Agora98 Language Manual

Wolfgang De Meuter
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels - Belgium
wdmeuter@vub.ac.be

Draft - November 14, 1997

Contents

1 Welcome to Agora98

1.1 What is Special about Agora98

Agora98 is a dynamic prototype-based object-oriented language. This means that it contains no classes. A programmer can build objects (called prototypes) 'out of the blue' without having to type in long static class declarations, and without having to go through a separate compilation phase.

There are several other prototype-based object-oriented programming languages. Examples are Self [9], Kevo [8] and Omega [1]. However, the prototype-based languages of the Agora-family (and thus also Agora98) fundamentally differ from these languages. Here are some of the most important characteristics of Agora98:

- The simplicity of Agora98 lies in the fact that it is entirely based on message passing. Extension of objects is accomplished by sending them messages. The methods that specify the nature of the extensions are called mixin-methods as their contents is mixed into the object upon invocation of the method. Creation of new objects is accomplished by cloning existing objects. In contrast to the conventional way of cloning objects using a cloning operator, Agora98 objects are cloned by sending them a message. The methods that are capable of delivering a clone of an object are called cloning methods.

- The fact that Agora98 (and all other Agora variants) is entirely based on message passing is also visible in its syntax. Agora98 programs are written by sending messages (so called reifier messages) to syntax prototypes. For example, a variable is declared by sending the `VARIABLE` message to an identifier. The identifier will thus install itself as a variable.

- Messages not only play an important role in the syntax and semantics of Agora98. They play the only role in it! This is reflected in the meta-object protocol of Agora98 which is very simple as it only contains message sending provisions. This corresponds very much to the meta level facilities of a language like Scheme, which only contains a function application operator.

- Representing Agora98 programs as messages to objects is not just a syntactic convention. Agora98 is implemented in an object-oriented fashion (in Java) and the semantics of Agora98 reifier messages is also really implemented by Agora98 messages sent to the Agora98 representation of syntactic constructions. So variables are really installed by sending the `VARIABLE` message to an Agora98 representation of an identifier.

- The way Agora98 syntax is specified, and the way Agora98 is implemented almost automatically turns Agora98 into a reflective programming language. Hence it is possible to extend Agora98 from within Agora98 by writing the behaviour of new reifier messages.

- Thanks to the way Agora98 messages are consistently mapped onto the underlying implementation language's messages (in the case of Agora98, this is Java), it is possible to send messages to the underying language. Hence, Agora98 allows full access to all Java API's.

1.2 The name 'Agora'

Often, people ask where the name Agora comes from. The answer is that the Agora language and its entire architecture and spirit were conceived by a number of people that 'wanted to do research together'. Everyone in the group was investigating a different aspect of the language (reflection, typing, semantics, calculus,...) such that the language was a common medium for these researchers. Hence the name Agora which means 'market square' in Greek: a place where people meet and exchange ideas.

Afterwards, we realized that Agora is also a **A** **G**reat **O**bject-oriented **R**eflective **A**rchitecture.

1.3 The Agora Evolution

Agora is an evolving language family. Thus it is important to clearly distinguish between the different implementations and versions. Here is a chronological overview of the existing implementations:

Agora

Agora is the term we will use to refer to an object-oriented framework for interpreting a space of object-oriented programming languages (OOPL's) of which Agora94 is a particular instance. The entire framework is written in Smalltalk and can be fetched from our FTP-site. It is documented by Patrick Steyaert's Ph.D. thesis (although the names used in the thesis do not entirely correspond to the identifiers used in the Smalltalk implementation) which can also be FTP'ed.

Agora94

Agora94 is a particular instance of the Agora-framework. Hence, Agora94 is a programming language while Agora is a generic interpreter for an entire design space of programming languages. The official description of Agora94 can be found in [4]. This paper is also available in electronic form.

Agora-S

Agora-S is a downscaled variant of Agora94 implemented in Scheme (hence the S). The difference between Agora-S and Agora94 is comparable to the difference between Scheme and more verbose functional programming languages like ML, Lisp or Haskell. Agora-S was designed to be a full-fledged programming language containing only a minimal number of constructs, much in the spirit of Scheme itself. Although Agora-S can be compared to Self [9] in its minimality, Agora-S fundamentally differs from Self in its underlying programming paradigm. Agora-S is a compiler compiling Agora-S code to Scheme. It is not very-well documented. Some minor documentation can be found on our WWW-site. The implementation can be FTPed for MacGambit.

Agora++

Agora++ is a port of the Agora-framework to C++. Although some design problems have been fixed, the framework has stayed pretty much the same.

The major difficulty we had to conquer in the Agora++ project was the static typing of C++ and the dynamic typing of Agora96.

Agora96

Agora96 is a particular instance of Agora++ in which some problems of Agora94 have been fixed and to which many extensions were made.

Agora97 (Agora96 2.0)

Agora97 is the same as Agora96, except that it has been provided with a graphical user interface. The interface originally was the front-end of Pico, a small language developed at our lab with the intention of teaching elementary programming to non computer science freshmen (see `http://progwww.vub.ac.be/`). The Agora96 evaluator itself did not change.

Agora98

Agora98 is a port of the Agora96 C++ code to Java. This involved the deletion of many *'s and &'s. Furthermore, the reflection has been reimplemented to allow full transparant access to the underlying Java (and thus also its API's). On top of this, the entire implementation of the reifiers has been revised in order to simplify the resulting language considerably.

1.4 More Information

More information about Agora can be found on the internet.

- The Agora home page:`progwww.vub.ac.be/prog/pools/agora.html`
- The documents about Agora on our FTP-site: `progftp.vub.ac.be` in the `agora` directory.
- Mail to Wolfgang De Meuter (`wdmeuter@vub.ac.be`), the implementor of Agora98.

1.5 The Agora98 System

There are two versions of the Agora98 system: Agorette and Agorique. Both versions have the same underlying evaluator. Their only difference is that Agorette is a Java applet while Agorique is a Java application. However, the same functionality is offered by Agorette and Agorique. The Agora98 system is still very basic. Both Agorette and Agorique show a text pane in which Agora98 expressions can be typed. After selecting an expressions, the `eval` button has to be pressed which gives rise to evaluation of the selected expression.

Important Note

Agora98 allows full access to all Java API's. If you use this feature, notice that Agora98 is implemented in Java 1.1. It uses no deprecated methods.

2 Agora98 Message Categories

Agora98 is a prototype-based object-oriented programming languages featuring nothing but objects and message passing to objects. Prototype-based object-oriented programming languages are class-free. Instead of using classes and instantiation to create new objects, only objects and messages are used. Since message passing is so fundamental in Agora98, this section concentrates on the sorts of messages Agora98 is equiped with. Section 2.1 introduces the three syntactic variants of messages (unary, operator and keyword) Agora98 uses. Section 2.2 introduces reifier messages. Reifier message are 'special messages' that are used to denote the Agora98 built-in syntactic constructs, just like spcial forms in Scheme. Section 2.3 discusses receiverless messages. Receiverless messages will be used whenever a message is sent to "the current interpretation context" (such as when looking up a variable).

2.1 Objects and Messages

Agora98 is a pure object-oriented programming language in the sense that *everything is the result of sending a message to an object.* As in Smalltalk, Agora98 messages can be classified as either unary, operator or keyword messages:

`x abs`	unary message `abs` to `x`
`x + y`	operator message `+` with `y` as argument
`x do:y with:z`	keyword message `do:with:` with `y` and `z` as arguments

Unary messages are messages without parameters. An example is the `abs` message that is sent to a number without any arguments. Once arguments are required, a keyword message should be used. Keyword messages consist of a number of keywords, each separating their name from their argument by a colon. For example, `gravityDict at:"Earth" put:9.81` is a keyword message `at:put:` sent to the object `gravityDict` with `"Earth"` as actual parameter for the `at` keyword and `9.81` as parameter for the `put` keyword. Operator messages are actually keyword messages that require one parameter. In such a case, the colon is not required since only one keyword is present. Operator messages are a combination of the characters $*$, !, @, #, \$, %, ˜, &, $-$, $=$, $+$, $<$, $>$, /, ., ',', | and ?. The following Agora98 expression illustrates all kinds of messages.

```
(java . "lang" . "System") out printString:"Hello World!"
```

First, the `.` operator message is sent to the `java` object with `"lang"` as argument. To the resulting object, again the operator message `.` is sent, but this time with `"System"` as argument. To the result of this message, we send the unary message `out` and to the result of this one, we send the keyword message `printString:` with `"Hello World!"` as the argument.

In the following example, an operator message `+` is sent to `5` with `9` as parameter. The operator message `*` with parameter `10` is sent to the result of the first operator message. Finally, the unary message `toString` is sent to the result of the latter operator message. The resulting string is used as argument for the `printString:` keyword message.

```
(java . "lang" . "System") out printString:((5 + 9 * 10) toString)
```

Although operator messages allow arithmetic to be written very naturally (i.e. we write `3+4` instead of `3+:4`), caution is required with the precedence rules.

The expression in the example must be read as a message passing expression (evaluated from left to right) and not as an ordinary arithmetic expression. The + operator message will thus be sent first. Parenthesis are required to force another evaluation order. Keyword messages have the lowest precedence and unary messages have the highest precedence (i.e. they are evaluated first).

More formally, the precedence rules are defined by the following grammar rules for expressions.

```
<message>     = <operatormsg>
              | <message><keyword>:<operatormsg>

<operatormsg> = <unarymsg>
              | <operatormsg><operator><unarymsg>

<unarymsg>    = <object> | (<message>) | <unarymsg> <unary>
```

2.2 Reifier Messages

As already said, an Agora98 program consists solely of messages sent to objects. To accomplish this, all keywords of Agora are message-sends to some object. For example, consider the following variable declaration:

```
hello VARIABLE: "Hello World"
```

This piece of Agora98 code must be read as sending the message `VARIABLE:` to the identifier `hello` with the expression `"Hello World"` as parameter. Such capitalised messages are called reifier messages as they literally reify[1] a message in the interpreter: that for declaring a new variable. In the section on reflection (section 7), we will see that reifier messages are actually implemented as message sends to Agora98 objects. Just like ordinary messages, reifier messages come in unary, operator and keyword form[2]. Here are some examples:

`hello VARIABLE: 5`	reifier keyword message `VARIABLE:`
`doSomething SUPER`	reifier unary message `SUPER`

Again, unary reifier messages have the highest precedence and keyword reifier messages have the lowest precedence. Reifier messages are always sent to a syntactic constructs such as identifiers or expressions. The interpreter literally sends reifier messages to these syntactic entities. For example, a variable declaration is a `VARIABLE:` message which is sent by the interpreter to the identifier specified by the programmer. The receiver (i.e. the identifier) will install itself as a variable with the argument as initial value.

Ordinary messages are evaluated by sending the message pattern to the evaluated receiver with the evaluated arguments (applicative order). Reifier messages are privileged messages that are intercepted by the interpreter in order to conduct the evaluation process in a special way. Consider for example

[1] To reify literally means to materialise.

[2] In the original definition of Agora, reifier messages were denoted boldfaced instead of using capitals. By switching over to capitals however, we lost the ability of having reifier operator messages since there is no capitalised analogue for the symbols used in operator messages. This doesn't give any problems since reifier operator messages weren't used anyway. In future implementations, we could replace the capitals back by boldfaced text, but of course this requires a decent programming environment.

```
condition IFTRUE: expression1 IFFALSE: expression2
```

The interpreter will recognise this as a reifier keyword message and will use the appropriate evaluation strategy for it: the evaluation of the arguments (`expression1` and `expression2`) must be delayed until the value of the condition is known. In other words, the `IFTRUE:IFFALSE:` reifier keyword message will be sent to the `condition` object thereby passing both expressions unevaluated. The difference between ordinary messages and reifier messages should sound familiar to Scheme [?] (or Lisp) programmers. Scheme contains values such as numbers and functions and allows a function `f` to be applied to arguments `a1 ... an` by putting parenthesis around it `(f a1 ... an)`. Scheme thereby prescribes applicative order evaluation. The keywords of Scheme have the same syntax as function application. Examples are `(lambda ...)`, `(define ...)`, `(cond ...)`, etc. These special function applications prohibit evaluation of their arguments. The functions themselves (such as `lambda`, `define`, `cond`, `if`, ...) are called special forms. Hence, reifier messages can be seen as the special forms of object-orientation. Reifier messages have precedence over ordinary messages. The resulting grammar integrating both ordinary messages and reifier messages is given below.

```
<message>     = <operatormsg>
              | <message><keyword>:<operatormsg>

<operatormsg> = <unarymsg>
              | <operatormsg><operator><unarymsg>

<unarymsg>    = <message> | (<message>) | <unarymsg> <unary>

<message>     = <operatormsg>
              | <message><keyword>:<operatormsg>

<operatormsg> = <unarymsg>
              | <operatormsg><operator><unarymsg>

<unarymsg>    = <object> | (<message>) | <unarymsg> <unary>
```

More intuitively, the precedence rules can be depicted as in figure 1.

The above grammar fragment explains most of the Agora98 syntax. Most of the design of the Agora98-language lies in the particular set of reifier messages that are implemented. As already stated in section 1, Agora98 is a particular concretisation of the Agora framework. This framework covers a variety of reflective object-oriented programming languages as long as their syntax can be fit into reifier message syntax. Therefore, deriving an instance of the Agora framework largely consist of implementing the pieces of the evaluator that take care of the evaluation of the designed reifiers. The bulk of this language manual explains the particular set of reifier messages that make up "the keywords" of Agora98. From section 7 on, we will delve into the reflective capabilities of Agora98 which will allow us to write our own reifiers in Agora98. Much of the implementation principles (such as the way reifiers are installed) will thereby be explained.

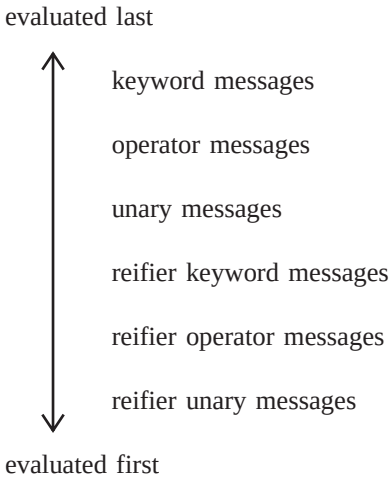

Figure 1: Evaluation Order

2.3 Receiverless Messages

Both (user) messages and reifier messages have an explicit receiver in the Agora98 program text. Reifier messages have a syntactic construct such as an identifier as receiver. Ordinary (user) messages have an 'ordinary' object as receiver. E.g., in the first example, the receiver of the `printString:` message is the `out` object. All such messages will be called receiverful messages. Hence, receiverful messages are user messages or reifier messages with an explicit receiver in the program text. Apart from receiverful messages Agora98 also features receiverless messages. Receiverless messages ought to be considered as messages to the interpreter, or more precise, messages to the current interpretation context. Examples of receiverless messages are

`myDict`	receiverless unary message
`SELF`	receiverless reifier unary message
`at:index put:it`	receiverless keyword message

Hence, receiverless messages can be considered as messages as defined by the above grammar. The difference with ordinary messages is that their receiver is syntactically missing. In the grammar, the receiver of ordinary messages is denoted by `<object>`. Receiverless messages thus correspond pretty much to function calls.

Lexical scoping of receiverless messages

Receiverless messages are sent to the current interpretation context. Since Agora98 is a lexically scoped language, the current interpretation context can also be seen as the current lexical scope. During the interpretation of Agora98 programs, the evaluator always uses an environment of currently visible items. This environment thus contains all names that are currently visible according to the lexical scope rules. Receiverless messages are sent to this environment.

The same goes for receiverless reifier messages like `SELF`. For the moment, it is not important to precisely understand how it all works. Just keep in mind that receiverless messages belong to the Agora98 syntax and that they are sent to the lexical scope. Hence, when sending a receiverless message, its declaration should have occurred somewhere in the lexical scope of that receiverless message. Also note that this scope is initially filled by a number of implicitly declared receiverless reifier messages such as `SELF`, `true`, `false`, `agora` and `null`. Of course, we will come back to receiverless messages later on.

2.4 Summary

The following table gives an overview of the kinds of Agora98 messages together with an example. Not all examples in this table have been really implemented. The table merely illustrates what the messages look like syntactically. We will discuss the precise set of implemented reifier messages in the following sections.

	ordinary
receiverless unary	identifiers and unary patterns (e.g. `abs`)
receiverless operator	operator patterns (e.g. `+ argument`)
receiverless keyword	keyword patterns (e.g. `at:index put:it`)
receiverful unary	`3 abs`
receiverful operator	`3 + 5`
receiverful keyword	`myDict at: 3 put: "Hello"`
	reifier
receiverless unary	`SELF`
receiverless operator	
receiverless keyword	`PROJECT: ... MAIN:...`
receiverful unary	`doIt SUPER`
receiverful operator	
receiverful keyword	`pattern METHOD: ...`

Hence, Agora98 messages vary along three dimensions:

- First, a distinction must be made between ordinary messages sent to Agora98 objects such as numbers or user created objects, and reifier messages sent to Agora98 syntactic constructs such as identifiers.
- Both kinds of messages come in receiverful and receiverless form.
- Finally, within the four categories formed by the above dimensions (i.e. receiverless messages, receiverful messages, receiverless reifier message and receiverful reifier messages) unary, operator and keyword messages can be used. In the current implementation, reifier operators are not used because we did not have the time to implement a decent Agora98 editor. As a consequence, reifiers are denoted by capitals instead of boldfaced text, and operators have no capitalised analogue.

