Vrije Universiteit Brussel
Faculteit Wetenschappen
Departement Informatica en Toegepaste Informatica

Language Symbiosis through a joint Abstract Grammar

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Informatica

Door: Adriaan Peeters
Promotor: Prof. Dr. Theo D'Hondt
Augustus 2003

Samenvatting

Eén enkel programmeerparadigma volstaat niet om alle programmeerproblemen op te lossen op de meest optimale manier. Door middel van taal symbiose en multi-paradigm programming kunnen meerdere programmeertalen of paradigma's door elkaar gebruikt worden. Hierdoor heeft de programmeur de beschikking over een veel breder aanbod aan mogelijkheden om het probleem op te lossen. De huidige implementaties van een multi-paradigm programmeertaal lijden echter aan een zwak ontwerp omdat ze eerst proberen de concrete syntax van de verschillende paradigma's te combineren en dan een evaluator implementeren voor deze nieuwe taal.

Wij stellen een omgekeerde aanpak voor. We vertrekken voor het ontwerp van een multi-paradigm programmeertaal van de abstracte syntax. We maken een goede implementatie van een evaluator voor deze abstracte taal door gebruik te maken van bestaande evaluatoren. In een tweede stap zullen we er een concrete syntax voor ontwerpen. We tonen de haalbaarheid van deze aanpak aan door de implementatie van de symbiose tussen een logische en een imperative taal te maken op basis van hun respectievelijke abstracte syntax en evaluatoren.

Abstract

A single programming paradigm is not sufficient to implement all computational problems in the most optimal way. Using language symbiosis and multi-paradigm programming different programming languages and paradigms can be used interchangeably allowing the programmer to use a wide range of possibilities to implement a solution for his problem. However, current implementations of multi-paradigm programming languages suffer from a weak design because they first try to combine the concrete syntaxes of the individual paradigms and then implement the evaluator for the newly created language.

We propose to use the converse approach. By starting the design of a multi-paradigm language from an abstract syntax, we can create a clean implementation of the evaluator for this new abstract language by combining existing evaluators. In a second step we will design a concrete syntax for it. We validate this approach by implementing the symbiosis of a logic and an imperative programming language on the basis of their respective abstract syntaxes and evaluators.

Acknowledgements

This dissertation would have never been finished without the great support of a lot of people. Therefore I wish to express my gratitude towards:

Prof. Dr. Theo D'Hondt for coming up with the subject of this dissertation, guiding me through the different stages and for promoting this dissertation.

Kris Gybels and Wolfgang De Meuter for proofreading even during the last stages. Without their help this dissertation would not be nearly as readable as it is now.

Maja D'Hondt for her comments and suggestions during the early stages of this dissertation.

The researchers of the Programming Technology Lab for their constructive comments during the thesis presentations.

My fellow thesis students for their help and support during the last four years.

Katrien Steurs for her linguistic proofreading and for her support over the last year.

The Vrije Universiteit Brussel and Departement Informatica for providing an excellent education in a fun and inspiring environment.

Last but not least my friends and family for supporting me and giving me the opportunity to study in the best possible circumstances.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programming languages are the key to the development of new ideas and applications. Over the years, lots of programming languages have been proposed and implemented. These programming languages were developed because there was a need for new features. Every language has its advantages and disadvantages: some languages have a very simple syntax while other languages can be used to generate extremely fast programs. Some are used for educational purposes, others are used for major business applications. But most of these languages share at least one thing: they can be classified on the basis of the general principles behind the design of the language. The principles that the languages in one group share is the programming paradigm. Each programming paradigm is best suited to solve certain computational problems, but in some cases it would be better to use a combination of paradigms to implement a solution. To make this possible multi-paradigm programming languages were developed. These languages allow programmers to use the different ideas from different paradigms in one programming language. Such a language is usually implemented as the symbiosis of two or more programming languages. By symbiosis we mean that they are able to use each others functionality and data structures.
For the languages to be usable, it is important to have a simple syntax and an easy system to program using the different programming paradigms. Therefore special attention has to be paid to the design of the syntax of the language, as well as its internal implementation. We notice however that it is hard to program using many multi-paradigm programming languages because the syntax is too complex. This is caused by the design decisions made during the development of the language. Usually one paradigm is added to a programming language that already supports another paradigm. C++ [Str91] is an example of this: object-oriented programming constructs were added while the original language supports imperative programming. This resulted in a syntax that makes it easier to program in an imperative style than in an object-oriented style. Another drawback of concentrating on the syntax distracts from the real problem: finding a well designed combination of both paradigms. This causes difficulty in sharing values between the different languages and sometimes forces the programmer to manually convert values between languages.

1.1 Thesis

Because the design of current multi-paradigm programming languages is too much focused on the syntax of the language, less attention is given to the implementation and the internal representation of the languages. To solve these weak design decisions, we propose a converse approach: we will first focus on the internal representation of the languages we want to combine. To do this we first distinguish two aspects of the symbiosis of paradigms:

Symbiosis on the syntactic level combines the syntax of the paradigms and attempts to design a usable syntax.

Symbiosis on the semantic level focuses on the internal representation of the programming elements of the paradigms.

Current multi-paradigm programming languages try to solve both problems at the same time or focus on the first, the syntactic level of the language. In this dissertation however we will focus on the latter as it is the most difficult one. The quality of design of the syntactic level of the language is highly subjective and will be addressed later. We claim that by combining languages on the semantic level, a multi-paradigm programming language can be obtained that has a clean design of its internal data structures and allows simple sharing of values between paradigms.
To support this claim we will design a multi-paradigm programming language by combining the abstract syntax of two programming languages. We want our resulting language to:

- Offer a simple system to share data between the different programming paradigms, such as variables and values.
- Have a clean and simple design and implementation.

We will accomplish these conditions by creating a programming language that:

- Merges the abstract grammar of the paradigms and contains data structures that are the same in both paradigms.
- Uses the existing evaluators of the original languages and changes them as little as possible.

We will demonstrate this by combining a meta-circular evaluator for Pico [D'H03] with a logic evaluator named Loco. We will combine their internal data structures and combine their respective evaluators and change them as little as possible. Using the basic reflective support of Pico we will illustrate how we can use our combined language to implement a limited example of Logic Meta Programming [Bru03].

1.2 Outline

In the rest of this dissertation we will introduce the terms and concepts used in programming and programming language design, and explain our experimental implementation. The text is structured as follows: in *chapter 2*, we will explain the technology that is needed in programming language design. We will introduce metaprogramming and show how it relates to program language design. We will also see how the syntax of a language is related to its implementation and explain why typing of data is important. In *chapter 3* we will explain what a programming paradigm exactly is and get an overview of the different paradigms that exist. *Chapter 4* takes a more in depth look at the logic programming paradigm. In *chapter 5* we introduce the notion of multi-paradigm programming, and illustrate the theory using existing multi-paradigm languages. In *chapter 6* we explain our experimental implementation of a symbiosis between two programming languages on the level of the abstract syntax. And finally in *chapter 7* we suggest some topics for future work and conclude this dissertation with some final remarks.

Chapter 2

Preliminaries

In this chapter we will introduce some concepts and techniques concerning programming languages and programming language design. We will look at technology used to reason about programs, and techniques used to implement a programming language and finally we will define the syntax for it.

2.1 Meta-Programming and Reflection

Programming languages are used to implement programs that help us with our work or for our amusement. Therefore the program has to reason about data and communicate with the user or with other programs. This data can be anything such as financial data, the text of a book or some animations for a film. But we can also use the program to program about another program. This is called *meta-programming*.

2.1.1 Meta-Programming

Meta-programming is a programming technique that enables us to program about programs. We can for example control the executing of the program, or reason about the program itself to extract information from it. Meta-programming structures the programming system in different levels. The first level, or base-level, is the basic program running the computation as a normal program. The second level, or meta-level, specifies a program that reasons about the basic program. This meta-level program is often the interpreter of the language. It will intervene in the basic computation when certain conditions are met or run simultaneously inspecting the program that is running. Depending on the implementation of the reflection system, the meta-program can reason about function calls, variable assignments or even every single statement. The meta-level hierarchy is not limited to two levels: when necessary and if the implementation supports it, one can program about a meta-program, called meta-meta-programming. In the ultimate case the amount of levels is indefinite. Figure 2.1 shows how the different levels can be structured.
Meta-programming can be used to resolve *cross-cutting concerns* [KLM+97]. A cross-cutting concern is a piece of functionality that is scattered throughout the

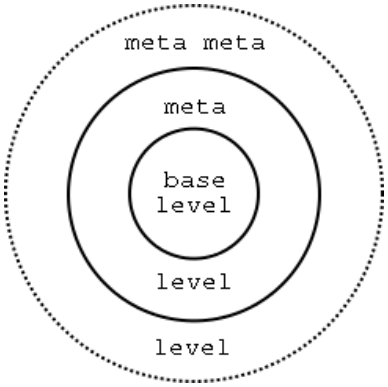

Figure 2.1: The base level encapsulated in the meta level

program to implement its functionality but that is loosely coupled with the code around it and it would be better if it was completely separated. A logging system is the standard example of a cross-cutting concern. It is usually added to a program to help the debugging process. To add this logging system to a program the programmer adds the logging code to certain parts of the program, but the code has no functional meaning at these locations. It would be better if we could separate the logging code from the rest of the program so that it can be removed easily when debugging is done. We can achieve this by hooking the logging system as a meta-program to certain function calls. This meta-program is separated from the base program and can be easily removed when logging is not necessary anymore. This programming method is called separation of concerns [LH95]: the different, independent parts of a program are separated from each other. We can compare this to functions that encapsulate common behaviours that are loosely coupled with each other. Meta-programming can be used to encapsulate different concerns and to isolate them from each other and from the base-level program.

2.1.2 Reification

Before programs can reason about other programs, the meta-level program must have a means to reference and represent the base-level program they want to reason about. So the meta-language must have a data structure to represent function calls, objects or whatever programming element the base-level language supports, and functionality to capture these data and act upon it. *Reification* is the term used to describe the process of making something that was previously implicit, explicitly available to the programming language for manipulation. It is the key to meta-programming languages as it is a programming language construct that allows to treat a program as data and reference to it. When a program wants to reason about any programming language element it has to reify or *absorb* the programming element it wants to reason about to a value that can be used in further computation by the program.

For example, if we want to refer to a function call, we want to be able to get the name of the function that is called, and the parameters that are given. Initially, this information is implicit, it is just a function call. By reifying this function call, we make it explicit by creating a data structure that contains the name of the function and its parameters. These reification data are *causally connected* to the actual reified information, so that when one of both is modified, the other is updated too. This is accomplished not by creating a data structure that contains the data we need, but instead by creating a data structure that refers to the function call and by adding functionality that will extract the name of the function and its parameters from this data structure when necessary.

2.1.3 Reflection

When the languages that are used for the base-level as well as for the meta-level are the same, we can use the language to reason about itself. This is called *reflection*: the ability of a system to inspect and interact with its own computation by reifying parts of itself. A system that supports this reflection is self-aware and can reason about its own process on a meta-level. The system has the possibility to inspect the state of its current computation and can possibly intervene in the flow of that computation. Reflection is important for components of artificial intelligence [Kam95], research in programming languages and debugging. It was first introduced by Brian Smith [Smi84] for procedural languages. Pattie Maes [Mae87] improved the ideas of computational reflection and introduced reflection for object-oriented programming languages. Support for reflection in a programming language can be implemented in many different ways. Some languages that support reflection have the ability to reason about and change themselves. Others can only inspect their own computation.
In the rest of this chapter we will describe the terminology used in reflection research, show some uses of reflection and give a short overview of the reflectional capabilities of some important programming languages.

2.1.4 Terminology

As not all implementations of reflection have the same power, we have to be able to classify the different implementations. Therefore we distinguish two aspects of reflection:

- **Introspection:** it is possible to inspect the program while it is running and take actions depending on the state of the program.
- **Intercession:** it is possible to change the program at runtime and to change its actual code such as variables, functions and methods.

A system that supports only introspection is often wrongly called a reflectional system. To distinguish these partial reflectional systems from real reflectional implementations, systems supporting introspection and intercession are called *fully reflectional*. This full reflection is important, without it only inspection of the program

can be done, without changing anything. It is thus possible to detect certain states of a program, but the actual state can not be changed. By adding intercession however, we could for example add methods to an object or add or change the values of variables, all at runtime. With intercession, we are thus able to change the state of the program. This allows the programmer to make his program totally adaptable to a situation: it can change its own state to fit the circumstances. Consider for example a window on the screen. In the same way that this window is updated with new fields or buttons by actions of the user, a program can be updated to fit the requirements of the particular situation. With an extensive implementation of full reflection it is even possible to completely change the program at runtime, so that it becomes a completely different program. This technique is used by some computer viruses as we will see in section 2.1.7 where we give a short overview of the reflectional properties of some important programming languages.
Apart from this classification of reflectional implementations using introspection and intercession, we can also classify them on the basis of their way of implementing reflection: whether they change the original program or not.

- **Behavioural Reflection:** it is possible to alter the way actions, such as method calls, are performed.
- **Structural Reflection:** it is possible to let the program alter the definitions of the data structures such as classes and methods.

Behavioural reflection does not change the original program. It only changes the behaviour of the program without changing its implementation. This is accomplished by using hooks at different locations in the program. These hooks define locations in the program where the reflectional system can act upon and are used by the compiler or the interpreter to implement the reflective capabilities. Depending on the implementation, the hooks can be placed on function calls, variable accesses or even on every individual statement. When the particular statement is called, the reflective system intercepts the call and runs its own code instead of the original code.
Structural reflection is more intrusive, it changes the original program by adding or deleting methods, variables, or even entire objects in the original program code. This way it is not only possible to change the behaviour of the program but really alter a part of the program itself or even its entire implementation.

2.1.5 Reflection in Object-Oriented Languages

Reflection is especially interesting in object-oriented programming: it is an important extension of standard object-oriented programming and makes separation of concerns [LH95], reusability and flexibility of the source code much easier. Reflection in object-oriented languages is usually achieved by associating a meta-object to every existing base-object. By sending messages — or calling methods — to this meta-object, we can make changes to the base object, for example:

- Functionality can be added before and after a method call, such as a logging system.
- A method call can be blocked, unless certain constraints are met.
- The base objects can be extended by new methods or variables.

Some program development environments, such as VisualWorks [How95] or Squeak [IKM+97] for Smalltalk, depend highly on reflection to implement the development environment. Most of the the development environment itself is written in Smalltalk, so reflection can be used to list all methods of an object or change and add methods or objects. The changes are made on the spot and are active as soon as they are added. Because the development environment is written in Smalltalk and is completely accessible to the programmer, he can change the development environment or debug it while at the same time working with it. There is no compilation stage and the development environment does not need to restart when changes to it are made. This allows really rapid program development.
The implementation of reflection in an object-oriented programming language is called the Meta-Object Protocol, in short: MOP [Meua, KdRB91]. It is the protocol that defines what programming constructs are available to access the meta-objects and interact with them. Depending on the implementation this Meta-Object Protocol can be a part of the main programming language or can be a new language on top of it. The notion of reflection can also be augmented to an extra meta-meta-level, where reasoning about the entire collection of objects is possible. For example, we can count the number of objects of a certain type or even detect design patterns [KP96] in source code.

