

The Cyan Language

José de Oliveira Guimarães
Campus de Sorocaba da UFSCar
Sorocaba, SP
Brasil
jose@ufscar.br
josedoliveiraguimaraes@gmail.com

October 20, 2014

Contents

1	An Overview of Cyan	6
2	Packages and File organization	41
3	Basic Elements	47
3.1	Identifiers	47
3.2	Comments	47
3.3	Keywords	48
3.4	Assignments	48
3.5	Basic Types	50
3.6	Operator and Selector Precedence	56
3.7	Loops, Ifs, and other Statements	57
3.8	Arrays	61
4	Objects	63
4.1	Constants	68
4.2	self	69
4.3	clone Methods	69
4.4	Shared Variables	69
4.5	new, init, and initOnce Methods	70
4.6	Order of Initialization	74
4.7	Keyword Methods and Selectors	75
4.8	On Names and Scope	77
4.9	Operator []	78
4.10	Method Overloading	78
4.11	Inheritance	80
4.12	Multi-Methods	83
4.13	Nil and Any, the Super-prototype of Everybody	84
4.14	Abstract Prototypes	91
4.15	Interfaces	92
4.16	Types and Subtypes	95
4.17	Union Types	95
4.17.1	The Union Prototypes	98
4.17.2	Operators that Use Unions	102
4.18	Mixin Inheritance	103
4.19	Runtime Metaobjects or Dynamic Mixins	108

5	Metaobjects	111
5.1	Pre-defined Metaobjects	112
5.2	Syntax and Semantics	116
5.3	Metaobject Examples	118
5.4	Codegs	119
5.5	Macros	123
6	Dynamic Typing	128
7	Generic Prototypes	135
7.1	Generic Prototypes with Real Parameters	140
7.2	Generic Prototype with a Varying Number of Parameters	142
7.3	Multiple Parameter Lists	142
7.4	Source File Names	143
7.5	Combining Generic Prototypes	144
7.6	Future Enhancements	145
8	Important Library Objects	146
8.1	System	146
8.2	Input and Output	146
8.3	Tuples	147
8.4	Dynamic Tuples	151
8.5	Intervals	152
9	Grammar Methods	155
9.1	Matching Message Sends with Methods	157
9.2	The Type of the Parameter	158
9.3	Unions and Optional Selectors	160
9.4	Refining the Definition of Grammar Methods	164
9.5	Context-Free Languages	166
9.6	Default Parameter Value	168
9.7	Domain Specific Languages	170
9.8	Groovy Builders	174
9.9	A Problem with Grammar Methods	177
9.10	Limitations of Grammar Methods	178
10	Functions	180
10.1	Problems with Anonymous Functions	182
10.2	Some More Definitions	185
10.3	Classifications of Functions: u-Functions and r-Functions	185
10.4	Type Checking Functions	195
10.5	Some Function Examples	196
10.6	Why Functions are Statically-Typed in Cyan	198
10.7	Functions with Multiple Selectors	198
10.8	The Type of Methods and Methods as Objects	199
10.9	Message Sends	202
10.10	Methods of Functions for Decision and Repetition	203
10.11	Context Functions	203
10.12	Implementing r-Functions as u-Functions	211

11 Context Objects	213
11.1 Using Instance Variables as Parameters to Context Objects	214
11.2 Passing Parameters by Copy	215
11.3 What the Compiler Does With Context Objects	216
11.4 Type Checking Context Objects	219
11.5 Adding Context Objects to Prototypes	220
11.6 Passing Parameters by Reference	221
11.7 Should Context Objects be User-Defined?	223
11.8 More Examples	223
12 The Exception Handling System	226
12.1 Using Regular Objects to Treat Exceptions	229
12.2 Selecting an eval Method for Exception Treatment	230
12.3 Other Methods and Selectors for Exception Treatment	233
12.4 Why Cyan Does Not Support Checked Exceptions?	236
12.5 Synergy between the EHS and Generic Prototypes	238
12.6 More Examples of Exception Handling	241
13 User-Defined Literal Objects	246
13.1 Literal Numbers	246
13.2 Literal Objects between Delimiters	248
14 The Cyan Language Grammar	255
15 Opportunities for Collaboration	260
16 The Cyan Compiler	262
17 Future Enhancements	266
Index	268

Foreword

This is the manual of Cyan, a prototype-based statically-typed object-oriented language. The language introduces several novelties that makes it easy to implement domain specific languages, reuse the code of methods and nested prototypes (the equivalent of nested classes), blend dynamic and static code, reuse exception treatment, and do several other common tasks. However, the reader should be aware that:

1. the design of Cyan has not finished. There are a lot of things to be designed, the metalevel being one of them (although we cite metaobjects a lot in the text, the compile-time metaobject protocol has not been defined). It is possible that the definition of some language features will be modified, although it is improbable that they will be great changes;
2. many language features need a more detailed description. Probably there are ambiguities in the description of some constructs. That will be corrected in due time;
3. Cyan is a big language. But this is fine since Cyan is an academic language. Its main goal is to publish articles. However, there are many small details that were added thinking in the programmer, such as nested if-else statements, while statement, literal objects, etc;
4. the compiler for a subset of the language is being built. It creates the AST (Abstract Syntax Tree) generates code for a small subset of the language. Cyan code will be able to use Java classes. Since the Cyan compiler produces Java code, this will not be difficult. Java code will be able to use Cyan prototypes (which is a little bit tricker). However, nowhere in this text we talk about interrelations between the two languages;
5. I have thought in how to implement some parts of the language in an efficient way (whenever possible). There was no time to put the ideas in this report. We believe that many flexible constructs, such as considering methods as objects, can be efficiently implemented (they would not be considered objects unless necessary);
6. If you have any suggestions on anything, please email me.

The main novelties of Cyan are, in order of importance:

- (a) grammar methods, Chapter 9;
- (b) an object-oriented exception handling system, Chapter 12;
- (c) context objects, Chapter 11;
- (d) statically-typed anonymous functions, Chapter 10;
- (e) many ways of mixing dynamic and static typing, Chapter 6;
- (f) codegs, Section 5.4;

- (g) context functions, Section 10.11;
- (h) literal objects delimited by user-defined symbols, Chapter 13. This feature is being defined;
- (i) generic objects with variable number of parameters, Chapter 7.
- (j) a restricted form of multi-methods (search for methods in the textually-declared order), Section 4.12;
- (k) compilation considers previous version of the source code (source code in XML), Chapter 2;

The address of the Cyan home page is <http://www.cyan-lang.org>.

Other features will probably be added to Cyan and its libraries. They are:

- (a) optional use of “;” at the end of a statement;
- (b) metaobjects in the project program (a file that describes all source codes and libraries used in a program). So a metaobject can be applied to all prototypes of a program or to all prototypes of a package;
- (c) enumerated constants;
- (d) concurrent constructs;
- (e) a library of patterns for parallel programming implemented as grammar methods
- (f) a library of Domain Specific Languages for Graphical User Interface made using grammar methods and based on Swing of Java (an initial version has been done already);
- (g) a library for XML and HTML handling using DSL’s made using grammar methods;
- (h) grammar methods with user-defined symbols. That would allow small parsers of arbitrary languages to be implemented as grammar methods;
- (i) a library of patterns implemented using codegs, regular metaobjects, and literal objects;
- (j) doc metaobject to document objects, methods, variables, and so on;
- (k) literal regular expressions (implemented as literal objects);
- (l) generic prototypes defined as metaobjects. Then “P<T1, T2, ... Tn>” would be a call to a method of metaobject P which would produce a real prototype that replaces “P<T1, T2, ... Tn>” in the source code. Tuples will be implemented in this way by the Cyan compiler;
- (m) generic methods.

Chapter 1

An Overview of Cyan

Cyan is a statically-typed prototyped-based object-oriented language. As such, there is no class declaration. Objects play the rôle of classes and the cloning and `new` operations are used to create new objects. Although there is no class, objects do have types which are used to check the correctness of message sending. Cyan supports single inheritance, mixin objects (similar to mixin classes with mixin inheritance), interfaces, a completely object-oriented exception system, statically-typed anonymous functions, optional dynamic typing, user-defined literal objects (an innovative way of creating objects), context objects (which are a generalization of anonymous functions and one of the biggest innovation of the language), and grammar methods and message sends (which makes it easy to define Domain Specific Languages). The language will have a compile-time metaobject protocol in a not-so-near future.

Although the language is based in prototypes, it is closest in many aspects to class-based languages like C++ [Str97] and Java than some prototype-based languages such as Self [US87] or Omega [Bla94]. For example, there is no workspace which survives program execution, objects have a `new` method that creates a new object similar to another one (but without cloning it), and a Cyan program is given textually. In Omega, for example, a method is added to a class through the IDE. Cyan is close to Java and C++ in another undesirable way: its size is closer to these languages than to other prototype-based languages (which are usually small). However, several concepts were unified in Cyan therefore reducing the amount of constructs in relation to the amount of features. For example, methods are objects, the exception system is completely object-oriented (it does not need many ad-hoc keywords found in other languages), and anonymous functions are just a kind of context objects.

In this Chapter we give an overview of the language highlighting some of its features. An example of a program in Cyan is shown in Figure 1.1. The corresponding Java program is shown in Figure 1.2. It is assumed that there are classes `In` and `Out` in Java that do input and output (they are in package `InOut`). The Cyan program declares objects `Person` and `Program`. Object `Person` declares a variable `name` and methods `getName` and `setName`. Keywords `var` and `fun` are used before an instance variable and a method declaration. `var` is optional. Method `getName` takes no argument and returns a `String`. The return type appears before the method body (the “{” and after “->”. Inside a method, `self` refers to the object that received the message that caused the execution of the method (same as `self` of Smalltalk and `this` of Java). A package is a collection of objects and interfaces — it is the same concept of Java packages and modules of other languages. All the public identifiers of a package become available after a “`import`” declaration.

Object `Program` declares a `run` method. Assume that this will be the first method of the program to be executed. The first statement of `run` is

```
var p = Person clone;
```

“`Person clone`” is the sending of message `clone` to object `Person`. “`clone`” is called the “*selector*” of the message. All objects have a `clone` method. This statement declares a variable called `p` and assigns

```

package program

private object Person
  private var String name = ""
  public fun getName -> String {
    ^ self.name
  }
  public fun setName: String name {
    self.name = name
  }
end

public object Program
  public fun run {
    var p = Person clone;
    var String name;
    name = In readLine;
    p setName: name;
    Out println: (p getName);
  }
end

```

Figure 1.1: A Cyan program

to it a copy of object `Person`. The code

```
var variableName = expr
```

declares a variable with the same compile-time type as `expr` and assigns the result of `expr` to the variable. The type of `expr` should have been determined by the compiler using information of previous lines of code. The compiler will issue a warning if `expr` is not a literal of a basic type, an one-dimensional array of these literals, or a message send whose method is not attached to metaobject `@typedClearly`. The next line,

```
var String name;
```

declares `name` as a variable of type `String`. This is also considered a statement. In

```
name = In readLine;
```

there is the sending of message `readLine` to object `In`, which is an object used for input. Statement

```
p setName: name;
```

is the sending of message “`setName: name`” to the object referenced to by `p`. The message selector is “`setName:`” and “`name`” is the argument. Finally

```
Out println: (p getName);
```

is the sending of message “`println: (p getName)`” to object `Out`. The message selector is “`println:`” and the argument is the object returned by “`p getName`”.

The parameters¹ that follow a selector in a method declaration may be surrounded by (and). So method `setName:` could have been declared as

```
public fun setName: (String name) { self.name = name }
```

This is allowed to increase the legibility of the code.

¹In this manual, we will use parameter and argument as synonymous.


```
package program;
import inOut;

private class Person {
    private String name;
    public String getName() {
        return this.name;
    }
    public void setName( String name ) {
        this.name = name;
    }
}

public class Program {
    public void run() {
        Person p;
        String name;
        p = new Person();
        name = In.readLine();
        p.setName(name);
        Out.println( p.getName() );
    }
}
```

Figure 1.2: A Java program

Definition and Declaration of Variables

Statement

```
var p = Person clone;
```

could have been defined as

```
var Person p;
```

```
p = Person clone;
```

Variable `p` is declared in the first line and its type is the object `Person`. When an object is used where a type is expected, as in a variable or parameter declaration, it means “the type of the object”. By the type rules of Cyan, explained latter, `p` can receive in assignments objects whose types are `Person` or sub-objects of `Person` (objects that inherit from `Store`, a concept equivalent to inheritance of classes).

Inheritance

The type system of Cyan is close to that of Java although the first language does not support classes. There are interfaces, single inheritance, and implementation of interfaces. The inheritance of an object `Person` from `Worker` is made with the following syntax:

```
object Worker extends Person
  private String company
  // other instance variables and methods
end
```

If a method is redefined in a sub-object (be it public or protected), keyword “`override`” should appear just after “`public`” or “`protected`”. Methods of the sub-object may call methods of the super-object using keyword `super` as the message receiver:

```
super name: "anonymous"
```

Cyan has runtime objects, created with `new` and `clone` methods, and objects such as `Person`, `Program`, and `Worker`, which are created before the program execution. To differentiate them, most of the time the last objects will be called *prototypes*. However, when no confusion can arise, we may call them *objects* too.

It is important to bear in mind the dual rôle of a prototype in Cyan: it is a regular object when it appears in an expression and it is a type when it appears as the type of a variable, parameter, or return value of methods.

Interfaces

Interfaces are similar to those of Java. One can write

```
interface Savable
  fun save
end
```

```
object Person
  public String name
  public Int age
end
```

```

object Worker extends Person implements Savable
  private String company
  fun save {
    // save to a file
  }
  ... // elided
end

```

Here prototype `Worker` should implement method `save`. Otherwise the compiler would sign an error. Unlike Java, interfaces in Cyan are objects too. They can be passed as parameters, assigned to objects, and receive messages.

Values

The term “variable” in Cyan is used for local variable, instance variable (attribute of a prototype), and parameter. A variable in Cyan is a reference to an object. The declaration of the variable does not guarantee that an object was created. To initialize the variable one has to use a literal object such as `1`, `3.1415`, `"Hello"` or to create an object with `clone` or `new`.

Object `String` is a pre-defined object for storing sequences of characters. A literal string can be given enclosed by `"` as usual: `"Hi, this is a string"`, `"um"`, `"ended by newline\n"`.

Any

All prototypes in Cyan but `Nil` inherit from prototype `Any` which has some basic methods such as `eq`: (reference comparison), `==` (is the content equal?), `asString`, and methods for computational reflection (add methods, get object information, and so on). Method

```
fun eq: Any other -> Boolean
```

tests whether `self` and `other` reference the same object. Method `==` is equal to `eq`: by default but it should be redefined by the user. Method `eq`: cannot be redefined in sub-prototypes. Method `neq`: returns the opposite truth value of `eq`:

Basic Types

Cyan has the following basic types: `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Char`, and `Boolean` (no `Void` prototype). Since Cyan will be targetted to the Java Virtual Machine, the language has exactly the same basic types as Java. Unlike Java, all basic types in Cyan inherit from prototype `Any`. Therefore there are not two separate hierarchies for basic types, that obey value semantics,² and all other types, which obey reference semantics.

However, methods `eq`: and `==` of all basic types have the same semantics: they return true if the contents of the objects are equal:

```

var Int I = 1;
var Int J = 1;
if I == J && (I eq: J) {
  Out println: "This will be printed"
}

```

²The declaration of a variable allocates memory for the “object”. Variables really contains the object, it is stack allocated. In reference semantics, variables are pointers. Objects are dynamically allocated.

Since the basic prototypes cannot be inherited and methods `eq:` and `==` cannot be changed or intercepted, not even by reflection, the compiler is free to implement basic types as if they obey value semantics. That is, a basic type `Int` is translated to `int` of Java.³ There are cases in which this should not be done:

- (a) when a basic type variable is passed as parameter to a method that accepts type `Any`:

```
object IntHashTable
  fun key: String aKey value: Any aValue { ... }
  ...
end
...
IntHashTable key: "one" value: 1;
```

In this case the compiler will create a dynamic object of prototype `Int` for the `1` literal;

- (b) when a basic type variable receives a message that correspond to a method of `Any` such as `prototypeName`:

```
// prints "Int"
Out println: (1 prototypeName);
```

But even in this case the compiler will be able to optimize the code since it knows which method should be called;

In practice, the compiler will implement basic types as the basic types of Java almost all of the time. The overhead should be minimal.

Nil and Union types

There is a special type in the language, the union type. The type `Union<A, B>` is considered, in assignments and parameter passing, as a supertype of both `A` and `B`.

```
var Union<Int, String> x;
x = 0; // ok
x = "Cyan"; // ok
```

The compiler automatically casts objects of `Int` and `String` to `x`. To retrieve the object stored in `x` it is necessary to use method `unionCase:do`:

```
x
  unionCase: Int do: {
    Out println: ("twice is " + 2*x)
  }
  unionCase: String do: {
    Out println: ("first char is" + x[0])
  };
```

Inside the block passed as parameter to `do`: the variable has the type given as parameter to `unionCase:..` Then we can multiply `x` by 2 in the first block and index the second `x`, which is of type `String` which supports indexing.

³The compiler being built translates `Cyan` to Java.

There is a special object called `Nil` which is not subtype or supertype of anything. It somehow plays the rôle of `nil` of Smalltalk, `NULL` of C++, and `null` of Java. As in Smalltalk, `Nil` knows how to answer some messages — it is an object. However, `Nil` can only be assigned to a variable of type `Nil`.

`Nil` cannot be assigned to a variable whose type is a prototype (that is, a prototype that is not `Nil` itself).

```
var String s;
var Person p;
s = Nil; // compile-time error
p = Nil; // compile-time error
```

Variables that can hold a regular object and `Nil` should be declared using unions:

```
var String s;
var Person p;
s = Nil; // compile-time error
p = Nil; // compile-time error
```

To allow `Nil` values in variables it is necessary to declare an union of `Nil` with a prototype.

```
var Nil|String s;
s = Nil; // ok
s = In readLine;
s
  unionCase: Nil do: {
    // in case s is Nil
  }
  unionCase: String do: {
    // s is a String here
  };
```

Then the runtime error “message send to `Nil`” does not happens in Cyan. That is, it can only happen in message sends to `Dyn`, the dynamic type.

A method that does not return anything can be declared as returning `Nil`. It is equivalent to declare `Nil` as the return value and do not declare a return value.

Constructors and Inheritance

Constructors have the name `init` and may have any number of parameters. The return value should be `Nil` or none. For each method named `init` the compiler (in fact, a metaobject) adds to the prototype a method named `new` with the same parameter types. Each `new` method creates an object and calls the corresponding `init` method. If the prototype does not define any `init` or `new` methods, the compiler supplies an empty `init` method that does not take parameters. Consequently, a `new` method is created too. A sub-prototype should call one of the `init` methods of the super-prototype (if one was defined by the user) using keyword `super`:

```
object Person
  fun init: String name { self.name = name }
  private String name
  ...
end
```

```

object Worker extends Person
  fun init: String name, String job {
    // this line is demanded
    super init: name;
    self.job = job;
  }
  private String job
  ...
end

```

All new methods return an object of the prototype. Therefore, `Person` has a method

```
Person new: String name
```

and

`Worker` has a method

```
Worker new: String name, String job
```

To make it easy to create objects, there is an alternative way of calling methods `new` and `new:`.

`P(p1, p2, ..., pn)` is a short form for

```
(P new: p1, p2, ..., pn)
```

Therefore we can write either

```
var prof = Worker("John", "Professor")
```

or

```
var prof = Worker new: "John", "Professor"
```

Of course, if a prototype `P` has a `new` method that does not take parameters we can write just `"P()"` to create an object.

Public and Protected Instance Variables

An instance variable can be declared private, protected, or public. In the last two cases, the compiler will create public or protected get and set methods for a hidden variable that can only be accessed, in the source code, by these methods. For a user-declared public instance variable `instVar` of type `T`, the compiler creates two methods and one hidden instance variable. For example, suppose `University` is declared as

```

object University
  public String name = ""
end

```

Then the compiler considers that this prototype were declared as

```

object University
  public fun name -> String { return _name }
  public fun name: (: _newName String ) { _name = _newName }
  private var String _name
end

```

The source code is not modified. The compiler only changes the abstract syntax tree used internally.

Note that methods `name` and `name:` are considered different. The hidden instance variable `_name` cannot be directly accessed in the source file. However, it can be accessed through the reflection library (yet to be made). It is as if the compiler replaced the declaration

```
public String name
```

by the above code. The same applies to protected variables. The instance variable should be used through methods:

```
University name: "UFSCar";
Out println: (University name);
  // compilation error in the lines below
University.name = "Universidade Federal de São Carlos";
Out println: University.name;
```

Future versions of Cyan may allow the use of dot to access public and protected instance variables. It is only necessary a syntax for grouping the get and set methods associated to a public or protected variable. As it is, this syntax is unnecessary. To replace a public instance variable by methods it is only necessary to delete the variable declaration and replace it by methods.

As an example of that, consider a prototype Point:

```
object Point
  public Float dist
  public Float angle
end
```

...

```
aPoint = Point new;
aPoint dist: 100;
aPoint angle: 30;
r = aPoint dist;
angle = aPoint angle;
...
```

Later we may be like to use cartesian coordinates. No problem:

```
object Point
  public fun dist -> Float { return Math sqrt: (x*x + y*y) }
  public fun dist: Float newDist {
    // calculate x and y with this new distance
    ...
  }
  public fun angle -> Float { ... }
  public fun angle: Float newAngle { ... }
  private Float x
  private Float y
end
```

...

```
  // no changes here
aPoint = Point new;
aPoint dist: 100;
aPoint angle: 30;
r = aPoint dist;
angle = aPoint angle;
```

...

Keyword Messages and Methods

Since Cyan is a descendent of Smalltalk, it supports keywords messages, a message with multiple selectors (or keywords) as in

```
var p = Point dist: 100.0 angle: 20.0;
```

Method calls become documented without further effort. Prototype Point should have been declared as

```
object Point
  fun dist: (Float newDist) angle: (Float newAngle) -> Point {
    var p = self clone;
    p dist: newDist;
    p angle: newAngle;
    return p
  }
  public Float dist
  public Float angle
end
```

Unlike Smalltalk, after a single keyword there may be multiple parameters:

```
object Quadrilateral
  fun p1: (Int x1, Int y1)
    p2: (Int x2, Int y2)
    p3: Int x3, Int y3
    p4: Int x4, Int y4 {
    self.x = x1;
    ...
    self.y4 = y4
  }
  ...
  private Int x1, y1, x2, y2, x3, y3, x4, y4
end
...
var r = Quadrilateral p1: 0, 0 p2: 100, 10
    p3: 20, 50 p4: 120, 70;
```

This example declares the parameters after the selectors in all possible ways.

Abstract Prototypes

An abstract prototype should be declared with keyword `abstract` and it may have zero or more public abstract methods:

```
public abstract object Shape
  public abstract fun draw
end
```


An abstract prototype does not have any `new` methods even if it declares `init` methods. Abstract methods can only be declared in abstract objects. A sub-prototype of an abstract object may be declared abstract or not. However, if it does not define the inherited abstract methods, it must be declared as abstract too.

To call an object “abstract” seem to be a contradiction in terms since “objects” in prototype-based languages are concrete entities. However, this is no more strange than to have “abstract” classes in class-based languages: classes are already an abstraction. To say “abstract class” is to refer to an abstraction of an abstraction.

Final Prototypes and Methods

A prototype declared as `final` meaning that it cannot be inherited.

```
public final object Int
  ...
end
...
public object MyInt extends Int
  ...
end
```

There would be a compile-time error in the inheritance of the final prototype `Int` by `MyInt`.

A final method cannot be redefined. This allows the compiler to optimize code generation for these methods.

```
public object Person
  public final fun name -> String { ^ _name }
  public final fun name: String newName { _name = newName }
  ...
end
...
var Person p;
...
p name: "Peter"; // static call
```

Public instance variables may be declared `final`. That means the get and set methods of the variable are final.

Decision and Loop Methods and Statements

Since Cyan is a descendent of Smalltalk, statements `if` and `while` are not necessary. They can be implemented as message sends:

```
( n%2 == 0 ) ifTrue: { s = "even" } ifFalse: { s = "odd" };
var i = 0;
{^ i < 5 } whileTrue: {
  Out println: i;
  ++i
}
```

However, `if` and `while` statements were added to the language to make programming easier and cascaded `if`'s efficient. The code above can be written as

```

if n%2 == 0 {
    s = "even"
}
else { // the else part is optional
    s = "odd"
};
var i = 0;
while i < 5 {
    Out println: i;
    ++i
}

```

cascaded if's are possible:

```

if age < 3 {
    s = "baby"
}
else if age <= 12 {
    s = "child"
}
else if age <= 19 {
    s = "teenager"
}
else {
    s = "adult"
};

```

Without the if statement, the above code would be much greater and would demand a lot of message sends and parameter passing which would require a lot of optimizations from the compiler.

There is also a short form of if that returns an expression:

```

oddOrEven = (n%2 == 0) t: "even" f: "odd" ;
// or
oddOrEven = (n%2 != 0) f: "even" t: "odd";

```

Unnecessary to say that this is a message send and that the arguments of both selectors should be of the same type.

Every prototype inherits a grammar method from Any that implements the switch “statement”. There is a metaobject attached to this method that checks whether the expressions after case: have the same type as the receiver. The statements after do: between { and } are objects of type Function<Nil>.

```

var n = In readInt;
if n >= 0 && n <= 6 {
    n
    case: 0 do: {
        Out println: "zero"
    }
    case: 1 do: {
        Out println: "one"
    }
    case: 2, 3, 5 do: {
        Out println: "prime"
    }
}

```

```

    }
    else: {
      Out println: "four or six"
    }
};

```

Cyan Symbols

There is a sub-prototype of `String` called `CySymbol` (for Cyan Symbol) which represents strings with an special `eq:` operator: it returns true if the argument and `self` have the same contents. This is the same concept as Symbols of Smalltalk. There are two types of literal symbols. The first one starts by a `#` followed, without spaces, by letters, digits, underscores, and `“:”`, starting with a letter or digit. The second type is a string preceded by a `#`, as `#"a symbol"`. These are valid symbols in Cyan:

```

#name #name:
#"this is a valid symbol; ok? ! &)"
#at:put: #1 #711
#"1 + 2" #"Hello world"

```

A symbol can be assigned to a string variable since `CySymbol` inherits from `String`.

```

var String s;
s = #at:put:;
  // prints at:put:
Out println: s;
s = #"Hello world"
  // prints Hello world
Out println: s;

```

We call the “name of a method” the concatenation of all of its selectors. For example, methods

```

fun key: (String aKey) value: (Int aValue) -> String
fun name: (String aName) age: (Int aAge) salary: (aSalary Float) -> Worker

```

have names `key:value:` and `name:age:salary:`.

Overloading

There may be methods with the same name but with different number of parameters and parameter types (method overloading). For example, one can declare

```

object MyBlackBoard
  fun draw: Square f { ... }
  fun draw: Triangle f { ... }
  fun draw: Circle f { ... }
  fun draw: Shape f { ... }
  private String name
end

```

There are four `draw` methods that are considered different by the compiler. In a message send

```

MyBlackBoard draw: fig

```

the runtime system searches for a `draw` method in prototype `MyBlackBoard` in the textual order in which

the methods were declared. It first checks whether `fig` references a prototype which is a sub-prototype from `Square` (that is, whether the prototype extends `Square`, assuming `Square` is a prototype and not an interface). If it is not, the search continues in the second method,

```
draw: Triangle f
```

and so on. If an adequate method were not found in this prototype, the search would continue in the super-prototype.

Subtyping and Method Search

The definition of subtyping in Cyan considers that prototype `S` is a subtype of `T` if `S` inherits from `T` (in this case `T` is a prototype) or if `S` implements interface `T`. An interface `S` is a subtype of interface `T` if `S` extends `T`. This is a pretty usual definition of subtyping.

In the general case, in a message send

```
p draw: fig
```

the algorithm searches for an adequate method in the object the variable `p` refer to and then, if this search fails, proceeds up in the inheritance hierarchy. Suppose `C` inherits from `B` that inherits from `A`. Variable `x` is declared with type `B` and refers to a `C` object at runtime. Consider the message send

```
x first: expr1 second: expr2
```

At runtime a search is made for a method of object `C` such that:

- (a) the method has selectors `first:` and `second:` and;
- (b) selector `first:` of the method takes a single parameter of type `T` and the runtime type of `expr1` is subtype of `T`. The same applies to selector `second:` and `expr2`;

The methods are searched for in object `C` in the *textually* declared order. The return value type is not considered in this search. If no adequate method is found in object `C`, the search continues at object `B`. If again no method is found, the search continues at object `A`.

The compiler makes almost exactly this search with just one difference: the search for the method starts at the declared type of `x`, `B`.

This unusual runtime search for a method is used for two reasons:

- (a) it can be employed in dynamically-typed languages. Cyan was designed to allow a smooth transition between dynamic and static typing. Cyan will not demand the declaration of types for variables (including parameters and instance variables). After the program is working, types can be added. The algorithm that searches for a method described above can be used in dynamically and statically-typed languages;
- (b) it is used in the Cyan exception system. When looking for an exception treatment, the textual order is the correct order to follow. Just like in Java/C++/etc in which the catch clauses after a try block are checked in the order in which they were declared after an exception is thrown in the try block.

The programmer should be aware that to declare two methods such that:

- (a) they have the same selectors and;
- (b) for each selector, the number of parameters is the same.

will make message send much slower than the normal.

Methods that differ only in the return value type cannot belong to the same prototype. Then it is illegal to declare methods `Int id` and `String id` in the same prototype (even if one of them is inherited).

The search for a method in Cyan makes the language supports a kind of multi-methods. The linking “message”-“method” considers not only the message receiver but also other parameters of the message (if they exist). Unlike other object-oriented languages, the parameter types are inspected at runtime in order to discover which method should be called.

Arrays

Array prototypes are declared using the syntax: `Array<A>` in which `A` is the array element type. Only one-dimensional arrays are supported. A literal array object is created using `{# element list #}`, as in the example:

```
var n = 5;
var anIntArray = {# 1, 2, (Math sqr: n) #};
var Array<String> aStringArray;
aStringArray = {# "one", "t" + "wo" #};
```

This code creates two literal arrays. `anIntArray` will have elements 1, 2, and 25, assuming the existence of a `Math` prototype with a `sqr` method (square the argument). And `aStringArray` will have elements "one" and "two". The array objects are always created at runtime. So a loop

```
1..10 foreach: { (: Int i :)
  Out println: {# i-1, i, i + 1 #}
}
```

Creates ten different literal arrays at runtime. The type of a literal array is `Array<A>` in which `A` is the type of the *first* element of the literal array. Therefore

```
var fa = {# 1.0, 2, 3 #};
declares fa as a Array<Float>.
```

Mixin Objects

A prototype can inherit from any number of mixin prototypes. A mixin prototype is declared as in the example

```
mixin(Window) object Border
  public Int borderColor;
  ...
  public override fun draw {
    drawBorder;
    super draw;
  }
  fun drawBorder { ... }
end
```

Here `Border` is a mixin prototype that may be inherited by prototype `Window` or its sub-prototypes (`Window` could be an interface too). Inside this mixin prototype, methods of `Border` may send messages to `super` or `self` as if `Border` inherited from `Window` (or as if `Border` inherited from some class implementing interface `Window`). Code

```
object Window mixin Border
  fun draw { ... }
```

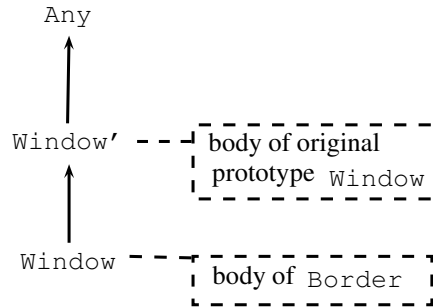


Figure 1.3: The resulting `Window` prototype after the mixin inheritance is applied

```

...
end

```

makes `Window` inherit from `Border`. The word “*inherits*” here is misleading. In fact, the compiler creates a prototype `Window'` with the contents of `Window`, creates a prototype `Window` with the contents of `Border`, and makes `Window` inherit from `Window'`.⁴ What the compiler does is a textual copy of the text of `Window` to a new text create for `Window'`. Then it deletes the text of `Window` putting in its place the text of `Border`. Figure 1.3 illustrates the resulting `Window` prototype.

A mixin declared as

```

mixin(A) object B
...
end

```

should obey the same restrictions as a prototype that inherits from `A`. In particular, if `B` declares a method

```
public override fun get -> T
```

that is already defined in `A`, it should be declared with the keyword `override` and its return value type should be equal to the return value type of method `get` of `A` or it should be a subtype of it:

```

object A
  fun get -> Person { ... }
end

mixin(A) object B
  public override fun get -> Worker { ... }
end

```

Prototype `Border` may add behavior to object `Window`. For example, it defines a `draw` method that draws a border and calls `draw` of `Window` using `super` — see the example. Then,

```
Window draw
```

first calls `draw` of `Border`. This method calls method `drawBorder` of `Border` and then `draw` of `Window` using `super`.

Mixin prototypes can also be dynamically attached to objects. Suppose mixin `Border` is not inherited from prototype `Window`. A mixin object of `Border` may be dynamically attached to a `Window` object using the `attachMixin:` method inherited from `Any`:

```

var w = Window new;
  // other initializations of w

```

⁴`Window'` is just a new name.

```

...
// calls draw of Window: no border
w draw;
w attachMixin: Border;
// calls draw of Border
w draw;

```

`Border` works like a runtime metaobject with almost the same semantics as shells of the Green language [dOGab]. Any messages sent to `Window` will now be searched first in `Border` and then in `Window`. When `Window` is cloned or a new object is created from it using `new`, a new `Border` object is created too.

Dynamic Typing

Although Cyan is statically-typed, it supports some features of dynamically-typed languages. A message send whose selectors are preceded by `?` is not checked at compile-time. That is, the compiler does not check whether the static type of the expression receiving that message declares a method with those selectors. For example, in the code below, the compiler does not check whether prototype `Person` defines a method with selectors `name:` and `age:` that accepts as parameters a `String` and an `Int`.

```

var p Person;
...
p ?name: "Peter" ?age: 31;

```

This non-checked message send is useful when the exact type of the receiver is not known:

```

fun printArray: Array<Any> anArray {
  anArray foreach: { (: elem Any :)
    elem ?printObj
  }
}

```

The array could have objects of any type. At runtime, a message `printObj` is sent to all of them. If all objects of the array implemented a `Printable` interface, then we could declare parameter `anArray` with type `Array<Printable>`. However, this may not be the case and the above code would be the only way of sending message `printObj` to all array objects.

The compiler does not do any type checking using the returned value of a dynamic method. That is, the compiler considers that

```
if obj ?get { ... }
```

is type correct, even though it does not know at compile-time if `obj ?get` returns a boolean value.

Dynamic checking with `?` plus the reflective facilities of Cyan can be used to create objects with dynamic fields. Object `DTuple` of the language library allows one to add fields dynamically:

```

var t = DTuple new;
// add field "name" to t
t ?name: "Carolina";
// prints "Carolina"
Out println: (t ?name);
// if uncommented the line below would produce a runtime error
//Out println: (t ?age);
t ?age: 1;

```

```

    // prints 1
Out println: (t ?age);
    // if uncommented the line below would produce a
    // **compile-time** error because DTuple does not
    // have an "age" method
Out println: (t age);

```

Here fields `name` and `age` were dynamically added to object `t`.

Metaobject `@dynOnce` used before a prototype makes types optional in the declaration of variables and return value types.

```

@dynOnce object Person
  public name
  public age
  fun addAge: n {
    // unnecessary and didactic code
    var sum;
    sum = age + n;
    age = sum
  }
  fun print {
    Out println: name + " ({age} years old)"
  }
end

```

With `@dynOnce`, types become optional in `Person`. Suppose that in the first run of the program, the following code is executed.

```

var p Person;
p name: "Turing";
p age: 100;
p print;

```

Then at the end of program execution, the compiler adds some of the missing types in the declaration of `Person`:

```

@dynOnce object Person
  public String name
  public Int age
  fun addAge: n {
    // unnecessary and didactic code
    var sum;
    sum = age + n;
    age = sum
  }
  fun print {
    Out println: name + " ({age} years old)"
  }
end

```

However, `addAge:` was not used in this run of the program and its parameter `n` does not have a type yet. In the next run, suppose statement


```
Person addAge: 1
```

is executed. Then all the missing types of `Person` have been established. Again the compiler changes the source code of `Person` to

```
object Person
  public String name
  public Int age
  fun addAge: Int n {
    // unnecessary and didactic code
    var Int sum;
    sum = age + n;
    age = sum
  }
  fun print {
    Out println: name + " (#{age} years old)"
  }
end
```

Since all variables and return value types are known, the call to metaobject `@dynOnce` is removed from the source code.

There is also a metaobject `@dynAlways` whose call should precede a prototype declaration. This prototype should not declare any types for variables or return value methods. The compiler will not issue an error because of the missing types. This prototype becomes a dynamically-typed piece of code. The source code is not changed afterwards by the compiler.

Type `Dyn` is a virtual type used for dynamic typing. A variable of type `Dyn` can receive in assignments an expression of any type. And an expression of type `Dyn` can be assigned to a variable of any type. All message sends to an expression of type `Dyn` is considered correct by the compiler.

Expressions in Strings

In a string, a `#` not preceded by a `\` should be followed either by a valid identifier or an expression between `{` and `}`. The identifier should be a parameter, local variable, or unary method of the current object. The result is that the identifier or the expression between `{` and `}` is converted at runtime to a string (through the `asString` method) and concatenated to the string. Let us see an example:

```
var name = "Johnson";
var n = 3;
var johnsonSalary Float = 7000.0;
Out println: "Person name = #name, id = #{n*n+1}, salary = #johnsonSalary";
```

This code prints

```
Person name = Johnson, id = 10, salary = 7000.0
```

The last line is completely equivalent to

```
Out println: "Person name = " + name + ", id = " + (n*n+1) + ",
            salary = " + johnsonSalary;
```

Generic Prototypes

Cyan also supports generic prototypes in a form similar to other languages but with some important differences. First, a family of generic prototypes may share a single name but different parameters. For

example, there is a single name `Tuple` that is used for tuples of any number of parameters (as many as there are in the library):

```
var Tuple<String> aName;
var Tuple<String, Int> p;
aName f1: "Lívia"
    // prints Lívia
Out println: (aName f1);
p f1: "Carol"
p f2: 1
    // prints "name: Carol age: 1". Here + concatenates strings
Out println: "name: " + (p f1) + " age: " + (p f2);
```

Second, it is possible to use field names as parameters:

```
var NTuple<name, String> aName;
var NTuple<name, String, age, Int> p;
aName name: "Lívia"
    // prints Lívia
Out println: (aName name);
p name: "Carol"
p age: 1
    // prints "name: Carol age: 1"
Out println: "name: " + (p name) + " age: " + (p age);
```

A generic prototype is considered different from the prototype without parameters too:

```
object Box
    public Any value
end

object Box<T>
    public T value
end

...
var giftBox = Box new;
var intBox = Box<Int> new;
```

An unnamed literal tuple is defined between `[.` and `.]` as in

```
var p = [."Lívia", 4 .];
Out println: (p f1), " age ", (p f2);
    // or
var Tuple<String, Int> q;
q = [."Lívia", 4 .];
```

A named literal tuple demands the name of the fields:

```
var p = [."name":"Lívia", age:4 .];
Out println: ((p name) + " age " + (p age));
    // or
var NTuple<name, String, age, Int> q;
q = [."name":"Lívia", age:4 .];
```

Multiple Assignments

Multiple assignments can be used to extract the values of literal tuples:

```
var Livia = [ . "Livia", 4 . ];
var Carol = [ . name:"Carolina", age:1 . ];
var String name;
var Int age;

// multiple assignment
name, age = Livia;
name, age = Carol;
```

The same applies to return value of methods:

```
object Person
...
  fun getInfo -> Tuple<String, Int> {
    return [ . name, age . ]
  }
  private String name
  private Int age
end

...
var Myself = Person new;
```

```
name, age = Myself getInfo;
```

Both unnamed and named tuples can be used in multiple assignments. If there are less tuple fields than the number of variables of the left of symbol “=”, then the compiler issues an error. If there are more fields than the number of variables, the extra fields are ignored.

Anonymous Functions

Cyan supports statically-typed anonymous functions, which are called blocks in Smalltalk. An anonymous function is a literal object that can access local variables and instance variables. It is delimited by { and } and can have parameters which should be put between (: and :) as in:

```
var b = { (: Int x :) ^x*x };
// prints 25
Out println: (b eval: 5);
```

Here { (: Int x:) ^x*x } is a function with one Int parameter, x. The return value of the function is the expression following the symbol “^”. The return value type may be omitted in the declaration — it will be deduced by the compiler. This function takes a parameter and returns the square of it. A function is an literal object with a method eval or eval: (if it has parameters as the one above). The statements given in the function can be called by sending message eval or eval: to it, as in “b eval: 5”.

A function can also access a local variable:

```
var y = 2;
```

```
var b = { (: Int x :) ^ x + y };
// prints 7
Out println: (b eval: 5);
```

As full objects, functions can be passed as parameters:

```
object Loop
  fun until: (Function<Boolean> test) do: (Function<Nil> b) {
    b eval;
    (test eval) ifTrue: { until: test do: b }
  }
end
...
```

```
// prints "i = 0", "i = 1", ... without the "s"
var i = 0;
Loop until: { ^ i < 10 } do: {
  Out println: "i = #i";
  ++i
}
```

Here prototype `Loop` defines a method `until:do:` which takes as parameters a function that returns a Boolean value (`Function<Boolean>`) and a function that returns nothing (`Function<Nil>`). The second function is evaluated until the first function evaluated to `false` (and at least one time). Notation `"i = #i"` is equivalent to `("i = " + i)`. If an expression should come after `#`, then we should do `"i = #{i + 1}"`, which is the same as `("i = " + (i+1))`. Note that both functions passed as parameters to method `until:do:` use the local variable `i`, which is a local variable.

Functions are useful to iterate over collections. For example,

```
var v = {# 1, 2, 3, 4, 5, 6 #};
// sum all elements of vector v
var sum = 0;
v foreach: { (: Int x :) sum = sum + x
};
```

Method `foreach:` of the array `v` calls the function (as in `"b eval: 5"`) for each of the array elements. The sum of all elements is then put in variable `sum`.

Sometimes we do not want to change the value of a local variable in a function. In these cases, we should precede the variable by `%`:

```
var y = 2;
var b = { (: Int x :) y = y + 1; ^ x + y };
// prints 8
Out println: (b eval: 5);
// prints 3
Out println: y;
y = 4;
// prints 10
Out println: (b eval: 5);
// prints 5
Out println: y;
```

```

var y = 2;
var c = { (: Int x :) %y = %y + 1; ^ x + %y };
  // prints 8
Out println: (c eval: 5);
  // prints 2
Out println: y;
y = 4;
  // prints 8
Out println: (c eval: 5);
  // prints 4
Out println: y;

```

The value of local variable `y` is copied to a variable called `y` that is local to the function. This copy is made at the creation of the function, when the value of `y` is assigned to an instance variable of the function object. Changes to this variable are not reflected into the original local variable.

However, both the original variable and the `%`-variable refer to the same object. Message sends to this object using the `%`-variable or the original variable will produce the same result. The variables are different but they refer to the same object.

```

var p = Person name: "Newton" age: 370;
{ // here %p and p DO refer to the same object
  %p = Person new;
  // here %p and p do NOT refer to the same object
  %p name: "Gauss" age: 235
} eval;
  // prints "Newton"
Out println: (p name);
{ // here %p and p DO refer to the same object
  %p name: "Gauss";
  // the name of the object Person was
  // changed to "Gauss"
} eval;
  // prints "Gauss"
Out println: (p name);

```

There are two kinds of functions: those that accesses local variables (without the `%` qualifier) or have a `return` statement and those that do not have any of these things. They are called restricted and unrestricted functions, respectively (for short, r-functions and u-functions). r-functions cannot be stored in instance variables. If this were allowed, the function could be used after the local variables it uses ceased to exist. There are special rules for type checking of r-functions. The rules make sure an r-function will not outlive the local variables it uses in its body. An u-function can be assigned to a variable whose type is an r-function. Since parameter passing is a kind of assignment, r-functions can be passed as parameters — it is only necessary that the formal parameters have r-functions as types. In this way, anonymous functions (functions) in Cyan are statically-typed and retain almost all functionalities of anonymous functions of dynamic languages such as Smalltalk.

Context Objects

Context objects are a generalization of functions and internal (or inner) classes. Besides that, they allow a form of language-C-like safe pointers. The variables external to the function are made explicit in a

context object, freeing it from the context in which it is used. For example, consider the function

```
{ (: Int x :) sum = sum + x }
```

It cannot be reused because it uses external (to the function) variable `sum` and because it is a literal object. Using context objects, the dependence of the function to this variable is made explicit:

```
// UFunction<Int, Nil> is a function that takes an Int
// as parameter and does not return anything
object Sum(Int &sum) extends UFunction<Int, Nil>
  fun eval: Int x {
    sum = sum + x
  }
end
```

```
...
// sum the elements of array v
var s = 0;
v foreach: Sum(s)
```

Context objects may have one or more parameters given between (and) after the object name. These correspond to the variables that are external to the function (`sum` in this case). This context object implements interface `Function<Int, Nil>` which represents functions that take an `Int` as a parameter and returns nothing. Method `eval:` contains the same code as the original function. In line

```
v foreach: Sum(s)
```

expression “`Sum(s)`” creates at runtime an object of `Sum` in which `sum` represents the same variable as `s`. When another object is assigned to `sum` in the context object, this same object is assigned to `s`. It is as if `sum` and `s` were exactly the same variable.

Prototype `Sum` can be used in other methods making the code of `eval:` reusable. Reuse is not possible with functions because they are literals. Context objects can be generic, making them even more useful:

```
object Sum<T>(T &sum) extends UFunction<T, Nil>
  fun eval: T x {
    sum = sum + x
  }
end
```

```
...
// concatenate the elements of array v
var v = {# "but", "ter", "fly" #};
var String s;
v foreach: Sum<String>(s)
```

Now context object `Sum` is used to concatenate the elements of vector `v` (which is a string array).

A context-object parameter may be preceded by `%` to mean that it is a *copy parameter*. That means changes in the context-object parameter are not propagated to the real argument:

```
object Sum(Int %sum) extends UFunction<Int, Nil>
  fun eval: Int x {
    sum = sum + x
  }
end
```

```

...
    // do not sum the elements of array v
var s = 0;
v foreach: Sum(s);
assert: (s == 0);

```

Method `assert:` of `Any` checks whether its argument returns `true`. It ends the program otherwise. In this example, the final value of `s` will be 0.

A context-object parameter may be preceded by `*` to indicate that the real argument to the context object should be an instance variable. This kind of parameter is called *instance variable parameter*. Parameters whose types are preceded by `&` are called *reference parameters* (see first example). Context objects that have at least one *reference parameter* are called restricted context objects and have the same restrictions as r-functions. All the other are unrestricted context objects and there is no restriction on their use.

Context objects are a generalization of both functions and nested objects, a concept similar to nested or inner classes. That is, a class declared inside other class that can access the instance variables and method of it. However, class B declared inside class A is not reusable with other classes. Class B will always be attached to A. In Cyan, B may be implemented as a context object that may be attached to an object A (that play the rôle of class A) or to any other prototype that has instance variables of the types of the parameters of B. Besides that, both referenced parameters and instance variable parameters implement a kind of language-C like pointers. In fact, it is as if the context-object parameter were a pointer to the real argument:

```

    // C
int *sum;
int s = 0;
sum = &s;
*sum = *sum + 1;
    // value of s was changed
printf("%d\n", s);

```

Context Functions

A context function is a kind of method that can be plugged to more than one prototype. It is also a kind of literal context object with a single `eval` or `eval:` method. A context function was created to allow a method to be attached to all objects of a prototype. This concept is very similar to an object that represent a “class method” in class-based object-oriented languages.

A context function that returns the name of a color is declared as

```

var colorNameCB = { (: IColor self -> String :)
    // colorTable takes an integer and returns
    // a string with the name of the color of
    // that integer
    return %colorTable[ self color ]
};

```

`self` appears in the first parameter declaration. Its type is `IColor`:

```

interface IColor
    fun color -> Int

```

```

    fun color: Int
end

```

Inside the function, `self` has type `IColor`. Message sends to `self` should match methods declared in the `IColor` interface. The only message send in this example is “`self color`”, which calls method `color` of `self`.

The context function referenced by `colorNameCB` may be added to any object that implements interface `IColor` using method `addMethod:` of `Any`.

```

private object Shape implements IColor
    fun color -> Int { ^_color }
    fun color: Int newColor { _color = newColor }
    ...
end

```

...

```

var colorNameCB = {
    (: IColor self :)
    return colorTable[ self color ]
};

```

```

Shape addMethod:
    selector: #colorName
    returnType: String
    body: colorNameCB;

```

```

Out println: (Shape ?colorName);

```

Of course, the method added to `Shape`,
 String colorName

should be called using `?` because it is not in the `Shape` signature. We could also have written

```

private object Button implements IColor
    ...
end
...

```

```

Button addMethod:
    selector: #colorName
    returnType: String
    body: {
        (: IColor self :)
        return colorTable[ self color ]
    };

```

```

var b = Button();
Out println: (b ?colorName);

```

Since a context function has a `self` parameter, it cannot be called as a regular function:

```

var s = colorNameCB eval;

```

That will result in a compile error because the context function does not have an `eval` method. Instead, it has a `bindToFunction:` method to set the function `self`:

```

var b = colorNameCB bindToFunction: Shape;

```



```
( b eval ) println;
```

Grammar Methods

A method declared with selectors `s1:s2` can only be called through a message send `s1: e1 s2: e2` in which `e1` and `e2` are expressions. Grammar methods do not fix the selectors of the message send. Using operators of *regular expressions* a grammar method may specify that some selectors can be repeated, some are optional, there can be one or more parameters to a given selector, there are alternative selectors and just one of them can be used.

A method that takes a variable number of `Int` arguments is declared as shown below.

```
    // a set of integers
object IntSet
  fun (add: (Int)+) t { ... }
  ...
end
```

The `+` after `(Int)` indicates that after `add:` there may be *one or more* integer arguments:

```
IntSet add: 0, 2, 4;
var odd = IntSet new;
odd add: 1, 3;
```

Maybe we would like to repeat the selector for each argument. That can be made:

```
    // a set of integers
object IntSet
  fun (add: Int)+ t { ... }
  ...
end
```

```
...
IntSet add: 0 add: 2 add: 4;
var odd = IntSet new;
odd add: 1 add: 3;
```

The `t` that appears after `+` is the method parameter. Every grammar method is declared using `(and)` and, after the signature there should appear exactly one parameter. Its type may be omitted. In this case the compiler will assign a type to it — this same type can be given by the programmer. There are rules for calculating the type of the single parameter of a grammar method (See Chapter 9). This type depends on the regular expression used to define the method. In both of the above examples, the type of `t` is

```
    Array<Int>
```

So the method could have been declared as

```
    // a set of integers
object IntSet
  fun (add: Int)+ Array<Int> t {
    // the simplest and inefficient way of
    // inserting elements into intArray
    t foreach: { (: Int elem )
```

```

        // inserts elem in intArray
        // create a new array if there is not
        // space in this one
        ...
    }
}
...
private Array<Int> intArray
end

```

A grammar method may use all of the regular expression operators: A^+ matches one or more A 's, A^* matches zero or more A 's, $A?$ matches A or nothing (A is optional), $A \mid B$ matches A or B (but not both), and AB matches A followed by B . The \mid operator may be used with types:

```
fun (add: Int | String) Union<Int, String> t { ... }
```

Method `add`: may receive as parameters an `Int` or a `String`.

Grammar methods are useful for implementing Domain Specific Languages (DSL). In fact, every grammar method can be considered as implementing a DSL. The advantages of using grammar methods for DSL are that the lexical and syntactical analysis and the building of the Abstract Syntax Tree are automatically made by the compiler. The parsing is based on the grammar method. The AST of the grammar message is referenced by the single parameter of the grammar method. Besides that, it is possible to replace the ugly type of the single parameter of the grammar method by a more meaningful prototype. Using annotations (Section 5.1) one can annotate a prototype with information and put it as the type of the grammar method parameter. The compiler will know how to use this prototype in order to build the AST of a grammar message. See Section 9.7 for more details.

There is one problem left: grammar methods are defined using regular expression operators. Therefore they cannot define context-free languages. This problem is easily solved by using parenthesis in the message send and sometimes more than one grammar method. The details are given in Section 9.5. However, we present one example that implements Lisp-like lists:

```
object GenList
  fun (L: (List | Int)* ) Array<Union<List, Int>> t -> List {
    // here parameter t is converted into a list object
    ...
  }
end

```

The grammar defining a Lisp list is not a regular grammar — there is a recursion because a list can be a list element: `(1, (2, 3), 4)`

This list can be built using parenthesis in a grammar message send

```
var b = GenList L: 1, (GenList L: 2, 3), 4;
```

Another example of domain specific language implemented using grammar method uses commands given to a radio-controlled car (a toy). The car obeys commands related to movement such as to turn left (a certain number of degrees) right, increase speed, decrease speed, move n centimeters, turn on, and off. Assuming the existence of a prototype `CarRC` with an appropriate grammar method, one could write

```
CarRC on:
  left: 30
  move: 100
  right: 20;
```

```

CarRC move: 200
      speedUp: 1
      move: 50
      speedDown: 1
      off;

```

These two message would cause the call of the same grammar method, which is declared as

```

object CarRC
  fun (on: | off: | move: Int | left: Int | right: Int | speedUp: Int | speedDown:
      Int)+
      Array<Union<Any, Any, Int, Int, Int, Int, Int>> t {

      // here should come the implementation of the commands
    }
    // other methods
    ...
end

```

The grammar method could really send orders to a real car in the method body, after interpreting the `t` parameter.

The uses of grammar methods are endless. They can define optional parameters, methods with variable number of parameters, and mainly DSL's. One could define methods for SQL, XML (at least part of it!), parallel programming, graphical user interfaces, any small language. It takes minutes to implement a small DSL, not hours.

Methods as Objects

Methods are objects too. Therefore it is possible to pass a method as parameter. As an example one can use

```

P getMethod: "s1:T1 s2:T2, T3"

```

to reference the sole method of prototype P.

```

object P
  fun s1: T1 p1
      s2: (T2 p2, T3 p3) {
      /* empty */
    }
end

```

The ability of referring to a method is very useful in graphical user interfaces as the example below shows.

```

object MenuItem
  fun onMouseClick: UFunction<Nil> b {
    ...
  }
end

object Help
  fun show { ... }

```

```

    ...
end

object FileMenu
  fun open { ... }
end

var helpItem = MenuItem new;
helpItem onMouseClick: (Help getMethod: "show" );
var openItem = MenuItem new;
openItem onMouseClick: (FileMenu getMethod: "open");
...

```

We could also have passed u-functions to method `onMouseClick`:

```
openItem onMouseClick: { self.helpObject show }
```

There may even exist a table containing methods and functions:

```

var codeTable = Hashtable<String, UFunction<Int, Int>> new;
codeTable key: "square" value: (Math getMethod: "sqr");
codeTable key: "twice" value: { (: Int n :) ^n*n };
codeTable key: "succ" value: { (: Int n :) ^n+1 };
codeTable key: "pred" value: { (: Int n :) ^n-1 };
  // 5 getMethod: "+ Int" is method "fun Int + (Int other)"
  // of object 5
codeTable key: "add" value: (5 getMethod: "+ Int");
...
  // read the function name from the keyboard
  // and get the u-function of it
var b = codeTable key: (In readLine);
  // call the command
Out println: (b eval: 2);

```

The Exception Handling System

The exception handling system of Cyan was based on that of Green. However, it has important improvements when compared with the EHS of this last language. Both are completely object-oriented, contrary to all systems of languages we know of. An exception is thrown by sending message `throw:` to `self` passing the exception object as parameter. This exception object is exactly the exception objects of other languages. Method `throw:` is defined in the super-prototype of every one, `Any`.

The simplest way of catching an exception is to pass the exception treatment as parameters to `catch:` selectors in a message send to a function.

```

var age Int;
{
  age = In readInt;
  if age < 0 {
    throw: ExceptionNegAge(age)
  }
}

```

```
} catch: { (: ExceptionNegAge e :) Out println: "Age #{e age} is negative" };
```

Here exception `ExceptionNegAge` is thrown by message send

```
throw: ExceptionNegAge(age)
```

in which “`ExceptionNegAge(age)`” is a short form of “`(ExceptionNegAge new: age)`”.

