The Cyan Language

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

June 30, 2025

Contents

Foreword

This is the manual of Cyan, a prototype-based statically-typed object-oriented language. The language introduces several novelties that make it easy to implement domain specific languages, blend dynamic and static code, reuse exception treatment, and do several other common tasks. Although Cyan is an academic language, its design was made for programmers. Every aspect of it was designed to make programming fun.

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

(a) the metaobject protocol, see the Thesis "The Cyan Language Metaobject Protocol" in the Cyan site, `http://www.cyan-lang.org`;

(b) generic objects with variable number of parameters, Chapter 6;

(c) an object-oriented exception handling system, Chapter 11;

(d) context objects, Chapter 10;

(e) static typing with optional dynamic typing, Chapter 5;

(f) clearly designed overloaded methods which are a restricted form of multi-methods, Section 4.14;

The address of the Cyan home page is `http://www.cyan-lang.org`. Go to it for research articles and up-to-date changes in the language.

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 role 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, interfaces, generic prototypes, a completely object-oriented exception system, statically-typed anonymous functions, context objects (which are a generalization of anonymous functions), non-nullable types, optional dynamic typing, and a powerful metaobject protocol (MOP).

The Cyan MOP allows the control of the compilation process in many different ways making it relatively easy to produce and change code during the compilation. It is the biggest innotation of the language. The MOP is implemented through metaobject classes or prototypes. Only the main language package, cyan.lang, has around 80 of such classes (and growing). These metaobject classes are used, for example, to produce code for generic prototypes and to check arguments to methods. In particular, through a metaobject called `grammarMethod` one can easily define Domain Specific Languages inside the Cyan code. And *Concepts* [GJS+06] for generic programming can be defined without any language support.

Although the language is based in prototypes, it is closer in many aspects to class-based languages like C++ [Str13] and Java [GJS+14] 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. In particular, many constructs of other languages are implemented by methods in Cyan. For example, to throw an exception, to catch an exception, and to test if an object is of a certain prototype are all made through methods. We consider that Cyan, without the MOP, is not a big language. Since the MOP is not to be used extensively by regular programmers, we believe that learning the language will not be a problem.

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 Figures 1.1 and 1.2. The corresponding Java program is shown in Figure 1.3. It is assumed that there are classes `In` and `Out` in Java that do input and output (assume they are in package `inOut`). The Cyan program declares objects `Person` and `Program` (they should start with an upper-case letter). These objects are called *prototypes* to differentiate them from objects created during runtime. Object `Person` declares a variable `name` and methods `getName` and `setName`. Keywords `var` and `func` are used before a field (instance variable) and a method declaration. Method `getName` of `Person` takes no argument and returns a `String`. The return type appears after "`->`". Inside a method,

```
package program

object Person
    private var String name = ""
    public func getName -> String {
        return self.name
    }
    public func setName: String name {
        self.name = name
    }
end
```

Figure 1.1: A Cyan program: `Person`

```
package program

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

Figure 1.2: A Cyan program: `Program`

`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). The value returned by a method should appear after the keyword `return`. A package is a collection of prototypes (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.

Prototype `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 prototype `Person`. "`clone`" is called the "*selector*" of the message. All objects have a `clone` method. This statement declares a variable called `p` and assigns 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 next line,

```
var String name;
```

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

```
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 1.3: A Java program

```
name = In readLine;
```
there is the sending of message `readLine` to prototype `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 prototype `Out`. The message selector is "`println:`" and the argument is the object returned by "`p getName`".

The parameters[1] that follow a selector in a method declaration may be surrounded by `(` and `)`. So method `setName:` could have been declared as
```
public func setName: (String name) { self.name = name }
```
This is allowed to increase the legibility of the code.

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 prototype `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 subprototypes of `Person` (objects that inherit from `Person`, 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 prototype `Person` from `Worker` is made with the following syntax:

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

If a method is redefined in a subprototype (be it public or protected), keyword "`override`" should appear just after "`public`" or "`protected`". Methods of the subprototype may call methods of the super-prototype using keyword `super` as the message receiver:
```
super name: "anonymous"
```

In order to a prototype to be inherited, its declaration must be preceded by identifier "`open`" as in

```
open
object Person
    // elided
end
```

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

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 role 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
    func save
end

open
object Person
    func init: String name, Int age {
        self.name = name;
        self.age = age
    }
    func getName -> String = name;
    func setName: String name { self.name = name }
    func getAge -> Int = age;
    func setAge: Int age { self.age = age }
    var String name
    var Int age
end

object Worker extends Person implements Savable
    private String company
    func 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, field (instance variable or attribute), 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 created 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 (get object information, get metadata). Method

```
func 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.[2] Method `eq:` cannot be redefined in subprototypes. Method `neq:` retorns the opposite truth value of `eq:`

Basic Types

Cyan has the following basic types: `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Char`, `Boolean`, `Nil`, and `String` (no `Void` prototype). Since Cyan will be targetted to the Java Virtual Machine, the language has one type for each of the basic types of Java except for `void`. Unlike Java, all basic types in Cyan but `Nil` inherit from prototype `Any`. Therefore there are not two separate hierarchies for basic types, that obey value semantics,[3] and all other types, which obey reference semantics.

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"
}
```

Since the basic prototypes cannot be inherited, the compiler is free to implement basic types as if they obey value semantics. That is, a basic type `Int` could be translated to `int` of Java.[4] 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
    func 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`:

[2] For union types, `==` has a different behavior than `eq:`.

[3] The 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.

[4] The compiler being built translates Cyan to Java.

```
    // 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 could 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 `A|B` is considered, in assignments and parameter passing, as a supertype of both `A` and `B`.

```
var Int|String x;
x = 0; // ok
x = "Cyan"; // ok
```

The compiler automatically casts objects of `Int` and `String` to `x`. To retrive the object stored in `x` it is necessary to use the `type` command:

```
type x
    case Int n {
        Out println: "twice is " ++ 2*n
    }
    case String s {
        Out println: "first char is" ++ s[0]
    }
```

Inside first `case` clause, "`case Int n`", `n` has type `Int` and the value of `x`. The same applies to the second `case` and `s`.

There is a special object called `Nil` which is not subtype or supertype of anything. It somehow plays the role of `nil` of Smalltalk, `NULL` of C++, and `null` of Java/C#. 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 not `Nil` or an union.

```
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 at least a regular prototype.

```
var Nil|String s;
s = Nil; // ok
s = In readLine; // readLine returns a String
type s
    case Nil {
        // in case s is Nil
    }
    case String s2 {
        // s2 is a String here
    }
```

Then the runtime error "message send to `Nil`" does not happens in Cyan. Caveat: the compiler currently does not check if a variable is initialized or not before used. Then the Java compiler may signal a compile-time error "variable was not initialized" or there may be a run-time error "Null pointer exception". But this is a flaw of the compiler, not of the language.

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` or `init:` and may have any number of parameters. The return value should not be supplied (not even `Nil`). For each method named `init` or `init:` the compiler adds to the prototype a method named `new` or `new:` with the same parameter types. Each `new` or `new:` method creates an object and calls the corresponding `init` or `init:` method. If the prototype does not define any `init` or `init:` methods, the compiler supplies an empty `init` method that does not take parameters and calls the superprototype `init` method (if any. If this method does not exist, a compiler error occurs). Consequently, a `new` method is created too.

A subprototype should call one of the `init` methods of the superprototype (if one was defined by the user) using keyword `super`. This call should be the first statement of the method:

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

object Worker extends Person
    func 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.

Keyword Messages and Methods

Cyan was initially based on Smalltalk. As a result, it supports keyword messages, a message with multiple keywords as in

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

Like Smalltalk, Cyan calls `dist:` and `angle:` keywords. Following Smalltalk terminology, `dist:angle:` is called a "selector".

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

```
object Point
    func 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
    func 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 keywords in all possible ways. By "keyword" we mean method keywords, not Cyan reserved keywords.

We call the "name of a method" the concatenation of all of its keywords, each one followed by its number of parameters and a white space. The trailing white space should be removed. For example, methods

```
    func key: (String aKey) value: (Int aValue) -> String
    func name: (String first, String last)
         age: (Int aAge)
         salary: (aSalary Float) -> Worker
```

have names `"key:1 value:1"` and `"name:2 age:1 salary:1"`.

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 func 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 subprototype 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 whose declaration is not preceded by "`open`" cannot be inherited. It is a `final` prototype.

```
object Int
    ...
end
...
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.

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

Decision and Loop Methods and Statements

The `if` statement takes a boolean expression and is always followed by a sequence of statements between `{` and `}`. It is not necessary to put parentheses around the boolean expression. The `else` part is optional. The `while` statement also takes a boolean expression and a sequence of statements between `{` and `}`.

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

The `repeat-until` command executes its statements until the expression is `true`:

```
var Int sum = 0;
var Int n = 1;
repeat
    sum = sum + n;
    ++n;
until n >= 4;
assert sum == 6;
```

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"
};
```

Cyan Symbols

There is a special form of literal strings that start with `#` called simply "symbol". A symbol one starts by a `#` followed, without spaces, by letters, dot, digits, underscores, and ":", starting with a letter or digit. These are valid symbols in Cyan:

```
#name
#name:
#at:put:
#1
#711
```

The type of a symbol is `String`.

```
var String s;
s = #at:put:;
   // prints at:put:
Out println: s;
s = #7;
assert s == "7" && #at:put: == "at:put:";
```

Overloading

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

```
object Printer
    func print: Document d { ... }
    func print: String s, String form -> Boolean { ... }
    func print: Float s, Int beforeDot, Int afterDot { ... }
end
```

All of these methods are considered different. They can have different return value types, they have in fact different names. This is not true overloading.

True method overloading happens when the method keywords and the number of parameters are equal but the parameter types are different:

```
object MyBlackBoard

    overload // keyword that prefixes an overloaded method
    func draw: Square f { ... }
    func draw: Triangle f { ... }
    func draw: Circle f { ... }
    func 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 subprototype from `Square` (that is, whether the prototype extends `Square`, assuming `Square` is a prototype and not an interface). If it is not, the searches 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 superprototype. Since all `draw:` methods have the same number of parameters, it is necessary to prefix the first method with the keyword "`overload`". This will be explained later.

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 keywords `first:` and `second:` and;

(b) keyword `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 keyword `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 excluding fields). 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 keywords and;

(b) for each keyword, 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 `id -> Int` and `id -> String` in the same prototype (even if one of them is inherited).

In the redefinition of a method in a subprototype, one can change the return value type of the subprototype method. This type can be a subprototype of the type of the return value of the method of the superprototype:

```
open
object Animal
    func matchWhat -> Animal { ... }
end

object Cow extends Animal
       // ok, Cow is subprototype of Animal
    func matchWhat -> Cow { ... }
end
```

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 many object-oriented languages, the parameter types are inspected at runtime in order to discover which method should be called.

Arrays and Maps

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<Double>`. Since there is no automatic convertion of values, this results in a compile-time error: the compiler will not cast `2` to `Double`.

Literal maps are defined in the following way:

```
let IMap<Int, String> map = [ 0 -> "zero", 1 -> "one", 2 -> "two" ];
cast elem = map[0] {
    assert elem == "zero";
}
```

`map[0]` returns the element associated with `0`. Its type is a union of `String` and `Nil`. Statement `cast` should be used to access the result. In the example, `elem` has type `String` and the `assert` is only executed if `map[0]` is a `String`. It would be `Nil` if `map` did not associate a string to `0`.

Dynamic Typing

Although Cyan is statically-typed, it supports some features of dynamically-typed languages. A message send whose keywords 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 keywords. For example, in the code below, the compiler does not check whether prototype `Person` defines a method with keywords `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:

```
func 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`.

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 by a valid identifier. The identifier should be a parameter, local variable, or field of the current object. The result is that the identifier 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 Float johnsonSalary = 7000.0F;
Out println: "Person name = $name, n = $n, salary = $johnsonSalary";
```

This code prints

```
Person name = Johnson,  n = 3, salary = 7000.0
```

The last line is completely equivalent to

```
Out println: "Person name = " ++ name ++ ", n = " ++ n ++
    ", 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 used field names as parameters:

```
var Tuple<name, String> aName;
var Tuple<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;
```

A 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 Tuple<name, String, age, Int> q;
```

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 fields. 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
    func 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)`. 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 use a constant instead of a variable or make a copy of the variable in the function.

```
var y = 2;
var b = { (: Int x :)
        // make a copy of z
    var z = y;
    var sum = 0;
    while z > 0 {
        sum = sum + z;
        --z;
    };
    ^ x + sum;
};
(v eval: 5) println;

// make sure k is not changed
let k = 0;
var f = { (: Int n :)
    var sum = 0;
    var i = 0;
    while i < k {
        sum = sum + i;
        ++i;
    };
    ^ n + sum;
};
```

There are methods that can play the role of statements `if` and `while`.

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

Anonymous functions in Cyan cannot have return statements. Then the functions that are parameters to methods `ifTrue:ifFalse:` and `whileTrue:` cannot have return statements.

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:

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

...
   // sum the elements of array v
var v = [ 3.14, 2.71, 1.557 ];
var String s;
v foreach: Sum<String>(s);
```

Now context object `Sum` is used to sum the `Double` elements of vector `v`.

A context-object parameter not preceded by & 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 Function<Int, Nil>
    func 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;
```

Macro `assert` checks whether its argument returns `true`. It issues a warning if not. In this example, the final value of `s` will be `0`.

Parameters whose types are preceded by & are called *reference parameters* (see first example).

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 fields 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 role of class `A`) or to any other prototype that has fields of the types of the parameters of `B`. Besides that, both referenced parameters and field 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);
```

Grammar Methods

A method declared with selector `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 selector of the message send. Using operators of *regular expressions* a grammar method may specify that some keywords can be repeated, some are optional, there can be one or more parameters to a given keyword, there are alternative keywords and just one of them can be used.

Two methods that take a variable number of `Int` arguments are declared in prototype `IntSet`. Each method is preceded by a metaobject annotation `grammarMethod` (to be seen later). This annotation has a code of a Domain Specific Language between `{*` and `*}`.

```
package grammar

object IntSet
    @grammarMethod{*
        (add: (Int)+)
```

```
    *}
    func addMany: Array<Int> array {
        for elem in array {
            set add: elem
        }
    }


    @grammarMethod{*
        (addEach: Int)+
    *}
    func addManyKeywords: Array<Int> array {
        for elem in array {
            set add: elem
        }
    }

    override
    func asString -> String = set asArray asString;

    let Set<Int> set = Set<Int>();

end
```

In the code of the first annotation, the `+` after `(Int)` indicates that after `add:` there may be *one or more* `Int` arguments:

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

In the second method, `addManyKeywords`, the code in the annotation `grammarMethod` is different. The `+` appears after the keyword `addEach:` with the `Int` parameter type. That means the keyword may be repeated:

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

The annotated method, `addMany` and `addManyKeywords` in the example, should have a single parameter that matches the code of the DSL of the annotation `grammarMethod`. The rules for calculating the type of this parameter are given in Chapter 8.

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 | B` matches `A` or `B` (but not both), and `A B` matches `A` followed by `B`. The `|` operator may be used with types:

```
    @grammarMethod{*
        (add: (Int|String)+)
    *}
    func addUnion: Array<Int|String> array {
        for elem in array {
            set add: elem
```

```
        }
    }
```

Method `addUnion:` may receive as parameters a list of `Int`s and `String`s.

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.

There is one problem left: grammar methods are defined using regular expression operators. Therefore they can only parse regular languages. Some languages that are not regular can be defined by using more than one grammar method.

Another example is a domain specific language for commanding a car.

```
package grammar

object Car
    @grammarMethod{*
        (do:
           (on: | off: | left: | right: | move: Int)+
        )
    *}
    func carPlay: Tuple< Any,
                        Array< Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int> >
                  > t -> String {

        var s = "";
        for elem in t f2 {
            type elem
                case Any f1 { s = s ++ "car on " }
                case Any f2 { s = s ++ "car off " }
                case Any f3 { s = s ++ "car left " }
                case Any f4 { s = s ++ "car right " }
                case Int f5 { s = s ++ "car move($f5) " }
        }
        return s
   }

end
```

The car obeys commands related to movement such as to turn left, turn right, move n centimeters, turn on, and off. The method does nothing but in a real setting it could, for example, send commands to a real remote controlled car:

```
Car on:
    left:
    move: 100
    right:;

Car on:
```

```
    move: 200
    left:
    move: 50
    off;
```

These two message would cause the call of the same grammar method, `carPlay`.

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

Method `functionForMethod:` of prototype `Any` allows one to consider methods as objects.

```
object MySet
    func add: String elem { }
    ...
end
```

Method `functionForMethod:` takes the name of a method and returns a anonymous function that represents the method. That is, when message `eval` or `eval:` is sent to the function, the method is called.

```
    let Array<String> strArray = Array<String>();
    let Function<String, Nil> addMethod = strArray functionForMethod: "add:1";
    addMethod eval: "A";
    addMethod eval: "B";
    assert strArray size == 2;
```

Of course, `addMethod` could be passed as a parameter. That simulates the passing of a method as parameter.

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

```
object MenuItem
    func onMouseClick: Function<Nil> b {
        ...
    }
end

object Help
    func show { ... }
    ...
end

object FileMenu
   func open { ... }
end

var helpItem = MenuItem new;
helpItem onMouseClick: (Help functionForMethod: "show" );
```

```
var openItem = MenuItem new;
openItem onMouseClick: (FileMenu functionForMethod: "open");
...
```

There may even exist a table containing methods as functions. Assume `Test` declares methods `add10:`, `twice:`, and `cube:`, each one taking an `Int` and returning an `Int`.

```
let Test t = Test();
let IMap<String, Function<Int, Int>> methodMap = [
    "add10" -> (t functionForMethod: "add10:1"),
    "twice" -> (t functionForMethod: "twice:1"),
    "cube" -> (t functionForMethod: "cube:1")
    ];
cast f = methodMap["twice"] {
    assert f eval: 3 == 6;
}
cast f = methodMap["add10"] {
    assert f eval: 3 == 13;
}
cast f = methodMap["cube"] {
    assert f eval: 3 == 27;
}
```

The Exception Handling System

The exception handling system of Cyan was based on that of language Green [Gui13] [Gui06]. 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 using statement `throw` that takes an exception object as parameter. The type of the exception object should be subprototype of `CyException`.

An exception is caught using a `try-catch-finally` statement.

```
var age Int;
try
    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

```
throw ExceptionNegAge(age)
```

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

After a `catch` *clause* there should be an expression whose type declares at least one `eval:` method that takes a parameter whose type is subprototype of `CyException`. In this example, the anonymous function defines a method

```
func eval: ExceptionNegAge
```

Assume `ExceptionNegAge` inherits from `CyException`.

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

In this specific case, when the exception is thrown, the control is transferred to the function given after `catch`. The error message is then printed.

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` clauses as necessary, each one taking a single expression.

```
var Int age;
try
    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` expression may result in an object with more than one `eval:` method, each of them accepting one parameter whose type is subprototype of `CyException`. So the following code is legal.

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

```
    }
catch ExceptionCatchAge;
```

Consider that `ExceptionCatchAge` is

```
object ExceptionCatchAge
    overload
    func eval: ExceptionNegAge e {
        Out println: "Age ", e age, " is negative"
    }
    func 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 object of the `catch` clause, 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` clauses.

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 Guimarães [Gui04].

Metaobjects

Compile-time metaobjects are objects that can change the behavior of the program, add information to it, or inspect the source code. A compile-time metaobject is activated by a metaobject annotation that may appear before a prototype, a method, a field, a local variable, as a statement of a method, as an expression, etc.

A metaobject annotation starts with `@` followed by the metaobject name:

```
@checkStyle
object University
   @log
   func name -> String { return uName }
   ...
end
```

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 field names, and local variables follow some conventions for identifiers (prototype in lower case except the first letter, method keywords 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).

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.

Chapter 2

The Compiler, Packages, and File organization

This Chapter describes how the Cyan source files of a program are organized and how the compiler should be called. To explain that we need to define some terms. We will call "program unit" a prototype declaration or interface. Every source file is a "compilation unit" and may contain one prototype declaration. Then a compilation unit is a "source file" containing one prototype.

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

(a) Keyword `public` may precede the program unit to indicate that it is public . A "public" program unit of a package `pp` is visible in all compilation units that import `pp`. Future versions of the language will allow visibility "package". A "package" program unit of a package `pp` is only visible in the compilation units of `pp`. Every file, with extension .cyan, declare exactly one program unit. Private program units are not currently supported by Cyan.

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

If no qualifier is used before "`object`", then it is considered public.

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
   var String name
   var Type type
   ... // methods
end
```

The prototype 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. In a package id1.id2. ... idn, each idi should start with a lower-case letter.

In the directory of a package the compiler and metaobjects may store files used by the compiler, the Cyan Metaobject Prototocol, and possibly some tools such as the IDE. These files may keep information on the source files and they link past and future compilations. This mechanism is called "link past-future (LPF)". See Section "Special Package Directories" in the Thesis "The Cyan Language Metaobject Protocol" available in www.cyan-lang.org.

The information stored in the LPF files can be used to catch errors at compile time that would otherwise go undetected or to improve current error messages. Based on the information, the compiler could check:

(a) if the textual order of declaration of overloaded methods[1] was changed;

(b) if methods were added to an overloaded method;

(c) if a compilation unit (source code) changed between compilations in such a way that the changes were prohibit by a metaobject used in the previous version of the compilation unit. This is a research topic (await!);

(d) the introduction of a new local variable may change the semantics of a method that accessed a field with that same name.

The "info files" could also store information collected at runtime. The compiler could insert code that checks, for example, if the numbers stored in `Int` variables are dangerously 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`" or `long`.

A package is a collection of prototype declarations and interfaces. Every Cyan prototype declared as `object ObjectName ... end` must be in a file called "`ObjectName.cyan`". 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
   func 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.

[1] Methods with the same name but different parameter types in each keyword.

More than one package may be imported; that is, the word `import` may be followed by a list of package names separated by commas. 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
   func 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 7.

A program is described by a file with extension "`pyan`". This file contains code of a Domain Specific Language called Pyan (Project cYAN) whose grammar follows.

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

Some itens 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. An identifier ending with two or more underscores is illegal (`"name__"` is not valid). 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 a escape character for any `x`. Then a FileName can be

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

As an example, a program in Pyan could be[2]

```
import styles.default at "C:\Cyan\my"
@checkStyle
@option(addQualifier)
program at "C:\Cyan\example01"
    main main.Program
    package main at "C:\Dropbox\Cyan\cyanTests\general-0002\main"
    @option(no_dynamic)
    package bank at "C:\Cyan\tests\bank"
    package cyan.util at "C:\Cyan\cyan\util"
    package account
    package database at "database\util"
```

Keyword `program` starts the project. Optionally "at" specifies the path of the program. If not specified, the *default project 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. If this directory is not absolute, it is considered to be relative to the project directory. All source files of a package should be in the same directory. In the above example, package `database` should be in directory

```
C:\Cyan\example01\database
```

The compiler considers that the package is in a sub-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\util`". This directory is the one specified by "`at`"

As an example, package `cyan.util` is in directory

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

In the above Pyan file, no directory is specified for package `account`. Therefore it is in the directory of the program in a sub-directory with the name of the package: `C:\Cyan\example01\account`

The program and the packages may be preceded by zero or more metaobject annotations. These are of the for `@meta` in which `meta` is the metaobject name. These calls may have parameters and an attached text (as `@meta(param){* text *}`). See the thesis on the Cyan MOP in the Cyan site for more details. In particular, the compiler options should be parameters of a metaobject `options`.

`ImportList` is a list of packages whose metaobjects are imported by this project file. The package should be in the directory that follows "`at`". For example, package "`styles.default`" should be in directory

```
C:\Cyan\my\styles\default
```

[2]This could be a program in Pyan. But since metaobject `option` is not yet implemented, it is not.

Only the metaobjects are imported. These metaobjects may be used in this file. Then `checkStyle` could be in package "`styles.default`". Or it could be in the cyan.lang directory, which is always imported.

In a project file, the compiler considers that every directory of the program directory corresponds to a package, including sub-directories. The project file may use the `package` keyword to include one package of the program directory, but that is optional. As an example, suppose file "`p.pyan`" is in directory

```
C:\Dropbox\Cyan\cyanTests\master
```

which has the following directory tree:

```
C:\Dropbox\Cyan\cyanTests\master
C:\Dropbox\Cyan\cyanTests\master\generic
C:\Dropbox\Cyan\cyanTests\master\generic\ga
C:\Dropbox\Cyan\cyanTests\master\main
C:\Dropbox\Cyan\cyanTests\master\shape
```

File "`p.pyan`" has the contents

```
program
    @checkStyle
    package shape
```

Since there is no "`at`" after "`program`", the compiler considers that the program directory is

```
C:\Dropbox\Cyan\cyanTests\master
```

The compiler consider that the program has the packages `generic`, `generic.ga`, `main`, and `shape`. It was necessary to declare explicitly package `shape` in "`p.pyan`" because there is a metaobject annotation preceding it.

Appendix A shows how the compiler should be called, the compiler options, etc.

2.1 Future Enhancements

In the directory of a package there should be zero or more Cyan source files or source files of Domain Specific Languages. A file name "Name.cyan" should contain a public or package prototype `Name`. There are special rules for names of source files with generic prototypes — see Chapter 6. A file with extension "`.syan`" is a script file of a language called ScriptCyan. This language has a slightly different grammar from Cyan. The source file does not start with the package declaration. It may start with import declarations. After that there are two options:

(a) statements that usually are inside a method;

(b) methods, shared fields, and fields that usually are inside a prototype declaration. In this case there should be at least one method and the first statement after the import declarations (if any) should be a method. Since there is no "`object`" keyword in the source, the user will have the impression that the methods are procedures and functions (not related to a prototype).

In case (a) the compiler will add a prototype declaration, import package `script`, and will insert the statements in a method `run: Array<String> args`. Of course, parameter `args` can be used inside the statements. As an example, suppose the code below is in a file called "`PrintArgs.syan`" of a directory "`C:\Cyan\tests\myTest\argsTest`". The project file informs that the program directory is "`C:\Cyan\tests`". Therefore the compiler deduces that the package name is "`myTest.argsTest`". The compiler will transform code

```
args foreach: { (: String s :)
    s println
};
```

into the code

```
package myTest.argsTest
import script;

open
object PrintArgs extends ScriptCyan
    func run: Array<String> args {
        args foreach: { (: String s :)
            s println
        };
    }
end
```

Package `script` will contain several prototypes for scripting. It has not been defined yet.

In case (b), the compiler will insert the methods, shared fields, and fields into a prototype that has the same name and package as in case (a).

It is expected that prototype `ScriptCyan` declares several methods to make it easy to build script files. These methods are not yet defined.

The file name of a ScriptCyan source file may contain a symbol "`-`" followed by "`s`" and a prototype name possibly preceded by a package:

```
PrintArgs-s-mypack-MyScript.syan
```

In this case prototype `PrintArgs` will have "`mypack.MyScript`" as superprototype. `mypack` should be a package of the project. Language `ScriptCyan` is not supported by the current Cyan compiler yet.

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 "`.pyan`" file (the project file). Method `run` cannot be a true overloaded method (Section 4.14) or be inherited. 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
    func 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, an identifier without letters or numbers, only with underscores, is not valid. An identifier cannot end with two underscores too (like "`one__`" or "`one___`"). 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`".

There is a restriction on identifiers in Cyan: prototype names should start with an upper-case letter and variables (local variables, parameters, and fields) names should start with a lower-case letter or underscore. For parameters, both the name and its type (but not both) are optional:

```
func with: Int do: action { ... }
```

The parameter of `with:` does not have a name and therefore it cannot be referenced inside this method. Usually parameters of interfaces (Section 4.13) do not have names. The parameter of `do:` has name `action` but it does not have a type. It is assumed its type is `Dyn`, the dynamic type. No message sends to expressions that have type `Dyn` are checked by the compiler. And all assignments to and from expressions of this type are not checked too.

3.2 Comments

Comments are parts of the text ignored by the compiler. Cyan supports two 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;

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		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	func	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 4.17 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;
```

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`. Besides that, there are prototypes `Nil` and `String`, also considered basic types.

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 but `Nil` inherit from `Any`. Therefore there are not two separate hierarchies for basic and regular 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` may be allocated in the stack.

Integral literal numbers without a postfixed letter are considered as having type `Int`. Numbers with a dot such as `10.0` as considered as `Double`´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 byte0;
var Float height;
var Double width;
   // ok
age = 21;
   // compile-time error, 0 is Int
byte0 = 0;
   // ok
byte0 = 0B;
   // ok
height = 1.65F;
```

```
   // compile-time error
height = 1;
   // compile-time error, 1.65 is Double
height = 1.65;
   // ok
height = 1F;
width = 1.65;
width = 1.65Double;
```

Underscores can be used to separate long numbers as in

```
1_000_000
```

Two or more 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 superprototype of anyone (this will be explained latter). `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` or `Dyn`. Of course, `Nil` cannot be inherited from a prototype.

Methods that do not declare a return type, as

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

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

```
func 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 immutable string. It has several methods such as `at:` `[]` (for indexing) and `==` (equality). String literals can be expressed in three forms: enclosed by `"` as in C/C++/Java, enclosed by `"""`, and starting with `#`.

A literal string delimited by `"` has the same syntax 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. There is a literal string that spans throught multiple lines. It starts with three characters `"` as in the example:

```
var s = """
    This is a text
    that uses
    more than one
    line""";
s println;
```

A special multi-line literal string starts with `|"""`, the `|` character before the quotes. Both `|` and all characters before it are eliminated. Every `|` should be in a different line, although there may be empty lines between two lines. All `|` characters should be in the same column. All white spaces before the closing `"""` are removed.[1] The result is a multi-line string in which `|` delimited the start of each line.

[1] `ch` is a white space if the Java method `Character.isWhitespace(ch)` returns true.

```
var String s;

s = |"""
    |first5

    |second5
      """;

s = |""" |first3
         |second3
       """;

s = |"""
         |first2
         |second2
      """;


s = |"""
 |first4
 |second4
      """;

s = |"""
 |first12

 |second12

      """;

s = |"""|first13

 |second13""";
```

A literal string can also be represented by "symbols". A symbol starts with # followed, without spaces, by letters, dot, digits, underscore, and any number of :´s, as in these examples:

```
#f #age #age:
#123 #_0 #field001
#foreach:do: #main.package
```

A single-quote literal string may start with n" or 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 '"'. Three quoted literal strings disable escape characters by default. Then """\n""" has two characters.