2.1.6 Aspect-Oriented Programming

Many programming problems can not be solved using standard object-oriented or procedural programming techniques. The adaptability of standard object-oriented design is not sufficient, for example it is not possible to add a logging system to a program without having to add it at different places in the program. So the implementation of certain functionality can become scattered throughout the entire program. These bits of functionality are called *aspects* and the reason that they are hard to design properly is that they *cross-cut* the basic functionality of the program. Aspect-Oriented Programming (AOP) [KLM+97], which is usually implemented as an extension of object-oriented programming, introduces a solution to these cross-cutting concerns. By isolating the aspect in code by separating it from the main source code and merging them later, at compile time, it becomes possible to cleanly implement the base program as well as the aspect without them interfering with each other. Our example of logging is an obvious example of an aspect. Its functionality is often added to objects for debugging reasons and has usually nothing to do with the main functionality of the object. When the debugging is finished and the logging code is not necessary anymore, it is often hard to find all its appearances in the entire source code to remove them. By separating the logging functionality in an aspect,

Programming language	Introspection	Intercession	Behavioural	Structural
C++				
Java	X			
Javassist	X	X	X	X
AspectJ	X		X	
Smalltalk	X	X	X	X

Table 2.1: A comparison of the reflectional capabilities of some important programming languages.

it is possible to add to, and remove logging from the program without changing the main objects or source code. It is like plugging a certain module in and out without telling the other modules about it. The drawback of using Aspect-oriented technology is that it is more difficult to get an overview of the entire programming project. And although it can help the programmer to debug the program, when used for other purposes it becomes even harder to debug it. In most implementations, the merge between the main source code and the aspects is done at compile time. While that is an advantage for solving cross-cutting problems, it is less obvious to do this merge as a human programmer, who tries to understand the program by getting an overview of it. Although aspect-oriented programming is often seen as a distinct technique, it is easy to show that AOP can be implemented using reflective programming techniques [BL02]. The only drawback with this approach is a sacrifice in performance, but we get extreme flexibility and reusability instead.

2.1.7 Reflection Implementations

Programming languages differ a lot in their reflective capabilities. Some programming languages claiming to support reflection only support a small subset of it. Table 2.1 gives an overview of the reflective capabilities of some important programming languages. Notice that languages that support structural reflection also support behavioural reflection. This is because when we can alter the program itself, it is of course possible to alter the behaviour of the program. In this section we will give a short overview of current programming languages and their reflective capabilities.

C++

Standard C++ has no support for reflection: neither introspection nor intercession is available. Several extensions of C++ were proposed [MIKC92, Vol00, MS01] to add support for reflection.

Java

Support for reflection is very limited in Java, only introspection is supported, so it is impossible to make changes to the state of the objects. The reflection capabilities are

available through the java.lang.reflect [Mic03a] library. Because of this very limited support for reflection, extensions of the reflection API were developed. Most of these extensions add behavioural reflection to specific kinds of operations such as method calls, field access and object creation. They are implemented in an aspect-oriented programming style, using hooks on these operations. In the next two sections we will describe two implementations of extended reflection for Java.

AspectJ

AspectJ [KHH+01] use the most wide-spread approach to implement reflection. It is the reference implementation of the Aspect-Oriented programming extension to Java. and implements behavioural reflection: it is possible to intercept method calls and thus change the behaviour of the program by executing the aspect code when the method is called. AspectJ uses *join points*, which are well-defined points in the execution of the program that can be hooked to method calls or variable accesses to implement a reflective system. Using those join points, the programmer can intercept the operations and add additional code to implement a cross-cutting concern. The AspectJ compiler compiles the source code to standard Java Bytecode that can be run on any Java Virtual Machine.

Javassist

As opposed to AspectJ, Javassist [Chi98] allows structural reflection to be performed when a class is loaded into the Java Virtual Machine. It is not possible in Java to implement full reflection at any time during the lifetime of a running program without altering the Java Virtual Machine or avoiding a big performance problem. Javassist extends the standard Java reflection API with structural reflection, thus allowing reflection on all parts of the program. The only restriction is that it allows modification only before a program is loaded into the runtime system, thus at load time. Javassist modifies the program by making direct changes to the Java Bytecode of the classes before they are loaded. Once loaded no more changes to the program are possible. This Java Bytecode is an intermediate code that can be run on almost any platform using a Java Virtual Machine. The modification of the Bytecode is implemented by a special class loader that is used instead of the system class loader. No precompiler is used, programs written with Javassist can be compiled using the standard Java compiler and can run on a standard Java Virtual Machine.

Smalltalk

Smalltalk has extensive support for reflection. Everything from a number to a meta-class is an object or can be reified as an object and used in further computation. All communication between objects is done using message passing so it is not possible to directly affect the state of an object, only indirectly, by sending messages. Because Smalltalk is almost entirely written in itself, it allows easy access to the source code, portability to other computer systems, debugging and meta-programming. Classes can be modified at runtime, the program has access to its own code and can change

itself completely. To support all these features, Smalltalk has full and structural reflection. Because of this extensive support for reflection, Smalltalk is an interesting language for programming language research. The SOUL programming language [WD01], developed at the Program Technology Lab, is a logic meta-programming language written in Smalltalk and used to reason about Smalltalk programs. Using logic programming rules, we can derive information from the Smalltalk code or even add code to the code base.

Viruses

Virus programming and research is often not considered when discussing reflective programming implementations. But current viruses use advanced structural reflection to hide their real meaning and disguise themselves as ordinary programs [Leb02]. Viruses are perhaps the most widespread usage of reflection and they show that advanced reflection is not only a research goal.

2.1.8 Conclusion

We have seen that reflection and meta-programming are interesting techniques to reason about programs. We can use these techniques to help with debugging, resolving crosscutting concerns and even programming language design. Where meta-programming allows programs to reason about *other* programs, reflection-capable programs can reason about *themselves*.

2.2 Continuations

A Continuation is the state of a running program at a certain point in time. Using reification, this state can be grabbed and made available as an explicit data structure that can be passed as a variable for using in further computation. It is a data structure mostly used in functional programming languages that captures the complete rest of the computation. We will show the theory with an example, inspired by [Meub]. Consider the following code excerpt in Pico:

```
{ x:1; y:2; z: x+y; display(z) }
```

This piece of code defines two variables `x` and `y` and puts two values in them, respectively `1` and `2`. It then defines a third variable `z` and puts the sum of the previous two variables in it. Finally it displays the value of `z` on the screen. The continuation of the second definition (`y:2;`) in this example is the rest of the code: *"add x and y, put the result in z and display z"*.
To be able to reify the continuation, we must have a language construct that grabs the continuation. The continuation can be grabbed as follows:

```
call(...expression using a variable called continuation...)
```

The expression in the argument of the `call` function has access to the variable `continuation`. This variable contains the future of the computation after the call and can be passed to the function `continue` as follows:

```
continue(a_continuation, any_value)
```

This will run `a_continuation` and force it to return `any_value`. Consider the following piece of code in Pico, again taken from [Meub]:

```
call({ n:10;
       while(true,
             if(n=0,
             continue(continuation,"ok"),
             n:=n-1)) })
```

This code counts down starting from 10 and when it reaches 0 it returns with the string `"ok"` as the result of the call of `call()`. Note that at first sight the computation is an infinite loop which runs forever. However, when the computation reaches the `contine()` statement, it runs its first argument, which is in this case the current continuation. As this continuation represents the code after the call, which is nothing, the computation stops and returns with `"ok"`. Using this technique we could implement a special loop construct:

```
{ cont:call(continuation);
  display("ok");
  continue(cont,cont) }
```

In the first line the continuation is saved to a variable `cont`. This continuation is the code after the *call*, thus the last two lines. The second line prints `"ok"` on the screen and the third line calls the continuation in `cont` and forces it to return with the value of the same variable `cont`. So the *call* in line one returns with this same continuation that it returned when it was first run and saves it again in the variable `cont`. Now the process starts all over again and a loop is accomplished. The result of this program will be a continuous printing of `"ok"` on the screen.

2.2.1 Coroutines

Continuations can also be used to implement a special style of programming. Instead of calling functions that return values, different continuations call each other and the actual computation jumps between the different continuations. Instead of returning values as the result of a function call and returning to the place where the function was called, the programmer has to code the swaps between continuations manually. This is called programming in a continuation-passing style. An example of this style are *coroutines*. These are functions or procedures that call each other but without *return information*. When one coroutine calls another coroutine it will not

```
{ loop: void;
  loop:=call({
         loop:continuation;
         while(true,loop:=call({
                           display("one");
                           continue(loop,continuation);
                           continuation}))});

  while(true,loop:=call({
                    display("two");
                    continue(loop,continuation);
                    continuation}))}
```

Table 2.2: A coroutine that alternately prints *one* and *two* on the screen.

automatically return to the first one. The only way to return is to explicitly call the first one.
Table 2.2 from [Meub] shows an example of two coroutines that alternately print `one` and `two` on the screen. Both routines run indefinitely, but after every `display` statement they jump to the other coroutine using the variable `loop` which always contains the future, in this case the other coroutine.

Coroutines vs threads

Different, independently running routines can also be achieved using *threads*. These routines can, like coroutines, compute their result independently from each other. The difference between coroutines and threads lies in the way the swap between the two computations is done. In coroutine programming the programmer has to code the jump to the other coroutine manually. In thread based programming however, this decision is done by the programming language or the operating system running the program on the basis of the priorities of the different threads. For the programmer threads seem to run simultaneously but in reality the implementation of the thread system schedules the threads and quickly swaps between them. When we would implement the example of table 2.2 using threads, the program would occasionally print `one` or `two` several times, without alternating with the other word. This is caused by the scheduling of the threading implementation.

2.3 Evaluators

Programming languages can be interpreted at runtime using an evaluator, or can be compiled to machine code or an intermediate binary format to be able to execute the program. It takes longer to run a program under an interpreter than to run the compiled code directly on the machine, but interpreting is still faster than the complete compile-run cycle. This is important while testing and debugging where

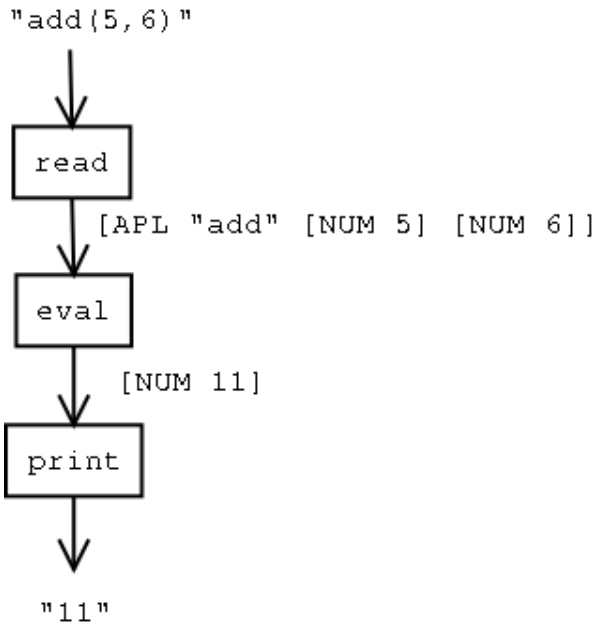

Figure 2.2: Example read-eval-print evaluation.

the delay of the compile cycle is a waste of time. Interpreters can also make it easier to add new concepts to the programming language and can help in testing them more quickly. Where a compiler converts the source code of a program to machine language, the evaluator interprets the source code at runtime by checking the syntax and evaluating it. An evaluator usually consists of three phases:

- **Read**: converts the sequence of characters from the source code to a data structure the evaluator can understand
- **Eval**: evaluates the output of the *reader* and returns a value
- **Print**: prints the output of the *evaluator* to a screen or file

The *reader* usually consists of two sub phases, the *scanner* (or *lexical analyser*) and the *parser* (or *structural analyser*). The scanner analyses the source code, detects syntactical errors and outputs a more abstract stream of data without comments, whitespace or other unneeded information. The parser takes this input and converts it to a parse three. The *evaluator* evaluates this parse tree and returns a value representing the result of the evaluation. The *printer* finally prints this value on the screen, representing the data in a readable way.
Figure 2.2 shows how the three main parts of the evaluator work together. The reader reads the input string `"add(5,6)"`, parses it and outputs the parse tree that consists of an application (`APL`) of the function `add` on two numbers (`NUM`) `5` and `6`. The evaluator takes this parse tree as its input and applies the function *add* to the

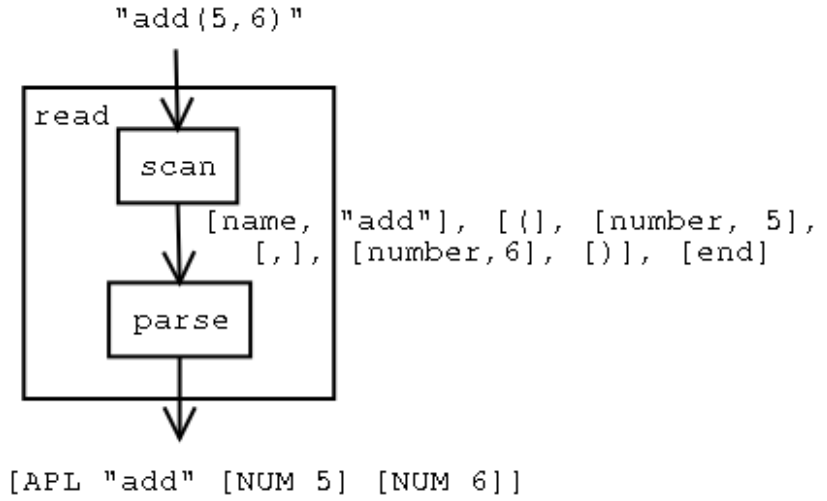

Figure 2.3: The two stages of the scanner.

two numbers. The function *add* adds the two numbers and returns the sum of both as a new number. The evaluator then passes this number on to the printer that converts it to a string and prints it on the screen.
In figure 2.3 we focus on the workings of the reader. The scanner, or *tokenizer*, converts the input string `"add(5,6)"` to a list of tokens. These tokens refer to the different elements in the input string. The scanner distinguishes successively a name `add`, a left parenthesis, a number, a comma, a number and a right parenthesis, finally it concludes with an end token. The *parser* takes this list of tokens, checks whether they meet the rules of the syntax and outputs the parse tree, which is evaluated by the evaluator as shown earlier in figure 2.2.
The evaluator loops over the three phases, successively running the reader, evaluator and printer. This is called the Read-Eval-Print Loop or REPL. In many functional languages this can be simply implemented as:

```
(loop (print (eval (read))))
```

2.3.1 Meta-Circular Evaluation

The evaluator for a programming language is often implemented as a meta-circular evaluator, which is an evaluator that is implemented in the same language that it interprets. Figure 2.4 illustrates the architecture of a metacircular interpreter. The interpreter *M1* is itself interpreted by the interpreter of *L1* and at the same time implements an interpreter for *L1*.
Programming languages such as Scheme [KCE98] or Pico [D'H03] allow very nice and simple meta-circular interpreters [ASS96, D'H03] to be written. It is a very common exercise when learning these languages to write a meta-circular interpreter for these languages to get familiar with the details of the semantics of the language

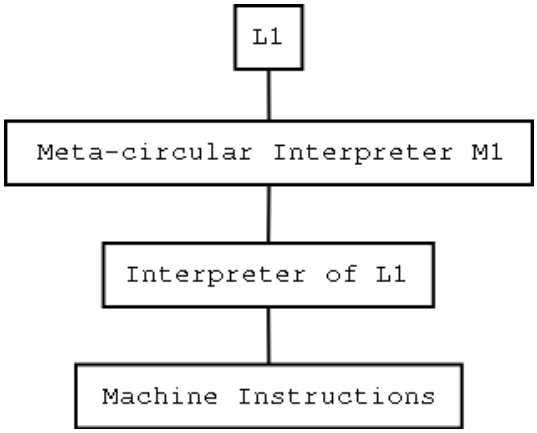

Figure 2.4: M1 is implemented by L1 and at the same time implements L1.

and the techniques required to implement it. It can help to understand the parts of the language that would otherwise be overlooked because they are too tricky or seemingly trivial. By implementing an interpreter, one cannot skip these parts and is forced to implement and thus understand the whole picture. Besides being useful for educational purposes, meta-circular interpreters can also be used for debugging programs, adding unusual or experimental control strategies and language extensions or even to allow a proof system to reason about the semantics of the language. Finally meta-circular evaluation can be used to add reflectional properties to a language that does not support reflection natively. We will use the meta-circular interpreter for Pico in our research in chapter 6 to implement an experimental language, based on Pico. It is clear that meta-circularity is an important concept for programming language design. A meta-circular evaluator can at the same time be a reference implementation and a test case for the real interpreter. It is usually simpler than the base interpreter written in a more low-level programming language and it is evaluated by a machine language interpreter. The meta-circular evaluator is thus very useful to teach the syntax and semantics of a language, as well as serve as a reference implementation for an interpreter or compiler.