3 Agora98 Objects

As explained above, the definition of Agora98 is determined by the set of reifier messages the Agora framework is instantiated with. This and the following

sections describe the set of reifiers the Agora98 instance of Agora is equipped with. This set can easily be extended, either by the reflective features of Agora98 (see section 7), or by directly changing the Agora98 instance of Agora. The reifiers discussed in this section are used to create new objects and to manipulate existing objects. In section 3.1 we explain the native Agora98 objects consisting of literals and underlying Java objects. From section 3.2 on, we explain how new objects can be created "ex nihilo" (i.e. from scratch) or by cloning an existing object. We will also explain how variables and methods are installed in objects. The inheritance mechanism and the reflective features of Agora98 are postponed to the following sections.

3.1 Agora98 Built-in Objects

Agora98 is a reflective language. This is a difficult term to indicate that it is possible to (partly) rewrite the Agora98 evaluator in Agora98. We will discuss this extensively in section 7. One of the requirements for having a reflective system is to access the underlying implementation level objects from within Agora98. If this is possible, it is said that the evaluator has reifying power, i.e. the evaluator can reify structures (read: objects) from its implementation language. There are three kinds of Agora98 objects that are actually reifications of the underlying Java objects. Literals (such as `3`) are implemented by a representation of the underlying Java literals. Second, there are objects created by accessing Java classes. These are created by the Agora98 programmer and have nothing to do with the evaluator. Examples hereof are windows, buttons and all other objects you create by accessing a Java class. The final kind of reified objects are the actual objects the evaluator is made of. Reifying these objects is what makes Agora98 reflective. This will be fully explained in section 7. The way literals and Java classes can be accessed from within Agora98 is explained here.

3.1.1 Literals

The first kind of native objects are those denoted by literals. These objects are automatically created when the evaluator encounters a literal. Examples are characters, strings, integers, reals, booleans and `null`. Apart from `null` these are all reifications of the associated classes in `java.lang`. I.e., an Agora98 integer literal is evaluated to an instance of the `java.lang.Integer` class. However, since this class does not implement a number of basic integer operations (such as +, -,...), we extended the `java.lang.Integer` class[3]. The same holds for characters and reals. The complete set of messages understood by the objects associated to Agora98 literals can be observed by inspecting one of these objects. So, evaluate the folowing expressions one by one and carefully inspect the interfaces of the objects:

```
3 inspect
```

```
3.14 inspect
```

[3] This is not entirely true because the `java.lang` classes cannot be extended because they are `final`. What we did is manually adding additional methods to the mechanism in the evaluator that makes the bridge between Agora98 messages and the underlying Java messages

```
"Example String" inspect

true inspect

false inspect

null inspect

'a' inspect
```

The interfaces you get to see in the inspector should speak for themselves. They are a mixture of the `java.lang` classes and our own additional methods. The latter (such as a + message on integers) should be clear. The former are explained in the `java.lang` documentation. The way Java methods are mapped onto Agora98 patterns is explained in the following section.

3.1.2 Interfacing Java API's

As already stated, Agora98 allows full and transparant access to all underlying Java code. Therefore, a variable `java` is included in the root object of the Agora98 system[4]. This is a so called 'package' object. A package object contains a string which is the name of the package. Hence, the `java` object contains the string `"java"`. It knows only one message `.` (dot) that takes another string as parameter. When sending this message, the package object contatenates both strings with a dot between them. If the resulting string is a valid java class name, a class object is returned. Otherwise, a new package with the concatenated strings is created. Hence, sending the message `java . "awt"` results in a new package object with string `"java.awt"`. This is because `java.awt` is not a valid class name. But if we send this package object the message `.` with argument `"Dialog"`, the `java.awt` receiving package object will concatenate both strings yielding `"java.awt.Dialog"`. Since this is a valid class name, the resulting class object is returned and no new package object is created.

Another way to get an Agora98 reference to a Java class object is to use the `JAVA` reifier. This reifier is sent to a string literal. If the string literal is a valid class name, the resulting Java class object is returned. If it is not a valid class name, an exception is thrown. In section 3.9.4 you will learn how to catch this exception in Agora98.

Now suppose that we get a valid reference to a Java class object. For example, we could evaluate `java . "awt" . "Dialog"` or, completely equivalently `"java.awt.Dialog" JAVA`. What can we do with this resulting class object? Of course, in order to understand what follows, you should now the basics of the Java language. Consult `www.javasoft.com` if this is not the case.

If you do have a valid reference to a class object, you can send it messages from within Agora98. Of course, the only messages a class object will understand, are constructors and methods or fields that were declared `static` in the corresponding Java class. The other messages only make sense if instances of the class object are created. Hence, class objects only understand

[4] This root object is accessible as the `agora` in the Agora98 system.

constructor messages, static field accessing messages and static method invocation messages. This can be noticed from inspecting a Java class from within Agora98, for example by evaluating `("java.awt.Dialog" JAVA) inspect` (or completely equivalent: `(java . "awt" . "Dialog") inspect`).

In order to send Agora98 messages to an underlying Java (class) object, there must be a mapping from Agora98 message patterns to the underlying Java methods, constructors and fields. In defining this mapping, we had to overcome two major problems:

- First, whereas Agora98 (and also Smalltalk!) uses keyword messages to denoted multi-argument methods, Java uses a parenthesis syntax. This means that a Java method `add(int i,int j)` can only be accessed from within Agora98 if we have an Agora98 pattern `add:XXX:` in order to denote the message in Agora98. But what should the `XXX` be ? One possibility is to duplicate the name of the method such that `add(int i,int j)` in Java would become `add:add:` in Agora98. However, this is a bit too simple:

- The above is complicated even further since Java has so called method overloading which means that two (or more) methods with the same name, but with different types may exist. Hence, the solution to the above problem that consist of duplicating the Java method name in the patterns would not work. This is because, if we have two Java methods named `add(int i,int j)` and `add(int i,float k)`, they would both be accessed by the pattern `add:add:`, which is not possible since the evaluator cannot know which method to choose when sending the `add:add:` message from within Agora98.

The second issue indicates that the type of the underlying Java method must be a part of the message pattern in Agora98. The solution we opted for is the following. Any Java method `m(t1 v1,t2 v2,... ,tk vk)` with name `m` and argument types `t1`, `t2`, ..., `tk`, is[5] mapped onto the Agora98 pattern `mt1:t2:...tk:`. For example, consider the method `void insert(MenuItem menuitem,int index)` from the `java.awt.Menu` class. If you have a menu object `m` (not to be confused with the `("java.awt.Menu" JAVA)` class object), and a menu item object `i1` and an integer object `i2`, you can invoke the method by typing `m insertMenuItem:i1 int:i2`. That is, the method is mapped onto the Agora98 message pattern `insertMenuItem:int:`. Of course, this rule is only useful for Java methods that take one or more arguments. Other Java methods will result in a Agora98 unary message. For example accessing the `String paramString()` method on a menu object `m` can be done in Agora by the message `m paramString`. Hence, Java methods without arguments are mapped onto Agora98 unary patterns and Java methods with arguments are mapped onto Agora98 keyword patterns. The first keyword is the result of combining the method name with the first argument type, and from the second keyword on, the argument types are used.

Now that we now how to access (public!) Java methods from within Agora, we can turn our attention to the way Java (public!) instance variables are accessed. As we will see in section 3.3.2, a Agora98 public variable `v` is read

[5] Notice that the names of the arguments are not accessible at runtime. This is because the reification in Agora98 is implemented by using `java.lang.reflect` in JDK1.1, and this package gives us no way to get the names of the arguments of a method.

by sending the containing object the message `v`. It is written by sending the message `v:`. The same access method has been made to access Java public fields. As an example, suppose you have an object `gbc` created by instantiating (you will learn about instantiating Java class objects in a minute!) the `java.awt.GridBagConstraints`, you can read its public field `gridy` by sending the message `gbc gridy`, and you can write the field by sending e.g. `gbc gridy:34`.
The final thing that needs to be covered concerning methods and fields are the following two pathological cases.

- There is a small problem when a Java (method, class or field) name consist of capitals only. This name cannot be used in Agora98, since capitals are reserved for reifier messages in Agora98. This is solved by putting a small `j` right before the name. Hence, the public variable `NAME` of some Java class is denoted `jNAME` seen from within Agora98.

- The second problem emerges when using type names (in keyword messages) as described above. If the Java type is an array type, the name will contain the characters `[` and `]`. As we will see, these brackets are reserved syntax in Agora98. The solution for this is to put a capital `A` behind the name of the type, if it concerns an array type. Hence, if a we have a Java method `void foo(int[][] i,int[] j, int k)`, the corresponding message in Agora98 is `foointAA: intA: int:`.

The complete algorithm for mapping Java methods, fields and constructors onto corresponding Agora98 patterns will be summarised at the end of this section.

Up until now, we have described how to get a reference to any Java class (using the `java` package variable in the `agora`, or by using the `JAVA` reifier), and how to send messages and access fields in any object that corresponds to a Java class. The final thing that needs to be solved is how these objects can be created given the appropriate Java class object. That is, we still need to describe how constructors can be called from within Agora98. The way to do this is really simple: every constructor is mapped onto the Agora98 message name `new`. If the constructor takes one (or more) arguments, a keyword message is taken and the same rules as above are used. Hence, in order to create a new object of the class `java.awt.Label`, we must first access the class itself using e.g. `"java.awt.Label" JAVA` and then invoke one of its constructors. This class has three constructors named `Label()`, `Label(String text)` and `Label(String text,int alignment)`. This means that there are three ways of creating an instance of the class: `("java.awt.Label" JAVA) new`, `("java.awt.Label" JAVA) newString:"any"` and `("java.awt.Label" JAVA) newString:"any" int:4`.

3.1.3 Summary

Let us summarize the way to access a Java method, field or constructor.

- A Java field named `f` is read by sending the message `f` and is written by sending `f:`.

- A parameterless Java method named `m` is invoked by sending the unary message `m`.

- A Java method with arguments is mapped onto an Agora98 pattern with several keywords. The first keyword is a combination of the method name and the type of the first argument. The remaining keywords are the types of the corresponding arguments.
- Constructors are always named `new`. If arguments are needed, the keyword syntax is used. The first keyword is a combination of `new` and the type of the first parameter. The remaining keywords are the types of the corresponding arguments in the constructor.
- If any of the type names is an array, a capital `A` is added to the type name.
- If any of the method or field names is completely capitalized, a small `j` is added in front of the name

In the following sections, we will learn some more Agora98 syntax. This will enable us to give an example of all these rules.

3.2 Ex-Nihilo Created Objects

Besides the built-in Agora98 literals, and the objects created from accessing Java classes, you will probably also want to create your own objects that combine the above primitive constructs. Therefore, an essential concept in Agora98 are ex-nihilo created objects. Ex-nihilo created objects group a number of expressions that are evaluated in the context of a "newly created" object. These expressions must be enclosed in `[` and `]` brackets and they must be separated by semicolons. The expressions are evaluated from left to right.

```
[ ...; ...;  . . . ; ... ]
```

Ex nihilo created objects are best described as 'objects created from scratch'. Ex nihilo created objects are pretty much like 'lambda's' in a functional programming language like Scheme. Whereas functional languages allow the programmer 'to just write down a function' using lambda notation, Agora98 allows the programmer to write down an object 'out of the blue' using the brackets syntax. The semantics of ex-nihilo created objects is to first reate a new object consisting of empty dictionaries, and second, to valuate each expression between `[` and `]` (from left to right) in the context of that new object. The empty dictionaries are linked to the dictionaries of the top level object in the Agora98 system. This object is known as the `agora` of the Agora98 environment. The `agora` is similar to Self's `lobby` object. The `agora` already contains a suite of standard variables and methods, and since each object is linked to the `agora`, each object automatically contains these items. In order to learn about them, evaluate `agora inspect`.

In section 3.3 and section 3.4 we will describe how ex-nihilo created objects can be filled with new variables and methods. In section 3.10 we will explain how these objects can be cloned. Inheritance and extension of objects is postponed to section 4.1.

3.3 Variables

Ex nihilo created objects contain expressions which are, of course, messages to objects. Some of these messages will install new items in the object while other messages will only use already installed items; making use of the fact that the expressions between `[` and `]` are evaluated from left to right. An example of such items is the notion of a variable.

3.3.1 Variable Declarations

Variables are declared by sending the `VARIABLE` unary reifier message or the `VARIABLE:` keyword reifier message to an identifier. The first message is used to declare a variable while the latter is used to declare a variable with an initial value. For example,

```
[
  window1 VARIABLE: ("java.awt.Frame" JAVA) newString:"Untitled";
  window2 VARIABLE;
  SELF window1 show;
  SELF window1 hide;
  SELF window2: (SELF window1);
  SELF window2 show;
  SELF window2 hide;
]
```

If your system is not too fast, this example will show and hide a window named 'Untitled' two times. In line 1, a new window object is created by invoking the appropriate constructor in the corresponding Java class. A new variable called `window1` is created with the new window as initial value. In line 2, a new variable `window2` without initial value is installed in the object. In line 3, the message `show` is sent to `window1`. In line 5, the `window2` variable is assigned as the value of the `window1` variable. Line 5 is particularly interesting as it illustrates the way variables are read (just send their name to the containing object) and the way variables are assigned by using their name followed by a colon. Notice that the above example evaluates to an object with two variables. All the other expressions inside `[...]` can be thought of as 'initialisation code'. The are evaluated one by one, but don't install new slots in the freshly created object.

Agora98 is a slot-based language. Objects consist of slots which can only contain methods. This is a direct consequence of the importance of message passing in Agora98. Because objects can only be accessed through message passing, a variable declaration will install two slots in an object. One slot contains a method to read the variable. The other slot contains a method to assign the variable. Hence, a declaration of a variable `window1` will install a unary message `window1` in order to read the value of the variable, and a keyword message `window1:` with one argument that can be used to set the value of the variable. This is the same as e.g. in Self [9]. In Agora98, variables can be explicitly declared only to contain a read slot. Such read-only variables are called constants and are installed by the `CONST:` reifier keyword message. Of course constants only make sense when they have an initial value such that we do have the `CONST:` reifier, but we do not have the `CONST` reifier. The previous

example can be rewritten as follows, without a loss of semantics. This is because the `window1` variable is never assigned in the program.

```
[
  window1 CONST: ("java.awt.Frame" JAVA) newString:"Untitled";
  window2 VARIABLE;
  SELF window1 show;
  SELF window1 hide;
  SELF window2: (SELF window1);
  SELF window2 show;
  SELF window2 hide;
]
```

Agora98 objects consist of a public and a local part which are both collections of slots. The public part of an object is accessible by anyone that has the object as an acquaintance. The local part of an object is only accessible by the object itself and by objects nested in it thereby following the well-known lexical scoping rule. Variables (i.e. receiverless unary messages) or constants can be made public or local by sending the appropriate reifier messages to them at declaration time. We therefore implemented two special unary reifiers: `PUBLIC` and `LOCAL`. Here's an example:

```
account VARIABLE: [ amount  PUBLIC VARIABLE: 0;
                    password LOCAL VARIABLE: "007"
                  ]
```

If we use both unary messages `PUBLIC` and `LOCAL`, the variable or constant will be both public and locally accessible.The default is public. Hence, when not sending the `PUBLIC` or `LOCAL` unary messages, the corresponding variable, constant (or method) will be installed in the public part of the object. Hence, the `account` variable will be installed in the public part of the `agora` when evaluating the above expression.

3.3.2 Variable Accessing

From the above, it follows that Agora98 knows three kinds of slots: public slots, local slots and public-local slots. Let us now focuss on the way this affects the usage of variable or constant slots:

- A public variable `v` is visible to anyone that is aware of the object that contains the variable. It can only be accessed by sending the messages `v` or `v:` to that object. Consider for example:

  ```
  [
    account PUBLIC VARIABLE: [ amount PUBLIC VARIABLE: 0
                             ];
    SELF account amount:3000;
  ]
  ```

  When a public variable is needed by the object containing that variable, the object should send the name of the variable to itself. Hence, the

`account` message is sent to `SELF` for accessing the public `account` variable in the outer object. Once we have the `account` variable (through `SELF account`) we can send it the messages `amount` and `amount:` for reading or writing the `amount` variable that resides in that object. This is possible since the `amount` variable is a public variable.

- Local variables are only accessible by those objects that are able to see the variable though lexical scoping.

```
[
  account PUBLIC VARIABLE: [ amount   PUBLIC VARIABLE: 0;
                             password LOCAL VARIABLE: "007"
                           ];
  clerk LOCAL VARIABLE: "Ben Tium";

  SELF account amount:30;
  SELF account password:"008"; ===> ERROR!(password is local)
  account amount:34;           ===> ERROR!(account is public)
  SELF clerk: "Max Intosh";    ===> ERROR!(clerk is local)
  clerk: "Max Intosh";
]
```

In this example, the `password` variable is not accessible for the outer object as it is local to the `account` object. Furthermore, trying to update the local `clerk` variable by sending `clerk:` to `SELF` also results in an error since `clerk:` is not in the public interface of the outer object because the `clerk` variable is declared local to that object. But it is possible to access the `clerk` variable. Instead of using a self-send, local variables are accessed through a special syntax. Since local variables are only accessible by objects with the right scoping, accessing local variables is determined by the interpretation context. In the previous section, we have seen that anything depending on the interpretation context is denoted by a message to that context. In the text of an Agora98 program, such messages are denoted by receiverless messages. Hence, accessing local variables must be accomplished by receiverless messages. Here is the rule: if `clerk` is a local variable it must be read by the `clerk` receiverless unary message, and it must be written by the `clerk:` receiverless keyword message. The way local variables are accessed is illustrated in the outer object where the `clerk` variable is re-assigned using the receiverless unary message `clerk: "Max Intosh"`.

