Portable Simula Revisited

Mapping Simula to Java

Runtime Design

Table of Contents

Background.

The 50th anniversary of Simula, the first object-oriented programming language, Was held at the Department of Informatics, University of Oslo on 27th September, 2017. James Gosling, the inventor of Java, gave overwhelming credit to the legacy of Simula in his lecture. He mentioned that a number of programming languages Nowadays compile to JVM. During his lecture, I began to wonder - why has no team made a simula implementation in java, written in java that compiles to java virtual machine(JVM).

Inspired by the lectures on the Simula Jubilee, I began to ask myself if I should try to make a Simula implementation using today's tools. It should certainly be a Recursive Descent parser which I believe is an excellent technique.

In early November I started work on such a compiler. I was surprised that it took So short time to complete a rough version that compiled to Java code. This, first Version, was finished just before Christmas. It was not in any way complete, but It generated some kind of Java code that could be compiled and executed. However, the original semantics of the Simula program were not preserved.

I presented these results in a meeting at Blindern, Ole-Johan Dahls Hus, just before Christmas in 2017. I was then encouraged by Stein Krogdahl and Dag Langmyhr to create a full-fledged Simula Implementation strictly after 'Simula Standard'.

From then on, I worked on a complete redesign of the Simula runtime system. The design is based on previous Simula implementations I have worked on, but this time some major changes have been made. Eg. a separate Garbage Collector is not necessary because the underlying Java system fixes that.

This document describes the runtime design of the new Portable Simula System. It is intended as a reference document that should be studied in parallel with the source code of the compiler and its associated runtime system.

Oslo, august 2019

Øystein Myhre Andersen

1. Block Instances at runtime.

Simula is a block oriented language. This means that the source text is organized as a set of nested blocks. They can be classes, procedures and sub-blocks. All of which are represented as subclasses of the Java class RTObject$ during execution.

The Java class RTObject$ has the following main attributes:

`OperationalState STATE$`; One of { ***detached***,***resumed***,***attached***,***terminated*** }

boolean isQPSystemBlock()
This is a static property generated by the compiler.
It will return true for Block or Prefixed Block with local classes.

boolean `isDetachable()`
This is a static property generated by the compiler.
It will return true for Classes which can be Detached.

`RTObject$ SL$` This is a pointer to the object of the nearest textually enclosing block instance, also called 'static link'.

`RTObject$ DL$` If this block instance is attached this is a pointer to the object of the block Instance to which the instance is attached (also called dynamic link), i.e. it points to the block instance which called this one.

`Coroutine CORUT$` This is a pointer to the Coroutine in which this block instance is running. If this block instance is detached it is used to save the complete reactivation point (call stack and the continuation point).

int `JTX$` `Jump Table Index used to implement goto`

`In addition a global variable` ***`CUR$`*** `will always` point to the 'Current Block'

The figure on next page shows the main structure of nested block instances at runtime.
Ole Johan Dahl had a similar figure in a compendium in 1980. The main differences are:

- There is no Prototype Pointer (we use Java's Object.getClass method instead)
- There is no Context Vector (we follows Static Links SL$)
- There is no Garbage Collector (we use Java's instead)

Variables in outer blocks are accessed through the SL$-chain. For example:

Variable c in current block	==>	c or ((Block1)CUR$).c
Variable b in P2	==>	((P)(CUR$.SL$)).b
Variable a in outermost block	==>	((Block2)(CUR$.SL$.SL$)).a