2.4 Concrete and Abstract Syntax

When designing a programming language, the syntax of the language highly influences its usability and readability. The syntax must also be parsable by the interpreter or the compiler and it must be deterministic, non-ambiguous. The parser should always be able to determine the correct meaning of the source code, without ambiguities. In section 2.3 we explained how evaluators interpret the text — or source code — of a program and evaluate it. In this section we will see which internal languages and data structures are used in these evaluators to support the implementation of a programming language.

```
<postal-address> ::= <name-part> <street-address> <zip-part>
<personal-part> ::= <name> | <initial> "."
<name-part> ::= <personal-part> <last-name> <EOL> | <personal-part> <name-part>
<street-address> ::= <house-num> <street-name> <EOL>
<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>
```

Table 2.3: Example BNF: the US postal address

2.4.1 Concrete Syntax

The definition of the syntax of a programming language is called the *Concrete Syntax* or *Concrete Grammar*. It specifies tokens and keywords that allow the parser to construct an abstract syntax tree unambiguously. This concrete syntax is usually expressed in a context-free grammar. In such a grammar, every production is of the form

$$V \rightarrow w$$

where every V is a non-terminal symbol and w is a string containing terminals and non-terminals. The grammar is called context-free because the variable V can always be replaced by w, no matter in what context it occurs. This makes the implementation of the parser easier as it only has to care about the current symbol, without knowing what it was before.
The concrete grammar is often expressed in Backus-Naur form (BNF), introduced by Backus and Naur [Nau63]. It is a meta-syntactic notation used to specify the syntax of programming languages and communication protocols, for example HTTP [FGM+99]. Table 2.3 contains an example of the BNF notation, in this case used to define the syntax of a US postal address. The left hand side of the `::=` sign equals the V mentioned earlier while the right hand side is the w part. We can see that a *postal-address* consists of a *name-part* followed by a *street-address* and a identification or *zip-part*. The `|` sign is an *or* operator indicating that or the left, or the right of the sign should be used. As an example we will check whether the following fictional address matches the syntax of a US postal address:

```
Adriaan Peeters
24 Highstreet
Donnel, DO 1337
```

Starting from *postal-address* we will start to try and match the address with the concrete syntax. We can match `Adriaan` with the *name* in the *personal-part*. The last-name `Peeters` can be matched to the *last-name* and together they form the *name-part*. This finishes the first line, we can match the rest of the address in a similar fashion. Finally we can conclude that the address matches the concrete syntax of a US postal address. When the concrete syntax is correct, this does not mean that the information is reasonable. When we refer to a non existing fictitious

person or a non existing street name, the information is useless, although the syntax is correct. To indicate the meaning of the syntax, we use the abstract syntax.

2.4.2 Abstract Syntax

The *Abstract Syntax* or *Abstract Grammar* is a specification of internal representation of the computer program that is independent of the machine-oriented structures and encodings, and also of the physical representation of the data, the concrete syntax. The internal representation of a program will be typically specified by an abstract syntax in terms of categories such as "statements", "expressions" and "identifiers". The abstract syntax is implemented as a simple data structure, and is used by the evaluator or the compiler to handle the program. It is usually a lot simpler than the concrete syntax and is usually implemented as an abstract syntax tree that can easily be interpreted by the evaluator or the compiler. The abstract syntax of a language is usually not communicated to the programmers that use the programming language, as it is normally only useful to the implementors of the programming language. It consists of all internal data structures representing the program that the compiler or evaluator uses to do its job. When the programming language is reflective — in which case the programmer has access to the internal state of the program — the organisation of this abstract grammar must be known in order to be able to reason about it. The data is usually tagged to indicate the type of data it represents. In the next section 2.5 we will explain the applications of this typing.
In table 2.4 we see an example of an Abstract Syntax, it is an excerpt from the Pico abstract syntax. We start with an expression and see that it can consist of a *number*, *text*, *variable*, *application* or *void*. Furthermore we see that an application is tagged with an *APL* tag and consists of a name and arguments. The other expressions have a similar interpretation.

2.5 Static and Dynamic Typing

Almost all programming languages have a type-system, a system that defines what types of data can be manipulated in the language and how these variables can be stored in variables and referenced. We will explain the different typing systems using some simple examples in pseudo code.

2.5.1 Types

To aid the execution of a program — whether it is interpreted or compiled — all data, or values, in a programming language are tagged with the *type* of value it represents. Most programming languages support data structures such as *integers*, *text strings* and *tables* or *arrays*. These data structures are stored in memory and tagged so that the evaluator or the compiler can quickly see what type of data it encounters and act accordingly. The tags that are used correspond to the tags in the abstract grammar as we saw in table 2.4. Without this tagging the compiler or evaluator should have to deduce the type of the value from the data itself. This

```
<expression>  ::= <number>
<expression>  ::= <text>
<expression>  ::= <variable>
<expression>  ::= <application>
<expression>  ::= <void>

<number>      ::= NBR <number>
<text>        ::= TXT <text>

<variable>    ::= VAR <identifier>
<application> ::= APL <identifier> <arguments>
<dictionary>  ::= DCT <identifier> <expression> <dictionary>
<void>        ::= VOI
<identifier>  ::= <text>

<arguments>   ::= <invocation>

<invocation>  ::= <variable>
<invocation>  ::= <application>
```

Table 2.4: An abstract syntax, excerpt from the Pico abstract syntax

is slow and sometimes difficult or even impossible to do. Because computers only manipulate bits, or numbers, there is no distinction in the hardware between memory addresses, instruction code, characters, strings and integers. Everything is stored in memory as numbers. An interpreter or compiler without typing can only see these numbers and is thus unable to deduce whether the number should be interpreted as an integer or as a text string. So there must be a way to distinguish the different types of data in memory. Table 2.4 shows in rows 6 and 7 how an integer in Pico is tagged with a *NBR* tag and a string value with a *TXT* tag. Because different programming languages use different data structures, they thus use different data types.

2.5.2 Static Typing

In programming languages that use *static typing* the type of the values used in the program is based on a static analysis of the program's source code. Static typed systems usually assign a single type to each variable, function parameter or any other reference to a data structure. Once this type is assigned, it is not possible to change it, so when the same variable is assigned to two values of a different type, an error is raised.
Consider the following example:

```
var x;       // (1)
x = 5;       // (2)
x = "hello"; // (3)
```

In this example, (1) declares a variable named `x`, (2) binds the integer value `5` to `x` and (3) binds the string value `"hello"` to `x`. In a statically typed programming language, this code fragment would raise an error as it is illegal to bind two values of a different type to a variable.
In contrast to this example, most statically typed programming languages define the type of a variable when declaring the variable. This way the evaluator or the compiler knows exactly what type of data each variable contains and can enforce this type throughout the entire program. Static type checking operates on the program source code rather than at execution time. Therefore, the interpreter or compiler can detect variable type violations without executing the programs which allows *early detection* of certain errors. Furthermore the compiler can optimise the code by including only functionality for the data types that are used and by avoiding type checks at runtime.

Implementations

Programming languages that use static typing include C [Str91] and Java [Mic03b]. Some statically typed languages such as C include functionality to circumvent the static type system. They are called *weakly typed* programming languages as opposed to *strongly typed* languages that do not have this "back door".

2.5.3 Dynamic Typing

Programming languages that use *dynamic typing* assign a type to each data element at *runtime*. All the different types of values can be assigned to a single variable and once a value of one type is assigned, it is still possible to assign a value of a different type to the variable without an error being raised. As opposed to statically typed languages where the variable and the value are tagged with the type they represent, dynamically typed languages only tag the value and all computation is done based on the tag of this value. The variables are not tagged and only contain a reference to the value they contain.
In dynamically typed languages, the example from section 2.5.2 will not cause an error, nor at compile time, nor at runtime.
As a result of this dynamic typing, errors related to the misuse of values — or *type errors* — are only detected at runtime, when the erroneous statement or expression is executed. Consider the following example:

```
var x = 5;         // (1)
var y = "hello";   // (2)
x + y;             // (3)
```

This code fragment binds the integer value `5` to `x` (1), the string value `"hello"` to `y` (2) and tries to add `x` and `y` (3). In a dynamically typed language, the values `5` and `"hello"` are respectively tagged as *integer* and *text string*, but the variables `x` and `y` are not tagged. So when compiling the program, the compiler can not know what type of value the variables will contain and can not foresee possible problems. When the program finally attempts to add both values when executing line (3), the system checks the type tags of the values, finds out that the operation `+` is not defined on *integer* and *text string* and signals an error.

Dynamic typing simplifies the programming task, because the programmer does not have to worry about data types when it is not necessary. The drawback is that typing errors are only detected when running the program. This complicates the task of verifying that the code is correct and requires debugging at runtime. When a typing bug is not discovered during the verifying stage of the development of a project, it can slip through in the final product and is then difficult to resolve. Dynamic typing advocates that these disadvantages are outweighed by the flexibility of the system.

Implementations

Many recent programming languages use dynamic typing. As computational power becomes cheaper, the optimised solution of static typing is not that important anymore. Type checking can be done easily at runtime, the performance drawback can almost be ignored. Programming languages such as Scheme [KCE98], Perl, PHP and Smalltalk [GR83] use a dynamic typing system.

2.5.4 Overloading

A typing system is not only important for performance reasons or to avoid errors, it can also be used to determine what computation should be performed. Consider the following code example in pseudo code:

```
fun add(int a, int b);
fun add(string a, string b);
```

The first line declares a function to add two integers and the latter defines a function to add two strings. Both functions have the same name. Without typing of the parameters, this would cause an error, but using typing the compiler can decide whether the first or the second function should be used on the basis of the parameters to the function when it is called. When we call the function `add` as follows

```
var y = 6;
add(5,y);
```

the compiler can infer that both parameters to `add` are integers and thus the first function from our example should be called. When we call the function using two strings

```
var last = "Doe";
add("John",last);
```

the compiler finds two strings as parameters and calls the second `add` function in this place.
This system is called *overloading*: several functions with the same name can be implemented, as long as they use different types of arguments. It is also possible to overload a function on the basis of the number of arguments or a combination of both argument types and argument count.
Although overloading can be very powerful, when used incorrect it can affect the readability of the program source code. The operator `<<` in C++ is a good example of how incorrect use of overloading can complicate the usability. The expression

```
a << 1
```

will return two times the value of `a` when `a` is an integer, but if `a` is an output stream it will write `1` into it.

2.5.5 Conclusion

Typing is an important aspect of the design of a programming language. Both static and dynamic typing have their advantages and disadvantages. Therefore some languages implement a mixture of static and dynamic typing. In Perl for example it is possible to choose between static and dynamic typing for every individual variable.

2.6 Summary

In this chapter we have introduced several techniques and terms concerning programming languages and programming languages research. We saw the different approaches to the implementation of reflection and how an evaluator for a programming language works. We explained what continuations and coroutines are and how they can be used. Furthermore we introduced the concept of concrete and abstract grammars which play an important role in the design and development of current and new programming languages. Finally we explained the importance of a typing system for programming language design and implementation.

Chapter 3

Programming Paradigms

In this chapter we will describe the most important programming paradigms developed and used during the history of computer science. First we will discuss what a programming paradigm is, explain the theory behind the solutions of computational problems and make clean why different programming paradigms emerged. Finally we will sketch an overview of the main programming paradigms or programming styles.

3.1 Paradigm

Before we can talk about programming paradigms, we have to decide what a programming paradigm is. We will start with the meaning of the words *programming* and *paradigm*. The definition of programming is clear, we will include it here for completeness.

> **pro·gram·ming** or **pro·gra·ming** *n.* **1.** The designing, scheduling, or planning of a program, as in broadcasting. **2.** The writing of a computer program.

We will use the definition in 2. More puzzling is the definition of paradigm. The American Heritage Dictionary of the English Language states paradigm as follows:

> **par·a·digm** *n.* **1.** One that serves as a pattern or model. **2.** A set or list of all the inflectional forms of a word or of one of its grammatical categories: *the paradigm of an irregular verb.* **3.** A set of assumptions, concepts, values, and practices that constitutes a way of viewing reality for the community that shares them, especially in an intellectual discipline.

A programming paradigm serves as a model for a certain type of programming practice. There exist many different programming paradigms and paradigm can be seen as a different approach to find a solution for a computational problem. Timothy A. Budd says [Bud95] a programming paradigm is the way of conceptualising what it means to perform a computation, and it describes how tasks that have to be carried

out should be structured and organised. Both 1. and 3. are accurate descriptions of what a programming paradigm is. It is the combination of methods, theories and standards of coding. In the real world, we can use different words to describe the same thing, but we can also use completely different languages, or even sign language to describe it. When programming we can use different languages or methods to implement our program. Therefore we make a distinction between programming languages and programming paradigms. A programming language is an implementation of a programming paradigm. A programming paradigm is a collection of properties that is shared by multiple programming languages and that defines a certain style of programming. We will see that we can apply the same system of different ways to express an idea to computer programming paradigms and languages.

3.2 Why different paradigms?

Budd notes that the language we use in our everyday life influences our view of the world. This is called the Sapir-Whorf hypothesis: people want to organise the world and the main tool they use is language. The language we use determines how we experience the world. So there is a close relationship between the structure and vocabulary of a language and the culture that uses the language. Although this hypothesis is largely discredited in modern linguistics, especially in its strongest form [Phi98], Budd claims that it can be applied to a more restricted form of language known as programming languages. He draws on this Sapir-Whorf hypothesis to make a case for multi-paradigm programming.
Budd demonstrates his claim by showing the influence of the programming paradigm on the solution that a programmer found for a specific problem. A genetic research student had to check if a short sequence (size M) of DNA was repeated in a large genetic sequence (size N). He solved the problem in what was for him the most efficient programming language: FORTRAN. He came up with the following:

```
    DO 10 I = 1, N-M
        DO 10 J = 1, N-M
            FOUND = .TRUE.
            DO 20 K = 1, M
20                IF X[I+K-1] .NE. X[J+K-1] THEN FOUND = .FALSE.
            IF FOUND THEN ...
10    CONTINUE
```

At first sight this program seemed to be able to solve the problem, but it needed more than ten hours to complete. The student then discussed the problem with another student and she proposed to reformulate the problem in APL [Bud95]. Because it is more natural to use sorting in APL instead of loops, she came up with a solution that divided the long sequence in short sequences with the length M to check. Then she sorted those short sequences and if a duplicate was found, the short sequence was repeated. With this solution, the program ran just a few minutes instead of hours. Although the overall performance of FORTRAN for standard computational

tasks was much better than APL, the APL solution was tens of times faster than the FORTRAN solution, because of the approach taken. This shows how important the influence of the paradigm is on the solution that is found. More examples prove that there is indeed an influence of the paradigm that is used.
With this result we can see that it is important to investigate whether it is possible to combine the best properties of multiple paradigms. If we can achieve a successful merge of multiple paradigms, we can allow the programmer to use the paradigm that fits best for one particular problem, while using a single programming language. Using abstraction it should be possible to reuse a piece of code written in another paradigm, without having to worry about which data structures need to be used to communicate between the paradigms. In chapter 5 we explain how we can implement such a language and see some example implementations.

3.3 Church, Turing

Before we will investigate the different programming paradigms, we will introduce the theory of computability and explain why we can compute the same result using different programming paradigms.
In the beginning of the nineteenth century mathematicians came up with some questions they wanted to be solved. One of them was the question whether there is an algorithm that can be applied to any mathematical assertion and will eventually tell whether that assertion is true of false. This question is known as the *Entscheidungsproblem* or *decision problem*. Both Alan Turing and Alonzo Church proposed a solution to this question and answered that it is *not* possible to construct such an algorithm.