The value of a variable can be inserted in a literal string at runtime by preceding its name, inside the literal string, by $. This does not work with symbols because they cannot contain $.

```
var n = 5, k = n*n;
assert "n = $n, k = $k" == "n = 5, k = 25";
```

```
var s = """
    The values of n and k are
        $n
        $k
    """;
s println;
```

Method `eq:` of `String` returns true if the argument and `self` have the same contents. It always give the same result as method `==`.

```
var s = "cyan";
var p = s;
assert p == s;
assert "cyan" == """cyan""";
assert #cyan == "cyan" && """cyan""" == #cyan;
assert s == "cyan" && #cyan == s && "cyan" == s;
assert p == s && p eq: s;

assert "\\n" == """\n""";

assert p eq: s;
assert "cyan" eq: """cyan""";
assert #cyan eq: "cyan" && """cyan""" eq: #cyan;
assert s eq: "cyan" && #cyan eq: s && "cyan" eq: s;
assert p eq: s && s eq: p;
```

Method `neq:` returns the negation of the result of `eq:`. For basic type objects, it always return the same value as `!=`.

```
assert "cyan" != #green;
assert """Cyan""" != "green";
assert ("cyan" != #green) == ("""Cyan""" != "green");
assert (#green != """cyan""") == (#green eq: """cyan""");
```

Types `Byte`, `Short`, `Int`, `Long`, `Float`, and `Double` support almost the same set of arithmetical and logical operators as the corresponding types of Java. The *binary or* operator in Cyan is `|||` instead of `|` because the later is used exclusively for union types.

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
    func eq: (Dyn other) -> Boolean
    func neq: (Dyn other) -> Boolean
    func + (Int other) -> Int
    func - (Int other) -> Int
    func * (Int other) -> Int
    func / (Int other) -> Int
```

```
    func % (Int other) -> Int
    func < (Int other) -> Boolean
    func <= (Int other) -> Boolean
    func > (Int other) -> Boolean
    func >= (Int other) -> Boolean
    func pred -> Int
    func succ -> Int
    func odd -> Boolean
    func even -> Boolean
    func prime -> Boolean
    func isMultiple: Int n -> Boolean
    func maxValue -> Int
    func minValue -> Int
    func == (Dyn other) -> Boolean
    func != (Dyn other) -> Boolean
    func <=> (Int other) -> Int
    func .. (Int theEnd) -> Interval<Int>
    func ..< (Int theEnd) -> Interval<Int>
    func - -> Int
    func + -> Int
    func & (Int other) -> Int
    func ||| (Int other) -> Int
    func ~| (Int other) -> Int
    func ~ -> Int
    func <.< (Int other) -> Int
    func >.> (Int other) -> Int
    func >.>> (Int other) -> Int
    func |> Function<Int, Int> f -> Int
    func asByte -> Byte
    func asShort -> Short
    func asLong -> Long
    func asFloat -> Float
    func asDouble -> Double
    func asChar -> Char
    func asBoolean -> Boolean
    func asInt -> Int
    func asString -> String
    func to: (Int max) do: (Function<Int, Nil> b)
    func to: (Int max) into: (InjectObject<Int> injectTo)
    func times: Function<Nil> b
    func repeat: Function<Int, Nil> b
    func to: (Int max)
    func in: (Iterable<Int> container) -> Boolean
    func between: Interval<Int> inter -> Boolean
    func hashCode -> Int
    func defaultValue -> Int
end
```

```
...


abstract object InjectObject<T> extends Function<T, Nil>
    override
    abstract func eval: T
    abstract func result -> T
end

interface Iterator<T>
    func hasNext -> Boolean
    func next -> T
end
```

Some of the basic types have have methods to simulate pipes:

```
(5 |> Sqr |> { (: Int elem :) ^2*elem } ) print;
```

Assuming that `Sqr` extends `Function<Int, Int>` and calculates the square of its parameter, this command will print `50`. It is equivalent to:

```
({ (: Int elem :) ^2*elem } eval: (Sqr eval: 5) ) print;
```

Prototype `Boolean` uses `=>` as an `if` command:

```
i < 3 => { "baby" println };
i > 19 => { "adult" println };
```

This method takes a literal function as argument.

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

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

Idem for `-`.

A prototype may declare an operator `[]` and use it just like an array (see Section 4.10).

Each basic prototype `T` but `Float`, `Double`, and `Nil` 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 implement this interface.

Intervals implement the `Iterable<T>` interface. Then we can write

```
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"
}
```

`&&` and `||` are not methods of prototype `Boolean`. They are instead operators of the language and use the regular short-circuit evaluation. That is, `aa && bb` is `false` if `aa` is `false`. In this case `bb` is not even evaluated. So the `if` statement below is safe.

```
if index < array size && array[index] == x {
    Out println: "found $x"
}
```

Prototype `Boolean` has a logical not, `!`.

```
if ! ok { Out println: "fail" }
if age < 0 || age > 127 { Out println: "out of limits" }
```

Method `++` defined in `Any`, the superprototype of every one but `Nil`, concatenate the string of the receiver plus the string of the argument:

```
assert 1 ++ 2 == "12" &&
       1 ++ 'A' == "1A" &&
       0 ++ "1" == "0" ++ 1;
```

Prototype `String` support the `in:` method:

```
func daysMonth: (String month, Int year) -> Int {
    if month in: [ "jan", "mar", "may", "jul", "aug", "oct", "dec" ] {
        return 31
    }
    else if month in: [ "apr", "jun", "sep", "nov" ] {
        return 30
    }
    else if month == "fev" {
        if leapYear: year { return 29 } else { return 28 }
    }
```

```
||
~||
&&
=>  ==>
!
==   <=   <   >   >=   !=   ===   !==   <=>   ~=
non-unary message send
|>
++  -- (binary)
..  ..<
+   -
/   *   %
|||  ~|   &
<.<   >.>    >.>>
.*  .+  .%
unary message send
+   -   !  ~   (unary)
```

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

```
    else {
        return -1
    }
}
```

3.6 Operator and Keyword Precedence

Cyan has special precedence rules for methods and operators whose names are the symbols given in Figure 3.1. 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 keywords. 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 and operators 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 keyword 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 5.

3.7 Loops, Ifs, and other Statements

Currently each message send, assignment, and local variable declaration should end with a semicolon (";"). However we expect to make the semicolon optional as soon as possible, at least in some cases. The semicolon is optional for all statements such as `if`, `while`, `for`, and `type-case`.

Decision and loop statements that not use the return statement can be implemented using 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 `t:f:` and `f:t:` in `Boolean` that return an expression or another according to the receiver:

```
var String s;
var Any any = ( n%2 == 0 ) t: "even" f: "odd";
```

```
type any
    case String s2 {
        s = s2
    }
any = ( n%2 != 0 ) f: "even" t: "odd";
type any
    case String s2 {
        s = s2
    }
```

If the expression is true, the expression that is parameter to `t:` is returned. Otherwise it is returned the parameter to `f:`. Since the return type of methods `t:f:` and `f:t:` is `Any`, the example uses command `type-case` to cast `any` to `String`.

As a future improvement, a metaobject `checkTF` will check whether the arguments of both keywords have the same type[2] (both are strings in this case). This metaobject will also cast the value returned to the correct type, `String` in the example. The return type is `Any`. That is why it is necessary to use command `type-case` in the example.

Note that an `if` statement that needs a `return` cannot be implemented using message sends:

```
(i == 0) ifTrue: {
    return false; // compile-time error.
};
```

The `return` statement cannot appear inside a function.

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 are shown in this example:

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

[2]For the time being, one cannot be subtype of another.

```
    Out println: i;
    ++i
}
```

The `}` that closes a `while` statement should be either in the same line as keyword `while` or in the same column as it. There should be no semicolon after the closing `}`.

There is a `repeat-until` statement that executes its statements until the `until` expression evaluates to `true`.

```
var Int sum = 0;
var Int n = 1;
repeat
    sum = sum + n;
    ++n;
until n >= 4;
assert sum == 6;
```

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"
}
```

Unlike the languages of the C family, the parentheses around the boolean expression are not necessary and the `{` and `}` are required. The `';'` after `if` and `while` statements are not necessary. There are several restriction on the formatting of `if` statements:

(a) the column of the `{` that follows keyword `if` should be in a column greater or equal than the column of the `if` keyword;

(b) the column of the first symbol of the `Boolean if` expression should be in a column greater or equal than the column of the `if` keyword;

(c) if the `}` that closes the statement list that follows an `if` is in line `lineR` and column `columnR`, than `lineR` should be equal to the line of the `if` or the line of the previous `else` or `columnR` should be equal to the column of the `if` or the column of the previous `else`;

(d) `else` should be in the same line or column of the previous `else` or `if` keywords;

(e) the `}` that closes the `else` statements should be in the same line as the `else` or in the same column;

(f) the `}` that closes an `if` or `else` may be in the same line as the next `else`:

```
if 0 < 1 {
    "ok" println;
} else if 1 < 2 {
    "ok" println;
} else {
    "not ok" println;
}
```

In this case, the `}` that closes the **next** `else` or `if` should be in the same column as the **previous** `}`.

These rules assert that `if` statements are almost always clearly formatted. See the examples with formatting errors.

```
if 0 < 1 { }
else if 0 < 4 {
}
else if 0 < 1 {
   } // '}' is not in the same column or line as 'else'


if 0 < 1 { }
else if 0 < 4 {
}
     else if 0 < 1 {
} // '}' should be in the same line or column as 'if'

if 0 < 1 { } else
    if 0 < 5 { }
    else
        if 0 < 4 { } else if
                // if expression in a column smaller than the if column
            0 < 1 { }
        else {
    };


if 0 < 1 { } else
    if 0 < 5 { }
    else
        if 0 < 4 { } else if
                        0 < 1 {
                        }
           // 'else' must be in the same line or column
           // as the previous 'if'
        else {
    };
```

```
if 0 < 1 { } else
    if 0 < 5 { }
    else
        if 0 < 4 { } else if
                        0 < 1 {
                        }
                        else {
       // this '{' closes an 'else' statement and should be
       // in the same line or column as it
    };


if 0 < 1 { } else
    if 0 < 5 {
          // the '}' that closes an 'if' should be in the same
          // line or column as it
        }
        else { }

if 0 < 5 { 0 println }
       // 'else' must be in the same line or column as the previous 'if'
    else { 1 println }


if 0 < 5 { 0 println
}
    // else should be in the same column as '{'
  else { 1 println }
```

These examples have no errors:

```
    if 0 < 6 { } else { }

    if 0 < 1 { }
    else if 0 < 4 {
         }
         else if 0 < 1 {
         }

    if 0 < 1 { }
    else
         if 0 < 4
         {

         }
         else if 0 < 1 {
         }
```

```
if 0 < 1 { } else
    if 0 < 5 { }
    else { }

if 0 < 1 { } else
    if 0 < 5 { }
    else
        if 0 < 4 { } else if
                        0 < 1 {
                        }
                        else {
                        };

if 0 < 1 { } else
    if 0 < 5 { }
    else { }


if 0 < 5 { 0 println }
else { 1 println }

if 0 < 1 { } else if 0 < 5 { }
            else { }

if 0 < 1 {
} else if 1 < 2 {
} else {
}
```

Statement `for` can be used to iterate over any object that implements a method

```
iterator -> Iterator<T>
```

Its syntax is

```
for [Type ] elem in list {
    // statements
}
```

The `Type` of `elem` is optional. `elem` will assume the elements given by method `next` of the iterator returned by method `iterator` of `list`. When method `hasNext` of the iterator returns `false` the loop ends. The `Type` should be exactly the type of the elements returned by the iterator. It cannot be a supertype of that type.

This command can be used as in

```
for elem in [ 2, 3, 5, 7 ] {
    "$elem is prime" println
}
for ch in 'a'..'z' {
    ("letter " ++ ch) println;
}
```

```
var sum = 0;
for i in 0..< 10 {
    sum = sum + i
}
sum println;
sum = 0;
for i in 1..10 {
    for j in 1..10 {
        sum = sum + i*j;
    }
}
```

The `}` that closes a `for` statement should be either in the same line as keyword `for` or in the same column as it. There should be no semicolon after the closing `}`.

`elem` should not have been previously declared as a local variable or parameter. `elem` is alive only inside the statements of this `for`. Its type is deduced by the compiler if it is not given.

Since variable `elem` is only alive inside the `for` command, it can be reused:

```
for elem in [ 2, 3, 5, 7 ] {
    "$elem is prime" println
}
for elem in 0..< 10 {
    elem println;
}
```

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;
```

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

```
    // this code prints numbers 0 1 2
var sum = 0;
var i = 0;
3 repeat: { (: Int n :)
    sum = sum + n
};
assert sum == 3;
    // 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: { (: Int n :)
        n println;
    };
        // 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
};
```

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>
    func == (Dyn other) -> Boolean
    func != (Dyn other) -> Boolean
    func add: (T elem)
    func add: (Int i, T elem)
    func clear
    func isEmpty -> Boolean
    func remove: (Int i)
    func [] at: Int index -> T
    func [] at: Int index put: (T elem)
    func last -> T
    func asString: (Int ident) -> String
    func slice: (Interval<Int> interval) -> Array<T>
    func concat: Array<T> other -> Array<T>
    func size -> Int
    func foreach: Function<T, Nil> b
    func filter: Function<T, Boolean> f -> Array<T>
    func filter: Function<T, Boolean> f foreach: Function<T, Nil> b
    func map: Function<T, T> f -> Array<T>
    func iterator -> Iterator<T>
    func contains: T elem -> Boolean
    func indexOf: T elem -> Int
    func apply: (String message)
    func apply: (String message) select: (String slot) -> Dyn
    func .* (String message)
    func .+ (String message) -> Dyn
end
```

Arrays supports some interesting methods: `apply:`, `.*`, and `.+`. The first two ones applies an operation given as string to all array elements. Method `.+` sums all array elements or return the first element if the array has just one element. It is assumed that the type of the array element supports a binary operation "+".

```
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 `slice:` 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;
```

3.9 Maps

A literal map is delimited by `[` and `]` as an array and uses `->` for mapping a *key* to *value*.

```
let IMap<String, Int> map = [ "Newton" -> 1642, "Gauss" -> 1777 ];
  // method asArray return an array with tuple elements
for elem in map asArray {
    Out println: elem key, " was born in the year ", elem value;
}

cast year = map["Newton"] {
    "Newton was born in $year" println;
}
```

The type of the literal map is `IMap<K, V>` in which `K` is the type of the *key* (appears before `->`) and `V` is the type of the values (after `->`). The expression `map[key]` returns a value of type `Nil`V| and, therefore, a `cast` statement (Section 4.12) is need for retrieving the value. All the keys should have the same type and all the values should have the same type. `IMap` is an interface. The literal object will have a prototype that implements `IMap`.

```
let errorMap = [ Any -> "Any", Int -> "Int" ];
```

Although `Int` is a subprototype of `Any`, the compiler will sign an error in this code.

Method `asArray` of `IMap<K,V>` returns an array with all elements of the map. Each element is a tuple

```
Tuple<key, K, value, V>
```

The first example of this section iterates over the map elements using a `for`. A single element can be got using indexing or the `get` method of `IMap`. The element returned has type `V|Nil`. If the *key* passed as parameter is in the map, the value returned is the *value* associated to it. If the *key* is not in the map, `Nil` is returned.

Chapter 4

Main Cyan Constructs

A prototype may declare zero or more *slots*, which can be variables (called fields) shared field variables (to be seen later), and methods (called instance methods). In Figure 4.1, there is one field, `name`, and two methods, `getName` and `setName`. Keywords `public`, `private`, `package`, and `protected` can precede a method declaration. Currently only keyword `private` (or none) can precede a field declaration. A public method can be accessed anywhere the prototype can. A private method and field can only be used inside the prototype declaration. Protected methods can be accessed in the prototype, its subprototypes, sub-subprototypes, and so on. A subprototype inherits from a prototype — that will soon be explained.

In the declaration of a field, there are four optional parts:

1. the visibility (only `private`);
2. keywords "`var`" or "`let`" 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. There are restriction on the expression `expr`. See Section 4.3.

A read-only field is declared with the word `let` ou without any keyword:

```
object Person
    let String name
    Int age
    ...
end
```

Both `name` and `age` are read-only. A read-only variable can receive a value in its declaration and it can receive a value in the constructor (to be seen). But a regular method cannot assign a value to it. This is the default, used if no keyword precedes the field declaration. If a variable should change its value after the object is created, declare it with `var` as in

```
object Person
    func init: String name, Int age {
        self.name = name;
        self.age = age
    }
    var String name
```

```
    var Int age
    func getName -> String = name;
    func setName: String other { self.name = other }
    func getAge -> Int = age;
    func setAge: Int other { self.age = other }
end
```

A single type can be used for more than one field:

```
object Rectangle
    ...
    Int x1, y1, x2, y2
end
```

But we can only assign a value to a declaration with a single value:

```
object Rectangle
    ...
    Int x1 = 0; // ok
    Int y1 = 0, x2, y2 // compile-time error
end
```

```
package bank

object Client
  func getName -> String {
     return self.name
  }
  func setName: String name {
     self.name = name
  }
  func print {
      Out println: name
  }
  private var String name = "";
end
```

Figure 4.1: An object in Cyan

Package `cyan.lang` supplies a metaobject `property` that creates methods for getting and setting the value of a field.

```
package people

object Person
    func init: String p_name, Int p_age {
        _name = p_name;
        age = p_age;
    }
    @property var String _name
```

```
    @property var Int age
end
```

The name of the methods created depend on whether the variable name starts with underscore or not. If it does, as `_name`, the methods created do not start with `get` and `set`. Otherwise methods starting with `get` and `set` are created. In this example, the methods would be

```
    func name -> String = _name;
    func name: String other { self._name = other }

    func getAge -> Int = age;
    func setAge: Int other { self.age = other }
```

`property` used with a read-only variable causes the creation of a `get` method only:

```
package people

object Person
    func init: String p_name, Int p_age {
        _name = p_name;
        age = p_age;
    }
    @property String _name // only get is created
    @property let Int age // only get is created
end
```

A method declared without a qualifier is considered `public`. A prototype declared without a qualifier is considered `public` (currently prototypes are always public). A field without a qualifier is considered `private` (currently they are always private). Then, a declaration

```
package Bank

object Account
      // constructor, to be seen later
    func init: Client client { self.client = client }

    func set: Client client {
        self.client = client
    }
    func print {
        Out println: (client getName)
    }

    var Client client
end
```

is equivalent to

```
package Bank

public object Account
      // constructor, to be seen later
    public
```

```
    func init: Client client { self.client = client }

    public
    func set: Client client {
        self.client = client
    }
    public
    func print {
        Out println: (client getName)
    }
    private Client client
end
```

The declaration of local variables is made with the syntax:

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

The last line declares four variables of type `Int`. Keyword `var` or `let` is demanded in the declaration of local variables. Variables declared with `let` are read-only and should receive an expression in the declaration:

```
let Double pi = 3.1415;
let Int maxSize = 100;
let Char endLine; // compile-time error
```

Further assignments to the variable are forbidden.

A declaration can have a single type and several variables as that of `x1`, `y1`, `x2`, and `y2` above. But a variable that receives an expression should be in its own declaration:

```
var String name;
var Int x1 = 0;
var Int y1, x2 = 0, y2; // compile-time error
```

We will call `block` a sequence of statements delimited by `{` and `}` that appear in a command like `while`, `if`, and `type-case`. A anonymous function is not a block. The scope of a local variable is from where it was declared to the end of the function or block in which it was declared:

```
1     func 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 `}`).

For short, you cannot declare a variable if there is another visible at that point:

```
1     func nothing: Int p -> Int {
2         var Int n = 0;
3         if 0 < p {
4             var Int k = n;
5             var Int n = 2; // redeclaration
6             return k + n
7         }
8         else {
9             var Int k = n + 1; // ok
10            return k
11        }
12    }
```

The two declarations of `k` are correct because the scope of the first `k` ends in line 7. The scope of `n` ends in the closing `}` of the method. Therefore the declaration of `n` in line 5 is illegal.

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

```
var String name;
```

prototype "`String`" plays the role of a type. Then a prototype name can play two roles: 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 field.

A local variable can be declared and assigned a value:

```
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, to the variable should be assigned a value before it is used. When the type is omitted, the syntax

```
var variableName = expr
```

should be used to define the variable as in:

```
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.

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;
```

A method is declared with keyword `func` followed by the method keywords 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 should be an identifier not followed by a ":". For example, `print` in Figure 4.1 is a unary method.

When a method takes parameters it should have one or more keywords, each one ending with ":" as in

```
func at: Int n { ... }
```

```
func at: Int n put: String s { ... }
func with: Int n
     with: Int another
     concat: String s -> String { ... }
```

This kind of method is called a keyword method or a method with keywords.
There may be a method that is not unary but that does not take parameters:

```
func open: String
     read: { ... }
```

Smalltalk does not allow that. However, it is illegal to declare a method without parameters that is a keyword method:

```
    // compile error
func read: -> String { ... }
```

An optional return value type can be given after keyword `func`. The return value should be given by the `return` command. The return expression should be subtype (Section 4.17) 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.

4.1 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. A field `age` can be accessed in a method of a prototype by its name or by the name preceded by "`self.`" as in

```
func getAge -> Int {
   return self.age
}
```

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

4.2 clone Methods

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

```
func clone -> P
```

that returns a *shallow* copy of the current object. In the shallow copy of the original to the cloned object, every field 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.

4.3 init and new Methods

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`. For each method named `init:` the compiler adds to the prototype a method named `new:` with the same parameter types. Each `new` method creates an object without initializing any of its slots and calls the corresponding `init` method (idem for `init:` and `new:`). If the prototype does not define any `init` or `init:` method, the compiler supplies an empty `init` method that does not take parameters and calls the superprototype `init` method (if there is one. If not, an error occurs).

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

(a) should be declared with no return type (it cannot be `Nil`);

(b) cannot be called using reflection at runtime. These methods are used to create `new` and `new:` method and them discarded. They do not exist at runtime. However, `new` and `new:` can be called using reflection. The do exist at runtime;

(c) should not be preceded by keyword `override`;

(d) should not be abstract or final;

(e) should not be indexing methods (See Section 4.10);

(f) an `init:` method should take at least one parameter;

(g) cannot be declared in interfaces;

(h) two `init:` methods can have the same number of parameters. However, there should be at least a number `n` such that the n^{th} parameter type in one method is not subtype or supertype of the n^{th} parameter type of the other method. If this were allowed, there would be an ambiguity when calling method new:

```
object Pet
    func init: Animal animal { ... }
    func init: Dog dog { ... }
    ...
end

// in other prototype:
    let Dog meg = Dog("Meg");
    let Pet myPet = Pet new: meg;
```

In the creation of object `Pet`, both the first and second `init:` methods could be called (if the compiler did not signal an error in this code — but it does).

To solve this problem, one can use unions:

```
object Pet
    func init: Dog|Animal animal { ... }
    ...
end
```

The following declaration is illegal because `Wrong(0, 0, 0)` would be ambiguous.

```
object Wrong
    func init: Any a, Int b, Any c { ... }
    func init: Any a, Any b, Any c { ... }
    ...
end
```

The following declaration is legal because the third parameter allows the compiler to differentiate between the two constructors.

```
object Fine
    func init: Any a, Int b, Int c { ... }
    func init: Any a, Any b, String c { ... }
    ...
end
```

(i) can only be called by immediate subprototypes using `super` as the message receiver. That is, if `C` inherits from `B` that inherits from `A`, then `C` cannot call the `init` or `init:` methods of `A`. To call these methods of the immedidate superprototype, use "`super init`", and "`super init: args`" as the first statement of the `init` or `init:` method of the subprototype. Currently `init` methods of the same prototype cannot be called (this will change, of course).

There are further restrictions related to methods `init`, `init:`, `new`, and `new:`, given below.

(a) A user-declared method cannot have name `new` or `new:`.

(b) No keyword of any user-declared method can be `init:` or `new:`. A future addition to the language would be to allow a constructor to start with `init:` and have other keywords:

```
object Person
    func init: String name { self.name = name; age = 0; salary = 0Float; }
    func init: String name
```

```
            age: Int age
            salary: Float salary {
            self.name = name;
            self.age = age;
            self.salary = salary
        }
        String name
        Int age
        Float salary
    end
```

The compiler would create, for this prototype, methods "`new: String`" and "`new: String age: Int salary: Float`".

Methods `new` and `new:` are only accessible through prototype objects. That means one cannot send a `new` or `new: args` message to an expression that is not a prototype:

```
object Test
    func init: String s { ... }
end

object Program
    func run {
        var t = Test clone;
        var Test u;
            // Ok !
        u = t clone;
            // compile-time error
        u = t new: "hi!";
            // ok
        u = Test new: "hi!";
    }
end
```

`init` and `init:` methods can have visibility `public`, `private`, `protected`, and `package`. The `new:` or `new` method created from the `init:` or `init` method has the same visibility as this one. A `private` method can only be called inside the prototype in which it was declared. A `protected` method can only be called in the prototype it was declared and subprototypes. A `package` method can be called in all prototypes of the same package in which it was declared. A `public` method is accessible anywhere.

A singleton prototype is created by declaring a single `private init` method, a redefined `clone` method, and no `init:` methods.

```
object Earth
    private func init { }
    private func clone -> Earth = Earth;
    // other regular methods
    // that do not create Earth objects
end
```

Unlike most object-oriented languages, Cyan demands that all fields be initialized before used. To assure that, the language puts severe restrictions on constructors:

(a) every `init` or `init:` method of a prototype should initialize every field of the prototype that is not initialized in its declaration. Shared fields are not considered because they have already been initialized by `initShared` methods or in their declarations. Then

```
object Manager
    func init: String name { self.name = name }
    var String name
    var Float salary = 1000F;
end
```

is legal but the following prototype is illegal.

```
object Manager
        // salary is not initialized
    func init: String name { self.name = name }
    var String name
    var Float salary
end
```

The initialization of a field should be made in the top-level statement list of the method. The current Cyan compiler is not smart enough to deduce `category` is initialized in the following example. Initialize a local variable inside the `if` statement and assign it to `category`.

```
object Person
    func init: String name age: Int age {
        self.name = name;
        if age >= 18 {
            category = "adult"
        }
        else {
            category = "minor"
        }
    }
    String name
    String category
end
```

A field can be initialized in its declaration with an expression. But this expression should be a "safe expression" (SE), defined recursively as:

(i) a basic type (such as `Int`, `Nil`, or `String`) is a SE;

(ii) a value of a basic type (such as `0` or `"Hello"`) is a SE;

(iii) an unary message to a literal value of a basic type (such as `-0` or `+3.14`) is a SE;

(iv) a literal array, a literal map, or a literal tuple whose elements are SE is a SE;

(v) an object creation of a prototype if the arguments used are SE. For example,

```
Array<Int>(5)
```

or

```
Array<String> new: Int new
```

is a SE. Unlike the `initShared` methods, there is no resctriction on the package of the prototype, it can be anyone.

If this is not required, a variable could be initialized with a call to a method of the same prototype that accesses a non-initialized variable:

```
object Test
    Int first = self next
    var Int nextValue = 9
    func next -> Int {
        ++nextValue;
        return nextValue
    }
end
```

In this example, the intention was to set `first` to `10` but at runtime, when `next` is called, variable `nextValue` has not been initialized and `next` returns a non-initialized variable `nextValue`.

(b) suppose `S` is the superprototype of a prototype `P`. If `S` defines a method `init:` but not a method `init`, then every `init` or `init:` method of `P` should call a method `init:` of `S`. A method `init` or `init:` of `P` may not have a super call to a method `init` or `init:` of `S` if `S` has a method `init` (the compiler will insert a call to `init` of `S`). This method `init` of `S` may have been added by the compiler;

(c) inside an `init` or `init:` method, fields can only be used in expressions after they have been initialized;

(d) `init` and `init:` methods cannot use `self` anywhere except in two situations:

(i) when the prototype is *final* and after all prototype fields have been initialized;

(ii) the method to be called is annotated with

```
@accessOnlySharedFields
```

The method that would be called need not to be `final`. These annotated methods can only access shared fields and `self` cannot be leaked. The overridden subprototype method of a superprototype method annotated with

```
@accessOnlySharedFields
```

should also be annotated with this same annotation.

There is an order of initialization of field of a prototype. When an object is created with an `init` or `init:` method, first the fields initialized in their declarations are set (in textual order). Then the statements of method `init` or `init:` are run. The example below shows the order of initialization. `one` is initialized before `two` and so on in an expression "`Order new`".

```
object Order
    func init {
        three = 3
    }
    Int three
    Int one = 1;
    func print {
        (one + two + three) println
    }
    Int two = 2;
end
```

If a prototype does not declare an `init` or `init:` method, the compiler will supply one if every field is initialized in its declaration and the superprototype (if any) has an `init` method. The method added by the compiler to the prototype is

```
func init {
    super init
}
```

Of course, considering that there is a superprototype.

If there is any field of a prototype that is not initialized in its declaration, then the prototype should declare at least one `init` or `init:` method. If the superprototype of a prototype `P` does not define an `init` method but defines a method `init:`, then `P` should declare an `init` or `init:` method. The compiler could not create a method

```
func init {
    super init: args
}
```

because it would not know which parameters `args` to pass in the call `super init: args`.

Metaobject `init` automatically create an `init` method that initializes fields. Consider a prototype `Proto` that declares fields `p1`, `p2`, ..., `pn` of types `T1`, `T2`, ..., `Tn`. Then a metaobject annotation

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

can be put before the declaration of `Proto`. When the compiler finds this metaobject annotation, it will add the following method to the prototype

```
func init: (T1 p1), (T2 p2), ... (Tn pn) {
    self.p1 = p1;
    ...
    self.pn = pn;
}
```

So, a prototype

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

end
```

can be used as

```
var p = Person new: "Carol", 1;
p name println;
```

There are abbreviations 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);
```

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).

```
open
object Tree
end

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

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

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

4.4 Limitations on the Use of Prototypes as Objects

Prototypes in Cyan are objects but with restrictions. Unlike every other prototype-based language, not every method of a prototype can be called and it is not always legal to assign it to a variable (this includes parameter passing). But why? Because a prototype is an object whose fields may not have been initialized. Let us see an example.

```
object Person

    func init: String name, Int age {
        self.name = name;
        self.age = age
    }
    @property
    let String name

    @property
    let Int age

end
```

Annotation `property` creates get methods for `name` and `age`. If methods `getName` or `getAge` of prototype `Person` is called before any other method, there will be an error: a field will be accessed before it has been initialized.

```
    // access non-initialized field 'name'
Person getName println;
var Any any = Person;
    // access non-initialized fields 'name' and 'age'
var personCopy = any clone;
    // method 'call:' may access non-initialized fields
    // 'name' and 'age'
otherObject call: Person;
    // ok!
var protoName = Person prototypeName;
if any isA: Person {
    "oh!!, a person!" println;
}
```

To prevent such runtime errors, the language has some rules regarding the use of prototypes. If a prototype has an `init` method, it can be used as a regular object. There are no restrictions in its use. The prototype itself is created using the `new` method of itself. So all of its fields are properly initialized.

If a prototype does not have an `init` method, it:

(a) cannot receive messages unless the corresponding method is

  (a) `new` or `new:`;

  (b) a `final` method declared in prototype `Any` with annotation `canBeCalledOnPrototypes`:

```
@annot(canBeCalledOnPrototypes)
final
func prototypeName -> String { ... }
```

(b) can be used in an expression if:

  (a) it is `Nil`;

  (b) it is an argument to method `isA:` or `notIsA:`

```
if any isA: Person { ... }
if any notIsA: Cicle { ... }
```

These restrictions aim to prevent prototype fields from being accessed before they are initialized.

A prototype, as an object, is initialized by its method `init` (if there is one). Therefore, during the execution of its `init` method, references to the prototype will result in a runtime error.

