fages.

In logic programming languages, loops have been introduced in various ways. One way of introducing them is by creating new languages which are not extensions of Prolog, like Picat, which provides a loop syntax that is actually similar to the one provided by imperative languages, making it intuitive for new users that come from imperative paradigms. It also allows the use of other iteration constructs, such as array and list comprehensions, which enable compact notation to perform some of the functionalities of loops [5]. We can see an example where the predicate uses loops and state variables to count the number of even elements from 1 to 10, illustrating the syntax allowed below:

```
example =>
    Res = 0,
    foreach (I in 1 .. 10)
        if I mod 2 = 0 then
            Res := Res + 1
        end
    end, println(Res).
```

```
Picat> example.
       5
```

Another way of introducing loops into logic programming is by making it from the point of view of Prolog, as an extension of it. This is made in this way in order to maintain Prolog and the compatibility to ISO-Prolog. One of the most important approaches in this way of doing it was done by ECLiPSe, a *constraint logic programming* system that has developed logical loops [31]. Logical loops have the form (IterationSpecifiers do Body), where the type of loop is defined in IterationSpecifiers. This is a close approach although the syntax may seem confusing as it does not take the usual loop form from imperative programming languages.

2.2.4 Array-Like Index Notation

The term *array-like* refers to objects that have a length N and allow elements to be accessed using an index, like lists or arrays. When we talk about index notation, we mean accessing these elements with syntax like `Array[I]`.

Most imperative programming languages, like Python or Java, use this style for arrays. Python also applies similar syntax to other data structures, such as dictionaries. In the logic programming paradigm, some languages like Picat [5] or SWI-Prolog [32] allow this kind of notation. As mentioned in Section 2.2.2, Picat allows placing an element, referred with array-like index notation, on the left side of an assignment. This feature enables a notation closer to imperative programming where structures, like arrays or lists, change their state by modifying one of their elements by its index.

Chapter 3

Single-sided Unification

Single-sided unification is a type of unification that modifies the process by which a call matches the head of a clause. This means that it determines whether a predicate's clause can be applied to a specific call. We focus specifically on `Picat`-style pattern-matching, which implies some a particular set of rules for a match to succeed.

3.1 Pattern-Matching Rules

In order to explain these rules we consider a predicate whose head is of the type $p(T_0, T_1, ..., T_n)$. The head constitutes the pattern that a call must follow to enter to its body. In particular, the way matching behaves is that a clause will be entered if:

1. The call and the head of the clause are exactly the same, or
2. The head of the clause can be instantiated to be exactly as the call, which implies the head of the clause is more generic than the call.

3.2 Syntax

The clauses that perform single-sided unification must be of the type:

```
Head, Guard => Body
```

```
Head, Guard ?=> Body
```

3.3 Semantics

The predicates which use single-sided unification are composed of clauses whose `Head` is of the type $p(T_0, T_1, ..., T_n)$ where p is the name of the of the predicate, n is the arity of `p`, and T_i are the arguments. These clauses also contain a `Guard` which acts as a condition that must be satisfied in order enter the clause. The `Guard` can be formed by one or more conditions or be empty. The clauses also

include a Body which contains a sequence of literals that are called if both the pattern matching and the guard succeed.

Apart from these parts, such clauses use on of two types of neck operators:

- '=>': This neck implies that this rule is not backtrackable which acts as if there was a cut in the neck. In order to compare with Prolog, the equivalence is:

  Head, Guard => Body
  ≡
  Call :- subsumes_term(Head,Call), Head = Call, Guard, !, Body

- '?=>': This neck implies that this rule is backtrackable which makes it more similar to standard Prolog clauses. In order to compare with Prolog, the equivalence is:

  Head, Guard ?=> Body
  ≡
  Call :- subsumes_term(Head,Call), Head = Call, Guard, Body

In the translations we use subsumes_term/2, the ISO-Prolog predicate explained in Section 2.2.1. If the pattern-matching rules succeed and the Guard is true, the body of the clause will execute. Otherwise, the system will continue to attempt pattern-matching with the remaining clauses. If none of the heads match, the call will fail instead of throwing a “no matching head exception.” The following example shows a single-sided unification predicate that succeeds if two structures are equal:

```
struct_equal(A,B), atomic(A) =>
        A == B.
struct_equal([H1|T1],[H2|T2]) =>
        struct_equal(H1,H2),
        struct_equal(T1,T2).
```

However, if the desired behavior is to throw an exception when the call does not match any head, this can be achieved as in the following example:

```
struct_equal(A,B), atomic(A) =>
        A == B.
struct_equal([H1|T1],[H2|T2]) =>
        struct_equal(H1,H2),
        struct_equal(T1,T2).
struct_equal(_,_) =>
        throw(error(no_matching_head, struct_equal/2)).
```

If we make a call to struct_equal_exc(A,B) in the original implementation, it will fail because it in not an instantiation of any of the clause heads. In the second implementation, if a call matches one of the heads above, it will not enter the

last clause due to the cut shown in the translation, behaving the same as the original implementation. However, if the call does not match any of the heads, it will always enter the last clause throwing the exception instead of failing.

The aim of the decision of making such cases fail is to allow backtrackable single-sided unification predicates to interact with existing predicates, such as findall/3. For this type of aggregation predicates, it would result in throwing the exception instead of producing the correct results, as seen in the following example:

```
my_member(Y,[X|_]) ?=> Y = X.
my_member(Y,[_|L]) => my_member(Y,L).

show_all(A) :- findall(X,my_member(X,[1,2,3,4]),A).
```

```
?- show_all(X).
{ERROR: No handle found for thrown exception error(no_matching_head,my_member/2)}
```

3.4 Compilation Rules

When a rule of the type mentioned in Section 3.2 is found, an option would be compiling it to an equivalent Prolog rule the way it is explained in Section 3.3. On the other hand, Ciao improves on the efficiency of subsumes_term/2 by transforming it into a series of simple terms and unifications which allow indexing to work on the rule. This optimization achieves better performance in the execution of calls but it only works if all the variables are free and new.

In order to visualize the rules and translations explained along all Section 3 we can see the fibonacci predicate, which returns the nth fibonacci number, written with the notation described in Section 3.2:

```
fib(0,F) => F = 0.
fib(1,F) => F = 1.
fib(N,F), N > 1 =>
        N1 is N - 1,
        N2 is N - 2,
        fib(N2,F2),
        F is F1+F2.
```

Its semantic translation explained in Section 3.3 with subsumes_term/2 is:

```
fib(X1,X2) :-
        subsumes_term(fib(0,F),fib(X1,X2)),
        fib(0,F) = fib(X1,X2),
        !,
        F = 0.
fib(X1,X2) :-
        subsumes_term(fib(1,F),fib(X1,X2)),
        fib(1,F) = fib(X1,X2),
        !,
        F = 1.
fib(X1,X2) :-
        subsumes_term(fib(N,F),fib(X1,X2)),
        fib(N,F) = fib(X1,X2),
        N > 1 ,
        !,
        N1 is N - 1,
        fib(N1,F1),
        N2 is N - 2,
        fib(N2,F2),
        F is F1 + F2.
```

Including the optimization performed by the compiler explained in this section instead of subsumes_term/2:

```
fib(N,F) :-
        nonvar(N),
        N = 0,
        !,
        F = 0.
fib(N,F) :-
        nonvar(N),
        N = 1,
        !,
        F = 1.
fib(N,F) :-
        true,
        N > 1,
        !,
        N1 is N - 1,
        fib(N1,F1),
        N2 is N - 2,
        fib(N2,F2),
        F is F1 + F2.
```

3.5 Compatibility With Other Modules

One of the most important features of Ciao is that extensions are modules which can be included only when each of them is needed, as explained in Section 2.1.1. The extension that implements single-sided unification, which is called `pmrule`, is compatible with the rest of the modules and packages of the system.

In this way, using `fsyntax`, it is possible to use functional notation in predicates that perform pattern matching. In these cases `pmrule` does the single-sided unification and delegates to the `fsyntax` package for functional part. Continuing with the fibonacci example, the following example shows this:

```
fib(0) := F => F = 0.
fib(1) := F => F = 1.
fib(N) := F, N > 1 => F = ~fib(N-1) + ~fib(N-2).
```

It is also important to mention that clauses of the type $p(T_0, T_1, ..., T_n)$ `:= X` will not be compiled with single-sided unification unless `=> true.` is written after them, so that the clause has the form $p(T_0, T_1, ..., T_n)$ `:= X => true` and can thus be distinguished from normal clauses.

This extension is also compatible with the higher order extension explained in Section 2.1.3. Moreover, it also combines nicely with the imperative extensions explained in the rest of this project as we will see in Chapter 7, where some of the examples integrate them all and function correctly.

Chapter 4

State Variables

State variables in Ciao Prolog are variables that change their value during the execution of a program. This implies that the variable mutates from State1, being the old value, to State2 representing the new value. State variables are also essential in loops, as they allow changing the values of the variables used to control them. Also, they allow implementing some common and useful patterns in loops, like the sum of all the elements of a list where the counter needs to change value along the iterations. This can of course be done declaratively by using two variables or other mechanisms, but the advantage of using state variable *notation* is that it makes loops more readable, making the system more attractive for new programmers coming from imperative languages such as Java or Python, as mentioned in Chapter 1.

To use state variables in a Ciao module, the statevars package must be loaded. This package:

- establishes the syntax via *notation* directives,
- loads the xsyntax package, flagging it to identify state variables syntax, and
- loads the xcontrol package.

We will go deeper into the utility of each of these packages along the chapter.

4.1 Syntax

The syntax is provided by a combination of *notation* directives, applied through the *xsyntax* package. The assignment statement is used to update the value of a variable. The operator used is ':='. The form of assignments is:

```
X := NewVal
```

where the left part must be a Variable and the right part can be any term, such as an atom, another variable, or the result of a call in functional syntax.

When the variable that changes state is included in the head of the clause this variable must be preceded with a '!', as `foo(A,!X)` , which indicates that X might

be subject to a change of state and that `X` will get the new value. This notation to represent state variables in the head of a clause or a call was inspired by Mercury, where the expression `!X` represented that a variable has a "`current`" and a "`next`" state [28].

If the '!' is not included, and the variable is subject to an assignment, it will continue having the old value instead of the new value at the end of the predicate execution.

An example of use of state variables in a predicate is '`inc(!X):-X :=X+1.`'. If this predicate is called with '`X =1`', then after the call `inc(!X)` is '`X =2`'. This predicate can also be called `inc(X,X1)` and after the call it would be '`X=1, X1=2`'. This implies that '`inc(!X):-X :=X+1.`' expands to '`inc(X,X1):-X1 is X+1.`' and after a call, the name of variable `X` will point to the new value `X1`. This approach thus combines imperative notation while remaining declarative underneath.