3.3.1 Turing Machine

Alan Turing defined a hypothetical machine to prove that the decision problem is undecidable. This *Turing Machine* [Tur36] consists of an infinitely long bidirectional tape, with symbols at regular intervals. The machine knows the current location on the tape and the current state, which is one of a finite set of internal states. At each step, the machine reads a symbol from the tape. Using the internal program that specifies the action to take for every current state and symbol read, we can decide what the next state will be, what symbol has to be written to the tape and whether the tape should move left, right or stay in its original position. This simple system implements the entire Turing machine. Using this machine, Turing proved that the answer to the decision problem is negative for elementary number theory and proved the incomputability of the stop problem [Tur36].

3.3.2 Church Conjecture

Another solution to prove that the decision problem can not be computed was proposed by Church: the lambda calculus. This calculus is the theoretical basis for functional programming. Now there were two solutions to the decision problem that

were both able to prove that the decision problem could not be proven. So they had the same computational power. This led Church to formulate the conjecture that all formalisms of computability are equivalent. This is later reformulated and named after him [HMU01]:

> **Church's Conjecture**: *Any procedure for which there exists an effective procedure can be realized by a Turing machine.*

Translated to current computer science terms, this means that any program can be executed by a Turing machine. If we accept Church's conjecture, then any language in which it is possible to simulate a Turing machine is powerful enough to implement *any* algorithm. The language is then called Turing complete. When we apply this to programming paradigms, we can solve a problem using any paradigm, if we can solve it using one of the other paradigms.
No machine matching the exact Turing design has ever been implemented, except for educational purposes. Although it would be theoretically possible to use it for any computation, this would be very difficult as the machine would be too slow and complicated to use and program. This is called the Turing tarpit: a language or system that is theoretically universal, but in practice too complicated to use.

3.4 Main paradigms

Turing Machines could be used to solve all computational problems, but they are very difficult to use. Programming a Turing machine is complicated, and fixing bugs even more. So turing machines were only used for theoretical computer science such as research in computability.
During the history of computer science, different programming paradigms were developed as different ideas about programming emerged. Every paradigm has its own purpose and background. Some became very popular, others did not, despite their excellent characteristics. When talking about programming paradigms, we distinguish four main paradigms.

- **Imperative paradigm**
  A sequential enumeration of instructions used to alter distinct memory locations.
- **Functional paradigm**
  Computation using functions, based on mathematical theory.
- **Logic paradigm**
  Queries that return result sets computed using logic theory.
- **Object-oriented paradigm**
  Models real world things, computation is done through messages between objects.

There exist other, less used paradigms, some of which are subsets or extensions of the four main paradigms, some examples are:

- **Parallel paradigm**
  A technique to allow programming for multi-processor architectures.
- **Visual paradigm**
  Programming based on combining components through images.
- **Constraint paradigm**
  Tries to meet constraints.

Constraint programming for example is closely related to logic programming. In this dissertation we will focus on the four main paradigms. We can also divide them in stateful and stateless paradigms [MMR95]:

- **stateful**
  In stateful paradigms, the computation goes from one state to another by applying functions to variables. The values in the variables are overwritten when they are changed.
- **stateless**
  In stateless paradigms, functions have no side-effects. Variables can not be assigned, the value of a variable can only be assigned when defining the variable.

Stateful paradigms are a more engineering view on programming, while stateless paradigms are more mathematically oriented. In the next sections we will see how this relates to the ideas behind the different paradigms.
Implementations of the different programming paradigms often combine bits and pieces from several paradigms. For example, object-oriented programming languages mostly use an imperative memory model and updates to the objects are done using destructive programming constructs. Programming languages that implement a single programming paradigm, without any influence of other paradigms, are called **pure** programming languages. Programming languages implementing a mixture of paradigms are called **impure**.
In the remainder of this section we will describe the four main programming paradigms and give some examples of programming languages that instantiate the paradigms.

3.4.1 Imperative Paradigm

The Imperative Paradigm is the "classical" model of computation. It is very close to the way real hardware machines work. The Paradigm is based on sequentions of conditional and looping constructs, and ways to destructively update the memory that is used. The Imperative Paradigm is often described as a system that goes from one state to another via incremental changes to the state. At every step in the program, the entire system goes from one state to another. During this computation the results are placed directly into memory. The memory locations where the data is

stored are fixed, but the contents change over time. The elements of a large structure, such as an array, are manipulated one by one to compute the result.
Programming in an imperative paradigm is not very human-friendly, as it is quite different from the way humans think. Because it is very close to the way hardware machines work, it is very difficult to get an overall picture of what the program does. Extending programs and debugging them is very difficult. To improve the usability, most instantiations of the Imperative Paradigm include special statements such as if/else and while statements. Variables and arrays are also often included. These extensions try to simplify programming in an imperative language an they do help a bit, but it's still quite messy. We will see that other paradigms try to resolve these problems of complexity by offering different solutions for encapsulation or abstraction of functionality and data.

Instantiations

The most well-known examples of imperative programming are Fortran [Ame78], C [KR88] and Pascal [Bur74]. All assembler languages are also apparent examples of the Imperative Paradigm.

3.4.2 Functional Paradigm

The Functional Paradigm takes a completely different approach. As the name states, all computation is done through functions. Functional programs contain no assignments: values can be created, but once created they can not be changed. Computation on structures — for example adding a value to every item of an array — does not change the array itself, but returns a new array containing the result of the computation. So as there are no changes to the state of the program over time, we say that functional programs are *atemporal* ("without time"). We can even delay some computations until the value is really needed. Because we have no assignments, the result of a function — given the same arguments — is always the same, no matter when the function is called. We call this property *referential transparency*. Functions that possess referential transparency can be more easily manipulated than mathematical objects and allow programs to be proven correct through mathematics.
Another important feature of functional languages is that a function itself is first-class. We can save them in variables, pass them as arguments to functions or return them as result from a function. This enables programmers to create high-level functions that accept functions as arguments. We can for example write a general sort function and pass the comparison function as a parameter. This way we can easily create many different sort functions. Quite complicated computations can thus become trivial to implement.

Instantiations

ML [MTH90] and Haskell [Jon03] are both pure functional programming languages. Other functional programming languages such as Scheme [KCE98] are often extended with imperative procedures and are thus impure.

3.4.3 Logic Paradigm

The Logic Programming approach arose from the research in Artificial Intelligence and Automatic Theorem Proving. It is used in AI to create an abstract model of the world and reason about it. A Logic Program consists of three basic statements: facts, rules and queries. The only data structure is a logical term, inherited from first-order logic. Using these facts and rules, the programmer specifies *what* he wants to be computed instead of *how* it has to be computed. This is called *Declarative Programming*: the desired answer is specified and the computation is a search of the space of possible solutions to match this answer. The search tries all facts and rules and uses backtracking when a rule can not be met.
Like Functional languages, the logic paradigm has no state. A query will always return the same result, given the fact that the database of facts and rules did not change. We will explain the details of logic programming in chapter 4.

Instantiations

The most well-known logic programming language is PROLOG [DEDC96]. It is the reference implementation of logic programming and "PROLOG" is very often used as a synonym for "logic programming". Other programming languages that use logic programming often combine it with other functionality, as we will see in chapter 5.

3.4.4 Object-Oriented Paradigm

Object-oriented programming (OO) can be considered as the most important programming paradigm. Although it has its flaws, it is the most used paradigm in new programming projects. In object-oriented programming everything is an object. The object contains both the data and the functions that can be applied to the data structure. Because the data in the object changes over time, the system is stateful so it does not possess the referential transparency property. Encapsulating data and functionality in one object is a fundamentally different approach in comparison to the other paradigms and is a major advantage for abstraction: different parts of a complex system can be easily separated and protected from each other. This offers options for reuse and independent code creation when working with different programmers on large projects. To implement the functionality of the program, the programmers define relationships between the different objects. Using *polymorphism* new objects, based on inherited data and functions from other objects, can be adapted by changing the inherited functionality. This way the object internal data structures can differ completely from the parent object, but this is not visible from the outside.
Objects are often compared to real-world objects and are usually implemented to represent the real-world object in a computational environment. These objects then act as components offering a specific service. Other objects can use the service but do not have to know how the service is performed and implemented. In large projects this becomes the most important quality of object-oriented programming. It allows

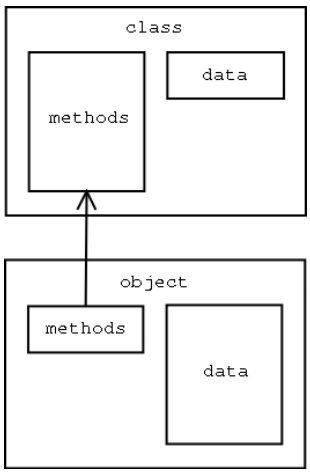

Figure 3.1: A class-based object.

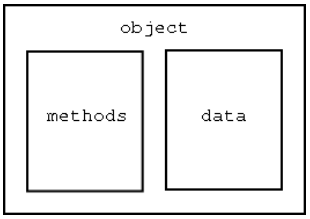

Figure 3.2: A prototype-based object.

to develop the different objects independently as well as replacing them with other objects without having to change anything of the rest of the project.
There are different views on object-oriented programming, based on how the objects are created:

Class-based Object-orientation is the traditional way of object-oriented programming. A class is used to organise the basic layout and functionality of other objects. New objects are created by instantiating this class and assigning memory for its internal state. The object contains only its state, which is the data, and a reference to the class it instantiates to be able to access the functionality. When interacting with the object, the code from the class is used, along with the data from the object. Figure 3.1 illustrates this.

Prototype-based Object-orientation takes another approach, there are only objects and no classes. Both functionality and state are contained in the object and only for its own usage. So an object is very simple, as we see in figure 3.2. When a new object has to be created, it is copied from an existing one. Functionality is added by adding methods and variables to the newly created object. This system makes it easier to adapt the functionality of an object to the physical object or to the information it represents.

A large amount of tools arose to aid object-oriented software design. Modelling languages such as UML[FS00] are almost indispensable in organising large object-oriented projects.

Design Patterns

In an attempt to capture often used patterns in object-oriented software development, Design Patterns [GHJV95] were developed. A design pattern describes the

different objects involved in a particular object-oriented design problem. Because the same problems had to be solved over and over again, the "gang of four" combined them in a book [GHJV95], along with extensive documentation. This book is crucial for object-oriented programming design and is very useful to avoid reinventing the wheel.

Instantiations

One of the first object-oriented programming languages is Smalltalk [GR83]. It is a pure, class-based object-oriented programming language in which everything is an object and computation is done by message passing between the objects. C++ [Str91] was the first well-known and widely used object-oriented programming language. It is an extension of C and is thus not designed from scratch as an object-oriented language and has some flaws because of that. Java [Mic03b] on the other hand ìs designed as an object-oriented programming language from scratch. A huge library of functionality is available thanks to the abstraction the object-oriented paradigm offers. This library functionality can be used without any knowledge of its implementation.

3.4.5 Other Paradigms

There exist other less-used paradigms such as constraint programming, parallel programming and others. Constraint programming is close to logic programming. It defines rules which should be satisfied during program execution. Parallel programming is used in multi-processor environments that require some special language constructs to aid the communication between different processors and to avoid deadlocks. We will not describe them in detail in this dissertation.

3.5 Conclusion

In this chapter we have seen what programming paradigms are and why it is important to know different paradigms. We have seen that any computational problem can be expressed in a Turing machine by the Church conjecture.
We have shown that, although almost all programming languages can implement a solution to a problem, using different programming languages is important to find the most efficient solution. Research in the psychology of programming [Pet89] shows that expertise in programming is far more strongly related to the number of different programming styles known by the programmer than it is to the number of years experience in programming. It is thus important to continue research in programming languages development and not to fall for the current main programming paradigm, object-oriented programming.
Finally we have seen which programming paradigms currently exist and we have given an overview of some implementations.

Chapter 4

Logic Programming

In this chapter we will explain a reference implementation of Logic Programming, namely PROLOG. We will describe the terminology used in logic programming, discuss the theory behind it and explain the theory with some examples. Because most implementations of the logic programming paradigm share more or less the same syntax and because most implementations are derived from PROLOG, we will focus on the PROLOG-style syntax and semantics to implement the examples of the theory presented. To finish this chapter we will explain how logic programming can be used to reason about other programs by introducing Logic Meta-Programming.

4.1 Syntax

Logic Programming differs from all other programming techniques in the way the programming language specifies the program. The programs specify **what** needs to be computed instead of **how** it is to be computed. We will show this declarative programming style by means of an example. The specification of the program is done using **facts** and **rules**. The facts define the data on which the logic program will compute the result, they can be seen as a database containing all the data. This database is called the knowledge base. The rules define relationships between the facts. The following example program specifies the relationship between some people, visualised in Figure 4.1. The program can compute the father and grandfather of the given people.

```
father(jan,bert).
father(bert,johan).
father(bert,bob).
father(johan,arne).

grandfather(X,Z):-
  father(X,Y),
  father(Y,Z).
```

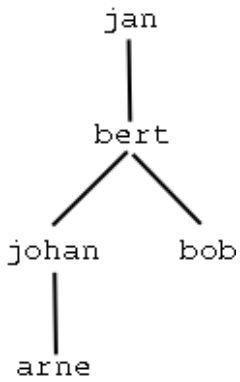

Figure 4.1: A small family tree.

The first four lines are facts, they define who the father is of whom. The first line for example says that *jan* is the father of *bert*. The last three lines define a rule, the grandfather relationship. Note that this rule does not define how to compute the grandfather relationship but rather defines what is considered a grandfather: `X` is the grandfather of `Z` if `X` is the father of `Y` **and** `Y` is the father of `Z`. The capital letters are variables that have to point to the same people in the entire rule.
With this knowledge base available, we can start asking questions. We could for example ask who the grandfather is of whom by firing a **query** on the logic program:

```
grandfather(X,Y).
```

The result will be an enumeration of variable bindings indicating the three grandfather relationships, for example *jan* is the grandfather of *johan* and *bob*:

```
X = jan
Y = johan ;

X = jan
Y = bob ;

X = bert
Y = arne ;
```

A logic programming language will, as opposed to most other programming paradigms, return all the possible solutions to a query.
Do note that the syntax does not specify the interpretation we used here, defining the interpretation of the data structures is — like with any other programming language — the task of the programmer.
We can see in the examples that facts, rules and queries all share the same basic syntax. This basic data element is called a **predicate**. It consists of a name, or predicate symbol, and a number of arguments between parenthesis. These arguments

are called **terms** and can be variables, functors or constants, such as numbers and labels. For example the predicate `father(jan,bert)` has the name `father` and the arguments are the labels `jan` and `bert`. Functors are used to group data, consider for example the predicate `person(name(jan,peeters))`, it has as name `person` and the argument is a functor that contains the first name and the last name of the person. The number of arguments of the predicate or the functor is called its **arity**. The example predicate `father` above has arity two, which is noted as `father/2`.
A rule consists of a conclusion, which is a predicate, and a body, which can be multiple predicates. A fact can be seen as a special case of a rule, one without a body.

4.2 Evaluating Logic Programs

The result of a logic program is computed by the logic evaluator, called the inference engine. This engine tries to satisfy a query in the basis of the facts and rules from the knowledge base. We will explain this process by means of an example. We will use the same knowledge base as in section 4.1.
The computation of a logic program starts with firing a query. The inference engine takes the query and tries to satisfy it. Using pattern-matching, facts and rules are matched against the query. When the name and arity of a predicate match, the parameters of the predicates are matched. This matching process is called *unification*. Two terms one from each predicate, are taken and compared:

- If both terms are constants then the result of the unification is a success when they are equal, or else failure.
- If one term is an unbound variable then this variable is bound to the other, which may be any term, and the unification succeeds.
- If both terms are structures containing bound variables or functors, then each pair of sub-terms is unified recursively and the unification succeeds if all sub-terms unify.