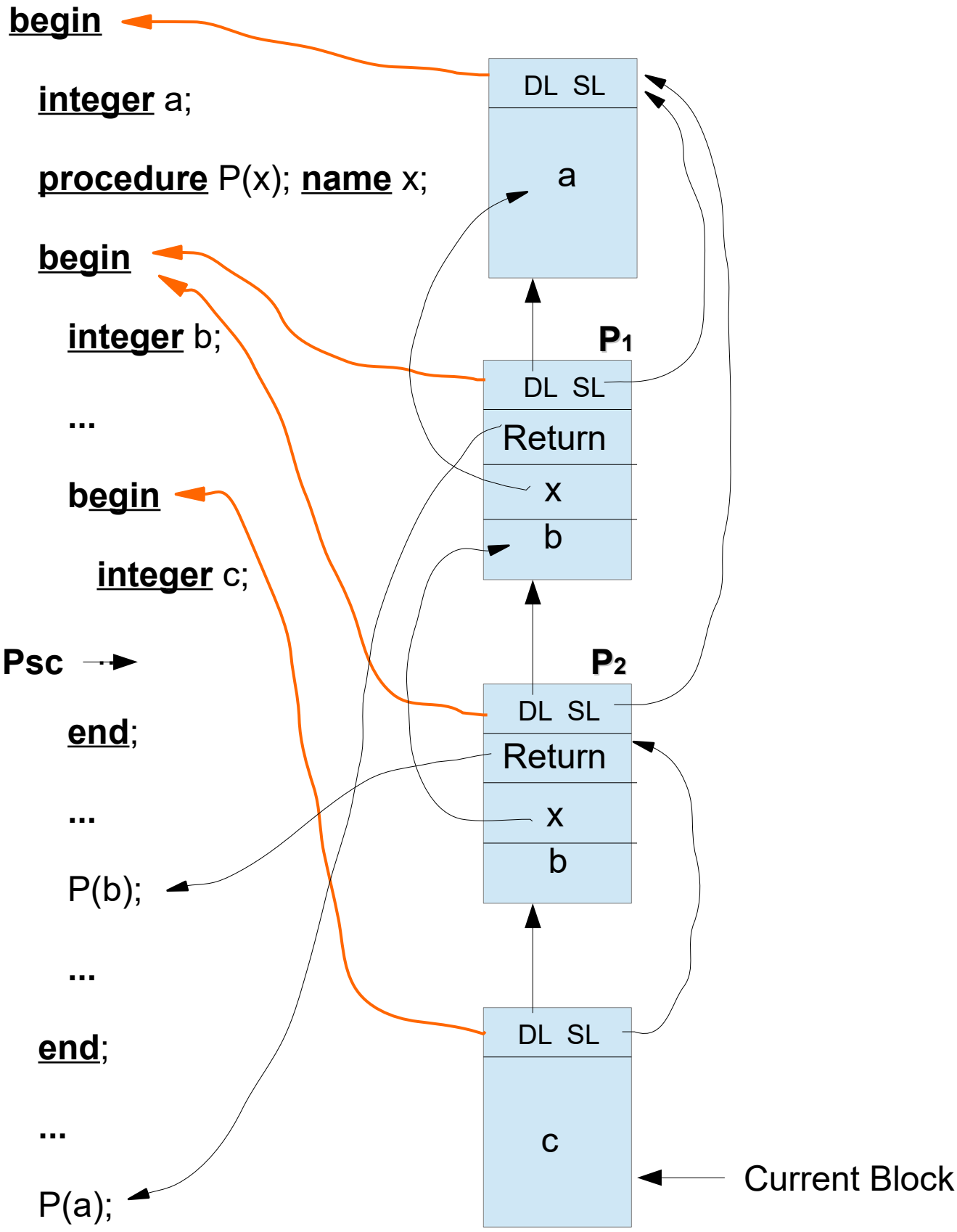

Fig. 1 Sample Block Structure

(due to Ole-Johan Dahl)

2. Mapping Simula Blocks to Java classes.

Each and every Simula Class, Prefixed Block, Procedure or Sub-Block is compiled into a Java Class.

2.1 Sub- Block.

A sub-block is the simplest of all Simula Blocks. For example, consider the following Simula code:

```
begin integer lim=40;
      text array TA(1:lim);
      integer i;
      i:=6;
end;
```

It is compiled to a Java class like:

```
public final class SubBlock extends BASICIO$ {
    // Declare locals as attributes
    final int lim=40;
    public ARRAY$<TXT$[]>TA;
    int i;
    // Normal Constructor
    public SubBlock(RTObject$ staticLink) {
        super(staticLink);
        BBLK();
        // Declaration Code
        int[] TA$LB=new int[1]; int[] TA$UB=new int[1];
        TA$LB[0]=1; TA$UB[0]=lim;
        BOUND_CHECK$(TA$LB[0],TA$UB[0]);
        TA=new ARRAY$<TXT$[]>(new TXT$[TA$UB[0]-TA$LB[0]+1],TA$LB,TA$UB);
     }
    // SubBlock Statements
    public RTObject$ STM$() {
        i=6;
        EBLK();
        return(this);
     }
}
```

The Java super class BASICIO$ is defined in the runtime system as a convenient way to enable all environment methods to this block. It is itself a Java sub-class of RTObject$.

The constructor set up the static environment (SL$ and CORUT$) of this RTObject$.

The BBLK method is used to initiate the block instance. It will attach the block to its dynamic environment (DL$ and CORUT$) and update the current block pointer(CUR$). The OperationalState become ***attached***;

The EBLK method is used to terminate the block instance when control passes through the final end (of the outermost prefix for class and prefixed block). This is a complex method dependent on the different kind and states of block instances.
The OperationalState become ***terminated***;

2.2 Simula Classes

Consider the following Simula Class:

```
Class A(p1,p2); integer p1; real p2;
begin integer lim=40;
      text array TA(1:lim);
      integer i; real r;
      i:=p1;
      inner;
      r:=p2;
 end;
```

It will be compiled to a Java Class like:

```
public class A extends CLASS$ {
  // Declare parameters as attributes
  public int p$p1;
  public float p$p2;
  // Declare locals as attributes
  public final int lim=40;
  public ARRAY$<TXT$[]>TA=null;
  public int i;
  public float r;
  // Normal Constructor
  public A(RTObject$ staticLink,int sp$p1,float sp$p2) {
      super(staticLink);
      // Parameter assignment to locals
      this.p$p1 = sp$p1;
      this.p$p2 = sp$p2;
      BBLK(); // Iff no prefix
      // Declaration Code
      int[] TA$LB=new int[1]; int[] TA$UB=new int[1];
      TA$LB[0]=1; TA$UB[0]=lim;
      BOUND_CHECK$(TA$LB[0],TA$UB[0]);
      TA=new ARRAY$<TXT$[]>(new TXT$[TA$UB[0]-TA$LB[0]+1],TA$LB,TA$UB);
  }
  // Class Statements
  public A STM$() {
      // Class A: Code before inner
      i=p$p1;
      // Class A: Code after inner
      r=p$p2;
      EBLK();
      return(this);
  }
}
```

Here we see that both parameters and local variables are declared as local members. The constructor consists of parameter transmission and Array declaration code.