4.2 Compilation Rules

As mentioned before, in order to use state variables, the `statevars` package must be imported into the module. This package integrates its compilation module and other system packages that implement specific steps of the compilation process, as detailed in Section 2.1.1. More specifically, we can identify two different phases that we will explain below. To illustrate this sequence we are going to focus on this example:

```
a(!Y) :-
    Y := 'a_val_2'.

st_example(!X) :-
    X = 'x_initial',
    X := 'x_val_2',
    (X := 'x_cond_val' ; true),
    A = 'a_initial',
    a(!A),
    X := A.
```

We can see that this example is specially interesting as it contains different changes of state through the assignment operator '`:=`' and two clauses whose argument contains '`!`'. Additionally, we can see that there is a change of state of variable `A` through the call to the '`a`' predicate.

4.2.1 First Phase

This first phase is performed by *xsyntax*, explained in Section 2.1.2, that identifies a set of predefined patters and translates them into a normalized internal representation. This allows the rest of the modules loaded afterwards to identify them, in a faster and simpler way, and give them the correct treatment. More

specifically, to compile state variables, the *statevars* compilation module loads *xsyntax* with its own flag, making it perform two translations.

The first translation occurs when xsyntax encounters an expression of the type `foo(!X)`, where foo is a functor and `!X` represents an argument where `X` is a state variable. This phase translates such an argument into a pair of two arguments:

- *before(X)*: represents the current state of the variable if it is in a call or the state of the variable before executing the clause if it is found in the head of a clause, acting as an input. We will denote *before(X)* more compactly as $X_\circ$.
- *after(X)*: represents the next state of the variable if it is in a call or the state of the variable after executing the clause if it is found in the head of a clause, acting as an output. Along this project we will denote *after(X)* as $X_\bullet$.

If the functor has multiple arguments, *xsyntax* processes each one individually, applying the same translation. For example, `foo(!X, K, !Y)` will be converted to foo($X_\circ$, $X_\bullet$, K, $Y_\circ$, $Y_\bullet$). This transformation implies that a term `name/A` will be modified to `name/A+N`, where N represents the number of state variables within its arguments.

The second translation occurs when *xsyntax* encounters an expression of the type '`X := Y`'. It normalizes this expression into '$X_\bullet$ = Y', but for better understanding reasons we will notate it through the text as '`assign(X,Y)`'. After applying these two term transformations, the resulting clauses serve as the input for the next phase of the compilation process.

In summary, the schematic representation of the two translations is:

$$foo(!X) \Longrightarrow \texttt{foo}(X_\circ, X_\bullet)$$

$$X := Y \Longrightarrow \texttt{assign}(X, Y)$$

Turning back to our example, after this compilation phase it will be transformed to:

```
a(Y∘,Y•) :-
    assign(Y'a_val_2').

st_example(X∘,X•) :-
    X = 'x_initial',
    assign(X,'x_val_2'),
    (assign(X,'x_cond_val') ; true),
    A = 'a_initial',
    a(A∘,A•),
    assign(X,A).
```

4.2.2 Second Phase

In order to apparently change the values of variables in an imperative way, Prolog does not provide a mechanism, since the basis of the programming languages is single assignment through unification (as well as being able to perform backtracking, etc.). For that reason, to simulate the imperative behavior we need to record and keep the track of the state of that variable.

The `Ciao Prolog` System manages the state at compile time, by generating a set of simple terms and unifications in order to deal with state variables in a particular scope. This part of the compilation process is done by *xcontrol*. In order to handle state we need to define an environment structure made up pairs that contain the state variable name and its current value of the form:

$$\texttt{Env} = \Gamma[StVar_1 \mapsto CurrSt_1, StVar_2 \mapsto CurrSt_2, ..., StVar_N \mapsto CurrSt_N]$$

where '$\Gamma[\ldots]$' represents an environment and '$X \mapsto Y$' represents that the current state of X is Y.

Apart from the structure that manages the current state, we need to set two other structures of the same form:

- **EnvIn:** contains the initial state of state variables found in the head of the clause, which is passed as an argument. Each pair is made up of the state variable and the variable substitution '$\texttt{X}_\circ$'.
- **EnvOut:** contains the final state of state variables found in the head of the clause, which is passed as an argument. Each pair is made up of the state variable and the variable substitution '$\texttt{X}_\bullet$'.

Therefore, the two structures are initialized at the beginning of this compilation phase where all the arguments of the head of the clause are parsed and treated in the following way:

- Each '$X_\circ$' expression is replaced by by a free variable which will represent the initial state of `X`. The pair state variable-initial state is added to `EnvIn`.
- Each '$X_\bullet$' expression is replaced by a free variable which will represent the final state of `X`, containing the result of the assignments made to `X` through the body. The pair state variable-final state is added to `EnvOut`.

In the st_example/2 clause of our example at this point EnvIn, EnvOut and the head of the predicate will be:

$$\texttt{EnvIn} = \Gamma[X \mapsto X0]$$
$$\texttt{EnvOut} = \Gamma[X \mapsto Xn]$$
$$\texttt{Head = st_example(X0,Xn)}$$

Once two environments have been set, the current environment is initialized. This is a combination of the parent environment, that for one scope is an empty list, and `EnvIn`. If the state variables appear in both environments, the current

environment will be set to the *EnvIn* variable. Following our example we will initialize our current environment as:

$$\texttt{Env} = \Gamma[X \mapsto X0]$$

To explain the translation process, it is important to explain first the translation of control structures, such as conjunctions and disjunctions:

- *Conjunctions:* when a conjunction appears, for example with the `','` operator, they are translated by sequentially threading the environments. For example, if we have the expression *`TermA, TermB`* it will process `TermA` resulting in the environment `EnvA` that will be the input environment of the translation of `TermB`.
- *Disjunctions:* when a disjunction appears, for example with the `';'` operator, the two branches need to be treated such that the environment, `FEnv`, remains consistent after it. Specifically, the translation processes each branch individually, applying updates based on the environment at the disjunction point and after translating both branches, their resulting environments are unified. The unification process is performed by comparing the resulting environments of each of the branches and the environment at the point of disjunction. To illustrate this, assume we have branch A with environment $\Gamma[St \mapsto ValA]$, branch B with environment $\Gamma[St \mapsto ValB]$, and the previous environment $\Gamma[St \mapsto Val0]$. The process will proceed as follows:
  - If the value has only changed in one of the branches, branch A, on the other branch we add the clause `'ValA = Val0'` such that the environment after it is '$EnvF = \Gamma[St \mapsto ValA]$' and both branches leave the last value of `St` in `ValA`.
  - If the value changes on both branches it unifies `ValA` and `ValB` without generating extra clauses and the environment after the disjunction is '$EnvF = \Gamma[St \mapsto ValA]$'.
  - If a state variable was created in one of the branches it is just added to `FEnv`.

Second, the translation of terms or literals, which is performed sequentially, is as follows:

- Whenever a term of type *assign(X,Y)* is found, the current state of `X` is changed by setting it to a new free variable. In order to make a transition of state of `X` variable, '=(NewFree,Y)' is added to the body and the variable in the current environment is updated to '$X \mapsto NewFree$'. If `X` is not in the environment, it is added to the environment as '$X \mapsto NewFree$'.
- Whenever a term of type '$X_\circ$' is found, it is replaced by the current state of `X` in the environment. If `X` is not in the environment it is replaced by itself.
- Whenever a term of type '$X_\bullet$' is found, it acts as an assignment where the current state of `X` is changed in the environment, setting it to a new free variable, '$X \mapsto NewFree$', and '$X_\bullet$' is replaced by the new variable. If `X` is

not in the environment, it is added to the environment as a state variable '$X \mapsto NewFree$'.

- Whenever other type of clauses are found, the occurrences of state variables, which are on the environment, are replaced by their current state in the environment. For example, if '=(Y,X)' appears and X is a state variable with current state XCS, the term is replaced in the body by '=(Y,XCS)'.

To visualize these rules in a more compact and clear way, Table 4.1 presents a schematic representation of the rules.

Table 4.1: Translation rules for state variables

Term Translation			**Environment**		
Term		**Translation**	**Env**$_0$		**Env**$_1$
assign(X,Y)	$\Longrightarrow$	X1 = Y	$\Gamma[X \mapsto X0]$	$\Longrightarrow$	$\Gamma[X \mapsto X1]$
$X_\circ$	$\Longrightarrow$	X0	$\Gamma[X \mapsto X0]$	$\Longrightarrow$	$\Gamma[X \mapsto X0]$
$X_\bullet$	$\Longrightarrow$	X1	$\Gamma[X \mapsto X0]$	$\Longrightarrow$	$\Gamma[X \mapsto X1]$
Other term	$\Longrightarrow$	Replace state variables with their current state	Env	$\Longrightarrow$	Env

The execution of these rules over the clauses, taking into account each clause will be compiled independently as it is a clause translation, will be as summarized in Table 4.2.

Table 4.2: Execution of compiling rules over the example.

Clause	**Translation**	**Environment**
a($Y_\circ$,$Y_\bullet$)	a(Y0,Yn)	$\Gamma[Y \mapsto Y0]$
`assign(Y,'a_val_2')`	Y1 = 'a_val_2'	$\Gamma[Y \mapsto Y1]$
st_example($X_\circ$,$X_\bullet$)	st_example(X0,Xn)	$\Gamma[X \mapsto X0]$
X = 'x_initial'	X0 = 'x_initial'	$\Gamma[X \mapsto X0]$
`assign(X,'x_val_2')`	X1 = 'x_val_2'	$\Gamma[X \mapsto X1]$
`(assign(X,'x_cond_val')`	(X2 = 'x_cond_val'	$\Gamma[X \mapsto X2]$
; true)	; X2 = X1)	$\Gamma[X \mapsto X2]$
A = 'a_initial'	A = 'a_initial'	$\Gamma[X \mapsto X2]$
a($A_\circ$,$A_\bullet$)	a(A,A1)	$\Gamma[X \mapsto X2, A \mapsto A1]$
`assign(X,A)`	X3 = A1	$\Gamma[X \mapsto X3, A \mapsto A1]$

At the end, after parsing the entire scope the translation checks whether state variables in `EnvIn`, in the head of the clause, are in the current environment. If they are, a term is added that unifies, for every variable, the current state and its final state found in `EnvOut`. If they are not, they are unified with their variable in `EnvIn`.

