# Adaptive Programming in Cyan

J. O. Guimarães and P. R. M. Cereda

***Abstract*—Adaptive devices make it possible to express algorithms in a concise and readable way. However, programming languages usually do not offer explicit support for this technology. The programmer has to employ obscure language constructs and tricks to make the runtime changes demanded by adaptivity. This article shows how the Cyan object-oriented language already supports the adaptive programming style in a limited way. Since the adaptive code is made with known language constructs (rutime reflection), it is relatively easy to learn, use and understand.**

***Keywords*—Adaptive code, Cyan, Object-oriented language**

## I. Introduction

Adaptive technology has emerged as an alternative and viable solution for tackling complex problems [1]. There is a quite vast record in the literature that describes methods and techniques on problem solving via adaptive devices such as robotics [2], data mining [3], natural language processing [4], compiler construction [5] [6], privacy and personalization [7], [8], access control [9], and much more. An adaptive device may spontaneously change its own behaviour over time due to some external stimuli.

This paper contributes to the adaptive technology area by showing how adaptive features are supported by the object-oriented language Cyan. The subject is not new: there are programming languages constructs such as that proposed by Freitas e Neto [10] to support adaptive programming. The features presented in this article do not allow illimited code modifications but they are implemented by the regular Cyan constructs.

This paper is organized as follows: the following section describes a subset of the Cyan language used in this paper. Section III shows two mechanisms by which Cyan supports adaptive programming: method insertion and code blocks. The last section concludes the article.

## II. The Cyan language

Cyan is a new prototype-based statically-typed object-oriented language that supports single inheritance, Java-like interfaces, statically-typed closures (similar to Smalltalk blocks), an object-oriented exception system, optional dynamic typing, runtime reflection, and many other features. The language has many innovations which are described in its manual [11]. In this paper, we will present only the features necessary to understand how Cyan supports adaptive programming.

Before we proceed, let's set up an example to be used as a proof of concept for presenting the adaptive features of Cyan. The adaptive automaton $M$, presented in Figure 1, recognizes strings in the form $a^n b^n$, with $n \in \mathbb{N}$, $n > 0$. That is, strings with an arbitrary number of $a$'s followed by the very same number of $b$'s. This language is of course context-free and could be recognized by a simple pushdown automata. The adaptive automaton is overpowered to this task since it has the same computational power of a Turing machine [12]. But it suits our needs to demonstrate how Cyan supports adaptive programming.

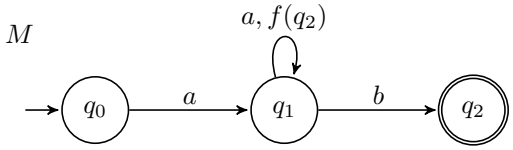


Fig. 1. An adaptive automaton that recognizes strings in the form $a^n b^n$, with $n \in \mathbb{N}$, $n > 0$.

The execution of $M$ is very straightforward: the automaton is initially prepared to recognize the "base case"; that is, $ab$. If it gets a second $a$, an adaptive function $f$ is triggered in order to change the topology, adding a new transition consuming $b$ to a newly created state and updating the original transition consuming the first $b$ to go from that state to the acceptance one, so the automaton is now prepared to recognize $aabb$. In other words, for every extra $a$ in the input string, $M$ will add a correspoding $b$ to the "$b$ recognition path". If we want our automaton to also recognize an empty string, we simply need to make $q_0$ an acceptance state as well.

Since now we have our example properly presented, let's go back to the language. Cyan is a prototype-based language, which means the concept of class is replaced by that of "prototype". A prototype is declared almost exactly like a class except that it is preceded by keyword "`object`". Objects are created from prototypes using methods called `new` or `new:`, which is similar to some class-based languages. Prototypes are grouped in packages which can be imported by other packages. Methods are declared with keyword "`fun`" which should be followed by the method selectors (explained later), return value type, and the method body (between `[` and `]`). An instance variable is declared by putting "`:`", the variable name and its type. An example of Cyan code is given below.

```
package main
   // this is a comment
   /* this is also a comment */

object AF_AnB
       // methods are public by default
       // the method name is "check:"
  fun check: (:in Array<Char>) -> Boolean
     self.in = in
     if ( recognize ) [
          // if the input has ended
          // after recognizition, accepts
```

jose@ufscar.br e paulo.cereda@usp.br.

```
        return index == in size;
    ]
    else [
        return false;
    ];
  ]

  private fun recognize -> Boolean [
    if (in size < 1) [ return false; ]
    else [
        // declare local variable i
        // of type Int with value 0
      :i Int = 0;
      while ( in[i] == 'a' ) [
        ++i;
      ];
        // sends message "size" to "in"
      if ( i < in size ) [
        if ( in[i] == 'b' ) [
           return true;
        ];
      ];
      return false;
    ];
  ]

    /* instance variables are
       private by default */
  :in Array<Char>
end
```