```
object FailInInit
    func init {
        message = "prototype refer to itself";
        FailInInit println;
    }
    var String message;
end
```

The use of `FailInInit` inside `init` will cause a `NullPointerException` and then an exception

```
java.lang.ExceptionInInitializerError
```

This is because the prototype name is translated into a static field called `prototype` of a class `_FailInInit` that represents the prototype. This field is in process of being initialized when it is used inside `init`. In Java, what happens is:

```
prototype = new FailInInit(); // call 'init'
```

Before this assignment, `prototype` is `null` (in Java). Inside `init`,

```
FailInInit println;
```

is translated into the Java code

```
_FailInInit.prototype._println();
```

`prototype` is `null` because it is being initialized.

Cyan prohibits references to a prototype inside its `init` method. However, this does not solve the problem. The prototype may be indirectly referenced by methods called inside method `init`.

4.5 Shared Variables and Method `initShared`

A prototype may declare a field as `shared`, as `today` in

```
object Date
    ... // methods

      // shared
    @property shared var Date today

    @property Int day
    @property Int month
    @property Int year
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. They are similar to "class variables" of some languages and "static" variables of C++/Java/C#. Every shared variable of a prototype should be initialized in its declaration or in a special method called `initShared`.

There are several restrictions on an `initShared` method. It

(a) should be declared with no return type (it cannot be `Nil`);

(b) should be private (this may change in the future);

(c) should not be preceded by keyword `override` (it is private);

(d) should not be abstract or final (it is private);

(e) should not be indexing methods (See Section 4.10);

(f) cannot be declared in interfaces;

(g) cannot initialize non-shared fields.

A *Restricted Safe Expression*, RSE, is recursively defined as

(i) a basic type (such as `Int`, `Nil`, or `String`) is a RSE;

(ii) a literal value of a basic type (such as `0` or `"Hello"`) is a RSE;

(iii) an unary message to a literal value of a basic type (such as `-0` or `+3.14`) is a RSE;

(iv) a literal array, a literal map, or a literal tuple whose elements are RSE is a RSE;

(v) an object creation of a prototype of package `cyan.lang` is a RSE if the arguments used are RSE. For example,

```
Array<Int>(5)
```

or

```
Array<String> new: Int new
```

is a RSE.

A RSE can be assigned to a shared field in its declaration. There should be an `initShared` method that initializes all shared fields not initialized in their declarations. Method `initShared` should have only assignments to shared variables and these should be initialized with *safe expressions* as defined for `initShared` methods, which are called SE'.

This prevents that a shared variable or a field of another prototype be accessed before it has been initialized. Of course, these restrictions will be relaxed as soon as possible. They prevent very common patterns such as to create an object and assign it to a shared variable — see example below.

```
object SolarSystem
    ...
    private func initShared {
          // compile-time error in these two lines
        earth = Planet new: "earth";
        saturn = Planet new: "saturn"
    }
    shared Planet earth, saturn
    ...
end
```

This prototype should use an union type and there should be a method, called before `SolarSystem` is used, that initializes the shared variables `earth` and `saturn`.

```
object SolarSystem
    ...
    private func initShared {
        earth = Nil;
        saturn = Nil
    }

    func completeinitShared {
        earth = Planet new: "earth";
        saturn = Planet new: "saturn"
    }

    shared Nil|Planet earth, saturn
```

```
    ...
end
```

Unfortunately, now every use of `earth` or `saturn` should test if the variable is `Nil`. We expect that this will change in a near future. The compiler will build a graph of package and prototype dependences and produce initializing code, to be called in the beginning of the program execution, that respects the dependence order among the packages and prototypes.

Another example of a correct use of `initShared` is given below.

```
object NameServer
    ...
    private func initShared {
        varName = ""
    }
    shared Int nextVarNumber = 0;
    shared String varName
end
```

4.6 Shared Methods

A method can be declared as `shared` meaning that it can only access shared fields. A shared method:

(a) should not be preceded by keyword `override` or `overload`;

(b) should not be abstract or final (it does not make sense);

(c) cannot be declared in interfaces;

(d) cannot use non-shared fields;

(e) should have a name different from any other method or field of the same prototype;

(f) should have a name different from any non-private method inherited from superprototypes.

An example of declaration follows

```
object MyMath
    shared
    func getMax -> Int = 12;
    shared
    func fatorial: Int n -> Int {
        if n == 0 { return 1 }
        else {
          return n*(fatorial: n-1)
        }
    }
    ...
end
```

The receiver of a message passing whose corresponding method is shared should be the prototype in which the method is declared.

```
n = 2* MyMath getMax;
(MyMath fatorial: 10) println;
```

or none. In the last case, the receiver is not specified:

```
n = 2*getMax;
(fatorial: 10) println;
```

There cannot be a shared and a non-shared method in a prototype with the same name. Shared methods should also have different names

4.7 Keyword Methods and Selectors

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

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

Command `return` returns the method value. The execution of the function is ended by the `return` command.

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

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

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

```
{ total = total - amount }
```

and

```
{ ret = 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

```
@init(w, h, x, y)
object Rectangle
    func width: Int w height: Int h {
        self.w = w;
        self.h = h;
    }
    func set: (Int x, Int y) { self.x = x; self.y = y; }
    func getX -> Int = x;
    func 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 *keyword*. So `width:` and `height:` are the keywords of the first method of `Rectangle`. Sometimes we will use "method with multiple keywords" instead of "keyword method". The concatenation of the method keywords is called the method selector. Then `width:height:` is a method selector.

The signature of a method is composed by its keywords, parameter types, and return value type. Then the signature of method "`width:height:`" is

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

The return type is `Nil` and it may not appear. The signature of `getX` is

```
getX -> Int
```

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

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

And so is the declaration

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

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

```
object Rectangle
   func width: (Int w) height: (Int h) {
      self.w = w;
      self.h = h;
   }
   func 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 Operator Methods

Operators can be method names in Cyan. However, there are limitations: binary operators should take one parameter, unary operators should not take parameters. Operators `+` and `-` can be both binary and unary.

```
object Complex
    func init: Double re, Double im {
        self.re = re;
        self.im = im
    }
      // unary -
    func - -> Complex =
       Complex(-re, -im);

    func + (Complex other) -> Complex =
       Complex(re + other getRe, im + other getIm);
```

```
    // binary -
    func - (Complex other) -> Complex =
       Complex(re - other getRe, im - other getIm);

    func * (Complex other) -> Complex =
       Complex( re*(other getRe) - im*(other getIm),
                re*(other getIm) + im*(other getRe) );

    override
    func asString -> String = "($re, $im)";

    func getRe -> Double = re;
    func getIm -> Double = im;
    Double re;
    Double im;
end
```

Operators are frequently used for goals not linked to the expected meaning of them. For example, to use - for removing a number from a list. The expected meaning would be to subtract something from the number. To prevent such kind of misuse, Cyan limits the use of the following operators:

`+ - * / << >> >.>> %`

Prototypes that declare methods with these operators should be read-only. That is, all fields should be declared with `let` or without the keyword `var`. The goal of this is to limit the use of these operators to mathematical structures, which are usually read-only. This is controversial, we know that.

4.9 On Names and Scope

Unary methods and fields of an object should have different names. Fields and shared variables can have names equal to local variables (which includes parameters):

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

An object can declare methods "`value`" and "`value:`" as in the following example:

```
object Store
    var Int _value = 0;
    func value -> Int = _value;
    func value: Int newValue {
        self._value = newValue
    }
end
```

And a method name can be a language keyword followed by ":"

```
    func while: Boolean expr { ... }
```

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;
    func name -> String = _name;
       // parameter with the same name as field
    func name: _name String { self._name = _name }
    func lifespan -> String = _lifespan;
    func lifespan: Int _lifespan { self._lifespan = _lifespan }
end
```

4.10 Operator []

It is possible to define operator `[]` for indexing:

```
object Table
     func [] at: Int index -> String {
         return anArray[index]
     }
     func [] 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 keyword should have only one parameter. When `t[expr]` appears inside an expression, it is considered the same as `(t at: expr)`. When `t[expr]` appears in the left-hand side of an assignment,

```
t[expr] = rightExpr
```

this is considered as

```
t at: (expr) put: (rightExpr)
```

One or both methods can be declared. But when both are declared, the type of keyword `at:` should be the same. The allowed signatures of these methods are:

```
at: T -> U
at: T put: W -> Nil
at: T put: W -> U
```

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

Indexing methods cannot be abstract and they should be public.

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 fields defined in `Person`. `Student` is called a sub-object or subprototype. `Person` is the superprototype or superprototype. The declaration of a superprototype should be preceded by identifier "`open`":

```
package main

open
object Person
    // elided
end
```

Note that "`open`" is not a keyword. To restrict the subprototypes to the package of the prototype one can use "`open(package)`" as in

```
package main

open(package)
object Person
    // elided
end
```

Every field of the subprototype should have a name different from the names of the public and protected methods of the superprototype (including the inherited ones) and different from the names of the methods and other fields of the subprototype. Since the name of a non-unary method includes the ":", there may be field `iv` and method `iv:`.

A public method of a prototype is visible anywhere the prototype is. A protected method is visible in the prototype and its subprototypes. A protected method is declared with the syntax

```
    protected
    func getList -> List<Int> { ... }
```

A `package` method is visible only in the package of the prototype in which the method is.

```
package main
open
object Employee
    package
    func getSalary -> Double { ... }
    // elided
end
 // other source file:

package company
```

```
object Manager extends Employee

    package
    func getSalary -> Double { ... }
    // elided
end
```

In this case, prototype `Manager`, that inherits from `Employee`, is trying to override method `getSalary`. This results in a compile-time error: the first method is visible only in package `main` and the `getSalary` of `Manager` is visible only in package `company`. Therefore there is a compilation error.

A public, package, or protected method of a subprototype may have the same keywords, parameter types and number of parameters for each keyword than a method of the superprototype.

```
open
object MovieList
    ...
    func add: String movieName
         director: String director
         year: Int year {
        ...
    }
    func search: String movieName
         year: Int -> Movie {
         ...
    }
end

object LoveMovieList extends MovieList
    ...
    override
    func add: String movieName
         director: String director
         year: Int year {
        ...
    }
    override
    func search: String movieName
         year: Int -> LoveMovie {
         ...
    }
end
```

Here method

```
    func add: String movieName
         director: String director
         year: Int year
```

of `MovieList` is redefined in `LoveMovieList`, its subprototype. Each keyword takes the same parameters, which means the same number of parameters and the same types. The subprototype method is preceded by the Cyan keyword `override`. The return value type of the subprototype method that was overridden may

be a sub-type of the return value type of the superprototype method. For example, method `search:year:` of `LoveMovieList` overrides a superprototype method and returns a `LovieMovie` that is a subprototype (then a sub-type) of `Movie`, which is the return type of the method of the superprototype. Assume that `LovieMovie` is a subprototype of `Movie`. The sub-type relationship is defined in Section 4.17.

A method can only be overridden by a method with the same visibility. That is, a public method can only override a public method. And a protected method can only override a protected method. A public or protected method of a subprototype with the same name as a private method of the superprototype is not overridden (of course!).

The Cyan keyword `override` should follow the qualifier `public` or `protected` if any of these are present. Currently this order is enforced.

```
open
object Person
    func init: String name, Int age {
        self.name = name;
        self.age = age;
    }
    override
    func print {
         Out println: "name: $name (age $age)"
    }
    @property String name
    @property Int age
end

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

There is a keyword called `super` used to call methods of the superprototype. 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 `initShared` are never inherited. However, `init` or `init:` methods of a subprototype may call `init` or `init:` methods of the superprototype using `super`:.

Keyword `override` should not be used 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 `init:` and `new:` are not inherited, there will be compile-time errors in method `nonsense`. See Section 4.3 for the many restrictions on `init`, `init:`, and `initShared` methods.

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

```
public final object Int
    ...
end
```

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

A method declared as "`final`" cannot be redefined in subprototypes:

```
public object Car
    final func name: String newName { _name = newName }
    final String func 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 subprototypes. Therefore this is a static call, much faster than a regular call.

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

The table below summarizes the allowed combination among keywords in a method declaration. Keyword `abstract` is explained in Section 4.16.

	public	protected	private	override	abstract	final
public				Y	Y	Y
package				Y	Y	Y
protected				Y	Y	Y
private						
override	Y	Y			Y	Y
abstract	Y	Y		Y		
final	Y	Y		Y		

Table 4.1: Keyword combination in method declaration

The order in which the method qualifiers can appear is rigid. It is `public`/`private`/`protected`/`package`, `final`, `override`, `abstract`, and `overload`. Then all the declarations below cause compile-time errors.

```
abstract protected func abs: Int n
override final func getList -> List { ... }
abstract override func parse: String code -> AST { .. }
```

Future versions of the language may demand the these keywords appear in alphabetical order: `abstract`, `final`, `overload`, `override`, `public`/`private`/`protected`. Or without any order.

In order to prevent some traps explained by Huang, Yang, and Chan [CYH04], a prototype that defines at least one method with `package` visibility can only be inherited by prototypes in the same package.

4.12 Downcasting with type-case and cast statements

Downcasting is to change the type of an object of a supertype to the type of a subtype. Usually this is made by a construction that does the casting and throws an exception if it is not possible. Cyan does not support constructions that does exactly that. But it does support command `type-case` for safe downcasting without exception signalling. Its syntax is

```
type expression
    case Type1 [ varName1 ] {
        statement list
    }
    // any other number of case clauses
    [ else {
        statement list
      }
    ]
```

The `varName1` is optional and so is the `else` part. There may be one or more `case` clauses. A `';'` after the last `case` clause or `else` clause is optional. At runtime, if `expression` has type `Type1`, it is cast to `Type1` and assigned to `varName1`. Then the statements of the `case` are executed. If `expression` has not type `Type1`, the following `case` clauses are tested, in textual order.

As an example, the following code will print `1 2 true B`

```
var Dyn d = 0;
for elem in [ d, "Hi", 5.0, 'A' ] {
    type elem
        case Char ch {
            (ch succ ++ " ") print
        }
        case String s {
            (s size ++ " ") print
        }
        case Int n {
            (n + 1 ++ " ") print
        }
        case Double n {
            ((n equal: 5.000001) ++ " ") print
        }
}
```

The type of a `case` clause cannot be subtype of the type of a previous `case` clause. Then the two `type-case` of the next example produce errors. Assume that `Circle` inherits from `Shape`.

```
    var Any elem = 0;
    type elem
        case Any {
        }
        case Int n { // error here
            ("An int equal to $n") print
        }
    var Shape shape = Circle(100.0, 10.0);
    type shape
        case Shape s {
        }
        case Circle { // error here
            "This will never be printed" println
        }
```

An `else` clause may follow the last `case` clause with the obvious meaning. A `type-case` statement should have at least one `case` clause.

Statement `cast` is a special downcast statement whose syntax is:

```
cast [ Type1 ] Id1 = Expr1, ..., [ Typen ] Idn = Exprn {
    statementListTrue
}
[
else {
    statementListFalse
}
]
```

If the type of `Expri` is `T|Nil` or `Nil|T`, with `T` different from `Nil`, `Typei` is optional and assumed `T`. If the cast succeeds, `statementListTrue` is executed. Otherwise, `statementListFalse` is executed. Exemple:

```
var Int|Nil intNil = 0;
var Any any = "ok";
cast elem = intNil, String s = any {
    assert elem == 0 && s == "ok";
}
else {
    assert false;
}
cast String s = any {
    "s = $s" println
}
```

`elem` is assumed to have type `Int`.

4.13 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
```

```
    func print
end
```

The `public` keyword is not necessary since all signatures are public. `func` 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
    func printObj
end

@init
open
object Person
    @property String name
    @property Int age
end

object Worker extends Person implements Printable
    @property String company
    override
    func printObj {
        Out println: "name: " ++ name ++ " company: " ++ company
    }
    ... // elided
end
```

Here prototype `Worker` should implement method `printObj` because this prototype is *implementing* interface `Printable` that defines a `printObj` method. Otherwise the compiler would sign an error. Note that the `printObj` method of `Worker` is preceded by `override`. This is demanded by the compiler.

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
    func setColor: Int newColor
    func colorPrint
end
```

Therefore Cyan supports a limited form of multiple inheritance. An interface that does not explicitly inherits from any other in fact inherits from prototype `Any` as any other prototype.

An interface is a regular prototype for many uses. It can receive messages for example. But there will be a runtime error if the message corresponds to a method declared in the interface. There will not be a runtime error if the method is inherited from `Any`.

The method signatures declared in an interfaces are transformed into public methods by the compiler. These methods throw exception `ExceptionCannotCallInterfaceMethod`:

```
   // interface ColorPrintable as a prototype
object ColorPrintable extends Printable, Savable
    func setColor: Int newColor {
        throw: ExceptionCannotCallInterfaceMethod("ColorPrintable::setColor");
    }
    func colorPrint {
        throw: ExceptionCannotCallInterfaceMethod("ColorPrintable::colorPrint");
    }
end
```

Interfaces are then objects with full rights: they be assigned to variables, passed as parameters, and receive messages. However, an interface declaration cannot be preceded by "`open`" or "`open(package)`". Interfaces are already open. To restrict the use to a package one can declare it with visibility `package`. In future versions of Cyan, this is not working yet.

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 `init` or `init:` methods. No object will ever be created from them. But the interface itself may receive messages and it may be cloned.

(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 4.15).

(e) If interface `Inter` declares a method signature and prototype `P` implements interface `Inter`, `P` should declare a method with that same signature except that the return value type can be a subtype of the return value type of the method signature of the interface. It is possible that `P` inherits the method. In this case `P` may not declare the method. But whenever `P` declares a method of a implemented interface, it should be preceded by keyword "`override`".

(f) An interface cannot declare a true overloaded method (Section 4.14). If prototype `P` implements interface `Inter` that defines a method signature `ms`, `P` should define a method `m` with that signature. `m` cannot belong to an overloaded method.

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);
```

4.14 Method Overloading

There may be methods with the same keywords but with different number of parameters. For example, one can declare

```
object MyPanel
    func print: Circle c, Int x, Int y -> Circle { ... }
    func print: String s, String format -> Boolean { ... }
    func print: { ... }
    func print: Int n -> Int { ... }
end
```

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

```
MyPanel print: anObj
```

there is no ambiguity on which method should be called. It can only be the last method, which is the only one that takes just one parameter. Since all methods are considered different, they may have different return value types. This is not true method overloading.

One may also use several keywords:

```
object Test
    func at: Int i, Int j put: Int k -> String { }
    func at: Int i put: Int k -> Int { ... }
    func at: String s put: String s, Int k -> Boolean { ... }
    ...
end
```

This object could be used as in

```
var f = Test;
f at: 0, 1 put: 2;
f at: #1 put: "one", 0;
f at: 0 put: 1;
```

We call the "name of a method" the concatenation of all of its keyword names, each one followed by its number of parameters and a white space. The trailing white space should be removed. For example, methods

```
    func key: String aKey
         value: Int aValue -> String
    func name: String first, String last
         age: Int aAge
         salary: aSalary Float -> Worker
```

have names `"key:1 value:1"` and `"name:2 age:1 salary:1"`.

A restricted form of multi-methods is allowed in Cyan. 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. This is called "overloading" in Cyan and the methods involved are called overloaded methods — this is **true** overloading. The compiler generates just one method for all overloaded methods with the same name in a prototype. Then we sometimes say "overloaded method", in the singular.

An overloaded method is composed by one or more regular methods with the same name declared in the same prototype. The first method should be preceded by the Cyan keyword `overload` (more on this soon).

```
open
object MyBlackBoard
      // keyword 'overload' precedes an overloaded method declaration
    overload
    func draw: Square f { ... }
    func draw: Triangle f { ... }
    func draw: Circle f { ... }
    func draw: Shape f { ... }
    private String name
end
```

Then in the example above there is one `draw` overloaded method composed by four methods. In the example below there is another overloaded method `draw` composed by two methods.

```
object MyOtherBlackBoard extends MyBlackBoard
    func draw: Polygon f { ... }
    func draw: Elipse e { ... }
end
```

The first method of an overloaded method in a prototype hierarchy should be prefixed with the keyword "`overload`". By "first" we mean the method that is higher in the hierarchy and that is textually before the others with the same name in its prototype. This first overloaded method should not override another method and all methods that override it are also considered overloaded methods. Overloaded methods can only be `public`. [1] Keyword `overload` should appear before the "`func`" keyword.

The compiler checks all methods of an overloaded method of a prototype at once. The rules below are based on a single method but once no error is detected, all methods of the overloaded method of the prototype are considered correct (with relation to its signature). Before proceding, remember that the name of a method is composed by joining each of its keywords followed by the number of parameters followed by a white space. The last space is discarded. So the name of method

```
func at: Int n with: String s, Long k add: Person p -> Boolean { ... }
```

is

```
"at:1 with:2 add:1"
```

The signature of a method is composed by the keywords and the full name of the parameter types and return value type. Then the signature of the above method is

[1] this may change later.

```
at: Int with: String, Long add: main.Person -> Boolean
```
Prototypes of package `cyan.lang` are not preceded by the package name.

Consider a method `m` not preceded by `overload` whose signature is `ms`. `m` is declared in a prototype `P`. During the semantic analysis, the Cyan compiler does some checkings on `m`. The compiler search for all public and protected methods with the same name in `P` and its superprototypes putting the list of methods found in `msList`. The search includes `m` itself. The compiler searches for all methods with the same name as `m` in all interfaces implemented by `P` putting the list of method signatures found in `interMSList`. Then `interMSList` contains zero or one signatures because interfaces cannot declare or inherit methods with different signatures and the same name. The list of all public and protected methods declared in `P` with the same name as `m` is put in list `pmsList` (so `pmsList` is a subset of `msList` and it does not include inherited methods). The rules for the validity of the declaration of `m` are given below.

(a) If `msList` has just method `m` and `interMSList` is empty, the declaration of `m` is correct. If `interMSList` is not empty, it should have just one method since interfaces cannot declare overloaded methods. If the single method signature of `interMSList` is different from the signature of `m` then the declaration of `m` is incorrect;

(b) Suppose all methods of `msList` are in `P` (`pmsList` is equal to `msList`) and `pmsList` has more than one element. The declaration of the methods of `pmsList` are correct if:

  (i) each two methods of `pmsList` have different signatures;

  (ii) all methods of `pmsList` have the same return value type;

  (iii) the first textually declared method is preceded by keyword `overload`. No other method is preceded by this keyword;

  (iv) if one method of `pmsList` is final, all methods of this list should be final too. If it is not final, no method of the list should be final;

  (v) no method of `pmsList` is protected or abstract;

  (vi) `interMSList` should be empty.

(c) Suppose there is at least one method in `msList` that is in a superprototype and:

  (i) no method of `msList` is preceded by `overload`;

  (ii) there is just one method in `pmsList`. It should be preceded by "`override`";

  (iii) the return value type of `m` is a subtype of the return value type of `m1`, which is the first method with name equal to `m` found in a search starting in the superprototype of `P` and continuing upwards;

  (iv) all methods of `msList` have the same signature except for the return value type. That would mean that each method of `msList` is in a different prototype;

  (v) if the method of `pmsList` is protected, so are all the methods of the list `msList`.

  Then the declaration of the method of `pmsList` is correct even if `interMSList` contains an element.

(d) Suppose there is at least one method in `msList` that is in a superprototype and there are at least two methods of `msList` that have different signatures. That includes the case in which `pmsList` has two methods (since they have the same name, they must have different signatures). The declaration of the methods of `pmsList` are correct if:

  (i) each two methods of `pmsList` have different signatures. Note that `pmsList` may have just one element, `m`, although `msList` should have at least two elements;

(ii) all methods of `pmsList` are preceded by keyword "`override`";

(iii) all methods of `pmsList` have the same return value type;

(iv) let `Q` be the first direct or indirect superprototype of `P` that declares a method with the same name as `m` and `directSMList` be the list of methods of `Q` that have the same name as `m`. All methods of `directSMList` have the same return value type `R`. The return type of all methods of `pmsList` should be subtype of `R`;

(v) let `T` be the superprototype of `P` that declares a method with the same name as `m` and that is higher in the `P` hierarchy. That is, no superprototype of `T` declares a method with the same name as `m`. Then the first textually declared method of `T` should be preceded by keyword `overload`. No other method in the `P` hierarchy should be preceded by this keyword;

(vi) either none or all methods of `pmsList` are final;

(vii) no method of `pmsList` is protected or abstract;

(viii) `interMSList` should be empty.

If a method `m` of a prototype `P` is declared with the `overload` keyword, then:

(a) no method with the same name should have been declared textually before it in the prototype hierarchy. That includes `P` and its superprototypes. That is, `m` should not override a superprototype method;

(b) no interface implemented by `P` should declare a method with the same name as `m`;

(c) no method with the same name in the prototype should be abstract;

(d) if the method is final, all methods with the same name should be final too. If it is not final, no method with the same name can be final;

(e) the return value type of all methods with the same name as `m` in `P` should be the same.

Let us see some incorrect examples:

```
object MyBlackBoard
    func draw: Square f { ... }
    overload // second method, compile-time error
    func draw: Triangle f { ... }
    func draw: Circle f { ... }
    func draw: Shape f { ... }
    private String name
end

open
object A
    func draw: Square f { ... }
end
object B extends A
    func draw: Shape p { ... }
end
```

Method `draw` of `A` should have been declared as `overload`.

```
interface I
    func draw: Shape
end
object A
    func draw: Square f { ... }
end
object B extends A implements I
    func draw: Square p { ... }
    override
    func draw: Shape p { ... }
end
```

`B` should implement "`draw: Shape`" thus becoming `draw` in an overloaded method. Then `A` should have declared this method as an overloaded method.

In a message send, the method found at compile-time may be different from the method called at runtime. This is true regardless the method found at compile-time is an overloaded method or not. If the method found at compile-time is an overloaded method, the runtime search for a method is as usual starting from the runtime prototype of the receiver and proceding to the top of the hierarchy (`Any`). However, there may be, in a prototype, two or more methods that have the name of the method found at compile-time. The compiler tests whether one of these methods, in the textual declaration order, can accepts parameters of the message send. To accept a parameter, each runtime argument should be subtype of the declared parameter of the method. If no method of the prototype can accept the runtime parameters, the search for a method continues at the superprototype.

To make the mechanism clearer, study the example below. Assume that `Grass`, `FishMeat`, and `Plant` are prototypes that inherit from prototype `Food`.

```
open
object Animal
    overload
    func eat: Food food { Out println: "eating food" }
end

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

object Fish extends Animal
    override
    func eat: FishMeat food { Out println: "eating fish meat" }

    override
    func eat: Plant food { Out println: "eating plants" }
end

object Program
    func run {
        var Animal animal;
        var Food 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
```

4.15 `Nil` and `Any`, the superprototype 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 5, `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

    func prototypeName -> String
    func asString -> String
    func asString: (Int ident) -> String
    func print
    func println
    func == (Dyn other) -> Boolean
    func === (Dyn other) -> Boolean
    func != (Dyn other) -> Boolean
    func !== (Dyn other) -> Boolean

end
```

Prototypes that are declared without explicitly extending a superprototype in fact extend an object called `Any`. Therefore `Any` is the superprototype 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);
```

could 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 identation of `n` white spaces. The indentation is made with `defaultIdentNumber` white spaces. This is a shared variable declared in `Any`.

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

```
package cyan.lang

public object Any

    func eq: (Dyn other) -> Boolean
    func neq: (Dyn other) -> Boolean
    func prototype -> Any
    final func prototypeName -> String
    final func prototypeParent -> Any
    final func prototypePackageName -> String
    final func isInterface -> Boolean
    final func isA: (Any proto) -> Boolean
    final func notIsA: (Any proto) -> Boolean
    final func throw: (CyException e) -> Dyn
    func clone -> Any
```

```
    final
    func ++ (Any other) -> String
    func asString -> String = asString: 0;
    func asString: (Int ident) -> String
    func asStringThisOnly: Int ident -> String
    func asStringQuoteIfString -> String
    func == (Dyn other) -> Boolean
    func === (Dyn other) -> Boolean
    func != (Dyn other) -> Boolean
    func !== (Dyn other) -> Boolean
    func isCase: (Any other) -> Boolean
    func assertxx: (Boolean expr)
    func assertxx: Boolean expr, String message
    func print
    func println
    func toAny: Dyn elem -> Any
    final
    func featureList -> Array<Tuple<key, String, value, Any>>
    final
    func featureList: (String slotName) -> Array<Tuple<key, String, value, Any>>
    final
    func slotFeatureList -> Array<Tuple<slotName, String, key, String, value, Any>>
    final
    func annotList -> Array<Any>
    final
    func annotList: (String slotName) -> Array<Any>
    func doesNotUnderstand: (String methodName, Array<Array<Dyn>> args) -> Dyn
    final
    func functionForMethod: String signature -> Any
    final func functionForMethodWithSelf: String signature -> Any
    func hashCode -> Int

end
```

Method `prototype` returns the prototype of an object. It is added to the compiler to every prototype — the cannot be user defined. Its return value type is the prototype. Then in a prototype `Student` method `prototype` is

```
    func prototype -> Student
```

Methods `new` and `new:` are added to the prototype if it 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. This method is not final but it can only be overridden in the basic subprototypes, including `String` (a metaobject checks that). For the basic types, excluding `Nil`, method `eq:` compares the contents of the receiver object with the parameter. It is equivalent to `==`.

Method `prototypeParent` returns the parent prototype of the receiver determined at runtime.

```
var Person p = Person("fulano", 32);
assert p parent == Person prototypeParent;
```

Method `prototypeParent` of an interface always return `Any` even if the interface inherits from several other ones. When the receiver of message `prototypeParent` is `Any`, the value returned is `Any`. A future change is to make `prototypeParent` return `Any|Nil`. Then `prototypeParent` on `Any` would return `Nil`.

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. Assuming that `Circle` inherits from `Elipse` that inherits from `Any`, we have

```
var Elipse e = Elipse(...); // elided
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;
```

Method `notIsA:` return the negation of `isA:`.

Method `throw:` throws the exception that is the parameter. See more on Chapter 11. `hashCode` returns an integer that is the hash code of the receiver object (this needs to be better defined).

The compiler adds method `clone` to every prototype that does not define this method. If a prototype defines `clone`, the compiler checks whether it has the correct method signature. `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 will be used in a `switch` statement that will be added to the language.

Method `assertxx:` takes a boolean expression as parameter and throws exception `ExceptionAssert` if `expr` is false. Methods `print` and `println` print information on the receives using methods `print:` and `println:` of prototype `Out`.

A feature is a metadata composed by a name and a value that can be attached to a prototype, field, shared variable, or method. `Any` defines methods for retrieving features from the prototype and its methods and fields.