When the exception is thrown the control is transferred to the function passed as parameter to `catch:`. The error message is then printed. Object `ExceptionNegAge` should be a sub-prototype of `CyException`.

```
object ExceptionNegAge extends CyException
  // '@init(age)' creates a constructor with
  // parameter age
  @init(age)
  public Int age
end
```

This example in Java would be

```
int age;
try {
    age = In.readInt();
    if ( age < 0 )
        throw new ExceptionNegAge(age);
} catch ( ExceptionNegAge e ) {
    System.out.println("Age " + e.getAge() + " is negative");
}
```

There may be as many `catch:` selectors as necessary, each one taking a single parameter.

```
var Int age;
{
    age = In readInt;
    if age < 0 {
        throw: ExceptionNegAge(age);
    }
    else if age > 127 {
        throw: ExceptionTooOldAge(age)
    }
} catch: { (: ExceptionNegAge e :) Out println: "Age #{e age} is negative" }
catch: { (: ExceptionTooOldAge e :) Out println: "Age #{e age} is out of limits" };
```

The `catch:` parameter may be any object with one or more `eval:` methods, each of them accepting one parameter whose type is sub-prototype of `CyException`. So we could write:

```
var Int age;
{
    age = In readInt;
    if age < 0 {
        throw: ExceptionNegAge(age);
    }
    else if age > 127 {
        throw: ExceptionTooOldAge(age)
    }
}
```

```

    }
} catch: ExceptionCatchAge;

```

Consider that `ExceptionCatchAge` is

```

object ExceptionCatchAge
  fun eval: ExceptionNegAge e {
    Out println: "Age #{e age} is negative"
  }
  fun eval: ExceptionTooOldAge e {
    Out println: "Age #{e age} is out of limits"
  }
end

```

This new implementation produces the same results as the previous one. When an exception `E` is thrown in the function that reads the age, the runtime system starts a search in the parameter to `catch:`, which is `ExceptionCatchAge`. It searches for an `eval:` method that can accept `E` as parameter in the textual order in which the methods are declared. This is exactly as the search made after a message send. The result is exactly the same as the code with two functions passed as parameters to two catch selectors.

The exception handling system of Cyan has several advantages over the traditional approach: exception treatment can be reused, `ExceptionCatchAge` can be used in many places, exception treatment can be organized in a hierarchy (`ExceptionCatchAge` can be inherited and some `eval:` methods can be overridden. Other methods can be added.), the EHS is integrated in the language (it is also object-oriented), one can use metaobjects with the EHS, and there can be libraries of treatment code. For short, all the power of object-oriented programming is brought to exception handling and treatment. Since the Cyan EHS has all of the advantages of the EHS of Green, the reader can know more about its features in an article by José [dOGaa].

Metaobjects

Compile-time metaobjects are objects that can change the behavior of the program, add information to it, or can inspect the source code. A compile-time metaobject is attached to a prototype using `@` as in

```

@checkStyle object University
  @log fun name -> String { return uName }
  ...
end
...
@singleton object Earth
  ...
end

object Help
  ...
end

...
var t = @text(<+ ... +>);
...

```

Metaobject `checkStyle` is activated at compile-time in the first line of this example. It is attached to specific points of the compiler controlling the compilation of prototype `University`. It could check whether the prototype name, the method names, the instance variable names, and local variables follow some conventions for identifiers (prototype in lower case except the first letter, method selectors, instance variables, and local variables in lower case). The compiler calls methods of the metaobject at some points of the compilation. It is as if the metaobject was added to the compiler. Which method is called at which point is defined by the Meta-Object Protocol (MOP) which has not been defined yet. Initially the Cyan metaobjects will be written in Java because the compiler is written in Java. Afterwards they will be made in Cyan.

There is a call in the example to a metaobject `text`. After the name there may appear a sequence of symbols. The argument to the metaobject call ends with a sequence that mirrors the start sequence. So (`<+` is ended by `>+`). Almost any sequence is valid and different metaobject calls may use the same symbol sequence. The text in between is passed as argument to a call to a specific method of this metaobject (say, `parse`) defined by the MOP. `text` is a pre-defined metaobject. A call to it is replaced by a literal array of `Char`'s with all the characters between the delimiters. Note that to say “*metaobject call*” is an abuse of language. Metaobjects are objects and objects are not called. Methods are called. The compiler will in fact call some specific methods of the metaobject which are not specified in the metaobject. It would be more precise to say that the metaobject is employed in the code in a line as

```
var t = @text(<+ ... >+);
```

The metaobject may return a string of characters that is the Cyan code corresponding to the metaobject call. It does so in the call to `text`. It does not in the call to `checkStyle` (which does not generate code). Instead of returning a string, the metaobject method may return a modified AST of the prototype or the method (in case of `log`). Metaobject `log` would add code to the start of the method to log how many times it was called. This information would be available to other parts of the code. Again, a metaobject does not return anything. It is an object. What happens is that a method of the metaobject, not specified in the code, is called and it returns something.

User-Defined Literal Objects

Cyan supports user-defined literal objects. One can define a literal object delimited by `<+(` and `>+` by a call to the pre-defined metaobject `literalObject`:

```
@literalObject<<
  start: "<+("
  parse: ParseList
>>
```

After `parse:` there should appear the name of a Java class that should have methods to parse the text that appear between the delimiters. In the source code that appears textually below the call to `literalObject`, one could write

```
var myList = <+( 1, "Hi", 3, 5, true )>+;
```

The text “ 1, "Hi", 3, 5, true ” will be passed as an argument to a method, say `parseRetString`, of the Java class `ParseList`. This method would return a string that replaces `<+(1, "Hi", 3, 5, true)>+`

`literalObject` supports several other options beside `start:` and `parse:`. Some of them allow one to define a metaobject using a regular expression just like a grammar method. Using this form, it is easy

to define, for example, a literal dictionary:

```
var dict = {* "Copernicus":"astronomy"  
            "Gauss":"math"  
            "Galenus":"medicine" *};
```

This literal object would be defined as

```
@literalObject<<  
  start: "{*"   
  regexpr: ( String ":" String )*   
  type: ListStringString   
  addAll: List<String, String>   
>>
```

It is possible to define named literal objects that resemble metaobjects but are not directly related to them. For example, the literal dictionary could have a name `Dict`. It would be used as

```
var dict1 = Dict{* "Copernicus":"astronomy" "Gauss":"math" *};  
var number = Dict<<@ "one":1 "two":2 @>>;
```

The start and end symbols can vary from literal to literal as in this example. The only requirement is that the end is the mirror of the start.

Linking Past-Future

The source code of a Cyan program can be given in XML (Chapter 2). When a source file is compiled, the compiler can write some information of the current version of the code in this XML file, which also contains the code. In future compilations the compiler can issue warning messages based on the previous versions of the code. For example, if the current version of a prototype changed the order of an overloaded method in relation of a previous version, the compiler may warn that some message sends may call know a method different from what it called in the previous version. As an example, suppose the first version of a prototype is as given below. `Manager` inherits from `Worker`.

```
object Company  
  fun pay: Manager manager { manager deposit: 9000 }  
  fun pay: Worker worker {  
    // pay worker, the details are not important  
  }  
  ...  
end
```

A message send

```
Company pay: Manager;
```

calls the first method of `Company` as expected. However, if the `pay` method are textually exchanged, the method called will be `pay: Worker`. This may be terrible consequences. The compiler could warn that this change may have introduced a bug in the program. See more about this in Section 4.5.

Another example of use of the past is when there is a sequence of `if`'s that test for the prototype of a variable:

```
var Person v;  
...  
if v isA: Student {
```



```
    ...  
}  
else if v isA: Worker {  
    ...  
}
```

Here **Person** is an abstract prototype whose only sub-prototypes are **Student** and **Worker**. Then the cascaded **if** statements cover all cases. However, when another sub-prototype of **Person** is created, this is no longer the case. The compiler or another tool could warn this based on the information kept in the XML file that contains the source code.

Chapter 2

Packages and File organization

We will call “program unit” a prototype declaration or interface. Every source file is a “compilation unit” and may contain one or more prototype declarations. Exactly one of them should be public. The others should be private. Then compilation units are “source files” each one containing one or more prototypes.

A Cyan program is divided in compilation units, program units, and packages that keep the following relationship:

- (a) every file, with extension `.cyan`, declare exactly one public or one protected program unit (but not both) and any number of private program units. Keywords `public`, `protected`, and `private` may precede the program unit to indicate that it is public, protected, or private:

```
...
public object Person
  public String name
  ... // methods
end
```

If no qualifier is used before “object”, then it is considered public. A protected program unit is only visible in its package. A private program unit can only be used in the file in which it is declared. No private entities can appear in the public part of a public or protected entity. So the following code is illegal:

```
...
private object ListElement
  public Int item
  public ListElement next
end

public object List
  private ListElement head
  // oops ... private prototype in the public
  // interface of method getHead
  public ListElement getHead { ^head };
  ... // other methods
end
```

The prototypes (which includes interfaces) defined in a source file hide any prototypes imported in the source file. So it is legal to define a prototype `Test` in a source file and import another prototype `Test` from a package.

- (b) every file should begin with a package declaration as “package ast” in

```
package ast

object Variable
  public String name
  public Type type
  ... // methods
end
```

All objects and interfaces declared in that file will belong to the package “ast”;

- (c) a package is composed by program units spread in one or more source files. The name of a package can be composed by identifiers separated by “.”. All the source files of a package should be in the same directory. The source files of a package id1.id2. ... idn should be in a directory idn which is a sub-directory of id(n-1), and so on. There may be packages id1.id2 and id1.id3 that share a directory id1. Although a directory is shared, the packages are unrelated to each other.
- (d) a Cyan source file is described in XML and has the following structure:

```
<?xml version="1.0"?>
<cyanfile>
<cyansource>
package ast

object Variable
  public String name
  public Type type
  ... // methods
end
</cyansource>
<!-- here comes other elements -->
</cyanfile>
```

The root XML element is `cyanfile`. There is child of `cyanfile` called `cyansource` that contains the source code in Cyan of that file. After that comes other elements. What exactly there are is yet to be defined. Certainly there will be elements that keep the interfaces of all prototypes that every prototype of this file uses. An object interface is composed by the signatures of its public methods. The signature of a method is composed by its selectors, parameter types, and return value type. This information will be put in the XML file by the compiler. The information stored in the XML file can be used to catch errors at compile time that would otherwise go undetected or to improve current error messages. Based in this information, the compiler could check:

- (a) if the textual order of declaration of multi-methods¹ was changed;
- (b) if the return value type of methods was changed (to replace a return value type T by its subtype S is ok. The opposite may introduce errors.);
- (c) if methods were added to multi-methods;

¹Methods with the same name but different parameter types or number of parameters in each selector.

- (d) if the textual order of declaration of the instance variables was changed. This is important for metaobject `@init` when it is called without parameters (See page 73);
- (e) if the type of a parameter of a generic prototype was changed. This can invalidate uses of the generic prototype.

Another use of the XML file would be to store information collected at runtime. The compiler could insert code that checks, for example, if the numbers stored in `Int` variables are dangerous near the limits allowed by this type. In the next compilation the programmer would receive a warning that `Int` should be changed to a library prototype “`BigInt`”.

The compiler could also put in one of the XML elements the restrictions that a generic parameter type should obey. For example, a generic object

```
object TwoItems<T>
  fun set: (T a, T b) { self.a = a; self.b = b; }
  fun max -> T {
    return (a > b) f: a f: b
  }
  private T a, b
end
```

would keep, in the XML file, that generic parameter `T` should support the operator `<`.

To know one more Cyan feature that uses the XML file, see page 115 on metaobject `onChangeWarn`.

If the source file does not start with `<?xml`, then it is assumed that the text contains only source code in Cyan. Then it is optional to put or not the source code in a XML file.

A package is a collection of prototype declarations and interfaces. Every public Cyan prototype declared as `object ObjectName ... end` must be in a file called “`ObjectName.cyan`” (even if it is a XML file). Preceding the object declaration there must appear a package declaration of the form `package packageName` as in the example given above.

Program units defined in a package `packB` can be used in a source file of a package `packA` using the import declaration:

```
package packA
import packB

object Program
  fun run {
    ...
  }
end
```

The public program units of package `packB` are visible in the whole source file. A program unit declared in this source file may have the same name as an imported program unit. The local one takes precedence. ‘;’ is optional after the package name and the import list.

More than one package may be imported; that is, the word `import` may be followed by a list of package names separated by comma. It is legal to import two packages that define two resources (currently, only prototypes) with the same name. However, to use one identifier (program unit) imported from two or more packages it is necessary to prefix it with the package name. See the example below.

```
package pA
```

```

import pB, pC, pD

object Main
  fun doSomething {
    var pB.Person p1; // Person is an object in both packages
    var pD.Person p2;
    ...
  }
end

```

This same rule applies when package pA and pB define resources with the same name.

An object or interface can be used in a file without importing the package in which it was defined. But in this case the identifier should be prefixed by the package name:

```

var v = ast.Variable;
var gui.Window window;

```

There is a package called `cyan.lang` which is imported automatically by every file. This package defines all the basic types, arrays, prototype `System`, function objects, tuples, unions, etc. See Chapter 8.

A program is described by a file with extension “`cyanp`”. This file contains code of a Domain Specific Language called CPL (Cyan Package Language) whose grammar is below. Some items are not described in the grammar: `LeftCharString`, `QualifId`, `RightCharString`, `TEXT`, and `FileName`. `LeftCharString` is any sequence of the symbols

= ! \$ % & * - + ^ ~ ? / : . \ | ([{ <

Note that `>`, `)`, `]`, and `}` are missing from this list. `RightCharString` is any sequence of the same symbols of `LeftCharString` but with `>`, `)`, `]`, and `}` replacing `<`, `(`, `[`, and `{`, respectively. The compiler will check if the closing `RightCharString` of a `LeftCharString` is the inverse of it.

`QualifId` is a sequence of one or more Cyan identifiers separated by “.”. An identifier is a sequence of letters, digits, underscore starting with a letter or underscore. The underscore alone is not considered an identifier. `TEXT` is any text. It may include any character but end-of-file. `FileName` is a string with a file name. The character “\” or “/” is used to separate directories (folders). Any one of these characters may be used. “\x” is not considered an escape character for any x. Then a `FileName` can be

```
"C:\Cyan Material\lib\cyan\lang"
```

The grammar of CPL follows.

```

Program          ::= [ ImportList ] [ CTMOCallList ] “program” [ AtFolder ]
                  [ “main” QualifId ]
                  { CTMOCallList Package }
ImportList       ::= FileName { “,” FileName }
Package          ::= “package” QualifId [ AtFolder ]
AtFolder         ::= “at” FileName
CTMOCallList     ::= { CTMOCall }
CTMOCall         ::= (“@” | “@@” ) Id [ LeftCharString TEXT RightCharString ]
QualifId         ::= { Id “.” } Id

```

As an example, a program in CPL is

```

@checkStyle
@option(addQualifier)
program at "C:\Cyan\example01"
    main main.Program
    @option(no_dynamic) package bank at "C:\Cyan\tests\Bank"
    package cyan.util at "C:\Cyan\"
    package account

```

Keyword `program` starts the project. Optionally “at” specifies the path of the program. If not specified, the default directory is that in which the project file is. After it keyword `main` may appear. It specifies the full path of the main prototype; that is, its package “.” its name. The execution starts in method `run` of this prototype. If not specified, execution will start at prototype `Program` of package `main`. After `program` or `main` (if present), there should appear one or more packages descriptions.

A package description is keyword `package`, the package name, and optionally “at” followed by a string with the package directory. All source files of a package should be in the same directory. The compiler considers that the package is in a directory whose name is the package name with “.” replaced by / or \ (it depends on the separator the operating system uses). In Windows, a package `cyan.util` should be in a directory (folder) “cyan\lang”. This directory is in the package directory, which is that specified by “at” or the program directory (if there is no keyword “at” for the package). As examples, package `cyan.util` is in directory

```
C:\Cyan\cyan\util
```

and package `account` is in directory

```
C:\Cyan\example01\account
```

In the directory of a package there should be zero or more cyan source files. These are of several kinds:

- (a) a file name “Name.cyan” should contain a public prototype `Name`. There are special rules for generic prototypes — see Chapter 7;
- (b) the file name stars with “-”. It is a collection of public prototypes. The compiler will create for each of them a source file starting with “--” in the same directory as the original file. All will import the same packages.

The compiler may be called passing a source file as parameter. The compiler will compile and call method `run: Array<String>` of this file. It should be self-sufficient. Optionally, the file name can starts with the “source name” followed by “-”. After “-” there may appear the name of a prototype. The file should have extension “.cyan”. The compiler will consider that the code inside this file belongs to a prototype whose name is “source name” that inherits from the prototype whose name appears after “-”. For example, if the file name is “MyScan-lib.Script.cyan”, the compiler will create a prototype `MyScan` and make it inherit from `lib.Script`. The source code may be of two kinds: (a) a sequence of methods starting with keyword `fun` (optionally preceded by `public`, `private`, or `protected`) or (b) a sequence of statements. In the first case these methods will be put in prototype `MyScan`. In the second case the compiler will create a public method `run: Array<String>` and put the statements inside it. Some packages will be automatically imported (which ones are yet to be defined). And the prototype will be a dynamically-typed one. That is, variables, parameters, and methods are not demanded to be declared with types — see metaobject `@dynAlwas` in Chapter 6. In the example, `MyScan` will be attached to metaobject `@dynAlwas`.

The program and the packages may be preceded by zero or more metaobject calls. These are of the for `@meta` in which `meta` is the metaobject name. These calls may have parameters and an attached text.

See Chapter 5 for more details. In particular, the compiler options should be parameters of a metaobject options.

ImportList is a list of file names that are imported by this project. These file names should be the directories of projects. Only the metaobjects of these projects are imported.

Chapter 3

Basic Elements

This chapter describes some basic facts on Cyan such as identifiers, number literals, strings, operators, and statements (assignment, loops, etc). First of all, the program execution starts in a method called `run` (without parameters or return value) or

```
run: Array<String>
```

of a prototype specified at compile-time through the compiler or IDE option. Type `Array<String>` is an array of strings. The arguments to `run` are those passed to the program when it is called. In this text (all of it) we usually call `Program` the prototype in which the program execution starts. But the name can be anyone. The program that follows prints all arguments passed to it when it is called.

```
package main

object Program
  fun run: Array<String> args {
    args foreach: { (: String elem :)
      Out println: elem
    }
  }
end
```

3.1 Identifiers

Identifiers should be composed by letters, numbers, and underscore and they should start with a letter or underscore. However, a single underscore is not considered a valid identifier. Upper and lower case letters are considered different.

```
var Int _one;
var Long one000;
var Float ___0;
```

It is expected that the compiler issues a warning if two identifiers visible in the same scope differ only in the case of the letters as “one” and “One”.

3.2 Comments

Comments are parts of the text ignored by the compiler. Cyan supports three kinds of comments:

- anything between `/*` and `*/`. Nested comments are allowed. That is, the comment below ends at line 3.

```
1  /* this is a /* nested
2     comment */
3     that ends here */
```

- anything after `//` till the end of the line;
- compiler-inserted comments of the form `/*# #*/`. The compiler is free to insert and remove this comments at any time.

A comment may appear anywhere (maybe this will change). A comment is replaced by the compiler by a single space.

```
var value = 1/* does value holds 10? */0;
```

This code is the same as `var value = 1 0` and therefore it causes a compile-time error instead of being an assignment of 10 to `value`.

3.3 Keywords

Cyan uses the following keywords:

abstract	char	enum	implements	long	override	slot	void
Any	const	extends	import	macro	package	stackalloc	volatile
Array	default	false	in	match	private	String	when
Boolean	delegate	final	Int		protected	switch	where
boolean	Double	Float	int	mixin	public	true	while
break	double	float	interface	mutable	return	type	with
Byte	Dyn	for	it	Nil	self	val	
byte	each	fun	let	null	shared	var	
case	else	heapalloc	local	object	Short	virtual	
Char	end	if	Long	of	short	Void	

Each of them should be preceded by space, the beginning of a line, or `'` (except `Nil`, `Boolean`, `Char`, `Byte`, `Int`, `Short`, `Long`, `Float`, `Double`, `String`, `self`, `true`, and `false`). Each of them should be followed by space, end of line, end of file, or `'`. A space is a character that makes method `Character.isWhiteSpace(char ch)` of Java return true.

Note that a lot of reserved words are not currently used in the language.

3.4 Assignments

An assignment is made with `"="` as in

```
x = expr;
```

After this statement is executed, variable `x` refer to the object that resulted in the evaluation of `expr` at runtime. The compile-time type of `expr` should be a *subtype* of the compile-time type of `x`. See Section 4.16 for a definition of subtype.

A variable may be declared and assigned a value:

```
var x = expr;
```

The type of `x` will be the compile-time type of `expr`. Both the type of the variable and the expression can be supplied:

```
var Int x = 100;
```

However, the compiler will issue a warning unless `expr` is:

- (a) a literal of a basic type, including `String`, `Nil`, or a prototype;
- (b) a message send whose selector is `clone`, `new` or `new:`. That includes implicit message sends to `new` or `new:` such as `"Person("Livia", 7)"`. These concepts will be seen later;
- (c) a message send whose method, found at compile time, has an attached metaobject called `typedClearly`. That is, in `"person = club president"`, the compiler searches in the compile-time type of `club` for method `president`. If it does not find a method, there is an error. If it finds one, this metaobject will prevent the compiler from issuing a warning. All methods that have a clearly defined return type can be declared with this metaobject. For example, `@clearlyType` is attached to method `readInt` of prototype `In`. It is very clear that this method returns an `Int`.

Cyan supports a restricted form of multiple assignments. There may be any number of comma separated assignable expressions in the left-hand side of `"="` if the right-hand side is a tuple (named or unnamed) with the compatible types. That is, it is legal to write

```
v1, v2, ..., vn = tuple
```

if `tuple` is a tuple with at least `n` fields and the type of field number `i` (starting with 1) is a subtype of the type of `vi`.

```
var Float x, y;  
x, y = [. 1280, 720 .];  
var tuple = [. 1920, 1080 .];  
x, y = tuple;
```

However, a variable cannot be declared in a multiple assignment:

```
var x, y = [. 1280, 720 .]
```

The compiler would sign an error in this code.

The assignment `"v1, v2, ..., vn = tuple"` is equivalent to

```
// tuple may be an expression  
var tmp = tuple;  
vn = tmp fn;  
...  
v2 = tmp f2;  
v1 = tmp f1;
```

A multiple assignment is an expression that returns the value of the first left-hand side variable, which is `v1` in this example.

A method may simulate the return of several values using tuples.

```
object Circle  
  fun getCenter -> NTuple<x, Float, y, Float> {  
    return [. x, y .]  
  }  
  ...  
  private Float x, y // center of the circle
```

```
    private Float radius
end
...
```

```
var Float x, y;
x, y = Circle getCenter;
```

3.5 Basic Types

Cyan has one basic type, starting with an upper case letter, for each of the basic types of Java: `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Char`, and `Boolean`.

Unless said otherwise, Cyan literals of the basic types are defined as those of Java. In particular, the numeric types have the same ranges as the corresponding Java types. `Byte`, `Short`, and `Long` literals should end with `B` or `Byte`, `S` or `Short`, and `L` or `Long`, respectively as in

```
var aByte = 7B;
var aShort = 29Short;
var aLong = 1234567L;
var bLong = 37Long;
var anInt = 223Int;
```

`Int` literals may optionally end with `I` or `Int`. All basic types inherit from `Any`. Therefore there are not two separate hierarchies for basic and normal types. All types obey “reference” semantics. Conceptually, every object is allocated in the heap. However, objects of basic types such as `1`, `3.1415`, and `true` are allocated in the stack most of the time.

Integral literal numbers without a postfixed letter are considered as having type `Int`. Numbers with a dot such as `10.0` as considered as `Float`’s. `Float` literals can end with `F` or `Float`. `Double` literals should end with `D` or `Double`. There is no automatic conversion between types:

```
var Int age;
var Byte byte;
var Float height;
    // ok
age = 21;
    // compile-time error, 0 is Int
byte = 0;
    // ok
byte = 0B;
    // ok
height = 1.65;
    // compile-time error
height = 1;
    // ok
height = 1F;
```

Underscores can be used to separate long numbers as in

```
1_000_000
```

Two underscores cannot appear together as in

1__0

The first symbol cannot be an underscore: `_1_000` would be considered an identifier by the compiler.

The `Boolean` type has two enumerated constants, `false` and `true`, with `false < true`. When `false` is cast to an `Int`, the value returned is 0. `true` is cast to 1. `Char` literals are given between `'` as in

```
'A'  '#'  '\n'
```

Prototype `Nil` has a special status in the language. It is the only prototype that does not inherit from `Any`, the super-prototype of anyone (this will be explained later). `Nil` is not supertype or subtype of anything. Then to a variable of type `Nil` can only be assigned prototype `Nil` and it can only be assigned to a variable or parameter of type `Nil`. Of course, `Nil` cannot be inherited from a prototype.

Methods that do not declare a return type, as

```
fun set: Int newValue { ... }
```

in fact return a value of type `Nil`. Therefore this declaration is equivalent to

```
fun set: Int newValue -> Nil { ... }
```

Any method that has `Nil` as the return type always return `Nil` at the end of its execution. The `return` statement (explained later) is required in methods that return anything other than `Nil`.

Since `Nil` does not have subtypes, a method returning `Nil` can be implemented as not returning a value. After all, it always return the same value.

Prototype `String` represents a read-only string. It has several methods such as `at: []` (for indexing) and `==` (equality). A literal string should be given enclosed by `"` as in C/C++/Java: `"Hi, this is a string"`, `"um"`, `"ended by newline\n"`. Cyan strings and literal characters support the same escape characters as Java. A literal string may start with `n"` to disable any escape character inside the string:

```
var fileName = n"D:\User\Carol\My Texts\text01"
```

In this case `"\t"` do not mean the tab character. Of course, this kind of string cannot contain the character `'`.

Types `Byte`, `Short`, `Int`, `Long`, `Float`, and `Double` support almost the same set of arithmetical and logical operators as the corresponding types of Java. We show just the interface of `Int`. Types `Float` and `Double` do not support methods `&`, `|`, `~|`, and `!`. All basic types are automatically included in every Cyan source code because they belong to package `cyan.lang`.

```
package cyan.lang
```

```
// method bodies elided
```

```
final object Int
```

```
  fun ++
```

```
  fun --
```

```
  fun + (Int other) -> Int
```

```
  fun - (Int other) -> Int
```

```
  fun * (Int other) -> Int
```

```
  fun / (Int other) -> Int
```

```
  fun % (Int other) -> Int
```

```
  fun < (Int other) -> Boolean
```

```
  fun <= (Int other) -> Boolean
```

```
  fun > (Int other) -> Boolean
```

```
  fun >= (Int other) -> Boolean
```

```
  fun == (Int other) -> Boolean
```

```
  fun != (Int other) -> Boolean
```

```
  fun === (Int other) -> Boolean
```

```

fun <=> (Int other) -> Int
fun .. (Int theEnd) -> Interval<Int>
fun ..< (Int theEnd) -> Interval<Int>
fun - -> Int
fun + -> Int
    // and bit to bit
fun & (Int other) -> Int
    // or bit to bit
fun | (Int other) -> Int
    // exclusive or bit to bit
fun ~| (Int other) -> Int
    // binary not
fun ~ -> Int
    // left shift. The same as << in Java
fun <.< (Int other) -> Int
    // right shift. The same as >> in Java
fun >.> (Int other) -> Int
    // right shift. The same as >>> in Java
fun >.>> (Int other) -> Int
fun cast: (Any other) -> Int
fun asByte -> Byte
fun asShort -> Short
fun asLong -> Long
fun asFloat -> Float
fun asDouble -> Double
fun asChar -> Char
fun asBoolean -> Boolean
fun asString -> String
fun to: (Int max) do: (Function<Nil> b)
fun to: (Int max) do: (Function<Int, Nil> b)
fun to: (Int max) do: (InjectObject<Int> injectTo)
fun repeat: Function<Nil> b
fun repeat: Function<Int, Nil> b
fun to: (Int max)
fun in: (Iterable<Int> container) -> Boolean
fun in: Interval<Int> inter -> Boolean
end

```

...

```

abstract object InjectObject<T> extends Function<Nil>
    abstract fun eval: T
    abstract fun result -> T
end

```

```

interface Iterable<T>
    fun foreach: Function<T, Nil>

```

```

    fun apply: (String message)
    fun .* (String message)
    fun .+ (String message) -> Any
end

```

Variables of types `Byte`, `Char`, `Short`, `Int`, and `Long` may be preceded by `++` or `--`. When `v` is a private instance variable or a local variable, the compiler will replace `++v` by

```
(v = v + 1)
```

Idem for `--`. When `v` is a public or protected instance variable, `++v` is replaced by

```
(v: (v + 1))
```

If `v` is public, `++v` can only occur inside the prototype in which `v` is declared. If it is protected, `++v` can only appear in sub-prototypes of the prototype in which it is declared.

A prototype may declare an `operator []` and use it just like an array (see Section 4.9). A variable whose type support both “`at: []`” and “`at: [] put:`” methods can be used with `++`. Then `++v[expr]` is replaced by

```

    // tmp1 and tmp2 are temporary variables
var tmp1 = expr;
var tmp2 = v[tmp1] + 1;
v[tmp1] = tmp2;

```

Each basic prototype `T` but `Float` and `Double` has an `in:` method that accepts an object that implements `Iterable<T>` as parameter. This call method `foreach:` of this parameter comparing each element with `self`. It returns `true` if there is an element equal to `self`. It can be used as in

```

var Char ch;
ch = In readChar;
( ch in: {# 'a', 'e', 'i', 'o', 'u' #} ) ifTrue: {
    Out println: "#ch is a vowel"
};
var Array<Int> intArray = {# 0, 1, 2, 3 #};
var List<Int> intList = List<Int> new;
intList add: 0;
intList add: 1;
var Int n = In readInt;
if n in: intArray || n in: intList {
    Out println: "#n is already in the lists"
}

```

The parameter to `in:` can be any object that implements `Iterable` of the correct type. In particular, all arrays whose elements are of a basic type implement this interface.

Each basic prototype `T` but `Float` and `Double` has an `in:` method that accepts an interval as parameter:

```

var Char ch;
ch = In readChar;
( ch in: 'a'..'z' ) ifTrue: {
    Out println: "#ch is a lower case letter"
};
var age = In readInt;
if age in: 0..2 { Out println: "baby" }
else if age in: 3..12 {

```

```

    Out println: "child"
}
else if age in: 13..19 {
    Out println: "teenager"
}
else {
    Out println: "adult"
}

```

Prototype Boolean has the logical operators && (and), || (or), and ! (not). Every method that starts with ! is a prefixed unary method.

```

if ! ok { Out println: "fail" }
if age < 0 || age > 127 { Out println: "out of limits" }
if index < array size && array[index] == x {
    Out println: "found #{x}"
}

```

In the last statement, there is a problem: the argument to && will be evaluated even if “index < array size” is false, causing the runtime error “array index out of bounds”. To prevent this error, the expression on the right of && should be put in a function.

```

if ! ok { Out println: "fail" }
// no need of a function here
if age < 0 || { ^ age > 127 } { Out println: "out of limits" }
if index < array size && { ^array[index] == x } {
    Out println: "found #{x}"
}

```

In the Boolean prototype, there are methods && and || that take a function as parameter. These methods implement short-circuit evaluation.

```
package cyan.lang
```

```

public final object Boolean
  fun && (Boolean other) -> Boolean
    // short-circuit evaluation
  fun && (Function<Boolean> other) -> Boolean
  fun || (Boolean other) -> Boolean
  fun || (Function<Boolean> other) -> Boolean
  fun ! -> Boolean
  fun < (Boolean other) -> Boolean
  fun <= (Boolean other) -> Boolean
  fun > (Boolean other) -> Boolean
  fun >= (Boolean other) -> Boolean
  fun == (Boolean other) -> Boolean
  fun != (Boolean other) -> Boolean
  fun - (Boolean other) -> Int
  fun ++
  fun --
  fun cast: (Any other) -> Boolean
  fun asInt -> Int

```

```

fun asString -> String
fun to: (Boolean max) do: (Function<Nil> b)
fun to: (Boolean max) do: (Function<Boolean, Nil> b)
fun ifTrue: (Function<Nil> trueBlock)
fun ifFalse: (Function<Nil> falseBlock)
fun ifTrue: (Function<Nil> trueBlock) ifFalse: (Function<Nil> falseBlock)
fun ifFalse: (Function<Nil> falseBlock) ifTrue: (Function<Nil> trueBlock)
@checkTF
fun t: (Any trueValue) f: (Any falseValue) -> Any
@checkTF
fun f: (Any falseValue) t: (Any trueValue) -> Any
end

```

Prototype Char has the usual methods expected for a character.

```

package cyan.lang

public final object Char
  fun ++
  fun --
  fun < (Char other) -> Boolean
  fun <= (Char other) -> Boolean
  fun > (Char other) -> Boolean
  fun >= (Char other) -> Boolean
  fun == (Char other) -> Boolean
  fun != (Char other) -> Boolean
  fun === (Char other) -> Boolean
  fun <=> (Char other) -> Int
  fun .. (Char theEnd) -> Interval<Char>
  fun ..< (Char theEnd) -> Interval<Char>
  fun pred -> Char
  fun suc -> Char
  fun - (Char other) -> Int
  fun cast: (Any other) -> Char
  fun asByte -> Byte
  fun asInt -> Int
  fun asShort -> Short
  fun asLong -> Long
  fun asBoolean -> Boolean
  fun asString -> String
  fun to: (Char max) do: (Function<Nil> b)
  fun to: (Char max) do: (Function<Char, Nil> b)
  fun in: (Iterable<Char> container) -> Boolean
  fun in: Interval<Char> inter -> Boolean
end

```

Prototype String support the concatenation method + and the in: method:

```

fun daysMonth: (String month, Int year) -> Int {
  if month in: {# "jan", "mar", "may", "jul", "aug", "oct", "dec" #} {

```



```

||
~||
&&
== <= < > >= != === <=> ~=
.. ..<
+ -
/ * %
| ~| &
<.< >.> >.>>
.* .+ .%
::
+ - ++ -- ! ~ (unary)

```

Figure 3.1: Precedence order from the lower (top) to the higher (bottom)

```

    return 31
}
else if month in: {# "apr", "jun", "sep", "nov" #} {
    return 30
}
else if month == "fev" {
    return Int cast: ((leapYear: year) t: 29 f: 28)
}
else {
    return -1
}
}

```

Other methods from this prototype will be defined in due time. Maybe the `String` class of Java will be used as the `String` prototype of Cyan.

3.6 Operator and Selector Precedence

Cyan has special precedence rules for methods whose names are the symbols given in Figure 3.1. The meaning of these methods is given in the declaration of the basic types that use them (see page 51). The precedence is applied to every message send that uses some of these symbols. So a message send

```

x + 1 < y + 2

```

will be considered as if it was

```

(x + 1) < (y + 2)

```

Then when we write

```

if age < 0 || age > 127 { Out println: "out of limits" }
if index < array size && array[index] == x {
    Out println: "found #{x}"
}

```

the compiler interprets this as

```

if (age < 0) || (age > 127) { Out println: "out of limits" }
if (index < array size) && (array[index] == x) {

```

```

    Out println: "found #{x}"
}

```

In a message send, unary selectors have precedence over multiple selectors. Then

```

obj a: array size
is the same as
obj a: (array size)

```

Every operator but `+`, `-`, `++`, `--`, `*`, `/`, `%`, `~`, `!`, `..`, and `..<` should be preceded and followed by a white space. That is, all binary operators but the arithmetical ones (`+`, `-`, `*`, `/`, `%`) should be surrounded by white spaces. Note that not all operators are used by the Cyan basic types (`.*`, for example).

Unary methods associate from left to right. Then

```

var String name = club members first name;

```

is the same as:

```

var String name = ((club members) first) name;

```

The method names of the last line of the Figure 3.1 are unary. All other methods are binary and left associative. That means a code

```

ok = i >= 0 && i < size && v[i] == x;

```

is interpreted as

```

(ok = i >= 0 && i < size) && v[i] == x;

```

This is true even when `Boolean` is not the type of the receiver.

The compiler does not check the type of the receiver in order to discover how many parameters each selector should use. When the compiler finds something like

```

obj s1: 1 s2: 1, 2 s3: 1, 2, 3

```

it considers that the method name is `s1:s2:s3` and that `si` takes `i` parameters. This conclusion is taken without consulting the type of `obj`. Therefore, code

```

// get: takes two parameters
var k = matrix get: (anArray at: 0), 1;

```

cannot be written

```

var k = matrix get: anArray at: 0, 1;

```

This would mean that the method to be called is named `get:at:` and that `get:` receives one parameter, `anArray`, and `at:` receives two arguments, 0 and 1. To know the reason of this rule, see Chapter 6.

3.7 Loops, Ifs, and other Statements

Currently each statement or local variable declaration should end with a semicolon (`;`). However we expect to make the semicolon optional as soon as possible.

Decision and loop statements are not really necessary in Cyan. As in Smalltalk, they can be implemented as message sends to `Boolean` objects and to function objects. There are four methods of prototype `Boolean` used as decision statements: `ifTrue:`, `ifFalse:`, `ifTrue:ifFalse:`, and `ifFalse:ifTrue:`.

```

( n%2 == 0 ) ifTrue: { s = "even" };
( n%2 != 0 ) ifFalse: { s = "even" };
( n%2 == 0 ) ifTrue: { s = "even" } ifFalse: { s = "odd" } ;
( n%2 != 0 ) ifFalse: { s = "even" } ifTrue: { s = "odd" } ;

```

They are self explanatory. Besides that, there are methods in `Boolean` that return an expression or another according to the receiver:

```
s = String cast: (( n%2 == 0 ) t: "even" f: "odd");
s = String cast: (( n%2 != 0 ) f: "even" t: "odd");
```

If the expression is true, the expression that is parameter to `t:` is returned. Otherwise it is returned the parameter to `f:`. A metaobject (Chapter 5) `checkTF` checks whether the arguments of both selectors have the same type¹ (both are strings in this case). However, the return value type of this method is `Any` and therefore a cast is needed.

Function objects that return a `Boolean` value have a `whileTrue:` and a `whileFalse:` methods.

```
var i = 0;
{^ i < 5 } whileTrue: {
    Out println: i;
    ++i
}
var i = 0;
{^ i >= 5 } whileFalse: {
    Out println: i;
    ++i
}
```

Of course, `whileTrue` calls the function passed as parameter while the function that receives the message is true. `whileFalse` calls while the receiver is false.

The `if` and the `while` statements were added to the language to make programming easier. The syntax of these statements is shown in this example:

```
if n%2 == 0 {
    s = "even"
}
else { // the else part is optional
    s = "odd"
};
var i = 0;
while i < 5 {
    Out println: i;
    ++i
}
```

Cascaded `if`'s are possible:

```
if age < 3 {
    s = "baby"
}
else if age <= 12 {
    s = "child"
}
else if age <= 19 {
    s = "teenager"
}
```

¹For the time being, one cannot be subtype of another.

```

else {
    s = "adult"
};

```

Unlike the languages of the C family, the parentheses around the boolean expression are not necessary. There are other kinds of loop statements, which are supplied as message sends:

```

i = 0;
// the function is called forever, it never stops
{
    ++i;
    Out println: i
} loop;

```

```

var i = 0;
{
    if v[i] = x {
        return i;
    }
    ++i
} repeatUntil {^ i >= size};
// the function is called till i >= size

```

Prototype Int also defines some methods that act like loop statements:

```

// this code prints numbers 0 1 2
var i = 0;
3 repeat: {
    Out println: i;
    ++i
};
// this code prints numbers 0 1 2
3 repeat: { (: Int j :)
    Out println: j
};
var aFunction = { (: Int j :) Out println: j };
// this code prints numbers 0 1 2
3 repeat: aFunction;

// prints 0 1 2
i = 0;
1 to: 3 do: {
    Out println: i;
    ++i
};
// prints 0 1 2
0 to: 2 do: { (: Int j :)
    Out println: j
};

```

Prototype Char also has equivalent `repeat:` and `to:do:` methods:

```
'a' to: 'z' do: { (: Char ch :)
  Out println: ch
};
```

Prototype Any, the super-prototype of every object, defines a grammar method (see Chapter 9) that can be used as a C-like switch statement:

```
var n = In readInt;
if n >= 0 && n <= 6 {
  n
  case: 0 do: {
    Out println: "zero"
  }
  case: 1 do: {
    Out println: "one"
  }
  case: 2, 3, 5 do: {
    Out println: "prime"
  }
  else: {
    Out println: "four or six"
  }
};
var Int command;
var String strCmd = In readLine;
strCmd
  case: "on" do: { command = 1 }
  case: "off" do: { command = 2 }
  case: "left" do: { command = 3 }
  case: "right" do: { command = 4 }
  case: "move" do: { command = 5 }
  else: { command = 1 }
```

A metaobject attached to this method checks whether the expressions after `case:` have the same type as the receiver. The function after `do:` is of type `Function<Nil>` (no parameters or return value). This grammar method uses method `isCase:` of case value to find the correct `do:` function to call. Therefore this “`case:do:`” method works even with non-basic type objects. Prototype Any, the super-prototype of everyone, defines `isCase:` to be equal to `==`.

The `isCase:` method makes `case:do:` very flexible as can be seen by the examples below.

```
var str = In readLine;
str
  case: /[A-Z]+0/ do: {
    "upper case and zero" println
  }
  case: "true" do: {
    "true" println
  }
  case: {# "one", "two" #} do: {
    "1 or 2" println
  }
```

```

};

var n = In readInt;
n
  case: 0..2 do: {
    "baby" println
  }
  case: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 do: {
    "child" println;
  }
  case: 13..19 do: {
    "teenager" println
  }
  else: {
    "adult" println;
  };

```

In the first `case:do:` we used a literal regular expression `/[A-Z]+0/` which should have a method `isCase:` that takes a `String` as parameter and returns `true` if the string matches the regular expression. In the same way, arrays such as `{# "one", "two" #}` should define an `isCase:` method that take a string as parameter and returns `true` if the string belongs to the array.

3.8 Arrays

Array is a generic prototype that cannot be inherited for sake of efficiency. It has methods that mirror those of class `ArrayList` of Java:

```

package cyan.lang

final object Array<T> implements Iterable<T>
  fun init {
  fun init: (Int intSizeArray) {
  fun add: (T elem) {
  fun add: (Int i, T elem) {
  fun clear {
  fun isEmpty -> Boolean {
  fun remove: (Int i) {
  fun [] at: Int index -> T {
  fun [] at: Int index put: (T elem) {
  fun slice: (Interval<Int> interval) -> Array<T> {
  fun concat: Array<T> other -> Array<T> {
  fun size -> Int {
  fun foreach: Function<T, Nil> b {
  fun apply: (CySymbol message) select: (CySymbol slot) {
  fun apply: (String message) -> Dyn {
  fun .* (String message) {
  fun .+ (String message) -> Any {
end

```

Arrays supports some interesting methods: `apply:`, `.*`, and `.+`. The first one applies an operation given as string to all array elements:

```
var Array<Int> v = {# 2, 3, 5, 7, 11 #};
v apply: #print; // print all array elements
v .* #print; // print all array elements
(v .+ "+") print; // print the sum of all array elements
(v .+ "*") print; // print the multiplication of all array elements
```

Intervals can be arguments to `at:` which allows the slicing of arrays:

```
var letters = {# 'b', 'a', 'e', 'i', 'o', 'u', 'c', 'd' #};
var vowels = letters slice: 1..5;
// print a e i o u
Out println: vowels;
```

There is a prototype `RawArray` which corresponds to the usual array of C/C++/Java.

```
package cyan.lang

object RawArray<T>
  fun init: (Int intSizeArray)
  fun at: -> T { Int index }
  fun at: { Int index }put: (T elem)
  fun size -> Int
end
```

Chapter 4

Objects

A prototype may declare zero or more *slots*, which can be variables, called instance variables, methods, called instance methods, or constants. In Figure 4.1, there is one instance variable, `name`, and two methods, `getName` and `setName`. Keywords `public`, `private`, or `protected` should precede each slot declaration. A public slot can be accessed anywhere the prototype can. A private one can only be used inside the prototype declaration. Protected slots can be accessed in the prototype, its sub-prototypes, sub-sub-prototypes, and so on. A sub-prototype inherits from a prototype — that will soon be explained.

In the declaration of an instance variable, there are four optional parts:

1. the visibility (`public`, `protected`, `private`);
2. keyword “`var`”, that may precede the type;
3. “`;`”, that may follow the declaration;
4. and “ `= expr`”, that may follow the variable name.

The only non-optional parts are the type and the name.

A prototype declaration is a literal object that exists since the start of the program execution. There is no need to create a clone of it in order to use its slots.

Public or protected instance variables are allowed. In this case, the compiler creates public or protected get and set methods for a hidden variable that is only accessed, in the source code, by these methods. If the source code declares a public instance variable `instvar` of type `T`, the compiler eliminates this declaration and declares:

- (a) a private instance variable `_instvar` (it is always underscore followed by the original name);
- (b) methods

```
public T instvar { return _instvar }  
public instvar: (: _newInstvar T) { _instvar = _newInstVar }
```

Methods `instvar` and `instvar:` are different. The compiler does not change the user source code. It only changes the abstract syntax tree it uses internally.

Information on the slots can be accessed through the Introspective Reflection Library (IRL, yet to be made). The IRL allows one to retrieve, for example, the slot names. The IRL will inform you that a prototype with a public instance variable `instvar` has a private instance variable “`_instvar`” and public methods `instvar` and `instvar:.` The same applies to protected variables, which are also accessed through methods, as the public variables. In the declaration of a public instance variable, to it can be


```

package bank

object Client
  fun getName -> String {
    ^ self.name
  }
  fun setName: String name {
    self.name = name
  }
  fun print {
    Out println: name
  }
  private String name = ""
end

```

Figure 4.1: An object in Cyan

assigned an expression `expr`. In this case, this expression is assigned to the private instance variable `_instvar`.

Inside the prototype, `instvar` should always be accessed through methods `instvar` and `instvar:` since there is no variable `instvar` and `_instvar` is inaccessible:

Prototype `Client` could have been declared as

```

package bank

object Client
  public String name = ""
  fun print {
    Out println: name
  }
end

```

The instance variable should be used as in:

```

Client name: "Anna";
Out println: (Client name);
// compilation error in the lines below
Client.name = "Maria";
Out println: Client.name;

```

In future versions of Cyan it may be possible to access `name` as in

```
Client.name = "Maria";
```

To allow that, it would be necessary a syntax for grouping the get and set methods associated to a public variable. Currently this syntax is unnecessary. To replace a public instance variable by methods it is only necessary to delete the variable declaration and replace it by methods. It is expected that the compiler helps the user in converting assignments like the above into

```
Client name: "Maria";
```

A method or prototype declared without a qualifier is considered `public`. An instance variable without a qualifier is considered `private`. Then, a declaration

```

package Bank

object Account
  fun set: Client client {
    self.client = client
  }
  fun print {
    Out println: (client getName)
  }
  Client client
end

```

is equivalent to

```

package Bank

public object Account
  fun set: Client client {
    self.client = client
  }
  fun print {
    Out println: (client getName)
  }
  private Client client
end

```

The declaration of local variables is made with the following syntax:

```

var String name;
var Int x1, y1, x2, y2;

```

The last line declares four variables of type `Int`. Keyword `var` is demanded in the declaration of local variables.

`var` may be used before the declaration of an instance variable:

```

object Person
  var String name
  public var Int age
  ...
end

```

However, its use is optional.

The scope of a local variable is from where it was declared to the end of the function in which it was declared:

```

1  fun p: Int x {
2    var String iLiveHere;
3    if x > 0 {
4      var Int iLiveInsideThenPart;
5      doSomething: {
6        var String iLiveOnlyInThisFunction;
7        ...
8      }

```

```

9         ...
10     }
11 }

```

Then `iLiveHere` is accessible from line 2 to line 11 (before the `}`). Variable `iLiveInsideThenPart` is live from line 4 to 10 (before the `}`). The scope of `iLiveOnlyInThisFunction` is the function that in between lines 6 and 8 (after the declaration and before the `}`).

Although the scope of a local variable is limited, no two variables or parameters can have the same name in a method. That includes parameters of anonymous functions.

The type of a variable should be a prototype or an interface (explained later). In the declaration

```
var String name;
```

prototype `String` plays the rôle of a type. Then a prototype name can play two rôles: objects and types. If it appear in an expression, it is an object, as `String` in:

```
anObj = String;
```

If it appears as the type of a variable or return value type of a method, it is a type. Here “variable” means local variable, parameter, or instance variable.

A local variable or an instance variable can be declared and assigned a value:

```
private var Int n = 0;
```

Both the type and the assigned value can be omitted, but not at the same time. If the type is omitted, it is deduced from the expression at compile-time. If the expression is omitted, a default value for each type is assigned to the variable. Therefore a variable always receive a value in its declarations. We call this “definition of a variable” (instead of just “declaration”). When the type is omitted, the syntax

```
var variableName = expr
```

should be used to define the variable as in:

```
private var n = 0;
```

Variable `variableName` cannot be used inside `expr`. It it could, the compiler would not be able to deduce the type of `expr` in some situations such as

```
var n = n;
```

In an assignment `var n = expr`, the type of the expression is deduced by the compiler using information collected in the previous lines of code. The Hindley-Milner inference algorithm is not used. In particular, the type of parameters and return value of methods are always demanded unless you are using some kind of dynamic typing in Cyan (see Chapter 6 for details).

The default value assigned to a variable depends on its type and is given by the table:

type	default value
Byte	0Byte
Short	0Short
Int	0Int
Long	0Long
Float	0Float
Double	0Double
Char	'\0'
Boolean	false
String	""
others	the prototype itself

Any type other than the basic types or `String` has the prototype itself as the default value. All

prototypes, including the basic types, are objects in Cyan. Then `Int` is an object which happens to be an ... integer! And which integer is `Int`? It is the default value of type `Int`. So the code below will print 0 at the output:

```
Out println: Int;
```

However, it is clearer to have a method that returns the default value. To every prototype `P` that is not a basic type or `String` the compiler adds a method

```
fun defaultValue -> P { ^P }
```

if the user does not define this method herself. For basic types, the compiler returns the value of the above table.

A method is declared with keyword `fun` followed by the method selectors and parameters, as shown in Figure 4.1. Following Smalltalk, there are two kinds of methods in Cyan: unary and keyword methods.

A unary method does not take any parameters and may return a value. Its name may be an identifier followed optionally by a “:” (which is not usual and is not allowed in Smalltalk). For example, `print` in Figure 4.1 is a unary method.

When a method takes parameters its name should be followed by “:” (without spaces between the identifier and this symbol). For example,

```
fun set: Client client { ... }
```

An optional return value type can be given after keyword `fun`. The return value should be given by the `return` command or by an expression after “~” (which should be in the outer scope of the method — that will be seen later). The return expression should be subtype (Section 4.16) of the return value type of the method. Using `Nil` as the return value type is the same as to omit the return type.

Methods without return type or declaring `Nil` always return `Nil`. Therefore one can write

```
(0 println) println
```

“0 println” returns `Nil`. Message `println` is therefore sent to `Nil`. It will be printed
0Nil

Objects are used through methods and only through methods. A method is called when a message is sent to an object. A message has the same shape as a method declaration but with the parameters replaced by real arguments. Then method `setName:` of the example of Figure 4.1 is called by

```
Client setName: "John";
```

This statement causes method `setName:` of `Client` to be called at runtime.

There are two kinds of literal strings in Cyan: one is equal to those of C/C++/Java, “Hello world”, “n = 0\n”, etc. This form allows one to put escape characters in the string. The other kind of literal string is using `n` in the start of the string. This form disables any escape characters:

```
var fileName = n"c:\texts\readyToPrint\nightPoem.doc"
```

This means that the string is really

```
c:\texts\readyToPrint\nightPoem.doc
```

This kind of string is not really in the language. It is in fact a metaobject call to a metaobject of package `cyan.lang` which is included in every Cyan source file.

Object `CySymbol` inherits from `String` and it is the prototype of all literal Cyan symbols. There are two kinds of literal symbols. The first one is `#` followed, without spaces, by letters, digits, and any number of ‘s, as in

```
#f #age #age:  
#123 #_0 #field001  
#foreach:do:
```

The second kind of literal symbol starts with `#` and ends with `"` and obey the same restrictions as regular literal strings:

```
#"Hello world - spaces are allowed"
#"valid: & \n this was a escape character"
#"1 + 2"
```

Method `eq:` of `CySymbol` returns true if the argument and `self` have the same contents. Method `eq:` of `String` tests whether the objects are the same:

```
var s = "I am s";
var p = s;
assert: ( #name eq: #name );
    // strings and symbols are of different prototypes
assert: ( #name neq: "name" );
    // s and p refer to the same object
assert: (s == p) && (s eq: p);
    // s is not equal to "I am s" because they are different objects
    // although they have the same contents
assert: !(s == "I am s");
```

Both `String` and `CySymbol` are final prototypes. This last prototype is the only prototype allowed to extend the first. Weird but necessary.

4.1 Constants

A constant object can be defined inside an object using keyword `const`:

```
object Date
    public const Int daysWeek = 7
    public const Int daysMonth = {# 31, 28, 30, 31, 30, 31, 31, 30, 31, 30, 31 #}
    public Int day, month, year
end
```

After `const` there should appear the type, constant name, `"="`, and a value assignable to variables of the given type. A value of type `S` is assignable to a variable of type `T` if `S is T` or `S` is a direct or indirect sub-prototype of `T`.

The constant can be public, private, or protected and its type can be anyone. It should be initialized at the declaration. The expression that initializes a constant is evaluated right before the prototype is created, before the program execution. The constants are created in the textual order in which they are declared:

```
object MyConstants
    public const A first = A new
        // second is created after first
    public const Int second = (B new: 100) add: 5
end
```

The access to a constant is made as if it were a unary method:

```
var Int numberOfWeeks = (Date daysMonth)/(Date daysWeek);
```

The more usual syntax, `"Date.daysMonth"` is not supported because it would be ambiguous: it could mean "the prototype `daysMonth` from package `Date`".

4.2 self

Inside a method of a prototype, pseudo-variable `self` can be used to refer to the object that received the message that caused the execution of the method. This is the same concept as `self` of Smalltalk and `this` of C++/Java. An instance variable `age` can be accessed in a method of a prototype by its name or by the name preceded by “`self.`” as in

```
fun getAge -> Int {  
  ^ self.age  
}
```

Then we could have used just “`age`” in place of “`self.age`”.

4.3 clone Methods

A copy of an object is made with the `clone` method. Every prototype `P` has a method

```
fun clone -> P
```

that returns a *shallow* copy of the current object. In the shallow copy of the original to the cloned object, every instance variable of the original object is assigned to the corresponding variable of the cloned object.

In the message send

```
Client setName: "John";
```

method `setName` of `Client` is called. Inside this method, any references to `self` is a reference to the object that received the message, `Client`. In the last statement of

```
var Client c;  
c = Client clone;  
c setName: "Peter";
```

method `setName` declared in `Client` is called because `c` refer to a `Client` object (a copy of the original `Client` object, the prototype). Now the reference to `self` inside `setName` refers to the object referenced to by `c`, which is different from `Client`.

The `clone` method of an object can be redefined to provide a more meaningful clone operation. For example, this method can be redefined to return `self` in an `Earth` prototype (since there is just one earth) or to make a deep copy of the `self` object.

In language Omega [Bla94], the pseudo-type `Same` means the type of `self`, which may vary at runtime. Method `clone` declared in the `Object` prototype returns a value of type `Same`. That means that in object `Object`, the value returned is of type `Object` and that in a prototype `P` the return value type of `clone` is `P`. In Cyan the compiler adds a new `clone` method for every prototype `P`. This is necessary because there is nothing similar to `Same` in the language.

4.4 Shared Variables

A prototype may declare a variable as `shared`, as in

```
object Date  
  public Int day, month, year  
  public shared Date today  
end
```

Variable `today` is shared among all `Date` objects. The `clone` message does not duplicate shared variables. By that reason, we do not call shared variables “instance” variables.

4.5 new, init, and initOnce Methods

It is possible to declare two or more methods with the same name if they have parameters with different types. This concept, called overloading, will soon be explained. A prototype may declare one or more methods named `init` or `init:`. All of them have special meaning: they are used for initializing the object. For each method named `init` the compiler adds to the prototype a method named `new` with the same selectors and parameter types. Each `new` method creates an object without initializing any of its slots and calls the corresponding `init` method. If the prototype does not define any `init` or `init:` method, the compiler supplies an empty `init` method that does not take parameters.

Some rules apply to the `init` and `init:` methods. They:

- (a) should either be declared with `Nil` as the return type or with no return type (the default is `Nil`);
- (b) should be public (this may change in the future). The compiler changes their visibility to `protected`, although only direct descendents can call them;
- (c) should not be preceded by keyword `override`;
- (d) should not be abstract or `final`;
- (e) should not be grammar methods (see Chapter 9);
- (f) should not be indexing methods (See Section 4.9);
- (g) can only be called inside the method in which they are declared or in immediate sub-prototypes. That is, if `C` inherits from `B` that inherits from `A`, then `C` cannot call the `init` or `init:` methods of `A`. To call the `init` or `init:` method of the prototype, use “`init`”, “`init: args`”, “`self init`”, or “`self init: args`”. To call these methods of the immediate super-prototype, use “`super init`”, and “`super init: args`”.