- The final category of variables are variables that are declared both local and public. It is important to realise that only a single copy of the variable exists which happens to be both publicaly and locally accessible. Hence, from the outside, a local public variable is only accessible using ordinary message passing , while it is both accessible using ordinary message passing (to `SELF`) and using receiverless messages by the object itself. Hence, the semantics of `SELF` is that it denotes the object itself, that is, the object as it can be seen by other users of the object. Hence, the messages that can be sent to `SELF` are all public. If an object wants to access a local variable, it should use a receiverless message.

3.3.3 Scope Rules Of Agora98

Let us summarise what we have seen so far concerning the way variables are accessed. An object `o` can contain a variable `v` that is declared public. Users of the object can read the value of that variable by the `o v` message. They can change the value using the `o v:` message. Variables that are declared local are only accessible by code that has the declaration of the variable in its lexical scope. If a statement s 'sees' a local variable `v` it should not try to access `v` using `SELF v` since this will result in a `message not understood error` because `v` does not belong to the interface exposed by `SELF`. Sending a message to `SELF` is like sending a message to ones own interface, which is of course the interface other clients will be able to see. Therefore, all access to local entities must be accomplished through receiverless messages. Receiverless messages are automatically sent to the local part of a object. This local part consists of the objects real local part plus the local parts of the objects in which it is nested. Hence the local part of an object is nothing but the lexical scope of that object.

3.3.4 Ex-Nihilo Created Objects Again

Now that we now how identifiers are being looked up in Agora98, we can further elaborate on the scope rules of ex-nihilo created objects. An ex-nihilo created object is an object created from scratch. As such, ex-nihilo created objects initially have an empty public part. The local part of ex-nihilo created objects is also initially empty but is linked to the private scope of their definition. Hence, conceptually, the local part of an ex-nihilo created object is initially filled by a reference to the entire local part of its (textually) surrounding object. All these local attributes are accessed using receiverless messages. Figure 2 should elaborate on the distribution of local parts when ex-nihilo created objects are nested into each other.

Each rectangle in figure 2 represents a newly created object. All of them are created using an initially empty public part. Their initially empty local part is lexically scoped (read 'linked') as indicated by the arrows. When a receiverless message is encountered it will be looked up following the arrows of the figure.

3.3.5 Summary

An Agora98 object is a collection of slots containing the messages it understands. The slots are divided into a local part and a public part. Slots in the public part are accessible by anyone that is aware of the object. Local slots are only accessible by the object itself and by any other object that is declared in the lexical scope of the former. Public slots are accessed by sending the object a message. Local slots are accessed using a receiverless message that is sent to the current lexical scope. Slots declared both public and local are accessible in both ways. Particular kinds of slots are read and write slots for variables or read slots for constants. If `v` is a variable, it is read though the `v` slot and it is assigned using the `v:` slot. Ex nihilo created objects are constructed with an empty public part and an empty local part which sees the entire lexical scope of its textual appearance. The identifiers in this lexical scope are also accessed by using receiverless message.

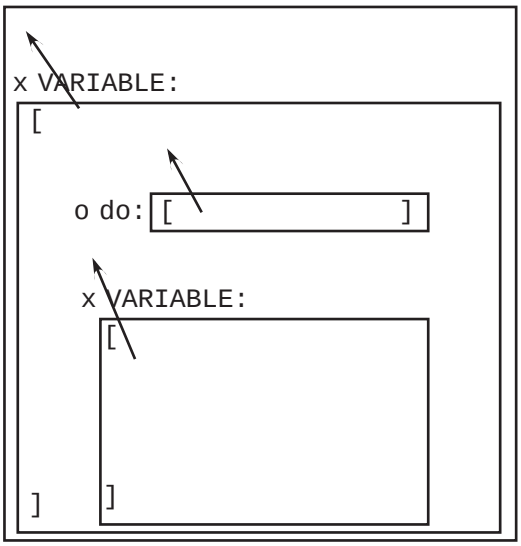

Figure 2: Scope Rules of Local Parts

3.4 Methods

Beside variable accessor slots, a newly created object can also list a number of method slots. Methods specify the dynamic behaviour of objects. They are installed in an object by sending the `METHOD:` reifier keyword message to the pattern forming the formal parameters of the method. The argument of this reifier keyword message is an expression constituting the body of the method. Here is an example.

```
[
  complex LOCAL VARIABLE:
          [ real PUBLIC LOCAL VARIABLE: 0;
            imag PUBLIC LOCAL VARIABLE: 0;
            + c METHOD: { real: (real + c real);
                          imag: (imag + c imag)
                        }
          ];

  complex real:17;
  complex imag:28;
  complex + complex;
  complex real inspect;
  complex imag inspect;
]
```

This program shows a complex number with one method. The operator message `+` is declared by sending the `METHOD:` reifier to the receiverless operator

message (i.e. operator pattern) `+ c`. Blocks { ... } are used if the body of a method contains more than one message expression. Hence, the usage of blocks for bodies of methods follows the same rule as the conventional begin-end pairs in an Algol-like block structured language. Notice that there is an enormous difference between the curly braces {} and the brackets `[]` notation. The latter are used to create a new object. The former are used to group a set of Agora98 expressions that are evaluated in the context of invocation. Hence no new object or context is created by a {...} expresion. The notation only serves as a grouping facility. The value of a {...} expression is the value of its rightmost expression as the expressions are evaluated from left to right.

It is very important to notice that actual parameters of a method depend on the interpretation context and are thus accessed by receiverless messages. This is e.g. illustrated in the `+` method body in the above example where `c` is accessed by a receiverless unary message `c`, just like local variables are. Assignments to actual parameters are only visible in the body of the method. Changes made to actual parameters by sending them a message are of course visible throughout the entire program: the actual parameters are objects like any other object and if a method decides to send a message to one of its incoming objects, that object might be changed.

The return value of invoking a method is the value of its body. If that body is a grouped expression, the return value is the result of evaluating the grouped expression, which is the result of evaluating the last expression of the group. Notice that when a method is invoked as a 'statement', its return value is ignored.

3.4.1 Public and Local Methods

Like variables, methods can be installed in the local part, in the public part or in both parts of an object. Public methods are accessible by any user of the object. Local methods are only accessible by statements that see the method through lexical scoping. Local methods are typically used for implementation issues of an object. The difference between public and local methods is also visible at the call-site of a method. Public methods are invoked by ordinary message passing syntax. Local methods are invoked using receiverless messages. This is illustrated in the following example.

```
{
  newton LOCAL VARIABLE:
          [ goodEnough:guess for:x LOCAL METHOD:
                   (guess * guess - x) abs < 0.001;
            guess: g for: x LOCAL METHOD:
                   x/g+g/2;
            sqrtIter:guess for: x LOCAL METHOD:
                 {
                   goodEnough:guess for:x
                    IFTRUE: guess
                    IFFALSE: (sqrtIter:(guess:guess for:x) for:x)
                 };
            sqrt: x PUBLIC METHOD:
                   sqrtIter:(SELF begin) for:x;
```

```
            begin PUBLIC VARIABLE:1.0
          ];

  ("java.lang.System" JAVA) out printString:((newton sqrt: 12)toString)
}
```

This program implements an object for calculating square roots using the well-known Newton approximation method. When y is an approximation of $\sqrt{x}$, the method predicts that $\frac{((\frac{x}{y})+y)}{2}$ is a better approximation. The object has two public methods. The `sqrt:` method must be invoked for calculating the square root of a number. The `begin` unary method returns the first approximation (in our case, 1.0) the algorithm will use. The other methods are implementation methods. The difference between calling local and public methods is illustrated in the body of the `sqrt:` method. Since `begin` is a public variable, it must be invoked using ordinary message passing syntax (to `SELF`). Since `sqrtIter:for:` is a local method, it is invoked by a receiverless message. As with variables, methods are public when the method is not annotated by `LOCAL` or `PUBLIC` in the program text. A client of an object can only invoke such a method using ordinary message passing syntax. The object itself can invoke the method by ordinary message passing to `SELF` since it is public.

3.4.2 Summary

Besides variable accessor slots, Agora98 objects contain method slots. The body of a method can be a message expression or a number of message expressions grouped in a block. Methods can be local, public or both local and public. Methods are accessed using the same rules as variables. Public methods can be invoked by anyone that is aware of the object by sending it a message. Local methods are visible to the object itself and to any other object that is in the lexical scope of the method. They are invoked by receiverless messages.

The return value of a method is the result of evaluating its body expression. If this expression is a group, the returned result is the value of evaluating the group, that is, the value of the last expression in the group. It is important to understand the difference between `...METHOD:[...]` and `...METHOD:{...}`. In the former case, a new ex nihilo created object is returned. In the latter case, the value of the last expression of the group is returned.

3.5 The Agora98 environment

Now that we have explained the bulk of the Agora syntax, we can give a few examples in which the Agora98 system is explained.

There are two versions of the Agora98 system: as an applet or as an application. The former is called Agorette, the latter is called Agorique.

When you startup Agorette or Agorique, a number of built-in variables and methods are already installed in the top level environment. This top level environment is called 'the agora of Agora98'. The agora is accessible through the `agora` variable. Inside the local part of the agora, a number of predefined receiverless messages are already installed. You can have a look at them by inspecting the agora using `agora inspect`. The `inspect` message is understood

by *each* object in Agora98. You can reimplement this message in your own ex nihilo created objects.

If you are running Agorette, a variable called `applet` is accessible. This is the applet itself. `applet` is `null` in Agorique. In Agorette, `applet` is an Agora98 object corresponding to a Java object of type `applet.Applet`. Hence, `applet` understands all the messages defined on this class. For example, if you evaluate `applet showStatusString:"Hello World"`, the string `"Hello World"` will appear in the status bar of your web browser.

3.6 Abbreviations

At first sight, programming in Agora98 seems a very verbose activity because of the long reifier names like `LOCAL` and `VARIABLE:`. Therefore, we have included a number of abbreviations for these reifiers. They are listed in section 6.

3.7 Arrays

Arrays are declared by sending the `ARRAY` unary reifier message or the `ARRAY:` keyword reifier message to a integer expression indicating the size of the array. The parameter in the case of the `ARRAY:` message is the initial value with which the entries of the array will be filled.

```
{
  numbers1 LOCAL VARIABLE: (5 ARRAY);
  numbers2 LOCAL VARIABLE: (5 ARRAY:10);
  numbers1 inspect;
  numbers2 inspect
}
```

Notice that the parameter in the case of the `ARRAY:` message will be evaluated for each entry. Hence, the initial value 10 in this example will be evaluated 5 times. Hence, if the number 10 would have been replaced by a message, the message will be sent 5 times. This is handy if one wants to initialise an array by a series of objects that are determined successively by a computation. This is illustrated by the following example:

```
{
   fac:n METHOD:( n = 0 IFTRUE: 1 IFFALSE: n*(SELF fac:n-1));

   counter LOCAL VARIABLE:0;

   facs LOCAL VARIABLE: (10 ARRAY:{ counter:counter+1; SELF fac:counter});

   facs inspect
}
```

In this example, a factorial method is implemented and a local counter is initialized to zero. Then a local variable `facs` is installed whose initial value is an array object. The array object has 10 entries and each entry is initialized by evaluating the argument of the `ARRAY:` reifier message. This group increments the counter and calls `fac:` with the counter. Since the last expression of the

group is the invocation of `fac:`, this will be the value of the group, and thus the value of that array entry.

Internally, array objects are represented as instances of the `java.util.Vector` class and the message defined in that class are the ones array objects understand. Try it out with `inspect` and a Java book at hand!

Note that Agora98 does not feature higher dimensional arrays. As in most modern languages, these have to be constructed with arrays of arrays. The following example illustrates this:

```
{
  matrix LOCAL VARIABLE:(10 ARRAY:(10 ARRAY));

  matrix inspect
}
```

3.8 Comments

The `COMMENT` reifier message completely ignores its receiver. Sending the reifier message to strings is the most common usage of the `COMMENT` reifier for commenting on your Agora98 code. However, `COMMENT` can be sent to any valid Agora98 expression, i.e. to any expression that gets through the parser. The following example illustrates this.

```
{
  "A comment usually looks like this" COMMENT;
  (x VARIABLE: [ a VARIABLE: 12 ]) COMMENT;
  "The above line is completely ignored by the evaluator" COMMENT
}
```

The second line of this example is a valid Agora98 expression. It consists of the `COMMENT` reifier message that is sent to another valid Agora98 expression. "Commenting away" complete expressions, as illustrated in this example, is a very handy technique for debugging programs.

3.9 Agora98 Control Structures

So far we discussed how objects can be created ex nihilo and how the objects can be filled with all kinds of methods and variables. Programming the methods of course requires a number of control flow instructions such as selections and iterations. In Agora98, these are also implemented using reifier messages. As such, the set of control flow instructions can easily be augmented with extra reifiers.

3.9.1 Selection Reifiers

The selection instructions are pretty much the same as in Smalltalk. There are four selections:

- The `IFTRUE:` reifier keyword message is sent to a boolean expression. Its actual argument will be invoked if that expression evaluates to the object `true`.

```
{
  0 = 0 IFTRUE:
            "Will be invoked" COMMENT;
  0 > 1 IFTRUE:
            "Will not be invoked" COMMENT
}
```

- The `IFFALSE:` reifier keyword will execute its argument expression whenever its receiver is the object `false`.

```
{
  0 = 0 IFFALSE:
            "Will not be invoked" COMMENT;
  0 > 1 IFFALSE:
            "Will be invoked" COMMENT
}
```

- The `IFTRUE:IFFALSE:` reifier is a combination of `IFTRUE:` and `IFFALSE:`. It is the object-oriented counterpart of the well-known if-then-else construction of imperative programming languages.

```
{
  0 = 0 IFTRUE:
            "Will be invoked" COMMENT
          IFFALSE:
            "Will not be invoked" COMMENT;
  0 > 1 IFTRUE:
            "Will not be invoked" COMMENT
          IFFALSE:
            "Will be invoked" COMMENT
}
```

- Except for its order, the `IFFALSE:IFTRUE:` reifier is identical to the previous combination of `IFTRUE:` and `IFFALSE:`.

3.9.2 Iteration Reifiers

Eight iteration reifiers are currently implemented:

- The `FOR:TO:DO:` construction represents a bounded loop. It must be sent to a variable that will vary during the loop. The following program prints the numbers from 1 to 10 in ascending order.

```
{
counter FOR: 1 TO: 10 DO:
  ("java.lang.System" JAVA) out printlnString:(counter toString)
}
```

  The variable to which the `FOR:TO:DO:` reifier is sent will be declared as a local read-only variable that is only visible in the body of the loop. If

a local variable with the same name was already declared, according to the lexical scope rules, the read slot of that variable will not be accessible in the body of the loop. The looping variable is of integer type. The boundaries of the loop should evaluate to integer objects. They will be evaluated only once, when the loop reifier is sent. The body of the loop can be any Agora98 expression including blocks.

- The `FOR:DOWNTO:DO:` construction has the same semantics but counts backwards.The following code prints the numbers from 1 to 10 in descending order.

```
{
counter FOR: 10 DOWNTO: 1 DO:
  ("java.lang.System" JAVA) out printlnString:(counter toString)
}
```

- A variant of the `FOR:TO:DO` reifier is the `FOR:TO:BY:DO:` reifier. It has an identical semantics except that the step size can be specified after the `BY:` keyword as an expression that evaluates to an integer object. Analogously, the `FOR:DOWNTO:BY:DO:` reifier is a variant of the `FOR:DOWNTO:DO:` reifier. Notice that the `FOR:DOWNTO:DO:` and the `FOR:DOWNTO:BY:DO:` reifiers are not essential. They can be simulated by `FOR:TO:BY:DO:` using a negative number as step size.

- The `WHILETRUE:` reifier must be sent to a boolean expression. Its argument is an expression that will be executed as long as the boolean expression stays true. This one prints the numbers from 1 to 10 in ascending order:

```
{
  counter LOCAL VARIABLE;
  counter: 1;
  counter<=10 WHILETRUE:
    {
       ("java.lang.System" JAVA) out printlnString:(counter toString)
       counter: counter+1
    }
}
```

- The `WHILEFALSE:` reifier has the same semantics but iterates the block as long as the boolean expression remains false.

- The `UNTILTRUE:` reifier iterates an expression until the boolean expression becomes true. It is guaranteed to iterate at least once thereby corresponding to the semantics associated to the repeat/until construction most imperative language feature. The semantics of `UNTILFALSE:` is analogous. The following example first counts from 1 to 10 and then counts backwards from 10 to 1 using these reifiers.

```
{
  counter LOCAL VARIABLE;
  counter: 1;
```

```
    {
      ("java.lang.System" JAVA) out println:(counter toString)
      counter: counter+1
    } UNTILETRUE:counter>10;
    {
      counter: counter-1;
      ("java.lang.System" JAVA) out println:(counter toString)
    } UNTILFALSE:counter>0
  }
```

Notice that while `WHILETRUE:` and `WHILEFALSE:` are sent to a boolean expression with the body of the loop as argument, `UNTILTRUE:` and `UNTILFALSE:` are sent to the body of the loop with the boolean expression as actual argument.