```
package main
  // feature attached to a prototype
@feature(xmlRoot, "A_person")
@init(name, age, city)
object Person
    func getName -> String = name;
    func getAge -> Int = age;
    func getCity -> City = city;
      // feature attached to a field
    @feature(xmlElement, "PersonName")
    String name
    @feature(xmlElement, "PersonAge")
    Int age
    @feature(xmlElement, "PersonCity")
    City city


      // feature attached to a method
    @feature(aName, "a complex name")
    func at: Int n with: String s { }
```

```
    @feature(over1, "at(Int)")
    overload
    func at: Int n { }
    @feature(over2, "at(String)")
    func at: String s { }
end
```

Method `featureList` returns an array with all features of the prototype. Method

```
func featureList: (String slotName) -> Array<Tuple<key, String, value, Any>>
```

returns the feature list of slot `slotName`, which may be the name of a field, method, or shared variable declared in the prototype. The method name includes the types of the parameters but not the return value type. This table gives several method names

method	method name for featureList:
`+ -> Int`	`"+"`
`+ Int -> Int`	`"+ Int"`
`at:  Int -> Int`	`"at: Int"`
`at: cyan.lang.Int, String with: Person p -> String`	`"at: cyan.lang.Int, String with: Person p"`

Unfortunately, the parameter `slotName` should be exactly as in the declaration. Which is not well specified. For example, a method

```
add: Table t
```

has name `"add: Table"` without any specification on the package of `Table`. And if a method is declared as

```
add: cyan.lang.Int elem
```

its name should be considered
`"add: cyan.lang.Int"`
and not the simpler
`"add: Int"`
It is not necessary to say that this is a flaw in the language that will be corrected in a near future.

Method `featureList` of `Any` returns a list of features of the prototype of the receiver:

```
for t in any featureList {
    Out println: "key = ", t key, " value = ", t value;
}
```

Method `slotFeatureList` returns a list of all features of all slots of the prototype of the receiver.

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

```
func annotList: (String slotName) -> Array<Any>
```

returns the annotation list of slot `slotName`. For methods, `slotName` is the concatenation of the keywords

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 keywords. The name of message

```
ht at: i put: obj with: #first
```

is "at:put:with:". The name is also called the "selector" of the message.

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

```
var Int n = 0;
var Dyn d = 0;
n ?push: 10; // runtime error
// runtime error if the previous line is commented
d at: 0 put: 10;
```

The second argument to method `doesNotUnderstand:` is an array whose elements are arrays, each one grouping the parameters of each keyword. 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

```
[
    [ first0 ],
    [ first1, name, age ],
    [ first2 ]
]
```

in which

```
var Dyn first0 = "%d%s%i";
var Dyn first1 = n;
var Dyn first2 = output;
```

The last assignments assure that the array is of type `Array<Dyn>`.

There are several missing methods in `Any` related to reflective introspection. These reflective introspection methods will be added to `Any` in a near future.

4.16 Abstract Prototypes

Abstract prototypes in Cyan are the counterpart of abstract classes of class-based object-oriented languages. The syntax for declaring an abstract prototype is

```
package myShapes

abstract object Shape
    func init: Int newColor {
       color: newColor;
       shapeColor = 0;
    }
    abstract func draw
    func color -> Int = shapeColor;
    func color: Int newColor { shapeColor = newColor }
    Int shapeColor
end
```

An abstract prototype is considered "`open`". It is not necessary to put this word before the prototype declaration. However, to restrict the subprototypes to be package of the abstract prototype, one may use "`open(package)`".

An abstract method is declared by putting keyword "`abstract`" before "`func`" and it can only be declared in an abstract prototype, which may also have non-abstract methods and fields. A subprototype of an abstract prototype may be declared abstract or not. However, if it does not define the inherited abstract methods, it must be declared as abstract.

Overloaded methods cannot be abstract. Future versions of Cyan may allow that.

It is a compile-time error to send a message `new` or `new:` to an abstract prototype. Since these methods can only be called through a prototype, no object will ever be created from an abstract prototype. Methods `new` and `new:` cannot be called even using reflection:

```
var shape = Shape;
  // prints 'Any', the superprototype of Shape
(shape ?new) prototypeName println;

   // compile-time error
(Shape ?new) prototypeName println;
```

It is not a compile-time error to call other methods of an abstract prototype:

```
Shape prototypeName println; // ok, prints "Shape"
Shape draw; // runtime error
let Any any = Shape;
any prototypeName println; // ok, prints "Shape"
any draw; // runtime error if previous error is commented
let Dyn dyn = Shape;
dyn prototypeName println; // ok, prints "Shape"
dyn draw; // runtime error if previous errors are commented
```

If the method called is abstract, as `draw`, there will be a runtime error. An exception will be thrown because the compiler adds a body to every abstract method. This body throws an exception `ExceptionCannotCallAbstractMet`

```
    func draw {
        throw: ExceptionCannotCallAbstractMethod("myShapes.Shape::draw")
    }
```

`init` and `init:` methods may be declared — they may be called by subprototypes.

An abstract prototype can inherit from a non-abstract prototype. However, an abstract method cannot override an inherited method.

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.

Suppose an abstract prototype `Solid` inherits from abstract prototype `Shape`. If `Solid` declares an abstract method `draw`, this should be preceded by keyword `abstract`:

```
abstract object Solid extends Shape
    abstract override
    func draw
end
```

4.17 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
open
object A implements I end
open
object B extends A end
open
object C extends B end
open
object D extends C implements J end
open
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;
var 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 expression. In the example

```
var Int x;
var typeof(x) y;
```

`x` and `y` have both the `Int` type. `typeof` works even with expressions, which are not evaluated at runtime:

```
var typeof( 1 + 2 ) result;
assert result prototypeName == "Int";
```

`result` will have the type `Int`.

4.18 Union Types

The language supports type unions that is the union of two or more types.

```
var String|Int si;
si = 0;
assert si == 0;
si = "zero";
assert si == "zero";
```

An *union type* is a *virtual type*; that is, a type for which there is no prototype. There is a restriction in relation to an union $\texttt{T}_1$, $\texttt{T}_2$, ... $\texttt{T}_n$: if $j \geqslant i$, then $\texttt{T}_i$ cannot be supertype of $\texttt{T}_j$. Hence, `Any|Int` is illegal because `Any` is supertype of `Int`. A special case of the second rule is that type cannot be repeated: `Int|Int` is illegal.

The type checking rules for unions are those expected:

(a) `A` is a supertype of $\texttt{T}_1\texttt{|}\texttt{T}_2\texttt{|}$ ... $\texttt{|T}_n$ if `A` is a supertype of every $\texttt{T}_i$ for $1 \leqslant i \leqslant n$;

(b) $\texttt{T}_1\texttt{|}\texttt{T}_2\texttt{|}$ ... $\texttt{|T}_n$ is a supertype of `A` if there is a $\texttt{T}_j$ that is supertype of `A`;

(c) $\texttt{T}_1\texttt{|}\texttt{T}_2\texttt{|}$ ... $\texttt{|T}_n$ is a supertype of $\texttt{U}_1\texttt{|}\texttt{U}_2\texttt{|}$ ... $\texttt{|U}_m$ if, for each $\texttt{U}_j$, $1 \leqslant j \leqslant m$, there is a $\texttt{T}_i$, $1 \leqslant i \leqslant n$, such that $\texttt{T}_i$ is a supertype of $\texttt{U}_j$.

Assume `B` and `C` inherits from `A`, `D` inherits from `C`. All prototypes declare an `init` method. The example shows some legal assignments.

```
var Any any = Any;
var A a = A();
var B b = B();
var C c = C();
var D d = D();
var B|C bc = b;
a = bc;
// b = bc; // error
any = bc;
var D|C dc = c;
// var C|D cd; // D should come first
c = dc;
a = dc;
bc = d;
var B|D bd = b;
bc = bd;
var Any|Nil anil = bd;
anil = bc;
anil = Nil;
var Dyn dyn = anil; // or anything
var B|Dyn bdyn = anil; // or anything
```

An expression whose type is an union can only receive messages `==` and `!=`. If other methods need to be called, use the `type-case` statement. In this example, assume `Shape` is a superprototype of `Circle`.

```
var Circle|Shape join;
...
```

```
    type join
        case Circle aCircle {
            ("join is a Circle with radius " ++ aCircle radius) println
        }
        case Shape aShape {
            "join is a Shape" println
        }
```

The `type-case` rules apply: the type of a `case` clause cannot be a subtype of the type of a previous clause and each `case` type should be a subtype of the expression type.

Unions are an alternative to method overloading. Instead of declaring several methods, one for each parameter type, there may be a single method:

```
object Company
    ...

    func apply: Manager|Director futureEmployee { ... }

    Array<Manager | Director> employeeList;

end
```

4.19 Tagged Unions

The generic prototype `Union` of package `cyan.lang` is used for *tagged unions*. An instantiation of this prototype should have the format

`Union<`$\texttt{id}_1$`, `$\texttt{T}_1$`, ... `$\texttt{id}_n$`, `$\texttt{T}_n$`>`

in which $\texttt{id}_i$ is a lower case identifier and $\texttt{T}_i$ is any Cyan type but `Nil`. Each object keeps just one object at a time, tagged with the identifier. For each identifier $\texttt{id}_i$ associated with type $\texttt{T}_i$, the union declares a method:

`func `$\texttt{id}_i$`: `$\texttt{T}_i$` elem -> Union< `$\texttt{id}_1$`, `$\texttt{T}_1$`, ... `$\texttt{id}_n$`, `$\texttt{T}_n$`>`

$\texttt{id}_i$`:` is used to store an element to the union object and associate it with $\texttt{id}_i$. `self` is the value returned by this method. The stored object can only be retrieved using statement `type-case`.

```
   // create the union object
var myUnion = Union<number, Int, numberStr, String> new;

// myUnion = "12"; // compile-time error if uncommented
  // associate 'number' to 12
myUnion number: 12; // ok
myUnion numberStr: "12"; // ok

type myUnion
    case Int number {
        (1 + 2*number) println
    }
    case String numberStr {
        if numberStr startsWith: "1" { numberStr println }
    }
```

The `type-case` statement should have one `case` clause for each tag of the union, in the order the tags appear in the `Union` prototype. The variable name of each `case` clause should be the tag. Since the identifiers identify the value kept by the tagged union object, types can be repeated.

```
var enery = Union<wattHour, Double, calorie, Double, joule, Double> new;

energy wattHour: 314.15;
type energy
    case Double wattHour {
       "watt-hours: $wattHour" println
    }
    case Double calorie {
        "I had $calorie calories" println
    }
    case Double joule {
        "I had $joule joules" println
    }
```

The types of union elements can be other tagged or non-tagged unions.

```
var chother = Union<ch, Char, other, String>();
chother other: "aaa";
var agename =
    Union<age, Int, name, Union<ch, Char, other, String>>();
agename name: chother;
type agename
    case Int age { printexpr age; }
    case Union<ch, Char, other, String> name {
        type name
            case Char ch { printexpr ch; }
            case String other { printexpr other; }
    }

var p = Union<age, Int, name, Char|String>();
p name: 'a';
type p
    case Int age { printexpr age; }
    case Char|String name {
        type name
            case Char ch { printexpr ch; }
            case String other { printexpr other; }
    }
```

4.20 Interoperability with Java

Not surprisingly, Cyan code call use Java classes. A Java class/interface can be the type of a field, local variable, parameter, or return value of a method. A Java package can be imported, a Java class can be imported. Java objects can be created and messages can be sent to them using Cyan syntax. Objects of the wrapper classes of Java are automatically converted to the corresponding Cyan prototypes and

vice-versa (`Byte`, `Integer`, `Short`, `Long`, `Character`, `Boolean`, `Float`, `Double`). Idem for values of the basic Java types (`byte`, `int`, `short`, `long`, `char`, `boolean`, `float`, `double`). Assignments from anything to `java.lang.Object` are allowed.

```
package javaInter.main

  // import Java package
import java.util
import java.lang
  // import Java class
import java.io.File

object JavaTest1
     // field whose type is a Java class
    var java.lang.StringBuffer sbWorker

       // return value is a Java class
    func retSB -> StringBuffer {
         // Cyan syntax for creating an object of
         // a Java class. There is automatic conversion of
         // "a" from Cyan to a Java string
        return StringBuffer("a");
    }
       // parameter type is a Java class
    func retSB: StringBuffer sb -> String {
         // automatic conversion from Java string to Cyan string
         // 'sb toString' is the sending of a message in Java.
        let String s = sb toString;
        // return s OR just the line below
        return sb toString;
    }

    func test {
         // automatic conversion
        var Int n = Integer(0);
        assert n == 0;
        n = Integer new: 1;
        assert n == 1;
        var String s = self retSB toString;
        s println;
        (self retSB: StringBuffer("abc")) println;
        /*
           self retSB toString println
        does not work. The type of 'self retSB toString' is java.lang.String
        which does not have a 'println' method.
        */

           // conversion from 'int' of Java to 'Int' of Cyan
        n = java.lang.String("abc") length;
```

```
        // assignments from anything to java.lang.Object
        // are allowed
      var java.lang.Object obj = 0;
      obj = "ok";
      obj = String;

  }
end
```

There is automatic conversion from `Integer` and `int` to `Int` in array indexing. Java arrays can be declared as in Java. Only one-dimensional arrays are supported.

```
package javaInter.main

  // import Java package
import java.util
  // import Java class
import java.io.File
import java.lang

object JavaTest2
    func test {
        var intArray = [ 7, 2, 3 ];

          // convertion from Java Integer(0) to Cyan 0
        assert intArray[ Integer(0) ] == 7;
        assert intArray[0] == 7;
          // convertion from Java Integer(9) to Cyan 9
        intArray[0] = Integer(9);
        assert intArray[0] == 9;
        assert intArray[Integer new: 2] == 3;

    }

    func messagePassingJavaTest {

          // File is a Java class
        var File file;
        file = File new: "C:\\Dropbox\\Cyan";
           // message send in Java
        if file isDirectory {
            Out println: (file getCanonicalPath), " is a directory"
        }
           // message send in Java
        file setReadable: true;
        let ok = true;
           // conversion of Boolean 'ok' in Cyan to boolean 'ok' in Java
        file setReadable: ok;
          // a Java array
```

```
        let File [] fileList = file listFiles;
           // for does not work with Java arrays yet
        for n in 1..<fileList length {
               // indexing with a Java array
            let File ff2 = fileList[n];
            // Out println: ff2 getCanonicalPath;
        }


        var java.lang.StringBuffer sb = java.lang.StringBuffer();

        sb append: "Append in String Buffer!";
        let String ssss = sb toString;
        assert ssss == "Append in String Buffer!";

          // conversion of Cyan string to Java string
        var java.lang.String js = "abcdef";
        js = "a b c d";
           // automatic type for Java local variable
           // strArray has type java.lang.String[]
        var strArray = js split: " ";
           // conversion from Java string to Cyan string
        var String cyanStr = strArray[0];
        assert cyanStr == "a";
           // indexing of a Java array
        cyanStr = strArray[1];
        assert cyanStr == "b";
        cyanStr = strArray[2];
        assert cyanStr == "c";
        /* the code
               assert cyanStr[2] == "c";

           does not work. cyanStr[2] has type java.lang.String and therefore the '=='
              operator
           used is that of Java. It compares the pointer, not the contents as in Cyan.
        */

    }
end
```

Generic classes of Java can be used but there should be no restrictions on the type parameters. The real type parameters can be both Java classes and Cyan prototypes.

```
package javaInter.main

  // import Java package
import java.util
  // import Java class
import java.lang
```

```
object JavaTest3

    func test {
           // parameter to ArrayList is Int, a Cyan prototype
        var ArrayList<Int> intArray = ArrayList<Int> new;
        intArray add: 88;
        intArray add: 99;
        var Int n = intArray get: 0;
        assert n == 88;
        n = intArray get: 1;
        assert n == 99;

           // parameter to ArrayList is StringBuffer, a Java class
        var ArrayList<StringBuffer> strArray = ArrayList<StringBuffer> new;
        strArray add: StringBuffer("aaa");
        strArray add: StringBuffer("bbb");
        var String s = strArray get: 0;
        assert s == "aaa";
        s = strArray get: 1;
        assert n == "bbb";

        var java.util.Set<Int> iset = java.util.HashSet<Int>();
        iset add: 0;
        iset add: 1;
        iset add: 2;
        var Boolean b = iset contains: 0;
        assert b;
        b = iset contains: 1;
        assert b;
        /*
            just
                 assert iset contains: 1;
            does not work. The return type of 'iset contains: 1' is java.lang.boolean
            which is not automatically cast to Boolean. Macro assert in fact sends
            unary message '!' to the expression.

        */

        b = iset contains: 3;
        assert !b;

    }

end
```

Generic prototypes of Cyan can take Java classes as real parameters. However, due to the fact that every Cyan prototype inherits from `Any`, this will hardly work.

```
package javaInter.main

open
object GP<T>

    func init: T elem { self.elem = elem }

    func get -> T = elem;
    func set: T elem { self.elem = elem }
    func getStr -> String {
        var java.lang.Object any = elem;
        var String s = any toString;
        return s
    }

    var T elem
end
```

Note the method `getStr` of prototype `GP`. The prototype expects a Java class as real parameter to the generic prototype. Therefore it uses special code to deal with Java objects. This prototype can be used as in the next example.

```
package javaInter.main

import java.lang

object JavaTest4

    func test {

        let a56 = GP<Integer>(9);
        assert a56 getStr == "9";
        a56 set: Integer(5);
        assert a56 getStr == "5";

    }

end
```

Java `Boolean` and `boolean` values can be used in `if`, `while`, and `repeat-until` statements.

```
package javaInter.main

import java.lang

object JavaTest5

    func ifWhileTest {
        var java.lang.Boolean b = false;
```

```
    if b { assert false; }
    else {
        assert true;
    }
    b = true;
    if b { assert true; }
    else {
        assert false;
    }
    while b {
        b = false
    }
    var Boolean ok = b;
    assert !ok;
}

end
```

There are some limitations on the use of Java inside Cyan code:

(a) whenever a prototype `T` and a Java class or interface `T` is imported, the compiler will use the prototype. The collision of names will not be considered an error. Then the word "`String`" in a Cyan code will always mean the Cyan prototype even when package "`java.lang`" is imported. To use the Java class, prefix it with the package:

```
java.lang.String
```

(b) the Java basic types, `int`, `char`, etc cannot be used in a Cyan code. They do not belong to package `java.lang`, they are native to Java. Use the wrapper classes (`Integer`, `Character`, etc) instead. If a method returns an array of a basic type, use a Java class that casts this array to an array of a wrapper class.

```
class CastArray {
    public Integer[] intToInteger(int []v) {
        Integer []newV = new Integer[v.length];
        int i = 0;
        for ( int elem : v ) {
            newV[i] = elem;
            ++i;
        }
        return newV;
    }
}
```

This is awful;

(c) `null` of Java cannot appear in a Cyan code. Use prototype `cyan.lang.Null` to get the `null` value. It is returned by

```
Null getNull
```

To compare an expression `expr` to `null`, use

```
Null equalNull: expr
```

It returns a `Boolean` object. Note that

```
expr == Null getNull
```

will not work. The compiler will look for a method `_equal_equal` (or something similar) in the class of `expr`. It will not use the original `==` method of Java.

(d) a Cyan prototype cannot inherit from a Java class because it must inherit, even indirectly, from `Any`;

(e) `for` statements do not accept Java types. They will do some day;

(f) Java methods that take a variable number of parameters cannot be called in Cyan.

The above limitations restric the usability of the mixing Cyan and Java. However, future versions of Cyan may improve the communication between the two languages.

Objects of basic Cyan types are used to do the communication between the Cyan code and the Java code inside Cyan code. Then the two codes are relatively separated from each other.

In order to use Java classes, it is necessary tell the compiler where to find the jar files with the Java packages — the Java code must be in a jar files. This is made with the compiler option "`-cp`" as in

```
saci "C:\Dropbox\Cyan\cyanTests\master"
     -cyanlang "C:\Dropbox\Cyan\lib"
     -cp "C:\Dropbox\Cyan\external-Java-libs\javassist.jar"
```

Option "`-java`" may be used if only the packages of the basic libraries of Java, file "rt.jar", will be used.

```
saci "C:\Dropbox\Cyan\cyanTests\master"
     -cyanlang "C:\Dropbox\Cyan\lib"
     -java
```

Caveat: interoperability with Java has not been thoroughly tested.

4.21 Future Enhancements

Operator `|` may be used as a method name. It is in prototype `Int`, for example. A future version of the language will use `|` only for unions. It will not be possible to use it as a method name.

Type `Dyn` is not a real prototype, it is *virtual* because there is not a source code associated to it. Then this type can appear as type where only types are expected, as variable type or return value type. But `Dyn` cannot receive a message in an expression:

```
var String s;
s = Dyn prototypeName; // compile error
```

`Dyn` is the only prototype of this kind in Cyan. Future versions of `Cyan` will make every non-tagged `Union` prototype virtual too. This saves the creation of an object for a union. The language will be faster.

The language may support the Elvis operator, `Nil`-safe message sends, and `Nil`-safe array access. They will be as follows.

The Elvis operator would be implemented as a method `ifNil:`, `Nil`-safe message sends would have all keywords prefixed with `?.`, and `Nil`-safe array access would be made with `?[` and `]?`. See the examples.

```
var String userName;
   // getUserName is a method name
var Nil|String gotUserName = UserDataBase getUserName;
userName = ifNil gotUserName, "anonymous";
```

The last line is the same as

```
    type gotUserName
        case String s99 {
            userName = s99
        }
        case Nil {
            userName = "anonymous"
        }
```

`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

```
type v
    case IndexedList<String> v8 {
        v8 at: 0 put: "Gauss";
    }
```

Or the same as

```
cast v8 = v {
        v8 at: 0 put: "Gauss";
}
```

There should not be any space between the `?.` and the keyword. And all keywords 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 Person|Nil firstMember;
cast c2 = clubMembers {
    firstMember = c2[0];
}
else {
    firstMember = Nil
}
```

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 = ifNil (clubMembers?[0]? ?.name), "no member";
```

Chapter 5

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, an exception is thrown. Statically-typed languages produce faster programs and the 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 some mechanisms that help to achieve this objective, described next.

A message send whose keywords 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 keywords. For example, in the code below, the compiler does not check whether prototype `Person` defines a method with keywords `name:` and `age:` that accepts as parameters a `String` and an `Int`.

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

The receiver of a message of this kind cannot be `super`. That would not make sense because message sends to `super` are static calls. We know which method will be called at compile-time.

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

```
func 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.

The expression that receives a `?`-message cannot be a prototype:

```
var a = A ?new;
```

[1] With a method `open`.

If every message keyword (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 keywords 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[2] that is supertype and subtype of every other prototype including `Nil`. Therefore assignments to and from `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, and return value of methods). At runtime the `Dyn` expression should refer to a prototype that is subtype of the type of the left-hand side variable.[3]

```
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` (which includes `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 keywords.

```
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 keywords. `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 receiver expression has type `Dyn`. Therefore the return value is not checked. In this example, the compiler consideres 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
}
```

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 `Dyn`, as expected.

```
    // n has type Dyn
var n = obj ?value;
```

Dynamic message sends, with keywords preceded by `?`, plus the reflective facilities of Cyan can be used to create objects with dynamic fields. Object `DTuple` of the language library is a tuble initially without fields, which can be added dynamically:

[2] Internally the compiler considers that `Dyn` is a prototype declared in package "`cyan.lang`".

[3] or indexing expression like "`a[0] = dynVar`".

```
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` are 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`.

`DTuple` keeps a list or hash table of pairs "(fieldName, fieldValue)". Each field has the name `fieldName` and a value `fieldValue`. When a `DTuple` object receives a message `id: aValue`, method `doesNotUnderstand:` is called[4] and a search is made in this list or hash table. If no field with name `id` is found, one is created with value `aValue`. If a field is found, its value is updated to `aValue`.

When the `DTuple` object receives a message `id` and `id` is not a method declared in `DTuple` or `Any`, method `doesNotUnderstand:` is called. It searches in the list or hash table for `id`. If it is not found, method `doesNotUnderstand:` of `Any` is called. If `id` is found, its value is returned. For more information, see file `DTuple.cyan` in package `cyan.lang`.

The previous `DTuple` example 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, ASCII 96) for calling a method whose keywords are in string variables. Each keyword used at runtime is the contents of each variable.

```
var String s = "print";
0 ‘s;
```

The last line sends message `print` to `0`.

In a message send with parameters the variable names should be followed by `:` as usual.

```
var String s = "at";
var p = "put";

let IMap<String, Int> map = [ "two" -> 2 ];

map ‘s: "one" ‘p: 1;
```

[4] Unless this is a method of `Any`.

```
assert map get: "one" == 1
```

The method to be called has keywords `s ++ ":"` and `p ++ ":"`, in which `++` is used for concatenating strings. That is, the method to be called is `at:put:`. One may add `:` at the end of the string too:

```
var s = "at:";
var p = "put:";

let IMap<String, Int> map = [ "two" -> 2 ];

map `s: "one" `p: 1;

assert map get: "one" == 1
```

Methods whose names are operators may also be called but the backquote variable should be followed by ":".

```
for op in [ "+", "*", "-" ] {
    Out println: (6 `op: 2);
}
// prints 8, 12, 4
```

Each keyword preceded by backquote should be a variable of type `String`, `CySymbol`, or `Dyn`. It cannot be a field accessed through `self` as `self.name`. The receiver of a backquote message send cannot be `super`.

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`.

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 `IMap` takes one parameter and `put:` of `MyTable` takes two parameters, we cannot write

```
let IMap<String, Int> map = [ "one" -> 1 ];
n = MyTable put: map get: "one", j
```

The compiler could easily check that the intended meaning is

```
n = MyTable put: (map get: "one"), j
```

by checking the prototypes `IMap` and `MyTable`.

However, if the type of `map` is `Dyn`, this checking would not be possible. The type information would not be availabe at compile time. Therefore Cyan consider that a message send includes all the keywords that follow the receiver and that are not in an expression within parentheses;

(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 90, the same methods are called regardless of the static type of the parameter to `eat:`. Therefore the use of static or dynamic typing does not change the semantics of message passing. That allows one to change from static to dynamic typing and vice-versa without fear of breaking the program.

There is one more reason to employ the runtime search algorithm for methods that Cyan uses, which does not consider the static, compile-time type of the receiver and parameters: 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 with 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 in the declaration of variables and parameters. For example, before a local variable declaration it is necessary to use "`var`", which asserts that a list of variables follow, preceeded or not by a type. For example, the declaration of `Int` variables in Cyan is

```
var Int a, b, c;
```

To declare these variables with type `Dyn` one can write

```
var a, b, c;
```

If an expression is assigned to a variable in its declaration, its type will be the compile-time type of the expression if the type is not supplied:

```
var count = 0; // count has type Int
var flavor = "vanilla"; // flavor has type String
```

If `let` is used instead of `var` the expression should always be supplied and therefore read-only variables will have the type of the expression.

Remember that method parameters without types have type `Dyn`:

```
func add: key, value { ... }
```

Both `key` and `value` have type `Dyn`. The return value type should always be supplied.

Chapter 6

Generic Prototypes

Generic prototypes in Cyan play the same role 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 `hashCode -> Int` (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<A>` 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 11.5, and in Chapter 7.

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** or *generic parameters* of the generic prototype. Each of them starts with an upper-case letter.

There should be no space between the prototype name, `P`, and the character "`<`". Space may follow "`<`" as in

```
package ds

object Stack< T >

   func push: T elem { ... }
   func pop -> T { ... }
   func 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 *generic parameter* `T` in prototype `Stack<T>` by `Int`. This process is called **instantiation** of a generic prototype and `Int` is called a **real parameter** or **real argument** to the generic prototype. There are restrictions on where a formal parameter can appear in the source code of the prototype (`Stack` in the example) and when it is replaced by a real parameter.

A formal parameter may appear as a keyword name (both in a method declaration and in a message send), type, identifier in an expression, parameter to an metaobject annotation, and after `#` (to define a symbol). No local variable or parameter will have the name of a **formal parameter** because the former start with a lowercase letter and the latter with an uppercase letter. 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. Only the part of the rules that matters are shown.

```
QualifId            ::= Id { "." Id }
ExprPrimary         ::= QualifId { "<" TypeList ">" }+ [ ObjectCreation ] |
                        QualifId { "<" TypeList ">" }+ |
                        "typeof" "(" QualifId [ "<" TypeList ">" ] ")"
MetSigUnary         ::= Id
SelecWithParam      ::= IdColon |
                        [ "[]" ] IdColon ParamList
InterMethSig2       ::= Id |
                        { IdColon [ InterParamDecList ] }+
```

SingleType ::= QualifId { “<” TypeList “>” } |
“typeof” “(” QualifId [“<” TypeList “>”] “)”
Annotation ::= “@” Id
[“(” ExprLiteral [“,” ExprLiteral] “)”]
[LeftCharString TEXT RightCharString]

The rule ExprPrimary expands to three things:

(a) a generic prototype instantiation as `Array<Int>`;

(b) an object creation of a generic prototype instantiation such as
`Array<Int>(100)`

(c) the `typeof` compile-time function:

```
typeof(x) prototypeName;
typeof(Array<Int>) prototypeName;
```

ExprPrimary expands only inside expressions.

MetSigUnary and SelecWithParam are used to produce names of unary methods and keyword names of keyword methods. They expand to code like these:

```
asString
read:
[] at: Int
between: Int start, Int theEnd
```

These rules are used to produce method signatures:

```
func asString -> String { ... }
func read: -> Array<Byte> { ... }
func [] at: Int -> Int { ... }
func between: Int start, Int theEnd -> Array<Int> { ... }
```

InterMethSig2 produces signatures of methods in interfaces. The difference with SelecWithParam/MetSigUnary is that the type of parameters should appear (this rule is not shown). The examples are the same as those of SelecWithParam/MetSigUnary.

SingleType produces a type. It may expand to

```
Person
Array<Int>
typeof(x)
typeof(cyan.lang.Int)
typeof( Array<Int> )
typeof( cyan.lang.Array<Int> )
```

Annotation expands to a metaobject annotation. It may expand to any of the following.

```
@checkStyle
@annot("main")
@concept{* T implements I *}
```

The Id in these rules given above can be generic prototype formal parameters. Then if `T` is a formal parameter, the following uses are legal. The lines do not compose a code, they are just a set of examples.

```
myProto = Array<T>;
myArray = Array<T>(100);
var protoName = typeof(T) prototypeName;
var String protoName2 = typeof(Array<T>) prototypeName;
func T -> String { ... }
func T: -> Array<Byte> { ... }
func [] T: Int -> Int
func T: Int start, Int theEnd -> Array<Int> { ... }
T println;
[ 0 ] T println;
var T x;
var Array<T> tArray;
var typeof(T) aT;
var typeof(Array<T>) anotherTArray;
var typeof(cyan.lang.Array<T>) anotherTArray;
@annot(T)
@feature(T, T)
```

A formal parameter can also appear inside the text of an annotation:

```
package main

@concept{*
    T has [ func next -> T ]
*}
object Element<T>
   ...
end
```

Let us see some examples. First a generic interface.