It is legal to declare methods with the name `new` or `new:`. However, these methods should be public and have the prototype as the return type.

```
object Test
  fun new -> Test { return Test }
  fun new: Int newValue -> Test {
    var t Test = new;
    t value: newValue;
    return t
  }
  // this is illegal: return type is not Test
  fun new: Float newValue -> Int {
    return Int cast: newValue
  }
  public Int value
end
```

It is illegal to declare an `init` method with the same signature¹ as a user-defined `new` method:

¹Signature will be defined later. For now, assume that is composed by the method name, parameter types, and return value types.

```

object Test
  // public is default
  @prototypeCallOnly
  fun new: Int k -> Test { ... }
  // legal
  fun init: String s { ... }
  // illegal for there is a new
  // method with the same parameters
  fun init: Int n { ... }
end

```

`init` and `init:` methods can only be called by `init` and `init:` methods of the prototype in which they were declared or in `init` methods of direct sub-prototypes of the prototype (the concept of sub-prototype, inheritance, will soon be explained). To explain that, suppose a prototype A is inherited by prototype B that is inherited by C. Then a `init` method of C cannot call a `init` method of A, which is not a direct super-prototype of C. But a `init` method of B may call a `init` method of A. Although all `init` and `init:` methods are “public”, the compiler changes all of them to “protected” so they can only be called in message sends to `self`.

Methods `new` and `new:` are only accessible through prototype objects. That means an object returned by `new` or `new:` cannot be used to create new objects of that prototype using “`new:`” or “`new`”:

```

object Test
  @prototypeCallOnly
  fun new: Int k -> Test { ... }
  // legal
  fun init: String s { ... }
end

```

```

object Program
  fun run {
    var t = Test clone;
    var Test u;
    // Ok !
    u = t clone;
    // compile-time error
    u = t new: 100;
    // compile-time error
    u = t init: "Hi";
    // ok
    var any = "just a test";
    u = Test new: any;
  }
end

```

The last two lines of method `run` exemplify the use of dynamic dispatch with `new:` methods. Prototype `Test` has two `new:` methods, one of them is user-defined and the other is created by the compiler from the `init:` method. Message send “`Test new: any`” will cause a method search at runtime for an adequate `new:` method. Method created from `init:` will be chosen. The important thing here is that the choice of the method to be called is made at runtime. This is the regular Cyan mechanism for method dispatching, which is not that nice when applied to `new:` constructors (it is slow). But it is probably worse to create

a search mechanism specific to constructors.

To summarize, `new` and `new:` methods follow the rules:

- (a) their return type should be the prototype in which they are declared;
- (b) they should be public (this may change in the future);
- (c) they should not be preceded by keyword `override`, `abstract`, or `final`;
- (d) they should not be grammar methods (see Chapter 9);
- (e) they should not be indexing methods;
- (f) they can be called only by sending a message to the prototype.

Every prototype `A` have a private method called `primitiveNew` that creates a new copy of it, just like the unary `new`:

```
private fun primitiveNew -> A
```

Unlike `new`, no initialization is made on the object. This method is added by the compiler and it cannot be redefined by the user.

Since `primitiveNew` is private, it can only be called by a message send to `self`. This method can be used, for example, to count how many objects were created:

```
public object University
  @prototypeCallOnly
  fun new -> University {
    // an easy way of creating an University: in code
    ++universityCounter;
    return primitiveNew
  }
  ...
  shared Int universityCounter = 0;
end
```

A prototype may declare a single method called `initOnce` without parameters or return value that will be called once in the beginning of the program execution. Or maybe this method will be called when the prototype is loaded into memory (this is yet to be defined). Method `initOnce` should be used to initialize shared variables or even the instance variables of the prototype. This method should be private. Therefore it cannot be called outside the object. It will rarely be called inside the prototype since it is automatically called once.

```
public object Lexer
  ...
  private fun initOnce {
    keywordsTable add: "public";
    keywordsTable add: "private";
    keywordsTable add: "object";
    ...
  }
  shared Set<String> keywordsTable
end
```

Metaobject `@init` automatically creates two methods: one that returns nothing and initializes instance variables and a `new` method. Consider a prototype `Proto` that declares instance variables `p1`, `p2`, ..., `pn` of types `T1`, `T2`, ..., `Tn`. Then a metaobject call

```
@init(p1, p2, ..., pn)
```

can be put anywhere a slot declaration may appear inside the `Proto` declaration. When the compiler finds this metaobject call, it will add the two following methods to the prototype, if they were not declared by the user.

```
fun v1: (T1 p1) v2: (T2 p2) ... vn: (Tn pn) {
  v1 = p1;
  v2 = p2;
  ...
  vn = pn;
}
```

If `vi` is a public or protected instance variable, `vi = pi` is replaced by `vi: pi` as expected.

```
@prototypeCallOnly
fun new: (T1 p1), (T2 p2), ... (Tn pn) -> Proto {
  var Proto p = self primitiveNew;
  // initialize variable vi with pi
  ...
  return p;
}
```

So, a prototype

```
object University
  @init(name, location)
  public String name
  public Int age
end
```

can be used as

```
var p = Person new;
p name: "Carol" age: 1;
var peter = Person new: "Peter", 3;
p age: 1 name: "Carol; // compile time error
```

One can use just `@init`, without parameters, to create to two methods above for all of the instance variables of a prototype. The order of the variables in both method is the textually declared order in the prototype. Of course, if a new instance variable is added to the prototype or the declaration order is changed an error will be introduced in the code. The compiler should warn the user that the changes made are dangerous. The information that the previous version of the prototype has a different order or a different number of instance variables is available in the XML file which contains the source code.

There is an abbreviation for calling methods called `new` or `new:` of a prototype. Expressions

```
P new
P new: a
P new: a, b, c
```

can be replaced by

```
P()
P(a)
P(a, b, c)
```

Using prototypes `Test` and `Person` we can write

```
var t1 = Test(0);
var Test t2 = Test("Hello");
var Person p = Person("Mary", 1);
var q = Person("Francisco", 5);
```

However, in this text we will usually employ method `new` or `new:` for object creation.

Using the short form for object creation, we can easily create a net of objects. In this example, `BinTree` inherits from `Tree` (Section 4.11).

```
object Tree
end

object BinTree extends Tree
  @init(left, value, right)
  public Tree left, right
  public Int value
end

object No extends Tree
  @init(value)
  public Int value
end
...

var tree = BinTree( No(-1), 0, BinTree(No(1), 2, No(3)) );
```

Cyan does not restrict the statements in an `init` or `new` method. That will soon change. An instance variable of a prototype `Proto` whose type is not an union of the type `Union<Nil, T>` cannot hold the `Nil` value. When an object of `Proto` is created, its initial value is given by method `defaultValue` of its type. However, this is not a meaningful value. The variable should be initialized with a better value before used. The compiler should demand that either the instance variable has type `Union<Nil, T>` or it is initialized in all `init` and `new` methods of the prototype in which it is declared. That will be demanded as soon as possible.

4.6 Order of Initialization

A prototype may have assignments of expressions to instance variables, shared variables, constants, and methods. Besides that, the `initOnce` method is called once to initialize instance or shared variables and expressions is compile-time metaobjects should be evaluated.

When a prototype is loaded into memory (or when it is created at the beginning of the program execution), the runtime system does some initializations. These correspond to assignments to constants, shared variables, and instance variables (it is legal to assign a value to each of them). For each assignment, the right-hand side is evaluated and assigned to the left-hand side exactly in the order just given. Inside each one of the group “constants”, “shared variables”, and so on, the assignments are made in textual order. After these initializations, `initOnce` is called.

```

object Test
  public const Int one = 1
  shared Int three = two + 1
  Int five = four + 1
  Int six = five + 1
  Int seven
  shared Int four = three + 1
  Int eight
  public const Int two = one + 1
  private fun initOnce {
    seven = 7;
    eight = 8;
  }
end

```

This example shows the order of initialization: it is the order given by the variable names. Then `six` is initialized before `two`, for example.

Every time a new object of the prototype is created, with `new`, `new:`, or `clone`, the expressions assigned to instance variables are evaluated and assigned again.

4.7 Keyword Methods and Selectors

The example below shows the declaration of a method. The method body is given between `and` and `.`

```

fun withdraw: Int amount -> Boolean { // start of method body
  (total - amount >= 0) ifTrue: {
    total = total - amount;
    return true
  }
  ifFalse: {
    return false
  }
} // end of method body

```

A function is a sequence of statements delimited by `{` and `}`. In the code above, there are three functions: the method body, one after `ifTrue:`, and another after `ifFalse:`. Functions become full closures at runtime and were inspired in Smalltalk blocks. However, the Cyan anonymous functions are statically typed. The syntax for declaring the body of a method between `and` and `.` came from language Omega. Based on this syntax we thought in considering methods as objects (to be seen later).

Command `return` returns the method value and, unlike Smalltalk, its use is demanded. The execution of the function is ended by the `return` command. Note that the method itself is a function which has inside other functions. It is legal to use nested functions. Symbol `^` returns the value of a function. However, it does not necessarily cause the method in which the function is to finish its execution. See page 181 for a more detailed explanation.

Method `withdraw` takes an argument `amount` of type `Int` and returns a boolean value (of type `Boolean`). It uses an instance variable `total` and sends message

```
ifTrue: { .. } ifFalse: { ... }
```

to the boolean value `total - amount >= 0`. The message has two function arguments,

```
{ total = total - amount; return true }
```

and

```
{ return false }
```

A message like this is called a *keyword message* and is similar to Smalltalk keyword messages. As another example, an object `Rectangle` can be initialized by

```
Rectangle width: 100 height: 50
```

This object should have been defined as

```
object Rectangle
  fun width: Int w height: Int h {
    self.w = w;
    self.h = h;
  }
  fun set: (Int x, Int y) { self.x = x; self.y = y; }
  fun getX -> Int { ^ x }
  fun getY -> Int { ^ y }
  Int w, h // width and height
  Int x, y // position of the lower-left corner
  ...
end
```

Each identifier followed by a “:” is called a *selector*. So `width:` and `height:` are the selectors of the first method of `Rectangle`. Sometimes we will use “method with multiple selectors” instead of “keyword method”.

The signature of a method is composed by its selectors, parameter types, and return value type. Then the signature of method “width:height:” is

```
width: Int height: Int
```

The signature of `getX` is

```
Int getX
```

It is important to note that there should be no space before “:” in a selector. Then the following code is illegal:

```
(i > 0) ifTrue : { r = 1 } ifFalse : { r = 0 }
```

And so are the declaration

```
fun width : Int w height : Int h {
```

To make the declaration of a keyword method clear, parenthesis can be used to delimit the parameters that appear after a selector:

```
object Rectangle
  fun width: (Int w) height: (Int h) {
    self.w = w;
    self.h = h;
  }
  fun set: (Int x, y) {
    self.x = x; self.y = y;
  }
  ...
end
```

Parameters are read-only. They cannot appear in the right-hand side of an assignment.

4.8 On Names and Scope

Methods and instance variables of an object should have different names. A local variable declared in a method should have a name different from all variables of that method. So, the declaration of the following method is illegal.

```
fun doAnything: Int x, Int y {
  var newY = -y; // equivalent to "var Int newY = -y;"
  (x < 0) ifTrue: {
    var Int newX = -x;
    (y < 0) ifTrue: {
      var Int newY = -y; // error: redeclaration of newY
      rotate newX, newY;
    }
  }
}
```

However, instance variables and shared variables can have names equal to local variables (which includes parameters):

```
fun setName: String name {
  self.name = name
}
```

An object can declare methods “value” and “value:” as in the following example:

```
object Store
  private Int _value = 0
  public fun value -> Int { ^ _value }
  public fun value: Int newValue {
    self._value = newValue
  }
end
```

```
object Program
  fun run {
    var s = Store clone;
    var Int a;
    a = In readInt;
    s value: a;
    Out println: (s value);
  }
end
```

Usually we will not use get and set methods. Instead, we will use the names of the attributes as the method names as in

```
var Fish fish = Fish new;
fish name: "Cardinal tetra";
fish lifespan: 3;
Out println: "name: ", (fish name), " lives up to: ", (fish lifespan);
```

Fish could have been declared as

```
object Fish
  String _name;
  Int _lifespan;
  fun name -> String { ^_name }
  // parameter with the same name as instance variable
  fun name: _name String { self._name = _name }
  fun lifespan -> String { ^_lifespan }
  fun lifespan: Int _lifespan { self._lifespan = _lifespan }
end
```

4.9 Operator []

It is possible to define operator [] for indexing:

```
object Table
  fun [] at: Int index -> String {
    return anArray[index]
  }
  fun [] at: Int index put: String value {
    anArray[index] = value
  }
  Array<String> anArray
end
...
```

```
var t = Table new;
t[0] = "One";
t[1] = "Two";
// prints "One Two"
Out println: t[0], " ", t[1];
```

This operator can only be used with methods `at:` and `at:put:`. Each selector should have only one parameter. The “at:” parameter should be between `and`, without parentheses. One or both methods can be declared. But when both are declared, the type of selector `at:` should be the same. The allowed signatures of these methods are:

```
U at: T
at: T put: W
U at: T put: W
```

Only one of the last two signatures may be used. Usually, $U = W$. But these types can be different from each other.

4.10 Method Overloading

There may be methods with the same selectors but with different number of parameters and parameter types (method overloading). For example, one can declare

```

object MyPanel
  fun draw: Square f { ... }
  fun draw: Triangle f { ... }
  fun draw: Circle f { ... }
  fun draw: Shape f { ... }
  String name
end

```

There are four draw methods that are considered different by the compiler. In a message send

```
MyPanel draw: fig
```

the runtime system searches for a draw method in prototype MyPanel in the textual order in which the methods were declared. It first checks whether fig references an object which is a subtype of Square (See Section 4.16 for a definition of subtype). If it is, this method is called. If it is not, the searches continues in the second method,

```
draw: Triangle f
```

and so on. If an adequate method is not found in this prototype, the search would continue in the super-prototype. In this case, that will never happens: the compiler will assure that a method will be found at runtime. After all, the language is statically-typed.

Method overloading is also possible when there is more than one selector:

```

object FullIndexable
  fun init: Int size { v = Array<String> new: size }
  fun at: Int i -> String { ^v[i] }
  fun at: CySymbol s -> String { ^v[ Int cast: s ] }
  fun at: String s -> String { ^v[ Int cast: s ] }

  fun at: Int i put: String value -> String {
    ^v[i] = value
  }
  fun at: CySymbol s put: String value -> String {
    ^v[ Int cast: s ] = value
  }
  fun at: String s put: String value -> String {
    ^v[ Int cast: s ] = value
  }
  Array<String> v
end

```

This object could be used as in

```

var f = FullIndexable new: 10;
f at: 0 put: "zero";
f at: #1 put: "one";
f at: "2" put: "two";
Out println: (f at: "0"); // prints "zero"
Out println: (f at: 1); // prints "one"
Out println: (f at: #2); // prints "two"

```

The name of a method is the concatenation of all of its selectors. So method

```
fun at: Int i put: String value -> String
```


has name “at:put:”. Methods of the same prototype with the same name should have the same return value type. Therefore the compiler would sign an error in the code

```
object Point
  fun dist: Int nx, Int ny -> Int { ... }
  fun dist: Float nx, Float ny -> Float { // compilation error here
    ...
  }
  ...
  Int x, y
end
```

There are two more restrictions on the use of overloaded methods:

- (a) all methods with the same name should appear in sequence. Then the only element allowed between two declarations of methods with the same name is another method with this same name.

```
object FullIndexable
  fun at: Int i -> String { ^v[i] }
    // init should not be here !
  fun init: Int size { v = Array<String> new: size }
    // compilation error
  fun at: CySymbol s -> String { ^v[ Int cast: s ] }
  ...
end
```

- (b) all overloaded methods should have the same qualifier (public, protected, or private).

4.11 Inheritance

A prototype may extends another one using the syntax:

```
object Student extends Person ... end
```

This is called inheritance. **Student** inherits all methods and variables defined in **Person**. **Student** is called a sub-object or sub-prototype. **Person** is the super-object or super-prototype. Every instance variable of the sub-object should have a name different from the names of the public methods of the super-object (including the inherited ones) and different from the names of the methods and other instance variables of the sub-object. Since the name of a non-unary method includes the “:”, there may be instance variable `iv` and method `iv:`.

A method of the sub-object may use the same selectors as a method of the super-object (their names may be equal). There is no restriction on the parameter types used in the sub-object method. However, the return value type of the sub-object method should be a subtype of the return value type of the super-object method. That is, Cyan supports co-variant return value type. This does not cause any runtime type errors, which is justified using the following example.

```
var Super v = anObject;
var A x;
x = v selector: 0;
```

The compiler checks whether the return value of method `selector:` of **Super** (which may be inherited from super-objects) is a subtype of the declared type of `x`, **A**. The code is only run if this is true. Then at runtime `v` may refer to a sub-object of **Super** referenced by `anObject`. This sub-object may declare

a `selector`: method whose return value type is `C`, a subtype of the return value type `B` of method `selector`: of `Super`. That will not cause a runtime type error because subtyping is transitive: `C` is subtype of `B` which is a subtype of `A`. Therefore, `C` is also a subtype of `A`. At runtime there will be an assignment of an object of `C` to a variable of type `A`, which is type-correct.

A public or protected method of the sub-object that overrides a super-object method (same selectors) should be declared with the word `override` following the qualifier (`public` or `protected`). See the examples.

```
object Animal
  fun eat: Food food { Out println: "eating food" }
end

object Cow extends Animal
  public override fun eat: Grass food { Out println: "eating grass" }
end

object Person
  fun print {
    Out println: "name: ", name, " (" , age, ")"
  }
  public String name
  public Int age
end

object Student extends Person
  public override fun print {
    super print;
    Out println: " School: ", school
  }
  public String school
end
```

If `override` precedes a method definition, all methods of the prototype with the same name (selectors) should be preceded by `override` too. A method can only be overridden by a method with the same visibility. That is, a public method can only override a public method.

There is a keyword called `super` used to call methods of the super-object. In the above example, method `print` of `Student` calls method `print` of prototype `Person` and then proceeds to print its own data.

Methods `init`, `init:`, `new`, `new:`, and `initOnce` are never inherited. However, `init` or `init:` methods of a sub-object may call `init` or `init:` methods of the super-object using `super:`

```
object Person
  fun init: String name , Int age {
    self.name = name;
    self.age = age;
  }
  fun print {
    Out println: "name: ", name, " (" , age, ")"
  }
  public String name
```

```

    public Int age
end

object Student extends Person
  fun init: String name, Int age , String school {
    super init: name, age;
    self.school = school
  }
  override fun print {
    super print;
    Out println: " School: ", school
  }
  fun nonsense {
    // compile-time error in this line
    // new: cannot be called
    var aPerson = super new: "noname", 0;
    // ok, clone is inherited
    var johnDoe = super clone;
  }
  public String school
end

```

Keyword `override` is not necessary in the declaration of method `init:` of `Student` because `init:` of `Person` is not inherited. The compiler adds to prototype `Person` a method

```

    Person new: String name, Int age
and to Student

```

```

    Student new: String name, Int age, String school

```

Since methods `clone` and `new` are not inherited, there will be compile-time errors in method `nonsense`.

A prototype may be declared as “final”, which means that it cannot be inherited:

```

public final object String
  ...
end

```

There would be a compile-time error if some prototype inherits `String`. The prototypes `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Char`, `Boolean`, and `String` are all final.

A public or protected method declared as “final” cannot be redefined in sub-prototypes:

```

public object Car
  final fun name: String newName { _name = newName }
  final String fun name { ^_name }
  String _name
  ...
end

```

Final methods should be declared in non-final prototypes (why?). Final methods allow some optimizations. The message send of the code below is in fact a call to method `name` of `Car` since this method cannot be overridden in sub-prototypes. Therefore this is a static call, much faster than a regular call.

```

var Car myCar;
...
s = myCar name;

```

Public instance variables can be declared final. That means the get and set methods associated to this variable are final.

The name of a method is the concatenation of its selectors, without considering its parameters and return type. Either all methods with the same name are final or none of them are.

```
object Bag
  final fun add: (String name) { ... }
  fun add: (Int name) { ... } // error
  ...
end
```

This is because of polymorphism. In a message send

```
b add: obj
```

in which the type of `b` is `Bag`, any of the two methods may be called if the type of `obj` is `Any`. The compiler would be unable to do any optimizations in this case.

The table below summarizes the allowed combination among keywords in a method declaration. Keyword `abstract` will be seen in Section 4.14.

	public	protected	private	override	abstract	final
public				Y	Y	S
protected				Y	Y	Y
private						
override	Y	Y			Y	Y
abstract	Y	Y		Y		
final	Y	Y				

Table 4.1: Keyword combination in method declaration

4.12 Multi-Methods

The mechanism of method overloading of Cyan implements a restricted form of multi-methods. In most languages, the receiver of a message determines the method to be called at runtime when the message is sent. In CLOS [Sei12], all parameters of the message are taken into consideration (which includes what would be the “receiver”). This is called multiple dispatch and the methods are called “multi-methods”. Cyan implements a restricted version of multi-methods: the method to be called is chosen based on the receiver and also on the runtime type of the parameters. To make the mechanism clearer, study the example below. Assume that `Grass`, `FishMeat`, and `Plant` are prototypes that inherit from prototype `Food`.

```
package main
...

private object Animal
  fun eat: Food food { Out println: "eating food" }
end

private object Cow extends Animal
  override fun eat: Grass food { Out println: "eating grass" }
```

```

end

private object Fish extends Animal
  override fun eat: FishMeat food { Out println: "eating fish meat" }
  override fun eat: Plant food { Out println: "eating plants" }
end

public object Program
  fun run {
    var Animal animal;
    var Fool food;
    animal = Cow;
    animal eat: Grass; // prints "eating grass"
    animal eat: Food; // prints "eating food"
    // the next two message sends prints the same as above
    // the static type of the parameter does not matter
    food = Grass;
    animal eat: food; // prints "eating grass"
    food = Food;
    animal eat: food; // prints "eating food"

    animal = Fish;
    animal eat: FishMeat; // prints "eating fish meat"
    animal eat: Plant; // prints "eating plants"
    animal eat: Food; // prints "eating food"
    // the next two message sends prints the same as above
    // the static type of the parameter does not matter
    food = FishMeat;
    animal eat: food; // prints "eating fish meat"
    food = Plant;
    animal eat: food; // prints "eating plants"
    food = Food;
    animal eat: food; // prints "eating food"
  }
end

```

4.13 Nil and Any, the Super-prototype of Everybody

Nil is a prototype outside the type hierarchy. It is not supertype or subtype of any other prototype. Therefore a variable whose type is Nil can only be assigned the value Nil. And Nil can only be assigned to a variable whose type is Nil. But when using dynamic typing this rule should not be obeyed. As shown in Section 6, Nil is also compatible with type Dyn. Any expression can be assigned to a variable whose type is Dyn and an expression whose type is Dyn can be assigned to any variable. That is, Dyn is supertype and subtype of anything, including Nil. See the example.

```

var Nil myEmptyness;
myEmptyness = Nil; // ok

```

```

var String s;
s = myEmptyness; // compile-time error
s = Nil; // compile-time error
myEmptyness = s; // compile-time error
Dyn myDyn = Nil; // ok

```

The declaration of Nil is given below. This prototype defines some basic methods that are not really necessary but were included to make Nil and Any have a common interface. Then there will never be an error if a message println is sent to a Dyn expression.

```
package cyan.lang
```

```
object Nil
```

```

final fun prototypeName -> String = "Nil"

fun asString -> String = "Nil"

fun asString: (Int ident) -> String {
  var String s = "";
  1..ident foreach: { (: Char ch :)
    s = s + " "
  };
  return s + "Nil"
}

fun print { Out println: "Nil" }

fun println { print; Out println: "\n" }

```

```
end
```

Prototypes that are declared without explicitly extending a super-prototype in fact extend an object called Any. Therefore Any is the super-prototype of every other object but Nil. It defines some methods common to all objects such as asString, which converts the object data to a format adequate to printing. For example,

```

Rectangle width: 100 height: 50
Rectangle set 0, 0;
Out println: (Rectangle asString);

```

would print something like

```

Rectangle {
  w: 100
  h: 50
  x: 0
  y: 0
}

```

Method asString: Int n also converts its receiver to a String. However, it does that with an indentation of n white spaces. The indentation is made with defaultIdentNumber white spaces. This is a constant declared in Any.

The methods declared in Any are given below. The method bodies are elided.

```
package cyan.lang

@prototypeCallOnly("cast:", "init", "init:", "new", "new:")
public object Any

  final fun eq: (Nil|Any other) -> Boolean {
  final fun neq: (Nil|Any other) -> Boolean { ^ ! (eq: other) }
  fun cast: (Any other) -> Any { ^other }
  fun prototype -> Any { ^self }
  final fun prototypeName -> String {
  final fun parent -> Any { @javacode<<<< return parent(); >>>> }
  final fun isInterface -> Boolean {
  @checkIsA
  final fun isA: (Any proto) -> Boolean {
  @checkThrow
  final fun throw: (CyException e) {
  fun hashCode -> Int {
  @typedClearly
  fun clone -> Any { ^self }
  private fun primitiveNew -> Any { ^Any }
  fun asString -> String { asString: 0 }
  fun asString: (Int ident) -> String {
  fun == (Nil|Any other) -> Boolean { @javacode<<< return this == _other >>> }
  fun isCase: (Any other) -> Boolean { ^self == other }
  fun === (Nil|Any other) -> Boolean { return self == other }
  fun != (Nil|Any other) -> Boolean { @javacode<<< return this != _other >>> }
  fun assert: (Boolean expr) {
  fun print { Out println: (self asString) }
  fun println { print; Out println: "\n" }
  fun defaultValue -> Any { ^ Any }
  @checkAttachMixin
  fun attachMixin: (Any mixProto) { }
  fun popMixin -> Boolean { }
  fun featureList -> Array<NTuple<key, String, value, Any>> {
  fun featureList: (String slotName) -> Array<NTuple<key, String, value, Any>> {
  fun slotFeatureList -> Array<NTuple<slotName, String, key, String, value, Any>> {
  fun annotList -> Array<Any> {
  fun annotList: (String slotName) -> Array<Any> {
  fun (selector: String (param: (Any)+)? )+ t -> Nil|Any { }
  fun (invokeMethod: selector: String (param: (Any)+)? )+ t -> Nil|Any { }
  @checkAddMethod
  fun (addMethod:
    (selector: String ( param: (Any)+ )?)
    )+
    (returnType: Any)?
    body: Any) t
  fun doesNotUnderstand: (CySymbol methodName, Array<Any> args)
```

```

@checkSwitch
fun (
    (case: (Any)+ do: Function<Nil>)+
    (else: Function<Nil>)?
) t
@checkGetMethod
final fun getMethod: String signature -> Any
@checkSetMethod
final fun setMethod: (String signature, Any method)
public const Int defaultIdentNumber = 4
end

```

Methods `prototype` and `primitiveNew` are added to the compiler to every prototype — the cannot be user defined. The compiler adds method `defaultValue` to every prototype that does not define this method. The same is true for the methods `clone`, `new`, `new:`, `attachMixin:`, and `popMixin`. If a prototype has one of these methods defined by the user, the compiler checks whether it has the correct method signature (parameter types and return value type). Of course, methods `new` and `new:` are only added if the prototype defines the correspondent `init` and `init:` methods.

Note that some of the `Any` methods are final and therefore they cannot be user-defined. As an example, `prototypeName` is final.

Method `eq:` returns `true` if `self` and the parameter reference the same object, `false` otherwise.

For every user-declared prototype `P` the compiler adds a method “`P cast: Any`” if the prototype does not define this method itself. A metaobject attached to this method issues an error if the receiver of a message `cast:` is not a prototype.

```

var A a;
var Proto p;
p = A cast: Person;
p = a cast: Person; // compile-time error

```

A prototype may redefine method `cast:` to take more appropriate actions:

```

object PolarPoint
@prototypeCallOnly
fun cast: Any other -> PolarPoint {
    if other is a regular point {
        convert other to a polar point p
        return p
    }
    else {
        // test whether other prototype is PolarPoint or
        // a sub-prototype of PolarPoint. If it is,
        // return other. Otherwise throw an exception
    }
}
end

```

The `cast:` method of the basic types do the usual conversion between these types (the same conversion Java does). The `cast:` method of a prototype `P` tests whether its parameter prototype inherits from `P` or if it is `P` itself. In these cases, the parameter itself is returned. Otherwise an exception `ExceptionCast` is thrown. User-defined method `cast:` should be public, non-abstract, non-final. It is not necessary to

attach metaobject `prototypeCallOnly` since the one attached to method `cast:` of `Any` will assure that message `cast:` can only be send to prototypes.

Method “`prototype`” returns the prototype that was used to create the object or the prototype itself (if it is the receiver):

```
var Person p = Person clone;
var w = Worker new;
assert: Person prototype == Person &&
       p prototype == Person &&
       w prototype == Worker &&
       w prototype != Person;
```

To every user-declared prototype `P` the compiler adds method

```
fun prototype -> P {
  return P
}
```

This method cannot be user-defined. Note that the return type of this method is a sub-prototype of the return type of the method of `Any`. This is legal: the return type of a method can be subtype of the type of the method of the same name defined in the super-prototype. `prototypeName` returns the name of the original prototype. It would be “`Person`” for prototype `Person` and “`Hashtable<String, Int>`” for an instantiation of generic object `Hashtable`.

Method `parent` returns the parent prototype of the receiver. If the receiver is not a prototype, it returns the parent of the receiver’s prototype:

```
var Person p = Person name: "fulano";
assert: (p parent == Person parent);
```

Method `parent` of an interface always return `AnyInterface` even if the interface inherits from several other ones. Method `parent` of `AnyInterface` returns `Any`. Method `parent` of `Any` returns `-1`.

Method `isInterface` returns `true` if the receiver is an interface. It cannot be redefined.

Method `isA:` returns `true` if the prototype of `self` is the same as `proto` or a descendent of it. Parameter `proto` should be a prototype, which is checked by a metaobject `checkIsA`. Assuming that `Circle` inherits from `Elipse` that inherits from `Any`, we have

```
var Elipse e = Elipse;
var Circle c = Circle x: 100 y: 200 radius: 30;
assert: (c isA: Elipse && c isA: Circle);
assert: (c isA: Any && Circle isA: Any && Circle isA: Circle);
/* if uncommented the statement that follows would
   cause a compile-time error */
assert: (c isA: e);
```

Method `throw:` throws the exception that is the parameter. See more on Chapter 12.

`hashCode` returns an integer that is the hash code of the receiver object (this needs to be better defined).

`clone` returns a cloned copy of `self`. It is used shallow copy. Method `asString` returns a string with the content of `self`. It can and should be override to give a more faithful representation of the object. Method `==` returns the same as `eq:` by default. But it can and should be user-defined. In the basic types, it returns `true` if the values are equal. Method `!=` returns `true` if `==` returns `false` and vice-versa. Method `isCase:` is the same as `==` in `Any`. This method is used in the “`case:do: ...`” method for

comparing the value of the receiver of the message with each of the values of the “cases”. See more about this in page 60.

Method `assert`: takes a boolean expression as parameter and throws exception `ExceptionAssert` if `expr` is false. Method `print` prints information on the receives using methods `print`: and `println`: of prototype `Out`. Method `defaultValue` returns the default value for the prototype — see page 67. It is a special value for the basic types (for example, 0 for `Int`) and the prototype itself for all other prototypes.

Method `attachMixin` attaches a mixin object to the current object. For each user-defined prototype `P`, the compiler adds a method

```
@checkAttachMixin
fun attachMixin: Any mixProto { ... }
```

If the prototype does not define itself this method. Metaobject `checkAttachMixin` checks whether `mixProto` is a mixin object that can be attached to objects of `P` and its sub-prototypes. This method is described in page 108. `popMixin` removes the last mixin object dynamically attached to the receiver. It returns `true` if there was a mixin attached to the object and `false` otherwise. A prototype may have `attachMixin`: and `popMixin` defined by the user or none of them. It is illegal a prototype to have just one of these methods defined by the user.

Section 5.1 explains in detail the `Any` methods that deal with features and annotations. Here we just comment briefly these methods.

Method `featureList` returns an array with all features of the prototype. Method

```
fun featureList: (String slotName) -> Array<NTuple<key, String, value, Any>>
```

returns the feature list of slot `slotName`, which may be the name of an instance variable, method, or constant declared in the prototype. The method name includes the types of the parameters but not the return value type. If the prototype is dynamically typed, type `Any` should be used for each parameter without a type.

Annotations are a special case of *features*. An attachment

```
@annot( #root )
```

is the same as

```
@feature("annot", #root)
```

Method `annotList` returns a list of annotation objects attached to the prototype. Method

```
fun annotList: (String slotName) -> Array<Any>
```

returns the annotation list of slot `slotName`.

Method

```
fun (selector: String (param: (Any)+)? )+ -> Nil|Any
```

is a grammar method (described in Chapter 9) used to call a method by its name. For example, suppose an object `Map` has a method

```
key: String value: Int
```

This method can be called using the grammar method `selector`: ... in the following way:

```
Map selector: "key:" param: "One"
  selector: "value:" param: 1;
```

This grammar method checks whether the object has the method at runtime (of course!). Then this example is equivalent to

```
Map ?key: "One" ?value: 1;
```

Note that there may be one or more parts “selector: ...”. Selector “param:” is optional. At least one parameter should follow selector “param:” if it is present. The value returned by this method is the object returned by the called method, which may be Nil if the return type is Nil.

Conceptually, a message send to object `obj` causes the execution of method

```
fun (invokeMethod: selector: String (param: (Any)+)? )+ t -> Nil|Any
```

of `obj`. That means the regular search for methods is used when searching for this method (in the prototype of `obj`, the super-prototype of the prototype, and so on).

Method `invokeMethod: ...` is then responsible for calling the method associated to the message. For example, suppose `Worker` inherits from `Person` and both define a `print` method.

```
var Worker w;
w = company getOldestWorker;
w print;
```

In the last message send, first method `invokeMethod: ...` of `Any` is called (assume that this method is not overridden in sub-prototypes). Then `invokeMethod: ...` of `Any` does a search for an appropriate method starting in `Worker`. Since a `print` method is found there, it is called.

Then, conceptually, usually Cyan does two searches for each method call:

- (a) one for finding an `invokeMethod: ...` method;
- (b) the other, inside this method of `Any`, to find the appropriate method. It is this call that is made in almost all object-oriented languages.

There are two occasions in which things happens a little different from described above:

- (a) when `invokeMethod: ...` calls itself, as in

```
var any = Any;
any invokeMethod: selector: "invokeMethod:"
```

There is an infinity loop;

- (b) when `invokeMethod: ...` is redefined in a prototype. In this case, the redefined method is responsible for calling the appropriate method.

Of course, we used the name `invokeMethod: ...` because of Groovy.

Method

```
fun (addMethod:
    (selector: String ( param: (Any)+ )?
    )+
    (returnType: Any)?
    body: Any)
```

adds a method dynamically to an object. It is explained in page 204 of Section 10.11.

Method `doesNotUnderstand:` is called whenever a message is sent to the object and it does not have an appropriate method for that message. The message name (as a symbol) and the arguments are passed as arguments to `doesNotUnderstand:..` This method ends the program with an error message. The name of a message is the concatenation of its selectors. The name of message

```
ht at: i put: obj with: #first
is “at:put:with:”.
```

Since Cyan is statically typed, regular message sends will never cause the runtime error “method not found”. But that can occur with dynamic message sends such as

```
s ?push: 10;
or
s selector: #push param: 10
```

The argument to method `doesNotUnderstand:` is an array that has an element for each selector of the message. Each selector may have zero or more real parameters. If a selector has two or more real parameters, these are packed in another array. Then if method `format:print:to:` does not exist in object `x`, the call

```
x format: "%d%s%i" print: n, name, age to: output
will cause method doesNotUnderstand: to be called with parameter
#{ Any cast: "%d%s%i", #{Any cast:n, name, age}#, output }#
```

There is always a cast to `Any` in the first element of any array. This assures that the array is of type `Array<Any>`.

Method `case:do:` ... implements the “switch” statement and is discussed elsewhere (Section 3.7).

Methods `getMethod:` and `setMethod:` get and set a method of an object. They are discussed in Section ??.

```
@checkGetMethod
final fun getMethod: String signature -> Any
@checkSetMethod
final fun setMethod: (String signature, Any method)
```

There are several missing methods in `Any` related to reflective introspection. These reflective introspection methods will be added to `Any` during the design of the Metaobject-Protocol for Cyan.

4.14 Abstract Prototypes

Abstract prototypes in Cyan are the counterpart of abstract classes of class-based object-oriented languages. It is a compile-time error to send a message to an abstract prototype, which includes messages `new` and `new:`. Since these methods can only be called through a prototype, no objects will ever be created from an abstract prototype. `init` and `init:` methods may be declared — they may be called by sub-prototypes.

The syntax for declaring an abstract object is

```
public abstract object Shape
  fun init: Int newColor { color: newColor }
  public abstract fun draw
  fun color -> Int { ^ shapeColor }
  fun color: Int newColor { shapeColor = newColor }
  Int shapeColor
end
```

An abstract method is declared by putting keyword “`abstract`” before “`fun`” and it can only be declared in an abstract object, which may also have non-abstract methods and instance variables. A sub-prototype of an abstract object may be declared abstract or not. However, if it does not define the inherited abstract methods, it must be declared as abstract.

The name of a method is the concatenation of its selectors, without considering its parameters and return type. Either all methods with the same name are abstract or none of them are.

```
abstract object Printable
  abstract fun print: Int
```

```
    fun print: String // error: should be abstract
end
```

Objects are concrete things. It seems weird to call a concrete thing “abstract”. However, this is not worse than to call an abstract thing “abstract”. Classes are abstraction of objects and there are “abstract classes”, an abstraction of an abstraction.

Since all prototypes are concrete things in Cyan, the compiler adds a body to every abstract method to throw an exception `ExceptionCannotCallAbstractMethod`:

```
    fun draw {
        throw: ExceptionCannotCallAbstractMethod("Shape::draw")
    }
```

Note that

```
    Shape draw
causes a compile-time error. And
```

```
var Shape s = Shape;
s draw;
```

causes a *runtime* error. The method `draw` added to the compiler is called. It is legal to assign an abstract object to a variable. To prohibit that would say that not all “objects” are really objects in Cyan. We could not pass `Shape` as parameter, for example. That would be bad.

Keyword “`override`” is optional when used with the “`abstract`” keyword:

```
public abstract object Shape
    abstract fun draw
    ...
end
```

```
abstract object Polygon extends Shape
    abstract fun draw
end
```

We could have used

```
    override abstract fun draw
```

4.15 Interfaces

Cyan supports *interfaces*, a concept similar to Java interfaces. The declaration of an interface lists zero or more method signatures as in

```
interface Printable
    fun print
end
```

The `public` keyword is not necessary since all signatures are public. `fun` is not necessary but it is demanded for sake of clarity (should it be eliminated too?).

An interface has two uses:

- (a) it can be used as the type of variables, parameters, and return values;

- (b) a prototype can *implement* an interface. In this case, the prototype should implement the methods described by the signature of the interface. A prototype can implement any number of interfaces. Name collision in interface implementation is not a problem.

Interfaces are similar to the concept of the same name of Java.

As an example, one can write

```
interface Printable
  fun printObj
end

object Person
  public String name
  public Int age
end

object Worker extends Person implements Printable
  String company
  fun printObj {
    Out println: "name: " + name + " company: " company
  }
  ... // elided
end
```

Here prototype `Worker` should implement method `printObj`. Otherwise the compiler would sign an error. Interface `Printable` can be used as the type of a variable, parameter, and return value:

```
var Printable p;
p = Worker clone;
p print;
```

An interface may extend any number of interfaces:

```
interface ColorPrintable extends Printable, Savable
  fun setColor: Int newColor
  fun colorPrint
end
```

An interface is a prototype that may inherit from any number of other interfaces. Therefore Cyan supports a limited form of multiple inheritance. An interface that does not explicitly inherits from any other in fact inherits from prototype `AnyInterface` (which inherits from `Any`).

```
object AnyInterface
end
```

The method signatures declared in an interfaces are transformed into public methods. These methods throw exception `ExceptionCannotCallInterfaceMethod`:

```
// interface ColorPrintable as a prototype
object ColorPrintable extends Printable, Savable
  fun setColor: Int newColor {
    throw: ExceptionCannotCallInterfaceMethod("ColorPrintable::setColor");
  }
  fun colorPrint {
```

```

        throw: ExceptionCannotCallInterfaceMethod("ColorPrintable::colorPrint");
    }
end

```

Interfaces are then objects with full rights: they be assigned to variables, passed as parameters, and receive messages.

Although interfaces are objects, the compiler puts some restrictions on their use and declaration.

- (a) An interface can only extend another interface. It is illegal for an interface to extend a non-interface prototype.
- (b) Interfaces cannot declare any `new`, `new:`, `init`, or `init:` methods. No object will never be created from them. But the interface itself may receive messages.
- (c) A regular prototype cannot inherit from an interface.
- (d) If the type of an expression is an interface `I`, then the compiler checks whether the messages sent to it match those method signatures declared in the interface, super-interfaces, and `Any` (See Section 4.13).

Besides that, method `isInterface` inherited from `Any` returns `true` when the receiver is an interface. The examples that follow should clarify these observations.

```

// ok
var Printable inter = Printable;
// ok, asString is inherited from Any
Out println: (inter asString);
// ok, Printable is a regular object
Out println: (Printable asString);
var Any any = Printable;
// ok
Out println: (any asString);
// it is ok to pass an interface as parameter
assert: (any isA: Printable);
assert: (any isInterface && Printable isInterface &&
        inter isInterface);

```

A note on implementation: for each interface `Inter` the compiler creates a regular prototype called `Proto_Inter`. `Inter` will produce a Java interface called `_Inter` and `Proto_Inter` will produce a Java class called `_Proto_Inter`. Whenever `Inter` is used as a type in the Cyan code, it will be replaced by `_Inter` in the Java code. But if it is used in an expression, the Java code uses `Proto_Inter`. Or better, it uses `_Proto_Inter.prototype`, which is the expression that refers to an object of `_Proto_Inter`.

Class `_Proto_Inter` implements `_Inter`. All methods defined in `Inter` have bodies that throw exception

```
ExceptionCannotCallInterfaceMethod
```

As an example, the code

```

var Shape sh; // Shape is an interface
sh = Shape;

```

will produce the following Java code:

```

_Shape _sh;
_sh = Proto_Shape.prototype;

```

4.16 Types and Subtypes

A type is a **prototype** (when used as the type of a variable or return value) or an interface. Subtypes are defined inductively. **S** is subtype of **T** if:

- (a) **S** extends **T** (in this case **S** and **T** are both prototypes or both interfaces);
- (b) **S** implements **T** (in this case **S** is a prototype and **T** is an interface);
- (c) **S** is a subtype of a type **U** and **U** is a subtype of **T**.

Then, in the fake example below, **I** is supertype of every other type, **J** is supertype of **I**, **J** and **D** are supertypes of **E**, and **B** is supertype of **C**, **D**, and **E**.

```
interface I end
interface J extends I end
object A implements I end
object B extends A end
object C extends B end
object D extends C implements J end
object E extends D end
```

Considering that the static type or compile-time type of **s** is **S** and the static type of **t** is **T**, the assignment “**t = s**” is legal if **S** is a subtype of **T**. Using the previous example, the following declarations and assignments are legal:

```
var I i;
var J j;
var A a;
var B b;
car D d;
var E e;
i = j; i = a; a = e; i = a;
j = d; b = d; j = e;
```

There is a predefined function `typeof` evaluated at compile-time that return the type of an variable, constant, or literal object. In the example

```
var Int x;
var typeof(x) y;
```

x and **y** have both the `Int` type.

4.17 Union Types

Cyan has a special generic prototype `Union` that takes at least two real parameters that should be prototypes (including `Nil`). An object of `Union` is a container for values of any of the real parameters. In an assignment in which the type of the left-hand side is

`Union<T1, T2, ..., Tn>`

and the type of the right-hand side is `Ti` for some *i* between 1 and *n*, the compiler generates code to create an object of the union and initializes it with the right-hand side expression. More specifically, if `W` is the type of the right-hand side expression, the generated code tests which `Ti` is a supertype of `W` from 1 to *n* (in this order). Then it calls a method of the union object with the right-hand side expression

keeping which T_i was found. At compile-time it is checked whether W can be a subtype of any of the types T_1, T_2, \dots, T_n . The type found at compile-time need not to be related to the runtime type found.

`Shape` is a super-prototype of `Circle`.

```
var Union<Circle, Shape> join;
var Shape s = Circle(10, 5, 3);
join = s;
```

For the last assignment the compiler generates the equivalent to:

```
join = Union<Circle, Shape> new;
if s isA: Circle {
    join f1: s
}
else {
    join f2: s
}
```

`f1` and `f2` are private methods of the `Union`. They are only accessible to the compiler. Since the compiler checks whether the right-hand side expression, which is `s`, is subtype of `Circle` or `Shape`, it is not necessary to test whether `s` is an object of `Shape` in the `else` part of the `if`.

The union object keeps which method was used to initialize it. This information is used to retrieve the object stored in the union, which is made with the `unionCase:do:` method.

```
var Union<Circle, Shape> join;
var Shape s = Circle(10, 5, 3);
join = s;
join
    unionCase: Circle do: {
        ("join is a Circle with radius " + (join radius) ) println
    }
    unionCase: Shape do: {
        "join is a Shape" println
    }
```

Inside the function passed as parameter to `do:` the type of `join` is the type passed as parameter to the previous `unionCase:.` Then in the first function the type of `join` is `Circle` and we can call method `radius`. This feature is implemented through a metaobject `checkUnionCase` and was taken from language Ceylon.

Of course, the assignment “`s = join`” produces a compile-time error because it is unsafe. There is no guarantee that `join` keeps an object of the type of `s`, which is `Shape`.

The `unionCase:do:` method takes parameters in the order they appear in the prototype `Union`. If the order is not respected they will be a compiling error. The compiler will not find the method.

It is illegal to declare an `Union` with two equal prototypes as `Union<Shape, Shape>`. But there is no restriction other than that. In particular, it is legal to declare a supertype before a subtype. However, in this case the subtype will never be used. When an assignment is made to a variable whose type is `Union`, the first type that fit is used (from left to right), not the best match.

```
var Union<Shape, Circle> join;
var Shape s = Circle(10, 5, 3);
// s is inserted as Shape, the first type that fit
join = s;
```

```

join // this union case is used
  unionCase: Shape do: {
    "join is a Shape" println
  }
  unionCase: Circle do: {
    ("join is a Circle with radius " + (join radius) ) println
  }

```

It is expected that the compiler issue a warning if a sub-prototype appears after its super-prototype in a generic prototype instantiation of Union.

An assignment in which the type of the left-hand side is Dyn and the right-hand side is Union<T1, ..., Tn> is transformed by the compiler into an assignment from the contents of the right-hand side to the left-hand side.

```

var Dyn d;
var Union<T1, T2, ..., Tn> myUnion;
d = myUnion;

```

The last assignment is transformed by the compiler into

```
d = myUnion getElem;
```

getElem is a private method that returns the object kept by the union.

In all places but in expressions an union Union<T1, T2, ..., Tn> may be replaced by

```
T1|T2|...|Tn
```

For example:

```

object Test
  fun print: Int | Char | String elem { ... }
  fun test {
    var Worker | Person p;
    ...
  }
  Manager | Director upperClass;
end

```

However, this syntax cannot be used in expressions because that would mean a message send with selector “|” to a prototype.

```
(Int | Char prototypeName) println;
```

Int | Char is the sending of message “|” to prototype Int with parameter Char. Prototype Int do have a method “|” but this takes another Int as parameter. There is a compile-time error. It is expected that that compiler detects that the programmer wanted to use Union<Int, Char>. It should offer to correct the error resulting in

```
(Union<Int, Char> prototypeName) println;
```

Method eq: declared in Any compares the receiver to the argument, returning true if they refer to the same object. When the receiver is an union, the comparison is made between the contained object with the argument.

```

var Person p = Person;
var Union<Person, Any> myUnion;

```

```
myUnion = p;
```

```

assert: (myUnion eq: p); // true
p = Person();
  // neq: is not eq:
assert: (myUnion neq: p); // true
var Int i = 0;
var Union<Int, Char> intChar;
intChar = 0;
assert: (i eq: intChar); // true

```

The first assertion is true because `myUnion` keeps the object referenced by `p`. The comparison

```
myUnion eq: p
```

is made between the object that `myUnion` contains and `p`. Since they are the same, the result is `true`. In the second assertion, a comparison is made between a reference to `Person` (contained by `myUnion`) and a reference to a newly created object (created by expression `Person()`).

Method `eq:` of the basic types (such as `Int`) compare for equal values, being equivalent to `==`. Therefore the last assertion is true.

Method `==` of the `Union` prototypes compare the contained object with the argument.

```

var String s = "Carol";
var Union<String, Any> myUnion;

myUnion = "Carol";
assert: (myUnion == s); // true
s = "Car" + "ol";
assert: (myUnion == s); // true
assert: (myUnion != "Carolina"); // true

var Union<Int, Char> intChar;
intChar = 0;
assert: (0 == intChar); // true

```

4.17.1 The Union Prototypes

If neither `A` or `B` is `Nil` and both are prototypes, prototype `Union<A, B>` is

```

package cyan.lang

final object Union<A, B>

  fun == (Any other) -> Boolean {
    return elem == other;
  }
  @checkUnionCase
  fun unionCase: A do: Function<Nil> Afunction
    unionCase: B do: Function<Nil> Bfunction {
      if which == #f1 {
        Afunction eval
      }
      else { // should be #f2
        Bfunction eval
      }
    }

```

```

    }
  }
private fun f1: (A other) { which = #f1; elem = other }
private fun f2: (B other) { which = #f2; elem = other }

// private methods go here
// the contained element
Any elem
// which element is kept by "elem"
CySymbol which
@javacode<*<
  @Override public boolean eq(Object other) {
    _Any another = (_Any ) getUnionElem();
    if ( another == null )
      return false;
    else
      return another._eq_dot(other);
  }

  @Override Object getUnionElem() { return _elem; }
*>
end

```

Method eq: is declared in Any and handles appropriately the union prototypes.

When one of the real parameters to the union is Nil, this prototype like this:

```

package cyan.lang

final object Union<Nil, B>

@checkIfNil
fun ifNil: (Any other) -> Any {
  if which == #f1 {
    return other
  }
  else {
    return elem
  }
}

fun == (Nil|Any other) -> Boolean {
  @javacode<*<
    if ( _which.s.equals("f1") ) {
      return _other == _Nil.prototype;
    }
    else {
      _Any another = (_Any ) getUnionElem();
      if ( another == null )
        return false;
      else

```

```

        return another._equal_equal(other);
    }
    >*>
    // this method is not implemented in Cyan
}
@checkUnionCase
fun unionCase: Nil do: (Function<Nil> Afunction)
  unionCase: B do: (Function<Nil> Bfunction) {
    if which == #f1 {
      Afunction eval
    }
    else { // should be #f2
      Bfunction eval
    }
  }
}
private fun f1: { which = #f1; }
private fun f2: (B other) { which = #f2; elem = other }

// private methods go here
// the contained element
Any elem
// which element is kept by "elem"
CySymbol which
@javacode<*>
@Override public boolean eq(Object other) {
  if ( _which.s.equals("f1") ) {
    return _other == _Nil.prototype;
  }
  else {
    _Any another = (_Any ) getUnionElem();
    if ( another == null )
      return false;
    else
      return another._eq_dot(other);
  }
}

@Override Object getUnionElem() { return _elem; }
>*>
end

```

There is an automatic conversion from an A expression to Union<A, B> in assignments of the kind
 Union<A, B> = A

However this does not mean that Union<A, B> is a supertype of A or B. Without the extra code added by the compiler this assignment would not be type-correct. The union prototype **does not** have the common methods of A and B.

Since methods eq:, neq:, ==, and === use the contained element of the union, an union behaves much like the contained element. This makes it easy to optimize unions. Objects of the unions need not to be created unless necessary. And they are only necessary when messages other than unionCase:do: are

sent to the union object. This will be rare. Most of the time the compiler can use the type `Any` as the type of the union. Only when necessary an object of the union can be created.

```
var Person|String ps;
ps = "Carol"; // may use Any as the type of s
var String s;
  // create a Union<Person, String> object here
(ps prototypeName) println;
```

Method `isA`: of all union prototypes return `true` if the receiver is a prototype of the union or any of the real parameters to the union. Then if `myUnion` is a variable of type

```
Union<T1, T2, ..., Tn>
```

all method calls below return `true`.

```
myUnion isA: Union<T1, T2, ..., Tn>
myUnion isA: T1
myUnion isA: T2
...
myUnion isA: Tn
```

The union prototypes can take a lower-case parameter before each prototype parameter. A prototype

```
Union<first, A, second, B>
```

causes the creation of the following prototype

```
package cyan.lang

final object Union<first, A, second, B>

  @checkUnionCase
  fun first: Function<Nil> Afunction
    second: Function<Nil> Bfunction {
      if which == #f1 {
        Afunction eval
      }
      else { // should be #f2
        Bfunction eval
      }
    }
  }
  fun first: ( A elem ) {
    which = #f1;
    self.elem = elem
  }
  fun second: ( B elem ) {
    which = #f2;
    self.elem = elem
  }

  // other methods as before
  // except that unionCase:do: is not used
end
```

Nil cannot be used as a prototype parameter. This kind of union can have repeated prototype parameters. Then

```
Union<wattsHour, Float, calorie, Float, joule, Float>
```

is legal. Because of that, there is no `unionCase:do:` method in this type of union prototype. And there is not automatic conversions in assignments. Methods with the names of the lower-case parameters to the generic prototype should be used to initialize the union object. This union object should be created before used, unlike the previous union objects that were automatically created.

```
// create the union object
var myUnion = Union<number, Int, numberStr, String> new;
```

```
myUnion = "12"; // compile-time error
myUnion number: 12; // ok
myUnion numberStr: "12"; // ok
// compiling error: no unionCase:do: method
myUnion
  unionCase: Int do: { ... }
  unionCase: String do: { ... };
```

```
myUnion
  number: {
    // myUnion is Int here
    (1 + 2*myUnion) println
  }
  numberStr: {
    // myUnion has type String here
    // string concatenation
    ("number is " + myUnion) println
  };
```

Let us see another example of this kind of union prototype.

```
var energy = Union<wattHour, Float, calorie, Float, joule, Float> new;
```

```
energy wattsHour: 314.15;
energy
  wattsHour: {
    "I had watt-hours" println
  }
  calorie: {
    "I had calories" println
  }
  joule: {
    "I had joules" println
  };
```

4.17.2 Operators that Use Unions

The language supports the Elvis operator, Nil-safe message sends, and Nil-safe array access. The Elvis operator is implemented as a method `ifNil:` (see page 99), Nil-safe message sends have all selectors

prefixed with `?.`, and Nil-safe array access is made with `?[` and `]?`. See the examples.

```
var String userName;
  // getUsername is a method name
var Nil|String gotUserName = UserDataBase getUsername;
userName = gotUserName ifNil: "anonymous";
```

The last line is the same as

```
userName = String cast: ((gotUserName == Nil) f: gotUserName t: "anonymous");
```

This is not exactly the Elvis operator of language Groovy. In Cyan both the receiver of message `ifNil:` and its parameters are evaluated.

Nil-safe message send:

```
var Nil|IndexedList<String> v;
  // it may associate Nil to v
v = obj getPeopleList;
v ?.at: 0 ?.put: "Gauss";
```

The last line is the same as

```
if v != Nil {
  v at: 0 put: "Gauss";
}
```

There should not be any space between the `?.` and the selector. And all selectors of a message should be preceded by `?.` in a Nil-safe message send.

Nil-safe array access:

```
var Nil|Array<Person> clubMembers;
...
var firstMember = clubMembers?[0]?;
```

The last line is the same as

```
var firstMember = Person cast: ((clubMembers != Nil) t: clubMembers[0] f: nil);
```

The result of `clubMembers?[0]?` when `clubMembers` is Nil is Nil too.

