The Cianeto Language

José de Oliveira Guimarães
Departamento de Computação
UFSCar - Sorocaba, SP
Brasil
e-mail: josedeoliveiraguimaraes@gmail.com

November 19, 2018

Cianeto is a statically-typed object-oriented language based on language Cyan [4]. Unlike Cyan, Cianeto is class-based and offers the basic features of object-oriented programming. A program is composed by a single source file. Every class can only refer to the classes that appear textually before it in this file.

An example of a program in Cianeto is shown below

```
class Store
    var Int n
    func get -> Int {
        return self.n;
    }
    func set: Int n {
        self.n = n;
    }
end

class Program {
    func run {
        var Store s;
        var Int a;
        s = Store.new;
        a = In.readInt;
        s.set: a;
        Out.println: s.get;
    }
end
```

Every program must have a class named `Program` with a parameterless method called `run`. To start the execution of a program, the runtime system creates an object of class `Program` and sends to it a message `run`.

In the following sections we define the main elements of the Cianeto language.

1 Classes

A class has the format

```
class Name
  public and private members
end
```

in which `Name` is an identificar, the class name, that should be different from all previously declared classes. A member of a class may be a private field (variable) or a public or private method. The next example shows the declaration of a class with private fields `x` and `ok` and public methods `sum` and `setOk`. Note methods are public by default, so the word "public" is not necessary before their declarations. A private field or method is invisible outside the class. So a subclass can define a field or a public or private method with the same name as a private field or private method of the superclass. And a subclass can define a private field or private method with the same name as a public or private method of the superclass. If there is a private method in a subclass with the same name as a public method of the superclass, the first is not overridden the second. Unary methods and fields share the same identifier space. So a field and an unary method cannot have the same name. A class may have zero members. The `run` method of class `Program` must be public and it should not take any parameters.

```
class A
    var Int x
    func sum: Int y -> Int {
        return self.x + y;
    }
    var Boolean ok;
    func setOk: Boolean ok {
        self.ok = ok;
    }
    ...
end
```

The syntax for method declaration follows. Any symbols between [and] are optional.

```
    [ qualifier ] func methodName: parameter-list [ -> ReturnType ] {
       Statement-List
    }
```

or

```
    [ qualifier ] func methodName [ -> ReturnType ] {
       Statement-List
    }
```

If the method takes parameters, its name is an identifier with an attached "`:`". There should not be a space between the two.

The return type may be a class or a basic type (`Int`, `Boolean`, or `String`). The syntax for local variable declaration is equal to Cyan and described in the next section. The name of a method or class field (instance variable) should be different from the name of the previous members of the class and different from the class name. A member of a class is an field or method. Statements include declaration of variables. So,

```
     var Int x, y;
```

counts as a statement.

Fields, local variables, and parameters whoses types are classes are in fact pointers, as in Cyan.

```
    ...

    var Store  s, t;
    s = Store.new;
    t = s;
```

Therefore, in the code above, the declarations of `s` and `t` do not create objects of class `Store`. An object of this class should be dynamically created with `new`:

```
      s = Store.new;
```
Memory deallocation is made by the garbage collector. The basic classes `Boolean`, `Int`, and `String` do not have a `new` method.

There is a global value `nil` that can be assigned to any variable whose type is a class. `nil` represents a value of a class that does not have methods.

1.1 Message Sends

The statement
```
      s.msg: b0, b1, ... bn
```
is the sending of the message "`msg: b0, b1, ... bn`" to the object pointed to by `s`. This message send is valid if:

(a) the declared class of `s` or one of its superclasses has a public method with name "`msg:`".

(b) method `msg:` takes `n` parameters and the type of each argument `bi` is convertible (see section 2.6) to the type of the `i`th formal parameter of `msg:`.

At runtime, the runtime system (RTS) makes a search for a method `msg:` in the class of the object referred by `s`. If it is not found there, the search continues in the superclass of the class of this object, the superclass of the superclass, and so on. When a method is found, it is called. The RTS will always found a adequate method except when the variable points to `nil`.