The resultant translation of our example after the compilation process finishes is:

```
a(Y0,Yn) :-
    Y1 = 'a_val_2',
    Yn = Y1.

st_example(X0,Xn) :-
    X0 = 'x_initial',
    X1 = 'x_val_2',
    (X2 = 'x_cond_val' ; X2 = X1),
    A = 'a_initial',
    a(A,A1),
    X3 = A1,
    Xn = X3.
```

If we write this translation inside the predicate `st_example_tr`, any call to the original `st_example(X, X1)` or the translated `st_example_tr(X, X1)` will produce the same result: `X = x_initial, X1 =a_val_2`.

Chapter 5

Implementation of Array Index Notation

This chapter describes how we have designed and implemented the array extension to allow a common notation for different types of arrays and lists. The goal of this extension is to provide a common syntax for different implementations of arrays and lists. This notation slightly shortens the gap between logic and imperative programming as it brings closer the syntax used in both. In this extension we provide the same syntax for three different types of array implementations, with different characteristics and efficiency, and one list implementation.

5.1 Design of the Package

This extension is composed of six files: four modules dedicated to implementing the four types of data structures used, and two additional files that integrate these implementations, enabling their use as a single package. The `arrays` package is loaded with the directive:

```
:- use_package(arrays).
```

This first file, `arrays`, defines the operators and notations required to implement the syntax described in Section 5.2. It also establishes the interface specifications, which are included in the `arrays_itf` module. The `arrays_itf` module serves as an interface by declaring the predicates for handling array-like objects as `multifile`. This interface can be found in Appendix C. The `multifile` declaration allows predicates to be defined across multiple modules instead of being confined to a single one. This approach ensures a unified syntax for various array-like objects while enabling a separate and unique implementations for each type of object, as operations often vary depending on the object.

The use of multifile predicates was chosen to enhance modularity. This design ensures that modifying one implementation does not affect others, providing a reliable implementation to support flexibility. Moreover, programmers

can easily add custom implementations by creating a new module, including `:- include(arrays_itf).` to indicate it implements the array interface, and then defining their custom operations as specified by the interface. This modular design encourages extensibility without compromising existing implementations.

One of the most important features of this package is that it relies on the state variables package, specially when changes need to be made in the nth element of a structure. This shows the power of the extension described previously, so that just by using the *notation* directive, new language extensions can be implemented.

5.2 Syntax

The syntax for all the different implementations is the same and only differs in the predicate needed to create a new array-like structure, which can be one of the following:

- `new_array_log/1`: This predicate creates a new extendable array, with logarithmic access time, and without fixed initial length, which is returned on the argument.
- `new_array_mut/2`: This predicate creates a new destructive array of a given length which is mutable and uses internally `setarg/2`.
- `new_array_fix/2`: This predicate creates a new array of a given length.
- In the case of lists, to create a list of a defined length, one can use a call to the `length/2` predicate, which is already defined in the system.

The following syntax is provided by utilizing a high-level translation through the notation directive, as explained in Section 2.1.4. The expressions using index notation are identified and translated by the *xsyntax* package. Specifically, in order to get the nth element of the data structure the notation is:

```
Array[Index]
```

In order to change the nth element of the structure the notation is:

```
Array[Index] := Element
```

Where `Index` must be greater or equal to 0 as this implementation starts indexing the structure at 0. Apart from this syntax, the package also provides as part of the interface an `array_length/2` predicate that returns in its second argument the length of an array-like structure.

5.3 Semantics

One of the main features of the extension is that *xsyntax* gives a meaning to the syntax explained above. But focusing on the assignment syntax, it relies solely

on the state variables extension, that replaces the whole array with the new one with the element changed. The specific translation, as defined with the *notation* directive, is:

Array[Index] := Element
⇓
replace_elem(Array,Index,Elem,Array1), Array := Array1

The specific translation, also defined with the *notation* directive, for the syntax provided to get the nth element of the structure using the `FSyntax` package, explained in Section 2.1.2, is the following:

Array[Index]
⇓
get_elem(Array,Index,Elem)

Other of the main aspects of the extension is the fact of being an interface with various implementations. Each of those implementations is loaded by including a directive of the type:

:- use_module(library(arrays/arrays_log)).

The following section describes and discusses each of the implementations of the syntax described above.

5.3.1 Mutable Arrays

This type of arrays are implemented as functors whose arity is one, with a single argument which is another functor whose arity is the length of the array. More specifically its representation is `array_mut(data(Elements))`. This is done to take advantage of `Ciao Prolog`'s indexing system. The length of the array is given as an argument at the moment it is created.

The main feature of this kind of arrays is that the change of the nth element of the array is made with `setarg/3`, which performs a destructive assignment of the nth element. This means that the element is changed without creating a copy, and this is performed by destroying the old array so that all the times the term is referred to, it points to the new array. The access to the nth element is done through the `arg/3` built-in predicate, as the structure is a functor with arguments.

```
destructive :-
    new_array_mut(5,Array),
    for (I in 1 .. 5) {
        Array[I-1] := I
    },
    ArrayCopy = Array,
    Array[2] := 25,
    write(copy=ArrayCopy),nl,
    write(array=Array),nl.
```

```
?- destructive.
    copy=array_sa(1,2,25,4,5)
    array=array_sa(1,2,25,4,5)
```

In this code we can appreciate that although we assign `ArrayCopy` before changing the third element, when we print it at the end it has also changed. This is due to the destruction of the old array so that all terms that were pointing to it, now point to this new version of the array.

5.3.2 Fixed Arrays

This type of arrays are implemented again as functors whose arity is one, with its single argument a functor whose arity is the length of the array, to take advantage of `Ciao Prolog`'s indexing system. The length of the array is given as an argument at the moment it is created, more specifically its representation is `array_fix(data(Elements))`.

The main difference between this type of arrays and those discussed in Section 5.3.1 lies in their behavior when an element is modified. In this case, changing an element results in the creation of a new array that is a copy of the original one with the changed element, while the old array is not destroyed. Consequently, this term will reference the new array, whereas the rest will continue to point to the old array. The access to the nth element is done through the `arg/3` predicate as the structure is a functor with arguments.

```
non_destructive :-
    new_array_fix(5,Array),
    for (I in 1 .. 5) {
        Array[I-1] := I
    },
    ArrayCopy = Array,
    Array[2] := 25,
    write(copy=ArrayCopy),nl,
    write(array=Array),nl.
```

```
?- non_destructive.
    copy=array_mut(1,2,3,4,5)
    array=array_mut(1,2,25,4,5)
```

In this code we can appreciate that as `ArrayCopy` is assigned before changing the third element, when we print it at the end it remains without the change. This implies that the assignment creates a new structure, and changes made to each instance do not affect the other one.

5.3.3 Extendable Arrays

Extendable arrays is an implementation which allows using arrays whose length is not fixed. This means that it is created without a defined maximum length and it can be extended as much as required during the execution. This makes

it suitable for situations were the length required of the array is not known in the moment of creating it. This extension just provides the syntax as all the predicates are already implemented in the existing `logarrays` module of the `Ciao Prolog` System, which performs with logarithmic complexity [33]. This implies that all the operations described can also be performed without using this syntax if preferred.

This type of arrays are represented with `array/2` whose first functor is an structure that contains the element and whose second functor is `Size`. The current number of elements is determined by `Size`, more specifically it is 2^{Size}. The structure is represented by the functor `$` and implemented as a balanced 4-tree. In particular it is a balanced tree in which each internal node contains a `$/4` structure and each leaf node contains a element of the array or it is empty. This can be observed in Figure 5.1

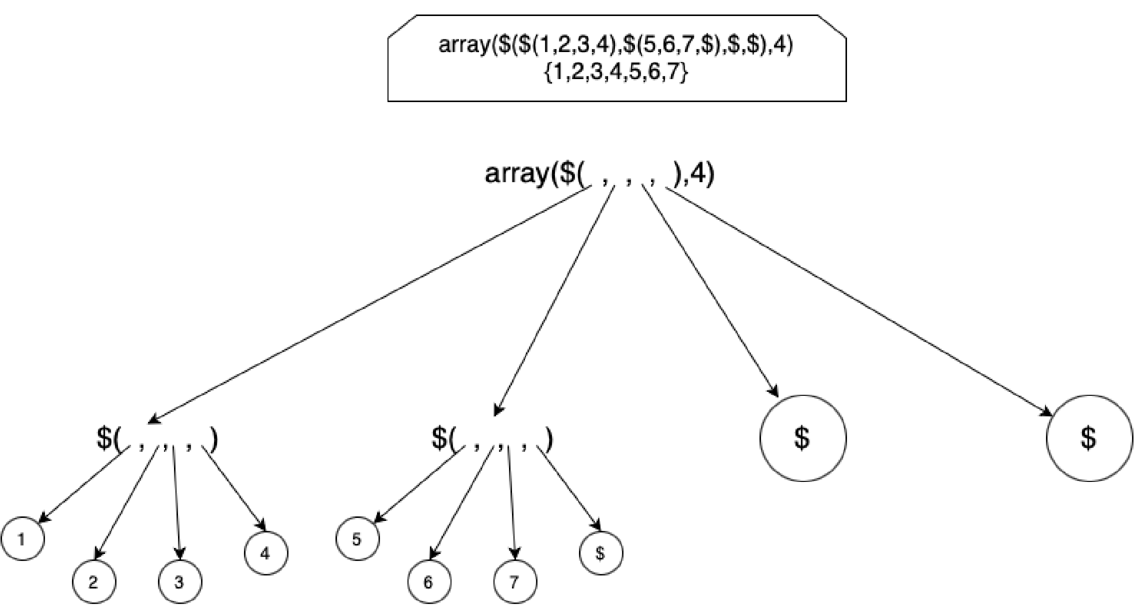


Figure 5.1: A tree representing an array of seven elements (from 1 to 7)

This makes it extensible as it allows creating structures dynamically by increasing de depth of the tree as the length of the array increments. It is important to mention that to access de nth element of the array we have chosen to use verb.aref/3., which fails if the nth element has not already been set, as it seems more natural and makes it more similar to the behavior of array-like structures in imperative languages.

5.3.4 Lists

The array-like structure can also be a list. All the operations described in Section 5.2 can be made over arrays except the one for creating the list. In order to create this type of structure of a defined length, the `length/2` predicate must be used. This predicate can be found in the existing library of lists. It can be used

for example as in `length(L,5)` which returns in L a list of length 5 where all its elements are free variables. The access to the nth element it is necessary to iterate recursively over the list until the element is found. The assignment creates a new list and iterates through the existing list copying its elements until the nth element, where it includes the new value in the new list, and then the rest of the original list is added to the new list.