A code

```
if clubMembers != Nil {
  clubMembers[0] = "Newton"
}
```

is equivalent to

```
clubMembers?[0]? = "Newton";
```

We can use all features at the same time:

```
var Nil|Array<Nil|Person> clubMembers;
...
var String firstMemberName = (clubMembers?[0]? ?.name) ifNil: "no member";
```

4.18 Mixin Inheritance

An object can inherit from a single object but it can be mixed with any number of other objects through the keyword `mixin`. This is called *mixin inheritance* and it does not have the problems associated to multiple inheritance.


```
object B extends A mixin M1, M2, ... Mn
  // method and variable declaratins go here
end
```

M1, M2, ... Mn are mixin objects. A mixin object M is declared as

```
mixin(S) object M extends N
  // method and variable declarations of the mixin
end
```

Mixin M extends mixin N. This inheritance is optional, of course. Mixin N should be declared as

```
mixin(R) object N extends P
  // method and variable declarations of the mixin
end
```

Here R should be a subtype of S. The difference of a mixin from a normal object declaration is that a mixin object does not extends Any, the root of the object hierarchy, even if it does not extends explicitly any other object.

Mixin M may be inherited by prototype S, specified using “mixin(S)”, or by sub-prototypes of S, or to prototypes implementing interface S. Methods of the mixin object M may call methods of S using super.

Object or interface S may not appear in the declaration of a mixin object.

```
mixin object PrintMe
  fun whoIam {
    Out println: "I am #{super prototypeName}"
  }
end
```

In this case the mixin methods may call, through self or super, methods of prototype Any.

```
mixin object Empty
end
```

Since mixin Empty does not inherit from any other mixin, it does not have any methods or instance variables. This mixin could have been declared as

```
mixin(Any) object Empty
end
```

Currently a mixin object cannot be the type of a variable, parameter, or return value. Maybe this restriction will be lifted in the future. If M defines a method already defined in object S (assuming that S is not an interface), then the method declaration should be preceded by **override**, as is demanded when a sub-object override a super-object method.

The compiler creates some internal classes when it encounters an object that inherits from one or more mixin objects. Suppose object B extends object A and inherits from mixin objects M1, M2, ... Mn (as in one of the previous examples). The compiler creates an object B' with the body of B. B' defines a method with an empty body for each method declared in the mixin objects M1, ..., Mn. Then the compiler makes B' inherit from A. After this, it creates prototypes M1', M2', ... Mn' in such a way that

- (a) each Mi has the same body as Mi';
- (b) M1' inherits from B' and M(i+1) inherits from Mi;
- (c) Mn' is renamed to B.

Note that:

- (a) there is never a collision between methods of several mixins inherited from an object;
- (b) the object that inherits from one or several mixins is placed above them in the hierarchy — the opposite of inheritance.

A prototype may call methods of its mixins:

```
mixin(Person) object Comparison
  fun older: Person other {
    ^return super age > other age
  }
end

object Person mixin Comparison
  ...
  fun compare: Person other {
    // calling method older: of mixin Comparison
    if older: other {
      Out println: "#{other name} is older than #{name}"
    }
  }
  public Int age
  public String name
end
```

It is legal to send message `other:` to `self` since the compiler adds a method

```
fun older: Person other { }
```

to prototype `Person`. In `Person` objects, the method called by “`older: other`” will be `older:` of `Comparison`.

A mixin object may declare instance variables. However a mixin object that declares instance variables cannot be inherited twice in a prototype declaration:

```
mixin object WithName
  fun print { ... }
  public String name
end

mixin object WithNameColor extends WithName
  fun print { ... }
  public Int color
end

mixin object WithNameFont extends WithName
  fun print { ... }
  public String font
end

mixin object PersonName mixin WithNameFont, WithNameColor
```

```

fun print {
  Out println: "Person: ";
  super print
}
end

```

Here `WithName` is inherited by `PersonName` by two different paths:

```

WithName => WithNameColor => PersonName
WithName => WithNameFont  => PersonName

```

Hence objects of `PersonName` should have two instances of instance variable `name`. Each one should be accessed by one of the inheritance paths. Confusing and that is the reason this is not allowed. This is the same problem of *diamond* inheritance in languages that support multiple inheritance.

As said above, Cyan does not allow a mixin object that declares instance variables to be inherited by two different paths. Then the introduction of an instance variable to a mixin object may break a working code. In the above example, the introduction of `name` to `WithName` after the whole hierarchy was built would cause a compile-time error in prototype `PersonName`. That is bad. The alternative would be to prohibit instance variables in mixin objects. We believe that would be much worse than to prohibit the double inheritance of a mixin object that declares instance variables.

Let us show an example of use of mixin objects.

```

mixin(Window) object Border
  fun draw { /* draw the window */
    drawBorder;
    super.draw
  }
  fun drawBorder {
    // draw a border of color "color"
    ...
  }

  fun setBorderColor: Int color { self.color = color }
  Int color;
  ...
end

object Window mixin Border
  fun draw { /* method body */ }
  ...
end

```

Object `Window` can inherit from mixin `Border` because the mixin object is declared as `mixin(Window)` and therefore `Window` and its sub-objects can inherit from it. Methods of `Window` can be called inside mixin `Border` using `super`.

The compiler creates the following hierarchy:

```

Any
  Window' (with the body of Window)
    Window (with the body of Border)

```

See Figure 1.3. When message `draw` is sent to `Window`, method `draw` of the mixin `Border` is called. This method calls method `drawBorder` of the mixin to draw a border (in fact, the message is sent to `self` and

in this particular example method `drawBorder` of `Border` is called). After that, method `draw` of `super` is called. In this case, `super` refer to object `Window`' which has the body of the original `Window` object. Then a window is drawn.

As another example, suppose we would like to add methods `foreach` to objects that implement interface `Indexable`:

```
interface Indexable<T>
    // get element "index" of the collection
    fun get: Int index -> T
    // set element "index" of the collection to "value"
    fun set: (Int index) value: (T aValue)
    // size of the collection
    fun size -> Int
end
```

The mixin object defined below allows that:

```
mixin(Indexable<T>) object Foreach<T>
    fun foreach: Function<T, Nil> b {
        var i = 0;
        {^ i < size } whileTrue: {
            b eval: (get i)
        }
    }
end
```

Note that there may be generic mixin objects (see more on generic objects in Chapter 7). The syntax is not the ideal and may be modified at due time.

Suppose object `PersonList` implements interface `Indexable<Person>`:

```
object PersonList implements Indexable<Person> mixin Foreach<Person>
    fun get: Int index -> Person { ... }
    fun set: (Int index) value: (Person person) { ... }
    fun size -> Int { ... }
    // other methods
end
```

Now method `foreach` inherited from `Foreach` can be used with `PersonList` objects:

```
PersonList foreach: { (: Person elem :) Out println: elem }
```

Method `foreach`: sends messages `size` and `get` to `self`, which is `PersonList` in this message send. Then the methods `size` and `get` called will be those of `PersonList`. `self` can be replaced by `super` in this case. Be it `self` or `super`, the compiler does not issue an error message because methods `size` and `get` are defined in `Indexable<Person>`, the prototype that appears in the declaration of `Foreach<T>`.

Another example would be to add a method `select` that selects, from a collection, all the elements that satisfy a given condition.

```
interface ForeachInterface<T>
    fun foreach: Function<T, Nil>
end

mixin(ForeachInterface<T>) object SelectMixin<T>
```

```

    fun select: (Function<T, Boolean> condition) -> Collection<T> {
        var c = Collection<T> new;
        foreach: { (: T elem :)
            if condition eval: elem {
                c add: elem
            }
        };
        return c
    }
end

```

```

object List<T> mixin SelectMixin<T>
    ...
end

```

```

...
var list = List<Person> new;
list add: peter, john, anne, livia, carolina;
var babyList = list select: { (: Person p :) ^ (p age) < 3 };

```

Here `babyList` would have all people that are less than three years old. Note that object `Collection` in mixin object `SelectMixin` could have been a parameter to the mixin:

```

mixin(ForeachInterface<T>) object SelectMixin<T, CollectTo>
    fun select: (Function<T, Boolean> condition) -> CollectTo<T> {
        var c = CollectTo<T> new;
        foreach: { (: T elem :)
            if condition eval: elem {
                c add: elem
            }
        };
        return c
    }
end

```

The same idea can be used to create a mixin that iterates over a collection and applies a function to all elements, collecting the result into a list.

4.19 Runtime Metaobjects or Dynamic Mixins

Mixin prototypes can also be dynamically attached to objects. Returning to the `Window-Border` example, assume `Window` does not inherit from `Border`. This mixin can be attached to `Window` at runtime by the statement:

```
Window attachMixin: Border;
```

Effectively, this makes `Border` a metaobject with almost the same semantics as shells of the Green language [dOGab]. Any messages sent to `Window` will now be searched first in `Border` and then in `Window`. When `Window` is cloned or a new object is created from it using `new`, a new `Border` object is created too.

As another example, suppose you want to redirect the `print` method of object `Person` so it would call the original method and also prints the data to a printer. This can be made with the following mixin:

```
mixin(Any) object PrintToPrinter
  override fun print {
    super print;
    // print to a printer
    Printer print: (self asString)
  }
end
```

“`self asString`” returns the attached object as a string, which is printed in the printer by method `print:.` This mixin can be added to any object adding a `print` method to it:

```
object Person
  public String name
  public Int age
  override fun asString -> String {
    ^"name: #name age: #age"
  }
end
...
var p = Person new;
p name: "Carol";
p age: 1;
p attachMixin: PrintToPrinter;
  // prints both in the standart output and in the printer
p print;
Person name: "fulano";
Person age: 127;
  // print only in the standard output
Person print;
```

Note that `attachMixin` is a special method of prototype `Any`: it is added by the compiler and it can only be called by sending messages to the prototype. These dynamic mixins are runtime metaobjects. Probably they can only be efficiently implemented by changing the Java Virtual Machine (but I am not so sure). Maybe efficient implementation is possible if the metaobjects (dynamic mixins) that can be attached to an object are clearly identified:

```
object(PrintToPrinter) Person
  public String name
  public Int age
  override fun asString -> String {
    ^"name: #name age: #age"
  }
end
```

Then only `PrintToPrinter` metaobjects can be dynamically attached to `Person` objects.

The last dynamic mixin attached to an object is removed by method `popMixin` defined in prototype `Any`. It returns `true` if there was a mixin attached to the object and `false` otherwise. Therefore we can remove all dynamic mixin of an object `obj` using the code below.

```
while obj popMixin {  
}
```

The above definition of runtime mixin objects is similar to the definition of runtime metaobjects of Green [dOGab]. The semantics of both are almost equal, except that Green metaobjects may declare a `interceptAll` method that is not supported by mixin objects (yet).

Chapter 5

Metaobjects

Observation: this chapter will be completely rewritten soon. Await to read it.

A metaobject is an object that can change the behavior of a program, add information to it, or it can just inspect the source code, using the collected information in the source code itself. Metaobjects may appear in several places in a Cyan program. A metaobject is attached to a prototype, method, interface, and so on using `@` as in

```
@checkStyle object Person
  fun print {
    Out println: "name: ", name, " (" , age, ")"
  }
  public String name
  public Int age
end
```

`checkStyle` is a metaobject written in Java. The compiler loads a Java class `checkStyle` from the disk and creates an object from it. Then it calls some methods of this object passing some information on the object `Person` as parameter. For example, it could call method `change` of the `checkStyle` object passing the whole abstract syntax tree (AST) of `Person` as parameter. Then method `change` of `checkStyle` could change the AST or issue errors or warnings based on it. The intended meaning of `checkStyle` is to check whether the identifiers of the prototype follow Cyan conventions: method names and variables should start with an lower case letter and prototype and interfaces names should start with an upper case letter. In this metaobject, the AST is not changed at all.

The interactions of metaobjects with the Cyan compiler will be defined by a Meta-Object Protocol (MOP). The MOP will define how and which parts of the compiler are available to the metaobjects, which can be written by common users. Only a small part of the MOP has been designed, which is described in the next paragraphs.

Each metaobject is associated to a Java class that should inherit from class `CyanMetaobject` of the compiler. The metaobject should be part of the package “`meta`” of the compiler. This restrictions will be lifted as soon as possible and metaobjects will be implemented as a Java class that does not belong to a package. Reflection will be used to call the metaobject methods and the metaobject will also use this technique to call the compiler methods.

All Java classes for metaobjects of a package should be in a directory “`meta`” of this package. When a package is imported, so are its metaobjects.

Class `CyanMetaobject` defines the following methods:

1. `String getName()` that returns the name of the metaobject. For example, metaobject `feature`

is defined in class `CyanMetaobjectFeature` and method `getName()` of this class always returns `"feature"`;

2. `boolean attachedToSomething()` that returns `true` if the metaobject is attached to an instance variable, variable, method, statement, or prototype. This means the metaobject call know what is the next declaration (variable, method, prototype) or statement. It is expected that it modifies something based on that, which is the usual case. Metaobject `javacode` is described by the Java class `CyanMetaobjectJavaCode` that defines a method `attachedToSomething` that returns `false`. Then a call to `javacode` is not attached to anything.

5.1 Pre-defined Metaobjects

There is a set of metaobjects that are automatically available in every Cyan source code: `prototypeCallOnly`, `javacode`, `annot`, `feature`, `text`, `dynOnce`, `dynAlways` (page 131), `doc`, `clearlyTyped`, `subprototypeList`, and `init` (see page 73).

Metaobject `prototypeCallOnly` should be followed by a public method declaration. It checks whether the method is only called through the prototype. A call to the method using a variable is forbidden:

```
object Person
  public String name
  @prototypeCallOnly fun create -> Person { ~ Person new }
end
...
var Person p;
p = Person create; // ok
p = p create; // compile-time error
```

All new methods are implicitly declared by the compiler with a metaobject `prototypeCallOnly`. Even if the user declares a new method the compiler attaches to it this metaobject.

The pre-defined metaobject `annot` attaches to an instance variable, shared variable, method, constant, prototype, or interface a feature given by its parameter, which may be any object. This feature can be retrieved at runtime by method

```
fun annotList -> Array<Any>
```

inherited by any object from `Any`.

As an example of its use, consider an annotation of object `Person`:

```
@annot( #Author ) object Person
  fun print {
    Out println: "name: ", name, " (", age, ")"
  }
  @annot( #Authorname )
  public String name
  public Int age
end
```

There could exist a prototype XML to create XML files. Method `write:` of `XML` takes an object `writeThis` as parameter and writes it to a file `filename` as XML code using the annotations as XML tags:

```
write: (Any writeThis) tofilename: (String filename)
```

The annotated instance and shared variables are written in the XML file. The root element is the annotation of the prototype. Therefore the code

```

Person name: "Carol";
Person age: 1;
XML write: Person tofile: "Person.xml";

```

produces a file "Person.xml" with the contents:

```

<?xml version="1.0"?>
<Author>
  <Authorname>
    Carol
  </Authorname>
</Author>

```

To see one more example of use of annotations, see page 147 of Chapter 8 on tuples and unions. `text` is another pre-defined metaobject. It allows one to put any text between the two sets of symbols. This text is passed by the compiler to a method of this metaobject which returns to the compiler an object of the AST representing an array of characters. So we can use it as in

```

var Array<Char> xmlCode;
xmlCode = @text<**
<?xml version="1.0"?>
<Author>
  <Authorname>
    Carol
  </Authorname>
</Author>
**>

```

`xmlCode` has the text of the XML code as an array of `Char`'s.

Metaobject `strtext` works exactly as `text` but it produces a `String`. In either one, `#{expr}` is replaced by the value of `expr` at runtime:

```

Person name: "Peter" salary: 10000.0;
var String text;
text = @strtext(+
  The name is #{Person name} and the salary is #{Person salary}
+);
// prints "The name is Peter and the salary is 10000.00"
Out println: text;

```

Every prototype has a list of features. A feature is simple a key-value pair in which the key is a string and value can be any object. Different objects of a prototype share the feature list.

At compile-time, a feature is associated to a prototype by the pre-defined metaobject `feature`:

```

@feature("compiler", #nowarning) @feature<* "author", "José" *>
object Test extends Any
  fun run {
    featureList foreach: { (: NTuple<key, String, value, Any> elem :)
      Out println: "key is #{elem key}, value is #{elem value}"
    }
  }
end

```

features are used to associate information to prototypes (including interfaces), instance variables, methods and constants. This information can be used by tools to do whatever is necessary. The example given in page 112 uses annotations (a kind of features) to produce a XML file from a tree of objects. Annotations are used in grammar methods to automatically produce an AST from a grammar message send.

The parameters to metaobject `feature` should be literal strings or Cyan symbols. features can be used to set compiler options. In the example above, the compiler is instructed to give no warnings in the compilation of `Test`. Maybe it would warn that `Any` is already automatically inherited.

Method `featureList` is inherited from `Any` by any prototype. It returns an array with all features of the prototype. This array has type

```
Array<NTuple<key, String, value, Any>>
```

That is, the elements are tuples with fields `key` and `value`. In the above example, method `run` scans the array returned by `featureList` and prints information on each feature. Since `elem` has type

```
NTuple<key, String, value, Any>
```

`elem` has methods `key` and `value` for retrieving the fields of the tuple. Method `run` will print

```
key is compiler, value is nowarning
key is author, value is José
```

Possibly the compiler will add some features to each prototype such as the compiler name, version, compiler options, date, author of the code, and so on.

Annotations are a special case of *features*. A call

```
@annot( #first )
```

is the same as

```
@feature("annot", #first)
```

Method `annotList` is inherited from `Any` by any prototype. It returns a list of annotation objects attached to the prototype.

```
@annot( #first ) @annot("second") object Test
  fun run {
    annotList foreach: { (: String annot :)
      Out print: annot + " "
    }
  }
end
```

When `run` is called, it prints

```
first second
```

Since methods are objects (see Section 10.8), one can discover the annotations of methods too:

```
object Test
  @annot( #f1 ) @annot( #firstMethod ) fun test { }
  fun run {
    // Test getMethod: "test" is the method test of Test
    ( (Test getMethod: "test") annotList) foreach: {
      (: String annot :)
      Out print: annot + " "
    }
  }
end
```

`doc` is a pre-defined metaobject used to document any kind of identifier such as prototypes, constants, interfaces, and methods.

```
@doc<<
  This is a syntactic analyzer.
  It should be called as
    Parser parse: "to be compiled"
>>
object Parser
  ...
end
```

This call is equivalent to “`@feature("doc", doctext)`” in which `doctext` is the text that appears between `<<` and `>>` in this example.

There is a pre-defined metaobject `javacode` that inserts Java code in Cyan programs:

```
object Out
  fun println: (String s) {
    @javacode(**
      System.out.println(s);
    **)
  }
  ...
end
```

As soon as possible this metaobject will be eliminated.

Metaobject `subprototypeList` should be attached to a prototype declaration. It limits the inheritance of the prototype.

```
package main

@subprototypeList(AddExpr, MultExpr, LiteralExpr)
abstract object Expr
  abstract fun eval -> Int
end
```

Prototype `Expr` can only be inherited by prototypes `AddExpr`, `MultExpr`, and `LiteralExpr`.

Metaobject `onChangeWarn` may be attached to a prototype (including interfaces), a method declaration, or an instance variable declaration. This metaobject adds information to the XML file describing the current source code. This information will be used in future compilations of the source code *even if the metaobject call is removed from the code*. Using `onChangeWarn` one can ask the compiler to issue a warning whenever the signature of a method was changed, even after the programmer deleted the call to `onChangeWarn`:

```
object Test
  @onChangeWarn( #signature,
    "This signature should not be changed." +
    " Keep it as 'fun test'")
  fun test { ... }
end
```

If the signature was changed to

```
object Test
  fun test: (Int n) { ... }
end
```

the compiler would issue the warning given in the second parameter of the call to `onChangeWarn`. All methods of the same name are grouped in the same set — they all can be considered as the same *multi-method*. By change in the signature we mean any changes in this set, which may be addition of method, deletion of method, changes in the parameter type of any method, changes in the return value type of all methods.

`onChangeWarn` takes two parameters. The first specifies the change, which may be:

- (a) `#signature` for changes in the method signature;
- (b) `#name` for changes in the name (used for prototypes);
- (c) `#type` for changes in the return type of a method, type of a variable;
- (d) `#qualifier` for changes in the visibility qualifier (public, protected, private);
- (e) `#all` for any changes whatsoever.

The second parameter gives the message that should be issued if the change specified in the first one was made. It should be a string. Other metaobjects that makes the linking past-future will be added to Cyan. Await.

There are other metaobjects used in the Cyan library. For example, there is `checkAddMethod` that checks whether the parameters to the grammar method `addMethod: ... of Any` are correct. And there are metaobjects for defining literal objects in the language — see Section 13.

5.2 Syntax and Semantics

Cyan metaobjects may take parameters which may be followed by an arbitrary text. The parameters are given between (and), { and }, or [and]. Commas separate the arguments as in a method call. The text that follows the parameters should be delimited by two sequences of symbols that we will call left and right symbol sequence. The left sequence of symbols should be put after the metaobject name or the character that closes the parameter list. The right symbol sequence should be the mirror of the first. A metaobject that may take parameters may accept zero parameters — it depends on the metaobject. In this case, there should appear the symbols (), [], or {} after the metaobject name:

```
@meta()
```

As an example of metaobject calls without the text we have:

```
1 @annot( #Author )
2 @annot{ #today }
3 @annot[ important ]
4 var String name;
5
6 @checkStyle // without parameters
7 @authors( {# "José", "Carolina" #} )
8 @version(3.21)
9 object Person ... end
```

Here `important` is not a basic Cyan literal. But it is a Cyan identifier which is converted to a symbol before being passed to the metaobject. Then

```
@annot[ important ]
```

is the same as

```
@annot[ #important ]
```

The following example gives metaobject calls with a text:

```
1 var Array<Char> myText = @text(trim_spaces)[(* this is
2     a text which ends with * ) ], but
3     without spaces
4     *)];
5
6 var Graph aGraph = @graph<**
7     (0, 1), (0, 2), (0, 3),
8     (1, 2), (2, 1), (3, 0),
9     (3, 2)
10    **>;
```

The valid symbols are:

```
!@$%&*()-+={}[ ] ^~<>:~/
```

Note that symbols " , ; . # ' ` " cannot be used. A metaobject may be called using any set of symbols.

Whenever there is a (, {, or [after the metaobject name, not followed by any valid sequence symbol, the compiler will assume that there is a parameter list. So, in `@meta(a, b)` there is a parameter list with parameters `#a` and `#b`. But in `@meta(+a, b+)` there is no parameter list. "a, b" is the text of the metaobject call.

When the compiler finds a metaobject call such as "`@annot(9)`", it searches for a Java class, subclass of `CyanMetaobject`, that treats this metaobject. That is, method `getName()` of an object of this class returns "annot".

Then the compiler calls method `mayTakeArguments` of this Java object. If it returns `true`, the metaobject may accept literal expressions of basic types as parameters. If it returns `false` the metaobject never accepts parameters.

Metaobject `annot` is of the first kind. `text` is of the second kind.

Both kinds of metaobjects are delimited by two sequences of symbols that we will call left and right symbol sequence. The left sequence of symbols should be put, without spaces, after the metaobject name. The right symbol sequence should be the mirror of the first. Then valid metaobject calls are:

```
1 @annot( #Author )
2 @annot<& #Author &>
3 @annot[#Author]
4 @name
5 @name([+Ok, this ...end+])
6 @name([+ Ok, this
7     ... end +])
8 @text<<< this is
9     a text which ends with < < <, but
10    without spaces
11    >>>
12 @text{ another text
```

```
13     this is the end: }
```

The valid symbols are:

```
!@$%&*()-+={}[]^~<>:~/
```

Note that symbols " , ; . # ' ` cannot be used. A metaobject may be called using any set of symbols. In lines 1, 2, and 3, `annot` is called with the same parameter in three different ways.

Depending on the methods defined by the metaobject, the compiler passes to a method of the metaobject:

- (a) the text between the two sequences of symbols;
- (b) the parameters between the two sequences of symbols;
- (c) the AST of the text between the two sequences of symbols.¹

The call of line 1 could be in the last case. Here “`#Author`” can be evaluated at compile-time resulting in a string that is passed as argument to a method of metaobject `annot`.

Options can be passed to the metaobject between (and) (or `and` or { and }). After the), there should appear at least one space and another set of symbols starting the text:

```
@meta(option1, option2, ... optionN) <<+ ... +>>
```

Of course, any valid delimiter may replace `<<+ or +>>`. The options may be any valid Cyan identifier that start with a letter.

Metaobject `text` has an option `trim_spaces` to trim the spaces that appear before the column of `@` in `@text`. As an example, variables `t1` and `t2` have the same content.

```
var t1 = @text(&
starts at
line 1
&);
var t2 = @text(trim_spaces) <<<*>>>
starts at
line 1
*>>>;
assert: (t1 == t2);
```

Assume that the text editor trims spaces before the last non-blank character in a line. If there is any non-blank character in a column smaller than the column of `@`, metaobject `text` issues an error.

5.3 Metaobject Examples

Compile-time metaobjects have thousands of applications. We describe next some metaobjects without giving any hint on how they will be implemented (there is no MOP yet).

A metaobject `singleton` may be used to implement the design pattern *singleton* [GHJV95].

```
// CTMO on an object
@singleton object Earth
  fun mass -> Float { ^earthMass }
  Float earthMass = 6e+24;
  ...
end
```

¹Currently the compiler does not support this.

The metaobject redefines method `clone` and `new`.

```
fun clone -> Earth { ^Earth }
@prototypeCallOnly
fun new -> Earth { ^Earth }
```

It also checks whether there is any other `init`, `clone`, or `new` method declared in the prototype body. If there is, it signs an error.

Metaobject `profilePrototype` inserts code before every method of the prototype to count how many times every method was called. The results are added to a file. At the end of the program, the runtime statistics of calls may be printed.

Metaobject `beforeCode` should be attached to a method. It inserts some code to be executed before the execution of the method code. For example, it could initiate a transaction in a data base or lock some data in a concurrent program.

5.4 Codegs

There is a special kind of compile-time metaobject called *Codeg* (code + egg) that makes the integration between the compiler and the IDE used with Cyan. Each codeg works like a plug-in to the IDE but with the added power of being a metaobject of the language. There are many technical details of the workings of a codeg. Few of them will be given here. For more information, read the report [Vid11] (in Portuguese).

Codegs have been fully implemented using the IDE Eclipse [?] by adding a plug-in to it. Therefore currently codegs work only in Eclipse. After installing the Codeg plug-in and defining a project as being a “Cyan project”, source files ending with “.cyan” will receive special treatment. Let us shown an example using codeg “color”². When the user type

```
@@color(red)
```

the Eclipse editor loads a Java class to memory that treats the codeg “color”. This text will be shown in a color different from the rest of the code (the color will be blue regardless of the codeg). By putting the mouse pointer above `@@color(red)`, a standard menu will appear which allows the editing of this codeg call.³ This menu is standard just by convention — the codeg designer is free to choose another one if she so wishes.

By clicking in an “edit” button in the menu, another window will appear with a disk of colors. A color may now be chosen with the mouse. After that, the user should click in the “Ok” button. All codeg windows will disappear and the source code editing may continue. Now when the mouse pointer is over the text “`@@color(red)`” the standard menu will appear with an edit button and a bar showing the chosen color (it is expected that “red” was chosen).

This is what happened at editing time. When the compiler finds the codeg call “`@@color(red)`” it will load the codeg class (written in Java), create an object from it, and calls method `getCode` of this object. This method takes a parameter of class `CodeGenerationContext` that gives information on the compilation itself. In future versions of the compiler, the AST of the current source code will be available from the `CodeGenerationContext` object. Currently this class only provides two methods: `getLanguage` and `getCodegsList`. The first method returns the target language of the codeg, which may be Java or Cyan. In due time, there will be only the options AST and Cyan. Method `getCodegsList` returns a list of codegs of the same source code. This allows communications among the codegs of the same file.

Method `getCode` returns an object of class `CodeGeneration`. This class has three methods that return the generated code:

²For didactic reasons, the codeg described here may differ from the real implementation.

³Since codegs are metaobjects, this is a metaobject call.


```
String getLocalCode()
String[] getImports()
String getGlobalCode()
```

The string returned by `getLocalCode` replaces the codeg call, “`@@color(red)`”. It should be compiled by the compiler and may contain errors although it is expected that it does not. This code may need packages that were not imported by the source file in which the codeg call is. The packages used in the code returned by `getLocalCode` should be returned by method `getImports`. Finally, `getGlobalCode` returns code that should be added just after the import section of the source code in which this code is. So, suppose we have a code like

```
package main

object Program
  fun run {
    Out println: @@color(red)
  }
}
```

Consider that method `getCode` of codeg `color` (which is a Java class called `ColorCodeg`) returns an object of `CodeGeneration` whose methods return the following:

```
String getLocalCode()    returns "RGBColor new: 255, 0, 0"
String[] getImports()    returns "RGB"
String getGlobalCode()   returns "/* global code */"
```

Then the compiler will add these strings to the source code in such a way that the program above will become

```
package main
import RGB

/* global code */

object Program
  fun run {
    Out println: (RGBColor new: 255, 0, 0)
  }
}
```

It is expected that prototype `RGBColor` is in package `RGB`.

Method `getCodegsList` is used for communication among the codegs of the same source file. Codegs `world` and `actor` use this method. They implement a very small programming learning environment that resembles Greenfoot [?].

```
package main
object Program
  fun run {
    @@world(myWorld)
    @@actor(ladybug)
    @@actor(butterfly)
  }
}
```

The `world` codeg call generates code that creates a windows in which all actions will happen. At editing time, the user may choose the size of the window and its background color.

Codeg `actor` defines an actor that will be added to the world at runtime. At editing time, the user may choose a color for the actor⁴ and the code the actor should obey. This code is given in a small language called *Locyan* defined in the report [dOGa11] (in Portuguese). It is a Logo-based language [LF12].

It is an error to define two codegs “actor” with the same name:

```
@@actor(ladybug)
@@actor(ladybug)
```

But how the actor codeg may detect this? Using `getCodegList`. A codeg actor call scans the list of codegs returned by this method searching for a codeg with the same name as the current one. If it finds one, it issues an error.

Codeg `world` is responsible for creating the actor objects and putting them in motion. This motion is specified by the *Locyan* code associated to each actor. Therefore the world codeg calls `getCodegsList` to retrieve the codegs of the same source code. It uses this information in order to generate code.


Several other codegs have been implemented:

- (a) `color`, `world`, and `actor`, already described;
- (b) `matrix`, which allows a two dimensional matrix to be edited like a spreadsheet;
- (c) `image`, that encapsulates the path of an image in the file system. A future improvement would be to keep the image itself in the codeg;
- (d) `file`, that encapsulates the path of a file and options for reading or writing. The generated code is the creation of an object representing a file with the options chosen;
- (e) `text`, which pops up a text editor and returns the edited text either as a `String` or an array of `Char`’s. A generalization of this would allow code in HTML or XML inside *Cyan* code.

Nowadays when the mouse pointer is on a codeg call such as “`@@color(red)`” the codeg plugin of the IDE shows a menu. This menu includes a bar with the color in this case. Or the image in the `image` codeg. Future versions of the plugin may replace the codeg call with an image. Then

```
var RGBColor color = @@color(black);
```

would be shown as

```
var RGBColor color = ;
```

in the editor. In the codeg `image`, the real image would be shown.

The metaobject protocol of codegs is minimal. That needs to be changed. It is necessary to add to the protocol:

- (a) better mechanisms for communication among codegs of the same source file;
- (b) communication of codegs of different source files. Then code generated in one source file may depend on options of another source file. This will probably be used in the implementation of Design Patterns that need more than one prototype;
- (c) access to compiler data such as the local variables, prototype name, source file name, compiler options, etc. For short, the whole AST and other compiler information should be available to the codegs;
- (d) methods that return the code generated by the codeg in form of the AST.

⁴It would be good to allow the user to choose a picture instead of just a color — this will be allowed some day.

There are endless uses for codegs. We can cite some codegs that we would like to implement. Most of them depend on features that are not available nowadays.

- (a) An interactive console for Cyan similar to those of scripting languages. The user could just type `@@console()` in any Cyan source file and experiment with the language.
- (b) Codeg `test` for testing. This codeg could show a spreadsheet with expressions that are evaluated at compile time:

code	checks
<code>:set = IntSet new;</code>	
<code>set add: 0;</code>	
<code>set add: 1</code>	<code>set size == 2</code>
	<code>name == "UFSCar"</code>

In this figure, “name” is used without qualification. It could be a local variable or an instance variable or method of the prototype in which the codeg is declared.

The codeg would load the last compiled prototypes cited in the spreadsheet. Then it would create, compile, and run the code of the cells, checking the results. Errors could be shown in red. More than that: all `test` codegs could communicate with a `programTest` codeg that would shown all the places with errors. The `test` codeg could offer tools for make it easy to do test the program.

- (c) Codegs that implement design patterns. The programmer gives the information and the codegs generate the code. There should be an option for replacing the codeg call with the generated code.
- (d) `PerfectHashtable` that generates a perfect hash table given a list of fixed keys.
- (e) codegs to help to build grammar methods — Chapter 9.
- (f) `FSM` which allows one to define graphically a finite state machine. The generated code would be an object of prototype `FSM`.
- (g) `TuringMachine`. As the name says, the user could define graphically a Turing machine (much like the `FSM`).

The problem with codegs is that they link tightly the source code and the IDE. Changing the IDE means losing the codegs if the new IDE does not support exactly the same set of codegs the old one supports. Although this can bring some problems, it is not so bad for two reasons:

- (a) the compiler will continue to work as expected because it uses the data collected in the old IDE. Although this data cannot be changed by a Codeg in the new IDE, the code will continue to compile;
- (b) it is easy to add to the compiler an option that makes it replace every codeg call by the code produced by that codeg call. This will eliminate all codegs from a source code. And with them the dependency from the old IDE.

Textual programming has dominated programming languages for a long time. By textual programming we mean that all source code is typed in a text editor as is made in C/C++/Ruby/Java/C#/Lisp/Prolog/etc. There has been several attempts to change that such as the integrated environment of Smalltalk and visual programming languages. It is difficult to imagine software development within one hundred years based on full textual representation of programs like most of the code made today. There should be some visual representation. Codegs are another attempt to achieve that.

5.5 Macros

A macro is a function called at compile time that produces code that replaces the macro call. They can be used to capture common patterns of code, implement Domain Specific Languages, eliminate repetitive statements and boilerplate code. Cyan supports a restricted but powerful macro feature. Macros in Cyan are also metaobjects. They have some but not all the power of regular metaobjects. There are two ways of defining a macro in the language: a low-level and a high-level way. Macros defined in a low-level way will be called LL-macro and HL-macro. First we will describe the low-level macros. The high-level ones are implemented in terms of the low-level ones. That is, a source code that defines a HL-macro is translated by the compiler into a source code of a LL-macro.

An LL-macro is a regular Java class which should inherit from class `CyanMetaobjectMacro` of package `meta`. This is a class supplied with the Cyan compiler. Many other Java classes of the Cyan compiler are employed in the definition of a macro (or any other metaobject). Of course, a Cyan prototype or a class of another language compiled to the Java Virtual Machine can be used as a LL-macro. It only has to obey the name conventions described in this text.

As a subclass of `CyanMetaobjectMacro`, an LL-macro should implement the following methods defined in its superclass:

(a) `abstract public String []getStartKeyword();`

This method should return a list of keywords that may start the macro. These keywords are specific to this macro, they are not Cyan keywords. For example, a macro “`assert`” with the syntax

```
assert size > 0;
```

should return `new String [] { "assert" } .`

(b) `abstract public String []getKeywords();`

This method should return a list of all keywords that this macro uses. For example, consider a macro that emulates the extended “`for`” of Java. It would be used, in Cyan, as

```
var Array<Int> aList = {# 0, 1, 2 #};
var sum = 0;
// sum the array elements
for n in aList {
    sum = sum + n
};
```

There are two keywords in this macro: “`for`” and “`in`”. Then method `getKeywords` of the LL-macro should be

```
public String []getKeywords() {
    return new String[] { "for", "in" };
}
```

This list is used by the Cyan compiler to parse expressions. Without this list, the compiler would guess that “`in aList`” in the above example is the sending of method `aList` to variable `in`. Or the sending of message `in` to variable `n`. With this list of keywords, the compiler knows where an expression or a statement ends.

(c) `public StringBuffer bti_cyanCodeThatReplacesMetaobjectCall(ICompiler_bti compiler, int offsetStartLine)`

If this method returns a non-null value, the compiler replaces the macro call by the string returned by this method. For example, suppose class

`CyanMetaobjectMacroAssert`

defines this method in such a way that, in a call to the macro “`assert`” in code

```
var Int size = anArray size;
assert size > 0;
Out println: "Ok";
```

this method returns

```
"if ( !(size > 0) ) { System exit: 1 };
```

Then the compiler will produce the code

```
var Int size = anArray size;
if ( !(size > 0) ) { System exit: 1 };
Out println: "Ok";
```

The compilation will resume at Cyan keyword “`if`”. Of course, the code returned by

`bti_cyanCodeThatReplacesMetaobjectCall`

can have other macro calls. If there is always a call to `assert` then we end up in an infinit loop.

Parameter `compiler` of type `ICompiler_bti` of this method is used for parsing the macro call. `ICompiler_bti` is defined as

```
public interface ICompiler_bti {
    void next();
    Symbol getSymbol();
    Expr expr();
    void error( Symbol sym, String specificMessage,
               String identifier, ErrorKind errorKind,
               String ...furtherArgs );
}
```

Method `getSymbol()` of parameter `compiler` returns the current token of the compilation, the one returned by the lexical analyzer. Class `Symbol` is defined in package `lexer` of the Cyan compiler. Method `next` gets the next token. After this method is called `getSymbol()` will return the symbol found by the call to `next`.

When the symbol is a macro keyword (of this specific macro), the symbol will be of class `SymbolMacroKeyword`. The token will be `MACRO_KEYWORD`. Then to discover if the current token is a macro keyword and this keyword is “`in`”, use the following code:

```
if ( compiler.getSymbol().token == Token.MACRO_KEYWORD &&
     compiler.getSymbol().getSymbolString().equals("in") )
    System.out.println("found 'in'");
```

A macro keyword of another macro that is not the currently being analyzed is treated as a regular identifier.

Method `expr()` parses an expression. It will sign an error if an expression is not found. This method returns an object of `Expr` that is a class of the Abstract Syntax Tree (AST) of the Cyan compiler. Method `error()` should be called to sign an error in the macro compilation. The first parameter is the symbol that caused the error, `null` if none. The second is the error message. The third is the identifier associated to the error, `null` if none. The fourth is a constant of enumeration `ErrorKind` of this error message. If none is appropriate, use `error_in_macro`. The last parameter should not be supplied (for the time being).

The second parameter to `bti_cyanCodeThatReplacesMetaobjectCall` is `offsetStartLine`. This is the number of characters from the start of the line to the first macro character. It is used with indentation purposes. To better understand how macros work, study this method of class `CyanMetaobjectMacroAssert`. This code uses static variables `whiteSpace` and `sizeWhiteSpace` of class `CyanMetaobjectMacro`. `whiteSpec` is a string of white spaces and `sizeWhiteSpace` is the size of this string.

```
package meta;

import java.util.ArrayList;
import error.ErrorKind;
import lexer.Token;
import ast.*;

/**
 * This class represents macro 'assert'
 *
 * @author José
 */
public class CyanMetaobjectMacroAssert extends CyanMetaobjectMacro {

public CyanMetaobjectMacroAssert() {
}

@Override
public ArrayList<MetaobjectError> errorMessageList() {
return null;
}

@Override
public String []getStartKeyword() {
return new String[] { "assert" };
}

@Override
public String[] getKeywords() {
return new String[] { "assert" };
}

@Override
```

```

public StringBuffer bti_cyanCodeThatReplacesMetaobjectCall(
ICompiler_bti compiler, int offsetStartLine) {

    compiler.next();
    Expr e = compiler.expr();
    if ( compiler.getSymbol().token != Token.SEMICOLON )
        compiler.error(compiler.getSymbol(), "';' expected", null,
            ErrorKind.semicolon_expected, new String[] {});
    else
        compiler.next();
    StringBuffer s = new StringBuffer();
    if ( offsetStartLine > CyanMetaobjectMacro.sizeWhiteSpace )
        offsetStartLine = CyanMetaobjectMacro.sizeWhiteSpace;
    String identSpace = CyanMetaobjectMacro.whiteSpace.substring(0, offsetStartLine);
    s.append("if !(");
    s.append(e.asString() + ") {\n");
    s.append(identSpace + "    System exit: 1;\n");
    s.append(identSpace + "};\n");
    return s;
}

}

```

Method `bti_cyanCodeThatReplacesMetaobjectCall` is called by the Cyan compiler “before typing interfaces” (bti). That is, before the compiler discover the types of instance variables, constantes, method parameters, and method return value types. And long before the compiler knows the types of local variables and expressions inside methods. Then this macro cannot check whether the assert expression is of type `Boolean` (it should be). A statement

```

    if ( e.getType() != Type.Boolean ) compiler.error(...)
after

```

```

    Expr e = compiler.expr();

```

would result in a runtime error signed by method `getType()`.

To know the type of expression `e` one should implement this macro using method

```

    dsa_cyanCodeThatReplacesMetaobjectCall

```

```

/**

```

```

 * the compiler will replace the macro call by the string
 * returned by this method, if it returns a non-null value.
 * If the return value is null, a list of error messages
 * can be got by calling errorMessageList
 * @param offsetStartLine TODO
 */

```

```

/**

```

```

 * The code returned by this method replaces the metaobject call. Then in a macro call
 *     assert i == 0;
 *

```

```

* if method bti_cyanCodeThatReplacesMetaobjectCall of the macro assert returns
*   "if ( !(i == 0) ) { System exit: 1 };"
* this string will replace "assert i == 0" in the code:
*
* This method is called during semantic analysis.
*
  @return
*/
public StringBuffer dsa_cyanCodeThatReplacesMetaobjectCall() {
return null;
}

```

A macro should be defined in a regular public prototype with one and just one of the methods prefixed by the keyword “macro”. Whenever a compilation unit imports the package of this prototype⁵ it will be importing the macro. Methods prefixed by keyword “macro” should be public and they can be grammar methods. In this section we will give only non-grammar methods as examples.

⁵Or imports only the prototype, which will soon be allowed in Cyan.

Chapter 6

Dynamic Typing

A dynamically-typed language does not demand that the source code declares the type of variables, parameters, or methods (the return value type). This allows fast coding, sometimes up to ten times faster than the same code made in a statically-typed language. All type checking is made at runtime, which brings some problems: the program is slower to run and it may have hidden type errors. When a type error occur, usually the program is terminated. Statically-typed languages produce faster programs and all type errors are caught at compile time. However, program development is slower.

The ideal situation is to combine both approaches: to develop the program using dynamic typing and, after the development ends, convert it to static typing. Cyan offers three mechanism that help to achieve this objective.

The first one is dynamic message sends. A message send whose selectors are preceded by `?` is not checked at compile-time. That is, the compiler does not check whether the static type of the expression receiving that message declares a method with those selectors. For example, in the code below, the compiler does not check whether prototype `Person` defines a method with selectors `name:` and `age:` that accepts as parameters a `String` and an `Int`.

```
var Person p;  
...  
p ?name: "Peter" ?age: 31;
```

This non-checked message send is useful when the exact type of the receiver is not known:

```
fun openArray: (Array<Any> anArray) {  
    anArray foreach: { (: Any elem :)  
        elem ?open  
    }  
}
```

The array could have objects of any type. At runtime, a message `open` is sent to all of them. If all objects of the array implemented an `IOpen` interface,¹ then we could declare parameter `anArray` with type `Array<IOpen>`. However, this may not be the case and some kind of dynamic message send would be necessary to call method `open` of all objects.

If every message selector (such as `open` in the above examples) is preceded by a `?` we have transformed Cyan into a dynamically-typed language. If just some of the selectors are preceded by `?`, then the program will use a mixture of dynamic and static type checking.

Keyword `Dyn` is used for a dynamic type in Cyan. `Dyn` is not a prototype. It is a virtual type that is supertype and subtype of every other prototype including `Nil`. Therefore assignments to and from

¹With a method `open`.

Dyn are always legal at compile-time. At runtime there is a check in assignments from Dyn to any other type (that includes, of course, parameter passing, which is a kind of assignment). At runtime the Dyn expression should refer to a prototype that is subtype of the type of the left-hand side variable.²

```
var Person p;  
var Dyn dynVar;  
...  
p = dynVar;
```

In the assignment the compiler inserts a check to verify whether `dynVar` refer to an object whose type is subtype of `Person`. `Nil` can be assigned to a `Dyn` variable and an expression whose type is `Dyn` can be assigned to a variable whose type is `Nil`.

Assignments whose left-hand side is `Dyn` need not to be checked either at compile or runtime. Since `Dyn` is not a prototype, it cannot be used as an expression:

```
(Dyn prototypeName) println; // compile-time error
```

A message sent to a receiver whose type is `Dyn` is not checked by the compiler. The return value type of the message send is considered to be `Dyn` too. Then if the type of a variable is `Dyn` we can send to it a regular message, without `?` preceding the selectors.

```
var Dyn p = Person;  
p name: "Peter" age: 31;
```

The compiler will not do any checking. This is equivalent to declare `p` with any other type and use `?` before the selectors. `Dyn` is considered a supertype and a subtype of any prototype. Of course, it is a virtual type, there is no source file `Dyn.cyan`.

The return value type of a message send is considered to be `Dyn` when the selector is `Dyn`. Therefore the return value is not checked. In this example, the compiler considers that `get:` returns `Dyn` and, since it is a subtype of `Boolean`, there is no error.

```
var Dyn t = MyHashtable<String, String>;  
if t get: "one" == "1" {  
  "found one" println  
};
```

A call

```
obj ?set: 11;
```

is roughly equivalent to

```
obj selector: "set:" param: 11;
```

in which `selector:param:` is a method inherited from `Any` by every `Cyan` object. It invokes the method corresponding to the given selector using the parameters after `param:`. Therefore `#message` sends are a syntax sugar for a call to the `selector:param:` method with one important difference: the compiler does not do any further type checking with the return type of the method. That is, any use of the return value is considered correct.

```
var stack = Stack<Int>  
  // no compile-time error here  
stack push: (obj ?get);  
  // no compile-time error here  
if obj ?value {  
  stack push: 0
```

²or indexing expression like `a[0] = dynVar`.

```
};
```

The compiler just checks, in “stack push: (obj ?get)” that `Stack<Int>` has a method `push:` that accepts one parameter. When the return value of a dynamic message is assigned to a variable declared without a type, the compiler considers that the type of the variable is `Any`.

```
var n = obj ?value;
assert: (n prototypeName) == "Any";
```

Several statically-typed languages such as Java allows one to call a method using its name (as a string) and arguments. Cyan just supplies an easy syntax for doing the same thing. Section 4.13 describes the `selector:param:` method which is in fact a grammar method. Grammar methods are described in Chapter 9.

Dynamic checking with `?` plus the reflective facilities of Cyan can be used to create objects with dynamic fields. Object `DTuple` of the language library allows one to add fields dynamically:

```
var t = DTuple new;
t ?name: "Carolina";
  // prints "Carolina"
Out println: (t ?name);
  // if uncommented the line below would produce a runtime error
//Out println: (t ?age);
t ?age: 1;
  // prints 1
Out println: (t ?age);
```

Here fields `name` and `age` were dynamically added to object `t`. Whenever a message is sent to an object and it does not have the appropriate method, method `doesNotUnderstand` is called. The original message with the parameters are passed to this method. Every object has a `doesNotUnderstand` method inherited from `Any`. In `DTuple`, this method is overridden in such a way that, when a `DTuple` object receives a message `?id: expr` without having a `?id: method`, `doesNotUnderstand` creates:

- (a) two methods, `id: T` and `T id`, in which `T` is the dynamic type of `expr`;
- (b) a field `_id` of type `T` in the `DTuple` object. Prototype `DTuple` inherits from a mixin that defines a hash table used to store the added fields.

Then message `?id: expr` is sent to the object (now it does have a `?id: method` and no runtime error occurs).

The above code can be made more legible by declaring `t` with type `Dyn`.

```
var Dyn t = DTuple new;
t name: "Carolina";
  // prints "Carolina"
Out println: (t name);
  // if uncommented the line below would produce a runtime error
//Out println: (t age);
t age: 1;
  // prints 1
Out println: (t age);
```

Cyan supports the ``` operator (backquote) which “link the runtime value of a `String` variable to a compile-time meaning of this value”. Each selector preceded by backquote should be a variable of type `String` or `CySymbol`. It cannot be an instance variable accessed through `self` as `self.name`.

```
var String s = "print";
0 's;
```

The real selector is not the variable name but the contents of the variable. That is, the method to be called is the value of the variable, "print", which is a string. In a message send with parameters the variable names should be followed by `:` as usual.

```
var String s = "key", p = "value";
MyHashtable 's: "one" 'p: "1";
```

The method to be called has name `s + ":" + p + ":"`, in which `+` is used for concatenating strings. That is, the method to be called is `key:value:`.

The return value of a backquote message send is `Dyn`. Then the return value can be assigned to any variable and passed as parameter to any method. The compiler will insert runtime checks in the code. Let us study one more example of backquote message sends.

```
var String selector;
selector = String cast: ((In readInt > 0) t: "prototypeName" f: "asString");
    // a runtime test is inserted to check if the result
    // is really a String
var result String = 0 'selector;
Out println: result;
```

Here `0 'selector` is the sending of the message given by the runtime value of `selector` to object `0`. If `selector` is "prototypeName", the result will be "Int". If `selector` is "asString", the result will be "0".

The backquote operator cannot be used in a chain of unary message sends. Then it is illegal to write either

```
club 'first 'second
```

or

```
club members 'second
```

That is, a chain of message sends in which there is a backquote should have size one.

Language Groovy has this mechanism for message sends:

```
animal."$action"()
```

The method of `animal` called will be that of variable `action`, which should refer to a `String`.

The second and third mechanisms that allow dynamic typing in Cyan are the metaobjects `dynOnce` and `dynAlways`. These are pre-defined metaobjects — it is not necessary to import anything in order to use them.

Metaobject `@dynOnce` makes types optional in declarations of variables and methods of the prototype it is attached to. For example, the instance variables and parameters of prototype `Person` are declared without a type:

```
@dynOnce object Person
    fun init: (newName, newAge) {
        name = newName;
        setAge: newAge
    }
    fun print {
        Out println: "name: ", name, " (" , age, ")"
    }
}
```

```

fun setAge: ( newAge ) {
    if newAge >= 0 and newAge <= MaxAge {
        age = newAge
    }
}
private name
private age
private const Int MaxAge = 126
end

```

The compiler will not issue no error or warning. After the program runs for the first time, it may be the case that an object of prototype `Person` is used — maybe `Person` itself receives messages or maybe an object created from `Person` using `clone` or `new` receives messages. In any case, metaobject `dynOnce` inserts statements in the generated `Person` code to collect information on the types of variables and return value types of methods. At the end of the first execution of the program, code inserted by the metaobject `dynOnce` can insert in the source code the type of some or all of the variable and method declarations. As an example, suppose object `Person` is used in the following code and only in this code:

```

Person name: "Maria";
Person setAge: 12;
Person print;

```

Code inserted by the metaobject detects in the first run of the program that instance variable `name` has type `String` and `age` has type `Int`. Then at the end of the execution another code inserted by the metaobject `dynOnce` changes the source code transforming it into

```

object Person
  fun init: (String newName, Int newAge) {
    name = newName;
    setAge: newAge
  }
  fun print {
    Out println: "name: ", name, " (" , age, ")"
  }
  fun setAge: ( Int newAge ) {
    if newAge >= 0 and newAge <= MaxAge {
        age = newAge
    }
  }
  String name
  Int age
  private const Int MaxAge = 126
end

```

If it was possible to discover the types of all variables and methods declared without types, the `dynOnce` metaobject call is removed, as in this example. But it may happens that part of the code is not exercised in a single run (or maybe in several or any execution of the program). In this case, a variable that did not receive a value at runtime do not receive a type. And the metaobject call `@dynOnce` is kept in the source code.

There are some questions relating to `dynOnce` that need to be cleared. The most important question is that the implementation of methods with typeless parameters and return value (untyped methods) is

different from the implementation of typed methods. Consequently, message sends to typed methods is different to message sends to untyped methods. Then how do we generate code for the statement

```
p print
```

knowing that `p` was declared as having type `Person`? The problem is that a subclass `Worker` of `Person` may not use `dynOnce` — it may be typed. And it may override method `print`, which means `p print` may call `print` of `Person` (call to an untyped method) or `print` of `Worker` (call to a typed method). The two calls should be different. A solution is:

1. to allow a prototype without `dynOnce` to inherit from a prototype with `dynOnce` but not vice-versa. Then there should be a special `Any` prototype for dynamic typing and there should be conversions between these two `Any` prototypes;
2. in a message send such as “`p print`”, the compiler generates a test to discover whether `p` refers to an object with or without `dynOnce`. Then two different call would be made according to the answer. At compile time, the type of `p`, `Person` was declared with `dynOnce`. Using this information, all message sends whose receiver has type `Person` would have the test we just explained. But if the receiver has a type which was not declared with `dynOnce` the generated code for the message send would be the regular one — it will be assumed that the message receiver can only refer to objects that were declared (or its prototype) without `dynOnce`.

The third mechanism that allows dynamic typing in Cyan is `dynAlways`. A prototype declaration preceded by `@dynAlways` should not use types for variables and methods. All declarations of variables (including parameters, instance variables, and shared variables) and method return values should not mention the type, as in dynamically-typed languages. This metaobject would generate code for message sends appropriately. This metaobject has problems similar to `dynOnce` (which are explained above).

It is important to note that we have not defined exactly neither how `dynOnce` and `dynAlways` will act nor how they will be implemented. This is certainly a research topic.

During the design of Cyan, several decisions were taken to make the language support optional typing:

- (a) types are not used in order to decide how many parameters are needed in a message send. For example, even if method `get:` of `MyArray` takes one parameter and `put:` of `Hashtable` takes two parameters, we cannot write

```
n = Hashtable put: MyArray get: i, j
```

The compiler could easily check that the intended meaning is

```
n = Hashtable put: (MyArray get: i), j
```

by checking the prototypes `Hashtable` and `MyArray` (or, if these were variables, their declared types). `get:` should have one parameter and `put:` should have two parameters. However, if the code above were in a prototype declared with `dynAlways` or `dynOnce`, this would not be possible. The type information would not be available at compile time. Therefore Cyan consider that a message send includes all the selectors that follow it and that are not in an expression within parentheses. To correct the above code we should write:

```
n = Hashtable put: (MyArray get: i), j
```

- (b) when a method is overloaded, the static or compile-time type of the real arguments are not taken into consideration to chose which method will be called at runtime. In the `Animal`, `Cow`, and `Fish` example of page 83, the same methods are called regardless of the static type of the parameter to

`eat`:. Therefore when metaobjects `dynOnce` or `dynAlways` are removed from a prototype (after giving the types of the variables and methods), the semantics of the message sends is not changed.

There is one more reason to delay the search to runtime: the exception system. Most exception handling systems of object-oriented languages are similar to the Java/C++ system. There are catch clauses after a try block that are searched for after an exception is thrown in the block. The catch clauses are searched in the declared textual order. In Cyan, these catch clauses are encapsulated in `eval` methods which are searched in the textual order too. The `eval` methods have parameters which correspond to the parameters of the catch clauses in Java/C++. The `eval` methods are therefore overloaded. The search for an `eval` method after an exception is made in the textually declared order of these methods, as would be made in any message send whose correspondent method is overload. This matches the search for a catch clause of a try block in Java/C++, which appear to be the best possible way of dealing with an thrown exception. And this search algorithm is exactly the algorithm employed in every message send in Cyan;

- (c) the Cyan syntax was designed in order to be clear and unambiguous even without types. For example, before a local variable declaration it is necessary to use “`var`”, which asserts that a list of variables follow, preceded or not by a type. For example, the declaration of `Int` variables in Cyan is

```
var Int a, b, c;
```

In a prototype declared with `dynOnce` or `dynAlways`, this same declaration would be

```
var a, b, c;
```

A method declaration would be

```
fun sqr: (Int x) -> Int { ^ x*x }
```

in a regular prototype and

```
fun sqr: x { ^ x*x }
```

in a prototype declared with `dynOnce` or `dynAlways`.

Chapter 7

Generic Prototypes

Generic prototypes in Cyan play the same rôle as generic classes and template classes of other object-oriented languages. Unlike other modern languages, Cyan takes a loose approach to generics. In many languages, the compiler guarantees that a generic class is type correct if the real parameter is subtype of a certain class specified in the generic class declaration. For example, a generic class `Hashtable` takes a type argument `T` which should be subtype of `Hashable`, an interface with a single method `Int hashCode` (using Cyan syntax). Then whenever one uses `Hashtable<A>` and `A` is subtype of `Hashable`, it is guaranteed that `Hashtable<A>` is type correct — the compiler does not need to check the source code of `Hashtable` to assert that. In Cyan, `Hashtable` has to be compiled with real argument `A` in order to assure the type correctness of the code. This has pros and cons. The pro part is that there is much more freedom in Cyan to create generic prototypes. The con part is that any changes in the code of a generic prototype can cause compile-time errors elsewhere. Cyan does not supports the conventional approach for two reasons: a) there would not be any novelty in it (no articles about it would be accepted for publishing) and b) the freedom given by the definition of Cyan generics makes them highly useful — see the examples given here, in Section 12.5, and in Section 8.