Surprisingly, the previous code implements our adaptive automaton $M$ from Figure 1 with built-in Cyan constructs. Prototype `AF_AnB` has public method `check:` (an `Array<Char>` as parameter, `Boolean` as the return value type) and private method `recognize` (no parameters, returns a `Boolean`). There is one instance variable, `in`, which is an array of `Char`. Inside another prototype of package `main`, one can send a message directly to `AF_AnB`, which is an object that exists at runtime:

```
package main
object Test
  fun test [
     // {# starts a literal array
    AF_AnB check: {# 'a', 'b' #};
  ]
  fun fat: (:n Int) -> Int [
    if ( n <= 0 ) [ return 1; ]
    else [
        return n*(fat: n - 1);
    ];
  ]
end
```

Statement

```
AF_AnB check: {# 'a', 'b' #}
```

is the sending of message `check:` with parameter `{# 'a', 'b' #}` to object `AF_AnB`. At runtime, there is a search in object `AF_AnB` for a method named `check:` that takes a `Array<Char>` as parameter. Since Cyan is statically-typed, the compiler checks wheather prototype `AF_AnB` has a method `check:` with an `Array<Char>` parameter. We use words "object" and "prototype" to refer to "prototypes". When an object is created at runtime, as in:

```
:myTest Test = Test new;
```

the word "object" is used to refer to it. Here message "`new`" is sent to object `Test` to create a new object (it would be "`Test new()`" in Java). `Test` is the type of local variable `myTest`.

Prototype `Test` declares a method `fat:` that takes one `Int` parameter and returns an `Int`. It can be called as "`Test fat: 5`". Inside this method there is a recursive call in "`fat: n - 1`". Since there is no receiver for this message, it is a message send to "`self`", which is the receiver of the original `fat:` message. `self` is similar to `this` of Java/C++/C#.

Following Smalltalk [13], "`fat:`" is called a *selector*. There may be more than one selector for a method each one taking zero or more parameters:

```
object Hash
    fun key: String value: Int [ ... ]
    ...
end
```

This method can be called as

```
Hash key: "one" value: 1
```

We can also declare a selector without parameters:

```
fun add: key: String value: Int
```

Closures, which are unnamed literal functions, are supported by Cyan. They are called "blocks" as in Smalltalk and are declared and used as:

```
:isZero Block<Int><Boolean>;
isZero = [ |:n Int -> Boolean|
    ^ n == 0;
];
if ( isZero eval: 0 ) [
    Out println: "0 is zero";
];
```

The type of variable `isZero` is `Block<Int><Boolean>`, which means a closure that takes an `Int` parameter and returns a `Boolean`. To `isZero` is assigned a closure, which starts with `[` and ends with `]`. Between the '`|`' symbols should appear the parameter and return value (which can be omitted because it can be deduced). This closure takes a parameter called `n`. After the last '`|`' there may appear a list of statements. The return value of the closure is given after symbol '`^`'. Since closures are objects in Cyan, they have methods. In particular, statements that appear inside the closure are put in a method called `eval` or `eval:` that take the same parameters and have the same return value as the closure itself. Then "`isZero eval: 0`" calls the statement inside the closure, which is "`^ n == 0`". Since `n` receives `0`, the parameter to `eval:`, this call returns `true`.

In a regular message send the Cyan compiler makes a search for a method that matches the message send, starting in the prototype that is the static type of the receiver. If a method is not found there, the search continues in the super-prototype[1], super-super-prototype, and so on. Then in a statement

```
person name: "Anna" age: 2
```

the compiler makes a search for a method

```
name: String age: Int
```

in the static type of `person`, which is a prototype. If `person` was declared as "`:person Person`", the search starts at prototype `Person`.