A method that does not return a value can be used as a statement:
```
      stack.print;
```
A method that returns a value must be called only within an expression as in
```
      if stack.getSize > 0 { insert: 0; }
```

1.2 Inheritance

Inheritance of a classe `A` by a class `B` is made with keyword "`extends`". In the following example, `Student` inherits `Person`.

```
open
class Person
    var String name
    func setName: String name { self.name = name; }
    func getName -> String { return self.name; }
    func asString -> String {
        return "Person(" + self.name + ")";
    }
end

class Student extends Person
    var Int number
    func setNumber: Int number { self.number = number; }
    func getNumber -> Int { return self.number; }
    override
    func asString -> String {
        return super asString + " Student(" + self.number + ")";
    }

end
```

Class `B` inherits all fields and methods of `A`. However, a class has to be declared with the word "`open`", before "`class`", in order to be inherited. A public method of `A` may be redefined in `B` if its signature (name, return value type, and parameter types) is not changed. The redefined method should also be public and it should be preceded by keyword `override`. If `public` is used, `override` should precede `public`.

It is allowed to redefine in `B` a private method of `A`. It is as if `B` defined a new method since no one knows the private methods of `A`. Class `B` has no access to private fields and methods of `A`.

The statement

```
    super.m:  p1, p2, ...  pn
```

inside a `B` method orders the compiler to look for a `m:` method starting at the superclass of `B`, `A`. If it is not found in `A`, the search continues in the superclass of `A` and so on. The parameters p_1, p_2, ... p_n must be convertible (section 2.6) to the formal parameters of the `m:` method found in the search.

Message sends to `super` are linked at compile time to a specific method of a superclass. No search for a method is made at runtime.

A final class is declared without using "`open`" before "`class`" as in

```
class Earth
    ..
end
```

A final class cannot be inherited. All message sends to its methods can be static. That is, there is no need of a search for a method at runtime.

A non-final class, also called an "*open class*", may have final methods, declared as

```
    final public func get -> Int {  ... }
```

A final method cannot be redefined in a subclass. Only public methods can be final and a final class cannot declare a "`final`" method (since the methods of a final class are already final). No other restrictions apply. A class `C` may extends a class `B` that extends a class `A` that defines a method `m` redefined in `B` and `C`. The `m` method of `C` may be declared final. The order of the qualifiers of a method is

```
    final override public
```

1.3 `self`

The keyword `self` represents a variable whose type is the class in which it is being used. `self` points to the object that received the message that caused the execution of the method. The same as in Cyan. Using class `Store` (first example), the code

```
    s = Store.new;
    s.set: 12;
```

causes the execution of method `set:` of `Store`. Inside this method, `self` points to the same object as `s`. This keyword is used to handle the fields and to call private and public methods of the class — observe class `Store`. The value of `self` cannot be modified by an assignment.

In a message send `self.m` or `self.m:  args` inside a class `A`, the compiler searches for a private method `m` or `m:`. If none is found, the search continues in the public methods of `A`. If none is found, the search continues in the public methods of the superclass of `A`, the superclass of the superclass, and so on.

2 Basic Cianeto Elements

2.1 Assert Statement

statement `assert` takes an expression and a string. At runtime, if the expression is false the string is printed.

```
class Test
    func run {
        var Int n;
        n = 0;
        n = n + 1;
        assert n > 0, "Wrong code generation for '>' or '+' with type 'Int'"
    }
end
```

2.2 Comments

Comments in the language are put between "/*" and "*/" and between // and the end of the line. Nested comments are not allowed, as in

```
/* comment /* another comment  */  end of the first comment */
```

The comment ends in the first `*/` found. Note that "/*" and "*/" may appear inside a comment started with // — they will not mean a comment. The converse is also true.

2.3 Basic Literals and Types

There are only three basic types in Cianeto: `Int`, `Boolean`, and `String`. Literals of type `Int` must be between `0` and `2147483647`. Any number of zeros before the number is allowed. Therefore the numbers

```
00000000000001
00000000000000
```

are legal. Type `Boolean` has only two values: `false` and `true`.