5.4 Results of the Array Implementations

To test this extension we have implemented four initial predicates. Each of them creates a different array-like structure of length N. These predicates all call the same predicate that fills the structure with random unordered values, and calls another predicate that implements the quicksort sorting algorithm. This predicates uses all the syntax and functionalities implemented in the package and the code can be found in the Appendix B.

More specifically, we execute these predicates introducing different lengths of the array-like structure (N). For these executions we measure the times to be able to make a comparison between the different implementations. These times can be found in Table 5.1 in seconds, although the *list* implementation has not been measured for N > 100000 as it takes more than 10 hours.

Table 5.1: Results of the quicksort execution in arrays

N	**Mutable**	**Fixed**	**Extendable**	**Lists**
10	0.000042	0.000057	0.000073	0.000062
100	0.000298	0.005191	0.001416	0.007436
1000	0.005336	0.943855	0.031179	1.332778
10000	0.117529	ERROR	0.87232	188.762667
100000	0.818229	ERROR	7.503203	23605.758907
1000000	10.569122	ERROR	108.954372	
10000000	121.771078	ERROR	1366.598981	

In Table 5.1 we can appreciate that all the implementations work correctly, but *fixed* arrays gives an error for N > 1000 and this is because it eventually runs out of memory due to the multiple copies it has to perform when changing an element. We can also appreciate *lists* are significatively slower than the *mutable* and *extendable* arrays. This is because its implementation recursively iterates the whole list until the nth element. Moreover, we can see the fastest array-like structures implemented are *mutable* arrays as its implementation is able to directly access the nth position and modify it while *extendable* arrays have logarithmic complexity.

Chapter 6

Loops

In this section, we will first describe the syntax of loops, then explain how it connects to the compilation process, and finally cover the compilation process that enables the use of loops.

To use loops in a `Ciao` module, the `loops package` must be loaded. This package:

- establishes the syntax via *notation* directives,
- loads the *statevars* package,
- loads the `xsyntax package`, flagging it to identify loop syntax,
- loads the `xcontrol package`, and
- loads the `hiordx` package.

We will explore the role of each component throughout the chapter.

6.1 Syntax

The syntax predefined in the `loops package` is shown below. However, it is important to note that this syntax is only an example of how the compilation process (described in Section 6.3) can be used. Users can define or redefine the syntax using the *notation* directive, as explained in Section 2.1.4.

While loop: It has the following form:

```
while (Condition) {
    Body
}
```

This loop iterates while the condition is satisfied.

For loop: It has the following form:

```
for (I1 in Iterable1, I2 in Iterable2, ..., IN in IterableN) {
    Body
}
```

Each iterable can be:

- A list, e.g., `[X1, X2, ..., ZN]`, written as `'I in list([X1, X2, ..., ZN])'`.
- A range of the form `'First .. Step .. Last'`, where `Step` must be greater than 0.
- A shorthand range `'First .. Last'`, equivalent to `'First .. 1 .. Last'`.
- A term whose arguments will be iterated over.

This loop iterates simultaneously over the N iterables using the appropriate iterator for each. If the iterables have different lengths, the number of iterations corresponds to the length of the shortest iterable. The scope of the iterators `I1, I2, ..., IN` is local to each iteration, meaning their values cannot be accessed outside the loop.

In addition to these loops, a conditional statement has been added to improve code readability and clarity. It is implemented using cuts, so it may prune solutions if used improperly. The conditional has the following form:

```
if (Condition) {
    IfBody
} else {
    ElseBody
}
```

It can also be used without the `else` part, or with `else if` clauses instead of `else`.

6.2 Loop Definition

In this section, we explain the bridge between syntax and the compilation process for loops, as provided by the system. This bridge represents the transformation of a syntactic construct so that it can later be recognized by the compiler. By default, this extension provides syntax for the two types of loops mentioned in Section 6.1, but it can be customized by the user through the use of directives. To achieve this, we must take into account that every loop can be expressed in the following general form:

```
'$loop'(Vars, Init, Cond, Goal)
```

Where:

- *Vars:* the iteration variables, scoped locally to each iteration.
- *Init:* the initialization of the iteration variables with their starting values.
- *Cond:* the conditions evaluated at the beginning of each iteration, checking whether the loop should continue execution.
- *Goal:* the body of the loop, which also includes the update (i.e., change of state) of the iteration variables for the next iteration.

The extension uses ':- notation/2', which is explained in more detail in Section 2.1.4, to transform the syntax defined in Section 6.1 into predicate form. Using ':- notation/2' to assign meaning to loop representations facilitates the definition of new loop types. However, the challenge lies in finding a suitable translation into the general form. The interpretation of loops depends on the type of loop used, as we will explain in the following sections.

6.2.1 While Loop

The definition of a *while* loop using the notation directive is as follows:

```
:- notation(while (Cond) { Body },
            '$loop'([], true, Cond, Body)).
```

In this type of loop, some arguments of the internal loop representation are not used, specifically Vars and Init, because the loop does not involve iterators.

For example:

```
while_loop :-
    I = 1
    while (I < 6) {
        write(I),
        I := I + 1
    }.
```

⇓

```
'$loop'([], true, I < 6, (write(I), I := I + 1))
```

6.2.2 For Loop

Considering a *for* loop:

```
for (I in Iterable) {
  Body
}
```

Iterable represents an iterable object implemented using a generic iterator. To support this, a creation predicate must be defined for each specific type of iterable. Additionally, a generic iterator interface is provided for iterable objects. It consists of two predicates whose implementations depend on the specific type of iterable:

- *iter_cond(Curr, X)*: Checks whether the iteration condition is satisfied. If it succeeds, it sets the current state of the iterator variable in X; otherwise, it fails.
- *iter_next(Curr, Curr1)*: Transitions from Curr to Curr1, representing the next state for the following iteration.

In *for* loops using generic iteration, their definition with the *notation* directive is:

```
:- notation(for (I in Iter) { Body },
            '$loop'([I],
                    Curr := Iter,
                    iter_cond(Curr, I),
                    (Body, Curr := ~iter_next(Curr)))).
```

Additionally, the translation of an iterable can be defined directly, instead of using the generic iterator, by means of the *notation* directive. This slightly improves efficiency, as the translation is performed directly without the need to search through the generic iterator mechanism. Specifically, the package provides direct translations for some of the most commonly used iterables, namely those explained in Section 6.1.

Table 6.1 presents the syntax and the corresponding transformations for each of the provided iterables into the arguments of the primitive `'$loop'/4`.

Table 6.1: Iterables substitutions in `'$loop'/4`

	Lists	**Terms**	**Ranges**
Form	I in list(List)	I in args(T)	I in A .. Step .. B
Vars	I	I	I
Init	Curr := List	functor(T,_, N), Curr := 1	End = B, Curr := A
Cond	Curr = [I| _]	Curr =< N, arg(Curr,T,I)	I = Curr, Curr =< End
Goal	Body, Curr = [_I|Xs], Curr := Xs	Body, Curr := Curr + 1	Body, Curr := Curr + Step

Having seen this, the use of the *notation* directive without relying on the generic iterator, for a *for loop* using a list iterator, is as follows:

```
:- notation(for (I in list(List)) { Body },
            '$loop'([I],
                    Curr := List,
                    Curr = [I| \_],
                    (Body, Curr = [\_I|Xs], Curr := Xs))).
```

If we use that definition of a loop in an example, the translation into the primitive `'$loop'/4` term is:

```
for_loop :-
    for (I in list([1,2,3,4,5])) {
        write(I)
    }.
```

⇓

```
'$loop'(I, List := [1,2,3,4,5], List = [I|ListS],
        (List = [X|Xs], List := Xs, write(I)))
```

6.3 Compilation Rules

State is a fundamental aspect of loops, as each iteration typically changes the values of variables, thereby altering the program's state. To handle this, loops rely on the compilation of state variables, as described in Section 4.2, by loading the `statevars package`.

However, the approach is not exactly the same: it generalizes the concept to multiple scopes, executing its second compilation phase for each scope, as will be explained later.

More specifically, the loop compilation process consists of three main phases, which act as a pipeline: the output of each phase serves as the input for the next. To illustrate this sequence of steps, we will focus on the following loop example:

```
loop_example :-
    S = 0,
    M = 2,
    for (I in 1 .. 5, J in list([9,8,7,6,5])) {
        A := I + J ,
        if (A mod M == 0) {
            S := S + 1
        } else {
            S := S - 1
        }
    }.
```

Now, we will explain the compilation process up to the point where loops are fully translated into plain `Ciao` predicates:

6.3.1 First Phase

This first phase is performed by the `xsyntax package`, explained in Section 2.1.2, and it is similar to the parsing process described in Section 4.2.1. It consists of detecting certain patterns and translating them into normalized internal representations that can be processed by subsequent compilation phases.

In the case of loops, several flags are activated to detect multiple patterns. Specifically, it activates the state variables flag (enabled by the `statevars package`)

and the loops flag (enabled by the `loops package`). The state variables flag is necessary for compiling loops because, even if state variables are not used explicitly, the control structure of loops implies state changes, as the iterator takes on different values in each iteration.

This phase includes three main translations, explained below.

Two of these translations correspond to the state variables flag and are detailed in Section 4.2.1. In general terms:

- The first translation replaces any expression of the form `!X` used as an argument in a functor with the pair of arguments $X_\circ$ and $X_\bullet$.
- The second translation transforms clauses of the form `X := Y` into the normalized clause `assign(X, Y)`.
- The third translation handles the loop syntax described in Section 6.1, using the loop definitions presented in Section 6.2.

To reach the primitive representation, the loops are processed according to the definitions established via the *notation* directive. At this point, all loops must be translated into the primitive form:

```
'$loop'(Vars, Init, Cond, Goal)
```

Turning back to our example[1] after this compilation phase, it would be transformed to:

```
loop_example :-
  S = 0,
  M = 2,

  '$loop'(
    (I,J),
    (End = 5, assign(C,1), assign(L,[9,8,7,6,5])),
    (I = C, C =< End, L = [J|_Rest]),
    (Tmp3 is I + J, assign(A,Tmp3),
      ((Tmp4 is A mod M, Tmp4 == 0) ->
          (Tmp5 is S + 1, assign(S,Tmp5))
          ; (Tmp6 is S - 1, assign(S,Tmp6))
      ),
      Tmp1 is C + 1, Tmp2 is Tmp1, assign(Curr,Tmp2),
      L = [_V| LS], assign(L,LS),
    )
  ).
```

[1]Note that the example uses the specific *notation* directive for *for loops* with ranges and lists, respectively, without using the generic iterator.