```
package main

interface InterNice<T>

    func add: T -> Int

end
```

A superprototype used in the next example.

```
package main

open
object SuperNice<T>

    func superMethod: Int n -> Int = n;

end
```

A nice example with all possible uses of generic prototypes

```
package main
```

```
object Nice<T, R, S, AsString> extends SuperNice<T> implements InterNice<T>

    func init: T x {
        self.x = x;
        people = "people";
        tArray = Array<T> new;
    }

    @annot(T)
    @feature(AsString, T)
    func example {
        var typeof(T) aT;
        var typeof(Array<T>) aTArray;
        var typeof(cyan.lang.Array<T>) anotherTArray;

        aTArray = Array<T>();
        anotherTArray = Array<T>();

        var myProto = Array<T>;
        var myArray = Array<T>(100);
        var protoName = typeof(T) prototypeName;
        var String protoName2 = typeof(Array<T>) prototypeName;
        Out println: protoName, protoName2, myProto prototypeName,
           myArray prototypeName;

        var Boolean found = false;
        for elem in annotList {
            if elem == "person.Person" { found = true }
        }
        assert found;
    }

    override
    func AsString -> String {
            // after a # to define a symbol
        return #T ++ #R ++ #S;
    }

    // keyword name
    func R: Char p -> Int { return p asInt }

        // type
    override
    func add: (T p) -> Int {
        var S.Person per = T("Carolina", 7);
        var per2 = S.Person("Livia", 11);
        var typeof(T) aPerson;
```

```
        var typeof(S.Person) otherPerson;

        return 0
    }

        // type
    T x
    var Array<T> tArray
    String people;

end
```

A prototype that uses `Nice` is

```
package main

import people

object Program

    func run {

        let Person livia = Person("Livia", 11);
        typeof(livia) prototypeName println;

        var nice = Nice<Person, charToInt, people, asString>(livia);

        assert nice asString == "people.PersoncharToIntpeople";
        assert (nice charToInt: 'a') == 97;
        nice add: Person("Carol", 7);
        assert (nice superMethod: 0) == 0;

    }
end
```

A formal parameter may be the name of a generic prototype:

```
object NiceExample< T >
    public var T<Int> value
end
```

In the instantiation `NiceExample<Empty>` the compiler checks whether there is an `Empty` prototype. That is, a non-generic prototype called "`Empty`". After the instantiation, when `NiceExample<Empty>` is compiled, the compiler checks whether there is a generic prototype `Empty` that takes one argument.

There is a compile-time error if the formal parameter is the name of a parameter because variables and parameter should start with a lowercase letter.

```
object Wrong< T >
    func myError: (Int T) { } // compile-time error
end
```

A formal parameter appearing as a substring of a Cyan symbol is not replaced.

```
object P<T>
    func print { #T1 print }
end
```

Prototype `P<Int>` is

```
object P<Int>
    func 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>
    func print { #T +++ 1 print }
end
```

Prototype `P<Int>` would be

```
object P<Int>
    func 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.

In a generic prototype instantiation, each real parameter should be a type or an identifier starting with a lower-case letter (which is called *identifier parameter*). The first one one can be a generic prototype instantiation. Then the general format of a real parameter is given by rule `RP` of the grammar below. IdLowerCase stands for "identifier starting with a lower-case letter".

RP	::= IdLowerCase | 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. If the identifier starts with a lower-case letter, it is not considered a type, 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 12.

A generic prototype may declare more than one generic parameter:

```
package cyan.lang

interface IMap<K, V> extends Iterable<Tuple<key, K, value, V>>

    func [] at: K key -> V|Nil
    func [] at: K key put: V value -> V|Nil
    ...
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 name "`Tuple`" or "`Interval`". You are invited to use single letters to formal parameter names.

Currently there is no way of declaring a private generic prototype in Cyan (or a private regular prototype). The implementation of this feature would make the compiler more complex. We believe private generic prototypes would be rarely used and almost never necessary.

A real parameter to a generic prototype cannot be an integer number as in C++ [Str13]. However, metaobject `extract` can be used to simulate the passing of an `Int` as parameter.

```
package main

object Store<T>
    func set: Int elem {
        if elem > @extract(T) {
            throw: ExceptionStr("Number out of limits in Store prototype")
        }
        self.elem = elem
    }
    func get -> Int = elem;
```

```
    var Int elem = 0;

end
```

If `T` has the form `intN` or `int_N` in which `N` is a literal `Int`, then

```
    @extract(T)
```

results in `N`.

```
var s100 = Store<int_100>();
s100 set: 99; // ok
s100 set: 200; // exception thrown
```

6.1 Generic Prototypes with real arguments

A prototype that is not generic can be declared using the generic prototype syntax:

```
package ds

object Stack<Int>
   func push: Int elem { ... }
   func 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 arguments**". Each one of the parameters that appear inside `<...>` will be called "**real argument**".

A *real argument* of a *generic prototype with real arguments* can be:

(a) *identifier parameter*, which is a single identifier starting with a lower-case letter such as "`t`" or "`add`". For example,

```
interface ISingle<write>
    func 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>
    func 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>
```

```
    func add: Person
end
```

This qualifier 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< Tuple<key, String, value, Int>, ds.Map<String, main.Person> >
   ...
end
```

By the above rules, a prototype can be used as a real argument 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>
   func push: main.Person elem { ... }
   func pop -> main.Person { ... }
   ...
end
```

In this way the compiler knows whether an identifier that appears after `<` is a formal parameter or a *real argument* of a *generic prototype with real arguments*. If the parameter:

(a) is composed by a single identifier that starts with a lower-case letter it is a *real argument*. 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 argument*;

(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 argument*;

(e) is a generic prototype instantiation, then it is a *real argument*.

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 arguments* may be useful for providing a more efficient implementation for a given type. For example, a `HashMap<Int, Int>` implementation could somehow be more efficient because `Int`´s are used.

6.2 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 ">" and there should be just one set of pairs "<" and ">". The generic prototype is used with any number of set of pairs "<" and ">". Then the prototype `P` of the example above is used for all of the following instantiations:

```
P<Int> P<Int, String> P<Int><Int> P<Int><Char, Double>
P<Char, Int><Float, String, Char><Int><Int, Int><Nil>
```

Of course, the syntax is misleading for it induces one to think there is just one set of pairs "<" 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 (See "The Cyan Metaobject Protocol" in the Cyan site). For example, prototype `Tuple` is declared as

```
package cyan.lang

@createTuple
object Tuple<T+>
end
```

In an instantiation of `Tuple`, as `Tuple<Int, String>`, metaobject `createTuple` has access to the list of real argument, `Int` and `String`. Basead on these parameters, `createTuple` generates Cyan code that replaces the metaobject annotation. That is, "`@createTuple`" is replaced by declarations of methods and fields produced by `createTuple`.

It is tempting to add language constructions to handle a variable number of real arguments. 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 implemented using generic prototypes with a varying number of arguments: `Tuple`, `Union`, and `Function`.

6.3 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 arguments;

(b) formal parameters without a + operator;

(c) a formal parameter (just one) 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+>
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
```

6.4 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 arguments**

```
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, consider the source file below.

```
package example

object Test<Int, Char><main.Person>
end
```

It should be in file

```
Test(Int,Char)(main.Person)
```

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+>
   ...
end
```

should be in file "`P(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`

6.5 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 arguments, or a varying number of parameters. All source files should be in the same package directory which means the source file names are different.

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 arguments that match exatly the real arguments `T1`, ... `Tn`, ... `U1`, ... `Um`. If none is 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 (file `P(1+).cyan`). If no adequate generic prototype is found, the compiler signals an error.

For example, suppose that the instantiation is

```
Tuple<key, Int, value, String>
```

First the compiler searches for a prototype `Tuple<key, Int, value, String>` which should be in a file

```
Tuple(key,Int,value,String).cyan
```

If this prototype does not exist, it searches for a generic prototype

```
Tuple<T1, T2, T3, T4>
```

with four formal parameters. This should be in a file

```
Tuple(4).cyan
```

If there is no such prototype, the compiler searches for

```
Tuple<T+>
```

which should be in a file `Tuple(1+).cyan`.

If the instantiation uses other instantiations the process is recursive. In

```
Tuple<key, Tuple<Array<Int>, Union<Int, Char>> >
```

the compiler searches for a prototype with this name which should be in file

```
Tuple(key,Tuple(Array(Int),Union(Int,Char))).cyan
```

6.6 Concepts

A generic prototype may assume that one of its parameters, say `T`, is a prototype that defines some methods, is subprototype of some other prototype, implements a certain interface, and so on.

```
package main

object Test<T>
    func run {
        var T x = T();
        x open;
        x write: "not all is ok";
        var IHas<String> h = x;
        if h has: "ok" {
            "has ok" println;
        }
    }
end
```

In this example, the code assumes that `T`:

(a) is a prototype;

(b) has a method `init` without parameters;

(c) has an unary method `open`;

(d) has a method `write:` that can accept a string as parameter;

(e) implements interface `IHas<String>`.

If `Test` is instantiate with a prototype that does not define `init`, `open`, `has: String`, `write: Any`, or `write: String`, a compilation error will occur. The compiler will show a stack of generic prototype instantiations:

```
In file C:\Dropbox\Cyan\cyanTests\negTestsJose\t10\main\--tmp\Test(Int).cyan
       (line 9 column 9)
object/interface main.Test<Int>
Method open was not found in prototype Int or its superprototypes
Stack of generic prototype instantiations:
    main.Test<Int> line 9 column 9
    main.OtherTest<Int> line 8 column 30
```

```
main.Program line 5 column 21

    x open;
```

This message says that in line 5, column 21, of `main.Program` prototype `main.OtherTest<Int>` was instantiated. Then in line 8, column 30, of `main.OtherTest<Int>` prototype `main.Test<Int>` was instantiated. Then in line 9, column 9, of `main.Test<Int>` there was the error "*Method open was not found in prototype Int or its superprototypes*". The last line shows the offending line:

```
    x open;
```

Although the error message helps to trace the error, it shows code internal to the generic prototype. The error message will not be easily understood by a user of the generic prototype. Instead of allowing this kind of error to happen one can use Concepts [GJS+06]. They can be used to specify restrictions that the real arguments to a generic prototype should have. If the real argument does not obey the restrictions, an error message is issued in the instantiation of the generic prototype. Code internal to the generic prototype is not shown.

Concepts are predicate on types and values. In Cyan, concepts are implemented using a metaobject and are predicates on types and *identifier parameter*.[1] The concepts of a generic prototype are made based on the semantic needed in the prototype. For example, suppose a generic prototype `GroupWork` has a generic parameter which should be a Group.[2]

In the methods of `GroupWork` it is assumed that `T` has methods `inverse`, `unit`, and binary `*`. And that these methods obey the semantics expected for a Group.

```
package main

object GroupWork<T>

    func work: T a {
        printexpr a;
        printexpr a inverse;
        printexpr a unit;
        printexpr a * a unit;

        assert a * a inverse == T unit;
        assert T unit == a inverse * a;
        assert a unit == T unit;
        assert a * T unit == a;

    }

    func workout: T a, T b, T c {
        printexpr (b inverse * a inverse) * a * b;
        printexpr (c inverse * b inverse * a inverse) * a * b * c;
```

[1] Identifiers starting with lowercase letters passed as parameters to generic prototypes, like `speed` in `BinaryTree<Int, speed>`.

[2] A group is a set G with an operation $*$ in such a way that, if $a, b, c \in G$, $a * b \in G$, $a * (b * c) = (a * b) * c$, there exist $e \in G$ (the unit of G) such that $a * e = e * a = a$ for all $a \in G$, and for all $a \in G$ there exists an element b called the inverse of a such that $a * b = b * a = e$.

```
        printexpr b inverse * b;
        printexpr c * b * a * ( a inverse * b inverse * c inverse );

        assert c * b * a * ( a inverse * b inverse * c inverse ) == T unit;
        assert a*(b*c) == (a*b)*c;
        assert c*(b*a) == (c*b)*a;

    }
end
```

As said before, prototype `GroupWork` may be instantiated with any type, which will result in a compile-time error if the real argument does not support the methods expected in the prototype body. And there will be runtime errors if the methods do not have the expected semantics.

The error messages can be made clearer with the use of metaobject `concept`. An annotation should be attached to the generic prototype:

```
package main

@concept{*
    // both kinds of comments are allowed

    T has [ func * (T other) -> T
            func unit -> T
            func inverse -> T ],
        "T should have methods *, unit, and inverse in order to be considered element of
             a Group",

    axiom opTest: T a, T b, T c {%

        if (a * (b * c) != (a * b) * c) ||
           (c * (b * a) != (c * b) * a {
            return "T is not associative"
        }
        return Nil
    %},

    axiom unitTest: T a, T b, T c {%

        if (a * a unit != a unit * a) ||
           (b * a unit != b unit * b) ||
           (a unit * b unit != c unit * c unit) {
            return "The unit element of T is not an identity"
        }
        return Nil
    %},

    axiom inverseTest: T a, T b, T c {%

        if (a * a inverse != b unit) ||
```

```
            (a unit != b inverse * b) ||
            (c inverse * c != T unit) {
             return "The inverse operation is not working properly"
        }
        return Nil
    %}


*}
object GroupWork<T>
    func work: T a, T b, T c {
        printexpr a asInt;
        printexpr a inverse asInt;
        printexpr a unit asInt;
        printexpr (a * a inverse) asInt;
        printexpr (a * a unit) asInt;
        printexpr ((b inverse * a inverse) * a * b) asInt;
        printexpr ((c inverse * b inverse * a inverse) * a * b * c) asInt;

        printexpr (b inverse * b) asInt;
        printexpr (c * b * a * ( a inverse * b inverse * c inverse )) asInt;

    }


    func workout: T a, T b, T c {
        printexpr ((b inverse * a inverse) * a * b) asInt;
        printexpr ((c inverse * b inverse * a inverse) * a * b * c) asInt;

        printexpr (b inverse * b) asInt;
        printexpr (c * b * a * ( a inverse * b inverse * c inverse )) asInt;

        let tunit = T unit;
        assert c * b * a * ( a inverse * b inverse * c inverse ) == tunit;
        assert a*(b*c) == (a*b)*c;
        assert c*(b*a) == (c*b)*a;

    }

end
```

The Domain Specific Language (DSL) of the `concept` metaobject annotation of this example specifies the restrictions the generic parameters should have. The first line,

```
    T has [ func * ...
```

means that parameter `T` should have the methods between `[` and `]`. The string that follows,

```
    "T should have methods * ..."
```

is the error message issued by the metaobject if `T` does not define the methods. Each method can have its own error message:

```
package main

@concept{*
    T has [ func * (T other) -> T "T should support operator * in order to be a Group",
            func unit -> T
            func inverse -> T ],
        """T should have methods *, unit, and inverse in order to be considered element
            of a Group""",
    ...
*}
object GroupWork<T>
    ...
end
```

If the message after the method signature is not given, the message after the predicate is used. If there is no message after the predicate, a standard message is used. In this last example, if a type `MyGroup` does not define method `*` in `GroupWork<MyGroup>`, message

```
"MyGroup should support ..."
```

is issued. If `MyGroup` does not define method `unit`, message

```
"MyGroup should have methods *, unit, and ..."
```

is issued. If there was no message

```
"T should have methods *, unit, and  ..."
```

then the standard message would be used in this last case.

`axiom` is a keyword of the `concept` DSL. It defines a method whose body appears between `{%` and `%}` in this example (any left char sequence can be used. See the definition of left char sequence for metaobjects). If the metaobject annotation has parameter `test` as in

```
...
@concept(test){*
   ...
*}
object GroupWork<T>
    ...
end
```

the metaobject will create test packages, prototypes, and methods. In particular, for each concept annotation there will be a test prototype. This prototype will have a method for each axiom and it will be in a test directory created in a directory `--test` of the program. For example, suppose the program is in directory

```
C:\Dropbox\Cyan\cyanTests\simple
```

Package `main` of this program contains the `GroupWork` prototype. The metaobject `concept` will create the path

```
C:\Dropbox\Cyan\cyanTests\simple\main_ut\
   groupwork_lt_main_d_intgroupplus_gt__axiom_test
```

if `GroupWork` is instantiated with parameter `IntGroupPlus` (a prototype). The path above is shown in two lines to fit in the page. Inside this directory there will be a prototype

```
GroupWork_lt_main_d_IntGroupPlus_gt__Axiom_Test
```

with the axioms. It is shown next.

```
package main_ut.groupwork_lt_main_d_intgroupplus_gt__axiom_test
```

```
object GroupWork_lt_main_d_IntGroupPlus_gt__Axiom_Test

    func opTest_0: main.IntGroupPlus a, main.IntGroupPlus b, main.IntGroupPlus c ->
        String|Nil {
        if (a * (b * c) != (a * b) * c) ||
           (c * (b * a) != (c * b) * a {
            return "main.IntGroupPlus is not associative"
        }
        return Nil
    }

    func unitTest_1: main.IntGroupPlus a, main.IntGroupPlus b, main.IntGroupPlus c ->
        String|Nil {


        if (a * a unit != a unit * a) ||
           (b * a unit != b unit * b) ||
           (a unit * b unit != c unit * c unit) {
            return "The unit element of main.IntGroupPlus is not an identity"
        }
        return Nil
    }


    func inverseTest_2: main.IntGroupPlus a, main.IntGroupPlus b, main.IntGroupPlus c ->
         String|Nil {
        if (a * a inverse != b unit) ||
           (a unit != b inverse * b) ||
           (c inverse * c != main.IntGroupPlus unit) {
            return "The inverse operation is not working properly"
        }
        return Nil
    }
end
```

Each axiom gives origin to a method with the same name with a suffix number.

Another path will be created:

`--test\main_ut\groupwork_test`

It will contain usually one prototype for each formal parameter of the generic prototype and a test prototype. In this example, the prototypes will be `GroupWork_Test` and `T`.

```
package main_ut.groupwork_test

object GroupWork_Test
    func run {
        var main.GroupWork<T> testVar;

    }
```

```
end
```

Usually but not always a prototype is created for each parameter with the restrictions it should have. In the example, the real argument for `T` should have some methods such as `*`, `unit`, and `inverse`. Then prototype `T` is declared with these methods:

```
package main_ut.groupwork_test

object T
    func * T other -> T = T;
    func unit -> T = T;
    func inverse -> T = T;

end
```

`T` could have other restrictions as to implement a certain interface `IMyInter`. If that was the case, prototype `T` would be declared as

```
object T implements IMyInter
```

To test whether the DSL used in the metaobject annotation `concept` of `GroupWork` is enough, one should compile `GroupWork_Test` as a program. It instantiates `GroupWork` using prototype `T` above. If `GroupWork` assumes that its formal parameter `T` has a method not described in the concept DSL, a compilation error will occurs when compiling `GroupWork_Test`. This is the goal of creating prototypes in package `main_ut.groupwork_test`.

The test cases are not inserted in the program that uses the `concept` metaobject. They have to be separated compiled. The code of the axioms are not checked by metaobject `concept`. They may contain invalid Cyan code. Each axiom should return `Nil` if the there is no error or an error message as a `String`.

The valid predicates of metaobject `concept` are given below. We use `T`, `U`, and `S` for types and `I` for identifier parameters. These types may be anyone, including generic parameter.

predicate	meaning
`T is U`	`T` should be equal to `U`
`T implements U`	`T` should implement interface `U`
`S subprototype T`	`S` should be subprototype of `T`
`S superprototype T`	`S` should be superprototype of `T`
`T interface`	`T` should be an interface
`T noninterface`	`T` should be a prototype that is not an interface
`T has [ list of methods ]`	`T` should have the methods in the list
`T in [ list of prototypes ]`	`T` should be one of the prototypes in the list
`I in [ list of identifiers]`	`T` should be one of the prototypes in the list
`T identifier`	`T` should be an identifier parameter
`!  any of the predicates`	the opposite of the predicate should be true
`axiom axiomMethod`	generates the test case `axiomMethod`, which is similar to a method declaration

The compile-time function `typeof` can be used in the DSL of a concept. It may even be recursive:

```
package main
```

```
@concept{*
    T has [
        func search: typeof(R get) -> typeof(T get: Int)
           // recursion
        func at: Int -> typeof(R at: Int)
        func get: Int -> Double
        ],
    R has [
       func get -> Program
           // recursion
       func at: Int n -> typeof(T at: 0)
    ],
    typeof(R at: Int) in [ Int, Long, typeof(Int asString) ]
*}
object Strange<T, R>
end
```

However, there will be an error if parameter `test` is passed to this `concept` metaobject annotation. The metaobject will not be able to generate the test prototype because of the `typeof` compile-time function.

There may be errors in the predicates. For example, the DSL of a `concept` annotation may:

(a) demand that `T`, the parameter, is both a prototype (not interface) and an interface;

(b) demand that `T` inherits from `A` and that `A` inherits from `T`;

(c) have inconsistences such as demand that `T` should be in a list of prototypes and that implement a certain interface but no prototype in the list implements the interface;

(d) requires that `T` has incompatible methods such as

```
func get: Int -> Int
func get: Double -> Double
```

Most of these errors are not caught by the metaobject `concept`.

A list of predicates can be put in a file and reused. For example, the whole DSL that appears between `{*` and `*}` of metaobject annotation `concept` of `GroupWork` can be put in a file

```
group(T).concept
```

of the directory `--data` of the directory of package `main`. Now the example can be written as

```
package main

@concept{*
   main.group(T)
*}
object GroupWork<T>
   // as before
   ...
end
```

The concept file "`group(T).concept`" can be used by other packages of the programa. The file name should be preceded by the package name:

```
package other

@concept{*
    main.group(T)
*}
object MyGroupWork<T>
    ... // elided
end
```

This prototype is in package `other`. Note that it is not necessary to import the package `main` in order to use `group(T).concept`.

Package `cyan.lang` has several concept files:

`addable(T).concept`,
`arithmetic(T).concept`,
`comparable(R,S).concept`,
`container(T,R).concept`,
`equatable(T).concept`,
`init(T).concept`,
`init(T,R).concept`,
`init(T,R, S).concept`,
`iterator(T).concept`,
`iteratorSize(T).concept`,
`lessThan(T).concept`,
`predicate(T).concept`

To discover an up-to-date description of each of them, open the files in a directory
`cyan\lang\--data`

Whenever one uses a concept file its axioms are incorporated in the test prototype of the concept that used it. Then if a concept attached to a prototype uses concept `arithmetic(T)` of cyan.lang, its prototype test will have the axioms of the concept.

Metaobject `concept` may be used with non-generic prototypes. This is useful to enforce that a prototype should obey some restrictions and that some test cases should be generated for it. See the example

```
package algebra

@concept{*
    cyan.lang.arithmetic(Matrix),
    cyan.lang.init(Matrix, Int, Int)
*}
object Matrix
    ... // elided
end
```

Test cases would be generated for `Matrix`.

Currently it is not possible to pass a generic prototype as a parameter to a concept file:

```
package structures
```

```
@concept{*
    cyan.lang.init(Vector<T>) // error
*}
object Vector<T>
    ... // elided
end
```

6.7 Message Sends To Generic Prototype Instantiations

Compile-time messages can be send to a generic prototype instantiation through the syntax

```
Function<String, Int, Char> .# writeCode
```

Currently only message "`writeCode`" can be send. This message calls a virtual method[3] `writeCode` at *compile-time*. The generic prototype created for this instantiation is in file

```
Function(String,Int,Char).cyan
```

Virtual method `writeCode` writes to file

```
full-Function(String,Int,Char).cyan
```

of the directory of the project. This code has all the parts added by the compiler and metaobjects. Method `writeCode` is very useful to discover what is inside the real generic prototype. When there is a metaobject annotation in the generic prototype, as in `Function`, errors may be difficult to discover without the full code of the prototype.

The syntax `.#` only works if the generic prototype instantiation is where a type is expected as in a variable declaration. One can check the final version of prototype `Program` and `Function<Int, Int>` using the example that follows.

```
package main

object Program
    func run {
        var Program .# writeCode p;
        var Function<Int, Int> .# writeCode f;
    }
end
```

6.8 Future Enhancements

The compile-time message send using `.#` will be replaced by metaobject annotations attached to types. Then

```
Function<String, Int, Char> .# writeCode
```

will be replaced by

```
Function<String, Int, Char>@writeCode
```

Now `writeCode` can be used even inside an expression:

```
package main

object Program
    func run {
```

[3] It does not exist really.

```
        // Program is an expression here
      Out println: Program@writeCode prototypeName;
        // Function<Int, Int> is a type here
      var Function<Int, Int>@writeCode f;
   }
end
```

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 would be declared by putting the generic parameters after keyword `func` as in

```
object MySet
   final
   func<T> T add: (T elem) { ... }
   ...
end
```

When the compiler finds a message send using `add:` of `MySet`, as in

```
p = MySet add: p
```

it creates a specific method for that type using the compile-time type of `p`. This method could not override any superprototype method and it could not be redefined in subprototypes. It should be a `final` method.

The difference between using a generic method `add:` and declaring a method

```
func 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 func<T>  relate: (T first, T second)
```

demands that the arguments to the method be of the same compile-time type.

Chapter 7

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.

7.1 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
func exit
    // ends the program with a return value
func exit: (Int errorCode)
    // runs the garbage collector
func gc
    // current time in milliseconds
func currentTime -> Long
    // prints the stack of called methods in the
    // standard output
func printMethodStack

    // execute a command
func exec: String command
func exec: Array<String> commandList
    // see the Java method System.exec for help
func exec: Array<String> commandList, Array<String> envpList, String dir
@doc{*
    this method can be used as a dynamic storage for global variables
*}
func globalTable -> IMap<String, Dyn> = mapGlobalVariables;
```

7.2 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
    func readInt -> Int
    func readFloat -> Float
    func readDouble -> Double
    func readChar -> Char
    func readLine -> String
    ...
end

public object Out
    func (println: (Any)*)
end
```

7.3 Tuples

A tuple is an object with methods for getting and setting a set of values of possibly different types. A literal tuple is defined in Cyan between "`[.`" and "`.]`" as in:

```
var t = [. name = "Lívia", age = 4 .];
Out println: "name: " ++ t name ++ " age: " ++ t age;
```

This literal object has type `Tuple<name, String, age, Int>`. This is a tuple in which the fields have user-defined names.

A literal tuple may also have unnamed fields which are further referred as `f1`, `f2`, etc:

```
var t = [. "Lívia", 4 .];
Out println: "name: ", t f1, " age: ", t f2;
```

The type of this literal tuple is `Tuple<String, Int>` which is exactly the prototype

```
Tuple<f1, String, f2, Int>
```

Prototype `Tuple` can take any number of parameters. A metaobject is responsible for creating its methods and fields. The real prototype created from `Tuple<name, String, age, Int>` is below.

```
package cyan.lang

public final object Tuple<name, String, age, Int>


    func init: (String g1, Int g2) {
        _name = g1;
        _age = g2;
    }
    func name: String g1 age: Int g2 -> Tuple<name, String, age, Int> {
        return Tuple<name, String, age, Int> new: g1, g2;
    }
    @annot( #name ) var String _name
    func name -> String = _name;
    func name: String other { _name = other }

    @annot( #age ) var Int _age
```

```
    func age -> Int = _age;
    func age: Int other { _age = other }

    override
    func == (Dyn other) -> Boolean {
        if other isA: Tuple<name, String, age, Int> {
            var Tuple<name, String, age, Int> another;
            @javacode{* _another = (_Tuple_LT_GP__name_GP_CyString_GP__age_GP_CyInt_GT )
                _other;
            *}
            if name != (another name) { return false }
            if age != (another age) { return false }
            return true
        }
        else {
            return false
        }
    }

    override func asString -> String {
         return "[. name = " ++ name asStringQuoteIfString ++ ", age = " ++
             age asStringQuoteIfString ++ " .]"
    }

    func copyTo: (Any other) { }

end
```

Metaobject `annot` attaches to a field, shared variable, method, prototype, or interface a feature given by its parameter. This feature can be retrieved at runtime by method `featureList:` of the object.

The `Tuple<name, String, age, Int>` prototype has methods for getting and setting each tuple field:

```
var Tuple<name, String, age, Int> t;
t name: "Carolina" age: 1;
Out println: (t name);
t name: "Lívia";
t age: 4;
Out println: "name: ", t name, " age: ", t age;
```

An empty tuple is illegal:

```
var t = [. .]; // compile-time error: empty tuple
var anotherError = [..]; // unidentified symbol '[..]'
```

7.3.1 Future Enhancements

Object `Tuple` will have a method `copyTo:` that copies the information of the tuple into a more meaningful object. We will shown how it works using an example. We want to copy a tuple of type

`Tuple<String, Array<String>, String, Int>`

into an object of `Book`.

```
package main

@init(name, authorList, publisher, year)
object Book

    @annot( #f1 )
    @property String name

    @annot( #f2 )
    @property Array<String> authorList

    @annot( #f3 )
    @property String publisher

    @annot( #f4 )
    @property String year

    override
    func asString -> String {
        return authorList[0] ++ " et al. " ++ name ++ ". Published by " ++ publisher ++
            ". " ++ year ++ ".";
    }

end
```

However, `copyTo:` has to know to which field of `Book` it should copy field `f1` of the tuple. This method cannot choose one field based on the types — there are two of them whose type is `String`. We should use annotations for that: Now the following code will work as expected.

```
var Tuple<String, Array<String>, String, Int> t;
var b = Book("", [ "" ], "", 0);
t = [. "Philosophiae Naturalis Principia Mathematica",
        [ "Isaac Newton" ],
        "Royal Society",
        1687
     .];
t copyTo: b;
b println;
```

Tuples inside tuples are copied recursively. The `Manager` example is

```
@init(person, company)
object Manager

    @annot( #f1 )
    @property Person person

    @annot( #f2 )
    @property String company
```

```
end

@init(name, age)
object Person

    @annot( #f1 )
    @property String name
    @annot( #f2 )
    @property Int age
end
...

var manager = Manager new: Person, 0;
var john = [. [. "John", 28 .], "Cycorp" .];
john copyTo: manager;
assert john person name == "John" &&
       john person age == 28 &&
       john company == "Cycorp";
```

Method `copyTo:` can be used in grammar methods to store the single method argument into a meaningful object:

```
object BuildBook
    @grammarMethod{*
        (bookname: String (author: String)* publisher: String year: Int)
    *}
    func build: Tuple< String, Array<String>, String, Int> t -> Book {
        var book = Book new("", [ "" ], "", 0);
        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;
```

7.4 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 `attr` 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

```
package cyan.lang

object DTuple

    func init { ... }

    override
    func doesNotUnderstand: (String methodName, Array<Array<Dyn>> args) -> Dyn ...
    func contains: String fieldName -> Boolean ...
    func size -> Int = fieldList size;

    func getFieldList -> Array<String> = fieldList;

    // elided
end
```

`DTuple` redefines method `doesNotUnderstand:` in such a way that a field is added dynamically if it does not exist. When a non-existing method "tt f:" in a message send "`f: value`" is called on the object, `doesNotUnderstand:` simulates the addition to the receiver of a field `_f` and methods `f: T` and `T f`. The methods set and get the field. `T` is the type of `value`.

7.5 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> inter;
inter = 3..5;
    // this code prints numbers 0 1 2
0..2 foreach: { (: Int i :)
    Out println: i
};
    // this code prints numbers 3 4 5
inter repeat: { (: Int i :)
    Out println: i
};
    // prints the alphabet
'A'..'Z' foreach: {
```

```
        (: Char ch :)
        Out println: ch
    };
    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`.