The result of this unification process is either failure or success with a set of variable bindings. PROLOG uses *depth-first* search to compute the proof and uses *back-tracking* when a path fails, to try another path. The computation continues until a solution is found or until there are no more possibilities.
Consider we fire the following query on the knowledge base from section 4.1:

```
father(johan,X).
```

Using pattern matching, the four facts are matched because they have the same name and arity as the predicate in the query. The unification process loops over these facts and fails to unify the first three facts because their first parameters are constants and differ from `johan`. The fourth matches as the first parameter is the same constant and the variable `X` can be bound to the constant `arne`. The inferencer returns this variable binding:

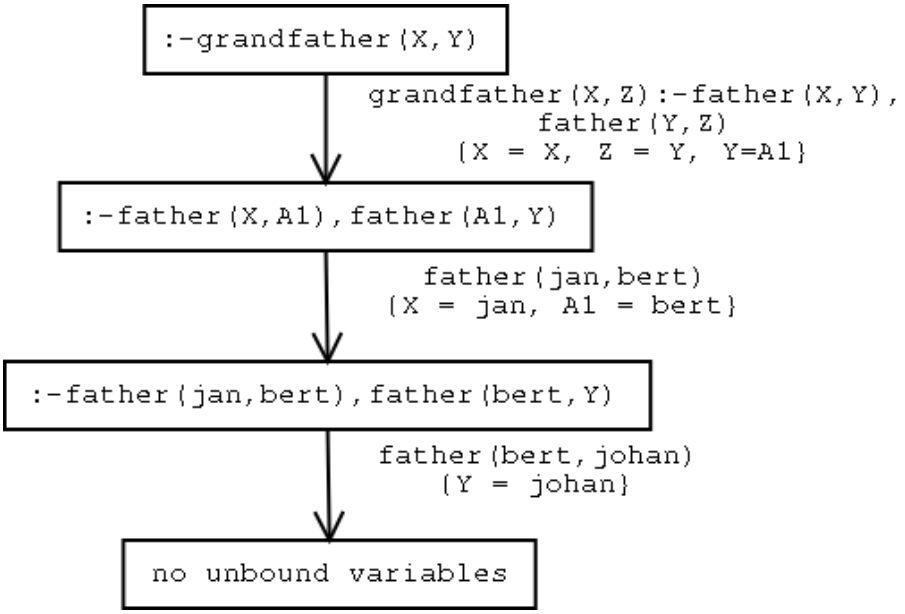

Figure 4.2: Derivation tree of the family tree example.

```
X = arne ;
```

In this example only one result is returned, but a query such as `father(bert,X).` would have returned `bert`'s two sons `johan` and `bob`.

Let us now focus on a more complicated example. We will see how rules are used and how backtracking is needed to find the correct solution. We will start by firing the following query:

```
grandfather(X,Y).
```

As we have seen in section 4.1 this query returns the grandfather relationships that can be derived from the facts that are given. Figure 4.2 shows the derivation of this result.
Pattern-matching is unable to match any facts to this query, only the grandfather rule can be matched. As both variables `X` and `Y` are unbound, we can unify them with the variables in the body of the rule and fire the body of the rule again.

```
father(X,A1),father(A1,Y).
```

Note that the original variable `Y` is renamed to `A1` to avoid variable name clashes. The pattern-matcher matches the first father-predicate in the new query with the knowledge-base and finds a match. The constants are unified with the variables and the updated query is fired again:

```
father(jan,bert),father(bert,Y).
```

The pattern-matcher again finds `father(jan,bert)` to match `father(bert,Y)` but the unification engine fails to match them because the first arguments differ. The inferencer now *backtracks* and tries the second fact, `father(bert,johan)`, which unifies and the computation is complete, all variables are bound. We can conclude that *jan* is the grandfather of *johan*.
Backtracking is not limited to one level, it can cancel an entire derivation tree because of a failure and try a different option. Because prolog uses *depth-first* search, where the inferencer first tries to satisfy the first part of a query before the rest is considered, backtracking is required to cancel the variable bindings that are already done and try a new path in the derivation tree. This process continues until a solution is found or until there are no more possibilities to try.

4.3 Theory

Logic programming is based on first-order logic, restricted to allow only Horn clauses [Hor51]. We can thus express a rule such as $A : -B_1, \ldots, B_m$ in Horn-clause logic as $A \leftarrow B_1 \wedge \ldots \wedge B_m$. Which means that when B_1 and $B_2 \ldots B_m$ are true, then it implies that A is also true. To solve a query $? - Q_1, \ldots, Q_n$ we have to find a rule $A : -B_1, \ldots, Bm$ where A matches Q_1 and solve $? - B_1, \ldots, B_m, Q_2, \ldots, Qn$. We know that $B_1, \ldots, Bm$ can be empty in case of a fact instead of a rule. This accordance between logic programming and first-order logic allows all known theories about logic to be applied to logic programming. Using this theory we can prove correctness and computability of logic programs.

4.4 Arithmetic

Standard arithmetic in logic programming languages is complicated. To be able to use numbers, we have to create a representation for it. To do this, we use the notion of *functors*, which we introduced earlier. Those special terms serve only as a data-container and are never evaluated. In the following example we use the functor `s(X)` as the successor of `X`. Using this representation we could implement natural numbers as:

```
natural(0).
natural(s(X)):-natural(X).
```

This is a recursive representation of numbers. A number is the successor of the previous number, or zero. To represent the number 5 we would get `s(s(s(s(s(0)))))`. A `plus/3` operator would be implemented as:

```
plus(0,Y,Y):-natural(Y).
plus(s(X),Y,s(Z)):-plus(X,Y,Z).
```

The result of the sum is computed in the third argument, which is unnatural: we would expect the result to replace the `plus/3` operator. The advantage of this implementation is that it is multi-directional, we can use any combination of bound and unbound variables and the inferencer will find the possible solutions. This is the power of declarative programming. We can use the `plus/3` operator to compute all possibilities to get a certain number, for example:

```
?- plus(X,Y,s(s(0))).

X = 0
Y = s(s(0)) ;

X = s(0)
Y = s(0) ;

X = s(s(0))
Y = 0 ;
```

The inferencer returns the three possibilities that yield the number two when we sum them. Using this technique, we can implement a `minus/3` operator in terms of the plus operator:

```
minus(X,Y,Z):-plus(Z,Y,X).
```

When we add a notion of lists to our logic programming language, we can see even more advanced applications of multi-directional computation. A list is represented as a comma-separated list enclosed by brackets, for example `[a,b,c]`. We can refer to the first item of the list using `[First|Rest]`. The variable `First` will then contain the first element of the list while the variable `Rest` will contain the rest of the list. Finally an empty list is represented as `[]`.
We can now implement an `append/3` predicate that appends two lists as follows:

```
append([],Y,Y).
append([H|T],Y,[H|Z]):-append(T,Y,Z).
```

This is very similar to the `plus/3` predicate and allows to append two lists or split a list in all the different possible distributions, when we query for example

```
append([a,b,c],X,[a,b,c,d,e,f]).
```

The inferencer will return the sublist that has to be appended to `[a,b,c]` to become the list `[a,b,c,d,e,f]`:

```
X = [d, e, f] ;
```

4.5 Forward Chaining and Backward Chaining

Search programs such as PROLOG can use the rules they are given in two directions: from body to head or forward, and from head to body or backward. These two directions correspond to the different approaches to logic programming that can be taken.

Forward chaining is a reasoning process driven by the data that will try to generate possible solutions starting from initial facts. Rules are used to derive new facts. The system will continue until a goal state is reached or no new facts can be derived.

Backward chaining on the other hand is a goal-driven reasoning process. The process starts from a given goal state or hypothesis and tries to prove it. During the computation the rules are used to derive new hypothesis. When a conclusion of a rule matches the goal, the body from the rule becomes the new hypothesis and the computation continues. It stops when either a hypothesis consists entirely of known facts or when no new hypothesis can be found.

PROLOG-like languages use backward chaining to compute their solutions. They try to prove a query with facts, using intermediate rules where necessary.

4.6 Logic Meta-Programming

Logic programming can also be used to reason about programs, which is called Logic Meta-Programming. It is a technique developed at the Programming Technology Lab (PROG) [Bru03]. When we use logic programming instead of another programming paradigm to reason about programs, we can use rules and queries to compute the result. Combining the power of logic programming with the capabilities of meta-programming yields a strong combination that can be used for different applications.

4.6.1 Implementations

There exist different implementations that use logic meta-programming. SOUL [WD01] is an extension of Smalltalk that allows to reason about Smalltalk objects. It has been used in many applications amongst others: to extract data from source code and to ensure the correctness of documentation in software [Kel03].

4.7 Summary

In this chapter we introduced logic programming. We saw the common syntax of logic programming languages and explained how a query can be fired and how the results are computed. We saw how the computation is performed using a detailed example and explained the computational model of a logic programming language. We visualised the computation using a derivation tree and explained the basic theory behind

logic programming. We also illustrated how arithmetic computation can be powerful using its bidirectional computation possibilities and explained the difference between forward and backward chaining. Finally we introduced logic meta-programming and saw some of its applications. In chapter 6 we will see how we can combine a logic programming language with another language to implement a symbiosis between the two.

Chapter 5

Multi-Paradigm Programming

In this chapter we will introduce the notion of multi-paradigm programming. We will define its meaning and show how it relates to language symbiosis. Furthermore we will see which combinations of paradigms are interesting to investigate, give an indication of the difficulty to combine the different paradigms and finally we will give an overview of current multi-paradigm implementations.

5.1 What is Multi-Paradigm Programming

In chapter 3 we gave an overview of the four main programming paradigms that are currently in use. We saw how each of these programming paradigms can be used to solve a problem in a different way. By definition, these programming paradigms offer a single view on programming that enables programmers to organise their ideas clearly and allows them to communicate these ideas with other programmers. After some time the programmers noticed that a lot of problems kept coming back over and over again. To avoid reinventing the wheel, standard solutions were developed, such as design patterns [GHJV95]. These solutions can easily be used in different programming languages that implement the same programming paradigm.
The disadvantage is that the paradigms are too narrowly focused to describe every aspect of a large, complex system. For example, one part of a program can be expressed best in an object-oriented style while another part that reasons about data requires a logic programming style. Because only one programming paradigm is available in most programming languages, a non-optimal solution is often implemented because the right programming paradigm for it is not available. To tackle this disadvantage a technique called *multi-paradigm programming* was developed. This allowed the use of different programming paradigms interchangeably, and allows the programmer to use the best suited paradigm for a given task.
Multi-paradigm programming was introduced by Zave [Zav89] and is implemented in many different programming languages [Bud95, MMR95, Kiz98, Spi94]. Since its introduction, a lot of research is done on multi-paradigm programming, each with its own angle of incidence. The implementations range from the simple combination of two programming languages to a completely new language that offers functionality

to program in the four main programming paradigms. In short, we can describe a multi-paradigm programming language as follows:

> *A programming language or environment that allows a developer to use the best suited programming paradigm for a specific task or problem.*

5.2 Why Multi-Paradigm Programming

The need for a multi-paradigm programming language can be shown easily. In chapter 3 we saw how each programming paradigm can be used to solve a problem in a different way. The example on searching in DNA sequences showed that the programming language that is used, influences the solution that is found. We saw that both the programming languages FORTRAN and APL can be used to solve the problem, but APL yielded a faster solution. From this example we can conclude that different programming languages or paradigms are useful and that although both languages can be used to implement the solution, one language is more suited than the other.
But why would we want a combination of these paradigms, is the choice between the different languages not enough? The answer is definitely *no*. It is easy to see that a single programming paradigm does not suffice anymore in larger, more complex programs. To achieve the most optimal solution in this situation, every part of the program has to be implemented in the language or paradigm that is most suited for it. To implement the entire program, a programmer thus needs multiple paradigms at his disposal in order to get the most optimal environment to implement a solution for the given problem.
Apart from the need for multi-paradigm programming from a software engineering point of view, it is needed in other circumstances too. For example when designing a system composed out of components, the different components can be implemented in different programming paradigms. These components should be able to collaborate and thus need a common system to communicate with each other. This is made possible by sharing common data structures between the programming paradigms so that every paradigm has knowledge of all data structures from other paradigms.

5.3 Multi-Paradigm Programming vs Meta-Programming

As we saw in section 2.1.1, standard meta-programming enabled languages can be divided into two groups: those that use the same programming language to implement the base level program and the meta level program that reasons about it, those are the reflective meta-programming languages, and those that use a different programming language to do meta-programming and are non-reflective. Multi-paradigm programming languages can be divided in the same way. Not all multi-paradigm languages support reflection and so they can not meta-program on themselves. But those languages that do support reflection use the same multi-paradigm language to implement the base as well as the meta-level program. So when a multi-paradigm

programming language has reflective capabilities, it can be used to meta-program about itself.

5.4 Multi-Paradigm Programming vs Language Symbiosis

Before we can even start implementing a multi-paradigm programming language, we must first define how the different programming paradigms will communicate with each other. Therefore we must determine how they will pass data and call each other's functions. As we can not program in a certain programming paradigm, but in a programming language that implements a programming paradigm, we have to define a symbiosis between the different programming languages that will be used in the implementation of a multi-paradigm programming language. This *language symbiosis* defines how functionality and data from one programming language can be used in another and vice versa. It defines the data structures that are shared between the different programming languages and specifies how variables and functions are handled between the languages. We can define language symbiosis as follows:

> *Transparent use of features from one particular programming language in another language*

The term language symbiosis was first introduced in RbCl [IMY92], a reflective object-oriented concurrent programming language. Tourwé [Tou02] states that language symbiosis is an absolute prerequisite for true multi-paradigm programming.

5.5 Combining Programming Paradigms

When combining programming paradigms there are certain requirements to be met as well as some problems to be tackled. In this section we will describe these requirements, investigate the problems that occur and explain how these can be handled.

5.5.1 Requirements

The goal of multi-paradigm programming is that features from one paradigm can be used easily together with programming constructs from other paradigms. This requires that certain program elements such as primitive data structures, objects and result sets are shared between the programming paradigms. It must also be possible to use functions, methods and queries interchangeably without too much hassle. We will also keep in mind that other paradigms could be added in the future. And finally it is clear that the paradigms should be well integrated so that the resulting multi-paradigm programming language is more powerful than the sum of the different parts.

5.5.2 Problems

Designing a multi-paradigm programming language is not a trivial task. In this section we will describe the problems that arise when combining different paradigms. First we have to find a system to combine the syntax of the paradigms as well as the different computational models. Second we will see how the combination of these models can result in paradigm leaks.

Syntax

Defining a concrete syntax for a multi-paradigm programming language is a real challenge. It requires us to combine all programming constructs of the different programming languages in a joint concrete syntax. For a language that implements the four main programming paradigms, this syntax has to combine imperative memory and loop constructs, functions, objects and logic queries all in one language. It is not trivial to design a syntax for this language that implements all these statements and is a usable language for the programmer.
Apart from a concrete syntax, we have to define the internals of the programming language, the abstract syntax. This syntax has to combine the different programming constructs used in the different programming paradigms so that the evaluator can decide how to compute the results for these constructs. It is important to design this abstract grammar well by sharing as much data structures as possible between paradigms while retaining all available functionality. The better the abstract grammar is designed, the easier it is to implement the multi-paradigm programming language.