3.9.3 Objects as Iterators

Most functional programming languages feature no control structures. Instead, control structuring is accomplished through higher order functions like map, zip, accumulate,... . This is possible thanks to the ability of functions to take functions a arguments. In object-oriented programming languages like Smalltalk, higher order functions are usually mimicked by passing blocks performing a computation to methods. In Agora98, this is accomplished by passing an ordinary ex nihilo object containing a method (conventionally called `do:`) that performs the computation. In the next example, we have implemented a method `forall:` that performs a computation on every entry of the array. The method `forall:` assumes that its actual parameter is an object containing a `do:` method that will perform the computation.

```
{
  arrayObj LOCAL VARIABLE:
           [
            incr    LOCAL VARIABLE:0;
            content LOCAL VARIABLE: (10 ARRAY:{incr:incr+1;incr});
            forall:block METHOD:
             { counter FOR:0 TO:9 DO:
                 content insertElementAtObject:
                            (block do:(content elementAtint:counter))
                         int:
                            counter;
               SELF
             };
            inspect METHOD: content inspect
           ];
  square LOCAL VARIABLE:[do:x METHOD: x * x];
  succ   LOCAL VARIABLE:[do:x METHOD: x + 1];
  arrayObj: ((arrayObj forall:square) forall:succ);
  arrayObj inspect
}
```

This technique of object passing is often used to implement iterators over more complicated objects than numbers or arrays. At this point, an important remark for Smalltalk programmers must be made. The technique of passing around ex-nihilo created objects as iterators is very similar to the technique of passing around blocks in Smalltalk. However, there is a very subtle difference. In Smalltalk, a block is a object that can be activated by sending it the `value:` message. This can also be done in Agora98 by passing an ex-nihilo created object that implements a `do:` message. However, suppose the program text of the block refers to `SELF`. Smalltalk will then select the self of the object in which the block was created. So, performing self-sends in the code of a Smalltalk block will invoke methods defined in the object that created the block. This is where the behaviour of Smalltalk blocks differs from the behaviour of Agora98 objects. Agora98 ex-nihilo created objects are full-fledged objects that have their own self. Hence, performing self-sends in an ex-nihilo created object that is used as a 'block', will result in a method lookup in the block itself, and not in the object that created that block.

3.9.4 Exception Handling

Most modern languages feature exception handling. So does Agora98. In Agora98, the try mechanism might take the form of a reifier message

```
<expression1> TRY: <expression2> CATCH:<expression3>
```

The idea of this expression is to evaluate `expression1` and to invoke `expression3` whenever somewhere in `expression1` the exception `expression2` was raised by an

```
<expression2> RAISE;
```

statement. However, there is a small problem with this setup. Since at several places, several exceptions can be raised and since several handlers can exist, the Agora98 interpreter will have to dynamically compare `expression2` in each handler it encounters during its tour down the runtime stack. If the handler corresponds to the exception raised, `expression3` has to be evaluated. Otherwise, the runtime stack must be further killed in the hope that an appropriate handler exists in a higher level of the call-chain. Hence, what is needed is an equality operator on `expression2`-like objects such that the interpreter can compare it when it encounters a potential exception handler. However, since Agora98 is completely dynamically typed, the raise statement might evaluate to an object with an interface the `TRY:CATCH:` statement has no knowledge of. Comparing the expression raised by the `RAISE` statement with the one of the `TRY:CATCH:` statement might even result in a `message not understood error`, if one or both objects simply do not implement a comparison method. This problem is completely due to dynamic typing of Agora98 and any other dynamically typed language will have to face it too. Languages like C++ or Ada base their exception handling completely on type-information. The idea is to raise an exception of type t. This exception will be caught by a handler that knows how to handle exceptions of type t or subtypes of type t. We might simulate this by dynamically comparing the interfaces of `expression2` at both points of the program. However, this is a costly operation. The solution we first adopted for is to restrict the type of `expression3` to string objects.

```
<expression1> TRY: "Error 1" CATCH:<expression2>
"Error 1" RAISE
```

But then this has the problem that it is no longer possible to transfer objects from the place in the program where the error occurred to the place in the program where the error is handled. This is often an essential part of exception handling. E.g. a parser might raise an exception thereby transferring the erroneous line number to the handler. This handler could use the line number to print an error message in the style of "Error at line XX". Hence, we might adapt the mechanism with a parameter passing mechanism to transfer objects from the 'raise-site' to the 'catch-site' as follows:

```
<expression1> TRY:"Error 1" CATCH:<expression3> WITH: object
"Error 1" RAISE: [ .... ]
```

The idea is to bind the formal object parameter to the expression `[...]` with which the exception was raised. But if we evaluate this construct with enough criticism, we seem to end up with an exception handling mechanism that uses 'strings with parameters' as exception identity. But this is exactly the entire idea of message passing! Hence, Agora98 uses the following mechanism for exception handling:

```
<expression1> TRY: <formal-pattern> CATCH:<expression3>
<actual-pattern> RAISE
```

Here is an example.

```
parser parsesum TRY: parseError:line CATCH:
     { ("java.lang.System" JAVA) out printString:"A parse error occurred at line ";
       ("java.lang.System" JAVA) out printString: (line toString)
     };
...
parseError:3 RAISE;
```

Of course, actual parameters in a `RAISE` statement need not be restricted to simple objects like numbers. A parser might for example also raise the exception thereby passing the so-far created program tree, which might be unparsed by the exception handler.

Catching Implementation Exceptions

Besides catching exceptions raised by Agora98 code, it is also possible to dynamically catch an implementation level error. The following example illustrates this:

```
1 + "not a number" TRY: agoraError:exception
                   CATCH:
                    { exception inspect;
                      agoraError:exception RAISE
                    }
```

As we can see in the example, a reserved pattern `agoraError:` exists that matches any exception thrown by the evaluator. The exception itself becomes visible to the programmer through the argument of the `agoraError:` pattern. Hence, inside the 'catch-code', this exception can be manipulated like any other object.

The `agoraError:` pattern can also be used by `RAISE`. Its argument has to be a valid Agora98 exception. This exception is then internally raised in the evaluator. Hence, the above example catches the implementation level error (raised by `+`), inspects this error and re-raises the error in the interpreter.

3.10 Cloning Objects

Agora98 is a prototype-based language. Apart from creating objects ex nihilo, these languages allow objects to be created by cloning (i.e. shallow copying) existing objects. Due to its very simple meta-object protocol, Agora98 has a very special cloning mechanism. This section elaborates on it.

3.10.1 Cloning Methods

Agora98 has a very special cloning strategy. In order to understand it, we have to explain a bit about the theoretical principles behind Agora98. The (Java) class implementing Agora98 objects approximately looks as follows:

```
class AObject
    {
      private
      ...
      public AObject send(Pattern message, AObject[] arguments);
     };
```

This class specifies the protocol of Agora98 objects seen from the point of view of the meta-language (i.e. the implementation language) Java. This protocol is also known as the meta-object protocol (MOP). Agora98 is a language whose objects know a very high degree of encapsulation. This is clearly visible in the meta-object protocol, which was designed to be as elementary as possible. It contains only one single method: that for sending a message to the object. It was much inspired by the MOP of functional languages. Functional languages featuring meta-level access implement a meta-function `apply` for applying a function to a list of arguments. Likewise, we implemented the MOP of Agora98 to contain a single message: that for sending a message to the object together with a list of parameters. Hence, the only operation that is possible on objects is message passing. This has several consequences:

- *Encapsulated Inheritance*: Since Agora98 objects can only be subject to message passing, the inheritance mechanism for extending Agora98 objects must also be set to work by message passing. Hence, Agora98 objects can only be extended by sending them a message. This will be discussed in section 4.
- *Encapsulated Cloning*: Since the MOP only features a message passing method, it becomes impossible to clone an object 'from the outside' since

this would require an extra `clone` method in the MOP. Hence a reifier `CLONE` for cloning objects 'from the outside' (e.g. `myDictionary CLONE`) is not implementable using this setup. The MOP requires that cloning an object must be accomplished by sending the object a message. The object itself will thus (as a consequence of encapsulation) implement the necessary cloning operator itself.

Both encapsulated inheritance and encapsulated cloning are based on the fact that objects are fully encapsulated (featuring only `send` in their MOP) towards message passing clients, but know themselves unencapsulated. That is, the implementation of the `send` operation knows about the building blocks objects are made off. Objects are therefore perfectly capable of extending and cloning themselves. The latter is accomplished through so called cloning methods. Briefly spoken, cloning methods are methods that are executed in a copy of the receiver. Their return value is the clone of the receiver in which they were executed. The next example illustrates the most simple use of cloning methods. Cloning methods are installed by sending the `CLONING:` reifier to a formal pattern.

```
{
  complex LOCAL VARIABLE:
          [ real PUBLIC LOCAL VARIABLE: 0;
            imag PUBLIC LOCAL VARIABLE: 0;
            new  CLONING: {};
            + c  METHOD: { real: (real + c real);
                           imag: (imag + c imag)
                         }
          ];

  complex real:17;
  complex imag:28;
  complex2 LOCAL VARIABLE: complex new;
  complex2 real: 27;
  complex2 imag: 18;
  complex + complex2
}
```

This example illustrates the most simple cloning method one can imagine. It has an empty body. Hence, upon invocation of this cloning method, the body will be executed in a copy of the receiver (doing nothing since the method contains no code) and finally, that copy will be returned. Hence, the `new` cloning method of the example simply returns a copy of the receiving object. Notice that cloning methods are executed in a shallow clone of the receiving object. So, in the following example,

```
{
  sicp LOCAL VARIABLE: [ title  VARIABLE: "Structure And Interpretation";
                         author VARIABLE: "Abelson and Sussman";
                         new    CLONING: {}
                       ];
```

```
  course LOCAL VARIABLE: [ book LOCAL VARIABLE: sicp ;
                           student LOCAL VARIABLE:
                            [ name VARIABLE: "Eva Luator";
                              address VARIABLE: "Brussels";
                              new  CLONING: {}
                            ];
                           new  CLONING: {}
                         ];

  course2 LOCAL VARIABLE: course new
}
```

`course2` will be a shallow clone of `course`. The contents of `course` (i.e. `book` and `student`) will not be copied by the unary cloning method `new` of `course`. The reference to their value is just copied into the shalow clone. The following adapted version of the above example performs a deep copy of the `course` object.

```
{
  sicp LOCAL VARIABLE:  [ title   VARIABLE: "Structure And Interpretation";
                          author VARIABLE: "Abelson and Sussman";
                          new CLONING: {}
                        ];

  course LOCAL VARIABLE: [ book LOCAL VARIABLE: sicp;
                           student LOCAL VARIABLE:
                            [ name VARIABLE: "Eva Luator";
                              address VARIABLE: "Brussels";
                              new CLONING: {}
                            ];
                           new CLONING:
                            { book: book new;
                              student: student new
                            }
                         ];

  course2 LOCAL VARIABLE: course new
}
```

It is important to understand the `new` cloning method of the `course` object. Since it is a cloning method, it is executed in a shallow copy of the receiver. Hence the `book:` and `student:` messages will set the `book` and `student` variables in the clone. Since cloning methods perform a shallow copy of the receiver, the `book` and `student` variables point to the originals upon entering the cloning method. Hence, `new` will be sent to the old `book` and `student` objects, but the assignment happens in the 'self', i.e. the shallow clone of the receiver.

3.10.2 Using Parameter Passing With Cloning Methods

Cloning methods need not always be unary. Also, operator and keyword cloning methods can be used. The style of programming that emerges from using such methods strongly resembles the way Smalltalk objects are created using class

methods. The next example shows how parameter passing with cloning methods is used to initialise the newly created object.

```
{
  Complex LOCAL VARIABLE:
            [ real LOCAL VARIABLE;
              imag LOCAL VARIABLE;
              real:r imag:i CLONING: { real:r;imag:i };
              inspect METHOD: { real inspect;
                                imag inspect }
            ];

  c1 VARIABLE: Complex real:1 imag:2;
  c2 VARIABLE: Complex real:2 imag:2;
  c3 VARIABLE: Complex real:4 imag:4;
  (SELF c1) inspect;
  (SELF c2) inspect;
  (SELF c3) inspect
}
```

3.10.3 The Prototype Corruption Problem

Cloning methods are useful for avoiding the so called *prototype corruption problem* [1] most prototype-based languages suffer from. Consider the following program fragment taken from [1], written in an imaginary prototype-based language.

```
welcomeMsg := String;
welcomeMsg add:"Hello Fred!";
...
goodByeMsg := String copy;
goodByeMsg add:"Good Bye, Fred!";
```

A subtle error that may be difficult to trace has been made here. By sending the `add:` message to the `welcomeMsg` alias of the `String` prototype, the prototype has become destructively changed. In a completely different part of the program, the `String` prototype might be cloned as in the last two lines in the program fragment. Although the programmer would expect the `goodByeMsg` to be an empty string object, freshly created from cloning the `String` prototype, the `goodByeMsg` will unexpectedly contain `"Hello Fred!Good Bye, Fred!"` as value. The problem is that a prototype is cloned after it was destructively changed through an alias of the prototype. This problem is called *the prototype corruption problem*. The prototype corruption problem can be tackled from two different corners. Either we restrict the access priorities of aliases. This is a difficult task and needs extensive data flow techniques. For example, right after the first two lines in the code fragment, we could decide to assign `welcomeMsg` back to `String`, making it very difficult for a system to keep track of real prototypes and aliases of prototypes. Another route to a solution is to control the way objects are cloned. Of course, this is exactly what cloning methods do. Cloning methods allow a programmer to protect its prototypes in several layers. For example, we might decide to define the `add:` method as a cloning method. This

way, many aliases of the `String` prototype can be created, but if one tries to touch an alias as in the case of the `welcomeMsg`, a clone is returned. Hence, the strongest level of safety a prototype can require is achieved by declaring all its destructive methods as cloning methods. A more relaxed way of ensuring safety in the above code fragment is to allow the `String` prototype to decide the way in which it will be cloned. In the hypothetical language used by the example, the `String` prototype is cloned 'from the outside' by the `copy` 'operator'. The `String` prototype has no control over what is happening. This can be avoided if the `copy` 'operator' is replaced by a cloning method `copy` that reinitialises its internal state upon invocation. This is very natural in Agora98 since cloning methods are the only way to clone an object. Hence, encapsulated cloning as present in Agora98 is a solution for avoiding the prototype corruption problem.

3.10.4 Cloning Arrays and Primitive Objects

Since primitive objects such as numbers, strings and characters have no internal state (that is, they cannot be changed by sending them messages), cloning them is completely irrelevant. Hence, instance variables containing primitive objects should not be cloned. However, since Agora98 is a dynamically typed language, a (cloning) method often does not know whether instance variables contain a primitive object or a compound user created object. We therefore implemented a unary method `primitive` in every object. It returns true if the object is primitive and false if the object is a user created one. Using the method, cloning methods performing a deep clone, can test whether instance variables contain primitive objects or user objects that need to be further cloned. Array objects must be cloned manually by creating a new array of the same size and copy (clones of) the entries of the original array into the destination array.

3.11 Summary

In this section we saw how Agora98 objects are created 'from scratch' by simply putting a sequence of expressions between brackets. Such an object consists of a local part and a public part. The public part is empty and the local part is linked to the local part of the object in which the former is nested. If the object is created at the top level of the Agora system, its local part is linked to the local part of the ever existing `agora` object. Objects contain all sorts of attributes which reside in the local part, in the public part or in both parts. Attributes residing in the public part are accessible for everyone that is aware of the object and to the object itself by sending a message to `SELF`. Attributes residing in the local part are only accessible to the object itself by means of a receiverless message. Attributes are declared local or public by sending the `LOCAL` or `PUBLIC` reifier messages to them at declaration time. The first kind of attributes are variables. Variables are declared using the `VARIABLE` or `VARIABLE:` reifier messages. The declaration of a variable installs read and write slots in the appropriate parts of the object. Variables with an initial value can be declared constant by `CONST:`. The second kind of attributes we saw are methods. Like variables, methods can reside in the local part, the public part or both parts. Upon invocation of a method, its body code is evaluated and the value of this is returned. If the body code consists of more than one expression, they must be grouped between curly braces { and }. In that case the return value will be

the value of the last expression in the group. The final kind of attributes we encountered in this section are cloning methods. Cloning methods are methods whose body is executed in a shallow copy of the receiver. Like ordinary methods, cloning methods can have parameters in order to initialise the variables of the newly created clone. Methods can be programmed using several iteration and selection reifier messages. They consist of reifier messages for bounded loops (for-to and for-downto), if-then(-else) tests and unbounded loops (whiles and repeats). Finally, we saw how code `<exp1>` can be tried using the `<exp1> TRY: <formal pattern> CATCH:<exp2>`. Whenever somewhere in the computation of `<exp1>`, a `<actual receiverless pattern> RAISE` statement occurs, and the formal pattern matches the actual pattern, `<exp2>` will be evaluated.

4 Inheritance With Mixin-Methods

Agora98 is a prototype-based language featuring dynamic object extension. The latter is somehow tricky as it seriously interferes with encapsulation. This was investigated in [7]. The general idea is that when objects can be dynamically extended, some client of the object can always extend it in order to add new slots that improve its access priorities. This makes it impossible to create encapsulated objects. When objects can be extended from the outside, anyone that can see an object as an acquaintance, can also extend it for bad purposes. This problem was also briefly mentioned in [5]. The authors of this paper rule out dynamic object extension in their taxonomy because they could not define an appropriate encapsulation mechanism for it.

As explained in section 3, the Agora MOP only contains a `send` method such that objects are fully encapsulated. Apart from message passing, it is impossible to perform an operation on an object 'from the outside'. But, the fact that objects know themselves unencapsulated allows them to clone or extend themselves. This means that object extension must be accomplished through message passing and that the extension procedure must be statically determined as a method inside the object. In Agora98, the methods taking care of the extension procedure are called mixin-methods.

4.1 Extending Objects through Mixin-Methods

Agora98 allows an object to extend itself by means of special methods we refer to as mixin-methods. The body of a mixin-method must be a block that will be mixed into the receiver upon invocation of the method. When a mixin-method is activated, an extension of the receiver is created. The resulting object will inherit all information of the parent object. Furthermore, the mixin-method is allowed to override attributes of the receiving object. Hence, invocation of mixin-methods has a semantics comparable to other extension mechanisms of object-oriented programming languages.