`String` literals must appear between quotes: `"Do you always come here?"`

The backslash \ can be used to remove the meaning of " and the backslash itself. In fact, the string `"\c"` has the same meaning as in Cyan (same as Java) regardless of the character `c`, which can be anything.

The comparison operators `<`, `<=`, `>`, `>=` can only be applied to `Int` values. The comparison operators `==` and `!=` can be applied to all basic types. If `LeftType` is `String`, `RightType` must be `String` or `nil`. If `RightType` is `String`, `LeftType` must be `String` or `nil`. `nil` cannot be compared with `nil`.

The operators `==` and `!=` can also be used to compare expressions whose types are classes. The result is a `Boolean` value, `true` if both sides refer to the same object, `false` otherwise. In a comparison `left == right` or `left != right`, if `LeftType` is the declared type of `left` and `RightType` is the declared type of `right`, one of three things must occur:

(a) `LeftType` is `RightType`;

(b) `LeftType` is convertible to `RightType`. See definition of *convertible* in section 2.6;

(c) `RightType` is convertible to `LeftType`.

That is, classes `LeftType` and `RightType` must be related by inheritance. All other possibilities are illegal, for we know the expression `left == right` will evaluate to `false` anyway.

Operators `+`, `*`, `-`, and `/` apply to `Int` values resulting in `Int` values. The semantics of these operators is the same as in Cyan (same as Java).

Operator `++` converts both the receiver and the parameter to strings and concatenates them. The receiver and the parameters should have type `Int` or `String`.

```
assert "a" ++ "b" ++ 0 == "ab0";
```

The binary operators `&&` and `||` and the unary `!` accept `boolean` operands and have the usual meaning. The evaluation of the expression

```
left && right
```

start in `left`. If `left` is `false`, all the expression is considered `false`, even without the evaluation of `right` (which can be an expression). If `left` is `true`, `right` will be evaluated.

The expression

```
left || right
```

is `true` if `left` is `true`. If `left` is `false`, the result has the value of `right`.

The type of a variable, parameter, or return value of a method must be a basic type (`Int`, `Boolean`, or `String`) or a previously declared class, which may be the current class.

2.4 Identifiers

Identifiers are composed by letters, digits, and underscore (`_`), starting with a letter. Here are some examples of valid identifiers:

```
getNum   x0   y1   get_Num
```

The identifiers

```
_main    3ab    get$Num     write
```

are ilegal. "`class`" is illegal because it is a keyword. The list of keywords of Cianeto is given below. Note that word "open" is not a keyword.

```
assert
Boolean
break
class
else
extends
false
final
func
if
Int
nil
override
private
public
repeat
return
self
String
super
true
until
var
while
```

There is no limit in the number of characters of an identifier. Upper and lower case letters are considered different. All identifiers must be declared before used. None can be declared twice in the same scope (see below).

2.5 Scope

The scope of a class name is from the place it appears to the end of the file. The scope of a field or method is from the place it is declared to the end of the class. However, fields and methods of the object that received the message must be handled using the keyword "`self`":

```
self.n = 0;
x = self.m: 5;
self.p.set: 0;
```

Therefore a code like

```
set: 0;
```

is always illegal because there is no receiver to the message "`set: 0`".

The scope of local variables and parameters of a method is from the place of declaration to the end of the method body. A local variable cannot have the same name as a parameter of the same method. Local variables and fields are always distinguished because fields are accessed using `self`. Local variables have precedence over class names, which are global.

2.6 Assignment

The assignment of an expression `rightExpr` to a variable `left` or to a field `f` is made as

```
left = rightExpr;
self.f = rightExpr;
```

If `LeftType` is the declared type of `left` (or `self.f`) and `RightType` the type of `rightExpr`, this statement is valid if:

(a) `LeftType` is a basic type (`Int`, `Boolean`, or `String`) and `LeftType` is `RightType`;

(b) `RightType` is a class which is subclass of `LeftType`. We consider that a class is subclass of itself;

(c) `LeftType` is a class or `String` and `rightExpr` is value `nil`.

If any of the items is valid, we say that `RightType` is convertible to `LeftType`.