Paradigm Leaks

The most difficult type of language symbiosis is between a symbiosis between a stateful and a stateless language. The definition of a stateless language says that every time a certain computation is done — whether it is a function call or a logic query — the result will always be the same. This is called referential transparency. But when we combine this stateless language with a stateful language, we can not guarantee this referential transparency. When a stateful part of the program is called, it can change the state of the program and thus the result of subsequent function calls or queries. This is called this *paradigm leaks*: the result of one of the stateful paradigm ends up in the programming language implementing the stateless paradigm. Another example of this paradigm leak, which is less difficult to tackle, is the result of a logic query. It returns multiple results when a logic query is launched. These multiple results are an example of a paradigm leak as the normal return of a function is a normal value. However, this can be solved easily by adding a construct that loops over all the results of the query. Finally when we want to combine logic programming with another paradigm, we need to be aware of the backtracking that occurs when evaluating logic queries.

	imperative	functional	logic	object-oriented
imperative	C, Assembler	Scheme	Leda	C++
functional		Haskell	λProlog	Haskell++
logic			Prolog	SOUL
object-oriented				Smalltalk

Table 5.1: Difficulty to combine the four main programming paradigms

Easy
Moderate
Difficult

Table 5.2: Legend for table 5.1

5.6 Overview

In this overview we will indicate the difficulties we encounter when trying to combine programming paradigms. As we noted earlier, we focus on the four main programming paradigms in this dissertation so we will see the different possible combinations of the imperative, functional, logic and object-oriented programming paradigms in this overview. A lot of research has been done on the possibilities to combine several programming paradigms, where possible we will refer to some implementations.
Table 5.1 outlines the different combinations of paradigms that exist. The colour indicates the difficulty to combine the two paradigms and is explained in table 5.2. This indication of the difficulty is based on a combination of factors including the difference between stateful and stateless paradigms and their computational model. From the table we can deduce that combining two languages of the same paradigm is the easiest to do. Combining logic programming with a stateful programming paradigm such as imperative or object-oriented programming seems to be the most difficult to achieve while the other possible combinations are average in difficulty. In the next sections we will give a short overview of the different combinations and refer to some examples of implementations.

5.6.1 Imperative and Functional

When we add imperative operations to a functional programming language, we lose the referential transparency and it is thus not possible anymore to prove the correctness of a program written in the combined programming language. Because of the stateful, destructive operations of the imperative programming paradigm, the memory model has to be adapted to support these operations. Scheme [KCE98] implements this memory model by using a *garbage collector* [ASS85]. This garbage collector consolidates the free memory by scanning the entire memory for unused space. This can be done easily because the entire memory model of Scheme is based on recursive lists, starting from a top list. When the system needs more memory

the garbage collector simply traverses the entire memory to find unused space. This space is consolidated and used to store new data.

5.6.2 Imperative and Logic

Combining an imperative and a logic programming language is much more difficult. Logic programming highly depends on its referential transparency property. To be able to compute a solution by proving it on the basis of rules and facts, it is necessary that these rules and facts stay the same during the entire computation, otherwise the result is useless. When implementing a language that implements both paradigms, and when programming in this new language, it is thus important to take this into consideration. No plain solutions for this combination could be found, although Prolog does have some imperative constructs such as input and output and Leda [Bud95] supports many different paradigms including imperative and logic programming.

5.6.3 Imperative and Object-Oriented

Because imperative and object-oriented programming are both stateful programming paradigms, they are quite simple to combine. Most common object-oriented programming languages support some sort of imperative programming in the form of loops and variable assignment. Languages implementing a combination of imperative and object-oriented programming are C++ [Str91] and Java [Mic03b]. The first is an extension of C [KR88] with added object-oriented functionality. The latter is designed as an OO language from the start, but supports some imperative programming constructs.

5.6.4 Functional and Logic

Pure functional and logic programming are both stateless paradigms. This allows a simple integration without too much hassle about paradigm leaks. Both function calls and logic queries return the same result on invocation at any time in the execution of the program. So when we combine both paradigms, the resulting system will be referentially transparent and its correctness can be proven using formal methods. λProlog [Mil91, NM91] is an extension of Prolog that adds higher-order functions, λ-abstractions and other functional capabilities to a logic programming language. It is entirely based on an extension of Horn clauses, as seen in chapter 4, which allows us to use similar techniques to perform the computation for a λProlog program.

5.6.5 Functional and Object-Oriented

Although a pure functional programming language is stateless and object-oriented programming languages are stateful, it is not difficult to combine the two. If we compare a method call with a function call, it becomes even trivial to combine functional and object-oriented programming: we can simply implement all functions as a method call to one object. The only drawback is that we cannot guarantee

referential transparency as with any other combination of a stateful and stateless language. But if we keep that in mind when using the programming language, we can program easily using this combined programming language. Haskell++ [HS] and Object-Gofer [SA] are extensions of respectively Haskell and Gofer. They add object-oriented programming constructs to the functional programming languages. Another solution is proposed by Qian [qia00].

5.6.6 Logic and Object-Oriented

The logic and object-oriented programming paradigms are often seen combined in meta-programming applications. For example SOUL [BGW02] is a reflective system that uses logic programming to reason about Smalltalk programs. As with all previous combinations of stateful and stateless paradigms, these applications have to deal with paradigm leaks so it is a hard task to implement them. Prolog++ [Mos94] is an extension of Prolog with added object-oriented programming facilities. Gybels [Gyb03] has written an overview of current logic object-oriented implementations.

5.6.7 Conclusion

We have shown that combining a logic programming language with any stateful language is a difficult task. For the reasoning process to yield correct results, the data it reasons about should not change while running. This can not be guaranteed in a stateful environment.

In the next two sections we will first describe a simple language symbiosis to illustrate the different approaches that can be taken to implement a programming language. After that we will describe some implementations of a multi-paradigm programming language.

5.7 Language Symbiosis Implementations

There are different approaches possible to implement a programming language. We can implement an evaluator for the language, as show in section 2.3, or we can compile the source code to machine code and run that on the machine without the need of an evaluator. We will show these different approaches by describing three implementations of a language symbiosis between Java and Scheme. We have an implementation using an evaluator, one using a compiler and one in between, that compiles in two steps. All implementations run on a Java Virtual Machine which can be compared to a hardware machine for our purposes.

5.7.1 Programming Languages

We chose to look at an example of language symbiosis that is interesting from a technical as well as an educational point of view. Combining the object-oriented

language Java [Mic03b] with the functional language Scheme [KCE98] offers possibilities to see how flexible symbiosis is in terms of passing values between styles, creating objects and calling functions or methods. It also illustrates how important the syntax is to have a successful and usable programming language.
We will first give a short introduction of the programming languages used in our overview.

Java

Java is an object-oriented programming language with control structures for loops and exceptions. It is an interesting language to explore as Java programs can run on different computer architectures without any change to the program. Partly because of this portability, Java quickly became one of the leading programming languages. It has a large library of functionality available which allows rapid application development. Java is used for end-user applications, small Java applets on websites, and major server applications [Fou03]. Especially internet and intranet applications use Java. Java is also becoming increasingly popular for research.
The major advantage of Java is the Bytecode-compilation. When compiling a Java program, the source code is converted to Java Bytecode, a binary representation of the program that can be executed on different computer architectures by a Java Virtual Machine [LY99]. This virtual machine is available for a wide range of computer architectures and evaluates the bytecode to run on the actual machine.

Scheme

Scheme is a functional programming language, based on Lisp [SG93]. The syntax is very easy, everything is based on a notion of linked lists. There are no precedence rules because there are no operators, prefix notation is used for all function calls. Not only data but also programs are represented as lists, which allows simple reasoning about programs without the need of external functionality from libraries for reflectional purposes. This list-representation makes it very easy to create a Metacircular Evaluator in Scheme [ASS85], it uses list manipulation to evaluate the program. Scheme has first-class functions, which allows functions to be passed as parameters to other functions or stored in variables. Functions can also be created anonymously. These properties make Scheme an interesting language for our research.

5.7.2 Implementations

When selecting implementations of Scheme in Java, we used some conditions that had to be met by those systems. They had to:

- Run standard Scheme code, i.e. be Revised^5 Report (R^{5}RS) [KCE98] compatible
- Be able to run Scheme code in Java
- Be able to run Java code from Scheme

```
(import s2j)
(import generic-procedures)
(define <java.awt.Frame> (java-class "java.awt.Frame"))
(define-generic show)
(define frame (make <java.awt.Frame> (->jstring "Hello")))
(show frame)
```

Table 5.3: SISC code to draw a window on the screen.

We selected three implementations of a symbiosis between Java and Scheme that met these conditions as good as possible and at the same time differed completely in implementation. We were not looking for implementations that simply implement a Scheme interpreter in Java without any possibility to access Java data structures from Scheme. Furthermore it was not only a prerequisite to be able to run Java code from Scheme, we also had to be able to use Scheme as a meta-programming language for Java. This could easily be done using the standard Java reflective capabilities [Mic03a].
All three implementations run on a standard Java Virtual Machine and allow to mix Java and Scheme code. These examples will show that a good symbiosis requires a good design. We will see that SISC and Kawa are maybe technically better implementations, but they lack a simple common syntax.
To illustrate the syntax of the implementations, we will show how to draw a window on the screen using the three different languages.

SISC: Second Interpreter of Scheme Code

SISC [sis03] is a Scheme interpreter, it does not compile the code. By using an optimised interpreter they achieve very good performance. The drawback is that a program should always be shipped together with the interpreter, as it is not possible to distribute a compiled version.
Automatic conversion of values between Java and Scheme is not available, everything has to be done manually. This makes the use of variables from one language in the other laborious. Table 5.3 shows how complex the syntax is to draw a simple window on the screen. We see for example how a Scheme string is converted to a Java string using `->jstring`.

Kawa

Kawa [kaw03] is a GNU [Sta85] project. It is a straight-forward implementation that compiles a Scheme program immediately to Java Bytecode. Debugging and inspection of the generated code is therefore restricted. While exchanging values between the different programming languages is simple, the performance is not optimal. Table 5.4 contains the example code to draw a window on the screen.

```
(define frame (make <java.awt.Frame> "Hello"))
(invoke frame "show")
```

Table 5.4: Kawa code to draw a window on the screen.

```
(import "java.awt.*")
(define win (Frame. "Hello"))
(.show win)
```

Table 5.5: JScheme code to draw a window on the screen.

JScheme

During the development of JScheme [jsc03], the emphasis was on integration with Java. This is clearly visible in the way how data are exchanged. The access to Java classes and methods is very simple and conversion of values between Java and Scheme is automated as much as possible. Compiling a Scheme program is done in two steps:

- Transcode JScheme source code to Java source code
- Compile Java source code to Java Bytecode

While JScheme is not fully R^5RS compatible, it is still the most interesting implementation of Scheme in Java. The intermediate step in Java source code and the very simple value conversion make it easy to debug and add new functionality.

5.7.3 Conclusion

By comparing tree implementations of a language symbiosis between Java and Scheme we showed that the syntax of a programming language highly influences its usability. The concrete syntax is important so that it is easy to write or read the programs but without a simple system to share variables between the languages it is a lot more difficult and complicated to write a program. We can solve this by using a well designed abstract grammar, so that manual conversions are needed as little as possible.

```
(jlib.JLIB.load)
(define win (window "Hello"
   (border (north (label "Hello")))))
(.show win)
```

Table 5.6: JScheme code to draw a window on the screen and adding a label using JLIB.

In the next section we will take a second step in language symbiosis as we describe some programming languages that implement more than two programming paradigms.

5.8 Multi-Paradigm Implementations

Many recent programming languages are multi-paradigm but most of the time we are not aware that we are using a multi-paradigm language. The existence of multi-paradigm languages — some even used frequently, such as C++ [Str91] which combines imperative and object-oriented programming — and the work done by Budd [Bud95] prove that it is useful to investigate other possible paradigm combinations. In this overview we will focus on multi-paradigm implementations that implement several programming paradigms, allowing these paradigms to be freely mixed together. We will see how sharing of variables and values between paradigms is handled and how the combination of paradigms is implemented.

5.8.1 Leda

Leda[Bud95] is a multi-paradigm programming language designed from scratch. The idea is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms. Therefore Leda supports imperative, object-oriented, logic and functional programming. Leda was developed to allow students to use different programming paradigms at the same time, so that they would not over-compartmentalise their knowledge of programming types and try to use one programming paradigm to solve every computational problem. It has been shown that this happens by the example of the DNA sequences in section 3.2. In Leda, the different programming paradigms can be used interchangeably. We can define functions, logic rules and queries, and objects, all in the same syntactic language. All communication between the different paradigms is done using functions, so a programmer does not have to know which paradigm is used to implement a function he uses, given that he knows the number and type of the arguments and the result type of the function. That way it does not matter whether the function is implemented as an imperative computation or as a logic query. Because all variables are in the same environment for all paradigms, the sharing of variables between paradigms is not an issue. Support for value sharing is accomplished by a special *undefined* value for variables, which allows variables to be bound to values when evaluating logic queries. The result of such a logic query is a *relation*, it can be used as a true or false value or the set of results can be used in the rest of the computation.

5.8.2 Oz

Oz [MMR95] approaches the multi-paradigm programming problem from a different point of view. Like Leda it is designed from scratch with special attention to the simplicity of the language, but the difference is that Oz consists of simple kernel languages on top of which other programming languages are built. The different

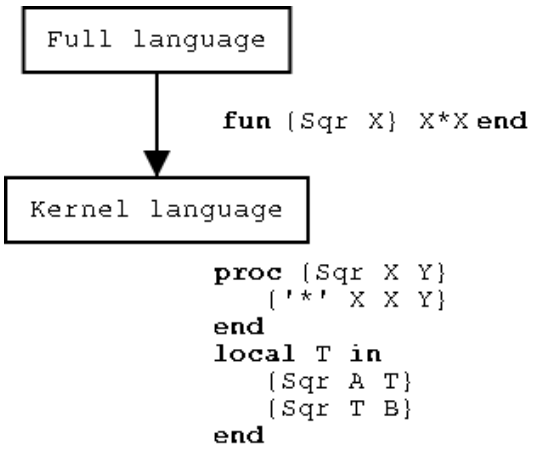

Figure 5.1: Comparing the full language and the kernel language in Oz

kernel languages are designed starting from a base kernel language by successively adding new concepts to the language. While the original simple kernel language only supports pure functional programming, the final kernel language can be used to express programs in functional, logic and object-oriented programming styles. The kernel languages are very limited in expressiveness, so on top of them full languages are added. These full languages provide useful linguistic abstractions and syntactic sugar to make programming easier. The separation between the kernel language and full language allows new linguistic abstractions to be added without changing the kernel language. It is even possible to design a completely new syntax while retaining the same kernel language. Figure 5.1 taken from [MMR95] illustrates the difference between the full language and its underlying kernel language: it shows how a function definition is converted to a kernel language procedure definition.
Although Oz supports programming in multiple programming paradigms, it is not a real multi-paradigm programming language. Unlike other multi-paradigm programming languages that combine elements from different paradigms, the Oz kernel languages support the different paradigms on a different level. A multi-paradigm programming language can be designed on top of the kernel languages as a full language. This allows different languages to be designed, all sharing the same kernel languages. When a solution does not suffice, we can simple start over without having to redesign the basis formed by the kernel languages.

5.8.3 Kiev

Kiev [Kiz98] is somewhere in between language symbiosis and multi-paradigm programming. It is a "multi-paradigm" programming language implemented as an extension of Java [Mic03b]. Is is based on an earlier Java extension named Pizza [OW97]. Kiev uses a compiler to generate Java bytecode that can be run on a standard Java Virtual Machine. This allows Kiev programs to be easily ported to other

computer architectures but at the same time limits the extensions that can be added. Kiev adds a lot of new features to Java, among others:

- a logic query engine
- a goto statement
- closures

Although these new features are an important extension of Java, Kiev is hardly a real multi-paradigm programming language. It is too tightly coupled to Java to allow easy integration of the different paradigms. This is for example caused by the syntax which is simply an extension of Java. Instead it would be better to try to design a syntax for Kiev that reflects the different programming paradigms that are supported. Although Kiev is more than the symbiosis between two programming languages, it is not as flexible as Leda and Oz.

5.9 Conclusion

In this chapter we introduced multi-paradigm programming, explained the reason for its research and the problems that occur when designing a multi-paradigm programming language. Finally we investigated some implementations of language symbiosis and multi-paradigm programming.
All mentioned implementations use a syntax oriented approach to implement language symbiosis. The different solutions to combine Java and Scheme try to make the symbiosis as transparent as possible by simplifying the concrete syntax. Leda tries to merge the programming constructs of the different paradigms in one syntactic language. Oz uses simple kernel languages to implement features that can be used to implement programming constructs for the different programming paradigms. And finally Kiev tries to extend Java with a lot of new features and constructs to add multi-paradigm functionality.
It is clear from the example multi-paradigm languages that the syntax highly influences the usability of the programming language. When the syntax is too complicated, one is not inclined to use the language.
In the next chapter we will try to resolve this problem with a solution based on abstract syntax. The most important part of a multi-paradigm language is the way values between different paradigms are shared. We will introduce a solution to this and implement an experimental symbiosis between an imperative and a logic programming language using the existing evaluators.