6.3.2 Second Phase

This second phase is performed by the package specifically created for the extensions explained in this project, called `xcontrol`. It mainly consists of transforming loops, which already have the form `'$loop'(Vars, Init, Cond, Goal)`, into closures with negative sharing. This process is carried out as follows:

1. First, it parses the entire `'$loop'/4` structure and generates a list containing all the state variables found within it. A state variable is considered to be any variable that appears as the first argument of the `assign/2` predicate or has a representation of its next state as a variable in the form `X.`

   Having generated this list, the structure is transformed from `'$loop'/4` into the following type:

   ```
   '$shloop'(Vars, StLoopVars, Init, Cond, Goal)
   ```

   All arguments except `StLoopVars` are exactly the same as those in the original `'$loop'/4` structure. The `StLoopVars` argument corresponds to the list of state variables generated previously. In our example, this will result in:

   ```
   loop_example :-
     S = 0,
     M = 2,

     '$shloop'(
       (I,J),
       [S,A,L,C],
       (End = 5, assign(C,1), assign(L,[9,8,7,6,5])),
       (I = C, C =< End, L = [J|_Rest]),
       (Tmp3 is I + J, assign(A,Tmp3),
         ((Tmp4 is A mod M, Tmp4 == 0) ->
             (Tmp5 is S + 1, assign(S,Tmp5))
             ; (Tmp6 is S - 1, assign(S,Tmp6))
         ),
         Tmp1 is C + 1, Tmp2 is Tmp1, assign(C,Tmp2),
         L = [_V| LS], assign(L,LS),
       )
     ).
   ```

2. At this point, we need to transform the `'$shloop'/5` structure into closures, as explained in Section 2.1.3, since closures are suitable for defining a specific scope that can capture variables from the parent scope. The general schema for loops is:

   ```
   <<begin>> :- Init, <<body>>.
   <<body>> :-
     ( Cond ->
         Goal, <<body>>
     ; true ).
   ```

Where <<begin>> and <<loop>> are closures that use negative sharing explained in Section 2.1.3, meaning all variables are captured from the parent environment except for the ones specified in the list '-[NonShVars] ->' . For getting there, we have to introduce intermediate steps to manage state variables:

2.1. We extend the list StLoopVars, the argument of '$shloop'/5 that contains all the state variables in the loop, such that each of the elements extends to the pair that represents state, $X_{\circ}$ and $X_{\bullet}$. For example, 'StLoopVars = [X,Y]' would be extended to $[X_{\circ},X_{\bullet},Y_{\circ},Y_{\bullet}]$. This new list is used as the arguments of the <<begin>> and <<loop>> closures to be able to bring the resulting state of a iteration or the parent scope into the next iteration. Additionally, the argument Vars of '$shloop'/5, which contains the list of variables that are local to each iteration of the loop (iterators), should be included as the list of non sharing variables on the <<loop>> closure.

   In the scheme above we could appreciate <<begin>> is the entry point of the loop that, after the initialization, calls to <<body>>. The <<body>> closure contains the condition such that if it accomplished it executes Goal and calls again <<body>> to execute another iteration. On the other hand, if the condition does not succeed, the loop ends with the true directive to continue executing the rest of the clause. Therefore, in order to start the loop we should add an additional term to the clause such that it invokes it, using the call/n predicate, specifically:

```
call(Begin,Arguments)
```

   The schematic representation at this point of the compilation process is:

```
Begin = {
  ''(!X1, ..., !Xn) :-
    Init,
    Body = {
      -[Vars] -> ''(!X1, ..., !Xn) :-
        ( Cond -> Goal, Body(!X1, ..., !Xn) ; true )
    },
    Body(!X1, ..., !Xn)
},
Begin(!X1, ..., !Xn).
```

Our example at this point will be:

```
loop_example :-
  S = 0,
  M = 2,
  Begin = {
    % Closure Arguments
    ''(S∘,S•,A∘,A•,L∘,L•,C∘,C•) :-
      % Init
      (End = 5, assign(C,1), assign(L,[9,8,7,6,5])),
      Body = {
        % Negative sharing
        -[I,J] ->
        % Closure Arguments
        ''(S∘,S•,A∘,A•,L∘,L•,C∘,C•) :-
          % Condition
          ((I = C, C =< End, L = [J|_Rest]) ->
              % Goal
              (Tmp3 is I + J, assign(A,Tmp3),
                ((Tmp4 is A mod M, Tmp4 == 0) ->
                    (Tmp5 is S + 1, assign(S,Tmp5))
                    ; (Tmp6 is S - 1, assign(S,Tmp6))
                ),
                Tmp1 is C + 1, Tmp2 is Tmp1,
                   assign(C,Tmp2),
                L = [_V|LS], assign(L,LS)
              ),
              % Call body for the next iteration
              call(Body,S∘,S•,A∘,A•,L∘,L•,C∘,C•)
              % If the loop ends
              ;true
          )
        },
        % Call body for the first iteration
        call(Body,S∘,S•,A∘,A•,L∘,L•,C∘,C•)
  },
  % Initiate the loop
  call(Begin,S∘,S•,A∘,A•,L∘,L•,C∘,C•).
```

2.2. After this the whole structures are parsed executing the steps defined in Section 4.2 to handle state variables or even executing the steps in Section 6.3 if there are nested loops. In this step we need to take into account that each of the closures act as a different scope, then they can be treated as a separate basic blocks or clauses. Therefore the compilation process explained in Section 4.2 can be applied to it as a individual clause or scope, except on this case the predicate is unnamed instead of named. Once done that parsing, it includes the variables in the arguments of the head of the closures in the list of non

sharing variables. This ensures that the value of each state variable in each iteration is taken from the arguments and not from the parent environment.

At this point, all occurrences of the `assign/2` predicate and all arguments of the form $X_\circ$ or $X_\bullet$ should have been correctly processed and replaced with their corresponding terms. After this compilation phase, the translation of our example is:

```
loop_example :-
  S = 0,
  M = 2,
  Begin = {
    % Negative Sharing
    - [S0,Sn,A0,An,L0,Ln,C0,Cn] ->
    % Closure Arguments
    ''(S0,Sn,A0,An,L0,Ln,C0,Cn):-
      % Init
      (End = 5, C1 = 1, L1 = [9,8,7,6,5]),
      Body = {
        % Negative Sharing
        - [I,J,S_0,S_n,A_0,A_n,L_0,L_n,C_0,C_n] ->
        % Closure Arguments
        ''(S_0,S_n,A_0,A_n,L_0,L_n,C_0,C_n) :-
        % Condition
          ((I = C_0, C_0 =< End, L_0 = [J|_Rest]) ->
               % Goal
               (Tmp3 is I + J, A_1 = Tmp3,
                 ((Tmp4 is A_1 mod M, Tmp4 == 0) ->
                     (Tmp5 is S_0 + 1, S_1 = Tmp5)
                     ; (Tmp6 is S_0 - 1, S_1 = Tmp6) [2]
                 ),
                 Tmp1 is C_0 + 1, Tmp2 is Tmp1, C_1 = Tmp2,
                 L_0 = [_V| LS], L_1 = LS
               ),
               % Call body for the next iteration
               call(Body,S_1,S_n,A_1,A_n,L_1,L_n,C_1,C_n)
               % If the loop ends
               ; (S_n = S_0, A_n = A_0, L_n = L_0, C_n = C_0)
          )
      },
      % Call body for the first iteration
      call(Body,S0,Sn,A0,An,L1,Ln,C1,Cn)
  },
  % Initiate the loop
  call(Begin,S,S__n,A__0,A__n,L__0,L__n,C__0,C__n).
```

[2] Note that there is a disjunction and, as explained in Section 4.2.2, it unifies the environments, specifically the variable S_1.

6.3.3 Third Phase

The third phase consists of the correct treatment of the negative sharing closures, which is performed by the *hiordx* package and the compiler itself. Although it is already part of another package, as explained in Section 2.1.3, we are going to go slightly over it to finish in this project the transformation from loops into plain predicates. This process consists of three main steps:

1. The first part consists on transforming negative sharing closures into positive sharing closures, which is done done by *hiordx* package. It takes as an input the negative sharing clauses from loops, but it can also be used directly by the user as we mentioned in Section 2.1.3.

   The process consists of parsing the closure scope by scope, annotating for each variable the line in which it appears. Then for each variable it checks:

   - The variable is not part of the excluded variables in negative sharing.
   - The variable appears in more than one scope.

   If both conditions are met, the variable is included inside the list of sharing variables across all relevant closures, ensuring it can be accessed from one occurrence to another. For example, if we have a set of closures '`Scope 0 {Scope 1 {Scope 2}}`' with negative sharing and a variable 'A' that appears in 'Scope 0' and 'Scope 2', then the variable 'A' would be added to the list of shared variables of 'Scope 1' and 'Scope 2'.

   Turning back to our example, after this compilation step it would be transformed to:

```
loop_example :-
  S = 0,
  M = 2,
  Begin = {
    % Positive Sharing
    [M] ->
    % Closure Arguments
    ''(S0,Sn,A0,An,L0,Ln,C0,Cn):-
      % Init
      (End = 5, C1 = 1, L1 = [9,8,7,6,5]),
      Body = {
        % Negative Sharing
        [End,M,Body] ->
        % Closure Arguments
        ''(S_0,S_n,A_0,A_n,L_0,L_n,C_0,C_n) :-
        % Condition
          ((I = C_0, C_0 =< End, L_0 = [J|_Rest]) ->
              % Goal
              (Tmp3 is I + J, A_1 = Tmp3,
                ((Tmp4 is A_1 mod M, Tmp4 == 0) ->
                    (Tmp5 is S_0 + 1, S_1 = Tmp5)
```

```
                ; (Tmp6 is S_0 - 1, S_1 = Tmp6)
            ),
            Tmp1 is C_0 + 1, Tmp2 is Tmp1, C_1 = Tmp2,
            L_0 = [_V| LS], L_1 = LS
          ),
          % Call body for the next iteration
          call(Body,S_1,S_n,A_1,A_n,L_1,L_n,C_1,C_n)
          % If the loop ends
          ; (S_n = S_0, A_n = A_0, L_n = L_0, C_n =
            C_0)
        )
    },
    % Call body for the first iteration
    call(Body,S0,Sn,A0,An,L1,Ln,C1,Cn)
},
% Initiate the loop
call(Begin,S,S__n,A__0,A__n,L__0,L__n,C__0,C__n).
```

It is important to highlight that the variable 'Body', which is the reference to the closure that contains the body of the loop, is also included in the list of shared variables of itself as it has to call itself for the next iteration.