```
package cyan.lang

   // T should be one of these types. Otherwise a compiler error is issued
@concept{*
    T in [ Byte, Short, Int, Long, Char ],
        "The parameter 'T' to this generic prototype instantiation should be Byte, Short
            , Int, Long, or Char"
*}
object Interval<T> implements Iterable<T>
    ...
       // method bodies elided
    override
    func == (Dyn other) -> Boolean
    func asArray -> Array<T>
    func times: Function<Nil> b
    func repeat: Function<T, Nil> b

    override
    func foreach: Function<T, Nil> b
    func filter: Function<T, Boolean> f -> Array<T>
    func filter: Function<T, Boolean> f foreach: Function<T, Nil> b
    func map: Function<T, T> f -> Array<T>
    func |> Function<Interval<T>, Interval<T>> f -> Interval<T>
    func + Iterable<T> other -> Iterable<T>
       // Smalltalk-like injection
    func inject: (T initialValue)
        into: Function<T, T, T> b
        -> T
       // injection method to be used with context object.
       // the initial value is private to injectTo
```

```
    func to: (T max)
        do: (InjectObject<T> injectTo)
        -> T
    func size -> Int
    func first -> T
    func last -> T

    func apply: (String message) -> Dyn
    func .* (String message)
    func .+ (String message) -> Any
    override
    func iterator -> Iterator<T>
    // elided
end



package cyan.lang

interface Iterable<T>
    func iterator -> Iterator<T>
    func foreach: Function<T, Nil>
end

package cyan.lang

abstract object InjectObject<T> extends Function<T, Nil>
    override
    abstract func eval: T
    abstract func result -> T
end
```

Intervals can be used with method `in:` of the basic types:

```
var String s = "";
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 8

Grammar Methods

Many languages support methods with a varying number of parameters. These parameters are usually accessed as an array:

```
  // Java
public void print(String format, Object... args) {
    ...
}
```

This method could be used as

```
out.print("Color %s %f", "red", 33.0);
```

Cyan goes beyond by allowing a varying number of parameter, a varying number of keywords, optional parameters, optional keywords, and much more. The pattern of a message passing can be given by a regular expression. That all is made using metaobject `grammarMethod` whose annotation should be attached to a method. The DSL of the metaobject annotation describes the pattern of possible message passings through a regular expression containing message keywords, types, and regular expression operators. The valid operators are `|` ("or"), `+` (one or more repetitions), `*` (zero or more repetitions), and `?` (optional).

```
package grammar

object Car
    @grammarMethod{*
        (do:
           (on: | off: | left: | right: | move: Int)+
        )
    *}
    func carPlay: Tuple< Any,
                        Array< Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int> >
                  > t -> String {

        var s = "";
        for elem in t f2 {
            type elem
                case Any f1 { s = s ++ "car on " }
                case Any f2 { s = s ++ "car off " }
                case Any f3 { s = s ++ "car left " }
```

```
            case Any f4 { s = s ++ "car right " }
            case Int f5 { s = s ++ "car move($f5) " }
        }
        return s
    }

end
```

The annotation for metaobject `grammarMethod` in this example is attached to method `carPlay:`. The attached method should always take one single parameter whose type is based on the DSL of the annotation. Latter we will describe how to calculate this type. Anyway, It is not necessary to know how to build the parameter type from the DSL. Simply declare the method, `carPlay:` in the example, without the type of the parameter. The metaobject will sign an error and tell you which should be the type of the parameter. Copy and paste this type to your code. There is no restriction on the method return type.

A `Car` object may receive messages that match the regular expression of the DSL of the metaobject annotation `grammarMethod`. The regular expression should be between parentheses. The `|` between the keywords mean "or". The `+` mean "one or more". The the regular expression mean "`do:` followed by zero or more of the following keywords: `on:`, `off:`, `left:`, `right:`, or `move:` (with an `Int` parameter). Then the message passings below are legal:

```
let car = Car();
car do: on:;
car do: on: off:;
car do: on: move: 50 left: move: 20 right: off:;
```

Prototype `Car` does not define methods `on:`, `on: off:`, and `on:move:left:move:right:off:`. Each of the message passings above should cause a compile-time error. They almost do. Before signalling the error the compiler searches for a metaobject whose annotation, attached to the prototype or a method, implements interface

`IActionMethodMissing_dsa`

This interface has a method that returns the code that should replace the message passing. It will be a message `carPlay:` with the appropriate parameter. The three messages of the last example will be replaced by the code that follows. How the parameter is generated will soon be explained.

```
car carPlay: [. Any,
     [ ( Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int>() f1: Any ) ]
   .];
car carPlay: [. Any,
     [ ( Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int>() f1: Any ) ,
       ( Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int>() f2: Any ) ]
   .];

car carPlay: [. Any,
     [ ( Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int>() f1: Any ),
       ( Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int>() f5: 50 ),
       ( Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int>() f3: Any ),
       ( Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int>() f5: 20 ),
       ( Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int>() f4: Any ),
       ( Union<f1, Any, f2, Any, f3, Any, f4, Any, f5, Int>() f2: Any ) ]
   .];
```

A method `add:` that accepts any number of real arguments that are subtypes of type `T` can be declared as

```
package grammar

object GMTest<T>
    @grammarMethod{*
        (add: (T)+ )
    *}
    func addAll: Array<T> args {
        all addAll: args
    }

    func init { all = Array<T>(); }

    override
    func asString -> String = all asString;

    func getAll -> Array<T> = all;

    let Array<T> all

end
```

The compiler will replace `T` inside the DSL of the metaobject annotation `grammarMethod` by the real argument.

```
    let GMTest<Int> ti = GMTest<Int>();
    ti add: 0, 1, 2, 3;
    assert ti getAll == [ 0, 1, 2, 3 ];
```

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 following code would be legal.

```
    let GMTest<Int> ti = GMTest<Int>();
    ti add: ;
    assert ti getAll == Array<Int>();
```

Instead of using keyword `add:` just one time, we may want to use a keyword `each:` before each element added in the array.

```
package grammar

object GMTest<T>
    @grammarMethod{*
        (add: (T)* )
    *}
    @grammarMethod{*
        (each: T)+
    *}
    func addAll: Array<T> args {
        all addAll: args
```

```
    }

    func init { all = Array<T>(); }

    override
    func asString -> String = all asString;

    func getAll -> Array<T> = all;

    let Array<T> all

end
```

Both metaobject annotations are attached to method `addAll:`. This is possible because both demand exactly the same parameter, `Array<T>`. This is not usually the case. Now elements may be added with one or more `each:` keyword:

```
    let GMTest<Int> ti = GMTest<Int>();

    ti add: 0, 1, 2, 3;
    assert ti getAll == [ 0, 1, 2, 3 ];

    ti each: 4 each: 5 each: 6;
    assert ti getAll == [ 0, 1, 2, 3, 4, 5, 6 ];
```

Here we should use `+` because we cannot have zero "`each:  value`" elements. If we used `*` the metaobject would issue the error

```
This regular expression matches an empty input, which is illegal
```

More than one keyword may be repeated as in

```
package grammar

object StringHashTable
    func init { map = HashMap<String, String>(); }

    @grammarMethod{*
        (key: String value: String)+
    *}
    func multKeyValue: Array<Tuple<String, String>> list {
        for t in list {
            map[ t f1 ] = t f2
        }
    }

    func getMap -> IMap<String, String> = map;

    let IMap<String, String> map
end
```

Part "`key: String, value: String`" is represented by `Tuple<key, String, value, String>`. Since there is a plus sign after this part, the whole method takes a parameter of type

```
Array<Tuple<String, String>>
```

An example of use is

```
let ht = StringHashTable();
ht key: "John" value: "Professor"
   key: "Mary" value: "manager"
   key: "Peter" value: "designer";
```

8.1 Matching Message Sends with Methods

A prototype may have one or more methods with an attached `grammarMethod` prototype. It may have other metaobject annotations that implement interface

```
IActionMethodMissing_dsa
```

These annotations may be attached to the prototype, to fields, local variables, etc or not attached to anything. The prototype may inherit from a prototype that has annotations of metaobjects that implement this interface.

The semantic analysis of a message passing starts with the semantic analysis of the receiver and real arguments. Then the compiler searches for an adequate method in the prototype that is the type of message receiver, say `T`, and its superprototypes. How this search is done is discussed elsewhere (page 86).

If no adequate method is done, the compiler puts in a list all metaobject annotations of `T` of metaobjects that implement

```
IActionMethodMissing_dsa
```

This list is ordered according to the textual order in which the annotations appear in the prototype `T`. The first element appears in the smaller line number of the source code of `T`.

Then the compiler calls method

```
dsa_analyzeReplaceKeywordMessage
```

of each metaobject. If two or more of them return a non-`null` value, an error is issued: the call is ambiguous. If one of them returns a non-`null` value, this value is a tuple with the code that should replace the original message passing. For example, in the example

```
ti add: 0, 1, 2, 3;
```

of prototype `GMTest<T>`, the compiler will replace this message passing by

```
ti addAll: [ 0, 1, 2, 3 ];
```

The code returned by the metaobject `grammarMethod`, in this case, is

```
"ti addAll: [ 0, 1, 2, 3 ]"
```

The call to method

```
dsa_analyzeReplaceKeywordMessage
```

of all metaobjects may return null. That is, metaobjects corresponding to metaobject annotations in `T`. In this case, the compiler searches for metaobject annotations of the superprototypes such that the metaobject implement interface

```
IActionMethodMissing_dsa
```

If there is no superprototype an error message is issued.

The `grammarMethod` annotation cannot be attached to `init:` methods because `init:` methods cannot be called by sending messages to expressions.

A grammar method annotation will try to match as much as possible the message with its regular expression. Then the second `add:` keyword of prototype `MyOddArray` will never be used. The second tuple element, an array, will always have size zero.

```
package grammar

object MyOddArray
    @grammarMethod{*
        ( (add: Int)+ (add: Int)* )
    *}
    func addAll: Tuple<Array<Int>, Array<Int>> t {
        all addAll: t f1;
        assert t f2 size == 0;
        all addAll: t f2;
    }

    func init { all = Array<Int>(); }

    override
    func asString -> String = all asString;

    func getAll -> Array<Int> = all;

    let Array<Int> all

end
```

Note that:

(a) there could be other metaobject annotations that may change message passing. A future restriction would be to restrict to public places an annotation of a metaobject that implements interface
`IActionMethodMissing_dsa`
Currently an annnotation internal to a method, for example, is taken into consideration;

(b) interfaces are not searched for. That is, it is legal to attach, to an interface or method signature of an interface, an annotation of a metaobject whose class implements
`IActionMethodMissing_dsa`
But it will not be taken in consideration;

(c) the grammar method metaobject does not do any further anaylysis in the annotation DSL. The in the prototype `MyOddArray`, there will not be any warning that the second `add:` keyword is never used. The only checking is whether the regular expression accepts the empty string;

(d) a method that has a metaobject annotation `grammarMethod` may be overridden in a subprototype. Then the method called may be that of the subprototype. See this example:

```
package grammar

object SubMyOddArray extends MyOddArray
    @grammarMethod{*
        ( (addThis: Int)+ (addOther: Int)* )
```

```
    *}
    override
    func addAll: Tuple<Array<Int>, Array<Int>> t {
        let Array<Int> array = getAll;
        for elem in t f1 {
            array add: elem + 1
        }
    }
end
```

The compiler replaces the message send based on the compile-time type.

```
var MyOddArray array = SubMyOddArray();
  // this is replaced by a call to addAll
  // the method called is that of SubMyOddArray
array add: 0 add: 1;
array println;
assert array asString == "[ 1, 2 ]";
// if uncommented, there would be a compile-time error
// array addThis: 0 addThis: 1;
```

(e) `init:` and `new:` are not allowed as keywords in the regular expression of a grammar method annotation.

8.2 Unions and Optional Keywords

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).

```
package grammar

object EnergyStore
    @grammarMethod{*
        (add: (wattHour: Double | calorie: Double | joule: Double))
    *}
    func addEnergy: Tuple<Any, Union<f1, Double, f2, Double, f3, Double>> t {
        addAmount: t f2
    }

    func addAmount: Union<f1, Double, f2, Double, f3, Double> value {
        type value
            case Double f1 {
                amount = amount + f1*3600.0
            }
            case Double f2 {
                amount = amount + f2*4.1868
            }
            case Double f3 {
                amount = amount + f3;
```

```
        }
    }

        // keeps the amount of energy in joules
    @property var Double amount = 0.0;
end
```

`Any` is the type associated to keywords without parameters such as `add:` of this example. We can use this prototype as

```
var EnergyStore store = EnergyStore new;
store add: wattHour: 5.0;
store add: joule: 10.0;
store add: calorie: 3.0;
store getAmount println;
```

The optional keywords may be repeated using "`+`" (one or more) or "`*`". We used "`+`" in the annotation of `addEnergyList:`.

```
package grammar

object EnergyStore

    @grammarMethod{*
        (add: (wattHour: Double | calorie: Double | joule: Double)+)
    *}
    func addEnergyList: Tuple<Any, Array<Union<f1, Double, f2, Double, f3, Double>>> t {
        for elem in t f2 {
            addAmount: elem
        }
    }

    @grammarMethod{*
        (add: (wattHour: Double | calorie: Double | joule: Double))
    *}
    func addEnergy: Tuple<Any, Union<f1, Double, f2, Double, f3, Double>> t {
        addAmount: t f2
    }

    func addAmount: Union<f1, Double, f2, Double, f3, Double> value {
        type value
            case Double f1 {
                amount = amount + f1*3600.0
            }
            case Double f2 {
                amount = amount + f2*4.1868
            }
            case Double f3 {
                amount = amount + f3;
            }
```

```
    }

        // keeps the amount of energy in joules
    @property var Double amount = 0.0;
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 transformed in a method call to `addEnergyList:`.

As another example, a stub of a prototype `MyFile` could be

```
package grammar

object MyFile
    @grammarMethod{*
        ( open: String (read: | write: ) )
    *}
    func openReadWrite: Tuple<String, Union<f1, Any, f2, Any>> t {
        let String name = t f1;

        type t f2
            case Any f1 {
                "open '$name' for reading" println;
            }
            case Any f2 {
                "open '$name' for writing" println;
            }

    }

end
```

Method `openReadWrite:` is called twice at runtime in this example:

```
        var MyFile myfile = MyFile();
        myfile open: "AAAA" read:;
        myfile open: "BBBB" write:;
```

Optional parts should be enclosed by parentheses and followed by "`?`", as in

```
package grammar

@init(name, age)
object Person

    @grammarMethod{*
```

```
        ( name: String
                (age: Int)? )
    *}
    func set: Tuple<String, Union<some, Int, none, Any>> t {
        self.name = t f1;
        type t f2
            case Int some {
                self.age = some
            }
            case Any none {
            }
    }

    override
    func asString -> String = "Person($name, $age)";

    @property var String name = "";
    @property var Int age = 0;
end
```

The type associated to `(age: Int)?` is
`Union<some, Int, none, Any>`
The type associated to `R?` will be
`Union<some, type of R, none, Any>`
Method `set:` of `Person` is called by the message passings

```
    let Person p = Person("Carolina", 7);
    assert p getName == "Carolina";

      // call set:
    p name: "Carol";
    assert p getName == "Carol";

      // call set:
    p name: "Carolina" age: 7;
    assert p getName == "Carolina";
```

In a grammar annotation, it is possible to use more than one type between parentheses separated by "|":

```
package grammar

object ArrayIS

    @grammarMethod{*
        (add: (Int | String)*)
    *}
    func addMany: Array<Union<Int, String>> unArray {
        for intStr in unArray {
            type intStr
```

```
            case Int elem { array add: elem }
            case String elem { array add: elem }
        }
    }


    override
    func asString -> String {
        let Array<Any> anyArray = Array<Any>();
        for intStr in array {
            type intStr
                case Int elem { anyArray add: elem }
                case String elem { anyArray add: elem }
        }
        return anyArray asString
    }


    @property let Array<Int|String> array = Array<Int|String>();
end
```

Message `add:` with a variable number of `Int` and `String` parameters can be sent to an `ArrayIS` object:

```
let ArrayIS isArray = ArrayIS();
isArray add: 0, "zero", 1, "one", 2, "three";
```

Care must be taken with alternative keywords in the DSL code of a grammar method.

```
package grammar


object ArrayIS

    @grammarMethod{*
        (addElem: Any | addElem: Int | addElem: String)+
    *}
    func addManyElem: Array<Union<f1, Any, f2, Int, f3, String>> unArray {
        for anyIntStr in unArray {
            type anyIntStr
                case Any f1 { "found Any" println }
                case Int f2 { array add: f2 }
                case String f3 { array add: f3 }
        }
    }


    @grammarMethod{*
        (add: (Int | String)*)
    *}
    func addMany: Array<Union<Int, String>> unArray {
```

```
        for intStr in unArray {
            type intStr
                case Int elem { array add: elem }
                case String elem { array add: elem }
        }
    }


    override
    func asString -> String {
        let Array<Any> anyArray = Array<Any>();
        for intStr in array {
            type intStr
                case Int elem { anyArray add: elem }
                case String elem { anyArray add: elem }
        }
        return anyArray asString
    }


    @property let Array<Int|String> array = Array<Int|String>();
end
```

Using this prototype, one can write

```
isArray addElem: 0
        addElem: "zero"
        addElem: Any
        addElem: 0.0
        addElem: 'a'
        addElem: 1;
assert isArray getArray size == 0;
```

No element is inserted in the field `array` of `ArrayIS` because the first keyword of the grammar method,

```
addElem: Any
```

is always chosen.

8.3 Refining the Definition of Grammar Methods

It is time to describe precisely the type of the parameter of a method that has a metaobject annotation `grammarMethod`. The association of regular expressions with types is given by the following table. `T1`, `T2`, ..., `Tn` are types and `R` is part of the signature of the code of the DSL of the metaobject annotation. 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 `T1`, `T2`, and so on. For example, in a list `R R R`, assume that the types associated to the three `R´s` are `T1`, `T2`, and `T3`, respectively. We used `typeof(S)` for the type associated, by this same table, to the grammar element `S`.

rule	type
T1	T1
R R ... R	`Tuple<T1, T2, ..., Tn>`
Id ":" R R ... R	`Tuple<T1, T2, ..., Tn>`
Id ":"	Any
Id ":" T	T, which must be a type
Id ":" "(" T ")" "∗"	`Array<T>`
Id ":" "(" T ")" "+"	`Array<T>`
"(" R ")"	`typeof(R)`
"(" R ")" "∗"	`Array<typeof(R)>`
"(" R ")" "+"	`Array<typeof(R)>`
"(" R ")" "?"	`Union<some, typeof(KeywordUnitSeq), none, Any>`
T1 "|" T2 "|" ... "|" Tn	`Union<f1, T1, f2, T2, ..., fn, Tn>`
R "|" R "|" ... "|" R	`Union<f1, T1, f2, T2, ..., fn, Tn>`

We will give now the precise definition of the type of the parameter of the method based on the grammar of the DSL of the metaobject annotation. It will be used "`typeof(P)`" for the type associated to the grammar production `P`.

The productions will be divided in cases.

KeywordGrammar ::= "(" KeywordUnitSeq ")" "∗"
KeywordGrammar ::= "(" KeywordUnitSeq ")" "+"
`typeof(KeywordGrammar) = Array<typeof(KeywordUnitSeq)>`
KeywordGrammar ::= "(" KeywordUnitSeq ")"
`typeof(KeywordGrammar) = typeof(KeywordUnitSeq)`
KeywordGrammar ::= "(" KeywordUnitSeq ")" "?"
Now `typeof(KeywordGrammar) = Union<some, typeof(KeywordUnitSeq), none, Any>`
KeywordUnitSeq ::= KeywordUnit
`typeof(KeywordUnitSeq) = typeof(KeywordUnit)`

When there are at least two KeywordUnit´s:
KeywordUnitSeq ::= KeywordUnit KeywordUnit { KeywordUnit }
`typeof(KeywordUnitSeq) = Tuple<typeof(KeywordUnit1), ..., typeof(KeywordUnitn)>`

in which `typeof(KeywordUniti)` is the i^th production.

When there are at least two KeywordUnit´s separated by "|"
KeywordUnitSeq ::= KeywordUnit "|" KeywordUnit { "|" KeywordUnit }

```
typeof(KeywordUnitSeq) = Union<f1, typeof(KeywordUnit1), ...,
                               fn, typeof(KeywordUnitn)>
```

in which `typeof(KeywordUnit)` is the i^{th} production.
KeywordUnit ::= SelecGrammarElem
`typeof(KeywordUnit) = typeof(SelecGrammarElem)`

KeywordUnit ::= KeywordGrammar
`typeof(KeywordUnit) = typeof(KeywordGrammar)`
SelecGrammarElem ::= IdColon
`typeof(SelecGrammarElem) = Any`
SelecGrammarElem ::= IdColon Type1, Type2, ... Typen
`typeof(SelecGrammarElem) = Tuple<Type1, Type2, ..., Typen>`
if IdColon is followed by two or more types or
`typeof(SelecGrammarElem) = Type1`
if
SelecGrammarElem ::= IdColon Type1

SelecGrammarElem ::= IdColon "(" Type ")" ("*" | "+")
`typeof(SelecGrammarElem) = Array<typeof(Type)>`
Note that a Type may be an union type. Then the type of
`Int | String`
is `Union<Int, String>`.
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<f1, Int, f2, String>`
`key: Int value: Float`	`Tuple<Int, Float>`
`nameList: (String)* (size: Int)?`	`Tuple<Array<String>, Union<some, Int, none, Any>>`
`coke:`	`Any`
`coke: | guarana:`	`Union<f1, Any, f2, Any>`
`(coke: | guarana:)*`	`Array<Union<f1, Any, f2, Any>>`
`(coke: | guarana:)+`	`Array<Union<f1, Any, f2, Any>>`
`((coke: | guarana:)+)?`	`Union<some, Array<Union<f1, Any, f2, Any>>, none, Any>`
`((coke: | guarana:)?)+`	`Array<Union<some, Union<f1, Any, f2, Any>, none, Any`
`amount: (gas: Float | alcohol: Float)`	`Tuple<Any, Union<f1, Float, f2, Float>>`

It is possible to have an annotation that does not use any regular operator. That is legal

```
@grammarMethod{*
    (format: (String form) print: (String s))
*}
func formatPrint: Tuple<String, String> t {
    ...
}
```

8.4 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;

To further exemplify grammar methods, we will give more examples of them.

```
@init(from, to)
object Edge
    @annot( #f1 ) @property Int from
    @annot( #f2 ) @property Int to
end

@init(numVertices, edgeArray)
object Graph

    @annot( #f1 ) @property Int numVertices Int
    @annot( #f2 ) @property Int edgeArray Array<Edge>
end

object MakeGraph
    @grammarMethod{*
        (numVertices: Int (edge: Int, Int)* )
    *}
    func make: Tuple<Int, Array<Tuple<Int, Int>> t -> Graph {
        let edgeArray = Array<Edge>();
        for elem in t f2 {
            edgeArray add: Edge(elem f1, elem f2);
        }
        return Graph new: t f1, edgeArray;
    }
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.

Flower [Fir12] 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 [Fir12] shows an annimation of this.

A grammar method implementing the above grammar is very easy to do:

```
package grammar

object Camera
    @grammarMethod{*
        (sizeHoriz: Int sizeVert: Int
               positionX: Int positionY: Int
               (move: Int (up: | down: | left: | right:) )+ )
    *}
    func camera:
       Tuple<Int, Int, Int, Int,
            Array<
               Tuple<Int,
                  Union<f1, Any, f2, Any, f3, Any, f4, Any>>>> 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 paralel programming that would implement some commom paralel 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.

Chapter 9

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
    func eval: (Int x) -> Int {
       return x*x;
    }
end
```

For every function the compiler creates a prototype like the above, although one that inherits from yet-to-be-seen prototypes `Function<...>`. 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 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 fields 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`.

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:

```
func aMethod: Int x, Int y -> Int {
   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
   return sqrt: ((Math sqr: x) * (Math sqr: y));
}
```

A Cyan function at runtime 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 };
```

[1] Which of course can easily be removed as the function can return the expression itself.

creates three functions, each of which captures variable `x`.

Functions can be curried; that is, we can supply some of the parameters and get a new functions with the remaining parameters:

```
var Function<Int, Int, Int> mult = { (: Int a, Int b :) ^a*b };
var Function<Int, Int> doubleNum = mult curry: 2;
var Function<Int> six = mult curry: 2, 3;
    // print 6
(doubleNum eval: 3) println;
six eval println; // print 6
```

`doubleNum` is the function

```
{ (: Int b :) ^2*b }
```

A function that takes n parameters has `curry:` methods that take from 1 to n parameters.

9.1 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. For the time being, return statements inside an anonymous function is prohibited in Cyan. But the examples of this section will show what would happen if they are allowed.

We will give examples of these errors. Assume that "`Function<Nil>`" is the type functions that does not take parameters and returns nothing.

```
object Test
    func init {
        function = { }
    }
    func run {
        prepareError;
        makeError;
    }
    func prepareError {
        function = { return };
        return;
    }
    func makeError {
        function eval;
    }
    Function<Nil> function
end
```

Suppose the execution starts at method `run` that calls `prepareError` that stores an anonymous function in field `function`. In `makeError`, the function stored in the field receives message `eval` and statement `return` of this function is executed. This is a return from method `prepareError` that is no longer in the stack. There is a runtime error.

```
object Test
    func run {
        returnFunction eval
    }
    func returnFunction -> Function<Nil> {
        return { return };
    }
end
```

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
    func init {
        function = { ^0 }
    }
    func run {
        prepareError;
        makeError;
    }
    func prepareError {
        var x = 0;
        function = { ^x };
    }
    func makeError {
        Out println: (function eval);
    }
    Function<Int> function
end
```

Suppose method run is called on an object of `Test`. 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
    func run {
        var a1 = 1;
        var Function<Nil> b1;
        if a1 == 1 {
            var a2 = 2;
            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
    func run {
        var a1 = 1;
        var Function< Function<Int> > b1;
        if a1 == 1 {
            var a2 = 2;
            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
    func run {
        var a1 = 1;
        var Function<Int> b1;
        if a1 == 1 {
            var b2 = {
                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
    func run {
        var a1 = 1;
        var Function<Function<Int>> b1;
        if a1 == 1 {
            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
    func run {
        Out println: test
    }
    func 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>`.

Currently the Cyan compiler allows a anonymous function to access any visible variable. There is never a runtime error because the local variables accessed inside a anonymous function are allocated in the heap. Note that fields belong to the `self` object and are always in the heap. The `return` statatement cannot be used inside an anonymous function.

9.2 Functions with Multiple Keywords

Regular functions only have one keyword, which is `eval:` or `eval` (when there is no parameter). It is possible to declare a function with more than one `eval:` keyword. 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
};
```

in which $k_i \geqslant 0$ for each i.

Consider a function with a method composed by `n eval:` keywords, each of them with at least one parameter. The i^{th} `eval:` keyword has k_i parameters. This function inherits from prototype

```
Function<T11, T12, ..., T1k1><T21, T22, ...  T2k2>...<Tn1, Tn2, ...  Tnkn, R>
```

This prototype declares just one abstract method, which is

```
abstract
func eval: ( T11 p11, T12 p12, ..., T1k1 p1k1)
    eval: ( T21 p21, T22 p22, ..., T2k2 p2k2)
           ...
    eval: ( Tn1 pn1, Tn2 pn2, ..., Tnkn pnkn)  -> R
```

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;
```

An `eval:` method can have zero parameters. In this case, `none` is used in the place of the type in the generic prototype. That is, function

```
{ (: eval: Int i, Int j
     eval:
     eval: Char ch :) ^ ch ++ i ++ j }
```

has type

```
Function<Int, Int><none><Char, String>
```

9.3 Methods as Functions

Method `functionForMethod` of `Any` takes a literal string with the name of a method and returns a function that call that method. The name of a unary method is the unary method. The name of a non-unary method is the joining of each keyword followed by its number of parameters, separated by a single white space. Then the names of the methods

```
object Test
    func run2 { ... }
    func run: Array<String> { ... }
    func aa: String s0, Int s1, Char s2
         bb: Int s3, Char s4
         cc: Char s5, String s6 -> String {
         return asString ++ s0 ++ s1 ++ s2 ++ s3 ++ s4 ++ s5 ++ s6
    }

    func * Int n -> Char = n asChar;
    func - -> String = "000";
    ...
end
```

are

```
run2
run:1
aa:3 bb:2 cc:2
*1
-
```

As an example, the code calls the methods of prototype `Test` given above

```
    let Function<Nil> mRun = Test functionForMethod: "run2";
    mRun eval;
```

```
let Function<Array<String>, Nil> mRun1 = Test functionForMethod: "run:1";
mRun1 eval: [ "0", "1", "2" ];

let Function<String, Int, Char><Int, Char><Char, String, String> mabc =
    Test functionForMethod: "aa:3 bb:2 cc:2";
(mabc eval: "0", 1, '2' eval: 3, '4' eval: '5', "6") println;

let Function<Int, Char> mMult = Test functionForMethod: "*1";
(mMult eval: 0) println;

let Function<String> mMinus = Test functionForMethod: "-";
mMinus eval println;
```

The function returned by `functionForMethod:`, when receives an `eval` or `eval:` message, send the original message to the receiver of `functionForMethod:`. For example,

```
mRun1 eval: [ "0", "1", "2" ]
```

sends message `[ run: "0", "1", "2" ]` to object `Test` because `Test` is the receiver of message

```
functionForMethod: "run:1"
```

The metaobject `changeFunctionForMethod` whose annotation is attached to `functionForMethod` changes

```
Test functionForMethod: "run:1"
```

to

```
{ (: Array<String> p0 :) Test run: p0 }
```

This metaobject can only be applied to methods `functionForMethod:` and `functionForMethodWithSelf:`.

Method `functionForMethodWithSelf:` return a function as `functionForMethod:` does but with one difference: there is an additional first parameter whose type is the type of the receiver of the message. In the function returned, the message is sent to this first parameter.