There are several ways of declaring a generic prototype in Cyan. In the first and simplest way, a list of parameters is given between `<` and `>` after the prototype name:

```
package ds

object P< T1, T2, ... Tn >
    ...
end
```

Parameters `T1, T2, ... Tn` are called **formal parameters** of the generic prototype. There should be no space between the prototype name, `P`, and the character “`<`”. Space may follow “`<`” as in

```
package ds

object Stack< T >

    fun push: T elem { ... }
    fun pop -> T { ... }
    fun print {
        array foreach { (: T elem :)
            elem print // message print is sent to an object of type T
        }
    }
}
```



```
...
end
```

After importing package `ds`, that declares `Stack`, one may use `Stack` if an argument is supplied:

```
var Stack<Int> intStack;
var Stack< Person > personStack;
Stack< Stack<Int> > prototypeName print;
```

```
intStack push: 0;
personStack push: aPerson;
```

However, there should be no space between the generic prototype name and “<”. That would cause a compile-time error. If there is a space between the object name and “<”, the compiler will consider “<” as the operator “less than”. Then in the code

```
if Stack < Int > {
  "compile-time error in the line above" println
};
```

the compiler will consider that `Stack` is receiving message “<” with parameter `Int` which is followed by “>”. The Cyan grammar does not allow multiple comparison operators in the same expression (that is, “`a < b < c > d`” is illegal) and “>” demands a parameter, which does not appear in the code above. Therefore there is a compile-time error even before the semantic analysis.

When the compiler finds “`Stack<Int>`” in a source code that imports package `ds`, it creates a brand new prototype whose name is “`Stack<Int>`” by replacing the formal parameter `T` in prototype `Stack<T>` by `Int`. This process is called **instantiation** of a generic prototype and `Int` is called a **real parameter** to the generic prototype. There are restrictions on where a formal parameter can appear and when it is replaced by a real parameter.

A formal parameter may appear as a selector name (both in a method declaration and in a message send), type, identifier in an expression, parameter to a metaobject, after `#` (to define a symbol), and instance variable name. It is illegal to declare any local variable and parameter whose name is a generic prototype parameter. In any other case an identifier equal to a formal parameter is ignored in the process of instantiation of a generic prototype. That is, the formal parameter is not replaced by the real parameter in any other case.

More specifically, the compiler replaces a formal parameter by a real parameter if it is in a symbol literal or it is an `Id` or `IdColon` of the following grammar rules. `Id` in `QualifId` is only replaced if `QualifId` appears in the rules below.

```
QualifId      ::= Id { “.” Id }
ExprPrimary   ::= “self” [ “.” Id ] |
                “super” UnaryId |
                QualifId { “<” TypeList “>” }+ [ ObjectCreation ] |
                QualifId { “<” TypeList “>” }+ |
                “typeof” (“(” QualifId [ “<” TypeList “>” ] “)”)
MetSigUnary   ::= Id
SelecGrammarElem ::= IdColon (“(” Type “)”) ( “*” | “+” )
                IdColon TypeList |

SelecWithParam ::= IdColon |
                IdColon [ “[” ] ParamList
MessageSendNonUnary ::= { [ BACKQUOTE ] IdColon [ RealParameters ] }+
```

```

InterMethSig2 ::= Id |
               { IdColon [ "[" ] [ InterParamDecList ] [ "]" ] }+
CTMOCall      ::= ( "@" | "@@" ) Id
               [ ( "(" | "[" | "{" ) ExprLiteral ( ")" | "]" | "}" ) ]
               [ LeftCharString TEXT RightCharString ]
SingleType    ::= QualifId { "<" TypeList ">" } | BasicType |
               "typeof" "(" QualifId [ "<" TypeList ">" ] ")"

```

Let us see some examples.

```

object Nice< T > extends SuperNice<T> implements InterNice<T>
  fun test {
    // after a # to define a symbol
    symbol = #T;
  }
  // selector name
  fun T: (Char p) -> Int { return 0 }
  // type
  fun add: (T p) -> Int {
    var T x = (T cast: 0) with: main.T;
    return 0
  }
  // type
  T x
end

```

The instantiation of Nice with Person will produce the prototype

```

object Nice< Person > extends SuperNice<Person> implements InterNice<Person>
  fun test {
    // after a # to define a symbol
    symbol = #Person;
  } // selector name
  fun Person: (Char p) -> Int { return 0 }
  // type
  fun add: (Person p) -> Int {
    var Person x = (Person cast: 0) with: main.Person;
    return 0
  }
  // type
  Person x
end

```

Another nice example:

```

object NiceToo< T >
  // parameter to a metaobject call
  @annot(T)
  // unary selector name
  fun T -> Int { return 0 }
end

```

The instantiation of NiceToo with Person will produce the prototype

```

object NiceToo<Person>
  // parameter to a metaobject call
  @annot(Person)
  // unary selector name
  fun Person -> Int { return 0 }
end

```

There is a compile-time error if the formal parameter is the name of a parameter:

```

object Wrong< T >
  fun myError: (Int T) { } // compile-time error
end

```

A formal parameter appearing as a substring of a Cyan symbol is not replaced.

```

object P<T>
  fun print { #T1 print }
end

```

Prototype P<Int> is

```

object P<Int>
  fun print { #T1 print }
end

```

because “T” is just a substring of “T1”.

In the same way, package names and imported packages are not replaced.

```

package T
import main.T
object P<T>
end

```

P<Person> is

```

package T
import main.T
object P<Person>
end

```

Currently there is no way of producing new symbols from formal parameters. There could be a `+++` operator that is executed at compile-time to concatenate formal parameters and something else:

```

object P<T>
  fun print { #T +++ 1 print }
end

```

Prototype P<Int> would be

```

object P<Int>
  fun print { #Int1 print }
end

```

Till now we have found no need for such operators or to compile-time commands such as “static if” of language D.

As said in Chapter 2, the source code of a Cyan source can be a XML file. In this file some information on the source can be kept, as the restrictions that a generic prototype should obey. So the XML file can

keep that the generic parameter `T` of `Hashtable` should have a `Int hashCode` method. In this way the compiler will be able to catch type errors in the instantiations of generic objects, such as to use an object `Person` that does not have a `hashCode` method as parameter to `Hashtable`. This error would be caught without instantiating `Hashtable` with `Person` and compiling the resulting code.

A type `T` may appear before a formal parameter `P` to demand that the real parameter that replaces `P` in an instantiation be a prototype that is subtype of `T`.

```
object P< Writable T >
  fun write: T elem {
    elem write
  }
end
```

```
interface Writable
  fun write
end
```

`P<Proto>` is only legal if `Proto` is a subtype of `Writable`. However, there may be errors in the instantiation of `P<Proto>` even if `Proto` is a subtype of `Writable`. The compiler does not enforce that the operations used inside the generic prototype `P` are only those allowed on subtypes of `Writable`.

A **qualified identifier** is a sequence of Cyan identifiers separated by zero or more dots (“.”) as “`cyan.lang.Int`” or “`Int`” (zero dots). In a generic prototype instantiation, each real parameter should be a *qualified identifier* or a type. This last one can be a generic prototype instantiation. Then the general format of a real parameter is given by rule `RP` of the grammar

```
RP          ::= QualifId | Type
Type       ::= SingleType { | SingleType }
SingleType ::= QualifId { "<" TypeList ">" } | BasicType
TypeList   ::= Type { "," Type }
QualifId   ::= Id { "." Id }
```

Anyway, the real parameter starts with a Cyan identifier. If this identifier starts with an upper-case letter, the compiler considers that the real parameter is a type. Therefore this type should be visible in the place of the generic prototype instantiation or a compile-time error will be signalled. However, the type cannot be a private prototype declared in the same source file in which the generic instantiation is. If the identifier starts with a lower-case letter the compiler does not do any checking in the place of the instantiation.

```
var Stack<A> s; // compiler checks if "A" is a prototype declared or imported
var Stack< Set<Char> > s; // compiler checks if "Set<Char>" is legal
var Nice<myId> n; // compiler does not check if "myId" is a prototype
```

The instantiation “`Wrong<Array>`” causes a compile-time error because there is no prototype “`Array`”.

```
object Wrong<T>
  T<String> myData
  ...
end
```

There is a generic prototype “`Array<T>`” in package `cyan.lang` which is not related to a non-existing non-generic `Array` prototype.

A call to the compile-time function `typeof` cannot be used as a parameter in a generic prototype instantiation.

```
var Int count = 0;
var Stack<typeof(count)> intStack; // compile-time error
```

Because of this restriction, the grammar for RP given above defines `SingleType` differently from the grammar of Section 14.

A generic prototype may declare more than one generic parameter:

```
package ds

object Map<T, U>
  fun key: (T aKey) value: (U aValue) { ... }
  ...
end
```

All formal parameters should have different names. Each of them should start with an upper-case letter and there should be no prototype in package `cyan.lang` with the same name as the parameter. So a parameter cannot have names “`Tuple`” or “`Interval`”.

Currently there is no way of declaring a private generic prototype in Cyan. The implementation of this feature would make the compiler more complex. We believe private generic prototypes would be rarely used and almost never necessary.

7.1 Generic Prototypes with Real Parameters

A prototype that is not generic can be declared using the generic prototype syntax:

```
package ds

object Stack<Int>
  fun push: Int elem { ... }
  fun pop -> Int { ... }
  ...
end
```

There may be both the generic prototype `Stack<T>` and this non-generic version in the same package. In this case, `Stack<Int>` will refer to the non-generic version (the one above) and `Stack<Char>` will be an instantiation of the generic prototype `Stack`. The details of this combination will soon be explained.

We will refer to a non-generic prototype declared using the generic prototype syntax as “**generic prototype with real parameters**”. Each one of the parameters that appear inside `<...>` will be called “**real parameter**”.

A *real parameter* of a *generic prototype with real parameters* can be:

- (a) a single identifier starting with a lower-case letter such as “`t`” or “`add`”. For example,

```
interface ISingle<write>
  fun write: Char
end
```

- (b) a single identifier starting with an upper-case letter such that there is a prototype in package `cyan.lang` with this same name. For example,

```
object P<Int>
  fun add: Int { ... }
end
```

- (c) a qualified identifier; that is, a sequence of identifiers separated by “.” with at least one dot such as “main.Person”. For example,

```
package ds
import main
interface MyList<main.Person>
    fun add: Person
end
```

This qualified identifier should be the full name of a prototype, which includes its package name;

- (d) a generic prototype instantiation possibly preceded by a package name such as “Tuple<Int, String>” or “ds.Stack<main.Person>”. For example,

```
package ds

object List< NTuple<key, String, value, Int>, ds.Map<String, main.Person> >
    ...
end
```

By the above rules, a prototype can be used as a real parameter if it is preceded by its package. This demand is dropped in prototypes of package `cyan.lang`. Therefore if `Person` is in package `main`, a prototype `Stack<main.Person>` should be declared as

```
package ds

object Stack<main.Person>
    fun push: main.Person elem { ... }
    fun pop -> main.Person { ... }
    ...
end
```

In this way the compiler knows whether an identifier that appears after < is a formal parameter or a *real parameter* of a *generic prototype with real parameters*. If the parameter:

- (a) is composed by a single identifier that starts with a lower-case letter it is a *real parameter*. See a previous example of prototype `ISingle` with parameter `write`;
- (b) is composed by a single identifier that starts with an upper-case letter and there is a prototype in `cyan.lang` with this same name, then it is a *real parameter*;
- (c) is composed by a single identifier that starts with an upper-case letter and there is no prototype in `cyan.lang` with this same name, then it is a *formal parameter*;
- (d) is qualified, with at least one “.” in it, then it is a *real parameter*;
- (e) is a generic prototype instantiation, then it is a *real parameter*.

The non-generic version of a generic prototype is a completely independent prototype. It can have different methods, inheritance, and so on. This feature is used to define a prototype `Function<Boolean>` that represents a function that does not take parameters and return a `Boolean`. This kind of function should support methods `whileTrue:` and `whileFalse:`

```

var i = 0;
{ ^ i < 10 } whileTrue: {
    i println;
    ++i
};

```

No other function prototype should have these methods.

A *generic prototype with real parameters* may be useful for providing a more efficient implementation for a given type. For example, a `Hashtable<Int, Int>` implementation could somehow be more efficient because `Int`'s are used.

7.2 Generic Prototype with a Varying Number of Parameters

Generic prototypes with a variable number of parameters are supported. They are declared by putting a `+` after the generic parameter name:

```

object P<T+>
    ...
end

```

There should be just one formal parameter between “<” and “>”.

There is no way to use formal parameter like `T` using regular Cyan syntax. The only way of doing that is through metaprogramming, using metaobjects (Chapter 5). For example, prototype `Tuple` could have been declared as

```

package cyan.lang

public object Tuple<T+>
    @createTuple
end

```

In an instantiation of `Tuple`, as `Tuple<Int, String>`, metaobject `createTuple` has access to the list of real parameter, `Int` and `String`. Based on these parameters, `createTuple` generates Cyan code that replaces the metaobject call. That is, “`@createTuple`” is replaced by declarations of methods and instance variables produced by `createTuple`. Currently `Tuple` is not declared in this way. But this will soon change.

It is tempting to add language constructions to handle a variable number of real parameters. However, that would be a mistake. The number of constructions needed to do something useful would be large. Since this kind of feature will be rarely used, they are best left for metaobjects. It is important to note that several library prototypes of Cyan are or will be implemented using generic prototypes with a varying number of arguments: `Tuple`, `Union`, `Function` etc.

7.3 Multiple Parameter Lists

A generic prototype may have more than one `<...>` list. Inside each list, there may appear more than one parameter as before.

```

package example

object Test<T1, T2><U1, U2>
end

```

It is illegal to mix different kinds of parameters. All parameters should be one of the following:

- (a) real parameters;
- (b) formal parameters without a + operator;
- (c) formal parameters with a + operator.

Then there are three possible ways of declaring a generic prototype:

```
package example

object Test<Int, Char><main.Person>
end
```

```
package example

object Test<T1, T2><U1, U2><R>
end
```

```
package example

object Test<T+><U+>
end
```

There will be a compile-time error if the different kinds of parameters are mixed as in

```
package example

object Test<T+><Int><U> // error
end
```

7.4 Source File Names

Cyan has rules for associating file names to prototypes. As seen, a public prototype P should be in a file called “P.cyan”. A generic prototype with **real parameters**

```
package pack

object P<T1, T2, ... Tn>...<U1, U2, ... Um>
...
end
```

should be in a file

```
P(T1,T2,...Tn)...(U1,U2,...Um).cyan
```

There should be no space in the file name. For example, the source file below should be in file “Test(Int,Char)(main.Person)”.

```
package example

object Test<Int, Char><main.Person>
end
```


The generic prototype

```
package pack

object P<T1, T2, ... Tn>...<U1, U2, ... Um>
...
end
```

should be in file “P(n) ... (m).cyan”. All parameters are formal ones.

The generic prototype

```
package pack

object P<T+>...<U+>
...
end
```

should be in file “P(1+) ... (1+).cyan”.

As examples of declarations and file names, see the table.

P<Int, Char, main.Person>	P(Int,Char,main.Person).cyan
P<R, S, T><U, V><W>	P(3)(2)(1).cyan
P<T+>	P(1+).cyan
P< Tuple<key, String, value, main.Person> >	P(Tuple(key,String,value,main.Person)).cyan
ISingle<write>	ISingle(write).cyan

7.5 Combining Generic Prototypes

A package may declare a non-generic prototype and several generic prototypes with the same name. A generic prototype may have formal parameters, real parameters, or a varying number of parameters. All source files should be in the same package directory which means the source file names are different. There is only one restriction on names: there cannot be a file name “P(n) ... (m).cyan” and “P(1+) ... (1+).cyan” if the number of pairs “()” are equal.

With this last restriction, the prototypes are never confounded. Without it there would be an ambiguity in some cases. For example, if there exists

```
object P<T><U>
```

and

```
object P<T+><U+>
```

then an instantiation P<Int><Char> would be ambiguous. Any prototype could be used.

Suppose an imported package declares several prototypes with name P — at most one is non-generic and the others are generic ones. When the compiler finds an instantiation

```
P<T1, ... Tn>...<U1, ... Um>
```

it tries to find a generic prototype P with real parameters that match exactly the parameters T1, ... Tn, U1, ... Um. If one is not found, the compiler searches for a generic prototype whose number of parameters in each <> list is equal to the instantiation. If none is found, it searches for a generic prototype P with a variable number of parameters with the same number of <> lists as the instantiation. If no adequate generic prototype is found, the compiler signals an error.

For example, suppose that the instantiation is

```
NTuple<key, Int, value, String>
```

First the compiler searches for a prototype `NTuple<key, Int, value, String>` which should be in a file

```
NTuple(key,Int,value,String).cyan
```

If this prototype does not exist, it searches for a generic prototype

```
NTuple<T1, T2, T3, T4>
```

with four formal parameters. This should be in a file

```
NTuple(4).cyan
```

If there is no such prototype, the compiler searches for `NTuple<T+>`

which should be in a file `NTuple(1+).cyan`. Note that there is either `NTuple(4).cyan` or `NTuple(1+).cyan`, never both.

If the instantiation uses other instantiations the process is recursive. In

```
NTuple<key, Tuple<Array<Int>, Union<Int, Char>>> >
```

the compiler searches for a prototype with this name which should be in file `NTuple(key,Tuple(Array(Int),Union`

7.6 Future Enhancements

Cyan does not support *generic methods*. However, it is very probably it will do in the future. We then give a first definition of this construct and show the characteristics it should have in the language.

A generic method is declared by putting the generic parameters after keyword `fun` as in

```
object MySet
  public fun<T> T add: (T elem) { ... }
  ...
end
```

When the compiler finds a message send using `add:` of `MySet`, as in

```
p = MySet add: Person
```

it considers that the return value of `add:` has type equal to the type of the parameter, which is `Person`. Then it checks whether `p` can receive a `Person` in an assignment.

The difference between using a generic method `add:` and declaring a method

```
fun add: (Any elem) -> Any
```

is that the compiler checks the relationships between the parameter and the return value. As another example, a generic method

```
public fun<T> relate: (T first, T second)
```

demands that the arguments to the method be of the same compile-time type.

After the generic parameter there may be a type:

```
public fun<Printable T> T add: (T elem) { ... }
```

Then the real arguments to `add:` should have a static type that is subtype of `Printable`.

Again, it is important to say that generic methods are not adequately defined. The paragraphs above just give an idea of what they can be.

Chapter 8

Important Library Objects

This Chapter describes some important library objects of the Cyan basic library. All the objects described here are automatically imported by any Cyan program. They are in a package called `cyan.lang`.

8.1 System

Prototype `System` has methods related to the runtime execution of the program. It is equivalent to the `System` class of Java. Its methods are given below. Others will be added in due time.

```
// ends the program
fun exit
  // ends the program with a return value
fun exit: (Int errorCode)
  // runs the garbage collector
fun gc
  // current time in milliseconds
fun currentTime -> Long
  // prints the stack of called methods in the
  // standard output
fun printMethodStack
```

8.2 Input and Output

Prototype `In` and `Out` are used for doing input and output in the standard devices, usually the keyboard and the monitor.

```
public object In
  fun readInt -> Int
  fun readFloat -> Float
  fun readDouble -> Double
  fun readChar -> Char
  fun readLine -> String
  ...
end

public object Out
```

```

    fun println: (Any)*
end

```

8.3 Tuples

A tuple is an object with methods for getting and setting a set of values of possibly different types. It is a concept similar to records of Pascal or structs of C. A literal tuple is defined in Cyan between “[.” and “.]” as in the example:

```

var t = [. name: "Lívia", age: 4, hairColor: "Blond" .];
Out println: "name: #{t name} age: #{t age} hair color: #{t hairColor}";

```

This literal object has type `NTuple<name, String, age, Int, hairColor, String>`, described below. There should be no space between the field name such as “name” and the symbol “:”.

A literal tuple may also have unnamed fields which are further referred as `f1`, `f2`, etc:

```

var t = [. "Lívia", 4, "Blond" .];
Out println: "name: #{t f1} age: #{t f2} hair color: #{t f3}";

```

The type of this literal tuple is `Tuple<String, Int, String>`.

Object `NTuple` is a generic object with an even number of parameters. For each two parameters, one describe the field name and the other its type. We will show only the `NTuple` object with four parameters:

```

public object NTuple<F1, T1, F2, T2>

    public fun init: (T1 g1, T2 g2) {
        F1: g1;
        F2: g2
    }

    public fun F1: (T1 g1) F2: (T2 g2) -> NTuple<F1, T1, F2, T2> {
        // create a new object
        var NTuple<F1, T1, F2, T2> t = self clone;
        t F1: g1;
        t F2: g2;
        return t
    }

    @annot( #f1 ) public T1 F1
    @annot( #f2 ) public T2 F2

    public fun copyTo: (Any other) {
    }
end

```

Metaobject `@annot` attaches to an instance variable, shared variable, method, prototype, or interface a feature given by its parameter. This feature can be retrieved at runtime by a method of the introspective library.

After compiling the above prototype, the compiler creates the following instance variables and methods:

```

@annot( #f1 ) private T1 _F1
@annot( #f2 ) private T2 _F2
@annot( #f1 ) fun F1 -> T1 { ^ _F1 }
@annot( #f1 ) fun F1: (T1 newF1) { _F1 = newF1 }
@annot( #f2 ) fun F2 -> T2 { ^ _F2 }
@annot( #f2 ) fun F2: (T2 newF2) { _F2 = newF2 }

```

So we can use this object as in

```

var NTuple<name, String, age, Int> t;
t name: "Carolina" age: 1;
Out println: (t name);
t name: "Livia";
t age: 4;
Out println: "name: #{t name} age: #{t age}";

```

Every object NTuple has a method

copyTo: Any

that copies the tuple fields into fields of the same name of the parameter. For example, consider a book object:

```

object Book
  public String name
  public Array<String> authorList
  public String publisher
  public String year
  fun print {
    Out println: (authorList[0] + " et al. "
    Out println: name + ". Published by " + publisher + ". " + year;
  }
end

```

```

...
// in some other object ...
var b = Book new;
var t = [. name: "Philosophiae Naturalis Principia Mathematica",
  authorList: {# "Isaac Newton" #},
  publisher: "Royal Society",
  year: 1687
.];
t copyTo: b;

```

The last line copies the fields of the tuple into the object fields. That is, “name” of t is copied to “name” of b and so on. The Book object may have more fields than the tuple. But if it has less fields, an exception ExceptionCopyFailure is thrown. Note that copyTo: can be used to copy tuples to tuples:

```

var t = NTuple<name, String, age, Int>;
var maria = [. name: "Maria", age: 4, hairColor: "Blond" .];
maria copyTo: t;

```

Method copyTo: uses reflection in order to copy fields. Since public instance variables are not allowed in Cyan, copyTo: uses the corresponding getters and setters to copy the values.

Method copyTo: creates new objects if any of the tuple fields is another tuple.

```
object Manager
  public Person person
  public String company
end
```

```
object Person
  public String name
  public Int age
end
```

Manager should have a reference to another object, Person. When copying a tuple corresponding to a manager, an object of Person should be created.

```
var manager = Manager new;
var john = [. person: [. name: "John", age: 28 .], company: "Cycorp" .];
john copyTo: manager;
```

Here copyTo: copies field company to object referenced by manager and creates an object of Person making manager.person refer to it.¹ After that copyTo: is called with the tuple

```
[. name: "John", age: 28 .]
```

and this new Person object. It is as if we had

```
var manager = Manager new;
var john = [. person: [. name: "John", age: 28 .], company: "Cycorp" .];
manager.person = Person new;
manager.person.name = "John";
manager.person.age = 28;
manager.company = "Cycorp";
```

Object Tuple is a generic object that takes any number of type parameters (up to 16). It is an unnamed tuple whose elements are accessed by names fi. Then UTuple<T1, T2> is almost exactly the same as

```
NTuple<f1, T1, f2, T2>.
```

In fact, this object is defined as

```
public object Tuple<T1, T2> extends NTuple<f1, T1, f2, T2>
end
```

An example of use of this object is

```
var Tuple<String, Int> t;
t f1: "Ade" f2: 23;
Out println: (t f1);
t f1: "Melissa";
t f2: 29;
Out println: "name: #{t f1} age: #{t f2}";
```

Object Tuple has a method copyTo: that copies the information of the tuple into a more meaningful object. We will shown how it works using the book example. We want to copy a tuple of type

```
Tuple<String, Array<String>, String, Int>
```

into an object of Book. However, copyTo: has to know to which instance variable of Book it should copy

¹In fact, it calls "manager person: (Person new)".

f1 of the tuple. This method cannot choose one instance variable based on the types — there are two of them whose type is `String`. We should use annotations for that:

```
object Book
  @annot( #f1 ) public String name
  @annot( #f2 ) public Array<String> authorList
  @annot( #f3 ) public String publisher
  @annot( #f4 ) public String year
  fun print {
    Out println: (authorList[0] + " et al. "
    Out println: name + ". Published by " + publisher + ". " + year;
  }
end
```

Now the following code will work as expected.

```
var Tuple<String, Array<String>, String, Int> t;
var b = Book new;
t = [. "Philosophiae Naturalis Principia Mathematica",
    {# "Isaac Newton" #},
    "Royal Society",
    1687
    .];
t copyTo: b;
b print;
```

As with method `copyTo:` of `NTuple`, tuples inside tuples are copied recursively. The `Manager` example with `UTuples` is

```
object Manager
  @annot( #f1 ) public Person person
  @annot( #f2 ) public String company
end

object Person
  @annot( #f1 ) public String name
  @annot( #f2 ) public Int age
end
...

var manager = Manager new;
var john = [. [. "John", 28 .], "Cycorp" .];
john copyTo: manager;
// john has the same values as in the example
// with NTuple
```

Method `copyTo:` can be used in grammar methods to store the single method argument into a meaningful object:

```
object BuildBook
  fun (bookname: String (author: String)* publisher: String year: Int)
    Tuple< String, Array<String>, String, Int> t
```

```

        -> Book {
    var book = Book new;
    t copyTo: book;
    return book
}

```

This method accepts as arguments all the important book information: name, authors, publisher, and publication year:

```

var prin = BuildBook bookname: "Philosophiae Naturalis Principia Mathematica"
    author: "Isaac Newton"
    publisher: "Royal Society"
    year: 1687;

```

An empty tuple is illegal:

```

var t = [ . . ]; // compile-time error: empty tuple
var anotherError = [..]; // unidentified symbol '[..]'

```

8.4 Dynamic Tuples

Object `DTuple` is a dynamic tuple. When an object of `DTuple` is created, it has no fields. When a dynamic message “`#attr: value`” is sent to the object, a field whose type is the same as `value` is created. The value of this field can be retrieved by sending the message “`#attr`” to the object. See the example:

```

var t = DTuple new;
t ?name: "Carolina";
    // prints "Carolina"
Out println: (t ?name);
    // if uncommented the line below would produce a runtime error
//Out println: (t age);
t ?age: 1;
    // prints 1
Out println: (t ?age);

```

Object `DTuple` is the object

```

object DTuple mixin AddFieldDynamicallyMixin
end

```

Mixin `AddFieldDynamicallyMixin` redefines method `doesNotUnderstand` in such a way that a field is added dynamically to objects of `DTuple`. When a non-existing method `f: value` is called on the object, `doesNotUnderstand` of `AddMethodDynamicallyMixin` adds to the receiver a field `_f` and methods `f: T` and `T f`. The methods set and get the field. `T` is the type of `value`.

Object `DTuple` has methods `checkTypeOn` and `checkTypeOff` that turn on and off (default) the type checking with dynamic fields:

```

var t = DTuple new;
t ?name: "Carolina";
t ?name: 100; // ok, default off
t ?name: "Carolina"; // ok, default off

```



```
t checkTypeOn;
  // runtime type error here. name: should receive
  // a string as argument
t ?name: 100;
```

Mixin `AddFieldDynamicallyMixin` has a method `remove`: that allows one to the remove a field from a prototype:

```
var t = DTuple new;
t ?name: "Carolina";
t remove: ?name;
  // runtime error in "(t name)"
Out println: (t ?name);
```

The mixin object `AddFieldDynamicallyMixin` is defined as

```
mixin(Any) object AddFieldDynamicallyMixin
  fun doesNotUnderstand: (CySymbol methodName, Array<Any> args) {
    if methodName indexOf: ':' == (methodName size) - 1 {
      if args size != 1 {
        super doesNotUnderstand: methodName, args
      }
      else {
        // add field to table addedFieldsTable and add
        // methods for getting and setting the field
        ...
      }
    }
  }
  fun remove: (CySymbol what) {
    addedFieldsTable remove: what
  }
  Hashtable<CySymbol, Any> addedFieldsTable
end
```

8.5 Intervals

A interval is the return value of methods `..` and `.. $<$` of the types `Byte`, `Short`, `Int`, `Char`, and `Boolean`. Then if `first` and `last` are integers, `first..last` returns an interval with all integers numbers between `first` and `last`, including this last one. And `first.. $<$ last` returns an interval with all integers between `first` and `last - 1` — it is equivalent to `first..(last - 1)`. If `last < first` the return is a valid interval but without elements.

```
var Interval<Int> I;
I = 3..5;
  // this code prints numbers 0 1 2
0..2 foreach: { (: Int i :)
  Out println: i
};
  // this code prints numbers 3 4 5
```

```

I repeat: { (: Int i :)
  Out println: i
};
  // prints the alphabet
'A'..'Z' foreach: {
  (: Char ch :)
  Out println: ch
};
  // false < true
false..true { (: Boolean b :)
  b println
};
var anArray = {# 0, 1, 2, 3 #};
0..<anArray size foreach: { (: Int n :) n println };

```

Operator “..” has smaller precedence than the arithmetical operators and greater precedence than the logical and comparison operators. So, the lines

```

i+1 .. size - 1 repeat: { ... }
if 1..n == anInterval { ... }

```

are equivalent to

```

((i+1) .. (size - 1)) repeat: { ... }
if (1..n) == anInterval { ... }

```

Prototype Interval is defined as follows. Generic parameter T can only be instantiated with types Byte, Short, Int, Long, and Char. Metaobject firstBelongsTo checks that and issue an error if T is not one of these types.

```

package cyan.lang

public object Interval<T> implements Iterable<T>
  @firstBelongsTo(T, Byte, Short, Int, Char)
  fun init: (T start, T theend) {
  fun == (Any other) -> Boolean {
  fun repeat: Function<Nil> b {
  fun foreach: Function<T, Nil> b {
  fun inject: (T initialValue)
  fun to: (T max)
  fun size -> Int { ^ 1 + (Int cast: (theend - start)) }
  fun first -> T { ^start }
  fun last -> T { ^theend }
  fun apply: (String message) -> Dyn {
  fun .* (String message) {
  fun .+ (String message) -> Any {
  private T start, theend
end

```

...

```

// this is declared in cyan.lang
interface Iterable<T>
  fun foreach: Function<T, Nil>
  fun apply: (String message)
  fun .* (String message)
  fun .+ (String message) -> Any
end

abstract object InjectObject<T> extends Function<Nil>
  abstract fun eval: T
  abstract fun result -> T
end

```

Intervals can be used with method `in:` of the basic types:

```

var s String = ""
var Int age = In readInt
if age in: 0..2 {
  s = "baby"
}
else if age in: 3..12 {
  s = "child"
}
else if age in: 13..19 {
  s = "teenager"
}
else {
  s = "adult"
}

```

Chapter 9

Grammar Methods

Cyan supports an innovative way of declaring methods and sending messages: grammar methods and grammar message sends. A grammar method is a method of the form

```
fun (regexpr) T v { ... }
```

in which `regexpr` is given as a regular expression that uses selectors, parameter types, and regular expression operators. There is only one parameter put after the regular expression. The parameter declaration may appear around parenthesis:

```
fun (regexpr) (T v) { ... }
```

Let us introduce this concept in the simplest form: methods that accept a variable number of real arguments. A method `add:` that accepts any number of real arguments that are subtypes of type `T` should be declared as

```
fun (add: (T)+) (Array<T> v) { ... }
```

After `fun` the method keywords should be declared between parentheses. Assuming this method is in an object `MyCollection`, it can be called as in

```
MyCollection add: t1;  
MyCollection add: t1, t2, t3;  
MyCollection add: t2, t1;
```

The `+` means one or more real arguments of type `T` or its subtypes. We could have used `*` instead to mean “zero or more real arguments”. In this case, the call “`MyCollection add: ;`” would be legal. In the call site, all real arguments are packed in an array of type `T` and then it is made a search for an appropriate method `add:`. The formal parameter `v` of `add:` will refer to the array object with the real arguments. The one formal parameter is declared after the parenthesis that closes the declaration of the method signature, which is a regular expression.

The compiler groups all real arguments into one array that is then passed as parameter to the method. It is as if we had

```
MyCollection add: {# t1, t2, t3 #}
```

in message send

```
MyCollection add: t1, t2, t3;
```

As another example, a method `add:` that accepts a variable number of integers as real parameters is declared as

```
object IntSet  
  fun (add: (Int)*) (Array<Int> v) {  
    v foreach: { (: Int elem :)}  
  }
```

```

        addElement: elem
    }
}
fun addElement: Int elem {
    ...
}
...
end

```

What if instead of saying

```
IntSet add: 2, 3, 5, 7, 11;
```

we would like

```
IntSet add: 2 add: 3 add: 5 add: 7 add: 11;
```

? No problem. Just declare the method as

```
object IntSet
    fun (add: Int)+ (Array<Int> v) {
        v foreach: { (: Int elem :)
            addElement: elem
        }
    }
    fun addElement: Int elem {
        ...
    }
    ...
end

```

Here we should use + because we cannot have zero “add: value” elements. Again, in a message send

```
IntSet add: 2, 3, 5, 7, 11
```

the compiler would group all arguments into one array: `IntSet add: {# 2, 3, 5, 7, 11 #}`

More than one keyword may be repeated as in

```
object StringHashtable
    fun (key: String value: String)+
        (Array<NTuple<key, String, value, String>> v) {
        v foreach: { (: NTuple<key, String, value, String> pair :)
            addKey: (pair key) withValue: (pair value)
        }
    }
    fun addKey: (String k) withValue: (String v) {
        ...
    }
    ...
end

```

Part “key: String, value: String” is represented by `NTuple<key, String, value, String>` — see Chapter 8 for the description of object `NTuple`. Since there is a plus sign after this part, the whole method takes a parameter of type

```
Array<NTuple<key, String, value, String>>
```

9.1 Matching Message Sends with Methods

The grammar method of `StringHashtable` defined in the last section can be called by supplying a sequence of `key:value:` pairs:

```
var StringHashtable ht;
ht = StringHashtable new;
ht key: "John" value: "Professor"
    key: "Mary" value: "manager"
    key: "Peter" value: "designer";
```

The last message send would be transformed by the compiler into something like

```
ht key:value: {#
    [. "John", "Professor" .],
    [. "Mary", "manager" .],
    [. "Peter", "designer" .]
#};
```

This is not valid Cyan syntax: although the object passed as parameter is legal, the selector `key:value:` is illegal.

When the compiler reaches the last message send of this example, it makes a search in the declared type of `ht`, `StringHashtable`, for a method that matches the message pattern. This matching is made between the message send and an automaton built with the grammar method. It is always possible to create an automaton from a grammar method since the last one is given by a regular expression. To every regular expression there is an automaton that recognizes the same language.

The compiler may implement the checking of a message send, the search for a method that correspond to it, in the following way:

- (a) every prototype has an automaton for method search. There is just one automaton for every prototype;
- (b) the automaton of a prototype has many final states. At least one for every method, including the inherited ones. And a state without outgoing arrows (transitions) meaning “there is no method for this message”;
- (c) when the compiler finds a message send
“*expr s1: p11, p12, ... sk: pk1, pk2, ... pkm*”
it gives this message as input to the automaton of the prototype or interface `T`, which is the static type of `expr`. If a final state is reached, there is a method in `T` that correspond to this message send. Each final state is associated to a method (including the inherited ones). The no final state is reached, there is no method for this message and the compiler signs an error.

A regular expression is transformed into a non-deterministic automaton which adds ambiguity to message sends in Cyan. For example, consider a method declared as

```
object A
  fun ( (a: Int)* (a: Int)* )
      (Tuple<Array<Int>, Array<Int>> t) { ... }
end
```

A message send

```
A a: 0 a: 1
```

can be interpreted in several different ways:

- (a) `a: 0 a: 1` refer to the first selector of the method. No argument is passed to the second selector. Then `t f1` is an array with two integers, 0 and 1 and `t f2` is an array with zero elements;
- (b) `t f1` has one element and `t f2` has one too;
- (c) `t f1` has zero elements and `t f2` has two integers.

To eliminate ambiguity, Cyan demands that every part of the regular expression of the method matches as much of the message as possible. Therefore the first way shown above will be the chosen by the compiler. Array `t f2` will always have zero elements.

This requirement of “match as much as possible” can be explained using regular expressions. A regular expression `a*a` matches a sequence of `a`'s followed by an `a`. However, a string

```
aaaaa
```

will be matched by the first part of the regular expression, `a*`, without using the last `a` (this is true if we demand “match as much of the input as possible”). Therefore it will not match the regular expression `a*a` because the last `a` will not match any symbol of the input. Conclusion: you should not use a symbol like `a` that is matched by the previous part of the regular expression.

This requirement of “match as much as possible” is necessary to remove ambiguity and to make things work as expected. For example, a regular expression `a*b` (without the quotes) should match the string

```
aab
```

However, this regular expression is transformed into a non-deterministic automaton which can try to recognize its input in several different ways. One of them is to match the first `a` of the input string with the part `a*` of the regular expression, leaving the rest of the string, `ab` to be matched with the rest of the regular expression, `b`. There will be no match and the match fails, demanding a backtracking that would not be necessary if we had used the requirement “match as much of the input string as possible”.

For the time being, it is not possible to override grammar methods in sub-objects. That is, if a grammar method of a superobject accepts a message `M` (considered as input to the automaton corresponding to the method), then no sub-object method can accept `M`. Then the following code is illegal:

```
object MyHash extends StringHashtable
  fun key: (String k) value: (String v) {
    ...
  }
end
```

A message send

```
MyHash key: "University" value: "UFSCar"
```

matches the method defined in `MyHash` and the grammar method of `StringHashtable`.

To avoid matching to selectors `init:` and `new:`, these two selectors are not allowed in grammar methods.

9.2 The Type of the Parameter

When the compiler finds the message `send` that is the last statement of

```
var StringHashtable ht;
ht = StringHashtable new;
ht key: "John" value: "Professor"
  key: "Mary" value: "manager"
  key: "Joseph" value: "designer";
```

it creates a single object of type

```
Array<NTuple<key, String, value, String>>
```

because this is the type of the parameter of the method that matches the pattern of this message send. It knows that three tuple objects should be added to the array and that every tuple should be initialized with the objects following `key:` and `value:`.

Using tuples and arrays to compose the type of the parameter of a grammar method is not generally meaningful. We can use more appropriate objects for that. As an example, the type of the `key:value:` method of `StringHashtable` can be changed to

```
object StringHashtable
  fun (key: String value: String)+ (Array<KeyValue> v) {
    v foreach: { (: KeyValue pair :)
      addKey: (pair key) withValue: (pair value)
    }
  }
  fun addKey: (String k) withValue: (String v) {
    ...
  }
  ...
end
```

Here `KeyValue` is declared as

```
object KeyValue
  @annot( #f1 ) public String key
  @annot( #f2 ) public String value
end
```

The compiler creates and adds to this object the following methods:

```
@annot( #f1 ) fun key: (String newKey) { _key = newKey }
@annot( #f1 ) fun key -> String { ^ _key }
@annot( #f2 ) fun value: (String newValue) { _value = newValue }
@annot( #f2 ) fun value -> String { ^ _value }
```

In a message send like

```
StringHashtable
  key: "John" value: "Professor"
  key: "Mary" value: "manager"
  key: "Joseph" value: "designer";
```

the compiler knows how to pack each group “`key: string value: string`” because of the annotations `#f1` and `#f2` in object `KeyValue`.

It is possible to declare a grammar method without given explicitly the type of the sole parameter:

```
object StringHashtable
  fun (key: String value: String)+ v {
    v foreach: { (: Tuple<String, String> pair :)
      addKey: (pair f1) withValue: (pair f2)
    }
  }
  fun addKey: (String k) withValue: (String v) {
```



```

    ...
}
    ...
end

```

In this case, the compiler will deduce a type for the parameter. It will always use either an array or an unnamed tuple. The type of `v` will be

```
Array< Tuple<String, String> >
```

The Cyan compiler will initially only support this form of declaration. See Chapter 8 for the description of object `Tuple`.

It is possible to declare a selector in a method without any parameters:

```

object MyFile
  fun open: (String name) read: { ... }
  fun open: (String name) write: { ... }
  ...
end

```

Here both methods start with `open:` but they are easily differentiated by the second keyword, which does not take any parameters. This object can be used in the following way:

```

var in = MyFile new;
var out = MyFile new;
in open: "address.txt" read: ;
out open: "newAddress.txt" write: ;
...

```

9.3 Unions and Optional Selectors

Unions are used to compose the type of the parameter of grammar methods that use the regular operator “|”. The signature “A | B” means A or B (one of them but not both).

```

object EnergyStore
  fun (add: (wattHour: Float | calorie: Float | joule: Float))
    (NTuple< add, Any, energy, Union<wattHour, Float, calorie, Float, joule,
      Float> > t) {

    var u = t energy;
    u
    wattHour: {
      // here u is a Float
      amount = amount + u*3600
    }
    calorie: {
      // here u is a Float
      amount = amount + u*4.1868
    }
    joule: {
      // here u is a Float
      amount = amount + u;
    }
  }

```

```

}
    // keeps the amount of energy in joules
    Float amount
    ...
end

```

Any is the type associated to selectors without parameters such as `add:` of this example. Hence “NTuple<add, Any, ...>”. We can use this prototype as

```

EnergyStore add: wattHour: 100.0;
EnergyStore add: calorie: 12000.0;
EnergyStore add: joule: 3200.67;

```

The optional selectors may be repeated as indicated by the “+” in the method declaration:

```

object EnergyStore
  fun ( add: (wattHour: Float | calorie: Float | joule: Float)+ )
      (NTuple< add, Any, energyArray,
        Array<Union<wattHour, Float, calorie, Float, joule, Float>> > t) {

    var v = t energyArray;
    v foreach: { (: Union<wattHour, Float, calorie, Float, joule, Float>> u :)
      u
        wattHour: {
          // here u is a Float
          amount = amount + u*3600
        }
        calorie: {
          // here u is a Float
          amount = amount + u*4.1868
        }
        joule: {
          // here u is a Float
          amount = amount + u;
        }
      }
    }
    // keeps the amount of energy in joules
    Float amount
    ...
end

```

Now we can write things like

```

EnergyStore add:
  wattHour: 100.0
  calorie: 12000.0
  wattHour: 355.0
  joule: 3200.67
  calorie: 8777.0;

```

This is a single method call.

As another example, we can rewrite the MyFile object as

```
object MyFile
  fun ( open: (String name) (read: | write:) )
      (NTuple<open, Any, name, String, access, Union<read, Any, write, Any>> t)
      {

    var u = t access;
    u
      read: {
        // open the file for reading
        ...
      }
      write: {
        // open the file for writing
        ...
      };
    ...
  }
  ...
end
```

Optional parts should be enclosed by parentheses and followed by “?”, as in

```
object Person
  fun ( name: String
        (age: Int)? )
      (NTuple<name, String, age, Union<none, Any, age, Int>> t) {
    _name = t name;
    var u = t age;
    u
      none: {
        _age = -1
      }
      age: {
        // u has type Int here
        _age = u
      }
  }
  ...
  String _name
  Int _age
end
...
var p = Person new;
p name: "Peter" age: 14;
var c = Person new;
  // ok, no age
c name: "Carolina";
```

...

Here it was necessary to use “age” as the name of the second field of the tuple and also as the name of the union. The name of the tuple field should be `age` and it seems that is no better name for the union field than `age` too.

In a grammar method, it is possible to use more than one type between parentheses separated by “|”:

```
object Printer
  fun (print: (Int | String)*) (Array<Union<Int, String>> v) {

    v foreach: { (: Union<Int, String> elem :)

      elem
        unionCase: Int do: {
          // elem has type Int here
          printInt: elem
        }
        unionCase: String do: {
          // elem has type String here
          printString: elem
        }
      };
    }
  }
  // definitions of printInt and printString
  ...
end
```

This method could be used as in

```
Printer print: 1, 2, "one", 3, "two", "three", "four", 5;
```

There is an ambiguity if, when putting alternative types, one of them inherits from the other. For example, suppose `Manager` inherits from `Worker` and there is an object with a method that can accept both a manager and a worker.

```
object Club
  fun (addMember: (Manager | Worker)*) (Array<Union<Manager, Worker>> v) {
    ...
  }
  ...
end
```

A code

```
Club addMember: Manager;
```

is ambiguous because `Manager` can be given as the first or second field of `Union<Manager, Worker>`. To eliminate this ambiguity, Cyan will use the first field that is a supertype of the runtime type of the object that is parameter.

More clearly, suppose an object whose runtime type is `S` is passed as parameter to a method

```
fun (m: T1 | T2 | ... | Tn) (Union<T1, T2, ..., Tn> u) { ... }
```

The runtime system (RTS) will test whether `S` is a subtype of `T1`, `T2`, and so on, in this order. If `S` is subtype of `Ti` and it is not a subtype of `Tj` for $j < i$, the RTS creates an object of `Union<T1, T2, ..., Tn>`

packing the parameter as field `i`.

There is also an ambiguity in methods with alternative selectors such as

```
object Company
  fun (addMember: Manager | addMember: Worker)+ (Array<Union<Manager, Worker>> v) {
    ...
  }
  ...
end
```

The treatment is exactly the same as with union prototypes. The first adequate selector/type combination is used.

9.4 Refining the Definition of Grammar Methods

A grammar method should have a single parameter and it is not necessary to give its type. In the last case, the compiler will associate a type to this parameter to you. To discover this type, the compiler associates a type for each part of the method declaration. The composition of these types gives the type of the single grammar method parameter.

The following table gives the association of rules with types. T_1, T_2, \dots, T_n are types and R is part of the signature of the grammar method. For example, R can be

```
add:
add: Int
at: Int put: String
add: Int | sub: Int
(add: Int)*
```

Whenever there is a list of R 's, assume that the types associated to them are T_1, T_2 , and so on. For example, in a list $R R R$, assume that the types associated to the three R 's are T_1, T_2 , and T_3 , respectively. We used `typeof(S)` for the type associated, by this same table, to the grammar element S .

rule	type
T_1	T_1
$R R \dots R$	$\text{Tuple}\langle T_1, T_2, \dots, T_n \rangle$
Id <code>“.”</code>	Any
Id <code>“.”</code> T	T , which must be a type
Id <code>“.”</code> <code>“(”</code> T <code>“)”</code> <code>“*”</code>	$\text{Array}\langle T \rangle$
Id <code>“.”</code> <code>“(”</code> T <code>“)”</code> <code>“+”</code>	$\text{Array}\langle T \rangle$
<code>“(”</code> R <code>“)”</code>	<code>typeof</code> (R)
<code>“(”</code> R <code>“)”</code> <code>“*”</code>	$\text{Array}\langle \text{typeof}(R) \rangle$
<code>“(”</code> R <code>“)”</code> <code>“+”</code>	$\text{Array}\langle \text{typeof}(R) \rangle$
<code>“(”</code> R <code>“)”</code> <code>“?”</code>	$\text{Union}\langle \text{typeof}(R) \rangle$
T_1 <code>“ ”</code> T_2 <code>“ ”</code> ... <code>“ ”</code> T_n	$\text{Union}\langle f_1, T_1, f_2, T_2, \dots, f_n, T_n \rangle$
R <code>“ ”</code> R <code>“ ”</code> ... <code>“ ”</code> R	$\text{Union}\langle f_1, T_1, f_2, T_2, \dots, f_n, T_n \rangle$

We will give now the precise definition of the type of a grammar method based on the grammar of it. It will be used `“typeof(P)”` for the type associated to the grammar production P .

The productions will be divided in cases.

```
SelectorGrammar ::= “(” SelectorUnitSeq “)” [ “*” | “+” ]
```

```

In this case, typeof(SelectorGrammar) = Array<typeof(SelectorUnitSeq)>
SelectorGrammar ::= "(" SelectorUnitSeq ")" [ "?" ]
Now typeof(SelectorGrammar) = Union<missing, Any, present, typeof(SelectorUnitSeq)>
SelectorUnitSeq ::= SelectorUnit { SelectorUnit }
typeof(SelectorUnitSeq) = Tuple<typeof(SelectorUnit1), ..., typeof(SelectorUnitn)>

```

in which `typeof(SelectorUniti)` is the i^{th} production.

```
SelectorUnitSeq ::= SelectorUnit { "|" SelectorUnit }
```

```
typeof(SelectorUnitSeq) = Union<f1, typeof(SelectorUnit1), ...,
                          fn, typeof(SelectorUnitn)>
```

in which `typeof(SelectorUniti)` is the i^{th} production.

```

SelectorUnit ::= SelecGrammarElem
typeof(SelectorUnit) = typeof(SelecGrammarElem)
SelectorUnit ::= SelectorGrammar
typeof(SelectorUnit) = typeof(SelectorGrammar)
SelecGrammarElem ::= IdColon
typeof(SelecGrammarElem) = Any
SelecGrammarElem ::= IdColon TypeList
typeof(SelecGrammarElem) = typeof(TypeList)
SelecGrammarElem ::= IdColon "(" Type ")" ( "*" | "+" )
typeof(SelecGrammarElem) = Array<typeof(Type)>
TypeList ::= Type { "," Type }
typeof(TypeList) = Tuple<Type1, Type2, ..., Typen>

```

Let us see some examples of associations of signatures of grammar methods with types:

Int	Int
add: Int	Int
add: Int, String	Tuple<Int, String>
add: (Int)*	Array<Int>
add: (Int)+	Array<Int>
(add: Int)*	Array<Int>
(add: Int)+	Array<Int>
(add: Int String)	Union<Int, String>
(add: (Int String)+)	Array<Union<Int, String>>
(add: Int add: String)	Union<Int, String>
key: Int value: Float	Tuple<Int, Float>
nameList: (String)* (size: Int)?	Tuple<Array<String>, Union<Int>>
coke:	Any
coke: guarana:	Union<Any, Any>
(coke: guarana:)*	Array<Union<Any, Any>>
(coke: guarana:)+	Array<Union<Any, Any>>
((coke: guarana:)+)?	Union<Array<Union<Any, Any>>>
((coke: guarana:)?)+	Array<Union<Union<Any, Any>>>
amount: (gas: Float alcohol: Float)	Tuple<Any, Union<Float, Float>>

By the above grammar, it is possible to have a method

```

fun (format: (String form) print: (String s))
    (Tuple<String, String> t) {
    ...
}

```

which starts with “(” (after keyword `fun`) but which does not use any regular expression operator. This is legal. As usual, all the parameter are grouped into a one, a tuple, declared as the single parameter.

Conceptually, every Cyan method takes a single parameter whose type is given by the associations of the above table. For example a method

```

fun at: (Int x, y) print: (String s) { ... }

```

conceptually takes a single parameter of type `Tuple< Tuple<Int, Int>, String >`.

The method name of a grammar method is obtained by removing the spaces, parameter declaration, and types from the method. Then, the method names of

```

fun ( add: (wattHour: Float | calorie: Float | joule: Float)+ )
    (NTuple< add, Any, energyArray, Array<NUnion<wattHour, Float, calorie,
        Float, joule, Float>> > t)
fun ( name: String
    (age: Int)? )
    (NTuple<name, String, age, NUnion<age, Int>> t) {

```

are

```

(add:(wattHour:|calorie:|joule:)+)
(name:(age:)?)

```

9.5 Context-Free Languages

Although regular expressions are used to define grammar methods, some context-free languages can be incorporated into a grammar method with the help of parentheses. Let us see an example. The grammar below uses `{` and `}` to means repetition of zero ou more times and anything between quotes is a terminal. Then `EList` derives zero or more `E`'s.

```

L ::= "(" EList ")"
EList ::= { E }
E ::= L | N
N ::= a integer number

```

This represents a Lisp-like list of integers:

```

() (0 1) ((0 1) (2) )

```

This grammar is not regular and cannot be converted into a regular grammar. There is no regular expression whose associated language is the same as the language generated by this grammar. The problem here is that `L` is defined in terms of `EList`, which is defined in terms of `E`, which is defined in terms of `L` and `N`. Therefore, `L` is defined in terms of itself. There is no way of removing this self reference: no grammar method will be able to represent this grammar. However, assuming prototype `List` keeps a list of integers, this grammar is representable through a grammar method. It is only necessary that some methods return a `List` object.

```

object List
    public Int value

```

```

    public List next
end

object GenList
  fun (L: (List | Int)* ) (Array<Union<List, Int>> t) -> List {
    // here parameter t is converted into a list object
    ...
  }
end

```

We will just have to use parentheses to delimit the construction of a list in terms of another list. Here the arguments that follow L: should be integers or objects of List (produced by any means, including by the return value of another call to this method). Therefore lists (1, 2, 3) and

```
( 1, (2, 3), 4)
```

are produced by

```

var a = GenList L: 1, 2, 3;
var b = GenList L: 1, (GenList L: 2, 3), 4;

```

init or new methods cannot be grammar methods — that could change, since there is no technical reason they should not. The prohibition is because the introduction of this feature would make the language more complex. If at least init or new methods could accept a variable number of parameters, we could have a code like

```
var b = List(1, List(2, 3), 3)
```

to create the list (1, (2, 3), 4). That would be much better than the previous example.

It is expected that there will be a prototype CyanCode capable of generating the whole of Cyan grammar. The grammar methods of CyanCode would return objects of the Abstract Syntax Tree of Cyan. These objects could be used in the reflection library (metaobjects of compile and runtime). In this way, meta-programming would be rather independent from a particular AST. That is, the AST would exist but the meta-programmer would not manipulate it directly.

We will show one more example on how to implement a context-free grammar (CFG) that is not regular (RG) using grammar methods. A CFG that is not regular defines some non-terminal (also called “variable”) in terms of itself. This necessarily happens. If it does not, there is a regular expression that produces the same language as the CFG.

The non-terminal defined in terms of itself should be associated to one grammar method that returns an object representing that non-terminal. For example, if the non-terminal is S and there is a prototype SObj that represents the non-terminal S, then the grammar method for S should return an object of SObj (which is the prototype of the Abstract Syntax Tree that keeps the data associated to S). Now any references to S in the grammar method should be replaced by a parameter of type SObj. Let us study the example below.

Suppose S is defined in terms of A which is defined in terms of S:

```

S ::= N A | C B
A ::= N S | C
B ::= N A | C
N ::= a number
C ::= a char

```

It is always possible to change the grammar (preserving the language it generates) in such a way that S is defined in terms of S and A in terms of S:


```

S ::= N N S | N C | C
A ::= N S | C
B ::= N A | C
N ::= a number
C ::= a char

```

Then we proceed as before to implement this grammar, using `first:`, `second:`, and `third:` to differentiate the grammar rules used in a message send. In `A ::= X | Y`, the associated grammar method is

```
fun (A: (first: X | second: Y)) t { }
```

in which the grammar rule, `A:`, is used as a selector.

The implementation of the grammar above follows.

```

object S ... end
object GenS
  fun ( S: (first: Int, Int, S | second: Int, Char | third: Char) ) t -> S { ... }
  fun ( A: (first: Int, S | Char) ) t -> A { ... }
  fun ( B: (first: Int, A | second: Char) ) t -> B { ... }
end

```

A string

```
0 1 2 'A'
```

can be derived from the grammar through the derivations

$$S \implies N N S \implies N N N C \xRightarrow{*} 0 1 2 'A'$$

The string `0 1 2 'A'` can be given as input to `GenS` through the message sends

```
GenS S: first: 0, 1, (GenS S: second: 2, 'A');
```

It is very important to note that there is an interplay between grammar terminals and Cyan literals. Here `N` means “any number” in the grammar. In the grammar method, we use `Int` in place of `N`, which matches a literal number of Cyan such as `0`, `1`, and `2`. The same happens with `Char` and `C`. Therefore we assumed that integers in Cyan are the same thing as integers in this grammar.