A dynamic message send is a message send in which the compiler does not check wheather the static type of the receiver has a method that matches the signature of the message. It is like a regular message send in which the message selectors are preceded by "?" as in

```
anObj ?name: "Anna" ?age: 2
```

A new method may be dynamically added to a prototype using a method `addMethod: ...` defined in prototype `Any`, the super-prototype of every one. We used "`...`" to elide the many method selectors. The message send that follows add a method called `comb:` to prototype `Test`, the receiver of the message.[2]

```
Test addMethod:
     selector: "comb:"
     param: Int, Int
     returnType: Int
     body: (:self Test) [
         |:n, :k Int -> Int|
         ^(fat: n)/( (fat: k)*(fat: n-k));
     ];
```

Selector `addMethod:` does not take parameters — it is only used to make the code clearer. Selector `selector:` takes one parameter which is the selector name. Selector `param:` is followed by the types of the parameters of `"comb:"` (two `Int`'s). The parameter names should not be cited. The return type is also of type `Int`. The parameter to "`body:`" is a *context block*, a special kind of closure that represents a method that may be added to a prototype at runtime. A context block

```
(:self T)[ ... ]
```

may be added to prototype `T` or its sub-prototypes using method `addMethod:` (as in the example above). Inside this context block, the compiler checks wheather the message sends to `self` match the methods of `T`. For example, `fat: n` is a message send to `self` (since the receiver is implicit). The Cyan compiler searches for a method `fat: Int` in prototype `Test` and its super-prototypes (since the declared type of `self` in this context block is `Test`).

A context block is like a method that is not attached to any prototype but can be added to several of them dynamically. By assigning the context block to a variable, we can pass this variable as parameter to selector `body:` in several calls to `addMethod: ...` thus adding the same context block to several different prototypes.

`addMethod: ...` can add a new method to a prototype, like in the previous example, or it may replace an existing method. In the first case, we cannot call the method as in

```
numComb = Test comb: 5, 2;
```

because the compiler would not find a method `comb: Int, Int` in prototype `Test`. This method should be called using a dynamic message send:

```
numComb = Test ?comb: 5, 2;
```

The context block

```
(:self Test) [ |:n, :k Int -> Int|
    ^ (fat: n)/( (fat: k)*(fat: n - k) );
]
```

takes two `Int` parameters and returns an `Int`. After the message send `addMethod: ...` of the example is sent, method `comb:` is added to `Test`. It is as if we had copied the text of the context block to a newly created method

```
comb: Int, Int -> Int
```

of `Test`. The above context block has three message sends with selector `fat:`. These are message sends to the object to which the context block is attached. In statement "`Test ?comb: 5, 2`", message `fat: n` is sent to `self`, which is `Test`. Method `fat: Int -> Int` of `Test` is called.

There is another way of doing dynamic calls in Cyan. It is using the ‘ operator, which is better explained using an example.

```
:k Int;
:s String = "fat";
   // same as "Test fat: 5"
k = Test ‘s: 5;
s = "test";
    // same as "Test test"
Test ‘s;
```

If `str` is of type `String` and holds string `strValue`,

```
anObj ‘str
```

is a dynamic message send equivalent to

```
anObj strValue
```

The compiler considers that the return value type is `Any`, the super-prototype of every one (the equivalent of `Object` of Smalltalk and Java).

## III. Development

As presented in the previous section, the reflective features of Cyan can be used for implementing adaptive code in an unlimited way. However, by lack of space in this article we will only show two ways of implementing adaptive features in the language: by *method insertion* and by simulating *code blocks*.

[1] Analogous to superclass.

[2] The method is always added to the receiver.

### A. Adaptive features by method insertion

The prototype AF_AnBn, which maps our adaptive automaton $M$ and it's given below, recognizes the language $L = \{a^n b^n \mid n \in \mathbb{N}\}$. Method check: calls method a to recognize $a^n$ and b to recognize $b^n$. It only returns true if both methods return true and the input was fully consumed.

Initially method b returns true regardless of the input. This should be changed according to the number of 'a' symbols in the beginning of the input in. When method a is called, it scans all the 'a' symbols it finds and then adds to prototype AF_AnBn a new method b that will scans a number of 'b' symbols that is equal to the number of 'a's already found. The addition of a new method b is made with addMethod: ... with a context object. Note that, when the context block is created, the value of local variable n is copied to an instance variable of the context block. That means any changes to variable n inside the context block does not propagate to the local variable n declared in method a.

```
package main

object AF_AnBn
  fun check: (:in Array<Char>) -> Boolean
     self.in = in
     index = 0;
     if ( a && b ) [
          // if the input has ended,
          // accepts
         return index == in size;
     ]
     else [
         return false;
     ];
  ]

  private fun a -> Boolean [
     :n = 0;
       // recognizes n symbols 'a'
     while ( index < in size &&
             in[index] == 'a' ) [
         ++index;
         ++n;
     ];
        // replaces method 'b -> Boolean'
     AF_AnBn addMethod:
        selector: #b
        returnType: Boolean
        body: (:self AF_AnBn) [
            | -> Boolean |
            :newN = 0;
            while ( index < in size &&
                    in[index] == 'b' ) [
                ++index;
                ++newN;
            ];
            return newN == n;
        ];
     return true;
  ]

  private fun b -> Boolean [
     return true;
  ]

  public :in Array<Char>
  public :index Int

end
```

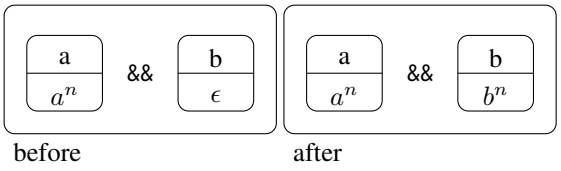


Fig. 2. Languages recognized by methods a and b.