Chapter 6

Implementing a Multi-Paradigm Programming Language

In the previous chapter we saw that most multi-paradigm programming languages are focused on teaching multi-paradigm programming and implement multi-paradigm programming on the syntax level of the language. Because they try to compress all elements from imperative, functional, logic and object-oriented programming in one language, the syntax becomes very complex and difficult to understand.
In this chapter we propose a solution to these problems. We will separate the design of the concrete syntax of the language from the internal implementation of the data structures used to implement a multi-paradigm programming language. To achieve this we implement the combination of programming paradigms on the level of the abstract syntax. In our experimental implementation we combine the abstract grammar of a meta-circular implementation of Pico and Loco. Pico is a programming language that is strongly inspired by Scheme. We use the meta-circular implementation that is comparable to the meta-circular evaluator of Abelson and Sussman [ASS85]. Loco is a logic evaluator written in Pico with special attention to its abstract grammar. We will combine the abstract grammars and the evaluators of these two programming languages to show that a symbiosis on this level results in a coherent implementation of a multi-paradigm programming language. But first we will introduce Pico and Loco, as they form the basis for our research.

6.1 Pico

Pico [D'H03] is a programming language developed by Theo D'Hondt at the Programming Technology Lab of the Vrije Universiteit Brussel. It was initially aimed to introduce computer programming to undergraduate students. There exist several versions of Pico with front-ends for Mac Os, Windows and Linux but they all share the same Pico evaluator. This evaluator is implemented in pure ANSI C and implements its own memory handling, syntax processing and evaluation, all combined in a small optimised memory footprint. For our research we will use a meta-circular implementation of the Pico evaluator. It is the reference implementation and it

uses the same concrete and abstract syntax as the C version of Pico. Apart from the memory handling, which is simpler in the meta-circular implementation, both implementations use the same ideas for the read-eval-print loop and are highly comparable. Because of this it is not too difficult to port an extension of meta-pico to the C version.
The concrete syntax of Pico is based on a C-style syntax while the semantics are much more related to Scheme than to C. Just like Scheme, Pico supports dynamic typing, first-class functions and lazy evaluation. We will show these properties and the concrete syntax of Pico using some examples. Consider the following piece of Pico code:

```
check(A,B):
 { if(>(A,B),
      A,
      B)
 };
```

This example defines a function `check` with two variable arguments: `A` and `B`. These arguments are evaluated when the function is called. If the value of `A` is larger than the value of `B` it will return the value of `A` otherwise the value of `B` will be returned. The implementation of the `if` statement in Pico differs from the implementation in most other programming paradigms. In Pico, this if statement is implemented as a *function*. It has three arguments: the check, an expression for true and an expression for false. This works when the expressions are simple data structures, such as numbers or variables, but if the expressions are function calls, they are both evaluated when the if function is called. This is not what we want, only one of them should be computed. To explain how this is resolved, we must first introduce the notion of *lazy evaluation* in Pico. With lazy evaluation we can indicate that the parameters to a function should *not* be evaluated when the function is called. To indicate this, we add parenthesis to the variables in the definition of the function. Consider for example:

```
delay(x()):x;
```

This simple function accepts one argument `x`, but instead of evaluating its value when the function `delay` is called, this evaluation is delayed. So `x` contains a *promise* to evaluate this argument, this promise is a function without arguments. We can compute its value by calling it:

```
force(x):x();
```

This will ultimately return the value of the evaluation of the initial parameter to `delay`. Using this technique we can even implement some basic elements such as true and false:

```
true(x(),y()):x();
false(x(),y()):y();
```

The first line defines the function representing true. It has two arguments: `x` and `y`. The function returns the value of the first argument as the result. Because both arguments are lazy evaluated, only the value of the first argument `x` is computed, the value of `y` will never be computed. The function `false` is implemented as the exact opposite: it computes the value of its second argument `y` while the value of the first argument is never computed.
This implementation of boolean numbers is uncommon, most programming languages implement them as special values or numbers. But using this implementation in Pico allows us to implement the if statement as a simple function:

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

This `if` statement has the same semantics as in other programming languages: when the first argument is true, the second argument is executed, otherwise the third argument is executed. The first argument `c` is computed when `if` is called, the evaluation of `t` and `e` is delayed. The argument `c` should be a boolean value and determines whether the second or the third argument of the `if` function should be returned as a result. Because the boolean value is implemented as a function as seen before, this function is called with both non-evaluated, lazy arguments `t` and `e`. If `c` is true, it will return the value of its first argument as its result and thus will the second argument of the `if` statement be computed. When `c` is false, it will compute its second argument which corresponds to the false branch of the `if` statement.
We can extend our check function so that it supports various checks on its parameters by using an extra parameter. Because functions are *first-class* in Pico, we can pass them as variables. Consider the following extended `check` function:

```
check(F,A,B):
 { if(F(A,B),
      A,
      B)
 };
```

This function applies the function that is passed via the variable `F` to the arguments `A` and `B`. Depending on the result of this function call the value of one of both is returned. We could for example call this function as:

```
check(<,5,10);
```

We see that we pass the function `<` as a normal parameter to `check`. When this function call to `check` is evaluated, the `if` statement will be called with respectively `<(5,10)`, `5` and `10` as its first, second and third argument. The evaluation of the first argument yields true so the result of this function call will be `5`.

6.1.1 Reflection

Apart from lazy evaluation and first-class functions, Pico also has limited support for reflection. When assigning a function to a variable, we can access the information that is contained in the function definition. Suppose we define the following function `foo`:

```
foo():"ok";
```

When we try to access the function `foo` as a table, we can derive a lot of information:

```
>foo
<function foo>
>foo[1]
foo
>foo[2]
<table>
>foo[3]
ok
>foo[4]
<dictionary>
```

We see that the first position in the table is the name of the function, while the third position is the body of the function, in this case simply `ok`. The second and fourth position in the table contain respectively the arguments, which is in this case an empty list, and the environment in which the function is defined. This last element is needed to allow access to the variables that were available when the function was defined. We will see that this result corresponds exactly to the abstract syntax of a function definition in appendix A.1.

6.2 Loco

Our second language, Loco, is a logic evaluator implemented in Pico. Its concrete syntax is a mixture between the syntax of Pico and the syntax of Prolog which we explained in chapter 4. Our example from section 4.4 in which we appended two lists, can be implemented in Loco as:

```
{ append([], x, x);
  append([u, x], y, [u, z]): append(x, y, z) }
```

The first line contains a fact describing the empty list and the second line contains a rule that describes what composes a list. Notice that they are combined in braces, like the body of a Pico function. We can query these rules to append a list as follows:

```
append(["a",["b",["c",[]]]],X,["a",["b",["c",["d",["e",["f",[]]]]]]])
```

Loco computes the answer and responds with:

```
append([a, [b, [c, []]]], [d, [e, [f, []]]], [a, [b, [c, [d, [e, [f, []]]]]]])
```

We see that `X` is replaced with the part of the list that has to be appended to the first argument of `append` to make it equal to the last argument.
During the design of Loco, the main focus was on its abstract grammar. The construction of this abstract grammar is similar to the abstract grammar of the meta-circular version of Pico we introduced earlier. In appendix A.2 we take a closer look at the Loco abstract grammar.

6.3 Implementing a Multi-Paradigm Programming Language

Now that we explained the languages we will use in our language symbiosis, we can start to combine them. As we saw in the previous chapter, there are different ways to implement multi-paradigm programming languages and we have to cope with a lot of issues when trying to combine them. We have seen that combining them is especially difficult when both languages are based on a different programming paradigms. So far we have seen two approaches to implement such a language:

- **Starting from scratch:** Languages such as Leda, Oz and Kiev are completely new languages designed with the main focus on symbiosis of the different programming paradigms.
- **Combining existing languages:** The different implementations of the symbiosis of Scheme and Java start from existing languages and try to combine them.

In our solution we will diversify from these approaches and implement multi-paradigm programming in a different way. We will try to meet the following goals:

- **Combined abstract grammar**: We will implement a multi-paradigm programming language by combining the abstract grammar of two languages.
- **Few changes to the evaluators**: We will make as few changes as possible to the existing evaluators of the two languages.

We will accomplish these goals by merging the abstract grammars of Pico and Loco by combining data structures that can be merged. We will also make as few changes as possible to the existing evaluators as we want to be able to use existing evaluators in our future work.

6.3.1 Combining Abstract Grammars

We solve the problem of multi-paradigm programming by combining existing abstract grammars because this allows us to focus on the internal working of our symbiosis without trying to design a concrete syntax to represent the symbiosis. We also

described this idea in [DGDP03]. We decided to share the basic data structures such as numerical values, text strings and tables while we kept the Pico and Loco variables separate. Function definitions, applications and other Pico data structures as well as Loco data structures such as facts and rules were not shared as well. Another design decision we made was not to share the same abstract grammar for queries and function calls as this would violate our goal to minimize the changes to the individual evaluators. In appendix A.3 we include the full merged abstract grammar.

6.3.2 Adapting Evaluators

With the combined abstract grammar available, we have to combine the existing evaluators of both Pico and Loco. Because both evaluators use similar variable names for the data structures that contain the facts in Loco and the environment in Pico, we had to rename them first. Both eval functions were also renamed to respectively pico_eval and loco_eval.
Because of the recursive design of both evaluators that recursively call *eval* during their execution, we had to change them at only one place in both evaluators where the other evaluator should be called. In Loco we had to change the unification system so that it is now able to unify a logic variable with a pico variable or function call. In Pico we added a new `foreach` statement that can be used to process the results of a logic query one by one.
In the next section we use our new programming language to use Pico functions in Loco queries as well as Loco queries in Pico functions. We will also see how we can use Loco to do Logic Meta Programming on Pico.

6.4 Experiments

In this section we will use our symbiotic language to show how we can use it to write programs that use both Pico and Loco functionality. Because we focused the design of the language on the abstract syntax, we use this abstract syntax in our examples. Note that we use alphabetical tags for our values where the real implementation uses numerical tags.
First we have to add some functionality and facts to respectively our environment and knowledge base:

```
loco_eval(loco_read("{same(x,x)})"));
loco_eval(loco_read("{data(1,2);data(1,3)}"));
pico_eval(pico_read("mul(x,y):x*y"));
```

The first line adds a rule `same` to the knowledge base to indicate two facts that are the same. The third line adds two simple data facts and the third line adds a function `mul` to the environment that multiplies its two parameters.
We can now run some queries on these data, for example. We will use a fictitious syntax and differentiate between Pico and Loco constructs using a question mark in front of it. For example:

```
?same(+(2,3), ?y)
```

It is in reality represented as:

```
[PAT,"same",[TAB,[[APL,"+",[TAB,[[NBR,2],[NBR,3]]]],[LVAR,"y",0]]]]
```

This is a Loco pattern (`PAT`) with the name `same` and as arguments a Pico application and a Loco variable `y`. The application (`APL`) is an application of `+` on two numbers, `2` and `3`. When we evaluate this using our evaluator, the result of the application is "returned" from `+(2,3)` and used in future computation of the query as if it was a term containing the value. This value is then bound to the variable `y` and is returned as a result from the query:

```
same(+(2,3), 5)
```

We can also use our evaluator to do basic logic meta-programming. Suppose we fire the query

```
?same(mul[2], ?x)
```

This query will bind the logic variable `x` to the result of the Pico tabulation `mul[2]`. We defined `mul` earlier as a function that multiplies its two arguments. But in this case we treat it as a table to get its second element. This corresponds — as we can see in appendix A.1 — to the name of the function. This is thus bound to `x` and we get:

```
same(mul[2], mul)
```

Let us now take a look at firing a logic query from Pico. Because the result of such a query can have multiple results, we introduced a new native Pico function **foreach** to handle these. With this new construct we can fire a logic query and handle its results as follows:

```
foreach(i,?data(1,?x),display(i))
```

The foreach statement fires the query in its second argument and successively assigns the different results to the variable `i`. For every result from the query, `foreach` runs its last parameter, which is in this case `display(i)`. So the final result will be that every result from the query will be printed on the screen.
We can easily use this functionality to count the number of results of a query as follows:

```
count:0
foreach(i,?data(1,?x),count:=count+1)
```

If we run this piece of code on our knowledge base, then the variable `count` will contain `2` afterwards because there are two facts in the knowledge base that match our query.

6.4.1 Remarks

Note that in our implementation we do not share the same abstract syntax for a function and a query, although on the basis of the concrete grammar this would seem appropriate. Consider this expression in concrete syntax:

```
add(1,2)
```

In Pico this is an application of the function `add` on two numbers `1` and `2`, whereas in Loco this is a query with the name `add` and terms `1` and `2`. We could have represented both using the same abstract grammar and look up the function in the environment of Pico and when it is not found try to fire the query using Loco, and vice versa. But this is not possible for various reasons. Suppose we can find `add` in both the Pico environment and the Loco knowledge base, then we would have to decide to use one of both. And if this choice is not consequent, we would have ambiguous results. This could be solved by checking both Pico and Loco when adding new functions or facts, but that would violate our goal to use the existing evaluators with as few changes as possible.

6.5 Conclusions

The goal of this chapter was to introduce our experimental implementation of an evaluator for the symbiotic language, which is based on the imperative programming language Pico and the logic programming language Loco. In the first part of the chapter we explained both languages using some example functions and queries. In the second part we summarized the existing approaches to implement multi-paradigm programming languages and formulated the goals for our solution. We then explained how we implemented the symbiosis and showed the feasibility of our solution using some experiments.
We can conclude that the symbiosis of two languages on the level of their abstract syntaxes is possible and allows us to share data between the languages easily. Because we made only minimal changes to the abstract syntaxes and the respective evaluators, we have also met our second goal as the evaluator for our language has a simple design and implementation.

Chapter 7

Conclusions

7.1 Summary

During the history of computer science different programming languages are developed to satisfy the different needs of programmers. These languages can be classified using the ideas behind the design of the language. These ideas are called paradigms and are usually used to distinguish programming languages that support imperative, functional, logic, object-oriented and some other programming styles. Because standard programming languages can only be used to program in one of those styles, multi-paradigm programming languages are developed. These languages allow the programmer to use the different programming styles interchangeably and thus yield stronger expressiveness of the programming language. Different parts of a program can be implemented using different paradigms and can be encapsulated by means of abstraction.
We have seen that the programming language that a programmer is used to, highly influences the result he formulates for a certain problem. This suggests that we should pay special attention to the usability of the language when designing a multi-paradigm language. It should be simple to use the different paradigms interchangeably to avoid that a programmer will prefer one paradigm over another because it is simpler to use. But the current multi-paradigm programming languages suffer from a weak design. Their design is too much focused on the syntax of the language which causes less attention to the internal representation and their evaluators. Because of this sharing of values between the different paradigms is hampered and it is often easier to program in one paradigm instead of in all paradigms. This is exactly the opposite of what we want.

7.2 Contributions

In this dissertation we proposed a solution to these problems. We divided the design of a multi-paradigm programming language between the symbiosis of the abstract syntaxes and the symbiosis of the concrete syntaxes. Current implementations are strongly focused on the symbiosis of the concrete syntaxes. We proposed to focus

on the symbiosis of the abstract syntaxes to implement a clean and simple design of the internal representation as well as the evaluator for the language. We claimed that by combining languages on a semantic level we could obtain a multi-paradigm programming language that has a clean design of its internal data structures and allows simple sharing of values between paradigms so that this sharing is no longer hampered.
We have validated our claims by implementing a multi-paradigm programming language by combining two programming languages. We made a symbiotic language of Pico and Loco by combining their abstract syntaxes as well as their evaluators. Due to the clean design of the abstract grammar we were able to combine both languages with minimal changes to their respective evaluators. We have shown that our symbiotic language is usable and allows us to use both Pico and Loco structures interchangeably. The sharing of values and functionality is simple due to the focus on the design of the abstract grammar.