4.1.1 Functional and Destructive Mixin-Methods

The following example introduces a mixin-method in a two-dimensional point that allows the point to become a circle. This mixin-method (installed by the

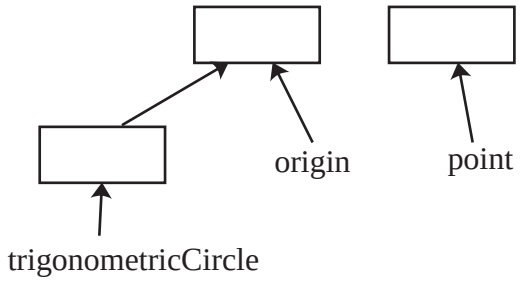

Figure 3: Functional Mixin-Method Applications

`VIEW:` reifier) is a functional mixin-method such that the original point still exists after the 'mixin' message is sent. Hence, the result of a invoking a functional mixin-method is a new object whose parent is the receiver.

```
{
  point VARIABLE: [ new CLONING: {};
                    x VARIABLE: 0;
                    y VARIABLE: 0;
                    circle: radius VIEW:
                      { r VARIABLE: radius
                      }
                  ];

  origin VARIABLE: (SELF point) new;
  trigonometricCircle VARIABLE: (SELF origin) circle:1;

  "ERROR! (origin is not affected!)" COMMENT;
  (SELF origin) r inspect;
}
```

Since the mixin-method is a functional one, it returns a functional extension of the object receiving the mixin message. Hence performing circle operations on the original origin point will result in an error. Functional mixin-methods (henceforth refered to as views) accomplish what makes prototype-based languages different from class-based languages: parent object sharing. In figure 3, the situation of the above example is depicted.

Mixin-methods can also be destructive thereby affecting all references to the object to which the 'mixin' message was sent. Destructive mixin-methods are installed with the `MIXIN:` reifier.

```
{
  Suzy LOCAL VARIABLE: [ cash PUBLIC LOCAL VARIABLE: 3000;
                         cinemaReduction METHOD: 50;
                         graduate MIXIN:
                          { cinemaReduction METHOD: 0 };
                         pay:amount METHOD: cash: cash - amount
```

```
                    ];

  doSchool:student LOCAL METHOD:
                { student graduate };

  seeCinema:person LOCAL METHOD:
                {  person pay: 100 - person cinemaReduction };

  Anne LOCAL VARIABLE: Suzy;
  "Anne is Suzy's alias trying to get a reduction" COMMENT;

  doSchool:Suzy;
  seeCinema:Anne;
  Anne cash inspect
}
```

Students like `Suzy` get a reduction of 50 at the entrance of the cinema as long as they have not graduated. After graduation the reduction invalidates. In the example, `Anne`, as an alias of `Suzy`, tries to get the 50 reduction after she has graduated. Fortunately the `graduation` mixin-method is a destructive one such that `Anne` as an alias (not to be confused with a clone) of `Suzy` will undergo the effect of the graduation mixin-method too. So `Anne` will pay the full price. In the example, the graduation mixin-method overrides the `cinemaReduction` method.

There are two remarks to be made here:

- First, both mixin-methods have a return value. In the case of views, this return value is the newly created object whose parent is the receiver. In the case of destructive mixin-methods, the return value is the receiver of the message. Hence, the result of a destructive mixin-method is the modified receiver.

- Destructive mixin-methods not only affect their receiver but *all the views laid on this receiver* as well. Hence, destructive mixin-methods destructively change the receiver, every alias (i.e. reference) of the receiver, and all descendants (i.e. views) of the receiver. Of course, clones of the receiver are left untouched. As such, in Agora98 it is possible to destructively change an entire hierarchy. Consider for example a hierarchy of graphical black and white objects. By applying a destructive 'colour' mixin-method to the root of this hierarchy, each object in the hierarchy will automatically inherit the 'colour' attributes. Hence, destructive mixin-methods make it possible to change an entire hierarchy of objects in one stroke. But this implies no loss of safety as the only way to do it is by sending the object a message (which happens to be implemented by a destructive mixin-method).

4.1.2 Closures

A notion very common to Scheme (or Lisp) programmers and users of λ-calculus is that of a closure. In Scheme [?], objects can be created as follows.

```
(define (make-account amount)
  (lambda (msg)
   (cond ((eq? (car msg) 'withdraw) (set! amount (- amount (cdr msg))))
         ((eq? (car msg) 'deposit) (set! amount (+ amount (cdr msg))))
         ((else (error "Message not understood in account")))))
```

The idea is to have a function `make-account` returning a function `(lambda (msg)...)` that dispatches messages. The implementation of the messages is allowed to see and change the parameters of the entire function (in our case `amount`) since the dispatcher is in the lexical scope of the entire function. Since `make-account` can be called several times, several versions of the dispatcher have to be created each having their own `amount` variable. Hence, each time the dispatcher is returned, Scheme has to create a copy of the entire lexical scope of the dispatcher. In other words, Scheme has to compute the reflexive and transitive closure of the lexical scope pointers starting in the dispatcher. Hence the name closures. Because mixin-methods can have parameters, and because mixin-methods are lexically scoped, Agora98 also features closures. Any method that was added to an object by a mixin-method is allowed to access the formal parameters of that mixin-method. This is illustrated below.

```
{
  makeAccount: amount VIEW:
  { withdraw:w METHOD: amount:amount-w;
    deposit:d  METHOD: amount:amount+d
  };

  a1 VARIABLE: SELF makeAccount:1000;
  a2 VARIABLE: SELF makeAccount:2000
}
```

Both `a1` and `a2` are accounts created by the `makeAccount:` view. Every method in that view can access the actual parameter of the view. Remember that parameters of (mixin) methods are accessed using receiverless messages because actual parameters can be seen as a temporal extension of the local part of the receiving object. The usage of closures corresponds to the use of local variables in the body of the view.

4.1.3 Public versus Local Mixin-Methods

Except from the fact that mixin-methods result in an extension of their receiver, mixin-methods are to be considered as ordinary methods in all their aspects. Like ordinary methods, mixin-methods can be public, local or a combination of these. As should be clear from the previous section, public mixin-methods are visible to anyone that is aware of the object. Local mixin-methods are of course not visible to clients of the object. They are only accessible by the object itself. As always with local attributes, local mixin-methods are invoked using a receiverless message. Local mixin-methods are often used when one wants to put restrictions on the applicability of a mixin method. Once the mixin-method becomes applicable, it is replaced by an public ordinary method that invokes the local mixin method. Consider for example the situation in which we want

to model males and females whose common properties are abstracted in the `MakePerson` mixin-method.

```
MakePerson PUBLIC VIEW:
{ ...
  MakeMale PUBLIC VIEW:
  { ... }
  MakeFemale PUBLIC VIEW:
  { ... }
}
```

Now suppose we want to model children by writing a `MakeChild` mixin-method. We do not want to allow the possibility that this mixin-method is applied immediately to persons. The `MakeChild` mixin-method should only be applicable to males or females. At first sight, the only option is to nest the `MakeChild` mixin-method both in the `MakeMale` and `MakeFemale` mixin-methods, but of course this kind of code duplication is highly unwanted. The solution is to put the `MakeChild` mixin-method at the same level as the `MakeMale` and `MakeFemale` mixin-methods, but declare it local. In the `MakeMale` and `MakeFemale` mixin-methods, an ordinary public method `MakeChild` is installed that invokes the local mixin-method.

```
MakePerson PUBLIC VIEW:
{ ...
  Child LOCAL VIEW:
  { ... }
  MakeMale PUBLIC VIEW:
  { ...
    MakeChild PUBLIC METHOD:
    { Child }
  }
  MakeFemale PUBLIC VIEW:
  { ...
    MakeChild PUBLIC METHOD:
    { Child }
  }
}
```

4.1.4 Parent References

Each mixin-method defines a frame containing attributes such as methods, variables, mixin-methods and so on. Method-lookup begins in the frame of the last sent mixin-method. This mechanism is responsible for overriding of (mixin/cloning) methods and variable accessor functions. Methods can perform operations in their parent through what is conventionally called super calls. In languages like Smalltalk, the parent is referred through a `super` pseudo variable. However, the `super` pseudo variable is a problematic language concept. Whenever a message is send to the `super` pseudo variable,

```
amethod METHOD: { ..... super amethod .....}
```

the `super` variable has to obey the late binding of self rule which means that all the self sends the code of the `super` variable performs should come to the original receiver. So, the self of the super must be the original receiver. However, if we have a `super` pseudo variable, we can also return it from a method since the `super` pseudo variable is an object like any other object.

```
amethod METHOD: super
```

In this case, we are actually returning the parent object. This object should have its own self reference because obeying the late binding of self rule would result in an object whose self is another object (the receiver). Hence, when the `super` pseudo variable is returned (or assigned to a variable), the self of the super must be the `super` itself.

Having syntactic constructs with a semantics that depends on the syntactic context in which it is used, is generally considered bad language design because the programmer has to keep two meanings of the construct in mind. In formal semantics, language concepts having more that one semantics depending on the context, are said to violate the principle of compositionality. Readers that have some knowledge of formal semantics immediately notice that the `super` pseudo variable does not have a compositional semantics.

In Agora98, this problem was solved by using a `SUPER` unary reifier message. Instead of sending a message to a `super` pseudo variable, the `SUPER` unary reifier message is sent to a pattern that will be looked up and invoked in the parent of the receiver, thereby always following the late binding of self rule.

```
amethod METHOD: [ ..... amethod SUPER .....]
```

The following example gives an example of how the `SUPER` reifier is used.

```
{
  Point VARIABLE:
   [ x LOCAL VARIABLE:0;
     y LOCAL VARIABLE:0;
     moveto:xx and:yy METHOD:{ x:xx;y:yy };
     Circle VIEW:
      { r VARIABLE:0;
        distFromOrig VARIABLE: (x square + y square) sqrt;
        grow:rr METHOD: r:rr;
        moveto:xx and:yy METHOD:
                  { moveto:xx and:yy SUPER;
                    distFromOrig: (x square + y square) sqrt
                  }
      }
   ];

  c VARIABLE: (SELF Point) Circle;
  (SELF c) grow:100;
  (SELF c) moveto:12 and:14;
  (SELF c) distFromOrig inspect
}
```

Notice that a selected method always belongs to a public or a local part. Each of these parts is attached to the corresponding part in the parent. The `SUPER` reifier will follow this link. Hence, super calls in a public method will be invoked in the public part of the parent while super calls in a local part will be sent to the local parent part (which is 'one level higher in the lexical scope of the object').

4.1.5 Overriding Cloning Methods

Since cloning methods are actually nothing but ordinary methods executed in a copy of the receiver, cloning methods can also be overridden. This means that is also possible to invoke a parent cloning method within a child cloning method using a super-send. Of course, when this is the case, only one copy of the entire receiver will be created. Each cloning method that is called (through a chain of parent references) is executed in the same copy.

4.1.6 Summary

This section discussed how mixin-methods can be used to create extensions of objects. The body of a mixin-method is a block whose contents will determine the extension. Like ordinary methods, mixin-methods can have formal parameters and can be declared local and public. Functional mixin-methods (installed by `VIEW:`) result in a new object that has the original receiver as 'subobject'. This new object can be considered as a view on the original object. Destructive mixin-methods (installed by `MIXIN:`)on the other hand, destructively change the receiver. No new object is created and every pointer to the receiver will notice the change. In the block of a mixin-method, one can specify attributes whose pattern is the same as some pattern in the parent. The latter is said to be overridden by the former. Attributes in mixin-methods can perform parent attributes through `SUPER`.

4.2 Nested Mixin-Methods

A powerful feature of mixin-methods is that they can be nested in each other.

4.2.1 The General Idea of Nested Mixin-Methods

The most simple way to think about nested mixin-methods is that a mixin-method B that is nested inside a mixin-method A cannot be applied before A is applied. The following example illustrates this.

```
{
  person: name VIEW:
       { getName    METHOD: name;
         setName:n  METHOD: name:n;
         print      METHOD: (SELF getname) print;
         female     VIEW:
                    { getName METHOD:
                        ("Miss " concatString: (getname SUPER))
                    };
         male       VIEW:
```

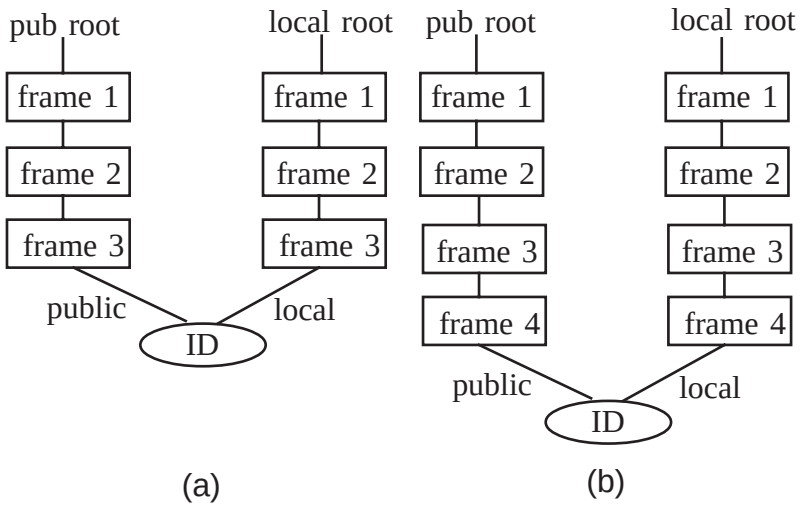

Figure 4: Destructive Mixin-Methods

```
                { getName METHOD:
                    ("Mister " concatString: (getname SUPER))
                }
        };

  Linda   LOCAL VARIABLE: (SELF person:"Linda") female;
  John    LOCAL VARIABLE: (SELF person:"John")  male;
  Linda inspect;
  John  inspect
}
```

This example declares a public `person` view containing some ordinary methods (getname, setname and print) and 2 nested mixin methods (`female` and `male`). These nested-mixin methods can only be invoked in objects to which the `person` view was already applied.

4.2.2 Scoping Rules of Nested Mixin-Methods

Figure 4.a. gives an idea of the structure of Agora98 objects. Every object consist of a local part and a public part which each consist of a number of linked frames. Each frame of the public part contains the public attributes of the mixin-method the frame originated from. Likewise for the local frames. Figure 4.b illustrates the effect of an imperative mixin method.

Mixin-methods can be nested into each other. As we explained previously, lookup of receiverless messages happens according to the lexical scoping of the program. Since these receiverless messages are used to access the local part of an object, the local part of an object is determined by the lexical structure of the program. Consider the following code fragment.

```
A VIEW:
```

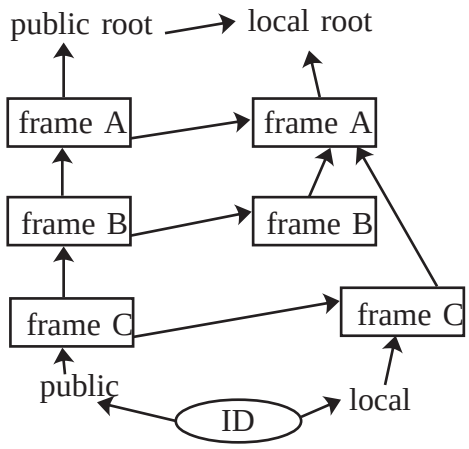

Figure 5: Scope Rules

```
{  B VIEW:...
   C VIEW:...
}
```

Suppose we use this code to create an object by successive application of `A`, `B` and `C`. Now suppose somewhere in the code of `C`, a receiverless message is sent to access the local part of the object. Since receiverless messages are looked up following the lexical structure of the program, the receiverless message should be found in the code of `C` or in the code of `A`. Since the contents of `B` is not in the lexical scope of `C`, the contents of `B` should not play a role in the method lookup of a receiverless message sent in the code of `C`. The object we created will thus have the structure depicted in figure 5

Each public frame will have an associated local frame which is linked following the lexical structure of the program. The local code resulting from sending the `B` mixin-method will be able to see everything that is in the local parts of `B` and `A` since `B` is nested in `A`. Likewise, the code of `C` is able to see the local code of `C` and `A` since `C` is nested in `A`. It is not allowed to see the local part of `B` since `B` is not in the lexical scope of `A`. This is the way objects are implemented in Agora98. The following more complicated example should further elaborate on the rules.

```
A VIEW:
  B VIEW:
    C VIEW:
    D VIEW:
      E VIEW:
  F VIEW:
    G VIEW:
      H VIEW:
```

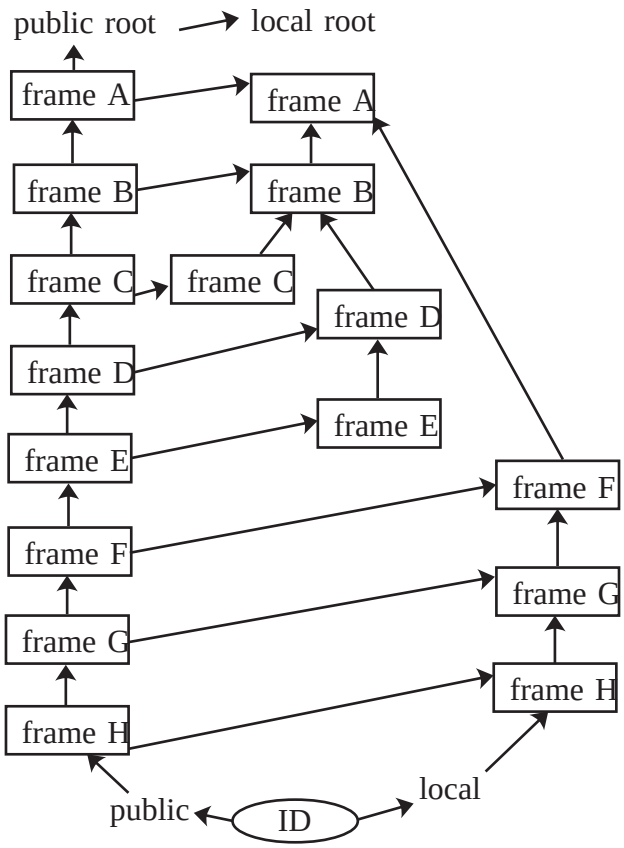

Figure 6: Scope Rules

Suppose we create a object by successive application of `A` through `H`. Figure 6 shows the resulting object structure.