Both methods return a function that works like a **method** of the receiver, but with an important difference. `functionForMethod:` returns what could be called a "*object method*", a method specific to the object that is the receiver. And `functionForMethodWithSelf:` returns a more generic method, one that demands that the receiver be passed as parameter. Let us see how `functionForMethodWithSelf:` works.

```
let t0 = Test();
let t1 = Test();
let Function<Test, Nil> mRunSelf = Test functionForMethodWithSelf: "run2";
mRunSelf eval: t0;
"aaa" println;
let Function<Test><Array<String>, Nil> mRun1Self = Test
    functionForMethodWithSelf: "run:1";
mRun1Self eval: t1 eval: [ "0", "1", "2" ];
"bbb" println;
let Function<Test><String, Int, Char><Int, Char><Char, String, String> mabcSelf
    =
    Test functionForMethodWithSelf: "aa:3 bb:2 cc:2";
(mabcSelf eval: t0 eval: "0", 1, '2' eval: 3, '4' eval: '5', "6") println;
"ccc" println;
let Function<Test><Int, Char> mMultSelf = Test functionForMethodWithSelf: "*1";
```

```
(mMultSelf eval: t1 eval: 0) println;
"ddd" println;
let Function<Test, String> mMinusSelf = Test functionForMethodWithSelf: "-";
(mMinusSelf eval: t1) println;
"eee" println;
```

`mRunSelf` is a function that takes a `Test` as parameter because `run2` is an unary method. This function is returned in a call to `Test` (line 3) but it is called in line 4 on object `t0`. `mRun1Self` takes two parameters, one for each `eval:` keyword. If the literal string argument of `functionForMethodWithSelf:` specifies a non-unary method, the returned function will have an `eval:` keyword taking a single parameter whose type is the type of the receiver of this message. Then

```
Test functionForMethodWithSelf: "run:1"
```

returns

```
{ (: eval: Test myself
     eval: Array<String> p0 :) ^myself run: p0
  }
```

If `Test` had a method

```
func open: String s read: close: -> String
```

the function returned by

```
Test functionForMethod: "open:1 read:0 close:0"
```

would be

```
{ (: eval: String p0
     eval:
     eval: :) ^Test open: p0 read: close:
  }
```

whose type is `Function<String><none><none, String>`.

The function returned by `functionForMethodWithSelf:` would be

```
{ (: eval: Test myself
     eval: String p0
     eval:
     eval: :) ^myself open: p0 read: close:
  }
```

whose type is `Function<Test><String><none><none, String>`.

Using methods as objects is very convenient in creating graphical user interfaces. Listeners can be regular methods. See the example.

```
object MenuItem
    func onMouseClick: (Function<Nil> b) {
        ...
    }
end

object Help
    func show { ... }
    ...
end
```

```
object FileMenu
   func open { ... }
end

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

9.4 Methods of Functions for Decision and Repetition

Object `Function<Boolean>` defines some methods used for decision and iteration statements. The code of these methods is shown below.

```
package cyan.lang

abstract object Function<Boolean>
    abstract func eval -> Boolean
    func whileTrue: (Function<Nil> aFunction) {
       (self eval) ifTrue: {
           aFunction eval;
           self whileTrue: aFunction
       }
    }
    func whileFalse: (Function<Nil> aFunction) {
       (self eval) ifFalse: {
           aFunction eval;
           self whileFalse: aFunction
       }
    }
end
```

An function that does not take any parameters and does not return a value inherits from `Function<Nil>`.

```
package cyan.lang

abstract object Function<Nil>
    abstract func eval
    func loop {
        while true {
            self eval
        }
    }

    func repeatUntil: (Function<Boolean> test) {
        self eval;
```

```
        while ! (test eval) {
            self eval
        }
    }

    @createCatchMethodsForFunctionNil

    @checkCatchParameter
    @grammarMethod{*
        ( (catch: Any)+ (finally: Function<Nil>)? )
    *}
    func catchFinally: Tuple<Array<Any>, Union<some, Function<Nil>, none, Any>> t {
    }

    func hideException {
        {
            self eval
        } catch: { (: CyException e :)
        };
    }
    // other methods

end
```

Method `loop` implements an infinite loop and `repeatUntil:` implements a loop that ends when the function parameter evaluates to `true`. There are many `catch:` methods that are not shown. They are produced by metaobject `createCatchMethodsForFunctionNil`.

9.5 Future Enhancements

This section proposes a future change to anonymous functions that would allow the compiler to allocate local variables in the stack even if they are used inside a function.

Metaobject annotations can to be attached directly to types as in

```
var Char@letter ch;
ch = 'A'; // ok
ch = '0'/ // compile-time error
```

Here a metaobject `letter` is attached to `Char` and controls the type checking of `ch`. This feature will be used with `Functions`. An annotation of metaobject `rf` will be attached to prototype `Function` to give the precise type of a function:

```
func Int test {
    var Int a1 = 0;
    var f1 = {
        ++a1;
    };
    var f2 = { ^0 };
    (f2 eval) println;
    f1 eval;
```

```
    f0 eval;
}
```

The types of the function variables will be:

f1	Function<Nil>@rf(1)
f2	Function<Int>@rf(-1)
f2	Function<Int>

Note that `Function` without the metaobject will be equal to `Function<Int>@rf(-1)`.

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:

```
func 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 `ai` 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, `a1` is external to the function passed as parameter to keyword `ifFalse:` in the previous example (any of the `ifFalse:` keywords). And `a1` and `a2` are external to the function that is argument to the keyword `ifTrue:`.

In the discussion that follows, it is important to remember that Cyan currently does not allow return statements inside anonymous functions. A function is called "function of level `-1`" if it does not access non-constant local variables. This kind of function can access only parameters, variables that are constants, and fields.

By "*access*" a 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 `a1` 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 func test {
```

```
2     var a1 = 1;
3     { var a2 = 2; // start
4        { var a3 = 3;
5          { ++a1 } eval;
6        } eval;
7     } eval // end
8  }
```

The following function, between lines 3-5, is a function of level -1.

```
1  func make: Int n {
2     let Int p = In readInt;
3     var f = { (: Int k :)
4         ^k + p + n;
5     };
6     (f eval: 0) println;
7  }
```

Let v1, v2, ..., vn be the external non-constant *local* variables accessed in a function B — B is a function, not a variable that refers to a function. Fields 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.

```
func 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 9.1 should clarify this point. Ellipses represent functions. A solid arrow from function C to

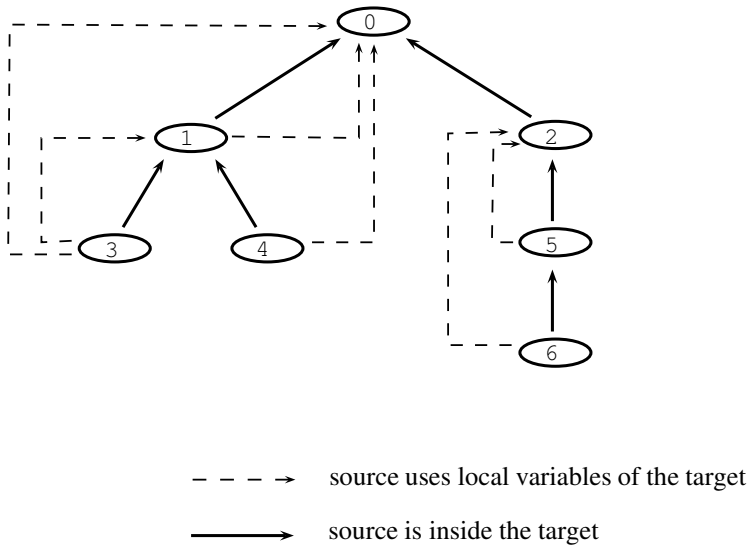

Figure 9.1: Nesting of functions

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).

```
func test {
    var v0 = 0;
    {
        var v1 = 1;
        {
            ++v1;
            ++v0;
            ^3
        } eval;
        var v11 = 2;
        {
            ++v0;
            ^4
        } eval;
        ^1
    } eval;
    {
        var v2 = 2;
        { ^5
            {
                ++v0;
                ++v2;
```

```
            ^6
        } eval;
    } eval;
    ^2
} eval;
return 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.[2] 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 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
func 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;
    return 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`.

[2] X is an ancestor of Y if Y is textually inside X.

Inner functions may be assigned to variables of outer functions without causing runtime errors:

```
func 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. 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 do not access non-constant local variables are called "functions of level -1". This kind of function may access parameters and variables declared as constants.

2. Functions that access at least one external non-constant local variable have their number bl(B) calculated as

   ```
   bl(B) = max{ lev(v1), lev(v2), ..., lev(vn) }
   ```

   `lev(v)` is the level of variable `v`.

   `v1`, `v2`, ..., `vn` are the external *local* non-constant variables accessed in function B. The "*function level*" of B is bl(B).

A function that has a reference to a external non-constant local variable of level `k` is at least a function of level `k`. However, it may be a function of level $\geq$ `k` (if it accesses an external non-constnat variable of a superior level). The use of fields, constants, or parameters is irrelevant to the calculus of the level of a function. Fields are not created with the method or when the function receives message `eval` or `eval:`. And parameters and constants are read only.

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.

```
func 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
          // 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`".

```
func test: (Int p) -> Int {
      // 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 c2 = 0;
        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_4 = { Out println: c2 }; // 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_3 = { ^a3 }; // function of level 3, 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 constants, parameters, and fields. 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 fields, 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 fields, returned from methods and functions, and there are limitations on the assignment of them to local variables. This will soon be explained.

An function of level that takes parameters of types `T1`, `T2`, ..., `Tn` and returns a value of type `R` inherits from prototype

```
abstract object Function<T1, T2, ..., Tn, R>
    abstract func eval: (T1, T2, ..., Tn) -> R
    // other methods --- explained later
end
```

If the function is restricted, a metaobject annotation `rf` should be attached to its type:

```
var Function<Int, String>@rf(2) f;
```

The parameter is the level number.

Every function has its own prototype that inherits from `Function<...>` 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>@rf(k)` in which `k` is a literal integer representing the level of the function.

As another example, the type of variable `add` in

```
var add = { (: Int n :) ^n + 1 };
```

could be `UFunction017`. Since this function inherits from `Function<Int, Int>` we can declare `add` before assigning it a value as

```
var Function<Int, Int> add;
add = { (: Int n :) ^n + 1 };
```

Or as

```
var Function<Int, Int>@rf(-1) add;
add = { (: Int n :) ^n + 1 };
```

Since this function does not access any local variable, its level is `-1`.

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 them a special kind of object, one that has only primitive methods. However, all prototypes that extend prototype `Function` can be passed as an argument to a method that expect a `Function` as a real argument. 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 Function<Int, Nil>
    func init { sum = 0 }
```

```
    @property var Int sum

    func 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 getSum;
```

9.5.1 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 `Function<..., R>` or `Function<..., R>@rf(-1)`. A field can have type `Function<..., R>`;

(b) fields cannot have type `Function<T1, ... Tn, R>@rf(k)` with `k` greater than `-1`;

(c) methods and functions cannot have `Function<T1, ... Tn, R>@rf(k)` as the return type if `k` is greater than `-1`;

(d) a variable `r` declared at level `k` whose type is `Function<T1, ... Tn, R>@rf(kp)` may receive in assignments:

- a variable `s` of level `m` if $m \leq k$ and the type of `s` is `Function<T1, ... Tn, R>@rf(mp)` or one of its subtypes, including `Function<T1, ... Tn, R>@rf(-1)`;
- an r-function of level `m` if $m \leq k$ and this r-function extends prototype `Function<T1, ... Tn, R>@rf(m)`;
- an u-function that extends prototype `Function<T1, ... Tn, R>@rf(-1)`;

(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 subprototype 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. But that is inefficient to say the least.

Based on the rules for type checking functions, one can conclude that:

(a) fields can be referenced by both u-functions and r-functions;

(b) the restriction "methods and functions can have `Function<T1, ... Tn, R>` as the return type" (but not `Function<T1, ... Tn, R>@rf(k)` with $k \geqslant 0$ 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;

(c) since parameters are read-only, it is not possible to assign a variable or function to any of them;

(d) both r-functions and u-functions can access fields since their use do not cause any problems — fields belong to objects allocated in the heap, a memory space separated from the stack. Then it is legal to return a function that accesses a field or to assign such a function to any `UFunction` variable:

```
@init(name, age)
object Person
    func init { }
    private func functionCompare -> UFunction<Person, Boolean> {
        return { (: Person p :) ^age > (p getAge) }
    }
    @property String name = "noname";
    @property Int age = 99;
end
...
var myself = Person new;
   // methods setName: String and setAge: Int are automatically created
myself setName: "José";
myself setAge: 14;
if (Person functionCompare) eval: myself {
    Out println: "Person is older than José";
}
```

(e) the type of a field or return method value cannot be an r-function. But it can be an u-function. Therefore there will never be a field 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;

(f) 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;

(g) the generic prototype `Array<T>` declares a field of type `T`. Therefore the generic array instantiation `Array<Function<T1, ... Tn, R>>` causes a compile-time error — r-functions cannot be types of fields. 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 = sum + it },
                  { (: Float it :) prod = prod*it },
                  { (: Float it :) sumSqr = sumSqr + it*it } ];
```

Of course, this restriction applies to a version of Cyan that uses the functions as defined in this section. The current version of Cyan allows this array.

The rules for checking the use of r-functions are embodied in metaobject `rf`. The compiler passes the control to this metaobject when type checking r-functions. It then implements the above rules.

9.5.2 Examples

In this example, an r-function is passed as a parameter. There is no runtime error.

```
object A
   func aMethod {
       var Int x;
       x = In readInt;
       Out println: (anotherMethod: { (: Int y :) ^y + x });
   }
   func anotherMethod: (Function<Int, Int> b) ) -> Int {
       return yetAnotherMethod: b;
   }
   func yetAnotherMethod: (Function<Int, Int> b) -> Int {
       return 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
   func test {
       { var n = 0;
            // function passed as parameter. The
            // real argument has type Any
         do: { ++n }
       } eval;
       makeError
   }
   func do: (Any any) {
       self.any = any
   }
   func makeError {
          // access to local variable n
          // that no longer exists
       any ?eval
   }
   Any any
end
```

9.5.3 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, ..., 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

$B_n \subset B_{n-1} \subset \ldots \subset B_1 \subset B_0$

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}, ..., v_{i_k} of functions B_{i_1}, B_{i_2}, ..., B_{i_k} with $i_1 < i_2 < \ldots i_{k-1} < i_k$. 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}, ..., 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

$j < i_1$
$j < i_2$
...
$j < i_{k-1}$
$j < i_k$

Since $i_1 < i_2 < \ldots i_{k-1} < i_k$, 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 9.5.1 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 fields (not allowed), and so on.

9.5.4 Adding Methods to Objects

This sections describes a possible future feature of Cyan, the possibility of adding methods to an object.

We add a grammar method to prototype `Any` (Section 4.15) for dynamically adding methods to prototypes. It is necessary to specify each keyword, the types of all parameters, the return value type, and the method body. This grammar method has the signature

```
func (addMethod:
            (keyword: String ( param: (Any)+ )?
              )+
             (returnType: Any)?
             body: Any)
```

Suppose we want to add a `print` method dynamically to prototype `Box`:

```
object Box
    func get -> Int { return value }
    func set: (Int other) { value = other }
    var 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

```
func 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:
    keyword: #print
    body: { Out println: get };
```

However, there is a problem here: it is used `get` in the function that is parameter to keyword `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:
    keyword: #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` (but using "`self`"), external parameters, and external constant local variables. For each parameter or constant the function declares a variable with the same type and name. At the function creation, the values of the external parameters and constants are copied to these function variables. "`super`" cannot be used inside a context function. All restrictions given above apply to regular function nested inside a context function.

```
var cf = { (: Person self, Int n :)
    var f = { get print // error: should use "self get"
        super println; // error: 'super' cannot be used in a context function
    };
    n println;
};
```

Methods of the current object can be accessed by means of a local variable:

```
let mySelf = self;
var b = {
     (: Any self :)
```

```
   Out println: (myself age)
   };
```

Fields 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:
    keyword: #print
    body: { (: Box self :) Out println: (self get) };
```

The "`self`" before "`get`" is mandatory. 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 keyword `body:` matches the keywords, parameters, and return value.

```
   // error: function with parameter, keyword without one
Box addMethod:
    keyword: #print
    body: { (: Box self, Int n :) Out println: n };
   // error: function has no Int parameter
   // and return value should be Int
Box addMethod:
    keyword: #add
    param: Int
    returnType: Int
    body: { (: Box self -> String :) ^(self 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>
    func 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 or there is such a statement but the type of `expr` is `Nil`.

The compiler creates a *context object*[3] 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>

    func 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

```
let Int i = 0;
var b = { (: Box self :) Out println: (i + (self get)) };
(b bindToFunction: Box) eval;
```

the compiler creates a regular object

```
object ContextObject001(Int i)
          implements ContextFunction<Box, Nil>

    func bindToFunction: Box -> UFunction<Nil>
        return {
                Out println: (i + (newSelf get));
            }
    }
end
```

And

```
var b = { (: Box self :) Out println: (i + (self get)) };
```

becomes

```
var b = ContextObject001(i);
```

As another example, consider

```
var Person p = Person("Carol", 5);
let Int otherAge = 8;

var cf = { (: Person self, Int age2 -> Boolean :)
     self name println;
```

[3] Chapter 10 define context objects, which are a generalization of functions.

```
    self name prototypeName println;
    ^ (self age) == age2 && (self age) == otherAge;
};
```

The compiler generates, for this context object, the following prototype:

```
object CFun_1__(Int otherAge)
   implements ContextFunction<Person, Int, Boolean>
   func bindToFunction: (Person newSelf__) -> UFunction<Int, Boolean> {
      return { (: Int age2 -> Boolean :)
          newSelf__ name println;
          newSelf__ name prototypeName println;
          ^ (newSelf__ age) == age2 && (newSelf__ age) == otherAge;
      }
   }
end
```

A context function with multiple keywords is a context function with multiple `eval:` keywords:

```
{ (: S self, eval: T11 t11, ... T1n t1n eval: T21 t21, ... T2m t2m,
      ... eval: ... Tkp tkp -> R :)
      ...
}
```

The type of this context function is

```
interface ContextFunction<S, T11, ..., T1n><T21, ... T2m>...<Tk1, ... Tkp, R>
    func bindToFunction: S -> UFunction<T11, ..., T1n><T21, ... T2m>...<Tk1, ... Tkp, R>
end
```

The `UFunction` returned by `bindToFunction:` is defined in Section 9.2. The type of a context function with multiple keywords 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 keyword. In a call

```
obj addMethod:
    keyword: sel
    param: T1 param: T2 ... param: Tn
    returnType: R
    body: expr
```

metaobject `checkAddMethod` checks whether:

(a) the parameters to all keywords of `addMethod: ... body:` are literals;

(b) the keyword `sel` is a valid method name;

(c) the keyword `sel` ends with ":" if `n > 0`;

(d) the keyword `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 more examples of use of *context functions*.

```
var myContextFunction = { (: Box self, Int p -> Int :) ^(self 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 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:
    keyword: #print
    body: { (: Box self :)
        Out println: "value = ", self get;
    };
Box addMethod:
    keyword: #print
    body: { (: Box self :) Out println: (self 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:
    keyword: #returnSum
    param: Int
    returnType: Int
    body: { (: Box self, Int p -> Int :) ^(self get) + p };
```

The metaobject whose annotation is attached to this grammar method checks whether the number of keywords (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
    @property Int color
    abstract func 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:
          keyword: elem
          body: { (: Shape self :) 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 subprototype of Shape
var Square sqr = Square new;
...
sqr ?black;
  // draws in black
sqr draw;
```

Assume that `draw` of subprototypes 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

```
func blue { color: ff_Hex }
```

Regular objects may be used as parameters to keyword "`body:`".

```
Box addMethod:
    keyword: #print
    body: PrintBox;
```

`PrintBox` is a regular prototype.

```
object PrintBox
       implements ContextFunction<Box, Nil>
```

```
    func bindToFunction: (Box newSelf) -> UFunction<Nil> {
        return { Out println: (newSelf get) }
    }
end
```

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>
     func at: Int -> Int
     func at: Int put: T
     func size -> Int
end
```

A context object used to add a method to an object could have more methods than just `bindToFunction:`.

```
object PrintFormatedBox
         implements ContextFunction<Box, Nil>

    func 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 func format: (Int n) -> String {
         var strn = (n asString);
         return ("0000000000" trim: (10 - strn size)) ++ strn
    }
end
```

After `PrintFormatedBox` is added to `Box` as in

```
Box addMethod:
    keyword: #print
    body: PrintFormatedBox;
```

the `print` method puts zeros before the printed number, if necessary. `format:` is a method that can only be used by the method `print` 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
     func get -> Int { return value }
     func set: (Int other) { value = other }
     var Int value = 0;
```

```
end

...
Box set: 0;
Box addMethod:
    keyword: #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 them 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:
    keyword: #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.

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?".

Chapter 10

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 = 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 = sum + (self calc: x) };
```

Although functions are tremendously useful, they cannot be reused because they are *literal* objects. A function that accesses local and fields 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>
    override
    func eval: (Int x) {
       sum = sum + x
    }
end
```

defines method `eval:` and uses a free `Int` variable `sum` which is binded in the object creation.

```
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.

A context object cannot define `init`, `init:`, or `clone` methods. The only way of creating a context object is by using a `new:` method created by the compiler.

When the type of a context object parameter is preceded by `&`, the real argument should be a local variable or field. It cannot be a parameter of the current method. Context objects can be inherited. In the code that follows, `Manager` extends `Employee`. In `Employee`, the context parameters `name` and `age` are not transformed into fields because they are passed as parameters to the constructor of `Person`. The same happens with all context parameters of `Manager`.

```
package people

open
object Person
    func init: String name, Int age {
        self._name = name;
        self._age = age
    }

    func name: String name age: Int age {
        self._name = name;
        self._age = age;
    }

    @property var String _name
    @property var Int _age
end

open
object Employee(String name, Int age, Int salary, String companyName,
         String &outp, Int &sum)
         extends Person(name, age)

    func getSalary -> Int = salary;
    func getCompanyName -> String = companyName;

    func doSum {
        outp = outp ++ name;
        sum = sum + salary
    }
end

object Manager(String name, Int age, Int salary, String companyName,
              String &outp, Int &sum)
       extends Employee(name, age, salary, companyName, outp, sum)

    func getSectionName -> String = sectionName;
    func setSectionName: String sn { sectionName = sn }

    var String sectionName = "";
end
```

10.1 Passing Parameters by Copy

When the & do not precede the parameter of a *context object*, a copy of the real argument is made when creating the object. Just like in the creation of a regular object.

```
object DoNotSum(Int sum)
   func 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 that are not preceded by & will be called "*copy parameters*". Parameters 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 fatorial: 5);
```

Therefore, method parameters can be real arguments to `DoNotSum`.

10.2 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)
    func value -> T { return v }
    func 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 keyword area: were by reference
    func 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
    func 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 a field 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, &amp;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;
}
```

10.3 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>

    func init { }
    func new: (Int &sum) -> Sum {
        var newSum = Sum new;
        newSum bind: sum;
        return newSum
    }
    public bind: (Int &sum) {
        self.sum = sum
    }
    Int &sum
    override
    func eval: (Int x) {
       sum = 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 a field 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.

10.4 More Examples

The example of trees of page 69 can be made even more compact with context objects:

```
open
object Tree
end

object BinTree(@property Tree left, @property Int value, @property Tree right) extends
    Tree
end
```

```
object No(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 fields `left`, `value`, and `right`:

```
object BinTree extends Tree
    func init { }
    func new: (Tree left, Int value, Tree right) -> BinTree {
        var newObj = BinTree new;
        newObj bind: left, value, right;
        return newObj
    }
    func bind: (Tree left, Int value, Tree right) {
        self.left = left;
        self.value = value;
        self.right = right;
    }
    @property Tree left
    @property Tree value
    @property 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 a field that is a reference (a C-language pointer) to the field of the `Car` that keeps the color. Since Cyan does not have C-like pointers, we can use context objects.

```
object CarDoor(Int &color)
    func getColor -> { return color }
    func setColor: Int newColor { color = newColor }
    ...
end

object Car
    func init: Int aColor {
        _color = aColor;
        _leftDoor = CarDoor(_color);
        _rightDoor = CarDoor(_color);
    }
    func color: (Int newColor) { _color = newColor }
    func color -> Int = _color;
    var Int _color
    @property CarDoor _leftDoor
    @property CarDoor _rightDoor
end
```

```
...
Car color: 255;
   // prints "color = 255"
Out println: "color = ", Car leftDoor getColor;

(Car rightDoor) setColor: 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 keyword `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>
    override
    func eval: (T elem) {
        total = total + elem
    }
    override
    func 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`.

10.5 Future Enhancements

This Section describes future enhancements to context objects.

10.5.1 Type Checking Context Objects

Context objects will be type-checked as functions will be. See Section 9.5.

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 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, fields, 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
    func run {
        [ 1, 2, 3 ] foreach: makeError
    }
    func 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 mixining 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 a field 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 (none or `&`) as the superprototype. `x` should precede the parameters defined only in the subprototype. After the keyword "`extends`" there should appear the superprototype with its parameters.

```
open
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 subprototypes need not to obey the rules given above. The subprototype 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>
    override
    func 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);
assert: (floatSum == 5);
assert: (intSum == 6);
```

10.5.2 Adding Context Objects to Prototypes

Section 9.5.4 (Future Enhancements) explain how to use the `addMethod: ...` grammar method of `Any` to add methods to a prototype.

```
    func (addMethod:
                (keyword: 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
    func addDoorColor {
        leftDoor addMethod:
            keyword: #getColor
            returnType: Int
            body: GetColor(color);
        leftDoor addMethod:
            keyword: #setColor
            param: Int
            body: SetColor(color);
    }
    ...
    public Door leftDoor, rightDoor
    Int color
end

object GetColor(Int &color)
```

```
      implements ContextFunction<Door, Int>

   func bindToFunction: (Door newSelf) -> UFunction<Int> {
       return { ^color }
   }
end

object SetColor(Int &color)
      implements ContextFunction<Door, Int, Nil>

   func 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.

Chapter 11

The Exception Handling System

The exception handling system of Cyan has underwent big changes. Now it is a regular language statement instead of a message passing. Soon this manual will be updated.

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. Tt 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 [Gui04] [Gui06] 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  try
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 statement `throw` as shown in line 7:

```
throw ExceptionZero
```

`ExceptionZero` is a prototype that inherits from `CyException`, the superprototype of all exception objects. Since this exception does not demand any useful additional information, the prototype does not have any fields:

```
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, the `try-catch-finally` statement catches the exceptions thrown during the execution of the statements between `try` and the first `catch` *clause* (or `finally`, if there is no `catch`). 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 expressions of the `catch` clauses. 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` clause. Since

```
{ (: ExceptionRead e :) Out println: "Error reading file" }
```

can accept an `ExceptionRead` object, message `eval` is sent to this function with the thrown exception as argument. After that, the `finally` statements are executed and the execution continues in the statement after `try-catch-finally`.

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` clauses and an optional `finally` clause. Every `catch` 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 the expression following a `catch` be a function or be a subprototype of any function.

11.1 Using Regular Objects to Treat Exceptions

Each `catch:` keyword may receive as argument an object that has more than one `eval:` method.

```
object ExceptionCatchFile
    overload
    func eval: (ExceptionOpen e) { Out println "Error opening file" }
    func eval: (ExceptionRead e) { Out println "Error reading file" }
    func 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 in the next example.

```
var Array<Char> charArray;
var f = MyFile new: "input.txt";
try
   f open;
```

```
   charArray = f readCharArray;
   if charArray size == 0 {
       throw ExceptionZero
   }
catch ExceptionCatchFile
finally {
   f close
}
```

When an exception is thrown, the runtime system starts a search for an `eval:` method (a handler) in the nearest `catch` clause, which is `ExceptionCatchFile`. Supposing that there was a read error, the correct `eval:` method should accept a `ExceptionRead` object as argument. 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`,

```
        func eval: (ExceptionRead e)   { Out println "Error reading file" }
```

is found and called. After that, the statements of the `finally` clause are executed and computation continues after the `try-catch-finally` statement.

11.2 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
    func eval: (CyException e) {
        /* prints the stack of called methods and ends the program
        */
    }
    ...
end
```

Maybe we may will add a `finally:` keyword to the `catch:` message allowing some code to be executed before the program ends.

Let us now explain what happens conceptually when an exception is thrown and caught. The implementation need not to be as described next.

When a message with at least one `catch:` keyword 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.

Consider the catch objects[1] and the example that follow. The prototypes are show as if they were in a single file.

```
   // number < 0, == 0, > 1000, or even
open
object ExceptionNum extends CyException
end

   // when the number is == 0
object ExceptionZero extends ExceptionNum
end

   // when the number is < 0
object ExceptionNeg extends ExceptionNum
end

   // when the number is > 1000
object ExceptionBig extends ExceptionNum
end

   // when the number is even
object ExceptionEven extends ExceptionNum
end

   // when the number is 5
object ExceptionFive extends ExceptionNum
end


object CatchZeroBig
    overload
    func eval: (ExceptionZero e) {
        Out println: "zero number";
    }
    func eval: (ExceptionBig e) {
        Out println: "big number";
    }
end
```

[1] Objects with `eval:` methods that treat exceptions.

```
object CatchNeg
    func eval: (ExceptionNeg e) {
        Out println: "negative number";
    }
end

object CatchEven
    func eval: (ExceptionEven e) {
        Out println: "even number";
    }
end

object CatchNum
    func eval: (ExceptionNum e) {
        Out println: "number < 0, == 0, > 1000, or even";
    }
end

object Program
    let Int MaxN = 1000;

    func 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 func process: (Int n) {
        { // 3
           check: n;
           if n > MaxN {
               throw: ExceptionBig
           }
        } catch: CatchNeg
        // 6
    }
    private func 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 subprototype of `ExceptionZero`. If it is not, it would test whether object `exc` is a subprototype 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 transfered 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>` has a method

```
@checkCatchParameter
func ((catch: Any)+ finally: Function<Nil>) t {
    ...
}
```

responsible for catching exceptions. The metaobject `checkCatchParameter`, whose annotation is attached to this method, checks whether each parameter to a `catch:` keyword has at least one `eval:` method, each of them accepting one parameter whose type is subprototype of `CyException`.

11.3 Other Methods and Keywords for Exception Treatment

Functions of type `Function<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.

Keywords `retry` or `retry:` may be used after all `catch:` keywords in order to call the function again if an exception was caught by any object that is argument to any of the `catch:` keywords. If keyword `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

```
    func 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:` keywords. If any exception is thrown in the function, the `eval` method of the argument to `retry:` is called and the function is called again.

```
   // 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"
  };
```

Keyword `tryWhileTrue:` may be put after the `catch:` keywords 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:` keywords are optional. Keyword `tryWhileFalse:` is similar to `tryWhileTrue`.