7.3 Future Work

Now that we have a multi-paradigm programming language that combines imperative and logic programming, we have to investigate how other paradigms can be added while still retaining the basic abstract grammar. The best way to do this is to design an abstract language for the new paradigm that is similar to the existing abstract grammars for Pico and Loco. We can then join the new grammar with our existing symbiotic grammar and change the respective evaluators where necessary.
A second extension will be to convert Loco and our symbiotic language to a C version similar to the original evaluator for Pico. Loco is designed to allow this conversion easily and Pico is already available as a C interpreter, so the conversion of the symbiotic language should be straightforward because only minimal changes are made to both evaluators.
A third extension which we introduced earlier is the design of a concrete syntax for our language. This is a hard, complex task because the feeling of usability of the language highly depends on the experience of the users of the language. The advantage of our work is that we can design the concrete syntax independently from the abstract syntax allowing us to adapt the syntax to our needs. It is even possible to design completely different syntaxes on the basis of the same abstract syntax. We could for example use a Scheme-style or Pico-style syntax, to aid the transition from those languages to our combined language.
Finally we must investigate how we can use our technique to combine other existing languages by symbiosis of their internal data structures. Because those languages will most likely use completely different internal representations, more changes will be necessary.

Appendix A

The Abstract Grammars

In this appendix we will explain the abstract grammars of both Pico and Loco. We will see how they are structured and how they are used to represent the internal data structures of both evaluators. Finally we will explain how the combination of both grammars is achieved. The notation we use for the grammars is BNF, which we introduced in section 2.4.1.

A.1 The Pico Abstract Grammar

First we include the Pico abstract grammar. It is organised starting from an `expression`. We see that every expression is a tagged value. For example a `number` consists of a tag `NBR` and a native — or base-level Pico — number:

```
<expression>  ::= <number>
<expression>  ::= <fraction>
<expression>  ::= <text>
<expression>  ::= <table>
<expression>  ::= <function>
<expression>  ::= <native>
<expression>  ::= <variable>
<expression>  ::= <application>
<expression>  ::= <tabulation>
<expression>  ::= <definition>
<expression>  ::= <assignment>
<expression>  ::= <void>

<number>      ::= NBR <number>
<fraction>    ::= FRC <fraction>
<text>        ::= TXT <text>
<table>       ::= TAB <table>
<function>    ::= FUN <identifier> <arguments>
                                   <expression> <dictionary>
```

```
<native>      ::= NAT <identifier> <function>
<variable>    ::= VAR <identifier>
<application> ::= APL <identifier> <arguments>
<tabulation>  ::= TBL <identifier> <expression>
<definition>  ::= DEF <invocation> <expression>
<assignment>  ::= SET <invocation> <expression>
<void>        ::= VOI

<identifier>  ::= <text>

<arguments>   ::= <table>

<invocation>  ::= <variable>
<invocation>  ::= <application>
<invocation>  ::= <tabulation>
```

A.2 The Loco Abstract Grammar

The Loco abstract grammar is, such as the Pico abstract grammar, organised starting from an `expression`. This expression can be anything from a `number` to the `result` set of a query. We see that the tagging system is similar to the system used in Pico, this will allow us to combine the grammars more easily. The last few structures such as `d-link` and `f-link` are used respectively for representing the knowledge base and the variable bindings in one result of the result set.

```
<expression>  ::= <number>
<expression>  ::= <fraction>
<expression>  ::= <text>
<expression>  ::= <table>
<expression>  ::= <assertions>
<expression>  ::= <negation>
<expression>  ::= <conjunction>
<expression>  ::= <disjunction>
<expression>  ::= <variable>
<expression>  ::= <pattern>
<expression>  ::= <fact>
<expression>  ::= <rule>
<expression>  ::= <dictionary>
<expression>  ::= <frame>
<expression>  ::= <result>
<expression>  ::= <void>

<number>      ::= NBR <number>
<fraction>    ::= FRC <fraction>
<text>        ::= TXT <text>
```

```
<table>       ::= TAB <table>
<assertions>  ::= AST <a-table>
<negation>    ::= NEG <query>
<conjunction> ::= CON <q-table>
<disjunction> ::= DIS <q-table>
<variable>    ::= VAR <symbol> <qualifier>
<pattern>     ::= PAT <symbol> <p-table>
<fact>        ::= FCT <pattern>
<rule>        ::= RUL <pattern> <query>
<dictionary>  ::= DCT <symbol> <assertion> <d-link>
<frame>       ::= FRM <variable> <expression> <f-link>
<result>      ::= RES <frame> <continuation>
<void>        ::= VOI

<a-table>     ::= <table>

<query>       ::= <negation>
<query>       ::= <conjunction>
<query>       ::= <disjunction>
<query>       ::= <pattern>

<q-table>     ::= <table>

<symbol>      ::= <text>

<qualifier>   ::= <number>

<p-table>     ::= <table>
<p-table>     ::= <variable>

<assertion>   ::= <fact>
<assertion>   ::= <rule>

<d-link>      ::= <dictionary>
<d-link>      ::= <void>

<f-link>      ::= <frame>
<f-link>      ::= <void>

<continuation>::= <number>
```

A.3 The Combined Abstract Grammar

In this section we explain our combined abstract grammar. We share data structures that are the same in both paradigms and include the other data structures from Pico and Loco. To be able to indicate these different pieces, we have split our abstract grammar in tree sections:

- **The Shared section** represents the data structures that are shared between the languages
- **The Pico section** represents the data structures of Pico that are not in the shared part
- **The Loco section** represents the data structures of Loco that are not in the shared part

These tree section together form the abstract grammar of our combined evaluator for Pico and Loco.

A.3.1 Shared Section

The original Pico and Loco abstract grammars use the same representation for numbers, fractions, text, tables and a void value. So we can share these same representations for the two evaluators:

```
<expression>  ::= <number>
<expression>  ::= <fraction>
<expression>  ::= <text>
<expression>  ::= <table>
<expression>  ::= <void>

<number>      ::= NBR <number>
<fraction>    ::= FRC <fraction>
<text>        ::= TXT <text>
<table>       ::= TAB <table>
<void>        ::= VOI
```

A.3.2 Pico Section

The Pico section of the abstract grammar consists of the original Pico abstract grammar minus data structures in the shared section of our grammar. Note that we renamed the tag for variables to `PVAR` to distinguish Pico variables from Loco variables that use the same original tag `VAR`.

```
<expression>  ::= <function>
<expression>  ::= <native>
<expression>  ::= <variable>
```

```
<expression>  ::= <application>
<expression>  ::= <tabulation>
<expression>  ::= <definition>
<expression>  ::= <assignment>

<function>    ::= FUN <identifier> <arguments>
                                   <expression> <dictionary>
<native>      ::= NAT <identifier> <function>
<variable>    ::= PVAR <identifier>
<application> ::= APL <identifier> <arguments>
<tabulation>  ::= TBL <identifier> <expression>
<definition>  ::= DEF <invocation> <expression>
<assignment>  ::= SET <invocation> <expression>

<identifier>  ::= <text>

<arguments>   ::= <table>

<invocation>  ::= <variable>
<invocation>  ::= <application>
<invocation>  ::= <tabulation>
```

A.3.3 Loco Section

Finally the Loco section consists of all Loco data structures that are not in the shared section. We also renamed the tag for variables to `LVAR` to be consistent with the tag for a Pico variable.

```
<expression>  ::= <assertions>
<expression>  ::= <negation>
<expression>  ::= <conjunction>
<expression>  ::= <disjunction>
<expression>  ::= <variable>
<expression>  ::= <pattern>
<expression>  ::= <fact>
<expression>  ::= <rule>
<expression>  ::= <dictionary>
<expression>  ::= <frame>
<expression>  ::= <result>

<assertions>  ::= AST <a-table>
<negation>    ::= NEG <query>
<conjunction> ::= CON <q-table>
<disjunction> ::= DIS <q-table>
<variable>    ::= LVAR <symbol> <qualifier>
```

```
<pattern>     ::= PAT <symbol> <p-table>
<fact>        ::= FCT <pattern>
<rule>        ::= RUL <pattern> <query>
<dictionary>  ::= DCT <symbol> <assertion> <d-link>
<frame>       ::= FRM <variable> <expression> <f-link>
<result>      ::= RES <frame> <continuation>

<a-table>     ::= <table>

<query>       ::= <negation>
<query>       ::= <conjunction>
<query>       ::= <disjunction>
<query>       ::= <pattern>

<q-table>     ::= <table>

<symbol>      ::= <text>

<qualifier>   ::= <number>

<p-table>     ::= <table>
<p-table>     ::= <variable>

<assertion>   ::= <fact>
<assertion>   ::= <rule>

<d-link>      ::= <dictionary>
<d-link>      ::= <void>

<f-link>      ::= <frame>
<f-link>      ::= <void>

<continuation>::= <number>
```

A.4 Summary

In this appendix we included the Pico and Loco abstract grammars for reference. We saw how our combined abstract grammar is organised and what changes had to be made to the original grammars. Note that although we described the grammar as tree different pieces does not mean that the respectively evaluators do not know about each other. When they must evaluate something they do not understand, they pass it on to the other evaluator.

Bibliography

[Ame78] American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. *ANSI Fortran X3.9–1978*, 1978.

[ASS85] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Susman. *Structure and Interpretation of Computer Programs*, chapter 5. MIT Press, second edition, 1996.

[BGW02] Johan Brichau, Kris Gybels, and Roel Wuyts. Towards linguistic symbiosis of an object-oriented and a logic programming language. In *Proceedings of the Workshop on Multiparadigm Programming with Object-Oriented Languages at ECOOP 2002*, 2002.

[BL02] N. Bouraqadi and T. Ledoux. Aspect-oriented programming using reflection. Technical Report 2002-10-3, Ecole des Mines de Douai, October 2002.

[Bru03] Programming Technology Lab : PROG Vrije Universiteit Brussel. Declarative meta-programming. http://prog.vub.ac.be/research/DMP/, januari 2003.

[Bud95] Timothy A. Budd. *Multiparadigm Programming in Leda*. Addison-Wesley Publishing Company, 1995.

[Bur74] W. F. Burger. Pascal manual. Technical Report CS-TR-74-22, University of Texas at Austin, Department of Computer Sciences, May 1 1974. Tue, 22 Jul 103 16:25:38 GMT.

[Chi98] S. Chiba. Javassist — a reflection-based programming wizard for java, 1998.

[DEDC96] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard Reference Manual*. Springer-Verlag, 1996.

[DGDP03] Theo D'Hondt, Kris Gybels, Maja D'Hondt, and Adriaan Peeters. Linguistic symbiosis through coroutined interpretation, 2003.

[D'H03] Theo D'Hondt. Pico: programming language. http://pico.vub.ac.be/, july 2003.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http, 1999.

[Fou03] Apache Software Foundation. The apache jakarta project. http://jakarta.apache.org/tomcat/, januari 2003.

[FS00] Martin Fowler and Kendall Scott. *UML Distilled: Applying the Standard Modeling Object Language*. Object Technology Series. Addison-Wesley, second edition, 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[Gyb03] Kris Gybels. A survey of object-oriented and logic multi-paradigm programming languages. 2003.

[HMU01] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to Automata Theory, Languages and Computation*. Low Price Edition. Addison Wesley Longman, Inc, Reading, Mass., USA, 2 edition, 2001.

[Hor51] A. Horn. On sentences which are true of direct unions of algebras. *Journal of Symbolic Logic*, (16):14–21, 1951.

[How95] Timothy G. Howard. *Smalltalk Developer's Guide to VisualWorks*. Cambridge University Press, 1995.

[HS] John Hughes and Jan Sparud. Haskell++: An object-oriented extension of haskell.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 318–326. ACM Press, 1997.

[IMY92] Yuuji Ichisugi, Satoshi Matsuoka, and Akinori Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture*, pages 24–35, 1992.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[jsc03] The jscheme web programming project. http://jscheme.sourceforge.net/jscheme/mainwebpage.html, januari 2003.

[Kam95] G. Kampis. Computability, self-reference, and self-amendment. *Communications and Cognition - Artificial Intelligence*, 12:91–110, 1995.

[kaw03] The kawa scheme system. http://www.gnu.org/software/kawa/, januari 2003.

[KCE98] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[KdRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[Kel03] A. Kellens. Using inductive logic programming to derive software views, 2003.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[Kiz98] Maxim Kizub. Kiev language specification. `http://www.forestro.com/kiev/`, 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KP96] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering*, pages 208–, 1996.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1988. 2nd edition.

[Leb02] Alexei Lebedev. What is self-similar? http://www.self-similar.com/what_is_self_similar.html, 2002.

[LH95] C. Lopes and W. Hursch. Separation of concerns, 1995.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification Second Edition*. Addison Wesley, 1999.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In *Proc. of the OOPSLA-87: Conference on Object-Oriented Programming Systems*, pages 147–155, Languages and Applications, Orlando, FL, 1987.

[Meua] W. De Meuter. Agora: The story of the simplest mop in the world - or - the scheme of object-orientation.

[Meub] W. De Meuter. All you ever wanted to know about continuations but were afraid to ask.

[Mic03a] Sun Microsystems. Javatm 2 platform, standard edition, v 1.4.1, api specification, package java.lang.reflect. http://java.sun.com/j2se/1.4.1/docs/api/java/lang/reflect/package-summary.html, januari 2003.

[Mic03b] Sun Microsystems. The source for java technology. http://java.sun.com/, januari 2003.

[MIKC92] Peter W. Madany, Nayeem Islam, Panos Kougiouris, and Roy H. Campbell. Reification and reflection in C++: an operating systems perspective. Technical Report UIUCDCS–R–92–1736, Department of Computer Science, Urbana-Champaign, 1992.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[MMR95] M. Muller, T. Muller, and P. Van Roy. Multiparadigm programming in oz, 1995.

[Mos94] Chris Moss. *Prolog++: The Power of Object-Oriented and Logic Programming*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[MS01] Duraid Madina and Russel K. Standish. A system for reflection in c++. In *Proceedings AUUG 2001: Always on and Everywhere*, 2001.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, USA, 1990.

[Nau63] Revised report on the algorithmic language algol 60. *Communication of the ACM*, 6(1):1–17, 1963.

[NM91] G. Nadathur and D. Miller. An overview of lambdaprolog. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proc. of the Fifth International Conference and Symposium (Volume 1)*, pages 810–827. MIT Press, Cambridge, MA, 1991.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[Pet89] Marian Petre. *Finding a Basis for Matching Programming Languages to Programming Tasks.* PhD thesis, Department of Computer Science, University College London, 1989.

[Phi98] C Phillips. *Language and Thought: The Sapir-Whorf Hypothesis.* 1998.

[qia00] Combining object-oriented and functional language concepts. *J. of Software*, (11):8–22, 2000.

[SA] Wolfram Schulte and Klaus Achatz. Functional object-oriented programming with object-gofer.

[SG93] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of Lisp. *ACM SIGPLAN Notices*, 28(3):231–270, 1993.

[sis03] Sisc: Second interpreter of scheme code. http://sisc.sourceforge.net/, januari 2003.

[Smi84] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35. ACM Press, 1984.

[Spi94] Diomidis Spinellis. *Programming Paradigms as Object Classes: A Structuring Mechanism for Multiparadigm Programming.* PhD thesis, Imperial College of Science, Technology and Medicine, London, UK, February 1994.

[Sta85] Richard Stallman. The GNU manifesto. *Dr. Dobb's Journal of Software Tools*, 10(3):30–??, March 1985.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, USA, 1991.

[Tou02] Tom Tourwé. *Automated Support for Framework-Based Software Evolution.* PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2002.

[Tur36] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, pages 230–265, 1936.

[Vol00] Detlef Vollmann. Metaclasses and reflection in c++. http://www.vollman.com/en/pubs/meta/meta/meta.html, 2000.

[WD01] Roel Wuyts and Stéphane Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *ECOOP 2001 International workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.

[Zav89] Pamela Zave. A compositional approach to multiparadigm programming. *IEEE Software*, 6(5):15–25, 1989.