This structure is important when invoking super sends from within a local method. Super sends from within a local part will also be looked up in the parent of the local part. Hence, super sends in the local part will 'jump a level higher in the lexical scope': the `SUPER` reifier sent from within a local part is much like the scope resolution operator :: in C++.

4.3 Extension from the outside

A fundamental critique against Agora94 and Agora-S was that mixin-method-based inheritance requires that all the potential extensions of an object are encapsulated within mixin-methods that reside in that object. Hence, an inheritance mechanism that is based on message passing alone (which is precisely what mixin-methods are), makes it impossible to extend an object 'from the

outside'. That is, when you buy objects from someone (e.g. a set of GUI objects), it is impossible to inherit from them unless the vendor has pre-installed all the mixin-methods you will ever need. Although this is a very fundamental objection against Agora, this kind of encapsulated inheritance can also be very useful. An example is the safety that is required in the distribution of objects over a network. Proof is the tremendous effort that has been made to make Java as safe as possible. Mixin-methods could enable dynamic inheritance in such languages, without giving up any kind of safety. But still, in 'ordinary programming', the mixin-method model isn't very realistic. We therefore worked on this problem and we can now safely say that this problem has been completely solved in Agora98. The solution is very Agora-like in the sense that extension from the outside is possible without circumventing the message passing philosophy of Agora. Since our solution uses the meta programming operators of Agora98, we cannot explain it at this point in the manual. Extension from the outside is therefore postponed until section 7.

4.4 Agora98 Evaluation Rules: Scope Creation

The precise evaluation rules for Agora98 will be explained in the section on reflection. However, in order to understand the evaluation rules of Agora98 and thus also the behaviour of Agora98 programs, it is important to understand when exactly a new scope is created. As already explained, each Agora98 object consist of a public and a private part. Hence, the context in which Agora98 code is executed can always be characterised by a triple $S = (object, local, public)$. The first component is an object whose local part is the second component and whose public part is the third component. At system-startup, there is a root local part $rootL$, a root public part $rootP$ and a root object $root$ whose local part is $rootL$ and whose public part is $rootP$. Every expression that gets through the parser is evaluated in the scope $(root, rootL, rootP)$. In the Agora98 environment, this tripple determines the `agora` object.

- When a method is executed in a scope $S = (O, L, P)$, a new scope is created $S' = (O, L', P')$. L' will contain the formals-actuals bindings and the local attributes that are possibly declared in the body of the method. Likewise, P' will contain the public attributes that are declared in the method body. No new object is created and when the method is finished, L' and P' are restored to L and P. Hence, the body of the method is evaluated in the scope S'.

- When a view is encountered in scope $S = (O, L, P)$, the scope is extended $(L => L')$ and $(P => P')$, a new object O' is created with L' and P', and the body of the view is executed in (O', L', P'). The result of the view is O'.

- Upon evaluating an mixin, L and P are extended towards L' and P' and L and P of O are reassigned to be L' and P'. The body of the mixin is evaluated in $(O(L', P'), L', P')$. The result of the mixin is $O(L', P')$.

- When a cloning method is evaluated, L and P are cloned (yielding L' and P'), a new object O' is created with L' and P' and the body of the method is evaluated in (O', L', P').

- When a receiverless message is evaluated in $S = (O, L, P)$, its pattern is looked up in L.
- When a self-send occurs, the send is done on O.
- When executing a `[...]` statement in a scope $S = (O, L, P)$, the local scope is extended (L => L'), the `agora` public scope is extended ($rootP$ => P') and a new object O' is created whose public is P' and whose local is L'. All the expressions occuring in `[...]` will be evaluated in the new scope. Hence, each new object created by `[...]` knows the lexical scope L but contains an empty protocol since P' is an extension of the $rootP$ (i.e. the public part of the `agora` object).
- When executing a {...} statement in a scope $S = (O, L, P)$, all the expressions occuring in {...} are simply evaluated in S. Hence {...} only groups expressions together without creating opening a new scope.

4.5 Summary

In this section, we saw how objects can be extended using mixin-methods. Mixin-methods are methods whose body is a block of expressions with which the receiving object is extended. Mixin-methods can be functional or destructive. Functional mixin-methods (or views) return a view on the receiving object, while destructive mixin-methods (or mixins) destructively change the receiver. Mixin-methods allow attributes to override existing attributes. The attributes in a mixin-method can decide to call one of the attributes in the original receiver. This is done by the `SUPER` reifier. Important to remember is that parent sends in the local (resp public) part, will stay in the local (resp public) part of the parent. Finally, we postponed extension from the outside to the sections on meta programming.

5 Programming in Agora98

This section describes a few programming techniques in Agora98. Section 5.1 describes a few ways to use mixin-methods. Section 5.2 explains how linearisation can be used to implement multiple inheritance.

5.1 Usage of Mixin-Methods

5.1.1 Parent Sharing

A feature that makes prototype-based programming languages unique is parent sharing. As we saw in the previous section, views return an object that is an extension of the receiving object. By applying two views on the same receiver, two objects with a shared parent exist. Consider for example the following code fragment.

```
o LOCAL VARIABLE: [ x VARIABLE:0;
                    m VIEW: { y VARIABLE:1 };
                    n VIEW: { z VARIABLE:2 }
                  ]
```

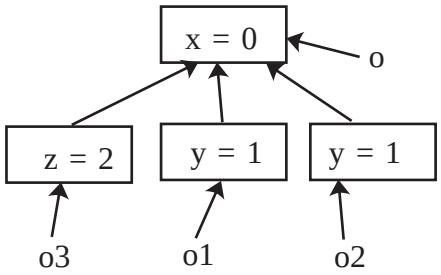

Figure 7: Parent Sharing

```
o1 VARIABLE: o m;
o2 VARIABLE: o m;
o3 VARIABLE: o n
```

Executing it will result in the structure depicted in figure 7.

Parent sharing is one of the characteristic features of prototype-based languages. One of its consequences is the notion of traits objects.

5.1.2 Traits Objects

The notion of traits objects comes from the Self-nomenclature (see [9]). The idea of traits objects is very simple. Suppose one has a two-dimensional point containing an x and y co-ordinate. The point has a number of methods that specify behaviour such as drawing, moving,.... Now suppose we want to create a set of polygons each consisting of points. These points will all be created by cloning some prototypical point. This is a potential source of space inefficiencies as each copy of the prototypical point will have its own pointers towards the code of the methods. The situation is depicted in figure 8a. This problem is solved by using a traits object. The idea is to let the traits object contain all the common behaviour and to create 'instances' that have the trait as parent. Each object will thus contain the behaviour of the traits by means of inheritance. Figure 8b uses a traits object to mimic the situation of figure 8a in a much more efficient way.

In Agora98, the 'instantiation' of a traits object can be accomplished by a view. The following code fragment illustrates the traits object of figure 8b in Agora98.

```
Point LOCAL VARIABLE: [ draw METHOD:...
                        move METHOD:...
                        newx:xx y:yy VIEW:
                         { x LOCAL VARIABLE:xx;
                           y LOCAL VARIABLE:yy }
                      ];
p1 VARIABLE: Point newx:1 y:2;
p2 VARIABLE: Point newx:4 y:8;
```

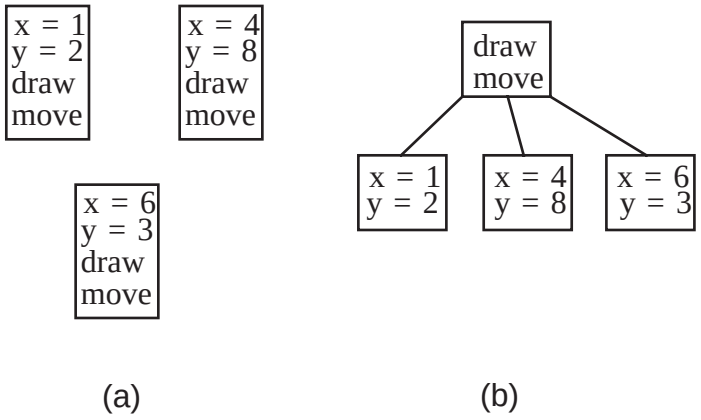

Figure 8: Traits Objects

```
p3 VARIABLE: Point newx:5 y:3;
```

The code illustrates a `Point` prototype that can be 'instantiated' using the `newx:y:` view that functionally extends the prototype in order to contain the instance variables of a particular point. This can be done as many times as we want. All thus-created points will have the original `Point` prototype as parent. Hence both `draw` and `move` methods will be shared between them. The notion of traits objects can also be used to re-introduce class-based features like class instance variables (or static members) in a prototype-based language. It is easy to realise that if we add a variable to the traits object, this variable will be shared between every instance of the traits object. Hence, this variable acts like a class variable in Smalltalk or a static member in C++.

5.1.3 Overriding of Mixin-Methods

As already stated, besides the fact that mixin-methods result in an extension of their receiver, mixin-methods are completely like ordinary methods. Hence, we can override them as well. The power of overriding mixin-methods has not been completely investigated so far. So examples that use this feature are more than welcome! Consider a `person` object that understands the `male` and `female` messages. These messages destructively change the `person` object. However, once a `male` extension is made, we do not allow further extension by the `female` mixin-method, since a person cannot be male and female at the same time (excluding hermaphrodites here). The same restriction goes the other way around. We say that both `male` and `female` mixin-methods are exclusive. This can be accomplished by overriding the `male` mixin-method in the `female` mixin-method with a method that does nothing (and vice versa).

```
{ person VARIABLE:
       [ name PUBLIC VARIABLE;
         male MIXIN:
               { ...;
                 female METHOD: {}
```

```
            };
        female MIXIN:
            { ...;
              male METHOD: {}
            }
      ];
  ...
}
```

Instead of just doing nothing, we might also decide to generate a run-time error. This can be accomplished though the `HALT` reifier. The `HALT` reifier must be sent to a string expression. The value of this expression will be used to generate the run-time error.

```
{ person VARIABLE:
      [ name VARIABLE;
        male MIXIN:
            { ...;
              female METHOD:
                { "female after male error" HALT }
            };
        female MIXIN:
            { ...;
              male METHOD:
                { "male after female error" HALT }
            }
      ];
  ...
}
```

5.2 Linearised Multiple Inheritance

Up until now, nobody has properly solved the question of how to cleanly embed multiple inheritance in a programming language. We therefore decided not to include an explicit multiple inheritance mechanism in Agora98. However, having mixin(-method)s in a language appears to weaken the need for an explicit multiple inheritance mechanism strongly. This section discusses how mixin-methods can be used to resolve most needs for multiple inheritance. First, it is important to realise that whenever we want an object O to have two parents P1 and P2, we actually want O to contain the behaviour and design of P1 and P2. The way this combined behaviour is thought of is in principle unrelated to the fact that the object actually has two parents in a certain programming language. This distinction between the need for multiple inheritance and the realisation of multiple inheritance is at the basis of what is often called linearised multiple inheritance. The idea of linearised multiple inheritance is as follows. Suppose we have a parent P1 that is constructed by a mixin-method M1. Suppose the second parent P2 is constructed by mixin-method M2. Let us then assume we want to create an object O that inherits from P1 and P2 and has an additional behaviour specified in a third mixin-method M3. In a language featuring an explicit multiple inheritance mechanism like C++, we would program M3 such

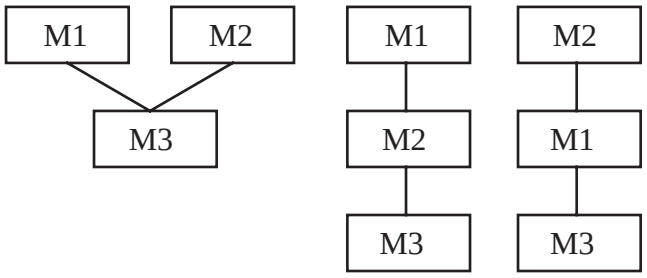

Figure 9: Linearised Multiple Inheritance

that is has two parent parameters. Linearised multiple inheritance simulates this by creating O as a successive application of M1, M2 and M3. Of course, it has to be decided whether we first apply M1 and then M2, or vice versa. The situation is depicted in figure 9.

The choice of applying first M1 and then M2 or the other way around is strongly related to how name collisions in the original hierarchy would have been resolved. Hence, in Agora98, the most trivial way to program multiple inheritance is to put all mixin-methods at top level and to create the objects by successive application of the required parts. This is illustrated in the following code fragment.

```
{ MakePoint VIEW:
    { x LOCAL VARIABLE;
      y LOCAL VARIABLE;
      draw METHOD: ...
    };
  MakeBounded VIEW:
    { bound LOCAL VARIABLE;
      ...
    };
  MakeHistory VIEW:
    { log LOCAL VARIABLE;
      ...
    };

  Point LOCAL VARIABLE: agora MakePoint;
  BoundedPoint LOCAL VARIABLE: Point MakeBounded;
  HistoryPoint LOCAL VARIABLE: Point MakeHistory;
  BoundedHistoryPoint LOCAL VARIABLE: BoundedPoint MakeHistory
}
```

The idea of the example is that the programmer wants to express that her system contains points, points that keep a log (historypoints) of their movements and points whose movements are bound by certain restrictions (boundedpoints). If the programmer suddenly decides to add points whose movements are bound and logged, she encounters a multiple inheritance situation. Her solution might be to simply linearise all the building blocks and apply them successively to

the Point prototype. This way, she implements her (need for) multiple inheritance situation by selecting a certain total order and applying ordinary single inheritance.

6 Reifier Summary & Abbreviations

So far we have introduced all Agora98 reifiers that are normally used for writing non-reflective Agora98 programs. Section 7 discusses the reifiers that languagezize the reflective architecture of the Agora++ framework. This section summarises the reifiers we encountered so far and presents their restrictions in a tabular form.

6.1 Variables

reifiers:

```
<ident> VARIABLE
<ident> VARIABLE:<expression>
<ident> CONST:<expression>
```

adjectives:

```
LOCAL | PUBLIC | LOCAL PUBLIC
```

defaults:

```
PUBLIC
```

abbreviations:

```
LOC   = LOCAL
PUB   = PUBLIC
VAR   = VARIABLE
VAR:  = VARIABLE:
```

6.2 Methods

reifiers:

```
<pattern> METHOD: <expression>
```

adjectives:

```
LOCAL | PUBLIC | LOCAL PUBLIC
```

defaults:

```
PUBLIC METHOD:
```

abbreviations:

```
LOC   = LOCAL
PUB   = PUBLIC
```

6.3 Views

reifiers:

```
<pattern> VIEW: <expression>
```

adjectives:

```
LOCAL | PUBLIC | LOCAL PUBLIC
```

defaults:

```
PUBLIC VIEW:
```

abbreviations:

```
LOC   = LOCAL
PUB   = PUBLIC
```

6.4 Mixins

reifiers:

```
<pattern> MIXIN: <expression>
```

adjectives:

```
LOCAL | PUBLIC | LOCAL PUBLIC
```

defaults:

```
PUBLIC MIXIN:
```

abbreviations:

```
LOC   = LOCAL
PUB   = PUBLIC
```

6.5 Cloning Methods

reifiers:

```
<pattern> CLONING: <expression>
```

adjectives:

```
LOCAL | PUBLIC | LOCAL PUBLIC
```

defaults:

```
PUBLIC CLONING:
```

abbreviations:

```
LOC   = LOCAL
PUB   = PUBLIC
```

6.6 Selections

reifiers:

```
<boolean> IFTRUE: <expression>
<boolean> IFFALSE:<expression>
<boolean> IFTRUE: <expression> IFFALSE:<expression>
<boolean> IFFALSE:<expression> IFTRUE: <expression>
```

6.7 Iterations

reifiers:

```
<ident> FOR:<integer> TO:<integer> DO:<expression>
<ident> FOR:<integer> TO:<integer> BY: <integer> DO:<expression>
<ident> FOR:<integer> DOWNTO:<integer> DO:<expression>
<ident> FOR:<integer> DOWNTO:<integer> BY: <integer> DO:<expression>
<boolean> WHILETRUE: <expression>
<boolean> WHILEFALSE: <expression>
<expression> WHILEFALSE: <boolean>
<expression> WHILETRUE: <boolean>
```

6.8 Object Accessing

reifiers:

```
SELF
<pattern> SUPER
```

6.9 Exception Handling

reifiers:

```
<expression> TRY: <pattern> CATCH:<expression>
<pattern> RAISE
```

6.10 Arrays

reifiers:

```
<integerexpression> ARRAY
<integerexpression> ARRAY: <expression>
```

6.11 Various

reifiers:

```
<string> HALT
```

6.12 Reflection (see the following sections)

reifiers:

```
<expression> QUOTE
<expression> UNQUOTE
<expression> DOWN
<expression> UP
<reifierpattern> REIFIER: <unarypattern> IS:<expression>
```

7 Reflective features of Agora98

This section explains the reflective facilities of Agora98. Before moving into full-fledged reflection, we will first explain the meta-level facilities of Agora98. These facilities allow programs to reason about programs, which is accomplished by making Agora98 program fragments accessible to Agora98 programs under the form of Agora98 objects.