Method check tests wheather a && b is true. Then method a and method b are called, in this order (there is no short-circuit evaluation). If method a finds n 'a' symbols in the beginning of the input, it adds to prototype Test a method b that checks if the next n symbols of the input are 'b' symbols. If this is false, the method returns false.

Figure 2 shows grafically what happens at runtime. The rectangle labelled "before" shows methods a and b and the strings they recognize in the first call to method check of AF_AnBn. Initially method b recognizes the empty string. Rectangle "after" shows these methods after method a is called. This call changes method b in such a way that it now recognizes language $b^n$ for $n \in \mathbb{N}$. The value of n is known after all 'a' symbols are found. This value is used to create an appropriated method b, which is a context object.

Method a replaces method b with a context object that can be attached to prototype AF_AnBn and its sub-prototypes (because it is declared as (:self AF_AnBn)[ ... ]). Then inside the context object one can send messages that match the methods of this prototype. Apparently there is no message send to self inside this context object. But in fact there are several of them because the public instance variables in and index of Test are transformed into public get and set methods that are called in statements like ++index and expressions like index < in size.

### B. Code blocks in Cyan

Methods can be dynamically added to a prototype and called using a string holding the method name (as in "Test 's"). This scheme can be used to simulate code blocks in Cyan.

A code block is simply an object with a run method (or a piece of code that is put in a list. Then at runtime the run methods of the objects are called by a special method that we call a "combinator". The *combinator* may use many algorithms to define the order in which the run methods are called. It may call one by one in the list order, it may call only the method of the first list object, it may use the integer return value of one run method to define the next method to call,

and so on. There may be more than one combinator for a list, each one implementing different algorithms. For example, one combinator can recognize language $L = \{a^n b \mid n \in \mathbb{N}\}$ and other may recognize $L = \{ab \mid n \in \mathbb{N}\}$, both starting with the same object list.

The run methods may as parameter a hashtable used as shared memory. The pairs in the table would be composed by a variable name and an object. This shared table can be used for communication among the list objects.

The list can be changed dynamically to adapt the code: methods can be removed, added, and replaced. Then it makes sense to have a combinator that calls always method run of the first list object. Thi is because the first object may change from call to call.

Code blocks are implemented in Cyan using operator ‘ that calls the method whose name is in a string. The example that follows shows a prototype whose method check: returns true if its parameter belongs to the language $\{a^n b^n : n \in \mathbb{N}\}$. Method check: initializes an array funArray of methods to be called by the compinator. Instead of objects with a run method, here we put the name of the methods, as strings, in the list (which is an array). The combinator gets each of the strings of funArray and calls them in the order they appear. Initially there is only one string, "a", in funArray[0]. Method combinator puts this string in variable methodName (see first statement after while in this method). Method a is called in the message send

```
self ‘methodName
```

Method a inserts in prototype CodeBlocks a method b and in funArray a string "b". Method b is inserted using addMethod: ... as before. Statement

```
funArray add: "b"
```

inserts "b" at the end of array funArray.

```
object CodeBlocks
  fun check: (:in Array<Char>) -> Boolean
    self.in = in
    index = 0;
    funArray = {# "a" #};
    return combinator;
  ]

  fun combinator -> Boolean [
    :i = 0;
    while ( i < funArray size ) [
        :methodName String = funArray[i];
        if ( self ‘methodName == false ) [
            return false;
        ];
        ++i;
    ]
    return true;
  ]

  fun a -> Boolean [
    :n = 0;
      // recognizes n symbols 'a'
    while ( index < in size &&
            in[index] == 'a' ) [
      ++index;
      ++n;
    ];
    if ( n != 0 ) [

      // replaces method 'b -> Boolean'
      CodeBlock addMethod:
         selector: #b
         returnType: Boolean
         body: (:self CodeBlocks) [
           | -> Boolean |
           :newN = 0;
           while ( index < in size
             && in[index] == 'b' ) [
             ++index;
             ++newN;
           ];
           return newN == n;
         ];
      funArray add: "b";
    ];
    return true;
  ]

  :funArray Array<String>
  public :in Array<Char>
  public :index Int
end
```

As seen previously, code blocks implemented as a list of objects with a run method need a hashtable that is used as a shared memory. That is not necessary with the implementation of code blocks in Cyan because the instance variables of prototype CodeBlocks are used as shared memory. Let us explain that.