Prototype `CatchIgnore` could be used instead of `CatchAll`. This generic prototype ignores the exceptions that are parameters to it. Any number of exceptions can be used. An instantiation of this prototype with parameter `ExceptionConnectFail` would be something like

```
object CatchIgnore<ExceptionConnectFail>
    func eval: ExceptionConnectFail e { }
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 Function<Boolean>
    func eval -> Boolean {
        --numTries;
        return numTries > 0;
    }
end

...
{
      // may throw an exception ExceptionConnectFail
   channel connect;
} tryWhileTrue: Times(5);
```

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"
```

```
};
```

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
    func getInfo -> CySymbol = #retry;
end

object Test
    func test {
        {
            connectToServer;
            buildSomething
        } catch: CatchStragegy
    }
    func 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:

```
    func connectToServer {
        {
           var Boolean fail = true;
           ...
              // if connection to server failed, signal
              // an exception
           if fail {
```

```
            exception eval: ExceptionConnection
        }
    } catch: CatchIgnore<ExceptionConnection>
}
```

11.4 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 [Gui04]:

```
private func 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`[2] is

```
interface EvalZeroNegEven
```

[2]Note that currently Cyan does not support the declaration of overloaded methods in interfaces.

```
    func eval: ExceptionZero
    func eval: ExceptionNeg
    func 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>`:

```
    func 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
    func 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
    func eval: ExceptionOpen
    func 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

```
    func 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.

11.5 Synergy between the EHS and Generic Prototypes

Package `cyan.lang` defines two generic prototypes that accept any number of prototype parameters: `CatchExit` and `CatchWarning`. The first one is used to catch exceptions and end the program. The later just issues a warning message. If they had just one parameter, they would be as shown below.

```
object CatchExit<T>
    func eval: (T e) {
       Out println: "Fatal error: exception " ++ T prototypeName ++
          " was thrown";
       System exit
    }
end

object CatchWarning<T>
    func eval: (T e) {
       Out println: "Exception " ++ T prototypeName ++ " was thrown"
    }
end
```

These prototypes can be used to exit the program for some exceptions or just issue a message for others.

```
...

{
  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. The `eval:` methods of this object do nothing. The definition of `CatchIgnore` with two parameters *would be*

```
object CatchIgnore<T1, T2>
    func eval: T1 e1 { }
    func eval: T2 e2 { }
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. That is what prototype `ExceptionConverter` does. This prototype is defined in package `cyan.lang` and accepts any number of even parameters. With two parameters it would be equivalent to:

```
object ExceptionConverter<Source, Target>
    func eval: (Source e) {
        throw: Target()
    }
end
```

In the example that follows, when an exception `ExceptionNegNum` is thrown, a `catch:` method captures it and throws a new exception from prototype `ExceptionOutOfLimits`.

```
...
{
  if i < 0 { throw: ExceptionNegNum };
  s = v[i];
  s println;
} catch: ExceptionConverter<ExceptionNegNum, ExceptionOutOfLimits>;
```

Another common pattern of exception treatment is to encapsulate exceptions in an exception container. This is what does generic prototype `ExceptionEncapsulator`. It takes any number of parameters (but at least two). The last one should be the container. `ExceptionEncapsulator` with two parameters would be:

```
object ExceptionEncapsulator<Item, Container>
    func eval: (Item e) {
        throw: Container(e)
    }
```

```
end
```

This prototype could be used as in this example.

```
...
{
  if i < 0 { throw: ExceptionNegNum(i) };
  s = v[i];
  s println;
} catch: ExceptionEncapsulator<ExceptionNegNum, ExceptionArithmetic>;
```

Whenever `ExceptionNegNum` is thrown in the function, it is packed into an exception of `ExceptionArithmetic` and thrown again.

11.6 More Examples of Exception Handling

One can design a `MyFile` prototype in which the error treatment would be passed as parameter:

```
object MyFile
    func new: (String filename) { ... }
    func 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` of package `cyan.lang` has an `init:` method that throws the exception that is its parameter.

```
object Throw
    func init: CyException e {
        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` catchs all exceptions. It prints a message specific to the exception thrown and prints the stack of called methods:

```
object CatchWithMessage
    func 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
    func 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 [Gui13] 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 subprototypes 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
    func init {
        table = [
            Any -> Any, // just to set the type of the table
            ExceptionZero -> CatchWarning<ExceptionZero>,
            ExceptionNeg -> CatchAll,
            ExceptionBig -> { (: ExceptionBig e :)
                Out println: "Number " ++ e number ++ " is too big"
                },
```

```
            ExceptionNum -> CatchNum

            ];
    }
    func eval: (CyException e) {
        type table[e prototype]
            case Any any {
                any ?eval: e
            }
            case Nil nil {
                throw: ExceptionStr("Exception " ++
                   (e prototypeName) ++ " is not supported " ++
                   "by table")
            }
    }
    IMap<Any, 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:` keyword as in the example that follows.

```
object ExceptionTriangle(public Double a, public Double b, public Double c)
end

object CatchTriangle
    func 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` of package `cyan.lang` is used as a generic exception which holds a string message.

```
package cyan.lang

object ExceptionStr(String _message) extends CyException

    public func eval: ExceptionStr e {
        Out println: (e message);
    }

    func message -> String = _message;

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 12

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 “CompilationUnit”. ScriptCyan programs are produced by rule “ScriptCompilationUnit”.

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 42 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 }
```

```
CompilationUnit       ::= PackageDec ImportDec { AnnotList ProgramUnit }
ScriptCompilationUnit ::= [ ImportDec ] ( StatementList | { SlotDec } )
PackageDec            :: "package" QualifId [ ";" ]
ImportDec             ::= { "import" IdList [ ";" ] }
ProgramUnit           ::= [ QualifProgUnit ] ( ObjectDec | InterfaceDec )
QualifProtec          ::= "private" | "public" | "protected"
QualifProgUnit        ::= "public" | "package"
AnnotList             ::= { Annotation }
Annotation            ::= "@" Id
                          [ "(" ExprLiteral [ "," ExprLiteral ] ")" ]
                          [ LeftCharString TEXT RightCharString ]
ObjectDec             ::= [ "abstract" | "final" ]
                          "object" Id { TemplateDec }
                          [ ContextDec ]
                          [ "extends" Type [ "(" IdList ")" ] ]
                          [ "implements" TypeList ]
                          { SlotDec }
                          "end"
TemplateDec           ::= "<" TemplateVarDecList ">"
TemplateVarDecList    ::= TemplateVarDec { "," TemplateVarDec }
TemplateVarDec        ::= Id [ "+" ]
ContextDec            ::= "(" CtxtObjParamDec { "," CtxtObjParamDec }")"
CtxtObjParamDec       ::= AnnotList [ "public" | "protected" | "private" ] Type
                          [ "&" ] Id
Type                  ::= SingleType { "|" SingleType }
SingleType            ::= QualifId { "<" TypeList ">" } | BasicType |
                          "typeof" "(" QualifId [ "<" TypeList ">" ] ")"
```

```
BasicType            ::= [ "cyan.lang" ] BasicTypeNoPackage
BasicTypeNoPackage   ::= "Byte" | "Short" | "Int" | "Long" |
                         "Float" | "Double" | "Char" | "Boolean"
SlotDec              ::= AnnotList QualifProtec ( ObjectVariableDec
                         | MethodDec )
ConstDec             ::= [ "shared" ] "let" Type Id "=" Expr [ ";" ]
MethodDec            ::= [ "final" ] [ "override" ] [ "abstract" ]
                         [ "overload" ] "func" MethodSigDec
                         ( MethodBody | "=" Expr ";" )
MethodSigDec         ::= ( MetSigNonGrammar | MetSigUnary |
                         MetSigOperator ) [ "->" Type ]
MetSigNonGrammar     ::= { SelecWithParam }+
MetSigUnary          ::= Id
MetSigOperator       ::= UnaryOp | BinaryOp ( ParamDec | "(" ParamDec ")" )
SelecWithParam       ::= IdColon |
                         [ "[]" ] IdColon ParamList
TypeOrParamList      ::= TypeList | ParamList
TypeList             ::= Type { "," Type }
ParamList            ::= ParamDec { "," ParamDec } |
                         "(" ParamDec { "," ParamDec } ")"
ParamDec             ::= [ Type ] Id
MethodBody           ::= "{" StatementList "}"
ObjectVariableDec    ::= [ "shared" ] "var" Type Id { "," Id } [ ";" ] |
                         [ "shared" ] "var" Type Id [ "=" Expr ] [ ";" ] |
                         [ "shared" ] [ "let" ] Type Id [ "=" Expr ] [ ";" ] |
                         [ "shared" ] [ "let" ] Type Id [ "," Id ] [ ";" ]
FunctionDec          ::= "" [ "(:" FuncSignature ":)" ] StatementList ""
FuncSignatureRet     ::= FuncSignature [ "->" Type ]
FuncSignature        ::= ( Type Id | Type "self" ) { "," Type Id } [ "->" Type ] |
                         [ Type "self" "," ] IdColon { Type Id } { "," Type Id }
QualifId             ::= Id { "." Id }
IdList               ::= Id { "," Id }
InterfaceDec         ::= "interface" Id [ TemplateDec ] [ "extends" TypeList ]
                         { "func" InterMethSig }
                         "end"
InterMethSig         ::= InterMethSig2 [ "->" Type ]
InterMethSig2        ::= Id |
                         { IdColon [ InterParamDecList ] }+ |
                         UnaryOp |
                         BinaryOp ( SingleInterParamDec | "(" SingleInterParamDec ")" )
InterParamDecList    ::= WithoutParentDecList | WithParentDecList
WithoutParentDecList ::= ParamTypeDecList { "," ParamTypeDecList }
ParamTypeDecList     ::= Type [ Id ]
WithParentDecList    ::= "(" WithoutParentDecList ")"
SingleInterParamDec  ::= Type Id
StatementList        ::= Statement { ";" Statement } | ϵ
Statement            ::= ExprAssign | ReturnStat | VariableDec | Annotation |
                         IfStat | WhileStat | ForStat | NullStat
```

```
                    PlusPlusStat | MinusMinusStat | CastStat |
                    TypeStat | TryStat | ThrowStat
VariableDec         ::= "var" [ Type ] Id [ "=" Expr ] { "," Type Id [ "=" Expr ] } |
                    "let" [ Type ] Id "=" Expr { "," Type Id "=" Expr }
ReturnStat          ::= "return" Expr | "^" Expr
ForStat             ::= "for" Id "in" Expr StatListBracket
IfStat              ::= "if" Expr StatListBracket
                    { "else" "if" Expr StatListBracket }
                    [ "else" StatListBracket ]
CastStat            ::= "cast" [ Type ] Id "=" Expr { [ Type ] Id "=" Expr } StatListBracket
                    [ "else" StatListBracket ]
WhileStat           ::= "while" Expr StatListBracket
TryStat             ::= "try" StatementList { "catch" Expr } [ "finally" StatListBracket ]
ThrowStat           ::= "throw" Expr ";"
TypeStat            ::= "type" Expr { CaseClause } [ "else" StatListBracket ]
CaseClause          ::= "case" Type [ Id ] StatListBracket
StatListBracket     ::= "{" StatementList "}"
NullStat            ::= ";"
PlusPlusStat        ::= "++" Id
MinusMinusStat      ::= "--" Id
ExprAssign          ::= Expr [ Assign ]
Assign              ::= { "," Expr } "=" Expr
Expr                ::= ExprOr [ MSendNonUn ] |
                    [ Annotation ] MSendNonUn |
MSendNonUn          ::= { [ BACKQUOTE ] IdColon [ RealParameters ] }+ |
                    { InterIdColon [ RealParameters ] }+ |
                    { InterDotIdColon [ RealParameters ] }+
BinaryOp            ::= ShiftOp | BitOp | MultOp | AddOp | RelationOp |
                    "~||" | ".." | "..<"
RealParameters      ::= ExprOr { "," ExprOr }
ExprOr              ::= [ Annotation ] ExprXor { "||" ExprXor }
ExprXor             ::= ExprAnd { "~||" ExprAnd }
ExprAnd             ::= ExprEqGt { "&&" ExprEqGt }
ExprEqGt            ::= ExprExc { ("=>" | "=>")  ExprExc }
ExprExc             ::= ExprRel { "&&" ExprRel }
ExprRel             ::= ExprMSendNonUn [ RelationOp ExprMSendNonUn ]
ExprMSendNonUn      :: = MSendNonUn
                    ExprOrGt [ MSendNonUn ]
                    "super" MSendNonUn
ExprOrGt            ::= ExprBPP { "|>" ExprBPP }
ExprBPP             ::= ExprInter { ("++" | "--") 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 ] |
                         "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 13

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) implement some Design Patterns using compile-time metaobjects;

(c) 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;

(d) to use Cyan to implement a lot of small Domain Specific Languages;

(e) to use Cyan to investigate language-oriented programming [War95];

(f) to add parallelism to the language and to design a library for distributed programming. That includes the implementation of patterns for parallel programming;

(g) to design code optimization algorithms for Cyan;

(h) to program the Cyan basic libraries for handling files, data structures, and so on;

Chapter 14

Future Enhancements

Some Cyan features may be changed and others may be added. This is a partial list of them:

1. more methods to intervals:

```
(2..500 but: 100..200, 37 select: Prime) print;
```

   `Prime` here is a context object:

```
object Prime extends Function<Int, Boolean>
    func eval: Int elem -> Boolean {
        return elem prime
    }
end
```

2. intervals with `Floats` and `Doubles`:

```
1.0..5.0 step: 0.01 repeat: { (: Float elem :)
    graphFun plot: elem, (sin: elem);
    };

-1.0..1.0 but: 0.0 step: 0.1 repeat: { (: Float elem :)
    graphFun plot: elem, (sin: elem);
    };
```

   `"-1.0..10 but: 0.0"` is an object of `Interval<Float>`. So is `"-1.0..10 but: 0.0"` and `"-1.0..10 but: -0.1`
   This prototype has methods `step:repeat:`, `but:step:repeat:`, `contains:`, etc (see prototype `Int`).
   Maybe `but:` should be replaced by binary minus (`"-"`).

   With `GraphFun` there could be supplied a context object:

```
object Plot<FunToPlot>(GraphFun graphFun) extends Function<Float, Nil>
    func eval: Float elem {
        graphFun plot: elem, (FunToPlot: elem);
    }
end
```

   A literal object could be aware of the context:

```
var GraphFun graphFun = GraphFun new;
graphFun x: -10, 10 y: -40, 40;
1.0..5.0 step: 0.01 repeat: @plot(sin);
```

`@plot` would produce

```
Plot<sin>(graphFun)
```

It would discover that there is a local variable `graphFun` of the correct type. If there are two, `plot` would sign a compilation error;

3. private generic prototypes, which are currently illegal;
4. `package` qualifier for prototypes and methods. These could only be used in the current package;
5. `typeof` may be legal as a real argument in a generic prototype instantiation:

```
var Int count = 0;
var Stack<typeof(count)> intStack; // ok
```

6. A `finally:` keyword 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: { (: Function<Nil> elem :) elem eval };
  };
```

In some other place:

```
DoomsdayWishList add: { "Good bye!" print };
```

The features given below were removed from Cyan. They may be added later maybe in a different form.

14.1 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 by 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 [Gui98]. 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 func 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
    @property String name
    @property Int age
    override func asString -> String {
        return "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
    @property String name
    @property Int age
    override func asString -> String {
        return "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 [Gui98]. The semantics of both are almost equal, except that Green metaobjects may declare a `interceptAll` method that is not supported by mixin objects (yet).

14.2 Multiple Assignments

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
    func getCenter -> Tuple<Float, Float> {
        return [. x, y .]
    }
    ...
    private Float x, y // center of the circle
    private Float radius
end
...

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

Bibliography

[Bla94] G. Blaschek. *Object-oriented programming with prototypes*. Monographs in Theoretical Computer Science - An Eatcs Series. Springer-Verlag, 1994.

[CYH04] Jien-Tsai Chan, Wuu Yang, and Jing-Wei Huang. Traps in java. *J. Syst. Softw.*, 72(1):33–47, 2004.

[Fir12] Writing your first domain specific language, 2012.

[GJS+06] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in c++. *SIGPLAN Not.*, 41(10):291–310, October 2006.

[GJS+14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[Gui98] José de Oliveira Guimarães. Reflection for statically typed languages. In Eric Jul, editor, *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 440–461. Springer, 1998.

[Gui04] José de Oliveira Guimarães. The green language exception system. *Comput. J.*, 47(6):651–661, 2004.

[Gui06] José de Oliveira Guimarães. The Green language. *Comput. Lang. Syst. Struct.*, 32(4):203–215, December 2006.

[Gui13] José de Oliveira Guimarães. The Green language, May 2013.

[Sei12] Peter Seibel. *Practical Common Lisp*. Apress, Berkely, CA, USA, 1st edition, 2012.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, December 1987.

[War95] M. P. Ward. Language oriented programming. *Software — Concepts and Tools*, 15:147–161, 1995.

Appendix A

The Compiler

The Cyan site, `www.cyan-lang.org`, has a download tab from which file `lib.zip` can be downloaded. Put this file in a directory, say, `C:\Dropbox\Cyan`. Uncompress it resulting in directory

```
C:\Dropbox\Cyan\lib
```

Directory `lib` contains several jar files, the Cyan runtime libraries, the compiler (saci.exe), and the source code of the basic libraries.

Set the system environment variable `CYAN_HOME` to directory `lib`. Set the system environment variable `JAVA_HOME_FOR_CYAN` to the JDK directory of Java 8. Then the value of this variable can be

```
C:\Program Files\Java\jdk1.8.0_241
```

The compiler should be called as follows.

```
saci projectDirectoryOrName compilerOptions
```

The compiler name is `saci.exe`. `projectDirectoryOrName` is the file name of the project or the directory name in which the project is. If `projectDirectoryOrName` is a file name, it should have extension "`.pyan`". Its contents should be as described in Section 2. If `projectDirectoryOrName` is a directory, the compiler will create a project file "`projectDirectoryOrName\project.pyan`". It will consider that the program consists of all directories inside `projectDirectoryOrName`. Each directory given origin to a package. The `.cyan` files inside each directory contain the prototypes of the package.

The easy way of compiling a Cyan program is to put all packages in the same super-directory — see the examples in the Cyan site. Then this directory should be `projectDirectoryOrName`.

`compilerOptions` is a list of options. The valid options are:

- `-noexec`, the Java code produced by saci is compiled by the Java compiler but it is not executed;
- `-nojavac`, the Java compiler is not called for compiling the Java code produced by the Cyan compiler;
- `-args argList`, arguments to the Cyan program. The arguments that follow '-args', argList, will be passed to the Cyan program if it is to be executed. It is not an error to have both options `-noexec` and `-args argList`. Of course, this should be the last option in the command line;
- `-sourcePath aPath` for supplying 'aPath' for the Java compiler. This option can appear any number of times. Each time can be composed by multiple paths, separated by ';'.
- `-cp aPath` for supplying 'aPath' for the Java interpreter. This option can appear any number of times. Each time can be composed by multiple paths, separated by ';';

- `-es filename` for interpreting the remaining arguments as Cyan code. After the interpretation, the compiler exits;
- `-ef filename` for interpreting the statements of file 'filename'. After the interpretation, the compiler exits. Example:
  ```
  saci -ef "C:\Dropbox\tests\first.syan"
  ```
  The recommended file extension is "tyan" which stands for "inTerpreted cYAN".

As an example of calling the compiler, we can have Example:

```
saci "C:\Dropbox\Cyan\cyanTests\simple"
   -cp "C:\Dropbox\Cyan\externalLibs"
   -args 0 "C"
```

The compiler can be also be called programmatically by calling methods `parseSingleSource` and `compileProject` of a `saci.Saci` object — see the source files of the Cyan compiler. These methods are usually called by the IDE. An IDE plugin should create a single object of class `Saci` for each Cyan project. It is important that a single `Saci` object is used because this object will keep information from the last compilation that will speed up the next one.

The IDE can call method `compileProject` to compile the whole project. The parameters to this method correspond to the parameters passed in the command line when calling the compiler. The source files will be read from the file system.

```
public void compileProject( String projectDirectoryOrName,
        String cyanLangDir,
        String javaLibDir,
        boolean exec, boolean callJavac, boolean parseOnly )
```

Before calling this method, the IDE plugin should save all source files associated to this project that are being edited.

To simulate incremental compilation, the compiler should be called whenever the user changes the source code being edited. However, the Cyan compiler goes through a long and complex compilation process and it is not feasible to compile a project or even a single source file after each editing command. However, an IDE plugin can call the Cyan compiler to parse the source being edited several times a second. That will take just a few miliseconds.

To parse a single source file the IDE plugin should call method `parseSingleSource`. Only parsing will be done.

```
public boolean parseSingleSource( String cyanLangDir, String javaLibDir,
        String packageName, String prototypeName, char []sourceCodeToParse,
        String projectDirectoryOrName,
        char []sourceCodeProject,
        boolean loadProjectFromFile )
```

Most of the parameters speak for themselves. `sourceCodeToParse` is the source code of prototype `prototypeName` of package `packageName`. The IDE plugin should not read it from disk. It should be retrieved from the IDE editor to make parsing faster. `projectDirectoryOrName` is the directory of the project or the project file name (ending in "`.pyan`"). The source code of the project is given by `sourceCodeProject`. It is the contents of the project file which is either `projectDirectoryOrName` or `projectDirectoryOrName\project.pyan`.

`packageName`, `prototypeName`, and `sourceCodeToParse` cannot be `null`. If `sourceCodeProject` is `null`, `loadProjectFromFile` is true, or the `Saci` object has not kept the "project object" from the last

compilation, then the project file is reload from disk to memory and compiled, producing a `saci.Project` object (see classes of the Cyan compiler). This object can be retrieved by calling method `getProject()` of `Saci`. Through this object one can get the source code of the project that should be passed to the call to `parseSingleSource` in the next time it is called. The flow of control would be:

```
   // flow, not source code --- you got the idea
Saci aSaci = new Saci();
char []sourceCodeProject = null;
...
aSaci.parseSingleSource(..., sourceCodeProject, true);
   // now the project does exist, if there was not any errors
Project p = aSaci.getProject();
if ( p != null )
    sourceCodeProject = p.getText();
aSaci.parseSingleSource(..., sourceCodeProject, false);
...
```

The IDE plugin should call `parseSingleSource` of `Saci` when the text is changed by adding or removing characters (and after the user pauses typing). Method `eventChangeSourceCodeBeingEdited` of `Saci` should be called by the IDE plugin whenever the source code being edited is changed to a new one. For example, the user was editing a file `Program.cyan` and she changes the focus to file `Test.cyan`. The new file is parsed and old data is discarded.

After the IDE plugin calls `parseSingleSource`, it can retrieve information on the symbols produced by the lexical analyzer. This is made by calling method `getSymbolList()` of `Saci`. This list has size `getSizeSymbolList()` and it can be used to highlight keywords, change the color of strings, etc. `getSymbolList()` returns an object of `Symbol[]`. See package `lexer` of the Cyan compiler for subclasses of `Symbol`. In general, `symbol.getSymbolString()` returns the symbol as a String.

When the mouse is over the source code being edited, the IDE should call method

```
searchCodegAnnotation(int line, int column)
```

or

```
searchCodegAnnotation(int offset)
```

of `Saci`. The parameters of the first method are the line and column of the text over which the mouse pointer is. The parameters of the second method is the offset in the text over which the mouse pointer is. If the mouse is over a Codeg, any of these methods will return an object of `CyanMetaobjectWithAtAnnotation` of package `meta` of the Cyan compiler (or `null` if the mouse is not over a Codeg). This is a class of the AST and the object represents an annotation, the Codeg annotation. This object should be passed to method `eventCodegMenu(Saci, CyanMetaobjectWithAtAnnotation)` of `Saci` that is responsible to call method `getUserInput` of the Codeg. This method should open a menu and accept user input through the mouse and keyboard. After the input is done, button `"Ok"` should be pressed (there should be such a button). If `moAnnotation` is a reference to the Codeg passed as parameter to `eventCodegMenu`, `moAnnotation.getUserInput(...)` returns a byte array that is the information of the Codeg (after showing a menu and getting user input). That is, user input through mouse and keyboard is translated into a byte array that is returned by `getUserInput`. This byte array is written to a special file in disk that is managed by the compiler. This array may be a DSL when converted to text (a String) — or it may be not. The IDE plugin should not call `getUserInput()`. This method is called by the compiler inside `eventCodegMenu()`. The byte array returned by `getUserInput` is passed as the second parameter in the next time this method is called on the same Codeg. In this way the Codeg has the previous information — the previous chosen color of Codeg `color`, for example. In the first time `getUserInput` is called, the second parameter is `null`.

After calling `eventCodegMenu`, the byte array returned by `getUserInput` can be retrieved by the IDE plugin by calling method `getCodegInfo()` of the Codeg annotation object. The flow of control would be like

```
CyanMetaobjectWithAtAnnotation moAnnotation = searchCodegAnnotation(line, column);
if ( moAnnotation != null ) {
    eventCodegMenu(moAnnotation);
    byte []codegInfo = moAnnotation.getCodegInfo();
    ...
}
```

The IDE plugin may show a list of Codegs of the current source file in a window. By clicking in one Codeg of the list, `eventCodegMenu` could be called.

To avoid access to the hard disk, information on the Codegs of the current source file is kept in memory as well as in files. When the information is updated by calling `getUserInput` it is written to disk. But when the information in only read it is read from memory. Then if some external source changes the Codeg files the compiler will not know that. It will continue to use the values that are in memory. A complete recompilation is necessary. When a source code is parsed the compiler reads Codeg information that is in files. The file associated to a Codeg annotation has the name composed by the annotation name and the first parameter to the annotation. It is created in a directory `--prototypeName` of the directory of the project. Only the compiler knows where it keeps these files.

Method `parseSingleSource` read Codeg information from the files in the first time it is called. From the second time on it reads the information from a hash table the compiler keeps with Codeg information of the current source code. Whenever the graphical interface of a Codeg changes the information a disk file is updated. When the Codeg information is just read it is read from the hash table in memory.

It is time to give some low-level details of handling Codegs and symbols.

```
Symbol sym = aSaci.getCodegList().get(0).getFirstSymbol();
```

`sym` is the first symbol of the first codeg of the last source code parsed (if there is one Codeg in the last source code parsed). By "first" we mean the Codeg with smaller (line, column) in the text of the last source code parsed. "`sym.getLineNumber()`" is the line of the Codeg call and "`sym.getColumnNumber()`" is the column of the "`@`" symbol. "`sym.getSymbolString()`" will return "`color`" if the Codeg call is that of the line

```
var Int c = (@color(red));
```

Although the type of `sym` is `Symbol`, the runtime class of it is `lexer.SymbolCyanMetaobjectAnnotation`.

The call

```
aSaci.getCodegList().get(0).getLastSymbol()
```

returns a object describing the symbol `)` in the Codeg call `@color(red)`. Then the IDE plugin have sufficient information to highlight the Codeg call.

Assume method `parseSingleSource` of object referenced by `aSaci` has been called. Then the list of symbols of the current source code is

```
aSaci.getSymbolList()
```

The IDE should color symbol referenced by variable `sym` with color `sym.getColor()`, that returns an `int`. The symbol is in line `sym.getLineNumber()` and column `sym.getColumnNumber()`. Its number of characters is usually

```
sym.getSymbolString().length()
```

In some cases, method `getSymbolString()` does not return the string of characters of the symbol. For example, this happens in a metaobject annotation. Some metaobject annotations may be represented as symbols but `getSymbolString` does not return the whole text of the call.

Metaobjects are used to implement Domain Specific Languages (DSL) that may need specific color schemes.

```
@demands{*
    T in [ "Int", "Short", "Byte", "Char", "Long" ],
    U implements Savable,
    R subtype Person
*}
object Proto<T, U, R>
    ...
end
```

To allow that, a metaobject class may redefine the inherited method

```
ArrayList<Tuple4<Integer, Integer, Integer, Integer>>
getColorList()
```

This method returns a list of tuples. Each tuple has the format

```
(colorNumber, lineNumber, columnNumber, size)
```

It means that the characters starting at `lineNumber` and `columnNumber` till `columnNumber + size` should be highlighted with color `colorNumber`.

The line number is relative to the first character of the metaobject annotation. Any metaobject class may redefine method `getColorList`, including number literals (example: `0101bin`) or string literals (example: `r"[a-z]+0*"`).

The compiler class `saci.IDEPluginHelper` defines a method `getColorList` that returns a list of tuples that can be used for syntax highlighting of the last source file parsed (method `parseSingleSource` should have been called first). `getColorList` takes a `Saci` object as parameter.

After compiling a Cyan program or parsing a Cyan source file, the errors can be retrieved by calling two `Saci` methods: `getProjectErrorList()` and `getCyanErrorList()`. The first method returns a list of errors of class `ProjectError`. A `ProjectError` is an error that occurs outside a Cyan source file. For example, an error in the project file (".pyan"), in the arguments of `parseSingleSource`, and so on.

Method `getCyanErrorList()` return a list of errors of class `UnitError`. Every object of this class describes an error that occurred inside a Cyan source file.

Index

A.1 Separate Compilation

A Cyan package can be packed into a jar file except its generic prototypes, which should continue to be in the package directory. To force the compilation of package `cyan.lang`, attach the following annotation to `program` in the project file:

```
@feature("compilePackageCyanLang", true)
program
    ...
```

Unless this annotation is in the project file, the compiler will look for a `cyan.lang.jar` file in directory given by the environment varible `CYAN_HOME`.

To compile a package named `aaa.bbb` that is not `cyan.lang` for separate compilation, create a project with the following contents:

```
program
    @annot("compilePackage")
    package aaa.bbb
    // possibly other packages
```

Now call the Java compiler to compile **all** of the Java files for the package. Saci will call the Java compiler but only the files reachable from the main class will be compiled. Therefore it is necessary to call the Java compiler again. Finally, call the `jar` program to create a jar file. More instructions are given in the Cyan site.

The jar file should be in the *base directory* of the package. That is, for package `aaa.bbb`, the directory structure should be as shown below.

```
aaa.bbb.jar
aaa\
    bbb\
        // package prototypes
```

Whenever a package is referenced, the Cyan compiler will look for a jar file in the base directory of the package. That is, if the the project file is

```
program
    package ccc.ddd at "C:\Dropbox\tests\00\ccc\ddd"
    // possibly other packages
```

then the compiler will look for

`C:\Dropbox\tests\00\ccc.ddd.jar"`

If there is no such file, Saci will compile the files of the package directory. Note that even if there is a jar file, generic prototypes of package `ccc.ddd` should be kept in the package directory. They will be searched for in that directory.

A.2 Known Compiler Errors

This is a list of known compiler errors.

1. a Java class that is not imported can be used if it is in file “rt.jar”;
2. The code

```
var Int n;
repeat
    if true {
        break
    }
    else {
        n = 0
    }
    n println
until true;
```

should not cause the error "local variable n may not have been initialized" in line

```
n println
```

However, it does.

3. literal hash tables (maps, dictionaries) with union types cause a Java compilation error.

```
let Int|String is95 = 0;
let myMap = [ is95 -> is95, "zero" -> 0, 0 -> "zero" ];
```

4. the line number of the error below is incorrect because of annotation `insertCode`

```
package main

object Program

    @insertCode{*
        for num in [ 2, 3, 4, 5 ] {
            insert: " func multBy$num: Int n -> Int = n*$num;"
        }
    *}

    func test {
        zeroTen = 11; // no variable zeroTen
    }
end
```