7.1 Meta Level Facilities of Agora98

Just like Scheme, Agora98 contains a number of meta-level facilities, which allow a program to reason about programs. In order to understand these operators, it is most appropriate to compare them with the 'commonly known' meta-level facilities of Scheme.

7.1.1 Meta Level Facilities in Scheme

Scheme provides the following meta level operations.

- `(quote <expression>)` The `quote` special form of Scheme allows the programmer to convert any Scheme expression into a Scheme data structure. For example,

  ```
  (quote (lambda (x) (* x x))
  ```

  results in a data structure that can be manipulated using the standard functions car and cdr. In Agora98 terminology, we say that, using `quote`, the evaluator reifies one of its internal structures (i.e. a parse tree) into a referable Scheme value (i.e. a list).

- `(eval <expression>)` The `eval` function takes any valid Scheme expressions, and evaluates it in the top level environment. `eval` can be seen as the inverse of `quote`.

- `(apply <expression> <expression>)` The `apply` function takes two expressions. It will evaluate them both and apply the first one to the second one. `apply` is invoked by `eval` whenever an application is encountered. Likewise, `eval` is invoked by `apply` in order to evaluate the body of the function to `apply`. This mutual recursive behaviour is one of the fundamental laws of Scheme. This remark will become important later on.

7.1.2 Evaluation contexts

Before we can move on to the meta-level facilities of Agora98, we have to say something about evaluation contexts. In Scheme, each expression is evaluated in a certain evaluation context. This context is also known as the environment the expression is evaluated in. Scheme contexts contain, amongst other things, a pointer to the lexical scope the expression was defined in, a pointer to the actual parameters in case the expression is a body of a procedure, and so on. In other words, the evaluator is parametrised by an evaluation context and this evaluation context is a data structure containing all the information the evaluator will need to evaluate the expression. Completely similar, Agora98 expressions are evaluated in a context. In Agora98, contexts contain a reference to the current self, the current super, the lexical scope (i.e. the current private part), etc... . The context in which an expression is evaluated almost completely defines the semantics of that expression.

7.1.3 Quoting Expressions

The first Agora98 meta-level operation we discuss is the `QUOTE` reifier. As all reifiers, `QUOTE` is sent to some piece of Agora syntax. The result of `QUOTE` is an Agora98 object that represents the parse tree corresponding to that syntax. Hence, `QUOTE` reifies an expression into a referable Agora98 object. The opposite of `QUOTE` is `UNQUOTE`. The semantics of `UNQUOTE` is as follows:

1. Evaluate the receiving expression.
2. The result is an Agora98 object representing a piece of syntax (i.e. a quoted expression). This Agora98 object is converted back to the original syntax fragment (i.e. it is unquoted).
3. The resulting syntax tree is evaluated in the current context.

The following example illustrates the most simple usage of `QUOTE` and `UNQUOTE`.

```
{
  anExpression LOCAL VARIABLE: [ a LOCAL VARIABLE:3 ; a inspect ] QUOTE;
  anExpression: (anExpression UNQUOTE)
}
```

First, the variable `anExpression` is bound to a quoted expression `[ a VARIABLE:3 ; a inspect]` . Notice that this is quite different from

```
anExpression VARIABLE: [ a VARIABLE:3 ; a inspect ]
```

In the first case (with `QUOTE`), the entire expression `[...]` is represented as an Agora98 object, and this object is the value of `anExpression`. In the second case (without `QUOTE`), the expression `[...]` is evaluated as usual: an ex nihilo created object is made with one variable `a` and this object is the value of `anExpression`. In the third line of the example, we find the expression `anExpression UNQUOTE`. Let us now follow the rules outlined above. First, the receiver of `UNQUOTE` is evaluated. The result is the Agora98 object representing the quoted expression (i.e. the value of `anExpression`). The second step is to unquote this object. This results in the original expression `[ a VARIABLE:3 ; a inspect ]`. Finally, this expression is evaluated in the context the `UNQUOTE` reifier appears in. The result is an ex nihilo created object with one variable `a`.

7.1.4 The structure of Agora98 syntax trees

In the previous section, we explained that `QUOTE` returns its receiving piece of syntax as an Agora98 object that represents that syntax. Now what can we do with these syntax trees? In Scheme, the answer is simple: a tree is nothing but a special kind of list. Hence, it can be manipulated using car and cdr. In Agora98, our syntax trees consist of patterns, message expressions, reifier message expressions and aggregates (such as `[...]` and {...}). At the implementation level, there are many operations defined on these trees. For example, a message expression object can be asked for its receiver, its pattern and its (indexed) arguments. If you want to know all about the interface of quoted expressions, quote an expression and send `inspect` to the resulting object. Keep in mind the mapping of Java methods onto Agora98 patterns as discussed in section 3.1.2.

We will explain the syntax trees of Agora98 in much more detail in section 7.5.1.

7.1.5 Extension from the outside - finally

As already explained, an old criticism against Agora94 and Agora-S was that (due to mixin-method-based-inheritance) it is impossible to extend objects 'from the outside'. Thanks to `QUOTE` and `UNQUOTE`, this criticism can be refuted. Consider the following example:

```
{
  myMixin VARIABLE: { r VARIABLE:3 } QUOTE;

  point VARIABLE: [ x VARIABLE: 0;
                    y VARIABLE: 0;
                    extend: aMixin VIEW: aMixin UNQUOTE
                  ];

  circle VARIABLE: (SELF point) extend:MyMixin
}
```

Let us explain what is happening here. In the first line, a variable `myMixin` is declared whose initial value is an Agora98 object representing the expression { `r VARIABLE:3` }. Then a point is declared as an ex nihilo created object. The point contains a mixin-method `extend:aMixin`. Upon invocation of this method, the body of the mixin-method will be evaluated in an extension of the receiver. This body is an unquoted version of the variable `myMixin`. And this is precisely the expression { `r VARIABLE:3` }. It is very important to notice here that `UNQUOTE` evaluates its receiver in the context where the `UNQUOTE` reifier is used. Hence, the expression { `r VARIABLE: 3`} will be evaluated in the context of the mixin-method. This context contains the self, the super, the local part ,... of the receiving object. Therefore, we really have extension from the outside here, that is, it is impossible to notice any difference with the same code where the mixin-method is already inside the point object.

So what happened to encapsulated inheritance on objects? In section 4.1 we explained that unrestricted inheritance in prototype-based languages breaches

encapsulation since anyone can add her extra access methods to an object. Consider the following example.

```
{
  dirtyMixin VARIABLE: { read METHOD:loc } QUOTE;

  object VARIABLE:
      [ loc LOCAL VARIABLE: "Not accessible from the outside";
        extend:aMixin VIEW: aMixin UNQUOTE
      ];

  dirtyView VARIABLE: (SELF object) extend:dirtyMixin;
  dirtyView read inspect
}
```

Here we see that extension from the outside is a very dangerous operation. By extending the object with an extra access functionality, local variables can be exported without any problems. Such examples are the reason mixin-methods were invented in the first place. So, what happened to encapsulated inheritance on objects now that we now how to circumvent inheritance with mixin-methods? The answer is that extension still happens via message passing. If the object under consideration decides not to implement the `extend: aMixin` method, it is still impossible to extend it from the outside. Hence, extension from the outside still is some contract between the object being extended and the extender. This is in sharp contrast to most prototype-based languages where an explicit `extend` operator exists. In such languages, the extender has full power and the object being extended has no way to express that it should not be extended.

What we actually want is the ability express that 'some' users are allowed to extend an object, while others are not. In Agora98 this is possible! The key idea is to start a negotiation between the extender and the receiver, before the extension is made. If the result of the negotiation is positive, the extension is made; if it is negative, the receiving object is left untouched. This might be important when putting prototype-based languages on the web: we don't want everyone extending our objects because of security reasons. For example, in Java applets can only read and write files from the machine they are comming from. The same rules for object extension could be applied: only clients that are on the right machine are allowed to make extensions of objects.

Consider the following example:

```
{
  myMixin LOCAL VARIABLE: { a VARIABLE: 3} QUOTE;

  myObject LOCAL VARIABLE:
       [ ...
         myMachine LOCAL VARIABLE: "143.005.45.22";
         ...
         extend:aMixin LOCAL MIXIN: aMixin UNQUOTE;
         extend:aMixin politician: negotiator METHOD:
          { (negotiator password = "007") &
            (negotiator machine = myMachine)
              IFTRUE:  extend: aMixin
```

```
            IFFALSE: "Wrong negotiator!"
        }
    ];

  deHaene LOCAL VARIABLE: [ password VARIABLE: "007";
                            machine  VARIABLE: "143.556.45.22"
                          ];

  claes   LOCAL VARIABLE: [ password VARIABLE: "007";
                            machine  VARIABLE: "143.005.45.22"
                          ];

  ext1 VARIABLE: myObject extend:myMixin politician: deHaene;
  "ext1 = Wrong Negotiator!" COMMENT;

  ext2 VARIABLE: myObject extend:myMixin politician: claes;
  "ext2 = the extension really happened!" COMMENT
}
```

In this example, a quoted mixin, an object and two politicians are declared. The key idea of this example is to make the `extend:aMixin` mixin-method local to the object and to provide another (ordinary) public method `extend:aMixin politician:negotiator`. The idea is that the receiving object first negotiates with the politician and that it invokes the local mixin-method whenever the negotiation turns out to be positive. Otherwise, extensions are refuted. Notice that this is a nice application of local mixin-methods.

7.2 Absorption and Reification

`QUOTE` and `UNQUOTE` are special cases of what is generally called *reification* and `absorption`. In this section, we explain `QUOTE` and `UNQUOTE` in terms of these more general concepts. These insights will prepare the reader for full reflection operators as explained in the following sections.

7.2.1 Base-level objects and Meta-level objects

The simplest way to think about reflection is to have the ability to "effectively add new lines to the interpreter" [?]. So reflection is about being able to program parts of the evaluator in the language that is being evaluated. In order to understand this section it is very important to keep in mind that Agora98 is entirely written in another object-oriented language (in this case Java). The fact that Agora (the language based on message passing alone) is written in an object-oriented language is part of the definition of Agora. Hence, this means that we will be constantly dealing with two kinds of objects in this section. The first kind of objects are the usual Agora98 objects we have already dealt with since section 3. We will call these objects base level objects or Agora98 objects. The second kind of objects are the objects from the implementation of Agora98, that is, the objects that make up the evaluator. These objects will be called the meta level objects. Of course, when programming an object in Agora98 (e.g. an ex-nihilo created object), this object will somehow be represented by a certain

structure of meta objects, that together determine the Agora98 object. E.g. an Agora98 object consists of several mixin-method frames, each containing methods and patterns and so on. All these components are meta level objects. The evaluator is then nothing but a huge set of objects that work together (via message passing) in order to evaluate an Agora98 program.

The messages the Agora98 programmer sends in an Agora98 program will henceforth be called base level messages. Likewise, the messages the evaluator objects send to each other will be called meta level messages. Again, sending base level messages is implemented by sending a number of meta level messages. Both base level objects and meta level objects implement a certain number of messages. These messages together determine the protocol of the objects. The set of messages that is understood by a base level object will be referred to as the base level protocol of that object. The set of messages a meta level object understands will be called the meta level protocol of that meta level object. This strong distinction in base level and meta level terminology will prove very useful later on.

7.2.2 Absorption and Reification

As already explained, each Agora98 object (i.e. base level object) is represented by some meta level object. The latter can consist of several other meta level objects, but there is one meta level object that determines the identity of the base level object. If one of them changes, the other one will change too. But of course, there are many other meta level objects that are not referred to in Agora98. We say that some meta level objects are reified in Agora98, that is, some meta level objects are made visible in Agora98, while other meta level objects are absorbed (or deified) in the implementation, that is, they are not visible in Agora98. We already saw many examples of reified objects.

- The primitive Agora98 objects are reified versions of the same primitive objects in the implementation (that is, the Agora98 number 3 is implemented as an object in Java that represents 3). In this process, the protocol an Agora98 programmer is allowed to use for the number 3, is the same as the protocol Java is able to use on the number 3 object.

- All the objects created from invoking a Java constructor on a Java class accessed by the `JAVA` reifier are also reified objects. Even the classes themselves are reified objects.

- Using `QUOTE`, we were able to reify meta level objects representing the parse tree of an Agora98 program. Hence `[ x VARIABLE: 3] QUOTE` is a base level object that is a reified version of the meta level object representing the syntax tree of `[ x VARIABLE: 3]`.

We also saw an example of an absorbed object. In section 7.1.2 we explained that every Agora98 expression is evaluated in a context that contains references to the 'current self', the 'current super' and the lexical scope of that expression. In the implementation, such a context is represented as a meta level object. But until now, these contexts remain invisible for the Agora98 programmer. We say that the context objects are absorbed in the implementation.

Being able to effectively add new lines to the interpreter requires the possibility of programming an object in Agora98 and push it into the implementation,

in such a way that the evaluator can send messages to this Agora98 object as if it where a piece of the evaluator. Hence, we must be able to absorbe base level objects into the implementation. In the following imaginary code fragment,

```
{ mySpecialTree LOCAL VARIABLE: [ evalContext:context METHOD: ... ]

  mySpecialTree absorbeAsExpression
}
```

we illustrate the idea: it must be possible to program a piece of syntax in which we specify our own evaluation method. Then, this base level object must be turned into a meta level object of type 'abstract grammar tree' such that the evaluator will talk to this tree as if it were one of its own trees. We already explained that the evaluator juggles around context objects in order to evaluate expressions. This means that when we absorb our tree, the evaluator will send it the message `evalContext:` with a context parameter. The `evalContext:` method must be provided with a formal parameter `context` in which the Agora98 evaluator will pass its internal evaluation context. But then, this means that the `evalContext:` method will be able to send messages to the `context` parameter. Hence, the evaluator must be able to reify its meta level context into a referable base level object, such that base level messages sent to the context (e.g. `context getSelf`) are translated into implementation level messages sent to the meta level representation `c` of this context (e.g. `c.gerSelf()`).

At this point, we can generalise what we said in the previous section about `QUOTE` and `UNQUOTE`. `QUOTE` is a reification operator. It is sent to a piece of syntax tree. This tree is reified as a referable Agora98 object which is the result of `QUOTE`. This object implements the `eval:context` message. The opposite is `UNQUOTE`. `UNQUOTE` takes any Agora98 object that implements the `eval:context` message (e.g. a quoted object) and transforms it to a meta level object that is further treated as a parse tree on which the evaluator can invoke the `eval:context` message. Hence, `UNQUOTE` is an absorption operator. It absorbs Agora98 objects as syntax meta level objects. So, `UNQUOTE` is precisely what we desired in the above example from `absorbeAsExpression`. The mechanisms of absorption and reification together form a so called *linguistic symbiosis* between Agora98 and its implementation language. This means that there are mechanisms that make it possible for base level objects to travel to the meta level and the other way around. Stated more directly, the current implementation makes it possible to send Java messages (using the `.` operator) to Agora98 objects and to send Agora98 messages to Java objects. `QUOTE` and `UNQUOTE` are the first reifiers that allow us to explicitly use this mechanism (or in a mind boggling way: that reify the mechanism).

The conclusion of this section is that, when we want to add new lines to the interpreter, the language should support absorption of base level objects and reification of meta level objects. In the Agora98 evaluator, both mechanisms are implemented. That is, it is possible to write Agora98 base level objects containing methods that will be invoked by meta level messages. The parameters of these meta level messages will be reified into base level objects such that it is possible to send them base level messages. The result of the method is a base level object that should be absorbed again such that the evaluator can handle it.

7.3 Writing your own reifiers - part I

This section gives a first glimpse on how to write your own reifiers. The examples in this section are especially chosen to be very simple. We will show more 'adult' reifiers at the end of section 7. Just like ordinary messages, reifier message come in both receiverful and receiverless form. For instance, `SELF` is a receiverless unary reifier message while `METHOD:` is a receiverful reifier keyword message. Receiverless reifier messages are - like receiverless user messages - looked up in the local part of the object in which they occur. This can easily be noticed in the inspector. The reifier `SELF` occurs in the local part of the root object. When it is accessed, the corresponding `SELF` message (at Java level) is sent to the 'current' private part of the object. Hence, receiverless reifiers are always sent to the current lexical scope. Up until now, `SELF` is the only receiverless reifier message in Agora98. The difference with ordinary messages is that reifier messages pass the evaluation context to their receiver and arguments. This is the same in Scheme:

- When an ordinary function application is evaluated in a context c, Scheme evaluates the function in c and evaluates all the arguments in c. But once the function and the arguments are known, the context c is thrown away and the function is applied to the arguments. The body of the function itself is evaluated in the context of its definition and not in the context c of application.
- When a reifier function (i.e. a special form) is applied, a hidden context is passed to the reifier function body since the arguments will have to be evaluated in the context of application. Consider an if-statement in Scheme

  ```
  (if <cond> <exp1> <exp2>)
  ```

  The evaluation of this reifier function application is to apply the if procedure to its three quoted arguments. The if procedure will evaluate the condition in the context where the if occurs, i.e. in the context of application of if and conditionally evaluates `<exp1>` or `<exp2>` in that context. For example, if we write `(if (= x 0)...)` , the `if` procedure will lookup `x` in the environment where the if reifier function is applied, and not in the environment of definition of the if reifier. Hence, the evaluator for the if-reifier must explicitly pass its context to the body of the if-procedure. So, when Scheme calls a reifier function, it must thereby explicitly pass the context of the reifier function application to the implementation of the reifier.