9.6 Default Parameter Value

Grammar methods can be used to implement default values for parameters. One should use

```
selector: T = defaultValue
```

for a parameter of type `T` following `selector:` that has a default value `defaultValue`. A grammar method with at least one parameter with default value cannot use the regular operators `+` and `*`.

```

object Window
  fun (create: x1: Int
        y1: Int
        (width: Int = 300)?
        (height: Int = 100)?
        (color: Int = CyanColor)?
      ) t {
    ...
  }
  ...
  public const Int Cyan = 00ffffHex

```

```
end
```

When the calling code do not supply the width, height, or color, the compiler initialize the parameter `t` with the values given in the declaration:

```
    // this call is the same as
    // Window create: x1: 0 y1: 0 width: 300 height: 150 color: CyanColor;
var w = Window create: x1: 0 y1: 0 height: 150;
    // this call is the same as
    // Window create: x1: 0 y1: 0 width: 300 height: 100 color: 0000ffHex;
var p = Window create: x1: 100 y1: 200 color: 0000ffHex;
```

Currently, there is one limitation in the use of default parameters: inside the optional part in the declaration of the grammar method, there should be only one parameter and one selector. Therefore it is illegal to declare

```
object Window
  fun (create: x1: (Int aX1)
      y1: (Int aY1)
      // error: two selectors
      (width: Int = 300 height: Int = 100)?
      // error: three parameters
      (rgbcolor: Int = 0, Int = 255, Int 255)?
    ) t {
    ...
  }
  ...
end
```

Note that one can declare default parameters using “or” as in

```
object EnergySpending
  fun (add: (wattHour: Float (hours: Int = 1)? ) |
      (calorie: Float (amount: Int = 1)? ) |
      (joule: Float (amount: Int = 1)? ) )
    ) t {
    ...
  }
  ...
end
```

A future improvement (or not ...) of the language would be to allow named parameters in grammar methods that do not use “|”, “*”, or “+” in their definitions and that have default values for parameters inside optional expressions (with “?”). So it would be legal to declare

```
object Window
  fun (create: x1: (Int aX1)
      y1: (Int aY1)
      (width: Int aWidth = 300)?
      (height: Int aHeight = 100)?
      (color: Int aColor = Cyan)?
    ) t {
```

```

        ) {
    x1 = aX1;
    y1 = aY1;
    width = aWidth;
    height = aHeight;
    color = aColor;
}
...
Int x1, y1
Int width
Int height
Int color
public const Int Cyan = 00ffffHex
end

```

This would make it easy to access the parameters with default values.

9.7 Domain Specific Languages

Grammar methods make it easy to implement domain specific languages (DSL). A small DSL can be implemented in Cyan in a fraction of the time it would take in other languages. The reasons for this efficiency are:

- (a) the lexical analysis of the DSL is implemented using grammar methods is the same as that of Cyan;
- (b) the syntactical analysis of the DSL is given by a regular expression, the signature of the grammar method, and that is easy to create;
- (c) the program of the DSL is a grammar message send. The Abstract Syntax Tree (AST) of such a program is automatically built by the compiler. The tree is composed by tuples, unions, arrays, and prototypes that appear in the definition of the grammar method. The single method parameter refer to the top-level object of the tree;
- (d) code generation for the DSL is made by interpreting the AST referenced by the single grammar method parameter. Code generation using AST's is usually nicely organized with code for different structures or commands being generated by clearly separated parts of the compiler;
- (e) it is relatively easy to replace the type of the single parameter of a grammar method by a very meaningful type of the AST. So, instead of using

```
Array<Tuple<Int, Int>>
```

one could use

```
Graph
```

in which `Graph` is a prototype with appropriate methods.

To further exemplify grammar methods, we will give more examples of them.

```

object Edge
  @annot( #f1 ) public Int from
  @annot( #f2 ) public Int to
end

```

```

object Graph
  @annot( #f1 ) public Int numVertices Int
  @annot( #f2 ) public Int edgeArray Array<Edge>
end

object MakeGraph
  fun ( numVertices: Int (edge: Int, Int)* ) (Graph t) -> Graph {
    ^t
  }
end

```

A call

```

var g = MakeGraph numVertices: 5
      edge: 1, 4
      edge: 3, 1
      edge: 1, 2
      edge: 2, 4;

```

would produce and return an object of type `Graph` properly initialized. Note that the grammar method of `MakeGraph` just return the method argument. This is a simple trick to produce an AST from a message send.

A small language with an if, list of commands (c1), assignment, while, and print statements is implemented by the following grammar methods:

```

object GP
  fun (if: Expr then: Stat (else: Stat)? |
      c1: (Stat)* |
      assign: String, (Expr | String | Int) |
      while: (Expr | String | Int) do: Stat |
      print: (Expr | String | Int)
      ) t
    -> Stat {
      // code to convert t into an AST object
    }
  fun (add: (Expr | String | Int), (Expr | String | Int) |
      mult: (Expr | String | Int), (Expr | String | Int)|
      lessThan: (Expr | String | Int), (Expr | String | Int)) t
    -> Expr {
      // code to convert t into an AST object
    }
end

```

It is assumed that variables in this language are automatically declared when used. Program

```

i = 0;
soma = 0;
while ( i < 10 ) {
  soma = soma + i;
  i = i + 1;
};
print soma

```

is represented by the following message send:

```
var program = GP cl:
  (GP assign: "i", 1),
  (GP assign: "soma", 0),
  (GP while: (GP lessThan: "i", 10)
    do: (GP cl: (GP assign: "soma", (GP add: "soma", "i") ),
      (GP assign: "i", (GP add: "i", 1) )));
program run: Hashtable();
```

The last message send would call a method to execute the program. Assume there is an abstract prototype Stat with sub-prototypes IfStat, StatList, AssigStat, and WhileStat. Each one of them has a run method that takes a hashtable as parameter. This hashtable holds the variables names and values. As an example, prototype IfStat would be as follows.

```
object IfStat extends Stat
  fun run: (Hashtable h) {
    // if variable is not in the table
    // it is inserted there
    h key: variable value: (expr run: h)
  }
  @annot( #f1 ) public String variable
  @annot( #f2 ) public Expr expr
end
```

Of course, the methods of prototype GP could just return the parameter as the Graph prototype if Stat, Expr, and other prototypes of the AST are properly annotated.

The possibilities of defining DSL's with grammar methods are endless. For example, one can define a grammar method for creating XML files:

```
var String xmlText = XMLBuilder root: "booklist"
  elem: "book" contain: (XMLElem elem: "author" contain: "Isaac Newton"),
    (XMLElem elem: "title" contain: "Philosophiae Naturalis
      Principia Mathematica"),
    (XMLElem elem: "year" contain: "1687")
  elem: "book" contain: (XMLElem elem: "author" contain: "Johann Carl Friedrich Gauss
    "),
    (XMLElem elem: "title" contain: "Disquisitiones Arithmeticae"),
    (XMLElem elem: "year" contain: "1801");
```

This method call would return the string

```
<booklist>
  <book>
    <author> Isaac Newton </author>
    <title> Philosophiae Naturalis Principia Mathematica </title>
    <year> 1687 </year>
  </book>
  <book>
    <author> Johann Carl Friedrich Gauss </author>
    <title> Disquisitiones Arithmeticae </title>
    <year> 1801 </year>
```

```
</book>
</booklist>
```

SQL queries could also easily be given as calls to grammar methods. Any syntax error would be discovered at compile-time. Horita [?] has designed a set of grammar methods for building graphical user interfaces. There are methods for building menus, buttons, etc. It is much easier to use grammar methods for GUI than to compose them by explicitly creating objects and calling methods. A similar approach for building user interfaces is taken by the `SwingBuilder` of language Groovy (page 132 of [Dea10]). Groovy builders are commented in Section 9.8.

Flower [flo12] gives an example of a DSL used to control a camera which is in fact a window of visibility over a larger image. As an example, we can have a 1600x900 image but only 200x100 pixels can be seen at a time (this is the camera size). Initially the “camera” shows part of the image and a program in the DSL moves the camera around the larger image, showing other parts of it. The DSL grammar is

```
<Program> ::= <CameraSize> <CameraPosition> <CommandList>
<CameraSize> ::= "set" "camera" "size" ":" <number> "by" <number> "pixels" "."
<CameraPosition> ::= "set" "camera" "position" ":" <number> "," <number> "."
<CommandList> ::= <Command>+
<Command> ::= "move" <number> "pixels" <Direction> "."
<Direction> ::= "up" | "down" | "left" | "right"
```

`CameraSize` is the size of the window visibility of the camera. `CameraPosition` is the initial position of the camera in the larger image (lower left point of the window). `CommandList` is a sequence of commands that moves the camera around the larger image. The site [flo12] shows an animation of this.

A grammar method implementing the above grammar is very easy to do:

```
object Camera
  fun (sizeHoriz: Int sizeVert: Int
      positionX: Int positionY: Int
      (move: Int (up: | down: | left: | right:)) + ) t {
    // here comes the commands to actually change the camera position
  }
end
```

This method could be used as

```
Camera sizeHoriz: 1600 sizeVert: 900
      positionX: 0 positionY: 0
      move: 100 up:
      move: 200 right:
      move: 500 up:
      move: 150 left:
      move: 200 down;
```

It takes seconds, not minutes, to codify the signature of this grammar method given the grammar of the DSL. Other easy-to-do examples are a Turing machine and a Finite State Machine.

A future work is to design a library of grammar methods for parallel programming that would implement some common parallel patterns. We could have calls like:

```
Process par: { Out println: 0 }, { Out println: 1 }
      seq: { Out println: 2 }, { Out println: 3 }
      par: (Graphics getMethod: "convert"), (Printer getMethod: "print");
```

Functions after `par:` would be executed in any paralel. Functions after `seq:` would be executed in the order they appear in the message send. Then 1 may appear before 0 in the output. But 2 will always come before 3. Remember methods are u-functions.

9.8 Groovy Builders

The excelent language Groovy [Dea10] supports a feature called “builders” that makes it easy to construct domain specific languages or tree-like structures. It would be very nice if Cyan had something similar. We tried to add an equivalent feature without introducing new language constructs. That was not possible. However, it is possible to define builders using dynamically typed message sends. To generate a html page, one can write in Groovy [Dea10]:

```
def html = new groovy.xml.MarkupBuilder()
html.html {
  head {
    title "Groovy Builders"
  }
  body {
    h1 "Groovy Builders are cool!"
  }
}
```

In Cyan, one could design a MarkupBuilder prototype that plays a rôle similar to the Groovy class:

```
var b = MarkupBuilder new;
b.html: {
  b head: {
    b title: "Groovy Builders"
  };
  b body: {
    b h1: "Groovy Builders are cool!"
  }
};
```

However, it is undeniable that the Groovy code is more elegant. The Cyan code does not look like a tree-like structure as the Groovy code because it is necessary to send messages to the `b` variable.

Context functions (Section 10.11) can be used to make Cyan more Groovy-like:

```
var b = MarkupBuilder new;
b.html: {
  (: MarkupBuilder self :)
  head: {
    (: HeadBuilder self :)
    title: "Groovy Builders"
  };
  body: {
    (: BodyBuilder self :)
    h1: "Groovy Builders are cool!"
  }
};
```

Prototype `MarkupBuilder` defines methods `html:`, `head:`, and `body:`. Method `html` calls the context function after initializing `self` to its own `self` (which is `b`). Then the call to `head:` is in fact “`b head: ...`” (idem for `body:`). Method `head:` of `MarkupBuilder` accepts a context function as parameter and sets the `self` of this context function to an object of prototype `HeadBuilder`. This prototype defines a method `title:`. Idem for `h1:` of `BodyBuilder`. An alternative implementation would use prototype `MarkupBuilder` in place of `HeadBuilder` and `BodyBuilder`. In this case, the code above would be almost equal to the Groovy code:

```
var b = MarkupBuilder new;
b html: {
    (: MarkupBuilder self :)
    head: {
        title: "Groovy Builders"
    };
    body: {
        h1: "Groovy Builders are cool!"
    }
};
```

This Groovy-like Builder implementation in Cyan has the advantage of being compile-time checked. Any mistakes in the tree building would be caught by the compiler:

```
var b = MarkupBuilder new;
b html: {
    (: MarkupBuilder self :)
    head: {
        (: HeadBuilder self :)
        // compile-time error
        // a HeadBuilder don't have a
        // method h1:
        h1: "Groovy Builders are cool!"
    };
    body: {
        (: BodyBuilder self :)
        // compile-time error
        // a BodyBuilder don't have a
        // method title:
        title: "Groovy Builders"
    }
};
```

One can define a method `sendToReceiver` in prototype `Builder` to make it easy to call methods of this prototype using a function. This method instantiates a context-object parameter with `self`. It should be declared as:

```
fun sendToReceiver: (ContextFunction<Builder, Nil> b) {
    var Function<Nil> function = b bindToFunction: self;
    function eval;
}
```

This method can be used to simulate Groovy builders.


```

var b = Builder new;
b sendToReceiver: { (: Builder self :)
  book: {
    author: {
      firstName: "Isaac";
      surname: "Newton";
    };
    title: "Philosophiae Naturalis Principia Mathematica"
  }
};

```

The compiler would check whether the message sends to `self` inside this context function refer to methods declared in prototype `Builder`. Then `book: Function<Nil>`, `author: Function<Nil>`, and so on should be methods of `Builder`.

Language Ruby has a method that plays the same rôle as `sendToReceiver`. It is `instance_exec`.

In Cyan methods and fields can be added to objects (Section 8.4). Using this feature, a prototype `Builder` could allow the definition of tree-like structure whose node names are not defined at compile-time. That would be very useful for defining a XML builder for example.

```

var XMLBuilder xml;
xml sendToReceiver: {
  (: XMLBuilder self :)
  root: "booklist";
  ?book:
    ?author: "Isaac Newton"
    ?title: "Philosophiae Naturalis Principia Mathematica"
    ?year: "1687";
  ?book:
    ?author: "Johann Carl Friedrich Gauss"
    ?title: "Disquisitiones Arithmeticae"
    ?year: "1801";
}

```

This method call would build a XML code in the form of an abstract syntax tree (AST), which is an internal representation of the XML code. When a message “`asString`” is sent to the `xml` variable, the string returned would be

```

<booklist>
  <book>
    <author> Isaac Newton </author>
    <title> Philosophiae Naturalis Principia Mathematica </title>
    <year> 1687 </year>
  </book>
  <book>
    <author> Johann Carl Friedrich Gauss </author>
    <title> Disquisitiones Arithmeticae </title>
    <year> 1801 </year>
  </book>
</booklist>

```

The only method `XMLBuilder` has is `root` (plus possible some auxiliary methods). When message `?book: ?author: ... ?title: ... ?year`

is sent to `xml`, method `doesNotUnderstand` create objects to represent the XML element

```
<book>
  <author> Isaac Newton </author>
  <title> Philosophiae Naturalis Principia Mathematica </title>
  <year> 1687 </year>
</book>
```

That would be made with every non-checked dynamic message send (those starting with `?`).

9.9 A Problem with Grammar Methods

There is a problem with grammar methods, related to functions, that can only be properly understood after reading Chapter 10.

```
object FunctionBox
  fun (add: Function<Nil>)* (Array<Function<Nil>> t) {
    b = t[0];
  }
  fun do {
    b eval;
  }
  Function<Nil> b;
end
...
if 0 < 1 {
  var Int i = 0;
  FunctionBox add: { ++i }
}
FunctionBox do;
```

Here function `{ ++i }` is passed as a parameter to method `add:` which assigns the function to instance variable `b`. When method `do` is called in the last line, `b` receives message `eval` which causes the execution of the function which accesses variable `i` causing a runtime error: this variable does not exist anymore. However, this error will never happens because: a) there cannot exist object `Array<Function<Nil>>` and b) assignment “`b = t[0]`” is illegal because r-functions cannot be assigned to instance variables. However, the restrictions on the use of functions limit too much the use of grammar methods taking functions as parameters. A `switch` grammar method declared as below would cause a compile-time error.

```
fun (
  (case: (T)+ do: Function<Nil>)+
  (else: Function<Nil>)?
) (Tuple<Any, Array<Tuple<Array<T>, Function<Nil>>>, Union<Function<Nil>>> t)
  {
    // method body
  }
}
```

It is necessary to use `Function<Nil>` instead of `UFunction<Nil>` to allow access to local variables:

```
var String lifePhase;
```

```

n
case: 0, 1, 2 do: {
    lifePhase = "baby"
}
case: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 do: {
    lifePhase = "child"
}
case: 13, 14, 15, 16, 17, 18, 19 do: {
    lifePhase = "teenager"
}
else: {
    lifePhase = "adult"
};

```

There are several solutions to this problem, none of them ideal:

- (a) change the definition of functions. Any local variable used inside a function without % is allocated in the heap. This causes performance problems but otherwise this is an ideal solution. Local variables are usually allocated in the stack which is very fast. Note that both the variables and the objects they refer to would be heap-allocated;
- (b) restrict the way the single parameter `t` of a grammar method can be used. This can take two forms. In the first one, any assignment from and to any field of `t`, even the nested ones, should be prohibited if there is any `Function` type appearing in the parameter type. The assignments are made using methods. So, if `t` has type

```

Tuple<Tuple<Array<Int>, Float>, Union<Int, Tuple<String, Function<Nil>>>>

```

then the message sends of the code below would be illegal.

```

1    t = u;
2    t f1: aif;
3    aNumber = (t f1) f2;
4    t f2: newUU;
5    newUU = t f2;
6    (t f2) f2: c;
7    k = ((f f2) f2) f2;
8    ((f f2) f2) f2: = gg;

```

That is too radical but simple. It is this restriction that is adopted by Cyan.

A less restrictive rule would be to prohibit assignments only in the path from `t` to any type `Function`. In this case, assignments of lines 2 and 3 would be legal.

Section h (page 196) makes a proposal for correcting this problem. For short, `Array<T>` will be a restricted array whenever `T` is a restricted function. This should probably work.

9.10 Limitations of Grammar Methods

There are two limitations of grammar methods:

- (a) polymorphism does not apply to them because all grammar methods are implicitly “final”. It is illegal to redefine a grammar method in a sub-prototype. So one cannot have multiple implementations of a Domain Specific Language and select one of them dynamically using a message send;

(b) grammar methods cannot be declared in an interface.

There is no technical problem (till I know) in removing these limitations. They only exist to make the compiler simpler. Probably they will be lifted in the future.

Chapter 10

Functions

Functions of Cyan are similar to blocks of Smalltalk or anonymous functions of other languages. A function is a literal object — an object declared explicitly, without being cloned of another object. A function may take arguments and can declares local variables. The syntax of a literal function is:

```
{ (: ParamRV :) code }
```

`ParamRV` represents the declaration of parameters and the return value type (optional items). A function is very similar to a method definition — it can take parameters and return a value. For example,

```
b = { (: Int x -> Int :) ^ x*x };
```

declares a function that takes an `Int` parameter and returns the square of it. Symbol `^` is used for returning a value. However, to `b` is associated a function, not a return value, which depends of the parameter. Functions are objects and therefore they support methods. The function body is executed by sending to the message `eval:` with the parameters the function demands or `eval` if it does not take parameters. For example,

```
y = b eval: 5;
```

assigns 25 to variable `y`. The `eval:` methods are similar to Smalltalk's `value` methods. We have chosen a method name different from that of Smalltalk because in Cyan a function may not return a value when evaluated. In Smalltalk, it always does.

The function `{ (: Int x :) ^ x*x }` is similar to the object

```
object LiteralFunction001
  fun eval: (Int x) -> Int {
    ^ x*x;
  }
end
```

For every function the compiler creates a prototype like the above. Then two identical functions give origin to two different prototypes. There are important differences between the function and this prototype which will be explained in due time.

The return value type of a function can be omitted. In this case, it will be the same as the type of the return value of the expression returned — all returned values should be of the same type. For example,

```
{ (: Int x, Int y :)
  var Int r;
  r = sqrt: ((x-x0)*(x-x0) + (y-y0)*(y-y0));
  ^ r }
```

declares a function which takes two parameters, `x` and `y`, declares a local or temporary variable `r`,¹ and

¹Which of course can easily be removed as the function can return the expression itself.

returns the value of `r` (therefore the return value type is `Int`). Assume that this function is inside an object which has a method called `sqrt`. Variables `x0` and `y0` are used inside the function but they are neither parameters nor declared in the function. They may be instance variables of the object or local variables of functions in which this literal function is nested. These variables can be changed in the function.

The language does not demand that the return value type of a function be declared. In some situations, the compiler may not be able to deduce the return type:

```
var b = { ^b };
```

To prevent this kind of error, when a function is assigned to a variable `b` in its declaration, as in this example, `b` is only considered declared after the compiler reaches the beginning of the next statement. Then in this code the compiler would sign the error “`b` was not declared”. In the general case, in an assignment “`var v = e`” variable `v` cannot be used in `e`.

Sometimes a function should return a value for the method in which it is instead of returning a value for the call to `eval` or `eval`. For example, in an object `Person` that defines a variable `age`, method `getLifePhase` should return a string describing the life phase of the person. This method should be made using keyword `return` as shown below.

```
fun getLifePhase -> String {
  if age < 3 { return "baby" }
  else if age <= 12 { return "child" }
  else if age <= 19 { return "teenager" }
  else { return "adult" }
}
```

If `^` were used, this would be considered to be the return of the function, not the return of the method. A `return` statement causes a return to the method that called the current method, as usual.

Generic arrays of `Cyan` have a method `foreach` that can be used to iterate over the array elements. The argument to this method is a function that takes a parameter of the array element type. This function is called once for each array element:

```
var Array<Int> firstPrimes = {# 2, 3, 5, 7, 11 #};
  // prints all array elements
firstPrimes foreach: { (: Int e :)
  Out println: e
};
var sum = 0;
  // sum the values of the array elements
firstPrimes foreach: { (: Int e :)
  sum = sum + e
};
Out println: sum;
```

An statement `^ expr` is equivalent to `return expr` when it appears in the level of method declaration; that is, outside any function inside a method body. See the example:

```
fun aMethod: (Int x, Int y) {
  var b = { ^ x < 0 || y < 0 };
  // method does not return in the next statement
  (b eval) ifTrue: { Error signal: "wrong coordinates" };
  // method returns in the next statement
```

```
  ^ sqrt: ((Math sqr: x) * (Math sqr: y));  
}
```

10.1 Problems with Anonymous Functions

Anonymous functions are extremely useful features. They are supported by many functional and object-oriented languages such as Scheme, Haskell, Smalltalk, D, and Ruby. However, this feature causes a runtime error when

- (a) an anonymous function accesses a local variable that is destroyed before the function becomes inaccessible or is garbage collected. Then the body of the function may be executed and the non-existing local variable may be accessed, causing a runtime error;
- (b) a function with a return statement live past the method in which it was declared. When the anonymous function body is executed, there will be a return statement that refers to a method that is no longer in the call stack.

We will give examples of these errors. Assume that “Function<Nil>” is the type functions that does not take parameters and returns nothing.²

```
object Test  
  fun run {  
    prepareError;  
    makeError;  
  }  
  fun prepareError {  
    function = { return };  
    return;  
  }  
  fun makeError {  
    function eval;  
  }  
  Function<Nil> function  
end
```

In `makeError`, the function stored in the instance variable receives message `eval` and statement `return` is executed. This is a return from method `prepareError` that is no longer in the stack. There is a runtime error.

```
object Test  
  fun run {  
    returnFunction eval  
  }  
  fun returnFunction -> Function<Nil> {  
    return { return };  
  }  
end
```

²This is not exactly true and this definition will soon be corrected.

Here `returnFunction` returns a function which receives message `eval` in `run`. Again, statement `return` of the function is executed in method `run` and refers to `returnFunction`, which is not in the call stack anymore.

```
object Test
  fun run {
    prepareError;
    makeError;
  }
  fun prepareError {
    var x = 0;
    function = { ^x };
  }
  fun makeError {
    Out println: (function eval);
  }
  Function<Int> function
end
```

In statement “function eval” in method `makeError`, the function body is executed which accesses variable `x`. However, this variable is no longer in the stack. It was when the function was created in `prepareError` because `x` is a local variable of this method. There is again a runtime error.

```
object Test
  fun run {
    var a1 = 1;
    var Function<Nil> b1;
    if a1 == 1 {
      var a2 = 2;
      var b1 = { Out println: a2 };
    };
    b1 eval
  }
end
```

Here a function that uses local variable `a2` is assigned to variable `b1` that outlives `a2`. After the `if` statement, `a2` is removed from the stack and message `eval` is sent to `b1`, causing an access to variable `a2` that no longer exists.

`Function< Function<Int> >` is the type of functions that return objects of type `Function<Int>`.

```
object Test
  fun run {
    var a1 = 1;
    var Function< Function<Int> > b1;
    if a1 == 1 {
      var a2 = 2;
      var b1 = { ^{ ^a2 } }
    };
    (b1 eval) eval;
  }
end
```


After the execution of “var b1 = { ^{ ^a2 } }”, b1 refers to a function that refers to local variable a2. In statement (b1 eval) eval, variable a2, which is no longer in the stack, is accessed causing a runtime error.

There are some unusual use of functions that would not cause runtime errors:

```
object Test
  fun run {
    var a1 = 1;
    var Function<Int> b1;
    if a1 == 1 {
      var b2 = {
        var b1 = { ^a1 }
      };
      b2 eval;
    };
    b1 eval
  }
end
```

No error occurs here because b1 and a1 are create and removed from the stack at the same time.

```
object Test
  fun run {
    var a1 = 1;
    var Function<Function<Int>> b1;
    if a1 == 1 {
      var b1 = { ^{ ^a1 } }
    };
    Out println: ( (b1 eval) eval );
  }
end
```

Here (b1 eval) eval will return the value of a1 which is in the stack. No error will occur.

```
object Test
  fun run {
    Out println: test
  }
  fun test -> Int {
    var Function<Nil> b1;
    {
      var b2 = {
        b1 = { return 0 };
      };
      b2 eval;
    } eval;
    b1 eval;
    Out println: 1
  }
end
```

After message send “b2 eval” a function is assigned to b1. After “b1 eval” statement “return 0” is executed and method test returns. The last statement is never reached. Note that function

```
{ return 0 }
```

is a function that does not return a value. Therefore its type is `Function<Nil>`.

10.2 Some More Definitions

A Cyan function is a closure, a literal object that can close over the variables visible where it was defined. More rigorously, the syntax `{ (: params :) stats }` creates a closure at runtime for the linking with the instance and local variables is only made dynamically. An object is created each time a function appears at runtime. Therefore the code

```
var Int x;
var Function<Int> a, b, c;
a = { ^ i*i + x };
b = { ^ i*i + x };
c = { ^ i*i + x };
```

creates three functions, each of which captures variable `x`.

Variables used inside a function can be preceded by a `%` to indicate that a copy of them should be made at the function creation. Then any changes of the values of these variables are not propagated to the environment in which the function is. See the example.

```
fun Int test {
  var x = 0;
  var y = 0;
  var b = {
    %x = %x + 1;
    Out println: %x;
    y = y + 1;
  };
  assert: (x == 0 && y == 0);
  b eval;
  assert: (x == 0 && y == 1);
  return x + y;
}
```

`%x` inside the function means a copy of the local variable `x`. The `y` inside the function means the local variable `y`. The changes to it caused by statement `b eval` will remain. It is illegal to use both `x` and `%x` in a function. It is illegal to use `%` with a parameter — since parameters are read only, it is irrelevant to use `%` with them.

10.3 Classifications of Functions: u-Functions and r-Functions

Before studying functions in depth, it is necessary to define what is “scope”, “variable of level `k`”, and “function of level `k`”. Each identifier is associated to a scope, the region of the source code in which the identifier is visible (and therefore it can be used). A scope can be the region of a method or of a function, both delimited by `{` and `}`. The scope of a local variable starts just after its declaration and goes to the enclosing “`}`” of the function in which it was declared. A scope will be called “level 1” if the delimiters

{ and } are that of a method. “level 2” is the scope of a function inside level 1. In general, scope level $n + 1$ is a function inside scope n :

```
fun test: (Int n) {
    // scope level 1
    var Int a1 = n;
    (n < 0) ifFalse: {
        // scope level 2
        var Int a2 = -a1;
        (n > 0) ifTrue: {
            // scope level 3
            var a3 = a2 + 1;
            Out println: "> 0", a3
        }
        ifFalse: { Out println: "= 0" }
    }
} // a1 and n are removed from the stack here
```

We will call “variable of level k ” a variable defined in scope level “ k ”. Therefore variable a_i of this example is a variable of level i . The level of parameters is considered -1 . There is no variable of level 0 .

The variables *external* to a function are those declared outside the code between `and` that delimits the function. For example, a_1 is external to the function passed as parameter to selector `ifFalse:` in the previous example (any of the `ifFalse:` selectors). And a_1 and a_2 are external to the function that is argument to the selector `ifTrue:`.

A function is called “function of level -1 ” if it

- (a) only accesses external local variables using `%`;
- (b) possibly uses parameters (always without `%` because it is illegal to use `%` with parameters);
- (c) possibly uses instance variables;
- (d) does not have return statements;

By “*access*” a local variable we mean that a local variable appear anywhere between the function delimiters, which includes nested functions. In the example that follows, the function that starts at line 3 and ends at line 7 *accesses* local variable a_1 which is external to the function. This access is made in the function of line 5 which is inside the function of lines 4-6 which is inside the 3-7 function. Therefore 3-7 is not a function of level -1 . And neither is the function of lines 4-6 or the function of line 5. However, the function that is the body of method `test` (lines 1-8) is a function of level -1 .

```
1 fun test {
2     var a1 = 1;
3     { var a2 = 2; // start
4         { var a3 = 3;
5             { ++a1 } eval;
6         } eval;
7     } eval // end
8 }
```

Let v_1, v_2, \dots, v_n be the external *local* variables accessed in a function B without `%` — B is a function, not a variable that refers to a function. Instance variables and parameters are not considered. If m is the

level in which B is defined, then B can only access external local variables defined in levels $\leq m$. But not all variables of levels $\leq m$ are visible in B for some of them may belong to sister functions or they may be defined after the definition of B. Variables defined in levels $> m$ are either inaccessible or internal to the function. The following example explains these points.

```

fun test {
  // level 1
  var a1 = 1;
  { // level 2
    var a2 = 2; // start of function B1
    { // level 3
      var a31 = 31; // start of function B2
      { ++a1 } eval; // function B3
    } eval; // end B2
    var a22 = 2;
    { // level 3
      var a32 = 32; // start of function B4
      { // start of function B5
        // level 4
        var a5 = 5;
        a2 = a1 + a2 + a5
      } eval // end B5
    } eval; // end B4
  } eval // end B1
}

```

Function B2 is defined at level 2 but it cannot access variable a22 of level 2 — it is defined after B2. Variable a5 defined at level 4 is not visible at function B2.

The important thing to remember is “B defined at level m can only access external local variables defined in levels $\leq m$ ”, although not all variables of levels $\leq m$ are accessible at B. The example of Figure 10.1 should clarify this point. Ellipses represent functions. A solid arrow from function C to function B means that C is inside B. A dashed arrow from C to B means that C uses local variables declared in B.

This Figure represents the functions of the example that follows. The root is the function of the method itself which is represented by the top-level ellipse in the Figure. The numbers that appear in the ellipses are the return values of the functions. This number is used to identify the functions (we will say function 0 for the function that returns 0). The values returned by all the functions are not used (the return value of a method may be ignored. Statements like “1 + 2” are legal).

```

fun test {
  var v0 = 0;
  {
    var v1 = 1;
    {
      ++v1;
      ++v0;
      ^3
    } eval;
    var v11 = 2;
    {

```

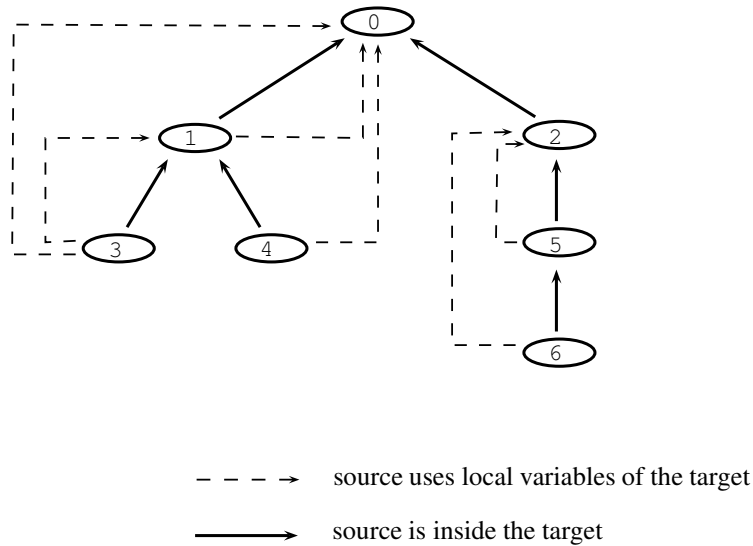


Figure 10.1: Nesting of functions

```

    ++v0;
    ^4
  } eval;
  ^1
} eval;
{
  var v2 = 2;
  { ^5
    {
      ++v0;
      ++v2;
      ^6
    } eval;
  } eval;
  ^2
} eval;
^0
}

```

By the scope rules of Cyan, a function B can only access its own local variables or variables from functions that are ancestors of B.³ Only variables declared before B are accessible. In this example, the function that returns 3 cannot access `v11` even though this variable is declared in an outer function (because the declaration appears after the declaration of function 3). In the Figure, a function B may access local variables of function A if there is a path in solid arrows from A to B (we will write just path from A to B).

When method `eval` or `eval:` of a function A is called, the runtime system pushes to the stack the local variables of A. Till the method returns, these local variables are there and they can be accessed by functions declared inside A. Using the Cyan example above and the Figure, when method `eval` of

³X is an ancestor of Y if Y is textually inside X.

function 0 is called, it pushes its local variables to the stack. Then function 1 is called and this function calls function 4 that accesses variable `v0` declared at function 0. No error occurs because `v0` is in the stack. To call function 4 it was first necessary to call function 1 and, before this, function 0 which declares variable `v0`.

However, this example could be modified in such a way that function 3 is assigned in function 1 to a variable `b1` declared at function 0 (suppose this is legal — it is not as we will see).

```
// unimportant functions were removed
fun test {
  var v0 = 0;
  var Function<Int> b1;
  {
    var v1 = 1;
    b1 = {
      ++v1;
      ++v0;
      ^3
    };
    var v11 = 2;
    ^1
  } eval;
  // compile-time correct, runtime error
  b1 eval;
  ^0
}
```

`b1` is visible in function 1 by the scope rules of Cyan. After functions 3 and 1 are removed from the stack and control returns to method `eval` of 0, `b1` receives an `eval` message. Since `b1` refers to function 3, the method called will try to access variable `v1` declared in function 1. This variable is no longer in the stack. There would be a runtime error. However, the rules of Cyan will not allow function 3 be assigned to variable `b1` of function 0. A function variable `b` will never refer to a function that uses external variables that live less than `b`.

Inner functions may be assigned to variables of outer functions without causing runtime errors:

```
fun test {
  var a1 = 1;
  var Function<Nil> b;
  {
    var a2 = 2;
    {
      {
        b = { ++a1; }
      } eval
    } eval;
    c eval;
  } eval;
  b eval;
}
```

Here a function `{ ++a1 }` is assigned to variable `b` declared at level 1. This does not cause errors because the function only refer to variables of level 1. Variable `b` and `a1` will be removed from the stack at the

same time. There is no problem in this assignment. In this example, if the function used `a2` instead of `a1`, there would be a runtime error at line “`b eval`”. Variable `a2` that is no longer in the stack would be accessed. To prevent runtime errors of the kind “reference to a variable that is no longer in the stack” Cyan only allows an assignment “`b = B`”, in which `B` is a function, if the variables accessed in `B` will live as much as `b`. This is guaranteed by the rules given in the next section.

Functions are classified according to the external local variables they access and if they have or not return statements. To a function `B` is associated a number `bl(B)` called “the level of function `B`” found according to the following rules.

1. A function that accesses local variables only using `%` and that do not have return statements (even considering nested functions) are called “functions of level -1”. This kind of function may access parameters;
2. Functions that access at least one external local variable (excluding parameters) without using `%` have their number `bl(B)` calculated as

$$\text{bl}(B) = \max\{ \text{lev}(v1), \text{lev}(v2), \dots, \text{lev}(vn) \}$$

`lev(v)` is the level of variable `v`. `v1, v2, ..., vn` are the external *local* variables accessed in function `B` without `%`. The “*function level*” of `B` is `bl(B)`.

3. A function that do not access any local variables (excluding parameters) without `%` but that does have a return statement (even in nested functions) is called “function of level 0”. This function may access variables with `%`.

For short, a function that have a return statement is at least of level 0. A function that has a reference to a external local variable of level `k`⁴ is at least a function of level `k`. However, it may be a function of level $\geq k$ (if it accesses an external variable, without `%`, of a superior level). The use of instance variables or parameters is irrelevant to the calculus of the level of a function. Instance variables are not created with the method or when the function receives message `eval` or `eval:`. And parameters are read only — it is as if every parameter were used with `%`.

The definition of `bl(B)`, the level of a function, is different from the definition “function defined or declared at level `k`” used previously. A function defined at level `k` is a function that is textually at level `k`. The *function level* of a function depends on the external local variables that appear in its body (including the nested functions inside it).

The higher the level of a local variable a function accesses, the more restrictive is the use of the function. For example, function `B3` in the next example can be assigned to any of the local function variables `bi` of this example. But `B4` cannot. If it is assigned to `b2`, for example, the message send “`b2 eval`” would access a local variable `a31` that is no longer in the stack.

```
fun test {
  // level 1
  var a1 = 1;
  var Function<Nil> b1;
  { // start of function B1
    // level 2
    var a2 = 2;
    var Function<Nil> b2;
    { // start of function B2
```

⁴Use the variable without `%`.

```

    // level 3
    var a31 = 31;
    var Function<Nil> b31;
    var b31 = { ++a1 }; // function B3
    var Function<Nil> b32;
    b32 = { ++a31 }; // function B4
    b2 = { ++a2 };
  } eval;
  b2 eval;
} eval;
b1 eval;
}

```

The example below should clarify the definition of “function level k”.

```

fun Int test: (Int p) {
  // level 1
  var a1 = 1;
  var b1_1 = { ^a1 }; // function of level 1, defined at level 1
  var b1_2 = { ^0 }; // function of level -1, defined at level 1
  var b1_3 = { // function of level 1 because it uses a1
    // level 2
    var a2 = 2;
    var b2_1 = { ^a1 }; // function of level 1, defined at level 2
    var b2_2 = { a2 = 1 }; // function of level 2, defined at level 2
    var b2_3 = { return p }; // function of level 0, defined at level 2
    var b2_4 = { Out println: %a2 }; // function of level -1, defined at level 2
    var b2_5 = { // function of level 2 because it uses a2
      // level 3
      var a3 = 3;
      var b3_1 = { b1_1 eval }; // function of level 1, defined at level 3
      var b3_2 = { ++a2; return }; // function of level 2, defined at level 3
      var b3_3 = { ^a3 }; // function of level 3, defined at level 3
      var b3_4 = { return }; // function of level 0, defined at level 3
      var b3_5 = { ^p }; // function of level -1, defined at level 3
    }
  };
  b1_3 eval;
  return 0
}

```

A function of level -1 may access local variables using %, parameters without using %, and instance variables. Functions of level -1 are called *u-functions* or *unrestricted-use functions*. There is no restriction on the use of u-functions: they may be passed as parameters, returned from methods, returned from functions, assigned to instance variables, or assigned to any variable. They only have the type restrictions of regular objects.

Functions of levels 0 and up are called *r-functions* or *restricted-use functions*. There are limitations in their use: they cannot be stored in instance variables, returned from methods and functions, and there are limitations on the assignment of them to local variables. This will soon be explained.

An r-function that takes parameters of types T1, T2, ..., Tn and returns a value of type R inherits from prototype

```
@restricted abstract object Function<T1, T2, ..., Tn, R>
  abstract fun eval: (T1, T2, ..., Tn) -> R
end
```

@restricted is a pre-defined metaobject that restricts the way this kind of function is used — see Section 10.4. These restrictions apply to this prototype only. It does not apply to sub-prototypes.

There is a special prototype Function<Boolean> with methods whileTrue: and whileFalse:

```
package cyan.lang

@restricted abstract object Function<Boolean>
  abstract fun eval -> Boolean
  fun whileTrue: (Function<Nil> aFunction) {
    (self eval) ifTrue: {
      aFunction eval;
      self whileTrue: aFunction
    }
  }
  fun whileFalse: (Function<Nil> aFunction) {
    (self eval) ifFalse: {
      aFunction eval;
      self whileFalse: aFunction
    }
  }
end
```

These methods implement the while construct as explained in Section 3.7.

An r-function that does not take any parameters and does not return a value inherits from

```
package cyan.lang

@restricted
abstract object Function<Nil>
  abstract fun eval
  fun loop {
    self eval;
    self loop
  }
  fun repeatUntil: (Function<Boolean> test) {
    self eval;
    (test eval) ifFalse: {
      self repeatUntil: test
    }
  }
  fun ( (catch: Any)+ finally: Function<Nil> ) t {
  }
  fun hideException {
    {
```

```

        self eval
      } catch: { (: CyException e :)
      };
    }

```

end

Method `loop` implements an infinite loop and `repeatUntil:` implements a loop that ends when the function parameter evaluates to `true`.

An u-function that takes parameters of types `T1, T2, ..., Tn` and returns a value of type `R` inherits from prototype `Function<T1, T2, ..., Tn, R>`

```

@unrestricted abstract object UFunction<T1, T2, ..., Tn, R>
    extends Function<T1, T2, ..., Tn, R>

```

end

Therefore the above prototype also defines a method

```

    abstract fun eval: (T1, T2, ..., Tn) -> R

```

An u-function that does not take any parameters inherits from

```

@unrestricted abstract object UFunction<Nil>
    extends Function<Nil>

```

end

There is a special prototype `UFunction<Boolean>` with methods `whileTrue:` and `whileFalse:` inherited from `Function<Boolean>`.

```

@unrestricted
abstract object UFunction<Boolean>
    extends Function<Boolean>

```

end

An u-function that does not take any parameters and returns nothing implements

```

@unrestricted abstract object UFunction<Nil>
    extends Function<Nil>

```

end

Every function has its own prototype that inherits from one of the `Function` or `UFunction` objects. When the compiler finds a function

```

{ ^n }

```

it creates a prototype `Function001` that inherits from `Function<Int>` (assume that `n` is a local `Int` variable). The name `Function001` was chosen by the compiler and it can be any valid identifier. If this function is assigned to a variable in an assignment,

```

var b = { ^n }

```

the type of `b` will be `Function<Int>`. Of course, `Function001` is declared with the `@restricted` metaobject.

As another example, the type of variable `add` in

```

var add = { (: Int n :) ^n + 1 };

```

could be `UFunction017`. Since this function inherits from `UFunction<Int, Int>` we can declare `add` before assigning it a value as

```
var UFunction<Int, Int> add;
add = { (: Int n :) ^n + 1 };
```

All methods of a function but `eval:` are regular methods. `eval:` methods of functions are called *primitive* methods. A primitive method is not an object. The only allowed operation on a primitive method is to call it.

Functions are then a special kind of object, one that has a method, `eval:` or `eval` that is not an object. Only `eval:` or `eval` methods of functions are primitive methods. However, all prototypes that extend prototype `UFunction` or `Function` can be passed as an argument to a method that expect an `UFunction` or `Function` as a real parameter. For example, an `Int` array defines a `foreach:` method that expects an r-function as parameter that accepts an `Int` parameter and returns `Nil`. One can pass as parameter a regular object:

```
object Sum extends UFunction<Int, Nil>
  public sum = 0
  fun eval: (Int elem) {
    sum = sum + elem
  }
end
...
var Array<Int> v = {# 2, 3, 5, 7, 11, 13 #};
v foreach: Sum;
Out println: "array sum = " + (Sum s);
```

Warning

Cyan will allow metaobjects to be attached directly to types as in

```
var Char@letter ch;
ch = 'A'; // ok
ch = '0' // compile-time error
```

Here metaobject `letter` is attached to `Char` and controls the type checking of `ch`. This feature will be used with `Functions`. A metaobject `rf` will be attached to prototype `Function` to give the precise type of a function:

```
fun Int test {
  var Int a1 = 0;
  var f0 = { return 0; };
  var f1 = {
    ++a1;
  };
  var f2 = { ^0 };
  (f2 eval) println;
  f1 eval;
  f0 eval;
}
```

The types of the function variables will be:

f0	Function@rf(0)	Note that <code>Function</code> without the metaobject will be the nowadays <code>UFunction</code> .
f1	Function@rf(1)	
f2	Function	

10.4 Type Checking Functions

Now it is time to unveil the rules that make functions statically typed in Cyan. The rules are:

- (a) there is no restriction on the use of u-functions and variables whose type is `UFunction<..., R>`. An instance variable can have type `UFunction<..., R>`;
- (b) instance variables cannot have type `Function<T1, ... Tn, R>`;
- (c) methods and functions cannot have `Function<T1, ... Tn, R>` as the return type;
- (d) a variable `r` declared at level `k` whose type is `Function<T1, ... Tn, R>` may receive in assignments:
 - a variable `s` of level `m` if $m \leq k$ and the type of `s` is `Function<T1, ... Tn, R>` or one of its subtypes, including `UFunction<T1, ... Tn, R>`;
 - an r-function of level `m` if $m \leq k$ and this r-function extends prototype `Function<T1, ... Tn, R>`;
 - an u-function that extends prototype `UFunction<T1, ... Tn, R>`;
- (e) a parameter whose type is `Function<T1, ... Tn, R>` is considered a variable of level 0. The real argument corresponding to this parameter may be a variable or function of any level. Of course, the type of the variable or function should be `Function<T1, ... Tn, R>` or one of its subtypes;
- (f) a variable or parameter whose type is **Any** **cannot** receive as real argument any r-function. Unfortunately this introduces an exception in the subtype hierarchy: a sub-prototype may not be a sub-type. For example, `Function<Int>` is not subtype of **Any**. Although a function like `{ ^0 }` inherits from **Any** (indirectly), its type is not considered subtype from **Any**. The only way of correcting this is allocating the local variables in the stack — see Section 10.12. But that is inefficient to say the least.

Based on the rules for type checking functions, one can conclude that:

- (a) instance variables can be referenced by both u-functions and r-functions;
- (b) a function that has a return statement but does not access any local variables is a function of level 0. Its type is `Function<T1, ... Tn, R>` for some types `Ti` and `R`;
- (c) the restriction “methods and functions can have `Function<T1, ... Tn, R>` as the return type (but not `Function<T1, ... Tn, R>` could be changed to “a method can only return u-functions and a function defined at level `k` can only return a function if it is of level `m` with $m \leq k$ ”. In the same way, a function defined at level `k` can have a variable as the return value if this variable is of level `m` with $m \leq k$. However, we said “*could*”, these more liberal rules are not used in Cyan;
- (d) since parameters are read-only, it is not possible to assign a variable or function to any of them;
- (e) both r-functions and u-functions can access instance variables since their use do not cause any problems — instance variables belong to objects allocated in the heap, a memory space separated from the stack. Then it is legal to return a function that accesses an instance variable or to assign such a function to any `UFunction` variable:

```
object Person
  @init(name, age)
  fun init { }
  private fun functionCompare -> UFunction<Person, Boolean> {
    return { (: Person p :) ^age > (p age) }
```

```

    }
    public String name = "noname"
    public Int age = 99
end
...
var myself = Person new;
// method name: String age: Int is automatically created
myself name: "José" age: 14;
if (Person functionCompare) eval: myself {
    Out println: "Person is older than José";
}

```

- (f) the type of an instance variable or return method value cannot be an r-function. But it can be an u-function. Therefore there will never be an instance variable referring to a function that has a reference to a local variable. And a function returned by a method will never refer to a local method variable;
- (g) a parameter that has type `Function<T1, ... Tn, R>` cannot be assigned to any variable of the same type because this variable is of level at least 1 and the parameter is of level -1;
- (h) the generic prototype `Array<T>` declares an instance variable of type T. Therefore the generic array instantiation `Array<Function<T1, ... Tn, R>>` causes a compile-time error — r-functions cannot be types of instance variables. In the same way, `Function<T1, ... Tn, R>` cannot be the parameter to most generic containers (yet to be made) such as `Hashtable`, `Set`, `List`, and so on.

This is regrettable. We cannot, for example, create an array of r-functions:

```

var sum Float = 0;
var prod Float = 0;
var sumSqr Float = 0;
mySet applyAll: {# { (: Float it :) sum += it },
                { (: Float it :) prod *= it },
                { (: Float it :) sumSqr += it*it } #};

```

Future version of Cyan could employ a different rule: a restricted function or any restricted object could be the type of an instance variable of prototype P if P is a restricted object (declared with metaobject `@restricted`). Arrays, tuples, and the like would automatically be restricted or not according to the parameter type. So `Array<Function<Int, Nil>>` would be a restricted type but `Array<UFunction<Int, Nil>>` would not.

The rules for checking the use of r-functions are embodied in metaobject `@restricted`. The compiler passes the control to this metaobject when type checking r-functions. It then implements the above rules.

10.5 Some Function Examples

In the example that follows, some statements are never executed when message `run` is sent to `A`. In particular, when message `eval` is sent to `b` the control returns to method `run` which prints 0. All the intervening methods are removed from the stack of called methods.

```

object A
  fun run {

```

```

    Out println: (self m)
  }
  fun m -> Int {
    p: { return 0 };
    Out println: "never executed";
  }
  fun p: (Function<Nil> b) {
    t: b;
    Out println: "never executed";
  }
  fun t: (Function<Nil> b) {
    b eval;
    Out println: "never executed";
  }
}
end

```

In this example, an r-function is passed as a parameter. There is no runtime error.

```

object A
  fun aMethod {
    var Int x;
    x = In readInt;
    Out println: (anotherMethod: { (: Int y :) ^y + x });
  }
  fun anotherMethod: (Function<Int, Int> b) ) -> Int {
    ^ yetAnotherMethod: b;
  }
  fun yetAnotherMethod: (Function<Int, Int> b) -> Int {
    ^ b eval: 0;
  }
  ...
}
end

```

Method `aMethod` calls `anotherMethod` which calls `yetAnotherMethod`. No reference to function `{ (: Int y :) ^y + x }` last longer than local variable `x`.

A parameter of type `Any` cannot receive an r-function as real argument. If it could, a runtime error would occur.

```

object Test
  fun test {
    { var n = 0;
      // function passed as parameter. The
      // real argument has type Any
      do: { ++n }
    } eval;
    makeError
  }
  fun do: (Any any) {
    self.any = any
  }
}

```

```

fun makeError {
    // access to local variable n
    // that no longer exists
    any ?eval
}
Any any
end

```

10.6 Why Functions are Statically-Typed in Cyan

This section does not present a proof that functions in Cyan are statically typed. It just gives evidences of that.

To introduce our case we will use functions B_0, B_1, \dots, B_n in which B_i is defined at level i and B_{i+1} is defined inside B_i . So there is a nesting

$$B_n \subset B_{n-1} \subset \dots \subset B_1 \subset B_0$$

It was used \subset to mean “*nested in*”. Function B_j declares a local variable v_j . Note that B_0 is the body of a method (functions of level 0 are always methods).

Suppose B_n uses external local variables $v_{i_1}, v_{i_2}, \dots, v_{i_k}$ of functions $B_{i_1}, B_{i_2}, \dots, B_{i_k}$ with $i_1 < i_2 < \dots < i_{k-1} < i_k$. It is not important whether B_n uses or not more than one variable of each function.

Let us concentrate on B_{i_k} which defines variable v_{i_k} accessed by B_n . Since there is a nesting structure, functions $B_{i_k+1}, B_{i_k+2}, \dots, B_{n-1}$ also have references to v_{i_k} (because B_n is nested inside these functions). This fact is used in the following paragraph.

B_n can be assigned to a function variable of B_j with $i_k \leq j < n$. This does not cause a runtime error because a function B_j with $i_k \leq j < n$ is only called when B_{i_k} is in the stack. B_j cannot be assigned to a variable b_t of level t with $t < i_k$ because B_j also has a reference to v_{i_k} and, by the rules, it can only be assigned to variables that appear in function B_t with $i_k \leq t < j$.

B_n also has a reference to variable $v_{i_{k-1}}$ of $B_{i_{k-1}}$. Therefore B_n could not be assigned to function variables of functions B_j with $j < i_{k-1}$. Considering all cases, B_n cannot be assigned to function variables of functions B_j with

$$\begin{aligned}
 j &< i_1 \\
 j &< i_2 \\
 &\dots \\
 j &< i_{k-1} \\
 j &< i_k
 \end{aligned}$$

Since $i_1 < i_2 < \dots < i_{k-1} < i_k$, we conclude that B_n cannot be assigned to a function variable of function B_{i_k} . Then B_n can only be assigned to a function variable of function B_j with $j \geq i_k$. This is what one of the rules of Section 10.4 says. Therefore these rules prevent any runtime errors of the kind “access to a function variable that does not exist anymore” related to the assignment of r-functions to local variables. It is not difficult to see that the other rules prevent all of the other kinds of errors related to r-functions such as the passing of parameters, assignment of functions to **Any** variables, assignment of r-functions to instance variables (not allowed), and so on.

10.7 Functions with Multiple Selectors

Regular functions only have one selector, which is `eval:` or `eval` (when there is no parameter). It is possible to declare a function with more than one `eval:` selector. One can declare

```

var b = { (: eval: (T11 p11, T12 p12, ..., T1k1 p1k1)
          eval: (T21 p21, T22 p22, ..., T2k2 p2k2)
          ...
          eval: (Tn1 pn1, Tn2 pn2, ..., Tnkn pnkn)
          -> R :) {
  // function body
};

```

Consider a function with a method composed by n `eval:` selectors. The i^{th} `eval` selector has k_i parameters. This function inherits from prototype

```
Function<T11, T12, ..., T1k1><T21, T22, ..., T2k2>...<Tn1, Tn2, ..., Tnkn, R>
```

A similar u-function inherits from the corresponding UFunction generic prototype.

The `eval` method corresponding to the above function is

```

fun eval: ( T11 p11, T12 p12, ..., T1k1 p1k1)
          eval: ( T21 p21, T22 p22, ..., T2k2 p2k2)
          ...
          eval: ( Tn1 pn1, Tn2 pn2, ..., Tnkn pnkn) -> R
  {
  // function body
}

```

As an example, one can declare a function

```

var Function<String><Int, Nil> b;
b = { (: eval: (String key) eval: (Int value) :)
      Out println: "key #key is #value"
};
// prints "key One is 1"
b eval: "One" eval: 1;

```

10.8 The Type of Methods and Methods as Objects

Methods are objects in Cyan although of a special kind: they are functions. Then every method

```

fun s1: (T11 p11, T12 p12, ..., T1k1 p1k1)
      s2: (T21 p21, T22 p22, ..., T2k2 p2k2)
      ...
      sn: (Tn1 pn1, Tn2 pn2, ..., Tnkn pnkn) -> R
  {
  // function body
}

```

extends

```
UFunction<T11, T12, ..., T1k1><T21, T22, ..., T2k2>...<Tn1, Tn2, ..., Tnkn, R>
```

Cyan methods are then unrestricted functions. A method of a specific object is got by calling method “`getMethod:`” of `Any` as in

```
obj getMethod: "signature"
```

in which `signature` is the method signature (selectors, parameter types, and return value type).

Consider the `Box` prototype:


```

object Box
  fun get -> Int { return value }
  fun set: (Int other) { value = other }
  Int value = 0
end

```

Methods of this prototype and of objects created from it can be accessed as in

```

var UFunction<Int> getMet;
var UFunction<Int, Nil> setMet;
var UFunction<Int, Nil> anotherGetMethod;

```

```

getMet = Box getMethod: "get -> Int";
setMet = Box getMethod: "set: Int";
var box = Box new;
box set: 10;

```

```

setMet eval: 5;
  // prints 5
Out println: (getMet eval);
  // prints 0
Out println: (box get);

```

This syntax can be used to set a method of a prototype such as

```

var Int local;
var Box b = Box new;
b setMethod: "get -> Int", { ^0 };
b setMethod: "set: Int", { (: Int n :) Out println: n };
assert: (b get == 0);
  // method getDay of Date returns an Int
b setMethod: "get -> Int", (Date getMethod: "getDay -> Int");

```

A method of a prototype may be set too. The existing objects of that prototype are affected — they will use the new method.

```

var Box before = Box new;
before set: 0;
Box setMethod: "get -> Int", { ^1 };
var Box after = Box new;
assert: (before get == 1);
assert: (after get == 1);

```

Method `getMethod:` returns an object of type `Any`. Then there should be a type error in the code

```
getMet = Box getMethod: "get -> Int";
```

But there is none because metaobject `checkGetMethod` attached to `getMethod:`

- (a) checks whether the parameter is a literal string. It issues an error if it is not;
- (b) checks whether the message receiver has a method with the same signature as the parameter;
- (c) changes the return type of the message send to the appropriate type. Then

```

Box getMethod: "get -> Int"
has type UFunction<Int>.

```

Metaobject `checkSetMethod` does similar checks to method `setMethod:.` It checks whether

- (a) the first parameter is a literal string, issuing an error if it is not;
- (b) the message receiver has a method with the same signature as the first parameter;
- (c) the signature given in the first parameter is equal to the signature of the expression of the second parameter. It signals a compiler error if it is not.