2. At this point we have a set of positive sharing closures nested one inside an other or consecutively. This leads to a inefficient execution due to the complexity of executing closures one after another, even for each iteration, without treating them during compile time. To avoid this, we flatten closures into plain predicates by giving them an internal name to turn unnamed predicates into common plain predicates. This leads to a execution complexity equivalent to the one of the commonly used recursion in `Prolog` and materializes as a better time performance.

   More specifically, if we perform the optimization over a simple example of a closure where none of the variables from the parent scope are shared, except the ones explicitly included in the list, and it is call from a module:

```
foo(P) :- Y = 2, P = {[Y] -> ''(X) :- X = 1 + Y}.
```

   We take the closure and give it an auxiliary name whose argument are the shared variables, in this case is `Y`. However, we can see that the argument of the unnamed predicate was `X` and it has to be an argument as well. Therefore, the arity of the auxiliary predicate created is $Arity = N^{\circ}\ Shared\ Variables + N^{\circ}\ Closure\ Arguments$. Following the example, the translation after the optimization is called the same way and would be:

```
foo_tr(P) :- Y = 2, P = foo_tr_aux(Y).
foo_tr_aux(Y,X) :- X = 1 + Y.
```

Moving back to our example, after this transformation it would be:

```
loop_example :-
  S = 0,
  M = 2,
  Begin = aux_begin(M),
  % Initiate the loop
  call(Begin,S,S__n,A__0,A__n,L__0,L__n,C__0,C__n).

aux_begin(M,S0,Sn,A0,An,L0,Ln,C0,Cn) :-
  % Init
  (End = 5, C1 = 1, L1 = [9,8,7,6,5]),
  Body = aux_body(End,M,Body),
  % Call body for the first iteration
  call(Body,S0,Sn,A0,An,L1,Ln,C1,Cn).

aux_body(End,M,Body,S_0,S_n,A_0,A_n,L_0,L_n,C_0,C_n) :-
  % Condition
  ( (I = C_0, C_0 =< End, L_0 = [J|_Rest]) ->
      % Goal
      (Tmp3 is I + J, A_1 = Tmp3,
        ((Tmp4 is A_1 mod M, Tmp4 == 0) ->
            (Tmp5 is S_0 + 1, S_1 = Tmp5)
            ; (Tmp6 is S_0 - 1, S_1 = Tmp6)
        ),
        Tmp1 is C_0 + 1, Tmp2 is Tmp1, C_1 = Tmp2,
        L_0 = [_V| LS], L_1 = LS
      ),
      % Call body for the next iteration
      call(Body,S_1,S_n,A_1,A_n,L_1,L_n,C_1,C_n)
      % If the loop ends
      ; (S_n = S_0, A_n = A_0, L_n = L_0, C_n = C_0)
    ).
```

It is worth noting that, for clarity in the example, we placed the shared variables as the first arguments of the internal plain predicates. However, the compiler actually inserts them at the end. Placing shared variables last is a design choice aimed at taking advantage of Ciao's predicate indexing system, though the ordering could be adjusted based on other criteria.

3. The final step of this compilation phase involves transforming dynamic calls (using the `call/n` predicate) into static calls. This is done by removing the references to `Body` and `Begin`, so that the call used for loop iterations becomes a standard Prolog call to a predicate with arity `n`. In this step, the `Body` variable should also be removed from the arguments of the auxiliary body predicate, as it is no longer needed for calling.

In our example, after the entire compilation process, the resulting translation is [3]:

```
loop_example :-
  S = 0,
  M = 2,
  % Initiate the loop
  aux_begin(M,S,S__n,A__0,A__n,L__0,L__n,C__0,C__n).

aux_begin(M,S0,Sn,A0,An,L0,Ln,C0,Cn) :-
  % Init
  (End = 5, C1 = 1, L1 = [9,8,7,6,5]),
  % Call body for the first iteration
  aux_body(End,M,S0,Sn,A0,An,L1,Ln,C1,Cn).

aux_body(End,M,S_0,S_n,A_0,A_n,L_0,L_n,C_0,C_n) :-
  % Condition
  ( (I = C_0, C_0 =< End, L_0 = [J|_Rest]) ->
      % Goal
      (Tmp3 is I + J, A_1 = Tmp3,
        ((Tmp4 is A_1 mod M, Tmp4 == 0) ->
            (Tmp5 is S_0 + 1, S_1 = Tmp5)
            ; (Tmp6 is S_0 - 1, S_1 = Tmp6)
        ),
        Tmp1 is C_0 + 1, Tmp2 is Tmp1, C_1 = Tmp2,
        L_0 = [_V| LS], L_1 = LS
      ),
      % Call body for the next iteration
      aux_body(End,M,S_1,S_n,A_1,A_n,L_1,L_n,C_1,C_n)
      % If the loop ends
      ; (S_n = S_0, A_n = A_0, L_n = L_0, C_n = C_0)
    ).
```

[3] Note that, as with state variables, if this is copied into a module, S will produce the same result as the original code, since both are equivalent.

As explained in Chapter 1, the lack of imperative constructs made it difficult to implement existing algorithms written in pseudocode. Specifically, we highlighted the challenges and low readability of implementing the *Eratosthenes Sieve* in Ciao, whereas in Python it could be implemented and understood much more easily. With the imperative constructs introduced throughout this project, the implementation can now be expressed as follows:

```
:- module(_, _, [loops,functional,arrays]).
:- use_module(library(arrays/arrays_log)).

primes_loops(N) := Res :-
    new_array_log(A),
    for (I in 2 .. N) {
        A[I] := 'true'
    },

    Sqrt is ~floor(~sqrt(N)),
    for (I in 2 .. Sqrt) {
        if (A[I] == 'true') {
            for (J in I*I .. I .. N) {
                A[J] := 'false'
            }
        }
    },

    Res = ResTail,
    for (K in 2 .. N) {
        if (A[K] == 'true') {
            accum(!ResTail,K)
        }
    },
    ResTail = [].

accum(!R, X) :- R = [X|Tail], R := Tail.
```

6.4 Other Implemented Loop Types

The loop schema used above includes a conditional, the -> operator, which introduces a *cut*. This means that loops cannot run "backwards," i.e., they do not perform backtracking on the condition. However, including loops that allow this behavior can be useful in certain use cases, such as program analysis.

To address this issue, the extension supports the definition of condition pairs, using the syntax *posneg(PosCond, NegCond)*. These pairs are processed to implement the following loop schema:

```
<<begin>> :- Init, <<body>>.
<<body>> :-
  ( NegCond ;
    PosCond,  Goal, <<body>>
  ).
```

Therefore, the compilation process will follow the same steps as explained in Section 6.3, but it will generate this new schema instead of the one containing the cut.

Chapter 7

Experimental Results

Once all the extensions have been implemented we should test that they work as expected. In order to achieve this, we have taken a set of examples that were already coded in `Picat`, which, as mentioned in Section 2.2, is a language (different from Prolog) that has loops, state variables, and array index notation. It is also known to be quite performant, and thus we consider it a very good point of reference. More specifically, we have chosen to translate the implementation of most of the first Euler Project problems which could be found in the `Picat` official website. These examples are then used to test the of use array-like structures, state variables, and loops explained in this project, as well as existing `Ciao` extensions like *clpfd*, *tabling*, and *assoc*.[1]

In the following we present the execution results for each problem, grouped in different tables by the extensions or utilities they use. All these examples run correctly,[2] producing identical results in both languages, and we have chosen to measure the execution time and the speedup or slowdown of each in `Ciao` with respect to `Picat`. Our objective with this is not to perform an in-depth performance comparison, since both the implementation and the problem encodings can still be improved, but rather to have an estimation of whether the approach is competitive.

[1] The examples are too many to include in the appendices but the can be found in this repo: `https://gitlab.software.imdea.org/ciao-lang/ciaoimp-benchmarks`

[2] Except for number 23 that runs out of memory due to the strong use of tabling. Testing it for smaller input numbers it does return the correct results.

Table 7.1: Examples that cannot be compared.

File	Picat Time (s)	Ciao Time (s)	Speed-up
p001	0.000	0.000176	-
p002	0.000	0.000085	-
p003	0.000	0.000159	-
p005	0.000	0.000016	-
p006	0.000	0.000018	-
p013	0.000	0.000032	-
p015	0.000	0.001113	-
p028	0.000	0.000329	-
p033	0.000	0.000037	-
p079	0.000	0.000350	-
p100	0.000	0.000013	-

In Table 7.1 we show a first group of examples, whose execution results are correct, but cannot be compared because the execution times are really too low to be significant.

Table 7.2: Examples that just use loops.

File	Picat Time (s)	Ciao Time (s)	Speed-up
p004	0.750	0.102908	7.288063
p007	0.299	0.172652	1.731807
p012	11.250	6.975017	1.612899
p016	0.001	0.000163	6.134969
p020	0.001	0.000076	13.157895
p022	0.021	0.021922	0.957942
p025	3.392	2.117667	1.601763
p029	2.353	2.288128	1.028352
p030	1.114	2.220842	0.501612
p034	0.403	0.289445	1.392320
p036	1.250	0.841712	1.485069
p045	0.043	0.030968	1.388530
p046	0.041	0.027016	1.517619
p048	0.826	0.017242	47.906275
p056	0.781	0.210811	3.704740

Table 7.2 presents examples that use only loops, including state variables, without any other extensions. These examples demonstrate that `Ciao`'s implementation of state variables and loops is competitive with `Picat`'s. We observe some performance differences, which may be due to variations in the virtual machine implementations.

Table 7.3: Examples that use loops and tabling.

File	Picat Time (s)	Ciao Time (s)	Speed-up
p021	0.173	0.137526	1.257944
p027	1.823	4.482436	0.406699
p037	3.029	4.278332	0.707986
p041	0.012	0.014385	0.834202
p053	0.041	0.036864	1.112196
p055	0.073	0.070912	1.029445

Table 7.3 presents examples that use tabling, showing similar performance overall, with some slight advantage overall for Picat. Comparing this with Table 7.2, we can infer that the difference is likely to be because of variations in the tabling implementation.

Table 7.4: Examples that use loops and index notation in lists.