Methods like b are added to prototype CodeBlocks using addMethod: ... with a context object targeted to prototype CodeBlocks (this is the type of self in the context object). Therefore all added methods can call methods of CodeBlocks in message sends to self. Then all added methods share the CodeBlocks instance variables, which work like a shared memory for them.

## IV. Final remarks

The Cyan constructs used in this paper are limited in their power — they are regular language constructs. However, their limitations makes them more foreseeable since many code transformations, like inserting a statement in the middle of the code, are not possible. By using regular language features we assure that a regular programmer can understand them rather easily.

By using adaptive programming, we aim at reducing the burden of writing complex code; once the transformation rules are defined, the code evolves through a well-established space

of configurations without external interference. Problems that require intricate interations may benefit from the techniques presented in this paper since they are made without adding any new constructs to the Cyan language. Therefore, they are relatively easy to understand and use.

## REFERENCES


[1] J. José Neto, "A small survey of the evolution of adaptivity and adaptive technology," *Latin America Transactions, IEEE (Revista IEEE America Latina)*, vol. 5, no. 7, pp. 496–505, 2007.

[2] A. R. Hirakawa, A. M. Saraiva, and C. E. Cugnasca, "Adaptive automata applied on automation and robotics (a4r)," *Latin America Transactions, IEEE (Revista IEEE America Latina)*, vol. 5, no. 7, pp. 539–543, 2007.

[3] P. R. M. Cereda and J. José Neto, "Mineração adaptativa de dados: Aplicação à identificação de indivíduos," in *Memórias do Sexto Workshop de Tecnologia Adaptativa*, São Paulo, 2012.

[4] R. L. A. Rocha, "Adaptive technology applied to natural language processing," *Latin America Transactions, IEEE (Revista IEEE America Latina)*, vol. 5, no. 7, pp. 544–551, 2007.

[5] J. José Neto, "Contribuições à metodologia de construção de compiladores," Tese de livre-docência, Escola Politécnica da Universidade de São Paulo, São Paulo, 1993.

[6] J. C. Luz and J. José Neto, "Tecnologa adaptativa aplicada à otimização de código em compiladores," in *IX Congreso Argentino de Ciencias de la Computación*, La Plata, Argentina, 2003.

[7] P. R. M. Cereda and S. D. Zorzo, "Formalismo com autômato adaptativo em mecanismo de privacidade e personalização," in *Latin American Computing Conference*, San José, Costa Rica, 2007.

[8] P. R. M. Cereda, R. A. Gotardo, and S. D. Zorzo, "Resource recommendation using adaptive automaton," in *16th International Conference on Systems, Signals and Image Processing*, 2009.

[9] P. R. M. Cereda and S. D. Zorzo, "Access control model formalism using adaptive automaton," *Latin America Transactions, IEEE (Revista IEEE America Latina)*, vol. 6, no. 5, pp. 443–452, 2009.

[10] A. V. Freitas and J. J. Neto, "Adaptivity in programming languages," *WSEAS Transactions on Information Science & Applications*, vol. 4, no. 4, pp. 779–786, Apr. 2007.

[11] J. de Oliveira Guimarães, "The Cyan language," 2012. [Online]. Available: http://www.cyan-lang.org

[12] R. L. A. Rocha and J. José Neto, "Autômato adaptativo, limites e complexidade em comparação com máquina de Turing," in *Proceedings of the second Congress of Logic Applied to Technology*, 2001.

[13] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983, ISBN: 0-201-11371-6.


**José de Oliveira Guimarães** José de Oliveira Guimarães is a UFSCar Professor since 1992. He has an undergraduate degree by USP/São Carlos, a master degree by Unicamp, and a Doctorate from USP/São Carlos. He researches in Programming Languages and Software Engineering (toos, user interface in IDE's, and design patterns). His main research area is object-oriented programming. José has designed the object-oriented languages Green and Cyan. Other interest areas are Computability and Axiomatic Set Theory.

**Paulo Roberto Massa Cereda** is a PhD candidate in Computer Engineering working on adaptive technology at the Languages and Adaptive Techniques Laboratory (LTA), at Escola Politécnica, University of São Paulo under the supervision of prof. João José Neto.