As seen before, `eval:` methods of functions are not objects. They are called *primitive* methods. Although it is legal to call a primitive method, it is illegal to retrieve or set one using `getMethod:` or `setMethod:.`

```
var UFunction<Int> b;
var UFunction<Int> getMet;

getMet = Box getMethod: "get -> Int";
    // runtime error in the next line
b = getMet getMethod: "eval -> Int";
    // runtime error in the next line
    // if the previous assignment is commented
b setMethod: "eval -> Int", { (: -> Int :) ^0 };
```

Of course, the second runtime error will never occur because of the first. But you got the idea.

Although “`b getMethod: "eval -> Int"`” has type `UFunction<Int>` (as variable `b` has), there is a runtime error: it is not legal to retrieve a primitive method as an object. They are not objects and cannot be treated as such.

Methods that access instance variables are duplicated for every object of the prototype (at least *conceptually*). If two objects of prototype `Box` are created then there are three instances that represent method `get` (and three for `set` too, of course). This occurs because the method object closes over the instance variables of the prototype in the same way a function closes over the local variables of a method. As there are three objects there are three instance variables called `value` and three methods for `get`, each one closing over one of the instance variables.

As an example of duplicating methods as objects, the following code compares two `get` methods of different `Box` objects. Since they come from different objects, they are different.

```
    // create a get method for the new object
var box = Box new;
    // create another get method for the new object
var other = Box new;
assert: (box getMethod: "get -> Int") eq: (other getMethod: "get -> Int");
```

The same situation occurs with functions:

```
var n = 0;
1..3 repeat: {
    var Int value = n;
    ++n;
    var getFunction = { ^value };
    Out println: (getFunction value);
};
```

Since the function that is parameter to `repeat`: is executed three times, three functions objects { `^value` } are created, each of them closes over the local variable `value`. Note that variable `value` is created three times since it is a local variable to the most external function.

A method defined as

```

fun s1: (T11p11, T12 p12, ..., T1k1 p1k1)
    s2: (T21p21, T22 p22, ..., T2k2 p2k2)
    ...
    sn: (Tn1pn1, Tn2 pn2, ..., Tnk_n pnkn) -> R
    {
// function body
}

```

assigns to `slot s1:s2: ...sn`: an u-function with the same method parameters and same return value type as the method. Note that when we use the sign = in a method declaration we are not assigning the expression to the method (as slot). Then 0 is not being assigned to method `zero`. Instead, 0 is the return value of this method.

```

fun zero -> Int = 0

```

10.9 Message Sends

When a message is sent, the runtime system looks for an appropriate method in the object that received the message. This search has already been explained in Section 4.11. After finding the correct method \tilde{m} , two actions may be taken:

- (a) if \tilde{m} is an u-function, the primitive method `eval`: of the function is called;
- (b) if \tilde{m} is a regular object, message `eval` or `eval: ...` that matches the original message is sent to \tilde{m} (with the original parameters). In this case, \tilde{m} should extends one of the `UFunction` prototypes.

Using methods as objects is very convenient in creating graphical user interfaces. Listeners can be regular methods. See the example.

```

object MenuItem
    fun onMouseClick: (UFunction<Nil> b) {
        ...
    }
end

object Help
    fun show { ... }
    ...
end

object FileMenu
    fun open { ... }
end

...
var helpItem = MenuItem new;

```

```

helpItem onMouseClick: (Help getMethod: "show");
var openItem = MenuItem new;
openItem onMouseClick: (FileMenu getMethod: "open");
...

```

10.10 Methods of Functions for Decision and Repetition

Objects `Function<Boolean>` and `UFunction<Boolean>` define some methods used for decision and iteration statements. The code of these methods is shown below.

```

package cyan.lang

@restricted abstract object Function<Boolean>
  abstract fun eval -> Boolean
  fun whileTrue: (Function<Nil> aFunction) {
    (self eval) ifTrue: {
      aFunction eval;
      self whileTrue: aFunction
    }
  }
  fun whileFalse: (Function<Nil> aFunction) {
    (self eval) ifFalse: {
      aFunction eval;
      self whileFalse: aFunction
    }
  }
end

```

10.11 Context Functions

Prototype `Any` (Section 4.13) defines a grammar method for dynamically adding methods to prototypes. It is necessary to specify each selector, the types of all parameters, the return value type, and the method body. This grammar method has the signature

```

fun (addMethod:
  (selector: String ( param: (Any)+ )?
  )+
  (returnType: Any)?
  body: Any)

```

Suppose we want to add a `print` method dynamically to prototype `Box`:

```

object Box
  fun get -> Int { return value }
  fun set: (Int other) { value = other }
  Int value = 0
end

```

We want to add a `print` method to every object created from `Box` or that has already been created using this prototype using `new` or `clone` (with the exception to those objects that have already added a `print` method to themselves). This method, if textually added to `Box`, would be

```
fun print { Out println: get }
```

Note that `Any` already defines a `print` method. However, the method `print` we define has a behavior different from that of the inherited method.

A first attempt would to add `print` dynamically would be

```
Box addMethod:  
  selector: #print  
  body: { Out println: get };
```

However, there is a problem here: it is used `get` in the function that is parameter to selector `body:`. The compiler will search for a `get` identifier in the method in which this statement is, then in the prototype, and then in the list of imported prototypes, constants, and interfaces. Anyway, `get` will not be considered as a method of `Box`, which is what we want. A second attempt would be

```
Box addMethod:  
  selector: #print  
  body: { Out println: (Box get) };
```

Here it was used `Box get` instead of just “`get`”. But then the `print` method of every object created from `Box` will use the `get` method of `Box`:

```
var myBox = Box new;  
myBox set: 5;  
Box set: 0;  
  // prints 0  
Box print;  
  // prints 0 too !  
myBox print;
```

Since the `print` method was dynamically added, it has to be called using `#`. In this example, both calls to `print` used the `get` method of `Box`, which returns the value 0.

This problem cannot be solved with regular functions. It is necessary to define a new kind of function, *context function* to solve it. A *context function* is declared as

```
{ (: T self, parameters and return type :) body }
```

Part “`T self`” is new. It means that inside the method body `self` has type `T`. The identifiers visible inside the function body are those declared in the function itself, those accessible through `T`, external parameters, and local variables preceded by `%`. For each parameter or local variable preceded by `%`, the function declares a variable with the same type and name. At the function creation, the values of the external parameters and local variables are copied to these function variables.

```
var b = { (: Any self :) Out println: %n };
```

Methods of the current object can be accessed by means of a local variable:

```
var mySelf = self;  
var b = {  
  (: Any self :)  
  Out println: (%myself age)  
};
```

Instance variables of the current object can be indirectly accessed by means of get and set methods of local variables such as `mySelf`.

With context functions, the `print` method of one of the previous example can now be adequately added to `Box`.

```
Box addMethod:
  selector: #print
  body: { (: Box self :) Out println: get };
```

Now the `print` method will send message `get` to the object that receives message `#print`:

```
var myBox = Box new;
myBox set: 5;
Box set: 0;
  // prints 0
Box print;
  // prints 5
myBox print;
```

Method `addMethod`: ... checks whether the context object passed in `selector` `body`: matches the selectors, parameters, and return value.

```
// error: function with parameter, selector without one
Box addMethod:
  selector: #print
  body: { (: Box self, Int n :) Out println: n };
// error: function has no Int parameter
// and return value should be Int
Box addMethod:
  selector: #add
  param: Int
  returnType: Int
  body: { (: Box self -> String :) ^get asString };
```

The type of the *context function*

```
{ (: S self, T1 t1, T2 t2, ..., Tn tn -> R :) ... }
```

is

```
ContextFunction<S, T1, T2, ..., Tn, R>
```

Interface `ContextFunction` is defined as

```
interface ContextFunction<S, T1, T2, ..., Tn, R>
  fun bindToFunction: S -> UFunction<T1, T2, ..., Tn, R>
end
```

Therefore the type of

```
{ (: S self, T1 t1, T2 t2, ..., Tn tn :) ... }
```

is

```
ContextFunction<S, T1, T2, ..., Tn, Nil>
```

Assuming that there is no statement “`~ expr`” in the body of the context function.

The compiler creates a *context object*⁵ from a context function. From

```
{ (: S self, T1 t1, T2 t2, ..., Tn tn -> R :) ... }
```

that uses local variables and parameters `v1, v2, ... vk` the compiler creates

```
object ContextFunction001(V1 v1, ..., Vk vk)
  implements ContextFunction<S, T1, T2, ..., Tn, R>

  fun bindToFunction: (S newSelf) -> UFunction<T1, T2, ..., Tn, R> {
    return { (: T1 t1, T2 t2, ..., Tn tn -> R :)
      // body of the context function with
      // self replaced by newSelf
      ...
    }
  }
end
```

For example, from the context function of the code

```
var Int i = 0;
var b = { (: Box self :) Out println: (%i + get) };
(b bindToFunction: Box) eval;
```

the compiler creates a regular object

```
object ContextObject001(Int i)
  implements ContextFunction<Box, Nil>

  fun bindToFunction: Box -> UFunction<Nil>
  return {
    Out println: (i + (newSelf get));
  }
end
```

And

```
var b = { (: Box self :) Out println: (%i + get) };
```

becomes

```
var b = ContextObject001(i);
```

A context function with multiple selectors is a context function with multiple `eval:` selectors:

```
{ (: S self, eval: T11 t11, ... T1n t1n eval: T21 t21, ... T2m t2m,
  ... eval: ... Tkp tkp -> R :)
  ...
}
```

The type of this context function is

⁵Chapter 11 define context objects, which are a generalization of functions.

```

interface ContextFunction<S, T11, ..., T1n><T21, ... T2m>...<Tk1, ... Tkp, R>
  fun bindToFunction: S -> UFunction<T11, ..., T1n><T21, ... T2m>...<Tk1, ... Tkp, R>
end

```

The `UFunction` returned by `bindToFunction:` is defined in Section 10.7. The type of a context function with multiple selectors that does not return a value is defined similarly.

In what follows, we will specify the checks made when calling `addMethod:` to add a method with a single selector. In a call

```

obj addMethod:
  selector: sel
  param: T1 param: T2 ... param: Tn
  returnType: R
  body: expr

```

metaobject `checkAddMethod` checks whether:

- (a) the parameters to all selectors of `addMethod:` ... but `body:` are literals;
- (b) the selector `sel` is a valid method name;
- (c) the selector `sel` ends with “:” if `n > 0`;
- (d) the selector `sel` does not end with “:” if `n == 0`;
- (e) the type of `expr` is subtype of `ContextFunction<S, T1, T2, ..., Tn, R>` in which `S` is supertype of `typeof(obj)` (the compile-time type of `obj`).

Even with these checkings there may be an error when the method `addMethod:` ... is called. For example, `obj` may refer to a `B` object although `typeof(obj)` is `A`. There is a final method `sel` in `B` that is not defined in `A`. The metaobject cannot detect that a final method is being changed. In case of error, method `addMethod:` ... throws exception `ExceptionAddMethod`.

It is possible that in future versions of Cyan all checking be postponed to runtime. At least if some of the parameters are not literals.

Let us see an example of use of *context functions*.

```

var myContextFunction = { (: Box self, Int p -> Int :) ^get + p };
Box set: 5;
var Function<Int, Int> b = myContextFunction bindToFunction: Box
assert: (b eval: 3) == 8;

var anotherBox = Box new;
anotherBox set: 1;
b = myContextFunction bindToFunction: anotherBox;
assert: (b eval: 3) == 4;

```

In one of the examples given above, a `print` method is added to prototype `Box` through `addMethod:` ... When this grammar method is called at runtime, method `print` will be added to all instances of `Box` that have been created and that will be created afterwards. However, if an instance of `Box` has added another `print` method, it is not affected:

```

var myBox = Box new;
myBox set: 10;
myBox addMethod:

```



```

    selector: #print
    body: { (: Box self :) Out println: "value = #{get}" };
Box addMethod:
    selector: #print
    body: { (: Box self :) Out println: get };
    // will print "value = 10" and not just "10"
myBox print;

```

Another method that takes a parameter and returns a value can be added to Box:

```

Box addMethod:
    selector: #returnSum
    param: Int
    returnType: Int
    body: { (: Box self, Int p -> Int :) ^get + p };

```

The metaobject attached to this grammar method checks whether the number of selectors (one), the parameter type, and the return value type matches the context function. It does in this case.

```

var myBox = Box new;
myBox set: 5;
assert (myBox ?returnSum: 3) == 8;

```

As another example, one can add methods to change the color of a shape:

```

object Shape
    public Int color
    public abstract fun draw
    ...
end
...

var colors = {# "blue", "red", "yellow", "white", "black" #};
    // assume that hexadecimal integer numbers can
    // be given in this way
var colorNumbers = {# ff_Hex, ff0000_Hex, ffff00_Hex, ffffff_Hex, 0 #};
var i = 0;
colors foreach: {
    (: String elem :)
    Shape addMethod:
        selector: elem
        body: { (: Shape self :) color: colorNumbers[i] };
    ++i;
};

```

Methods blue, red, yellow, white, and black are added to Shape. So we can write

```

var Shape myShape;
...
myShape ?blue;
    // draws in blue
myShape draw;
myShape ?red;

```

```

    // draws in red
myShape draw;
    // Square is a sub-object of Shape
var Square sqr = Square new;
...
sqr ?black;
    // draws in black
sqr draw;

```

Assume that draw of sub-prototypes use the color defined in Shape.

We could have got the same result as above by adding all of these methods to Shape textually. For example, method blue would be

```
fun blue { color: ff_Hex }
```

Regular objects may be used as parameters to selector body:.

```

object PrintBox
    implements ContextFunction<Box, Nil>

    fun bindToFunction: (Box newSelf) -> UFunction<Nil> {
        return { Out println: (newSelf get) }
    }
end

```

This object is added to Box as usual:

```

Box addMethod:
    selector: #print
    body: PrintBox;

```

There could be libraries of context objects that implement methods that could be added to several different prototypes. For example, there could be a Sort context object to sort any object that implements an interface

```

interface Indexable<T>
    fun at: Int -> Int
    fun at: Int put: T
    fun size -> Int
end

```

A context object used to add a method to an object could have more methods than just bindToFunction:.

```

object PrintFormattedBox
    implements ContextFunction<Box, Nil>

    fun bindToFunction: (Box newSelf) -> UFunction<Nil> {
        return { Out println: (format: (newSelf get)) }
    }

    /* one could declare a context function
       with one more method like format:
       this method fills the first positions
       with 0. Then

```

```

        format: 123
        should produce "0000000123"
    */
private fun format: (Int n) -> String {
    var strn = (n asString);
    return ("0000000000" trim: (10 - strn size)) + str
}
end

```

format: is a method that can only be used by the method print that is added to Box. It is like a private method of print.

Suppose you want to replace a method by a context function that calls the original method after printing a message. Using the Box prototype, we would like something like this:

```

object Box
    fun get -> Int { return value }
    fun set: (Int other) { value = other }
    Int value = 0
end

```

```

...
Box set: 0;
Box addMethod:
    selector: #get
    returnType: Int
    body: { (: Box self :)
        Out println: "getting 'value'";
        self get
    };

```

It is a pity this does not work. In a call “Box get” made after the call to addMethod: ..., the context function will be called. It prints

```
getting 'value'
```

as expected but then it calls get, which is a recursive call. There is an infinity loop. What we would like is to call the original get method. That cannot be currently achieved in Cyan. However, it will be possible if context functions are transformed into “*literal dynamic mixins*” (LDM) or “*literal runtime metaobjects*” (LRM). This feature is not yet supported by Cyan. But the description of it would be as follows.

The syntax of LRM’s would be the same as that of context functions except that “super” could be used as receiver of messages. Calls to super are calls to the original object. Then the code above can be written as

```

Box set: 0;
Box addMethod:
    selector: #get
    returnType: Int
    body: { (: Box self :)
        Out println: "getting 'value'";
        super get
    };

```

In this way a call “Box get” would print “getting 'value'” and the original get method would be called. Exactly what we wanted.

We are unaware of any language that allows literal runtime metaobjects. That would be one more innovation of Cyan.

This feature has not been introduced into Cyan because:

- (a) it seems to be difficult to implement (which may not be a good reason). The compiler being built generates Java code and literal runtime metaobjects probably demand code generation at runtime, which would be difficult with Java (although not impossible);
- (b) there are some questions on what is the type of a LDM/LRM. This is the same question of “what is the type of a mixin prototype?”.

10.12 Implementing r-Functions as u-Functions

There is a way of implementing every function as an u-function. It is only necessary to allocate all local variables used in functions in the heap. That is, not only the objects the local variables refer to are dynamically allocated. Space for the variables should also be put in the heap. Usually local variables are put in the stack. Then if local variable `n` of type `Int` is used inside a function, `n` will refer to an object that has a reference to an integer. There will be a double indirection. We will explain how to allocate variables in the heap using an example.

```
var Int n;  
n = 0;  
var Int k;  
k = n;  
assert: (n == 0);  
var b = { ++n };  
b eval;  
assert: (n == 1);
```

Since Cyan is targeted to the Java Virtual Machine, we will show the translation of this code to Java.

```
IntBox n = new IntBox();  
n.value = 0;  
int k;  
k = n.value;  
assert(n == 0);  
Closure00001 b = new Closure00001(n);  
b.eval(); // ++b.n.value  
assert(n == 1);
```

`IntBox` is just a box for an `int` value. It is this class that implements the double indirection.

```
class IntBox {  
    public int value;  
}
```

Each function such as `++n` is translated to a Java class with an `eval` method:

```

    // { ++n }
class Function00001 {
    public Function00001(IntBox n) { this.n = n; }
    public void eval() { ++n.value; }
    IntBox n;
}

```

The assignment `k = n` is translated into `k = n.value` since `k` is assigned the integer value of `n`. The Java class generated by `++n` contains an object of `IntBox`. Its instance variable `n` represents the variable `n` used inside the function, which should be a mirror of the local variable `n`. This is achieved by declaring both variables, the instance and the local variable, as objects of `IntBox`. Both variables refer to the same `IntBox` object. Changes in the value of the `n` variable in the Cyan code, be it the local variable or the instance variable, are translated as changes in the attribute `value` of this `IntBox` object. Since the `IntBox` variable is referred to by both variables, changes in it are seen by both variables.

Using this kind of function implementation, there would not be any runtime error in returning a function that accesses a local variable:

```

fun canWithdraw -> Function<Float, Boolean> {
    var limit = getLimit;
    return { (: Float amount :) ^amount < limit }
}

```

Since several functions would access the same variable, unusual objects can be dynamically created:

```

...
// n is a local Int variable
var Int n = 0;
var h = Hashtable<String, Function<Nil>> key: "inc" value: { ++n }
                                key: "dec" value: { --n }
                                key: "show" value: { Out println: n };
h["inc"] eval;
assert: (n == 1);
h["inc"] eval;
assert: (n == 2);
h["sub"] eval;
// prints 1
h["show"] eval;

```

The only disadvantage of allocating local variables accessed by functions in the heap would be efficiency. But in most cases in which functions are used the compiler could optimize the code. Most of the time functions are passed as parameters to methods of objects `Boolean` or to methods of another functions (such as `whileTrue:`), which do not keep any references to them. Therefore in all of these common cases the compiler would not allocate local variables in the heap.

The Cyan compiler will generate Java code. This language does not support pointers to local variables, which are needed in order to efficiently implement functions in Cyan. Neither do the Java Virtual Machine. Therefore the Cyan compiler will allocate in the heap, as shown above, all local variables accessed in functions without `%`. Unfortunately.

Chapter 11

Context Objects

A Cyan function becomes a closure at runtime for it can access variables from its context as in the example:

```
// sum the vector elements
var sum = 0;
v foreach: { (: Int x :) sum += x };
```

Here the sum of the elements of vector `v` is put in variable `sum`. But `sum` is not a local variable or parameter of the function. It was taken from the environment. Then to use a function it is necessary to bind (close over) the free variables to some variables that are visible at the function declaration. `self` is visible in the function and messages can be sent to it:

```
v foreach: { (: Int x :) sum += self calc: x };
```

Although functions are tremendously useful, they cannot be reused because they are *literal* objects. A function that accesses local and instance variables is specific to a location in the source code in which those variables are visible. Even if the programmer copy-and-past the function source code it may need to be modified because the variable names in the target environment may be different. A generalization of functions would make the free variables and the message sends to `self` explicit. That is what context objects do.

In Cyan it is possible to define a *context object* with free variables that can be bounded to produce a workable object. For example, the context object

```
object Sum( Int &sum ) extends Function<Int, Nil>
  fun eval: (Int x) {
    sum += x
  }
end
```

defines method `eval:` and uses a free `Int` variable `sum`. A message send

```
Sum eval: 5;
```

would use a compiler-initialized variable `sum` (default value 0). However, since it is a private variable, there would be no way of retrieving its value.

A context object cannot define any `init`, `init:`, `new`, `new:`, or `clone` methods. The only way of creating a context object is by using a `new:` method created by the compiler (this will soon be explained).

A free variable of a context object such as `Sum` can be bounded by method `bind`. The free variables should be given as parameters:

```

var Array<Int> v = {# 1, 2, 3 #};
...
var Int s = 0;
  // binds sum of Sum to local variable s
Sum bind: s;
v foreach: Sum;
assert: (s == 6);

```

or

```

var v = {# 1, 2, 3 #};
var Int s = 0;
v foreach: Sum(s);
assert: (s == 6);

```

The syntax `Sum(s)` means the same as

```
(Sum new: s)
```

which is the creation of an object from `Sum` passing `s` as a parameter. However, this is not a regular parameter passing — it is passing by reference as we will soon discover.

When the type of a context object parameter is preceded by `&`, the real argument should be a local variable. It cannot be a parameter of the current method or an instance variable.

11.1 Using Instance Variables as Parameters to Context Objects

In the last example, the free variables passed as parameters should be local variables of the method. They cannot be instance variables. Instance variables can only be passed as parameters if the parameter is prefixed with `*`.

```

object Sum(Int *sum) extends Function<Int, Nil>
  fun eval: (Int x) {
    sum += x
  }
end

```

```

object Test
  fun totalSum: (Array<Int> array ) {
    array foreach: Sum(total)
  }
  fun getTotal -> Int { ^total }
  Int total
end

```

The next example shows object `IntSet` and context object `ForEach`. This last one works as an “inner class” or a “nested class” of the former. Whenever method `getIter` of an object `Obj` of type `IntSet` is called, it returns a new object `ForEach` that keeps a reference to the instance variable `intArray` of `Obj`.

```

object ForEach(Array<Int> *array) implements Iterable<Int>
  fun foreach: (Function<Int, Nil> b) {
    0 ..< (array size) foreach: {
      (: Int index :)
    }
  }

```

```

        b eval: array[index]
    }
}
end

// a set of integers
object IntSet
  fun init {
    intArray = Array<Int> new
  }
  fun getIter -> ForEach {
    ^ForEach(intArray)
  }
  // methods to add, remove, etc.

  Array<Int> intArray
end

```

One could write

```

var set = IntSet new;
set add: 0 add: 1 add: 2;
var iter = set getIter;
iter foreach: {
  (: Int elem :)
  Out println: elem + " "
};

```

11.2 Passing Parameters by Copy

As with functions, it is possible to use % to mean “a copy of the value of *s*”. However, % should be put only before the parameter.

```

object DoNotSum(Int %sum)
  fun eval: (Int x) {
    sum = sum + x
  }
end
...

```

```

var Int s = 0;
v foreach: DoNotSum(s);
assert: (s == 0);

```

Here a copy of the value of *s*, 0, is passed as a parameter to the context object. This “parameter” is then changed. But the value of the original variable *s* remains unchanged. Parameters whose type is preceded by % will be called “*copy or % parameters*”. Parameters whose type is preceded by * are the “*instance variable parameters*” or * parameters. The ones preceded by & will be called “*reference parameters*” or & parameters.

A context object with a copy parameter may have any expression as real argument:


```
v foreach: DoNotSum(0);
{# 0, 1, 2 #} foreach: DoNotSum(Math factorial: 5);
```

Therefore, method parameters can be real arguments to `DoNotSum`. If no symbol is put before the parameter type of a context object, it is assumed that it is a copy parameter.

11.3 What the Compiler Does With Context Objects

The context object `Sum` is transformed by the compiler into a prototype

```
object Sum extends Function<Int, Nil>
  @prototypeCallOnly
  fun new: (Int &sum) -> Sum {
    var newSum = self primitiveNew;
    newSum bind: sum;
    return newSum
  }
  public bind: (Int &sum) {
    self.sum = sum
  }
  Int &sum
  fun eval: (Int x) {
    sum += x
  }
end
```

Symbol `&` put before a parameter means that the type of it is a “*reference type*”. It is the same concept as a pointer to a type in language C. To make Cyan type-safe, reference types can only be used in the declaration of parameters of context objects. But the compiler can use them as in the production of the above `Sum` prototype from the original `Sum` context object. By restricting the way reference types are used, the language guarantees that no runtime type error will ever happen due to a reference to a variable that is no longer in memory. In language C, one of these errors would be

```
int *f() { int n; return &n; }
void main() {
  printf("%d\n", *f());
}
```

A local variable `n` which is no longer in the stack would be referenced by expression “`*f()`”.

We can use `Sum(s)` to call the `new:` method of prototype `Sum` built by the compiler, as usual (See page 74). The compiler will take the code of prototype `DoNotSum` and transform it internally in the following object:

```
object DoNotSum
  @prototypeCallOnly
  fun new: (Int sum) -> DoNotSum {
    var newSum = self primitiveNew;
    newSum bind: sum;
    return newSum
  }
  fun bind: (Int sum) {
```

```

        self.sum = sum
    }
    var Int sum
    fun eval: (Int x) {
        sum += x
    }
end

```

An instance variable parameter is transformed by the compiler, internally, into two variables: a reference to the variable and a reference to the object in which it is. So prototype `ForEach` will be transformed into

```

object ForEach
    @prototypeCallOnly
    fun new: (Array<Int> &array, Any otherSelf) -> ForEach {
        var newObj = self primitiveNew;
        newObj bind: array;
        self.otherSelf = otherSelf;
        return newObj
    }
    @checkSelfBind fun bind: (Array<Int> &array, Any otherSelf) {
        self.array = array;
        self.otherSelf = otherSelf;
    }
    Array<Int> &array
    Any otherSelf

    fun foreach: (Function<Int, Nil> b) {
        0..< (array size) foreach: {
            (: Int index :)
            b eval: array[index]
        }
    }
end

```

An expression

```
ForEach(intArray)
```

is transformed internally by the compiler into

```
ForEach(intArray, self)
```

It is necessary to pass `self` as parameter in order to prevent the garbage collector to free the memory of the object while there is a pointer to one of its instance variables, `intArray`. If `self` is not passed as parameter, they may be the case that an object of `ForEach` has a reference to an instance variable `intArray` of a `IntSet` object and there is no other reference to this object. Then the garbage collector could free the memory allocated to this object.

Metaobject `checkSelfBind` checks, in this example, whether the second real argument to `bind:` is `self`. There would be a compiler error if it is not:

```
var f = ForEach bind: intArray, 0
```

0 is a subtype of `Any`. But it is not `self`.

Object `ForEach` could have been implemented as a regular object because:

- (a) instance variable `intArray` of `IntSet` always refer to the same object. Therefore `intArray` could be passed by copy to `ForEach`;
- (b) `ForEach` does not assign a new object to `intArray`.

A copy or % parameter of a context object may be preceded by keywords `public`, `protected`, or `private` to mean that the parameter should be declared as an instance variable with that qualification.

```
// prod is also a copy parameter
object Test(public Int sum, protected Int prod) extends Function<Int, Nil>
  fun eval: (Int elem) {
    sum = sum + elem;
    prod = prod*elem
  }
  fun getProd -> Int { ~prod }
end
...
var s = 0;
var p = 1;
{# 1, 2, 3 #} foreach: Test(s, p);
  // call to public method sum
Out println: "Sum is #{Test sum}";
Out println: "Product is #{Test getProd}";
...
```

The default qualifier is `private`. Prototype `Test` would be transformed by the compiler into

```
object Test extends Function<Int, Nil>
  @prototypeCallOnly
  fun new: (Int sum, Int prod) -> Test {
    var newText = primitiveNew;
    newText bind: sum, prod;
    return newText
  }
  public bind: (Int sum, Int prod) {
    self.sum = sum;
    self.prod = prod
  }
  fun eval: (Int elem) {
    sum = sum + elem;
    prod = prod*elem
  }
  fun getProd -> Int { ~prod }
  public Int sum
  protected Int prod
end
```

Reference (&) and instance variable (*) parameters are always private.

A context object that only has copy parameters is a regular prototype. There is just one difference: the compiler adds an `new:` method with the parameters of the context object. This method initializes the instance variables that have the same name as the parameters. See the `DoNotSum` example above.

11.4 Type Checking Context Objects

There are two kinds of context objects:

- (a) the ones with at least one reference parameter such as `Sum`. These are called *restricted* context objects, r-co for short;
- (b) the ones with no reference parameter. These have one or more instance variable or copy parameters (with `*` or `%`). These are called *unrestricted* context objects, u-co for short.

There is no restriction on the use of unrestricted context objects (as expected!). They can be types of variables, instance variables, return values, and parameters. u-co are a generalization of u-functions.

Restricted context objects are a generalization of r-functions. Both suffer from the same problem: a context object could refer to a dead local variable:

```
var Sum mySum;
var b = {
  var Int sum1 = 0;
  mySum = Sum(sum1);
};
b eval;
mySum eval: 1;
```

The message send “`b eval`” makes `mySum` refer to a context object that has a reference to `sum1`. In the last message send, “`mySum eval: 1`”, there is an access to `sum1`, which no longer exists.

Another error would be to return a r-co from a method:

```
object Program
  fun run {
    {# 1, 2, 3 #} foreach: makeError
  }
  fun makeError -> Sum {
    var sum = 0;
    return Sum(sum);
  }
```

Here `Sum(sum)` has a reference to a local variable `sum`. When `foreach:` calls method `eval:` of the object `Sum(sum)`, variable `sum` is accessed causing a runtime error.

To prevent this kind of error, r-co have exactly the same set of restrictions as r-functions. In particular, the compiler would point an error in the assignment “`mySum = Sum(sum1)`” of the example above.

A context object that does not inherit from anyone inherits from `Any`, as usual. Both r-co’s and u-co’s can inherit from any prototype and implement any interface. However, there are restrictions on assignments mixing restricted and unrestricted types. A r-co RCO that inherits from an unrestricted prototype P or implements an unrestricted interface I is not considered a subtype of P or I. That is, if p is a variable of type P or I, an assignment

```
p = RCO;
```

is illegal.

Apart from the rules for type checking, context objects are regular objects. For example, they may be abstract, have shared variables, and inherit from other prototypes. Inheritance demands some explanations. When a context object with an instance variable or reference parameter x is inherited by another context object, this last one should declare x in its list of parameters with the same symbol preceding the parameter (`%` or `&`) as the super-prototype. x should precede the parameters defined only

in the sub-prototype. After the keyword “extends” there should appear the super-prototype with its parameters.

```
object A(Int &x)
  ...
end

object B(Int &x, Int %y, String &z) extends A(x)
  ...
end
```

Since A is a r-co, B is a r-co too. A context object cannot be inherited by a regular prototype.

Note that context objects that use only copy parameters are regular prototypes. Therefore sub-prototypes need not to obey the rules given above. The sub-prototype does not even need to be a context prototype.

A context object can also be a generic object. Sum can be generalized:

```
object Sum<T>(T &sum) extends Function<T, Nil>
  fun eval: (T x) {
    sum = sum + x
  }
end

...
var intSum = 0;
var Float floatSum = 0;
var String abc = "";
{# 1, 2, 3 #} foreach: Sum<Int>(intSum);
{# 1.5, 2.5, 1 #} foreach: Sum<Float>(floatSum);
{# "a", "b", "c" #} foreach: Sum<String>(abc);
assert: (floatSum == 5);
assert: (intSum == 6);
assert: (abc == "abc");
```

11.5 Adding Context Objects to Prototypes

Section 10.11 explain how to use the addMethod: ... grammar method of Any to add methods to a prototype.

```
fun (addMethod:
  (selector: String ( param: (Any)+ )?
  )+
  (returnType: Any)?
  body: Any)
```

A context object can be used instead of a context function. One has just to extends the appropriate ContextObject prototype.

```
object Car
  fun addDoorColor {
    leftDoor addMethod:
```

```

        selector: #getColor
        returnType: Int
        body: GetColor(color);
    leftDoor addMethod:
        selector: #setColor
        param: Int
        body: SetColor(color);
    }
    ...
    public Door leftDoor, rightDoor
    Int color
end

object GetColor(Int *color)
    implements ContextFunction<Door, Int>

    fun bindToFunction: (Door newSelf) -> UFunction<Int> {
        return { ^color }
    }
end

object SetColor(Int *color)
    implements ContextFunction<Door, Int, Nil>

    fun bindToFunction: (Door newSelf) -> UFunction<Int, Nil> {
        return { (: Int newColor :) color = newColor }
    }
end

```

After

```
Car addDoorColor
```

the left door will share a color with the car. Changes in one will reflect in the other.

11.6 Passing Parameters by Reference

Some languages such as C++ support passing of parameters by reference. In this case, changes in the parameter are reflected in the real argument, which should be a variable (it cannot be an expression). Cyan does not support directly this construct. However, it can be implemented using the generic context object Ref:

```

object Ref<T>(T &v)
    fun value -> T { ^v }
    fun value: (T newValue) { v = newValue }
end

```

Now if you want to pass a parameter by reference, use Ref:

```

private object CalcArea
    // it is as if parameter to selector area: were by reference
    fun squareSide: (Float side) area: (Ref<Float> refSqrArea) {

```

```

        // by calling method value: we are changing the parameter
        // of the context object
    refSqrArea value: side*side
}
end

public object Program
    fun run {
        var side = In readFloat;
        var Float sqrArea;

        /* encapsulate the reference parameter inside a
           context object. That is, use "Ref<Float>(sqrArea)"
           instead of just "sqrArea".
           Local variable "sqrArea" is changed inside
           method squareSide:area: of prototype CalcArea when message
           value: is sent to refSqrArea
        */
        CalcArea squareSide: side area: Ref<Float>(sqrArea);

        Out println: "Square side = #side";
        Out println: "area = #sqrArea"
    }
end

```

Of course, the “passing by reference” syntax in Cyan is not straightforward. However, it has two advantages:

- (a) it does not need a special syntax;
- (b) and, most importantly, it is type-safe. Context objects use the same rules as the static functions of Cyan. That means, for example, that an instance variable of prototype `Calc` cannot refer to a parameter of type `Ref<Float>`. That guarantees there will never be a reference to local variable of `run` of `Program` after this method is removed from the stack.

There will never be an error in Cyan equivalent to the following error in a C program, in which pointer `mistake` refers to a local variable that has been removed from the stack.

```

#include <stdio.h>

const float pi = 3.141592;

float *mistake;
void calc(float radius, float *area) {
    mistake = area;
    *area = pi*radius*radius;
}
void run() {
    float area;
    calc(1, &area);
}

```

```

}
float useStack() { float ten = 10; return area; }
int main() {
    run();
    useStack();
    // mistake refers to a variable that has been
    // removed from the stack
    // 10 is printed in some compilers
    printf("%f\n", *mistake);
    return 0;
}

```

11.7 Should Context Objects be User-Defined?

An alternative definition of Cyan could get rid of context objects. They could not be defined as shown in this text. Instead, one could use *reference types* like `&Int` to declare a restricted prototype directly. So the programmer could define a prototype like

```

object Sum extends Function<Nil, Int>
  fun new: (Int &sum) -> Sum {
    var newSum = self.primitiveNew;
    newSum.bind = sum;
    return newSum
  }
  public bind: (Int &sum) {
    self.sum = sum
  }
  Int &sum
  fun eval: (Int x) {
    sum += x
  }
end

```

This new version of Cyan would have a concept called “*restricted type*” defined inductively as:

- (a) a *reference type* is a *restricted type*;
- (b) any prototype that declares an instance variable of a restricted type is a *reference type*.

All the restriction on the use and type checking defined nowadays for context objects would apply to *reference types*.

With this feature, the programmer herself would explicitly create her own context objects. And this alternative Cyan would solve a problem related to grammar methods presented at the end of Chapter 9.9 at page 177. The solution comes from the fact that an array of a reference type would be a reference type too. Idem for tuples and unions: `Array<Function<Int>>`, `Tuple<String, Function<Nil>>`, and

`Tuple<Int, Union<Function<Int, Char>, Char>, String>`
 would be reference types.

11.8 More Examples

The example of trees of page 74 can be made even more compact with context objects:


```

object Tree
end

object BinTree(public Tree left, public Int value, public Tree right) extends Tree
end

object No(public Int value) extends Tree
end
...

var tree = BinTree( No(-1), 0, BinTree(No(1), 2, No(3)) );
Out println: ((tree left) value);

```

When the compiler finds a class like `BinTree`, it creates a regular class with public instance variables `left`, `value`, and `right`:

```

object BinTree extends Tree
  @prototypeCallOnly
  fun new: (Tree left, Int value, Tree right) -> BinTree {
    var newObj = self primitiveNew;
    newObj bind: left, value, right;
    return newObj
  }
  public bind: (Tree left, Int value, Tree right) {
    self.left = left;
    self.value = value;
    self.right = right;
  }
  public Tree left
  public Tree value
  public Tree right
end

```

Suppose there is a sport `Car` prototype that has two doors, `left` and `right`. The colors of these doors should always be the same as the main color of the car. One way of assuring that is declaring in the `CarDoor` prototype an instance variable that is a reference (a C-language pointer) to the instance variable of the `Car` that keeps the color. Since `Cyan` does not have C-like pointers, we can use context objects.

```

object CarDoor(public Int *color)
  ...
end

object Car
  fun init {
    leftDoor = CarDoor(_color);
    rightDoor = CarDoor(_color);
  }
  fun color: (Int newColor) { _color = newColor }
  fun color -> Int { ^ _color }
  Int _color

```

```

    public CarDoor leftDoor, rightDoor
end

...
Car color: 255;
    // prints "color = 255"
Out println: "color = #{(Car leftDoor) color}";

(Car rightDoor) color: 0;
    // prints "color = 0"
Out println: "color = #{Car color}";

```

`inject:into:` methods in Smalltalk are used to accumulate a result over a loop. For example, `var sum = (1 to: 10) inject: 0 into: { (: Int total, Int elem :) total + elem }` accumulates the sum from 1 to 10. Initially `total` receives 0, the argument to the selector `inject:.` Then the function is called passing `total` and the current index (from 1 to 10). In each step, the value returned from the function, `total + elem`, is assigned to `total` (Smalltalk returns the last block expression).

The basic types of Cyan support a Smalltalk-like `inject` method and another form made to be used with context objects.

```

object InjectInto<T>(T %total) extends InjectObject<T>
    fun eval: (T elem) {
        total = total + elem
    }
    fun result -> T {
        ^total
    }
end

```

Now the total is kept in the context object and we can write

```

var inj = InjectInto<Int>(0);
1 to: 10 do: inj;
Out println: "Sum = #{inj result}";
print the sum of the numbers from 1 to 10.

```

Chapter 12

The Exception Handling System

Exception handling systems (EHS) allow the signalling and handling of errors or abnormal situations. There is a separation from the detection of the error and its treatment which can be in different methods or modules. The exception handling systems of almost all object-oriented languages are very similar. An exception is thrown by a statement such as “`throw e`” or “`raise e`” and caught by one or more catch clauses. We will show an example in Java. Assume there is a `MyFile` class with methods for opening, reading and closing a file and that methods `open` and `readCharArray` of this class may throw exceptions `ExceptionOpen` and `ExceptionRead`.

```
1 char []charArray;
2 MyFile f = new MyFile("input.txt");
3 try {
4     f.open();
5     charArray = f.readCharArray();
6     if ( charArray.length == 0 )
7         throw new ExceptionZero();
8 } catch ( ExceptionOpen e ) {
9     System.out.println("Error opening file");
10 }
11 catch ( ExceptionRead e ) {
12     System.out.println("Error reading file");
13 }
14 finally {
15     f.close();
16 }
```

An exception is thrown by statement `throw` (see line 7). We can also say that an error is signalled by a `throw` statement. The class of the object following `throw` should be a direct or indirect subclass of class `Throwable`. In this example, all statements that can throw exceptions are put in a `try` block (which is between lines 4 and 7). The exceptions thrown inside the `try` block at runtime will be treated by the `catch` clauses that follow the `try` block. There are two `catch` clauses and one `finally` clause. Each `catch` clause accepts a parameter and treats the error associated to that parameter. Therefore

```
catch ( ExceptionOpen e ) { ... }
```

will treat the error associated to the operation of opening a file.

If file `f` cannot be read, method `readCharArray` throws exception `ExceptionRead` with a statement `throw new ExceptionRead(filename);`

After that, the runtime system starts a search for an appropriate handler for this exception. A handler

is a piece of code, given in a `catch` clause, that can treat the exception. This search starts in method `readCharArray` which does not have any catch clauses. It continues in the stack of called methods. Therefore an appropriate handler (or catch clause) is looked for in the code above. The runtime system checks whether the first catch clause can accept an object of `ExceptionRead`, the one thrown by the `throw` statement. It cannot. Then it checks whether the second catch clause can accept this object as parameter. It can. Then method `readCharArray` is terminated and control is transferred to the catch clause

```
catch ( ExceptionRead e ) {
    System.out.println("Error reading file");
}
```

Parameter `e` receives the object “`new ExceptionRead(filename)`” which was the parameter to statement `throw` and the body of the clause is executed. After that the execution continues in the `finally` clause, which is always executed — it does not matter whether an exception is thrown or not in the try block. When an exception is thrown, the stack of called methods is unwound till an appropriated catch clause is found and the control is transferred to this catch clause.

The exception handling system (EHS) of Cyan is similar in several aspects of the model just described. However, it was based on the object-oriented exception handling system of Green [dOGaa] and it is object-oriented in nature. The throwing of an exception is a message send, exception treatment (catch clauses) can be put in prototypes and inherited, and polymorphism applies to exception treatment. All the arsenal of object-oriented programming can be used with exception signalling and treatment, which is not possible possible, to our knowledge, in other languages but Green. The exception handling system (EHS) of Cyan goes well beyond that of Green which is awkward to use if local variables should be accessed to treat the error. In Cyan the EHS is both easy to use and powerful. However, it is not a checked exception system like that of Java or Green. An exception may be thrown and not caught as in C++ or C#.

The Java example in Cyan would be

```
1 var Array<Char> charArray;
2 var f = MyFile new: "input.txt";
3 {
4     f open;
5     charArray = f readCharArray;
6     if charArray size == 0 {
7         throw: ExceptionZero
8     }
9 } catch: { (: ExceptionOpen e :) Out println: "Error opening file" }
10 catch: { (: ExceptionRead e :) Out println: "Error reading file" }
11 finally: {
12     f close
13 }
```

An exception is thrown by sending message `throw:` to `self` as in line 7:

```
throw: ExceptionZero;
```

`throw:` is a final method defined in `Any` (therefore inherited by all prototypes). `ExceptionZero` is a prototype that inherits from `CyException`, the super-prototype of all exception objects. Since this exception does not demand any useful additional information, the prototype does not have any instance variables:

```
object ExceptionZero extends CyException
end
```

Every exception prototype should inherit from `CyException`, which inherits from `Any` and does not define any methods.

In the above Cyan example, function

```
{ f open; charArray = ... }
```

receives message

```
catch: ... catch: ... finally: { f close }
```

at runtime. The method executed will be a grammar method (more about that will soon be explained). This method calls the function body (sends message `eval` to it) and catches the exceptions it throws. That is almost the same as in the Java code. When an exception is thrown in the function body, as `ExceptionRead`, the runtime system searches for an adequate handler in the parameters to the `catch:` methods. First it checks whether method `eval:` of the first function,

```
{ (: ExceptionOpen e :) Out println: "Error opening file" }
```

can accept an object of `ExceptionRead` as real argument. It cannot. Then the search continues in the second `catch:` selector. Since

```
{ (: ExceptionRead e :) Out println: "Error reading file" }
```

can accept a `ExceptionRead` object, message `eval` is sent to this function. Then the function that is parameter to `finally:` is called and the execution continues in the first statement after the original function. This works exactly the same as the exception system of Java/C++ and many other object-oriented languages. In Cyan there may be one or more `catch:` selectors and an optional `finally:` selector. Every `catch:` selector accepts as argument an object that has at least one method

```
eval: (E e)
```

in which `E` is a prototype that inherits from `CyException` (directly or indirectly). Functions

```
{ (: ExceptionOpen e :) Out println: "Error opening file" }
```

```
{ (: ExceptionRead e :) Out println: "Error reading file" }
```

satisfy these requirements. For example, the first function has a method

```
eval: (ExceptionOpen e) { Out println: "Error opening file" }
```

It is not necessary that `E` be a function or be a sub-prototype of any function. The `catch:` selectors may receive an r-function as parameter. This does not cause any runtime errors because the method neither store a reference to the object that is the real argument nor passes a reference to this object to another method.

The parameter to method `throw:` should not be a restricted context object. That is checked by metaobject `checkThrow` attached to this method. If a restricted context object is parameter to `throw:`, a runtime error could occur:

```
object ExceptionContext(public Int &number) extends CyException
end
```

```
object Test
  fun run {
    {
      test
    } catch: { (: ExceptionContext e :)
      // "number" refer to "n" which does not
      // exist anymore
    }
  }
end
```

```

        Out println: (e number)
    }
}
fun test {
    var n = 0;
    throw: ExceptionContext(n)
}
end

```

12.1 Using Regular Objects to Treat Exceptions

Each `catch:` selector may receive as argument an object that has more than one `eval:` method.

```

object ExceptionCatchFile
    fun eval: (ExceptionOpen e) { Out println "Error opening file" }
    fun eval: (ExceptionRead e) { Out println "Error reading file" }
    fun eval: (ExceptionWrite e) { Out println "Error writing to file" }
end

```

Prototype `ExceptionCatchFile` treats all errors associated to opening, reading, and writing to files (but not to closing a file). This kind of object, to treat exceptions, will be called *catch objects*. It can be used as

```

var Array<Char> charArray;
var f = MyFile new: "input.txt";
{
    f open;
    charArray = f readCharArray;
    if charArray size == 0 {
        throw: ExceptionZero
    }
} catch: ExceptionCatchFile
finally: {
    f close
}

```

When an exception is signaled in the function, the runtime system starts a search for an `eval:` method (a handler) in the nearest argument to `catch:`, which is `ExceptionCatchFile`. Supposing that there was a read error, the correct `eval:` method should accept a `ExceptionRead` object as parameter. The runtime system searches for the `eval:` method in `ExceptionCatchFile` using the same algorithm used for searching for a method after a message is send to an object. That is, the runtime system tries to send message `eval:` with a `ExceptionRead` as argument to object `ExceptionCatchFile`. By the regular algorithm, the second textually declared method of `ExceptionCatchFile`,

```

    fun eval: (ExceptionRead e) { Out println "Error reading file" }

```

is found and called. After that the function that is argument to selector `finally:` is called and computation continues in the first statement after the outer function in the example.

12.2 Selecting an eval Method for Exception Treatment

A Cyan program starts its execution in a method called `run` of a prototype designed at compile-time. For this example, suppose this prototype is `Program`. To start the execution, method `run` is called inside a function that receives a `catch:` message:

```
{
  Program run: args
} catch: RuntimeCatch;
```

Method `eval:` of prototype `RuntimeCatch` just prints the stack of called methods:

```
object RuntimeCatch
  fun eval: (CyException e) {
    /* prints the stack of called methods and ends the program
    */
  }
  ...
end
```

Maybe we may will add a `finally:` selector to the `catch:` message allowing some code to be executed before the program ends.

When a message with at least one `catch:` selector is sent to a function, a grammar method is called. We will call this grammar method `catch-finally` (this is just a name for explaining this text). Method `catch-finally` pushes the parameters to `catch:` in a stack `CatchStack` in the reverse order in which they appear in the call. So

```
{
  ...
} catch: c1
  catch: c2
  catch: c3;
```

pushes `c3`, `c2`, and `c1` into the stack, in this order. Therefore `c1` is in the top. When an exception is thrown by the message `send throw: obj`, method `throw:` of `Any` searches the stack `CatchStack` from top to bottom until it finds an `eval:` method that accepts `obj` as parameter. Inside each stack object the search is made from the first declared `eval:` method (in textual order) to the last one. `CatchStack` is a prototype that just implements a stack. It cannot be changed by regular programming. But programmers will be able to inspect its contents:

```
object CatchStack
  public fun asArray -> Array<Any> { ^stack clone }
  private fun push: (Any catchObj) { ... }
  private fun pop -> Boolean { ... }
  ...
  var Array<Any> stack
end
```

Consider the catch objects¹ and the example that follow.

```
// number < 0, == 0, > 1000, or even
private object ExceptionNum extends CyException
```

¹Objects with `eval:` methods that treat exceptions.

```

end

    // when the number is == 0
private object ExceptionZero extends ExceptionNum
end

    // when the number is < 0
private object ExceptionNeg extends ExceptionNum
end

    // when the number is > 1000
private object ExceptionBig extends ExceptionNum
end

    // when the number is even
private object ExceptionEven extends ExceptionNum
end

private object CatchZeroBig
    fun eval: (ExceptionZero e) {
        Out println: "zero number";
    }
    fun eval: (ExceptionBig e) {
        Out println: "big number";
    }
end

private object CatchNeg
    fun eval: (ExceptionNeg e) {
        Out println: "negative number";
    }
end

private object CatchEven
    fun eval: (ExceptionEven e) {
        Out println: "even number";
    }
end

private object CatchNum
    fun eval: (ExceptionNum e) {
        Out println: "number < 0, == 0, > 1000, or even";
    }
end

object Program
    const Int MaxN = 1000

```



```

fun run: Array<String> args {
    // 1
    var n = In readInt;
    { // 2
        process: n
    } catch: CatchZeroBig
        catch: CatchEven
        catch: CatchNum;
    // 5
    Out println: "this is the end"
}
private fun process: (Int n) {
    { // 3
        check: n;
        if n > MaxN {
            throw: ExceptionBig
        }
    } catch: CatchNeg
    // 6
}
private fun check: (Int n) {
    // 4
    if n == 0 {
        throw: ExceptionZero
    };
    if n < 0 {
        throw: ExceptionNeg
    };
    if n%2 == 0 {
        throw: ExceptionEven
    }
}
}
end

```

There are four exceptions, `ExceptionZero`, `ExceptionNeg`, `ExceptionBig`, and `ExceptionEven` that inherit from `ExceptionNum` and four catch objects, `CatchZeroBig`, `CatchEven`, `CatchNeg`, and `CatchNum`. The program execution starts at point “// 1”. At line // 2, message `catch:catch:catch:` has been send and the function that has just “`process: n`” has been called. At point // 2, `CatchStack` has objects `CatchNum`, `CatchEven`, and `CatchZeroBig` (last on top).

Inside the function that starts at // 2, if message “`throw: exc`” is sent to `self`, the search for a method would start at `CatchZeroBig` and proceeds towards `CatchNum` at the bottom of the stack. First method `throw:` would check whether object `exc` is sub-object of `ExceptionZero`. If it is not, it would test whether object `exc` is a sub-object of `ExceptionBig`. If it is not, the search would continue in `CatchEven`.

At line marked as // 3, object `CatchNeg` has already been pushed into the stack `CatchStack`. At point // 4 in the code, if statement

```

    throw: ExceptionEven

```

is executed, there is a search for an `eval:` method that can accept `ExceptionEven` as parameter, starting

at the `CatchNeg` object. This method is found in object `CatchEven` pushed in the `run:` method. Therefore control is transferred to the first statement after the message send

```
{ // 2
  process: n
} catch: CatchZeroBig
  catch: CatchEven
  catch: CatchNum;
```

which is “`Out println: "this is the end"`”. This is exactly like the exception handling system of almost all object-oriented languages.

Before returning, the `throw:` method of `Any` removes the objects pushed into `CatchStack` together and after `CatchEven`.

Every function of type `Function<Nil>` or `UFunction<Nil>` has a method

```
@checkCatchParameter
fun ((catch: Any)+ finally: Function<Nil>) t {
  ...
}
```

responsible for catching exceptions. The metaobject `checkCatchParameter` attached to this method checks whether each parameter to a `catch:` selector has at least one `eval:` method, each of them accepting one parameter whose type is sub-prototype of `CyException`.

12.3 Other Methods and Selectors for Exception Treatment

Functions of type `Function<Nil>` or `UFunction<Nil>` have a method `hideException` that just eats every exception thrown in them:

```
n = 0;
{
  n = (In readLine) asInt
} hideException;
```

Of course, this method should be rarely used.

Selectors `retry` or `retry:` may be used after all `catch:` selectors in order to call the function again if an exception was caught by any object that is argument to any of the `catch:` selectors. If selector `retry:` is used, it should have a function as parameter that is called before the main function is called again.

```
// radius of a circle
Float radius;
{
  radius = In readFloat;
  if radius < 0 {
    throw: ExceptionRadius(radius)
  }
} catch: CatchAll
  retry: {
    Out println: "Negative radius. Type it again"
  };
```

CatchAll has a method

```
fun eval: (CyException e) { }
```

that catches all exceptions. This prototype is automatically included in every file. It belongs to package cyan.lang.

One can just write `retry:` without any `catch:` selectors. If any exception is thrown in the function, the `eval` method of the argument to `retry:` is called and the function is called again. If `retry` is used, the function is called again if an exception is thrown in the function.

```
// radius of a circle
var Float radius;
{
  radius = In readFloat;
  if radius < 0 {
    throw: ExceptionRadius(radius)
  }
  else if radius == 0 {
    // end of input
    return 0
  }
} retry: {
  Out println: "Negative radius. Type it again"
};
```

Selector `tryWhileTrue:` may be put after the `catch:` selectors in order to control how many times the function is retrieved. The argument to `tryWhileTrue:` should be a `Function<Boolean>` function. If an exception was thrown in the function and the argument to `tryWhileTrue:` evaluates to `true`, the function is called again.

```
numTries:= 0;
{
  // may throw an exception ExceptionConnectFail
  channel connect;
  ++numTries;
} catch: CatchAll
  tryWhileTrue: {^ numTries < 5 };
```

The above code tries to connect to a channel five times. Each time the connection fails an exception is thrown by method `connect`. Each time the function after `tryWhileTrue:` is evaluated. In the first five times it returns `true` and the main function is called again. If no exception is thrown by `connect`, the argument to `tryWhileTrue:` is not called. Again, the `catch:` selectors are optional. Selector `tryWhileFalse:` is similar to `tryWhileTrue`.

Prototype `CatchIgnore` could be used instead of `CatchAll`:

```
object CatchIgnore<T>
  fun eval: T { }
end

...
numTries:= 0;
{
  // may throw an exception ExceptionConnectFail
```

```

    channel connect;
    ++numTries;
} catch: CatchIgnore<ExceptionConnectFail>
  tryWhileTrue: { ^ numTries < 5 };

```

This example can be made more compact with the use of a context object to count the number of attempts:

```

object Times(Int %numTries) extends UFunction<Boolean>
  fun eval -> Boolean {
    --numTries;
    return numTries > 0;
  }
end

...
{
  // may throw an exception ExceptionConnectFail
  channel connect;
} tryWhileTrue: Times(5);

```

A future improvement to the EHS of Cyan would be to make it support features of the EHS of Common Lisp (conditions and restarts). That would be made by allowing communication between the error signaling and the error handling. This could be made using a variable “exception”. A catch object could have other meaningful methods besides “eval: T”. For example, a catch object could have an “getInfo” method describing the error recovery to be chosen afterwards:

```

object CatchStrategy
  fun getInfo -> CySymbol { ^ #retry }
end

object Test
  fun test {
    {
      connectToServer;
      buildSomething
    } catch: CatchStragegy
  }
  fun connectToServer {
    {
      var Boolean fail = true;
      ...
      // if connection to server failed, signal
      // an exception
      if fail {
        throw: ExceptionConnection
      }
    }
  } catch: { (: ExceptionConnection e :)
    // if connection to server failed,
    // consult getInfo for advice.
  }

```

```

        if exception getInfo == #retry {
            connectToServer
        }
    }
}
...
end

```

Maybe there should be another method that obeys automatically instructions given by objects like `CatchStrategy`. Maybe `catch` itself should automatically retry when “exception `getInfo`” demands it:

```

fun connectToServer {
    {
        var Boolean fail = true;
        ...
        // if connection to server failed, signal
        // an exception
        if fail {
            exception eval: ExceptionConnection
        }
    } catch: CatchIgnore<ExceptionConnection>
}

```

12.4 Why Cyan Does Not Support Checked Exceptions?

Cyan does not support checked exceptions as Java in which the exceptions a method may throw are described in its declaration:

```

// this is how method "check" of Program
// would be declared in Java
private void check(int n)
    throws ExceptionZero, ExceptionNeg,
           ExceptionEven {
    // 4
    if ( n == 0 )
        throw new ExceptionZero();
    if ( n < 0 )
        throw new ExceptionNeg();
    if ( n%2 == 0 )
        throw new ExceptionEven();
}

```

Here method `check` may throw exceptions `ExceptionZero`, `ExceptionNeg`, and `ExceptionEven`. We could add a syntax for that in Cyan following language Green [dOGaa]:

```

private fun check: (Int n)
    EvalZeroNegEven exception {
    // 4
    if n == 0 {

```

```

        exception eval: ExceptionZero
    };
    if n < 0 {
        exception eval: ExceptionNeg
    };
    if n%2 == 0 {
        exception eval: ExceptionEven
    }
}

```

Pseudo-variable `exception` would be declared after all regular method parameters. Inside the method this variable is type-checked as a regular variable. Then there would be an error if there was a statement

```
exception eval: ExceptionRead
```

in method `check` because there is no `eval: method` in `EvalZeroNegEven` that can accept a `ExceptionRead` object as parameter. Interface `EvalZeroNegEven` is

```

interface EvalZeroNegEven
    fun eval: ExceptionZero
    fun eval: ExceptionNeg
    fun eval: ExceptionEven
end

```

Green employs a mechanism like this, which works perfectly in a language without functions.