File	Picat Time (s)	Ciao Time (s)	Speed-up
p017	0.011	0.016534	0.665296
p019	0.051	0.050183	1.016280
p024	1.150	8.596679	0.133773
p026	0.074	1.946955	0.038008
p040	0.197	0.161726	1.218110
p042	0.025	0.030145	0.829325
p060	142.493	91.048326	1.565026
p076	0.003	0.057377	0.052286
p077	0.013	0.204024	0.063718

Table 7.4 presents examples that use loops and index notation over lists from the new array extension, explained in Chapter 5.1. The performance is generally similar in both languages, with some exceptions such as p024, p026, p077, and p062. These are likely to be due to the fact that replacing an element in a list is destructive in Picat but not in Ciao.

Table 7.5: Examples that use loops and index notation in arrays.

File	Picat Time (s)	Ciao Time (s)	Speed-up
p010_log	-	9.778831	
p010_mut	0.909	0.933479	0.973777
p047_log	-	10.553734	
p047_mut	0.804	1.184987	0.678488
p050_log	0.282	1.909871	0.147654

Table 7.5 presents examples that use loops and index notation over arrays from the new array extension, explained in Chapter 5.1. Specifically, they are imple-

mented in Ciao using both logarithmic and mutable arrays, distinguished by the last part of the file name. Regarding the results, performance in mutable arrays is similar to Picat's, as Picat's array implementation is also mutable. However, as expected, in logarithmic arrays Ciao has a logarithmic order of magnitude higher performance.

Table 7.6: Examples that use loops and other extensions.

File	**Picat Time (s)**	**Ciao Time (s)**	**Speed-up**
assoc			
p032	2.398	2.005475	1.195727
p044	1.465	2.458069	0.595996
p062	0.032	0.168411	0.190011
pmrule			
p052	0.418	0.538781	0.775825
p049	0.138	0.113220	1.218866

Table 7.6 presents examples that use loops and two different extensions. First, the examples using *assoc*, a Ciao extension that implements maps, show that Picat is generally faster and this is likely because replacing an element in a map is destructive in Picat but not in Ciao. Second, the examples that include *pmrule*, the extension explained in Chapter 3, have similar performance.

Chapter 8

Impact Analysis

Integrating imperative notation into a declarative logic-programming language shortens the gap with widely used languages like Python. This approach makes it easier for programmers to adopt the logic-programming paradigm without abandoning familiar imperative constructs such as loops. Additionally, it allows developers to implement existing algorithms directly from pseudocode without having to convert them into recursive versions, which leads to more readable and debuggable code. Moreover, it simplifies state management by eliminating the need to rely on predicate arguments to handle current and future states throughout iterations. Since Ciao is often introduced to university students with no experience in declarative programming, imperative notation provides a smoother transition, as they usually have a background in languages like Java.

On the other hand, this approach could have a negative impact, as it might feel like loosing the declarative essence of Prolog languages. It could discourage learners from understanding recursion, since they may rely on the new imperative constructs to handle iterations instead. Additionally, it might dilute basic concepts of declarative programming, such as backtracking or recursive iteration.

Looking into the Agenda 2030, the sustainable development goals which are more related to the project are Goal 9, in terms of innovation in technology, and Goal 4, in terms of facilitating education. This projects constitutes an innovation in the logic-paradigm, extending a Prolog-based language to allow imperative constructs making more notable its multi-paradigm characteristic. The result of the project also facilitates learning logic programming, by making it closer to more popular languages and reducing the learning curve.

Chapter 9

Conclusions and Future Work

Although the *logic programming* paradigm is quite powerful, it natively lacks some imperative features that can simplify the implementation of certain algorithms. Moreover, most programmers begin by learning imperative languages and tend to think in terms of imperative constructs when designing and coding, which makes them feel more comfortable.

In this work, we have provided a flexible and elegant way to include imperative features in *logic programs*, such as loops, state variables, and array index notation. We believe this will potentially contribute to the broader adoption of *declarative languages* in general and help close the gap with popular imperative languages, such as `Python`. In particular, we have proposed a number of imperative-style constructs that build upon and extend the `FSyntax` and `Hiord` syntactic extensions to `Prolog`. Furthermore, we also provide a single-sided unification extension that enables pattern matching.

The proposed extensions have been designed so that they combine well with the basic functional notation and the higher-order facilities as well as with other extensions, such as constraints, tabling, etc. In addition, our approach is based on a set of primitives and a simplified, higher-level expansion mechanism that together have allowed us to easily introduce the imperative features mentioned above.

Specifically, we have introduced two new mechanisms in *xcontrol*:

- Management of state variables through an environment.
- Transformation of loops into negative-sharing closures.

These mechanisms, combined with the *notation* directive, open up a wide range of possibilities for extending the language, not only with the syntax proposed in this project but also with other types of loops and imperative constructs.

We have demonstrated how basic state variable management can serve as a foundation for additional extensions. In this project, for instance, we implemented array index notation for array-like structures using state variables in operations that modify specific elements (e.g., the nth element of an array).

Furthermore, to evaluate the proposed mechanisms, we have implemented the proposed mechanisms by defining a set of imperative features and exercising their usefulness by translating idiomatically, in imperative style, a large collection of small but interesting programs from the *Euler Project*. Apart from their intrinsic interest, the choice of these benchmarks was motivated by the fact that encodings of many of these problems have been done in a number of imperative languages, and in particular a good number of them are available for the `Picat` language, which, as we have argued, is a very good point of reference. We have also studied the performance of the translated programs.

While some imperative-style constructs were previously available in some form or another in some `Prolog` systems, and more comprehensively in non-`Prolog` systems like `Picat`, we argue that our Prolog-based proposal is comprehensive, coherent, and extensible, as well as offering competitive performance.

Also, we have illustrated that the lack of imperative constructs complicates the implementation of well-known algorithms described in pseudocode. Specifically, we highlighted the difficulty and reduced readability involved in implementing the *Sieve of Eratosthenes* in `Ciao` compared to a more straightforward and recognizable implementation in `Python`. We then presented an implementation using the proposed extensions, which closely resembles the pseudocode and improves both readability and understandability.

Overall, the proposed extensions advance the state-of-the-art in the *logic programming* paradigm, particularly in the context of the `Ciao` multi-paradigm system. These enhancements increase its potential for practical use and facilitate the development of programs that require state representation. Moreover, our approach achieves this without departing from the core principles of logic programming, demonstrating that introducing imperative constructs does not require a new language.

Future lines of work include:

- Studying the implementation of array index access with improved performance and introducing it as a backend implementation for the `arrays` interface.
- Integrating the extensions developed in this project into the `Ciao` static analyzer.
- Performing a deeper analysis of the *Euler Project* problem set to identify the strengths and limitations of the proposed extensions and improve them accordingly.
- Comparing the execution times of the *Euler Project* problems with implementations in other programming languages beyond `Picat`.

Bibliography

[1] L. Rajapakshe. (2025) Understanding the imperative programming paradigm in software development. [Online]. Available: https://medium.com/@lahirurajapakshe.stack/understanding-the-imperative-programming-paradigm-in-software-development-ec04a4ee5f02

[2] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, and G. Puebla, "An Overview of Ciao and its Design Philosophy," *Theory and Practice of Logic Programming*, vol. 12, no. 1–2, pp. 219–252, January 2012. [Online]. Available: http://arxiv.org/abs/1102.5497

[3] C. Andersen and T. Swift, "The Janus System: A Bridge to New Prolog Applications," in *Prolog - The Next 50 Years*, ser. LNCS, D. S. Warren, V. Dahl, T. Eiter, M. Hermenegildo, R. Kowalski, and F. Rossi, Eds. Springer, July 2023, no. 13900.

[4] N.-F. Zhou, "Picat: A scalable logic-based language and system," in *2nd Symposium on Languages, Applications and Technologies*, ser. Open Access Series in Informatics (OASIcs), vol. 29. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013, pp. 5–6. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.SLATE.2013.5

[5] The Picat Team, *Picat Guide*, 2025. [Online]. Available: https://picat-lang.org/download/picat_guide.pdf

[6] A. Casas, D. Cabeza, and M. V. Hermenegildo, "A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems," in *The 8th International Symposium on Functional and Logic Programming (FLOPS'06)*, Fuji Susono (Japan), 04 2006, pp. 142–162.

[7] D. Cabeza, M. V. Hermenegildo, and J. Lipton, "Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction," in *Ninth Asian Computing Science Conference (ASIAN'04)*, ser. LNCS, no. 3321. Springer-Verlag, December 2004, pp. 93–108.

[8] T. Software, "Tiobe index," 2025. [Online]. Available: https://www.tiobe.com/tiobe-index/

[9] M. O'Neill, "The genuine sieve of eratosthenes," *J. Funct. Program.*, vol. 19, pp. 95–106, 01 2009.

[10] W. contributors. Sieve of eratosthenes. [Online]. Available: https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

[11] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "MiniZinc: Towards a standard CP modelling language," in *In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming*. Springer, 2007, pp. 529–543.

[12] The MiniZinc Team, *MiniZinc Documentation*. [Online]. Available: https://docs.minizinc.dev/en/stable/index.html

[13] J. Schimpf, "Logical loops," in *In Proceedings of the 18th International Conference on Logic Programming, ICLP 2002*. SpringerVerlag, 2002, pp. 224–238.

[14] M. Carlsson and P. Mildner, "SICStus Prolog – the First 25 Years," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 35–66, 2012.

[15] T. Swift and D. S. Warren, "XSB: Extending Prolog with Tabled Logic Programming," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 157–187, Jan 2012.

[16] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[17] V. Santos Costa, R. Rocha, and L. Damas, "The YAP Prolog system," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 5–34, 2012.

[18] F. Fages, "A constraint-based mathematical modeling library in prolog with answer constraint semantics," 2024. [Online]. Available: https://arxiv.org/abs/2402.17286

[19] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia, "Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)," *Science of Computer Programming*, vol. 58, no. 1–2, pp. 115–140, October 2005.

[20] D. Cabeza and M. Hermenegildo, "The ciao module system documentation," 2000. [Online]. Available: https://ciao-lang.org/ciao/build/doc/ciao.html/modules.html#module/3

[21] D. Cabeza and M. V. Hermenegildo, "A New Module System for Prolog," Facultad de Informática, UPM, Technical Report CLIP8/99.0, 09 1999.

[22] J. F. Morales, M. V. Hermenegildo, and R. Haemmerlé, "Modular Extensions for Modular (Logic) Languages," in *Proceedings of the 21th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR'11)*, ser. LNCS, vol. 7225. Odense, Denmark: Springer, 07 2011, pp. 139–154.