The same rules apply to parameter passing to methods and return value by means of the command `return`. That is, statements like

```
bb.m: pr;
return r;
```

are valid if assignments

```
pf = pr;
x = r;
```

are valid, in which `pf` has the same type as the formal parameter of method `m` and `x` is a variable with the same type as the return value type of the method in which this statement is.

A single local variable may be declared and receive a value using an assignment:

```
var Int sum = 0;
```

However, only a single variable should follow the type.

```
    // compilation error
var Int count, sum = 0;
```

2.7 Decision Statement

The `if` statement has the form

```
if expr {
    Statement
}
[
else {
    Statement
}
]
```

The `else` part is optional. `expr` must be a `Boolean` expression.

2.8 Repetition Statements

The statement

```
while expr {
    Statement
}
```

means that `Statement` is to be executed while the `Boolean` expression `expr` evaluates to `true`. This loop may be ended by executing statement `break`.

There is also a `repeat-until` statement with the syntax

```
repeat
    statementList
until expr
```

If `expr` evaluates to true after `statementList` is executed, the loop ends. `break` can be used inside a `repeat-until` statement to ends the statement. Of course, it is illegal to use `break` outside a `while` or `repeat-until` statement.

2.9 Method Return Value

A method returns a value with the command

```
 return expr;
```

If the type of the return value `expr` is `U` and the declared return type of the method is `T`, then `U` must be convertible to `T` (see section 2.6). That is, an assignment

```
 t = u;
```

should be legal, in which the types of `t` and `u` are `T` and `U`, respectively. A method without a return value type cannot have a `return` command. A method with return value should have at least one `return` command.

2.10 Input and Output

Input is made by methods `readInt` and `readString` of class `In`.[1]

```
var Int age;
age = In.readInt;
var String name;
name = In.readString;
```

Each variable is read in a separate line in the standard input.

Printing to the standard output device is made with class `Out` and methods `print:` and `println:`. Each takes one parameter that may be either an `Int` or `String`. Of course, `println:` prints the parameters and a character `\n`.

```
Out.print: "name = " ++ person.name;
Out.print: " age = ";
Out.println: person.age;
```

[1]They would be like `static` methods of Java.

3 Metaobjects

This section need not to be implemented in Lab. de Compiladores/2018.

Cianeto supports a very limited version of metaobjects. A metaobject is an object that controls other objects. In this case, a metaobject is an object that controls the compilation process. We have no space here to describe them completly so we will just explain how the available metaobjects works.

Metaobject `ce` is a metaobject that communicates to the compiler that there is a compiler error in the source code. It is used through an *annotation* in the source code as `@cep` in the first line of the following example.

```
class Stuff
   // nothing
end
@cep(4, "O nome da classe está ausente",
       "Missing identifier",
       "comp.Compiler.classDec()")

class
    func run { }
end
```

The annotation takes from two to four parameters (the last two ones are optional). The first one is the offset, in relation to the current line, of the line that has the error. The second one is a string with a description of the error. The third one is the error that the compiler should sign. This is a suggestion — the compiler implementer is free to use other error messages. The fourth parameter is the method of the Cianeto compiler that should be modified in order to make the compiler sign this error.

In this example, the annotation `@cep(...)` informs the compiler that there is an error in line 4 + current line number. The error description is

`"O nome da classe está ausente"`

The compiler should issue an error message that is something like

`"Missing identifier"`

To make the Cianeto compiler to sign this error, you should change method

`comp.Compiler.classDec()`

After compiling the source code of this example, the Cianeto compiler will issue a warning if it did not sign an error message. It should. If it signed, it will show the actual error message and the error message `"Missing identifier"`. Then the user can compare if the compiler really signed the error with the correct error message. The Cianeto compiler will also check if the line number of the error signed by itself is 4.

There is just one more metaobject: `nce`. It does not take parameters and informs the compiler that there should be no compilation errors in the source code. If there is any, the Cianeto compiler will point out that it has a flaw.

```
@nce

class Program
    func run { }
end
```

4 The Cianeto Grammar

This section defines the language grammar. The reserved words and language symbols are shown between " and ". A sequence of symbols between { and } can be repeated zero or more times and a sequence of symbols between [and] is optional. Parentheses group symbols. For example,

D ::= (A|B) { C }

means A or B followed by any number of C´s. Id is an identifier and IdColon is an identifier with a suffix “`:`”.