This is the same in Agora98. Hence, ordinary message expressions evaluated in a context will evaluate their receiver and their arguments in that context. When everything is evaluated, the message is sent and the context is thrown away. Reifier messages on the other hand, quote their receiver and arguments and pass the evaluation context as an explicit parameter to the reifier method. Hence, reifiers get an explicit context argument which is the context of the place were the reifier is applied.

The following program illustrates our first home-made reifier.

```
[
  (SELF) LOCAL REIFIER:context IS:
      { ("java.lang.System" JAVA) out printString:"Self accessed!";
        SELF SUPER
      };
  test PUBLIC METHOD:
      { ("java.lang.System" JAVA) out printString:"tested!" };
  SELF test
]
```

The program overrides `SELF` in the local part. Whenever `SELF` is accessed, a message is printed and the old self (i.e. `SELF SUPER`) which resides in the root local part is returned. Notice that the syntax of the reifier is `REIFIER:IS:`. The first argument is a name for the context that will be used. The second argument is the implementation of the reifier. The receiver of `REIFIER:IS:` is a reifier pattern (in our case a unary reifier pattern) that will be the name of the reifier. This is `SELF` in our case. The following example shows how we can write a home-cooked 'if' that always selects its first expression by 'anding' its condition with false (for the sake of the example).

```
{
  (IF:cond THEN:exp1 ELSE:exp2) LOCAL REIFIER:context IS:
  { test LOCAL VAR: cond evalContext:context;
    test: test & false;
    test
     IFTRUE:  exp1 evalContext:context
     IFFALSE: exp2 evalContext:context
  };

  x LOCAL VARIABLE: false;
  ("java.lang.System" JAVA) out printString:(IF: x THEN: "doesn't work" ELSE: "works" )
}
```

Notice that the evaluation context context is used here to evaluate (i.e. lookup) the `x` variable which resides in the context where the `IF:THEN:ELSE:` reifier is sent and not in the context where the reifier is defined. But if your enter the above program, a run time error says that the `test` variable does not understand the `&` message while we correctly passed `x` whose value is a boolean as actual argument of the `IF:` keyword. The reason is that reifier code is meta level code. It is invoked by the evaluator. Hence, inside the code, you can access the `cond`, `exp1` and `exp2` parameters by the same protocol the evaluator can. For instance, you can send `evalContext:context` to the `cond` argument while `evalContext:context` is a message from the meta level. In the same way, you can send meta level messages to the `context` argument. For example, sending the message `context getSelf` would be perfectly legal, since `getSelf` is a meta level message by which the evaluator retrieves the 'current' self from its context objects. Hence, when the evaluator invokes a reifier message, all its arguments are meta level arguments. The result of a reifier message is again a meta-level object. But this is automatically brought back to the base level by the part of the evaluator the invoked the reifier message. That is why the `SELF` example worked.

The meta representation of the arguments is precisely their quoted representation since quoted syntax tree objects understand `evalContext:context` and that is precisely the way the evaluator talks to its syntax objects. Now we can understand why the if example doesn't work. The `test` variable contains the result of sending `evalContext:context` to `cond`. But since `cond` and `context` are meta level objects and `evalContext:context` is a meta level message, the result will again be a meta level object. In this case, the result is a meta level object that represents the base level object `false`. So sending `&` to `test` will result in a `message not understood error`. Although `false` understands the message `&`, it's meta level representation does not. It only understands `send` since `send` is the way the evaluator talks to meta level objects. We will explain the precise meaning and usage of `send` later on. So in this case, we need an operation to transform the `test` object (i.e. the meta level representation of `false`) into its base level analogue (i.e. `false`). This operation is implemented by the `DOWN` reifier message. The following version of the if example will work correctly (although the if does not behave as expected, but that is of course an error in the way we programmed it!):

```
{
  (IF:cond THEN:exp1 ELSE:exp2) LOCAL REIFIER:context IS:
  { test LOCAL VAR: (cond evalContext:context) DOWN;
    test: test & false;
    test
     IFTRUE:  exp1 evalContext:context
     IFFALSE: exp2 evalContext:context
  };

  x LOCAL VARIABLE: false;
  ("java.lang.System" JAVA) out printString:(IF: x THEN: "doesn't work" ELSE: "works" )
}
```

Hence, the rule for evaluating reifier messages is to bring all the arguments (including the `context` argument) to their meta level representation and to bring the result back to the base level. If inside the body of the reifier you perform a computation that needs the base level version of one of the objects, you need to explicitly down it to the base level.

The following receiverless reifier message will be used several times throughout the remainder of this manual:

```
GRAB LOCAL REIFIER: cont IS: cont
```

`GRAB` is used in the same way as `SELF`. The result of `GRAB` is to return its explicit context argument. However, the context argument cannot simply be returned. When you enter `GRAB` as above, Agora98 will complain that the result of the reifier message is an invalid base level object. As already explained, Agora98 evaluates the body of the reifier with Agora98 representations of its arguments (0 here!) and its explicit context argument. The meta level object that is returned by the reifier is automatically brought back to the base level (by an implicit invocation of down). Hence, if we would simply return `cont` as the result of the reifier, the downed version of `cont` would be returned. This is an implementation level object that cannot be used as an explicit base level

object from within Agora98. Hence, we need an operation to bring the returned `cont` object 'one level higher' such that `cont` will be the result of the reifier and not its down'ed version (which is an implementation level object to which no Agora98 messages can be sent). This operation is the inverse of `DOWN`. We therefore called it `UP`. By returning an up'ped object from a reifier, the object itself will be the result since the result of a reifier is implicitly brought back to the base level by an implicit invocation of down. Hence, the correct version of the `GRAB` reifier becomes:

```
GRAB LOCAL REIFIER: cont IS: (cont UP)
```

7.4 Level Shifting

Let us summarize what we saw in the previous section.

Whenever the evaluator invokes a reifier message, it runs the body of the reifier message with implementation level objects (syntax and context) as explicit parameters. The body of the reifier can send Agora98 messages to these implementation level objects because the evaluator has upped them before the reifier is ran. Inside the reifier, the implementation level objects (i.e. the meta level objects) can be sent ordinary Agora98 messages. The results of these messages will again be meta level objects. For example, sending `evalContext:context` to a syntax object will yield a meta level object that is the representation of the corresponding base level object. After running the reifier, the result of the reifier is brought back to the base level by down'ing it. Two 'dirty' situations can happen here:

- If, inside the reifier, you want to treat a meta level object as a base level object (e.g. if you want to inspect the result of an `evalContext:`) you have to explicitly 'cast' that object to the base level by `DOWN`.
- If you want to return one of the up'ped implementation level objects (such as the context or one of the syntax objects) it will be automatically downed such that it becomes useless to your Agora98 code that invoked the reifier. You therefore have to `UP` it once more.

The process of travelling from the base level to the meta level and the other way around, is known as *level shifting*. The `REIFIER:IS:` reifier automatically does level shifting for you in a very consistent way: all the arguments are brought to the meta level, and the result is brought back to the base level. But very often, you will need explicit level shifting operations like `UP` and `DOWN`.

At this point, we can explain the behaviour of `UP`, `DOWN`, `QUOTE` and `UNQUOTE` more accurately.

- The `UP` reifier message is - like any other reifier message - sent to a syntax object. The implementation of `UP` evaluates its receiver and shifts the resulting Agora98 object one level up.
- The `DOWN` reifier evaluates its receiver. The resulting Agora98 is shifted one level downwards. If this is not possible (which happens when the object is already a base level object), an error is raised.

- The `QUOTE` reifier is sent to a syntactic object. The object is immediately shifted one level higher. Hence `QUOTE` and `UP` differ in the fact that `UP` evaluates its receiving expression while `QUOTE` doesnt.
- `UNQUOTE` shifts its receiving expression one level down and sends (at the level of the interpreter) `evalContext:c` to the resulting object where `c` is the context in which the `UNQUOTE` reifier was sent.
- The `REIFIER:IS:` reifier shifts its arguments (which are implementation level syntax objects) and its context argument (which is an implementation level context object) one level up and automatically shifts the result (if any) one level down.

7.5 Reflective Programming

In the previous section we have seen how the `REIFIER:IS:` reifier allows us to 'make new keywords'. We also saw that very often `UP` and `DOWN` are needed in order to do level shifting by hand. Due to these constructions, we can reify the Agora98 implementation into the Agora98 language, which makes it possible to actually use the internal structure of the evaluator to write our own reifiers and reflective code. This section further elaborates on the internal working of the Agora98 evaluator. Of course, such a direct communication with the interpreter is much more powerful than what we have seen so far, but it is also much more difficult to use since it requires us to known all about the interpreters internal workings. In this section we introduce the evaluator 'gently'. We don't have a reference manual about the evaluator, but if you are writing reflective code, you can inspect any object you encounter by sending `inspect` to it. The information in the inspector together with the HTML-documentation that comes with the Agora98 implementation, serve as a reference manual.

7.5.1 The Basic Agora architecture:Send and Eval

The first step in understanding the evaluator is to know that Agora98 programs are internally represented by a parse tree. This tree consists of a set of objects that are linked together. All these objects are somehow a subtype of the class object `grammar.Expression`. All the subtypes of `grammar.Expression` are valid (stand alone) Agora98 expressions. Each such expression implements two methods that take care of the evaluation of the expression. The first one is `defaultEval`. `defaultEval` evaluates the expression in the context of the root object. In the implementation, `defaultEval` is only invoked at the very beginning of the evaluation cycle, when the programming environment evaluates an expression that came out of the parser. `defaultEval` immediately creates a new context object with pointers to the root self, the root private, ... and invokes `evalContext:context` on itself with the new context. `evalContext:context` evaluates the expression in the given context.

This explanation holds for every possible tree component. These components are all objects from the inheritance hierarchy depicted in figure 10. Of course, there are other operations defined on these expressions. For example, in the case of a `grammar.UserMessage`, there are operations to access the expression denoting the receiver and so on. In order to know the interface of expression objects, take an expression object and inspect it. For example, if you want to

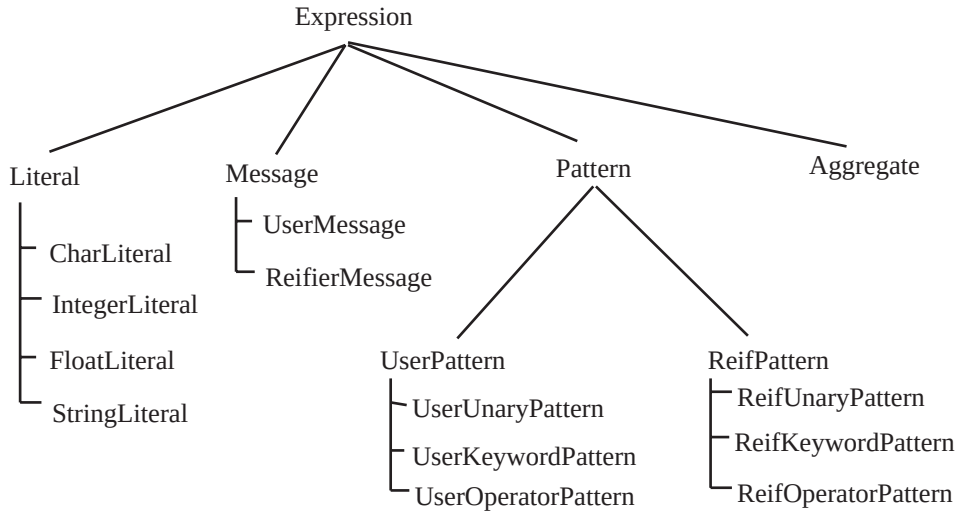

Figure 10: Agora Expression Hierarchy

know the interface of user message expressions, evaluate something like `(3 + 4 QUOTE) inspect`. If you want to know the interface of aggregates, try `([] QUOTE) inspect`. Besides the usual accessor methods, you will also see the implementation of the reifiers, because reifier messages are messages to grammar components.

The most frequently used expression are user messages, objects of type `grammar.UserMessage`. These messages are evaluated in a normal applicative order way. Hence, the internal implementation of the `evalContext:context` for user messages consist of forwarding `evalContext:` to the receiver and al the arguments of the user message. Then the runtime pattern is constructed. Finally the message is sent. This is achieved by `sendAbstractPattern:Client:`. The first argument of this message is the message to be sent. The second argument is a client object, an object in which the actual arguments reside. Hence, each time `evalContext:` is sent to some syntax tree, the result is an 'Agora object', an object whose implementation understands the message `sendAbstractPattern:Client:`. When this message is invoked, the pattern is being looked up inside the object. The corresponding method is then again evaluated by `evalContext:`, which, in most cases will be a user message again. Hence, roughly spoken, the Agora evaluation cycle consists of an alternation between `evalContext:` and `sendAbstractPattern:Client:`, just like the Scheme evaluation cycle basically consists of an alternation between `eval` and `apply`.

The objects understanding `sendAbstractPattern:Client:` are the meta objects of Agora98. These are the normal objects seen though the eyes of the evaluator. You can have a look at them, by inspecting the meta level interface of an ordinary Agora98 object. For example, try `(3 UP) inspect` or `([] UP) inspect`.

7.5.2 Context and Client Objects

The basic idea of evaluating an Agora98 program is that the evaluator sends `evalContext:` to the program. Since the only control structure in Agora98 is message passing, this program is usually a message expression. For ordinary (i.e. non reifeir) message expressions, the evaluation strategy is to recursively send `evalContext:` to the receiver and to all the arguments. The resulting objects are all 'Agora objets', objects of type `objects.AgoraObject`. Then the message is sent by invoking `sendAbstractPattern:Client:` on the receiving Agora object. The first argument is a pattern, the second argument is a so called client object containing the actual parameters. If the receiver finds the pattern, the method associated to this pattern is again evaluated using `evalContext:`.

Hence, basically, the evaluator is a recursive interplay between `evalContext:` and `sendAbstractPattern:Client:`. This means that at each time during the evaluation cycle, either a context object exists or a client object exists. Contexts exist when the evaluator is running, client objects exists when a message is sent.

This knowledge was used to implement a contract between client and context objects. When a message arrives in an object, a client is used to transfer the actual parameters to the object. This client is immediatly asked for a new empty context which is then filled with all the internal information about the receiver. Then method lookup starts and when the method is found, it is evaluated in that context. When the method is a message expression, the receiver and all arguments are evaluated in that context. Finally, the context is asked for a new client in which the actuals will be stored. This client is used for sending the messages. Hence, the Agora98 evaluator never creates a context or a client itself. Instead, clients and contexts create each other.

This contract is useful when the Agora98 evaluator has to be extended. The fact that contexts and clients create each other allows us to copy information through the evaluator in a transparant way. For example, in order to enable exception handling, the evaluator constantly juggles around the 'last encountered exception handler'. This is passed down the evaluator in the contexts and the clients. We have done other experiments in which the contract is used to transport 'system wide' information through the evaluator. All that has to be done is to create a subclass of `runtime.Context` and a subclass of `runtime.Client` and make sure that these subclasses are able to create objects of each other. This is accomplished by implementing the methods `newClient` on contexts and `newContext` on clients appropriately.

If you want to know more about the interface of contexts, use the `GRAB` reifier (implemented above) and send `inspect` to it. The consult the HTML documentation that came with Agora98.

Only one more thing needs to be said about clients and contexts. Agora98 features ordinary messages and reifier messages. The method associated to an ordinary message is evaluated in the context of the receiver, i.e. the context created by the client after message sending. Reifier messages on the other hand need to be evaluated in the context in which they were called, and not in the context of their receiver. Consider for example the reifier message `IFTRUE:IFFALSE:`. When this reifier arrives at the receiving boolean expression, one of its arguments will be evaluated. But this evaluation process must be done in the context where the reifier was sent (dynamically scoped) because if for example, one of the branches uses a variable (say `x`), that variable must be looked

up where the reifier is used, and not where the reifier is implemented. Hence, reifier messages are parametrised by an extra (hidden) context argument. This means that when a reifier message is evaluated, its client will contain the actual arguments together with the context of invocation. In order to accomplish this, a different sort of clients is used: `runtime.ReifierClient`. This kind of client objects behaves in the same way ordinary clients do. The difference is that they cary around an extra context argument, the context of invocation of the message that uses the client. Furthermore, when this kind of client is asked for a new context, no real new context is created, but the context of invocation is simply returned.

If you want more information about clients, evaluate `("runtime.Client" JAVA) inspect` or `("runtime.ReifierClient" JAVA) inspect` and consult the HTML pages that came with Agora98.

7.6 To be continued...

8 References

References

[1] G. Blaschek. *Object-Oriented Programming with Prototypes*. Springer-Verlag, 1994.

[2] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Computer Science,University of Utah, 1992.

[3] G. Bracha and W. Cook. Mixin-based Inheritance. In No Idea, editor, *Proceedings of Joint OOPSLA/ECOOP'90 Conference*, pages 303–311. ACM Press, 1990.

[4] W. Codenie, De Hondt K., Steyaert P., and D'Hondt T. Agora: Message Passing as a Foundation for Exploring OO Languages. *ACM Sigplan Notices*, 29(12):48, December 1994.

[5] C. Dony, J. Malenfant, and P. Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *Proceedings of the OOPSLA'92 Conference*, ACM Sigplan Notices, page 201. ACM Press, 1992.

[6] J. Lamping. Typing the Specialisation Interface. In No Idea, editor, *Proceedings of OOPSLA'93 Conference*, pages 201–214. ACM Press, 1993.

[7] P. Steyaert and W. De Meuter. A Marriage of Class and Object-based Inheritance Without Unwanted Children. In *Proceedings of the ECOOP'95 9th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, page 127. Springer Verlag, 1995.

[8] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-oriented Programming*. PhD thesis, Department of Computer Science, University of Jyvaskyla, 1993.

[9] D. Ungar and R. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA '87 Conference*, ACM Sigplan Notices, page 227. ACM Press, 1987.