[23] D. Cabeza, A. Casas, M. Hermenegildo, and J. F. Morales, *Functional Notation in the Ciao System*, 2024. [Online]. Available: https://ciao-lang.org/ciao/build/doc/ciao.html/fsyntax_doc.html#28

[24] R. Kowalski, F. Sadri, M. Calejo, and J. Dávila Quintero, *Combining Logic Programming and Imperative Programming in LPS*, 06 2023, pp. 210–223.

[25] I. O. for Standardization, *Information technology — Programming languages — Prolog — Part 1: General core*, 2017. [Online]. Available: https://www.iso.org/standard/73194.html

[26] J. Wielemaker, *SWI-Prolog – Single Sided Unification Rules*. [Online]. Available: https://www.swi-prolog.org/pldoc/man?section=ssu

[27] P. V. Roy, “A Useful Extension to Prolog’s Definite Clause Grammar Notation,” *ACM SIGPLAN Notices*, vol. 24, no. 11, pp. 132–134, November 1989.

[28] M. Project, *The Mercury Language Reference Manual: State Variables*. [Online]. Available: https://mercurylang.org/information/doc-release/mercury_ref/State-variables.html

[29] C. Dictionary. Loop - definition. [Online]. Available: https://dictionary.cambridge.org/dictionary/english/loop

[30] M. Consortium, *Loop Documentation in Mozart Programming Language*, 2004. [Online]. Available: http://mozart2.org/mozart-v1/doc-1.4.0/loop/

[31] J. Schimpf, “Logical loops,” in *International Conference on Logic Programming*, 2002. [Online]. Available: http://eclipseclp.org/reports/loops.pdf

[32] J. Wielemaker, *SWI-Prolog – Block Operators*. [Online]. Available: https://www.swi-prolog.org/pldoc/man?section=ext-blockop

[33] C. Team. Arrays in ciao. [Online]. Available: https://ciao-lang.org/ciao/build/doc/ciao.html/arrays.html

Appendices

Appendix A

Sieve of Eratosthenes: Implementations in (Ciao) Prolog

```
:- module(_,_,[functional]).

primes(Limit) := ~sift(~integers(2, Limit)).

integers(Low, High) := ( Low =< High ? [Low | integers(Low+1,High)] | [] ).

sift(L) := ( L=[]      ? []
           | L=[I|Is] ? [I|sift(~remove(Is,I))]).

remove(L,P) := ( L=[]      ? L
               | L=[I|Is] ? (I mod P =\= 0 ? [I|remove(Is,P)] | remove(Is,P))).
```

Figure A.1: Classical declarative example for computing primes (using Ciao fsyntax).

```
:- module(_, _, []).
:- use_module(library(logarrays)).

primes(N,Res) :-
    new_array(A), % Initialize an extendable array
    To is floor(sqrt(N)), % Just need to go to sq root of n
    complete_sieve(2,To,N,A,CompleteSieve), % Complete the sieve
    % Create a list with the primes
    take_primes(2,N,CompleteSieve,Res).

complete_sieve(Curr,To,_N,Sieve,RSieve):-
    Curr > To, !, RSieve = Sieve.
complete_sieve(Curr,To,N,Sieve,RSieve) :-  % If it is marked (0) it is not prime
    aref(Curr,Sieve,_El), !,
    NewCurr is Curr + 1,
    complete_sieve(NewCurr,To,N,Sieve,RSieve).
complete_sieve(Curr,To,N,Sieve,RSieve) :-  % Gets here if it is not marked (0)
    From is Curr * Curr,
    set_multiples(From,Curr,N,Sieve,Sieve1), % Mark with 0 the multiples of a prime
    NewCurr is Curr + 1,
    complete_sieve(NewCurr,To,N,Sieve1,RSieve).

set_multiples(Curr,_Step,To,Sieve,RSieve) :-
    Curr > To, !, RSieve = Sieve.
set_multiples(Curr,Step,To,Sieve,RSieve) :-
    aset(Curr,Sieve,0,Sieve1),
    NewCurr is Curr + Step,
    set_multiples(NewCurr,Step,To,Sieve1,RSieve).

take_primes(Curr,N,_Sieve,Res) :-
    Curr > N, !, Res = [].
take_primes(Curr,N,Sieve,Res) :- % If it is marked (0) it is not prime
    aref(Curr,Sieve,_El), !,
    NewCurr is Curr + 1,
    take_primes(NewCurr,N,Sieve,Res).
take_primes(Curr,N,Sieve,Res) :- % Not marked (0): add it to Res
    Res = [Curr| Rest],
    NewCurr is Curr + 1,
    take_primes(NewCurr,N,Sieve,Rest).
```

Figure A.2: The Sieve of Eratosthenes algorithm, direct coding in Prolog

```
:- module(_, _, [fsyntax, lazy]).

:- use_module(library(lazy/lazy_lib), [take/3, nums_from/2]).

:- lazy fun_eval cut/1.
cut([])       := [].
cut([H | T]) := [H | ~cut(~cut_(T, H))].

:- lazy fun_eval cut_/2.
cut_([], _)         := [].
cut_([H2 | T], H1) := R :-
    R = ( H2 mod H1 > 0 ? [H2 | ~cut_(T, H1)] | ~cut_(T, H1) ).

:- lazy fun_eval primes/0.
primes := ~cut(~nums_from(2)).

test_primes(N) := ~take(N, ~primes).
```

Figure A.3: Classical example for computing primes (*lazy* version, Ciao fsyntax).

Appendix B

Arrays Testing Code

```
:- module(quicksort,_,[loops,functional,arrays]).

% (all implementations)
:- use_module(library(arrays/arrays_fix)).
:- use_module(library(arrays/arrays_mut)).
:- use_module(library(arrays/arrays_log)).
:- use_module(library(arrays/arrays_lists)).

:- use_module(library(lists)).

test_mut(N,Res) :-
    new_array_mut(N,Data),
    before_quicksort(N,Data,Res).

test_fix(N,Res) :-
    new_array_fix(N,Data),
    before_quicksort(N,Data,Res).

test_ext(N,Res) :-
    new_array_log(Data),
    before_quicksort(N,Data,Res).

test_lists(N,Res) :-
    length(Data,N),
    before_quicksort(N,Data,Res).

before_quicksort(N,Data,Res) :-
    A = 1664525,
    C = 1013904223,
    Seed = 12345,
    To is N - 1,
    for (I in 0 .. To) {
        Seed := (A * Seed + C) /\ 0xffffffff,
```

```
        Data[I] := (Seed mod N) + 1
    },
    quicksort(Data,0,To,Res).

quicksort(Array,Low,High,FinalArray) :-
    if (Low < High) {
        partition(Array,Low,High,NewArray,Pi),
        quicksort(NewArray,Low,Pi - 1,LeftArray),
        quicksort(LeftArray,Pi + 1, High, FinalArray)
    } else {
        FinalArray = Array
    }.

partition(Array,Low,High,NewArray,Res) :-
    Pivot = Array[High],
    I = Low - 1,
    To = High - 1,
    for (J in Low .. To) {
        if (Array[J] =< Pivot) {
            I := I + 1,
            Temp := Array[I],
            Array[I] := Array[J],
            Array[J] := Temp
        }
    },

    Temp2 := Array[I + 1],
    Array[I + 1] := Array[High],
    Array[High] := Temp2,
    NewArray = Array,
    Res = I + 1.
```

Appendix C

Interface of Arrays Package

```
% get_elem(+Array,+Index,-Elem)
:- multifile get_elem/3.
% replace_elem(+Array,+Index,+Val,-NewArray)
:- multifile replace_elem/4.
% array_length(+Array,-Length)
:- multifile array_length/2.
```

Appendix D

Plagiarism Results

The result of submitting the entire project through Turnitin is:

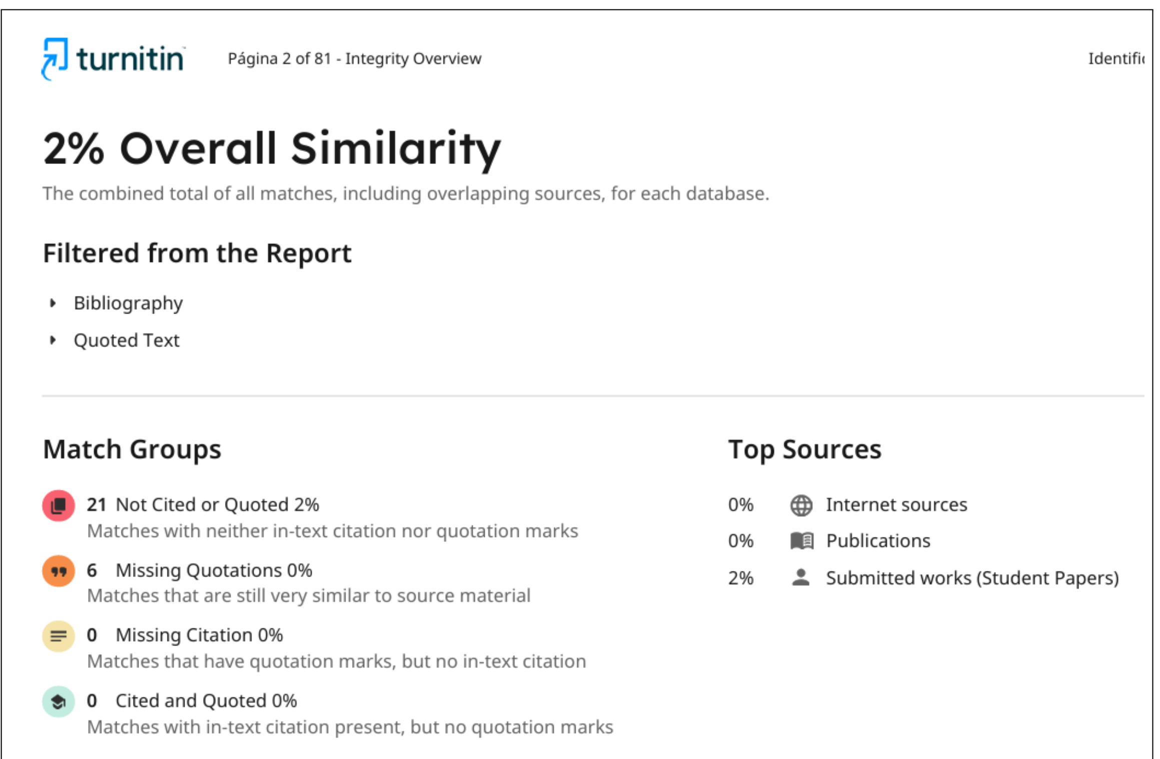


Figure D.1: Screenshot of the originality report of Turnitin.

Este documento esta firmado por

	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jun 03 21:59:58 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)