The non-terminal `StringValue` represents a string of characters between quotes as in Cyan: `"Hi !!"`. `IntValue` is an integer number. The rules for both `StringValue` and `IntValue` are not in the grammar.

The initial grammar non-terminal is Program.

Annot	::= “@” Id [“(” { AnnotParam } “)”]
AnnotParam	::= IntValue | StringValue | Id
AssertStat	::= “assert” Expression “,” StringValue
AssignExpr	::= Expression [“=” Expression]
BasicType	::= “Int” | “Boolean” | “String”
BasicValue	::= IntValue | BooleanValue | StringValue
BooleanValue	::= “true” | “false”
ClassDec	::= [“open”] “class” Id [“extends” Id] MemberList “end”
CompStatement	::= “{” { Statement } “}”
Digit	::= “0” | ... | “9”
Expression	::= SimpleExpression [Relation SimpleExpression]
ExpressionList	::= Expression { “,” Expression }
Factor	::= BasicValue |
	“(” Expression “)” |
	“!” Factor |
	“nil” |
	ObjectCreation |
	PrimaryExpr
FieldDec	::= “var” Type IdList [“;”]
FormalParamDec	::= ParamDec { “,” ParamDec }
HighOperator	::= “*” | “/” | “&&”
IdList	::= Id { “,” Id }
IfStat	::= “if” Expression “{” Statement “}”
	[“else” “{” Statement “}”]
IntValue	::= Digit { Digit }
LocalDec	::= “var” Type IdList [“=” Expression]
LowOperator	::= “+” | “−” | “||”
MemberList	::= { [Qualifier] Member }
Member	::= FieldDec | MethodDec
MethodDec	::= “func” IdColon FormalParamDec [“->” Type]
	“{” StatementList “}” |
	“func” Id [“->” Type] “{” StatementList “}”
ObjectCreation	::= Id “.” “new”
ParamDec	::= Type Id
Program	::= { Annot } ClassDec { { Annot } ClassDec }
Qualifier	::= “private”
	“public”
	“override”
	“override” “public”
	“final”
	“final” “public”
	“final” “override”
	“final” “override” “public”

ReadExpr	::= “In” “.” [“readInt” | “readString”]
RepeatStat	::= “repeat” StatementList “until” Expression
PrimaryExpr	::= “super” “.” IdColon ExpressionList |
	“super” “.” Id |
	Id |
	Id “.” Id |
	Id “.” IdColon ExpressionList |
	“self” |
	“self” “.” Id |
	“self” ”.” IdColon ExpressionList |
	“self” ”.” Id “.” IdColon ExpressionList |
	“self” ”.” Id “.” Id |
	ReadExpr
Relation	::= “==” | “<” | “>” | “<=” | “>=” | “! =”
ReturnStat	::= “return” Expression
Signal	::= “+” | “−”
SignalFactor	::= [Signal] Factor
SimpleExpression	::= SumSubExpression { “++” SumSubExpression }
SumSubExpression	::= Term { LowOperator Term }
Statement	::= AssignExpr “;” | IfStat | WhileStat | ReturnStat “;” |
	WriteStat “;” | “break” “;” | “;” |
	RepeatStat “;” | LocalDec “;” |
	AssertStat “;”
StatementList	::= { Statement }
Term	::= SignalFactor { HighOperator SignalFactor }
Type	::= BasicType | Id
WriteStat	::= “Out” “.” [“print:” | “println:”] Expression
WhileStat	::= “while” Expression “{” StatementList “}”

References

[1] Stroustrup, Bjarne. *The C++ Programming Language*. Second Edition, Addison-Wesley, 1991.

[2] Lippman, Stanley B. *C++ Primer*. Addison-Wesley, 1991.

[3] Guimarães, José de Oliveira. *The Green Language*. http://http://www.cyan-lang.org/jose/green/green.htm.

[4] Guimarães, José de Oliveira. *The Cyan Language*. http://www.cyan-lang.org/