But think of method `ifTrue:` of functions of types `Function<Boolean, Nil>` and `UFunction<Boolean, Nil>`:

```

fun ifTrue: (Function<Nil> b)
    T exception {
        if self == true {
            b eval
        }
    }
}

```

What is the type `T` of exception? In

```

(i < 0) ifTrue: {
    throw: ExceptionRead;
}

```

`T` should be

```

interface InterfaceExceptionRead
    fun eval: ExceptionRead
    // possibly more methods
end

```

But in another call of this method `T` should be different:

```

(i <= 0) ifTrue: {
    if openError {
        throw: ExceptionOpen
    }
    else if i == 0 {
        throw: ExceptionZero
    }
}

```

```

    }
}

```

In this case T should be

```

interface InterfaceOpenExceptionZero
    fun eval: ExceptionOpen
    fun eval: ExceptionZero
    // possibly other methods
end

```

Then the type of T depends on the exceptions the function may throw. We have a solution for that but it is too complex to be added to a already big language. Without explaining too much, method `ifTrue:` would be declared as

```

fun ifTrue: (Function<Nil> b)
    (b getMethod: "eval") .exception exception {
    if self == true {
        b eval
    }
}

```

The declaration means that the type of `exception` in `ifTrue:` is the type of variable `exception` of the method `eval` of function `b` at the call site. If `ifTrue:` could throw exceptions by itself, these could be added to the type “(b getMethod: "eval") .exception” using the type concatenator operator “++” (introduced just for this use here).

For short, we could have checked exceptions in Cyan but it seems they are not worthwhile the trouble.

12.5 Synergy between the EHS and Generic Prototypes

Generic prototype instantiations can be used as parameters to `catch:` message sends. With them, one can reuse code for common tasks as shown in the following example.

```

object CatchExit<T>
    fun eval: (T e) {
        Out println: "Fatal error";
        System exit
    }
end

object CatchWarning<T>
    fun eval: (T e) {
        Out println: "Exception " + (T prototypeName) + " was thrown"
    }
end

...

{
    line = In readLine;
    if line size == 0 {

```

```

    throw: ExceptionEmptyLine
} else if line size > MaxLine {
    throw: ExceptionLineTooBig(line)
};
Out println "line = " + line
} catch: CatchExit<ExceptionLineTooBig>
    catch: CatchWarning<ExceptionEmptyLine>;

```

Object `CatchExit<ExceptionLineTooBig>` treats exception `ExceptionLineTooBig` because it has an `eval:` method that accepts this exception as parameter. This method prints an error message and ends the program execution.

Object `CatchWarning<ExceptionEmptyLine>` treats exception `ExceptionEmptyLine`. Method `eval` of this object just prints a warning message.

Generic object `CatchIgnore` accepts any number of parameters up to ten. The `eval:` methods of this object do nothing. The definition of `CatchIgnore` with two parameters is

```

object CatchIgnore<T1, T2>
    fun eval: T1 { }
    fun eval: T2 { }
end

```

If we want to ignore two exceptions and treat a third one, we can write something like

```

{
    line = In.readLine;
    if line size == 0 {
        throw: ExceptionEmptyLine
    } else if line size > MaxLine {
        throw: ExceptionLineTooBig(line)
    } else if line[0] == ' ' {
        throw: ExceptionWhiteSpace
    };
    Out.println "line = " + line
} catch: CatchIgnore<ExceptionLineTooBig, ExceptionEmptyLine>
    catch: { (: ExceptionWhiteSpace e :)
        Out.println: "line cannot start with white space";
        System.exit
    };

```

With generic prototypes, it is easy to implement the common pattern of encapsulating some exceptions in others. When an exception `Source` is thrown, a `catch:` method captures it and throws a new exception from prototype `Target`.

```

object ExceptionConverter<Source, Target>
    fun eval: (Source e) {
        throw: Target()
    }
end
...
{
    ...
} catch: ExceptionConverter<ExceptionNegNum, ExceptionOutOfLimits>;

```


ExceptionConverter can be defined for 2, 4, 6, etc. parameters.

Another common pattern of exception treatment is to encapsulate exceptions in an exception container.

```
object ExceptionEncapsulator<Item, Container>
  fun eval: (Item e) {
    throw: Container(e)
  }
end

...
{
  // ExceptionNegNum may be thrown here
  ...
} catch: ExceptionEncapsulator<ExceptionNegNum, ExceptionArithmetic>;
```

Whenever ExceptionNegNum is thrown in the function, it is packed into an exception of ExceptionArithmetic and thrown again.

Several exceptions that have the same treatment can be treated equally using the generic context object CatchMany. This prototype can take up to ten generic parameters. Here we show the version of it with two parameters.

```
object CatchMany<T1, T2>(UFunction<Nil> b)
  fun eval: (T1 e) {
    b eval
  }
  fun eval: (T2 e) {
    b eval
  }
end

...

{
  line = In readLine;
  if line size == 0 {
    throw: ExceptionEmptyLine
  } else if line size > MaxLine {
    throw: ExceptionLineTooBig(line)
  } else if line[0] == ' ' {
    throw: ExceptionWhiteSpace
  };
  Out println "line = " + line
} catch: CatchMany<ExceptionEmptyLine, ExceptionLineTooBig>(
  { Out println: ("Limit error in line " + line) } )
catch: CatchMany<ExceptionWhiteSpace, ExceptionRead>(
  { Out println: "Other error happened" });
```

We used % in the declaration of the parameter of `b` of `CatchMany` in order to allow expressions to be parameters to this context object.² It was used `UFunction` instead of `Function` as the type of the parameter `b` because `b` is declared by the compiler as an instance variable of this prototype (see Section 11.3). Although this syntax is not too complex, it is not as clean as the equivalent feature of the new version of Java:

```
try {
    ...
} catch ( ExceptionEmptyLine | ExceptionLineTooBig e) { ... }
    catch ( ExceptionWhiteSpace | ExceptionRead e) { ... }
```

A catch object can declare a grammar method with alternative parameters:

```
object CatchLineExceptions
  fun (eval: ExceptionEmptyLine | ExceptionLineTooBig) t {
    Out println: ("Limit error in line " + %line)
  }
end
...
```

```
{
  line = In readLine;
  if line size == 0 {
    throw: ExceptionEmptyLine
  } else if line size > MaxLine {
    throw: ExceptionLineTooBig(line)
  } else if line[0] == ' ' {
    throw: ExceptionWhiteSpace
  };
  Out println "line = " + line
} catch: CatchLineExceptions
  catch: CatchMany<ExceptionWhiteSpace, ExceptionRead>(
    { Out println: "Other error happened" });
```

The effect is the same as the previous code.

Using union types, we can catch several exceptions with a single function:

```
{
    ...
} catch: { (: ExceptionEmptyLine | ExceptionLineTooBig e :)
  Out println: "Limit error in line " + line
}
  catch: { (: ExceptionWhiteSpace | ExceptionRead e :)
  Out println: "Other error happened"
};
```

12.6 More Examples of Exception Handling

One can design a `MyFile` prototype in which the error treatment would be passed as parameter:

²The default qualifier is %. Then we could have omitted this symbol in the declaration of parameter `b`.

```

object MyFile
  @prototypeCallOnly
  fun new: (String filename) { ... }
  fun catch: (ExceptionCatchFile catchObject) do: (Function<String, Nil> b) {
    {
      open;
      // readAsString read the whole file and put it in a String,
      // which is returned
      b eval: readAsString;
      close;
    } catch: catchObject
  }
end

```

Context object `Throw` extends `UFunction<Nil>` and has an `eval` method that throws the exception that is the parameter of the context object.

```

object Throw(CyException e) extends UFunction<Nil>
  fun eval {
    throw: e
  }
end

```

It makes it easy to throw some exceptions:

```

{
  line = In readLine;

  if line size == 0 { Throw(ExceptionEmptyLine) }
  else if line size > MaxLine { Throw(ExceptionLineTooBig(line)) }
  else if line[0] == ' ' { Throw(ExceptionWhiteSpace) };

  Out println "line = " + line
} catch: CatchIgnore<ExceptionLineTooBig, ExceptionEmptyLine>
catch: { (: ExceptionWhiteSpace e :)
  Out println: "line cannot start with white space";
  System exit
};

```

Prototype `CatchWithMessage` catches all exceptions. It prints a message specific to the exception thrown and prints the stack of called methods:

```

object CatchWithMessage
  fun eval: (CyException e) {
    Out println: "Exception #{e prototypeName} was thrown";
    System printMethodStack;
    System exit
  }
end

```

An exception prototype may define an `eval:` method in such a way that it may be used as a catch parameter:

```

object ExceptionZero extends CyException
  fun eval: (ExceptionZero e) {
    Out println: "Zero exception was thrown";
    System exit
  }
end
...

// inside some method
{
  n = In readInt;
  if n == 0 { throw: ExceptionZero };
  ...
} catch: ExceptionZero;

```

This is confusing. But somehow it makes sense: the exception, which represents an error, provides its own treatment (which is just a message). Guimarães [dOGaa] suggests that a library that may throw exceptions should also supply catch objects to handle these exceptions. It could even supply an hierarchy of exceptions for each set of related exceptions. For example, if the library has a prototype for file handling, it should also has a catch prototype with a default behavior for the exceptions that may be thrown. And sub-prototypes with alternative treatments and messages.

Since exceptions and theirs treatment are objects, they can be put in a hash table used for choosing the right treatment when an exception is thrown.

```

object CatchTable
  fun init {
    // assume {* and *} delimit a literal hash table
    table = {*
      ExceptionZero : CatchExit<ExceptionZero>,
      ExceptionNeg : CatchAll,
      ExceptionRadius: { (: ExceptionRadius e :)
        Out println: "Radius #{e radius} is not valid"
      },
      ExceptionTriangle: CatchTriangle
    *};
  }
  fun eval: (CyException e) {
    var catch = table[e prototype];
    catch
      typeCase: Any do: {
        catch ?eval: e
      }
      typeCase: Nil do: {
        throw: ExceptionTable
      }
  }
}
Hashtable<CyException, Any> table

```

```
end
```

CatchTable can be used as the catch object:

```
// inside some method
{
    ...
} catch: CatchTable;
```

If an exception is thrown in the code "...", method `eval:` of `CatchTable` is called (its parameter has type `CyException`, the most generic one). In this method, the hash table referenced by variable `"table"` is accessed using as key `"e prototype"`, the prototype of the exception. As an example, if the exception is an object of `ExceptionTriangle`, `"e prototype"` will return `ExceptionTriangle`. By indexing `table` with this value we get `CatchTriangle`. That is,

```
assert: table[e prototype] == CatchTriangle
```

in this case. Here `table[elem]` returns the value associated to `elem` in the table.

Message `?eval: e` is then sent to object `CatchTriangle`. That is, method `eval:` of `CatchTriangle` is called. The result is the same as if `CatchTriangle` were put in a `catch:` selector as in the example that follows.

```
object ExceptionTriangle(public Double a, public Double b, public Double c)
end
```

```
object CatchTriangle
  fun eval: (ExceptionTriangle e) {
    // "e a" is the sending of message "a" to object "e"
    // that returns the side "a" of the triangle
    Out println: "There cannot exist a triangle with sides #{e a}, #{e b}, and #{e c}"
  }
end
```

```
// inside some method
{
    ...
  if a >= b + c || b >= a + c || c >= a + c {
    throw: ExceptionTriangle(a, b, c)
  };
  ...
} catch: CatchTriangle;
```

Then we can replace `catch: CatchTriangle` in this code by `"catch: CatchTable"`. However, if an exception that is not in the table is thrown, exception `ExceptionTable` is thrown. Assume that `Nil` is returned by indexing the hash table when the key is not found. That is, `"table[e prototype]"` returns `Nil` if the prototype is not found in the table.

Exception `ExceptionStr` is used as a generic exception which holds a string message.

```
object ExceptionStr(public String message) extends CyException
  fun eval: (ExceptionStr e) {
    Out println: (e message);
    System exit
  }
end
```

```
    }  
end
```

It can be used as

```
{  
  var s = In.readLine;  
  if s.size < 2 {  
    throw: ExceptionStr("size should be >= 2")  
  } else if s.size >= 10 {  
    throw: ExceptionStr("size should be < 10")  
  };  
} catch: ExceptionStr;
```

Chapter 13

User-Defined Literal Objects

Objects can be created in Cyan using methods `clone` and `new`. A literal object is an object created implicitly without using these two methods. For example, `1`, `'a'`, `3.1415`, `"this is a string"`, and `{ ^x <= 1 }` are literal objects. The first three are objects from the basic types. The function `{ ^x <= 1 }` is a literal object of a compiler-created object as explained in Chapter 10. And every user-defined prototype such as `Person` is a literal object.

Cyan will support literal objects through pre-defined metaobjects. We will give a glimpse of the syntax of these metaobjects, which will not be adequately specified and may be subject to change. So consider that this Chapter does not really define a language feature. It just gives an overview of a feature that will be supported by the language in the future.

13.1 Literal Numbers

Metaobject `literalNumber` is used for defining literal objects that start with a number. In Cyan, it is possible to define literals such as

```
100meters 50yards 50kg 30lb
3000reaais 500dollars 2000_euros
10this_is_unknown 0_real_1img
```

These will be called *literal numbers*. When the compiler finds a token that starts with a number but ends with a sequence of letters, numbers, and underscore, it searches for a metaobject capable of treating that token. A metaobject that treats a literal number is declared using another metaobject, `literalNumber`:

```
@literalNumber<**
  endsWith: "bin", "Bin", "BIN"
  type: Int
  parser: BinaryNumberParser
**>
```

The body of this metaobject should contain a sequence of pairs tag-value. One of the tags is `endsWith` that specifies a sequence of letters, digits, and underscore (starting with a letter or underscore) that ends this literal number (before the sequence there should appear at least a digit). In this example, there are several alternatives: `bin`, `Bin`, or `BIN`. The `parser` tag specifies the Java class responsible for treating literal numbers ended by strings given in the `endsWith` tag. Then a number

```
101Bin
```

will be processed by the Java class `BinaryNumberParser`. Future versions of Cyan will use Cyan prototypes instead of Java classes. What happens is that, when the compiler finds a number ending by `bin`,

Bin, or BIN, it loads the Java class `BinaryNumberParser`, creates an object from it, and calls a method of this object passing string

```
"    101Bin"
```

and an object of class `PCI` as parameters (this will soon be explained). The method to be called depends on what interface `BinaryNumberParse` implements. If it implements `RetString`, method `parseRetString` is called. If it implements interface `RetASTExpr`, method `parseRetASTExpr` is called. It is an error to make this class implement both or none of these interfaces. Suppose that, in this case, `BinaryNumberParser` implements interface `RetString` and the compiler calls method `parseRetString` of the newly created `BinaryNumberParser` object.

Method `parseRetString` will return a string that will be passed to the Cyan compiler. This string will replace `101Bin`. It is expected that the string returned from `parseRetString` of `BinaryNumberParser` will be 5 in base 10. The Java class could be declared as

```
public class BinaryNumberParser implements RetString {
    public String parseRetString(String text, PCI compiler) {
        // cut the last three characters, which is
        // bin, Bin, or BIN
        String number = text.substring(0, text.length() - 3);
        int n = 0;
        for(i = 0; i < number.length(); i++)
            n = 2*n + number.charAt(i);
        return n + "";
    }
}
...
interface RetString {
    public String parseRetString(String);
}
```

Class `PCI` is the *Public Compiler Interface*, which is a restricted view of the compiler class that compiles the Cyan program. Through the `PCI` object passed as parameter some compiler methods can be called and some compilation information can be obtained. In this example, no information is needed and no compiler method is called.

Tag `type` of the `literalNumber` call gives the type of the literal object. This tag may be omitted. Class `BinaryNumberParser` could have defined a method

```
Expr parseRetASTExpr(String text, PCI compiler)
```

that returns an object of the compiler AST representing an integer.

```
public class BinaryNumberParser {
    public Expr parseRetASTExpr(String text, PCI compiler) {
        // cut the last three characters, which is
        // bin, Bin, or BIN
        String number = text.substring(0, text.length() - 3);
        int n = 0;
        for(i = 0; i < number.length(); i++)
            n = 2*n + number.charAt(i);
        return new ExprLiteralInt(n, compiler.currentToken());
    }
}
```


`ExprLiteralInt` is the AST class of the compiler that represents a literal integer. Both `n` and the current token of the compiler is passed as arguments in the creation of the `ExprLiteralInt` object, which represents a literal number `n`. The current token of the compilation is an object of class `Token` representing `101Bin`. This object is necessary when there is a compilation error for it contains the number of the line of the token.

A literal number should be defined outside any prototype. It may be declared private, protected, or public by tags `private:`, `protected`, and `public:` without arguments. A private literal can only be used in the source file in which it was declared. A protected literal can only be used in its package. A public one can be used in any package that imports the package in which it was defined.

In the future Cyan will replace Java as the metaobject protocol language. Then the Java AST expression will not be returned by the method. Line

```
return new ExprLiteralInt(n, compiler.currentToken());
```

would be replaced by something like

```
compiler exprLiteralInt: n;
```

This message send calls a grammar method that creates a literal `Int` which replaces the original expression (`101Bin` in the example). Using other grammar methods of parameter `compiler`, one will be able to create any Cyan expression.

Literal numbers may specify anything, not just numbers. For example, one could call `literalNumber` to create a graph with the syntax

```
var Graph g = 1_2__2_3__3_1_graph;
```

That would be the graph $G = \{(1, 2), (2, 3), (3, 1)\}$.

13.2 Literal Objects between Delimiters

A metaobject call is delimited by a pair like `<<+` and `+>>`. The rules of formation of these pairs were explained in Section 5.2. Metaobject `literalObject` allows one to define literal objects which are delimited by pairs like `<<+` and `+>>` optionally preceded by an identifier. The language offers several examples of these literal objects:

```
// arrays
var Array<Int> v = {# 1, 2, 3 #};
// unnamed tuples
var Tuple<String, Int> t1 = [. "first", 0 .];
// named tuples
var t2 = [. university: "UFSCar" country: "Brazil" .];
```

Besides these ones, the programmer may define her own literal objects, which may be preceded by an identifier. So we could have things like

```
var Graph g1 = GraphBuilder<<+ (1, 2), (2, 3), (3, 1) +>>;
var Graph g2 = GraphBuilder(** (1, 2), (2, 3), (3, 1) **);
var Hashtable dictionaryEnglishPortuguese = Dict( "one":"um", "two":"dois" );
```

`GraphBuilder` is the name of the literal object. Its declaration, explained in the following paragraphs, defines the syntax of the code between `<<+` and `+>>` and how it is used to create a `Graph` object. The delimiters are not attached to a particular literal object and the literal object is not attached to a particular pair of delimiters.

Literal object names have their own name space. So we can have a literal object name `Graph` and a literal object name `Graph` (instead of `GraphBuilder` as in the example).

A literal object may be specified in several different ways. The simplest one is by giving a start delimiter and a Java class for parsing the object:

```
@literalObject<<
  start: "<@"
  parse: ParseList
>>
```

When the compiler finds “<@”, it loads the Java class `ParseList`, creates an object of this class, and calls a `parseRetString` method of this object passing to it the text between <@ and @> and a `PCI` object (*Public Compiler Interface*). The `parseRetString` method should return a string that creates an object that replaces

```
<@ ... @>
```

For example, suppose we have

```
var myList = <@ "hi", 23, 3.14 @>;
```

and method `parseRetString` of `ParseList` returns

```
var tmp0001 = List new: 3;
tmp0001 add: "hi";
tmp0001 add: 23;
tmp0001 add: 3.14;
```

Then `var myList = <@ "hi", 23, 3.14 @>;` will be transformed into

```
var tmp0001 = List new: 3;
tmp0001 add: "hi";
tmp0001 add: 23;
tmp0001 add: 3.14;
var myList = tmp0001;
```

The type of the object resulting from the literal object may be given by tag `type`:

```
@literalObject<<
  start: "<*"
  parse: ParseIntSet
  type: Set<Int>
>>
```

The name of the literal object is given by tag `name`:. In this option, the `start`: tag should not be given.

```
@literalObject<<
  name: GraphBuilder
  parse: GraphParser
  type: Graph
>>
```

```
...
var Graph g1 = GraphBuilder<<<+ (1, 2), (2, 3), (3, 1) +>>;
var Graph g2 = GraphBuilder(** (1, 2), (2, 3), (3, 1) **);
```

Both named and unnamed literal objects may have a `regexpr`: tag. After this tag should appear a regular expression. The literal object should obey the grammar defined by this literal object which is composed by the regular expression operators, type names, and strings of symbols given between quotes. Each type means an expression of that type, just like in a method signature. Let us see an example.

```

@literalObject<<
  start: "<@"
  regexpr: ( String ":" Int )*
  type: ListStringInt
  parse: ParseListStringInt
>>
...
var myList = <@ "um": 1 "dois" : 2 @>;

```

The regular expression given after `regexpr:` matches pairs of strings and integers separated by “:”. The literal object uses this regular expression to check the literal object before calling method `parseRetString` (or other equivalent to it) of `ParseListStringInt`. Therefore the checking is made twice by the metaobject `literalObject` and by the Java class.

Tag `addAll` allows one to easily define a literal object. The `parse:` tag should be omitted if we use a `addAll:` tag. Suppose we want a dictionary literal which should be an object of prototype `Dict<String, String>`. Assume this prototype defines a method

```

  addEverything: Array<Tuple<String, String>>

```

Then the literal could be defined as

```

@literalObject<<*>
  start: "{*"
  regexpr: (String ":" String)*
  type: Dict
  addAll: (Dict<String, String> getMethod: "addEverything: Array<Tuple<String, String
    >>")
*>>

```

And used as

```

var dict = {* "John" : "professor" "Peter":"engineer" "Anna":"artist" *};

```

Metaobject `literalObject` would create an object based on the real literal object given between `{*` and `*`. The algorithms used in grammar methods would be used here. Then for the above example, the metaobject would create a object of

```

  Array<Tuple<String, String>>

```

Then an object of `Dict<String, String>` would be created and the above object would be passed to method `addEverything` of this object. This method is specified in the tag `addAll`. Of course, the method name could be anyone such as “`createFrom`”.

Then the code

```

  var dict = {* "John" : "professor" "Peter":"mechanic" "Anna":"Artist" *};

```

would be converted into

```

var obj001 = Array<Tuple<String, String>> new: 3;
obj001[0] = [. "John", "professor" .];
obj001[1] = [. "Peter", "engineerer" .];
obj001[2] = [. "Anna", "artist" .];
var tmp0001 = Dict<String, String> new;
tmp0001 addEverything: obj001;
var dict = tmp0001;

```

In the regular language

```
( String ":" Int )*
```

the most external operator is an *. Therefore we can choose to add one element at a time by using tag add: instead of addAll:.

```
@literalObject<<*>
  start: "{*"
  regexpr: (String ":" String)*
  type: Dict
  add: (Dict<String, String> getMethod: "add: Tuple<String, String>")
*>>
```

In this case,

```
var dict = {* "John" : "professor" "Peter":"mechanic" "Anna":"Artist" *};
```

would be converted into

```
var tmp0002 = Dict<String, String> new;
tmp0002 add: [ "John", "professor" .];
tmp0002 add: [ "Peter", "engineer" .];
tmp0002 add: [ "Anna", "artist" .];
var dict = tmp0002;
```

Annotations could be used to allow the literal object to be build using an AST as in grammar methods. So we could write

```
@literalObject<<*>
  start: "{*"
  regexpr: (String ":" String)*
  type: Dict
  addAll: Dict<String, String>
*>>
```

if prototype Dict<String, String> has an annotation #f1 in method addEverything. The metaobject literalObject would then know that this method should be used in order to create the literal object. The root prototype could have references to other objects that should be annotated too, just like grammar methods. This feature makes small Domain Specific Languages very easy to implement.

By default, comments are allowed in the literal objects. This can be changed by using option comments off:

```
@literalObject<<*>
  start: "{*"
  regexpr: (String ":" String)*
  type: Dict
  addAll: Dict<String, String>
  comment: off
*>>
```

A future improvement in literal objects would be to allow generic types. Like

```
@literalObject<<*>
  start: "{*"
  genericType: T, U
  regexpr: (T ":" U)*
```

```

type: Dict<T, U>
addAll: (Dict<T, U> getMethod: "addEverything: Array<Tuple<T, U>>")
*>>

```

There is one special delimiter for literal objects: “\”. A literal object whose left delimiter is “\” ends with “\” too. Then a regular expression literal object can be easily defined:

```

@literalObject<<
  start: "\"
  parse: ParseRegularExpression
>>

```

Class `ParseRegularExpression` is very simple:

```

public class ParseRegularExpression implements RetString {
  public String parseRetString(String text, PCI compiler) {
    // cut the first and last characters, which are \
    String regExpr = text.substring(1, text.length() - 1);
    return "(RegExp new: \"" + regExpr + "\")";
  }
}

```

Assume the existence of a `RegExp` prototype for regular expressions. `RegExp` has a method

```

fun =~ (String str) -> Boolean

```

that returns `true` if the regular expression that receives the message matches `str`. When the compiler finds a literal regular expression object

```

\reg-expr\

```

it replaces it by

```

(RegExp new: "reg-expr")

```

The compiler only recognizes this literal object if the call to metaobject `literalObject` is in a package imported by the current source file — see Figure 13.1.

As an example of code,

```

if \[A-Za-z]+\ =~ ident {
  Out println: "found an identifier";
}

```

would be replaced by

```

if (RegExp new: "[A-Za-z]+") =~ ident {
  Out println: "found an identifier";
}

```

Literal objects could be used to embed small languages inside the Cyan source code. That is, literal objects can be used for implementing Domain Specific Languages. For example, a literal object named `AwkCode` could store code of language AWK:

```

Awk with: "fileName" do: AwkCode[#
  ([A-Z]*\ ~$0) : { v[n++] = $0 }
  END : { for (j = 0; j < n; j++) println v[j]; }
#]

```



Figure 13.1:

The same can be made for a small version of Prolog:

```

Prolog query: {? fat(5, X) ?} database: {+
fat(0, 1).
fat(N, F) :- N1 is N - 1, fat(N1, H), F is H*N.
+}

```

In the same vein, SQL code can easily be embedded in Cyan:

```

var v = [## select name, age from Person where age > 18 ##];

```

Cyan does allow nested comments. If it did not, comments delimited by `/*` and `*/` could be easily be implemented as literal object.

```

@literalObject<<*
start: "/*"
parse: ParseComment
*>>

```

The parser should only return a single space character — in Cyan, any comment counts as a single space.

```

public class ParseComment implements RetString {
public String parseRetString(String text, PCI compiler) {
return " ";
}
}

```

Generic prototypes may be defined as metaobjects. Then “`P<T1, T2, ... Tn>`” results in a compile-time call to a method `createRealPrototype` of metaobject `P` which would produce a real prototype that replaces “`P<T1, T2, ... Tn>`” in the source code. Tuples will be implemented in this way by the Cyan

compiler. In this approach, generic prototypes are just functions that return a prototype or interface at compile-time.

For example, suppose we want to define metaobject `NTuple` for generic tuples. It would be a Java class with a method `createRealPrototype` that accepts the real arguments to the tuple type. This method would return the source code of the `NTuple` prototype (probably as an array of `Char`'s). For each set of real arguments there would be a different source code. The Java method `createRealPrototype` should generate the methods of a prototype `TupleT1T2...Tn` with those arguments. There should be a loop somewhere in this method that iterates on the real arguments because for each field there should be generated two methods (one for getting and the other for setting the field).

It would be nice to have a metaobject `genericPrototype` that helps the creation of metaobjects that represent generic prototypes. Metaobject `genericPrototype` could support a macro language¹ that make it easy to generate code for the generic prototype. To define a prototype `NTuple` we could write

```
@genericPrototype<<*
  name: NTuple
  code: {**
    // code of NTuple with macro commands
    object NTuple...
    ...
    end
  **}
*>>
```

Tag “name” gives the name of the “generic prototype”, which is in fact a metaobject. Tag “code” accepts a code between “{**” and “**}” as shown above. This code would be the code of prototype `NTuple`. It would consist of Cyan code and commands of a macro language defined by the creator of `genericPrototype`. This language would have decision and repetition statements that would make it easy to generate code.

¹Following a suggestion of Rodrigo Moraes.

Chapter 14

The Cyan Language Grammar

This Chapter describes the language grammar. The reserved words and symbols of the language are shown between “ and ”. Anything between

- { and } can be repeated zero or more times;
- { and }+ can be repeated one or more times;
- [and] is optional.

The program must be analyzed by unfolding the rule “Program”.

There are two kinds of comments:

- anything between /* and */. Nested comments are allowed.
- anything after // till the end of the line.

Of course, comments are not shown in the grammar.

The rule CharConst is any character between a single quote '. Escape characters are allowed. The rule Str is a string of zero or more characters surrounded by double quotes ". The double quote itself can be put in a string preceded by the backslash character \. Rule AtStr is @" followed by a string ended by double quotes. The backslash character cannot be used to introduce escape characters in this kind of string.

A literal number starts with a number which can be followed by numbers and underscore (_). There may be a trailing letter defining its type:

```
35b // Byte number
2i // integer number
```

There should be no space between the last digit and the letter. User-defined literal numbers start with a digit and may contain digits, letters, and underscore:

```
100Reais 2_3_5_7_prime_0_2_4_even
```

All words that appear between quotes in the grammar are reserved Cyan keywords. Besides these words, there are other keywords cited in Section 3.3 that are not currently used by the language.

Id is an identifier composed by a sequence of letters, digits, and underscore, beginning with a letter or underscore. But a single underscore is not a valid identifier. IdColon is an Id followed by a “:”, without space between them, such as “ifTrue:” and “ifFalse:”. InterIdColon is an Id followed by a “:” and preceded by “?” as in “?at:” (dynamic unchecked message send). InterId is an Id preceded by “?” such as “?name”. TEXT is a terminal composed by any number of characters. Symbol ‘ is

terminal BACKQUOTE, ASCII 96. InterDotIdColon is an Id followed by a “:” and preceded by “?” as in “?.at:”. (nil-safe message send). InterDotId is an Id preceded by “?” as in “?.name”.

LeftCharString is any sequence of the symbols

= ! \$ % & * - + ^ ~ ? / : . \ | ([{ <

Note that >,),], and } are missing from this list. RightCharString is any sequence of the same symbols of LeftCharString but with >,),], and } replacing <, (, [, and {, respectively. The compiler will check if the closing RightCharString of a LeftCharString is the inverse of it. That is, if LeftCharString is

(*=<[

then its corresponding RightCharString should be

[>=*)

SymbolLiteral is a literal symbol (see page 67 for definition). There are limitations in the sequences of symbols that are considered valid for literal objects. They cannot start with ((,)), ([,]), [[,]], (:, {(:, >{, {^, :[, :(, {., and ::. For short, they cannot start with any sequence of symbols which can appear in a valid Cyan program. For example, [(: is illegal because we can have a function declared as

{ (: Int n :) ^ n*n }

SymUnary is a Cyan symbol #ident in which ident is a valid identifier. For example,

#get #b100 #array #size

SymColor is a Cyan symbol #ident: in which ident is a valid identifier. For example,

#set: #at: #do: #f34_34:

CompilationUnit	::= PackageDec ImportDec { CTMOCallList ProgramUnit }
PackageDec	:: “package” QualifId [“,”]
ImportDec	::= { “import” IdList [“,”] }
ProgramUnit	::= [QualifProtec] (ObjectDec InterfaceDec)
QualifProtec	::= “private” “public” “protected”
CTMOCallList	::= { CTMOCall }
CTMOCall	::= (“@” “@@”) Id [“(” “[” “{” ExprLiteral “)” “]” “}”] [LeftCharString TEXT RightCharString]
ObjectDec	::= [“mixin” [“(” Type “)”] “abstract” “final”] “object” Id { TemplateDec } [ContextDec] [“extends” Type] [“mixin” TypeList] [“implements” TypeList] { SlotDec } “end”
TemplateDec	::= “<” TemplateVarDecList “>”
TemplateVarDecList	::= TemplateVarDec { “,” TemplateVarDec }
TemplateVarDec	::= [Type] Id [“+”]
ContextDec	::= “(” CtxtObjParamDec { “,” CtxtObjParamDec } “)”
CtxtObjParamDec	::= [“public” “protected” “private”] Type [“%” “&” “*”] Id

```

Type ::= SingleType { “|” SingleType }
SingleType ::= QualifId { “<” TypeList “>” } | BasicType |
    “typeof” “(” QualifId [ “<” TypeList “>” ] “)”
SlotDec ::= CTMOCallList QualifProtec ( ObjectVariableDec
    | MethodDec | ConstDec )
ConstDec ::= “const” Type Id “=” Expr [ “;” ]
MethodDec ::= [ “override” ] [ “abstract” ] “fun” MethodSigDec
    ( MethodBody | “=” Expr [ “;” ] )
MethodSigDec ::= [ Type ] ( MetSigNonGrammar | MetSigUnary |
    MetSigOperator | MetSigGrammar )
MetSigNonGrammar ::= { SelecWithParam }+
MetSigUnary ::= Id
MetSigOperator ::= UnaryOp | BinaryOp ( ParamDec | “(” ParamDec “)” )
MetSigGrammar ::= SelectorGrammar [ Type ] Id
SelecWithParam ::= IdColon |
    IdColon [ “[” ] ParamList
SelectorGrammar ::= “(” SelectorUnitSeq “)”
    [ “*” | “+” | “?” ]
SelectorUnitSeq ::= SelectorUnit { SelectorUnit } |
    SelectorUnit { “|” SelectorUnit }
SelectorUnit ::= SelecGrammarElem | SelectorGrammar
SelecGrammarElem ::= IdColon |
    IdColon TypeList |
    IdColon “(” Type “)” ( “*” | “+” )
TypeOrParamList ::= TypeList | ParamList
TypeList ::= Type { “,” Type }
ParamList ::= ParamDec { “,” ParamDec } |
    “(” ParamDec { “,” ParamDec } “)”
ParamDec ::= [ Type ] Id
MethodBody ::= “{” StatementList “}”
ObjectVariableDec ::= [ “shared” ] [ “var” ] Type Id [ “=” Expr ]
    { “,” Id [ “=” Expr ] } [ “;” ]
FunctionDec ::= “” [ “(” FuncSignature “:” ] StatementList “”
FuncSignatureRet ::= FuncSignature [ “->” Type ]
FuncSignature ::= ( Type Id | Type “self” ) { “,” Type Id } [ “->” Type ] |
    [ Type “self” “,” ] IdColon { Type Id } { “,” Type Id } }
QualifId ::= Id { “.” Id }
TypeList ::= Type { “,” Type }
IdList ::= Id { “,” Id }
InterfaceDec ::= “interface” Id [ TemplateDec ] [ “extends” TypeList ]
    { “fun” InterMethSig }
    “end”
InterMethSig ::= [ Type ] InterMethSig2
InterMethSig2 ::= Id |
    { IdColon [ “[” ] [ InterParamDecList ] [ “]” ] }+ |
    UnaryOp |
    BinaryOp ( SingleInterParamDec | “(” SingleInterParamDec “)” )
InterParamDecList ::= WithoutParentDecList | WithParentDecList

```

```

WithoutParentDecList ::= ParamTypeDecList { “,” ParamTypeDecList }
ParamTypeDecList    ::= Type [ Id ]
WithParentDecList   ::= “(” WithoutParentDecList “)”
SingleInterParamDec ::= Type Id
BasicType           ::= “Byte” | “Short” | “Int” | “Long” |
                    “Float” | “Double” | “Char” | “Boolean”

StatementList       ::= Statement { “,” Statement } | ε
Statement           ::= ExprAssign | ReturnStat | VariableDec | CTMOCall |
                    IfStat | WhileStat | NullStat
VariableDec         ::= [ “var” ] [ Type ] Id [ “=” Expr ] { “,” [ Type ] Id [ “=” Expr ] } [ “,” ]
ReturnStat          ::= “return” Expr | “~” Expr
IfStat              ::= “if” Expr StatListBracket
                    { “else” “if” Expr StatListBracket }
                    [ “else” StatListBracket ]
WhileStat           ::= “while” Expr StatListBracket
StatListBracket     ::= “{” StatementList “}”
NullStat            ::= “;”
ExprAssign          ::= Expr [ Assign ]
Assign              ::= { “,” OrExpr } “=” OrExpr
Expr                ::= OrExpr [ MessageSendNonUnary ] | MessageSendNonUnary
                    “super” MessageSendNonUnary
MessageSendNonUnary ::= { [ BACKQUOTE ] IdColon [ RealParameters ] }+ |
                    { InterIdColon [ RealParameters ] }+ |
                    { InterDotIdColon [ RealParameters ] }+
BinaryOp            ::= ShiftOp | BitOp | MultOp | AddOp | RelationOp
                    “||” | “~||” | “&&” | “..” | “..<”
RealParameters      ::= ExprOr { “,” ExprOr }
ExprOr              ::= ExprXor { “||” ExprXor }
ExprXor             ::= ExprAnd { “~||” ExprAnd }
ExprAnd             ::= ExprRel { “&&” ExprRel }
ExprRel             ::= ExprInter [ RelationOp ExprInter ]
ExprInter           ::= ExprAdd [ ( “..” | “..<” ) ExprAdd ]
ExprAdd             ::= ExprMult { AddOp ExprMult }
ExprMult            ::= ExprBit { MultOp ExprBit }
ExprBit             ::= ExprShift { BitOp ExprShift }
ExprShift           ::= ExprColonColon [ ShiftOp ExprColonColon ]
ExprColonColon      ::= ExprDotOp { “:” ExprDotOp }
ExprDotOp           ::= ExprUnaryUnMS { DotOp ExprUnaryUnMS }
DotOp               ::= “. *” | “. +”
ExprUnaryUnMS       ::= ExprUnary { UnaryId }
UnaryId              ::= [ BACKQUOTE ] Id | InterId | InterDotId
ExprUnary            ::= [ UnaryOp ] ExprPrimaryIndexed
ExprPrimaryIndexed  ::= ExprPrimary Indexing
Indexing            ::= “[” Expr “]” | “?” Expr “]?”
UnaryOp             ::= “+” | “-” | “++” | “--” | “!” | “~”
ExprPrimary         ::= “self” [ “.” Id ] |
                    “self” |
                    “super” UnaryId |

```

	QualifId { "<" TypeList ">" }+ [ObjectCreation]
	QualifId { "<" TypeList ">" }+
	"typeof" "(" QualifId ["<" TypeList ">"] ")"
	ExprLiteral "(" Expr ")"
ObjectCreation	::= "(" [Expr { "," Expr }] ")"
ExprLiteral	::= ByteLiteral ShortLiteral IntLiteral
	LongLiteral FloatLiteral DoubleLiteral CharLiteral
	BooleanLiteral Str AtStr SymbolLiteral "Nil"
	LiteralArray FunctionDec
	LeftCharString TEXT RightCharString LiteralTuple
BooleanLiteral	::= "true" "false"
LiteralArray	::= "{#" [Expr { "," { Expr } }] "#}"
LiteralTuple	::= "[." TupleBody UTupleBody "."]"
TupleBody	::= (IdColon Id ":") Expr { "," IdColon Expr }
UTupleBody	::= Expr { "," Expr }
ShiftOp	::= "<.<" ">.>" ">.>>"
BitOp	::= "&" " " "~ "
MultOp	::= "/" "*" "%"
AddOp	::= "+" "-"
RelationOp	::= "==" "<" ">" "<=" ">=" "!="

Chapter 15

Opportunities for Collaboration

There are many research projects that could be made with Cyan and on Cyan:

- (a) to implement metaobjects `@dynOnce` and `@dynAlways` and to design algorithms that help the transition of dynamically-typed Cyan to statically-typed Cyan. There are a great deal of work here, at least several master thesis. This work can involve the discovery of types statically (at least most of them), the use of a profiler to discover some types at runtime, the combination of static and dynamic type information, refactorings directed by the user (he/she chooses the type of each troublesome variable/parameter/return type, for example), help by the IDE, etc.

It would be very important to have a language in which the programmer could develop a program without worrying about types in variables/parameters/return values and then convert this program to statically-typed Cyan. I would say that this is one of the central points of the language;

- (b) to design the metalevel appropriately. The design of the metalevel is of fundamental importance to the language. Usually metalevel programming is too difficult and regular programmers do not use it (not considering Lisp macros). The challenge is to design a “simple” metalevel. Maybe grammar methods may be useful here: instead of allowing the user to modify the abstract syntax tree of the code, she or he should change the code through one or more grammar methods. For example, to add a method to a prototype, one could use the code

```
Obj addMethod:
  selector: #sum:
  param: #a type: Int
  param: #b type: Int
  returnType: Int
  ASTbody:
    returnStat: (Expr add: #a, #b);
```

This is a grammar method of prototype `Any`. This call would add method

```
Int sum: (Int a, Int b) { return a + b }
```

to object `Obj`;

- (c) implement some Design Patterns in Cyan with the help of: 1) compile-time metaobjects and b) literal objects. The use of literal objects makes it easy the codification of some design patterns. This is a good project and it seems one of the easy ones. The metaobject protocol could be improved to deal with the most used patterns. I said “improved”, not modified just to make the patterns more easily implemented;

- (d) implement some literal objects which are the code of some small languages such as AWK and SQL. It would be nice if Cyan code could be used inside the code of the language;
- (e) to use Cyan to implement a lot of small Domain Specific Languages;
- (f) to use Cyan to investigate language-oriented programming [War95];
- (g) to implement a lot of small compile-time metaobjects for small tasks. One of my students [?] has already made some codegs. There are millions of interesting compile-time metaobjects to build;
- (h) to add parallelism to the language and to design a library for distributed programming. That includes the implementation of patterns for parallel programming;
- (i) to design code optimization algorithms for Cyan;
- (j) to program the Cyan basic libraries for handling files, data structures, and so on;
- (k) enumerated constants. They could be Java like or C# like — which would be better?

Chapter 16

The Cyan Compiler

Every Cyan program should be described by a `project` in `xml`. This project file should be in a directory with the same name as the file (without the extension `xml`). Every subdirectory of this directory corresponds to a package of the language. All the source files of a package should be in the same directory. For example, the following directory tree describes a program named `myFirstProg` which contains packages `main` and `DS`. Package `main` has source files `Program.cyan`, `A.cyan`, and `B.cyan`. `DS` has files `Stack.cyan` and `List.cyan`. We use right shift for sub-directories.

```
myFirstProg
  myFirstProg.xml
  main
    Program.cyan
    A.cyan
    B.cyan
  DS
    Stack.cyan
    List.cyan
```

The project file, `myFirstProg.xml` in this example, describes the Cyan program: `author` (tag `author` of `xml`, there may be more than one `author`), compiler options (attribute `options` of the root element, `project`), main package (tag `mainPackage`), main object (tag `mainObject`), path of other Cyan packages (tag `cyanpath`), path of Java files (tag `javapath`), and its packages (tag `packageList`). Each package is described by an element whose tag is `package`. The children of `package` can be `name` and `sourcefile` (one for each Cyan source file of the package). There may be an attribute `options` in each source file and in each package. The compiler options of a source file are the union of those defined in the project, package, and source file. The last ones have precedence over the options for project and package. And the package options have precedence over the ones of the project. For example, if the project has an option “-ue” and the source an option “+ue”, it is the last one that is valid in the tag `sourcefile`, if there is one. If there is none, the options are those of its package.

Let us see an example, file `myFirstProg.xml`:

```
<?xml version="1.0"?>
<!-- This is a comment -->
  <!-- options are the compiler options separated by spaces -->

<project options = "-ue -sfe" >
  <author>
```

```

Jose de Oliveira Guimaraes
</author>
<mainPackage>
main
</mainPackage>
<!-- This is the object in which the execution starts. It should be
      in the main package described above. The execution starts
      in method run of this object. -->
<mainObject>
Program
</mainObject>
<cyanpath>
  c:\cyan\lang
</cyanpath>

```

```

<packageList>
<package options = "-of" >
  <name> main</name>
  <sourcefile> Program.cyan </sourcefile>
  <sourcefile> A.cyan </sourcefile>
  <sourcefile> B.cyan </sourcefile>
</package>
<package>
  <name> DS </name>
  <sourcefile> Stack.cyan </sourcefile>
  <sourcefile> List.cyan </sourcefile>
</package>
</packageList>

```

```
</project>
```

A package may be in a directory which is not subdirectory of the project directory (`myFirstProg` in the example). In this case, the `package` element in the XML file describing the project should have an attribute `dir` indicating the directory of the package. In the following example, package `ds` is in directory `c:\user\jose\ds`.

```

<?xml version="1.0"?>
<project>
  <author>
    Jose de Oliveira Guimaraes
  </author>
  <mainPackage> main
</mainPackage>
  <mainObject> Program </mainObject>
  <cyanpath> c:\cyan\lang </cyanpath>

<packageList>

```



```

<package dir = "c:\user\jose\ds" >
  <name> ds </name>
  <sourcefile> Stack.cyan </sourcefile>
  <sourcefile> List.cyan </sourcefile>
  <sourcefile> MyArray.cyan </sourcefile>
</package>
<package>
  <name> main </name>
  <sourcefile> Program.cyan </sourcefile>
</package>
</packageList>
</project>

```

The source files of a package a.b.c.d should be in a directory d and the package name defines a directory tree:

```

C:\myProjects
  a
    b
      c
        d
          // source files of package d
        e
          // source files of c
      // source files of package c

```

There should be directories a, b, c, and d. But it is not necessary that packages a, b, and c exist.

There is one XML element called `codegpath` composed by a complete path of a single directory that should contain codegs. This element should appear in the same level as `author`. More than one `cyanpath` element can appear:

```

<?xml version="1.0"?>
<project>
  <author> Jose de Oliveira Guimaraes </author>
  <mainPackage> main </mainPackage>
  <mainObject> Program </mainObject>
  <cyanpath> c:\cyan\lang </cyanpath>
  <codegpath> D:\compilers\Cyan\lib\codeg </codegpath>
  <codegpath> E:\eu\cyan\codeg </codegpath>
</project>
<packageList>
  <package>
    <name> main </name>
    <sourcefile> Program.cyan </sourcefile>
  </package>
</packageList>

```

```
</package>
</packageList>
```

```
</project>
```

When a codeg `MyCodeg` is found by the Eclipse environment (with the codeg plugin), it searches for a Java class called `MyCodegCodeg` that will handle that codeg. This search is made in the following directories, in this order:

- `CYANPATH\codeg`, in which `CYANPATH` is a environment variable;
- the directories of the element `codegPath`, in the order they appear.

Using the above example, the compiler would search for, in this order:

- `CYANPATH\codeg\MyCodegCodeg`
- `D:\compilers\Cyan\lib\codeg\MyCodegCodeg`
- `E:\eu\cyan\codeg\MyCodegCodeg`

The current compiler options are:

1. “-cyanLang Path” in which Path is the path of the package `cyan.lang`. This package contains all the basic types, arrays etc;
2. “-add” that adds automatically the qualifier “public” for objects and methods declared without a qualifier and “private” for instance variables declared without qualifier. The compiler changes a source code like

```
package bank
```

```
object Account
  fun init: Client client {
    self.client = client
  }
  fun print {
    Out println: (client getName)
  }
  Client client
end
```

into

```
package bank
```

```
public object Account
  fun init: Client client {
    self.client = client
  }
  fun print {
    Out println: (client getName)
  }
  Client client
end
```

Chapter 17

Future Enhancements

Some Cyan features may be changed and others may be added. This is a partial list of them:

1. private generic prototypes, which are currently illegal;
2. `package` qualifier for prototypes and methods;
3. `typeof` may be legal as a real parameter in a generic prototype instantiation:

```
var Int count = 0;
var Stack<typeof(count)> intStack; // ok
```

4. A `finally:` selector may be added to the initial function that starts the program execution. That would allow finalizers, code that is called when the program ends. There could be a list of methods to be called when the program ends. This is odd, but someone will certainly like it.

```
{
} catch: RuntimeCatch
  finally: {
    DoomsdayWishList foreach: { (: UFunction<Nil> elem :) elem eval };
  };
```

In some other place:

```
DoomsdayWishList add: { "Good bye!" print };
```

Bibliography

- [Bla94] G. Blaschek. *Object-oriented programming with prototypes*. Monographs in Theoretical Computer Science - An Eatcs Series. Springer-Verlag, 1994.
- [Dea10] Fergal Dearle. *Groovy for Domain-Specific Languages*. Packt Publishing, 1st edition, 2010.
- [dOGaa] José de Oliveira Guimarães. The Green language exception system. *The Computer Journal*.
- [dOGab] José de Oliveira Guimarães. Reflection for statically typed languages. In *Proceedings of the 12th European Conference on Object-Oriented Programming*.
- [dOGa11] José de Oliveira Guimarães. O ambiente cyanhand e a biblioteca bicap, 2011. Scientific Initiation report. Available with the author.
- [flo12] Writing your first domain specific language, 2012. <http://www.codeproject.com/KB/recipes/YourFirstDSL>
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [LF12] The Logo Foundation. The logo language, 2012. <http://el.media.mit.edu/logo-foundation/index.html>.
- [Sei12] Peter Seibel. *Practical Common Lisp*. Apress, Berkely, CA, USA, 1st edition, 2012.
- [Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997.
- [US87] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, December 1987.
- [Vid11] Allan Vidal. Implementação de codegs de Cyan, 2011. <http://www.cyan-lang.org>.
- [War95] M. P. Ward. Language oriented programming. *Software — Concepts and Tools*, 15:147–161, 1995.

Index

- ++, 53
- , 53
- ?., 103
- ?[, 103
- [], 53

- abstract, 15, 91
- AddFieldDynamicallyMixin, 151
- addMethod, 205
- annot, 112, 147
- anonymous function, 182
- anonymous functions, 26, 180
- Any, 10, 85
- arithmetical operator, 51
- Array, 61
- array, 20
- assignment, 48
 - declaration, 66
 - multiple, 26, 49
 - tuple, 49
- attachMixin, 109

- block, 26, 180
- Boolean, 10, 50
- Byte, 10, 50

- C language, 216
- catch clause, 226
- catch object, 229
- catch objects, 229
- catch:, 36
- CatchIgnore, 234, 239
- Char, 10, 50
- clone, 6, 9, 69
- closure, 26, 185
- codeg, 119
- comment, 47
- Common Lisp, 235
- compiler, 262
- const, 68
- constant, 68

- constructor, 12
- context functions, 203
- context object, 28
- context objects, 213
 - copy parameters, 215
 - instance variable parameters, 215
 - reference parameters, 215
- Context-Free Languages, 166
- copy parameters, 215
- copyTo:, 148
- CyException, 36, 227
- CySymbol, 18, 67

- decision statement, 57
- default value, 66, 168
- design pattern, 260
- doc, 115
- Domain Specific Language, 170, 252
- Double, 10, 50
- DSL, 170, 252
- DTuple, 151
- dynAlways, 131
- dynamic typing, 22, 128
- dynOnce, 131

- Eclipse, 119
- EHS, 226, 227
- Elvis operator, 102
- eq:, 10
- exception, 35
 - checked, 236
 - unchecked, 236
- Exception Handling System, 226
- exception handling system, 35
- exit, 146

- feature, 113
- file, 41
- final, 16, 82
- Float, 10, 50
- formal parameters of generic prototype, 135

- fun, 67
- Function, 192
- function
 - context, 203
 - multiple selector, 198
 - r-function, 191
 - restricted, 28
 - restricted-use, 191
 - u-function, 191
 - unrestricted, 28
 - unrestricted-use, 191
- function return, 181
- functions, 26, 180

- generic prototype, 25
- generics, 135
- grammar method, 32, 155
- grammar, Cyan, 255
- graphical user interfaces, 202
- Green, 227
- GUI, 202

- handler, 226
- hideException, 233
- Hindley-Milner, 66

- identifier, 47
- if, 16, 58
- ifFalse:, 57
- ifNil, 103
- ifTrue:, 57
- import, 43
- In, 146
- indexing, 53
- inheritance, 9, 80
 - mixin, 103
- init, 12, 70
- initOnce, 70
- inner class, 214
- instance variable, 63
- instance variable parameters, 215
- instantiation, 136
- instantiation of a generic prototype, 136
- Int, 10, 50
- interface, 9, 92
- intervals, 152
- Introspective Reflection Library, 63
- IRL, 63

- javacode, 115
- keyword
 - abstract, 91
 - const, 68
 - fun, 67
 - super, 81
 - var, 65
- keyword message, 15

- language
 - Omega, 69
- Lisp, 235
- literal
 - String, 51, 67
 - tuple, 25, 147
- literal number, 246
- literal object, 38, 248
- literal objects, 246
- literalNumber, 246
- literalObject, 38, 249
- literals, 50
- local variable, 65
- Locyan, 121
- logical operator, 51
- Long, 10, 50
- loop, 59

- message
 - grammar, 32
 - keyword, 15
 - non-checked, 22
- message send, 15, 67, 202
- Meta-Object Protocol, 38
- metalevel, 260
- metaobject, 37, 111
 - annot, 112
 - call, 118
 - codeg, 119
 - doc, 115
 - dynAlways, 131
 - dynOnce, 131
 - javacode, 115
 - literalNumber, 246
 - literalObject, 249
 - onChangeWarn, 115
 - pre-defined, 112
 - strtext, 113
 - text, 113

- method, 15, 67
 - context, 213
 - grammar, 32, 155
 - hideException, 233
 - init, 70
 - initOnce, 70
 - keyword, 67
 - multi, 83
 - name, 79, 166
 - new, 70
 - object, 34
 - overloading, 18, 78
 - primitive, 194, 201
 - primitiveNew, 72
 - private, 64
 - protected, 64
 - public, 64
 - retry, 233
 - return, 75
 - return value, 67
 - selector, 76
 - signature, 42, 76
 - tryWhileFalse, 234
 - tryWhileTrue, 234
 - unary, 67
- mixin, 20, 103
- mixin inheritance, 103
- MOP, 38
- multi-methods, 83
- name of a method, 79
- nested class, 214
- new, 9, 70
- Nil, 12
- NTuple, 25, 147
- object, 63
 - abstract, 91
 - catch, 229
 - CatchIgnore, 234, 239
 - context, 28, 213
 - CyException, 227
 - final, 82
 - generic, 135
 - literal, 38, 246
 - metaobject, 111
 - method, 34
 - mixin, 20, 103
 - slot, 63
 - type, 95
 - Omega, 69
 - onChangeWarn, 115
 - Out, 146
 - overloaded methods, 18, 78
 - overloading, 18
 - override, 9, 81
 - package, 41
 - parallelism, 261
 - parameter, 76
 - default value, 168
 - PCI, 247
 - precedence, 56
 - primitive method, 194
 - primitive methods, 194, 201
 - primitiveNew, 72
 - private, 13
 - program unit, 41
 - protected, 13
 - public, 13
 - Public Compiler Interface, 247
 - r-function, 28
 - r-functions, 191
 - real parameter to generic prototype, 136
 - reference parameters, 215
 - reference type, 216
 - repeat:, 59
 - repeatUntil, 59
 - retry, 233
 - return, 75
 - return value, 67
 - runtime search, 19
 - scope, 77
 - selector, 6, 15, 76, 80
 - self, 69
 - shared variable, 69
 - Short, 10, 50
 - signature, 42, 76
 - String, 7, 10, 51
 - strtext, 113
 - subtype, 19, 95
 - super, 81
 - Symbol, 18
 - symbol, 18
 - System, 146

- text, 113
- throw, 36, 226
- to:do:, 59
- try block, 226
- tryWhileFalse, 234
- tryWhileTrue, 234
- Tuple, 25
- tuple, 49, 147
 - dynamic, 151
- type, 95

- u-function, 28, 191
- UTuple, 149

- var, 65
- variable, 9
 - default value, 66
 - local, 65
 - parameter, 76
 - shared, 69

- while, 16, 58
- whileFalse:, 58, 203
- whileTrue:, 58, 203

- XML, 42, 262