The member method STM$ contains the concatenated statement code belonging to this Simula Class.

CLASS$ is a subclass of RTObject$ which, among other things, define the Detach method.

2.3 Sub-classes

Now let's look at a sub-class of Simula Class A:

```
A Class B(p3); long real p3;
begin
      array RA(1:lim);
      RA(14):=r;
      inner;
      r:=RA(14);
end;
```

Which will be compiled to a Java Class like:

```
public class B extends A {
    // Declare parameters as attributes
    public double p1$p3;
    // Declare locals as attributes
    public ARRAY$<float[]>RA=null;
    // Normal Constructor
    public B(RTObject$ staticLink,int sp$p1,float sp$p2,double sp1$p3) {
        super(staticLink,sp$p1,sp$p2);
        // Parameter assignment to locals
        this.p1$p3 = sp1$p3;
        // Declaration Code
        int[] RA$LB=new int[1]; int[] RA$UB=new int[1];
        RA$LB[0]=1; RA$UB[0]=lim;
        BOUND_CHECK$(RA$LB[0],RA$UB[0]);
        RA=new ARRAY$<float[]>(new float[RA$UB[0]-RA$LB[0]+1],RA$LB,RA$UB);
    }
    // Class Statements
    public B STM$() {
        // Class A: Code before inner
        i=p$p1;
        // Class B: Code before inner
        RA.Elt[14-RA.LB[0]]=r;
        // Class B: Code after inner
        r=RA.Elt[14-RA.LB[0]];
        // Class A: Code after inner
        r=p$p2;
        EBLK();
        return(this);
    }
}
```

Here we see that:

- Parameters are accumulated via the Constructors.

- Declaration code concatenation is done via the constructors.

- Concatenation of Statement code is done explicitly by copying the Java code.

Object generation, i.e. new B(1,2,3) is compiled to:

```
x=new B(CUR$,1,2f,3d).STM$();
```

Where *CUR$* is a global variable pointing at the 'Current Block'

2.4 Prefixed Block.

A prefixed block is very much like a sub-class:

```
B(1,2,3) begin
begin
      Text array TA(1:lim);
      Integer i;
      i:=6;
end;
```

Will be compiled to:

```
new PBLK(CUR$,1,2f,3d).STM$();
```

Where PBLK is a Java Class like:

```
public final class PBLK extends B {
    // Declare locals as attributes
    public ARRAY$<TXT$[]>TA=null;
    int i$2=0;
    // Normal Constructor
    public PBLK(RTObject$ staticLink,int sp$p1,float sp$p2,double sp1$p3) {
        super(staticLink,sp$p1,sp$p2,sp1$p3);
        // Declaration Code
        int[] TA$LB=new int[1]; int[] TA$UB=new int[1];
        TA$LB[0]=1; TA$UB[0]=40;
        BOUND_CHECK$(TA$LB[0],TA$UB[0]);
        TA=new ARRAY$<TXT$[]>(new TXT$[TA$UB[0]-TA$LB[0]+1],TA$LB,TA$UB);
    }
    // Class Statements
    public PBLK STM$() {
        // Class A: Code before inner
        i=p$p1;
        // Class B: Code before inner
        RA.Elt[14-RA.LB[0]]=r;
        // Class PBLK: Code
        i$2=6;
        // Class B: Code after inner
        r=RA.Elt[14-RA.LB[0]];
        // Class A: Code after inner
        r=p$p2;
        EBLK();
        return(this);
    }
}
```

Again we see that:

- Parameters are accumulated via the Constructors.

- Declaration code concatenation is done via the constructors.

- Concatenation of Statement code is done explicitly by copying the Java code.

2.5 Procedure Declaration

Procedures however; follow a different scheme.

```
integer Procedure P(p1,p2); integer p1; real p2;
begin text Array TA(1:40); integer i;
      P:=p1;
      i:=6;
end;
```

No concatenation in this case but the procedure must be prepared to be called as a formal procedure without parameter specifications.

```
public final class P extends PROC$ {
 // Declare return value as attribute
 public int RESULT$;
 public Object RESULT$() { return(RESULT$); }
 // Declare parameters as attributes
 public int p$p1;
 public float p$p2;
 // Declare locals as attributes
 public ARRAY$<TXT$[]>TA=null;
 int i=0;

 // Parameter Transmission in case of Formal/Virtual Procedure Call
 public P setPar(Object param) {
     try { switch($nParLeft--) {
             case 2: p$p1=intValue(param); break;
             case 1: p$p2=floatValue(param); break;
             default: throw new RuntimeException("Too many parameters");
     }   }
     catch(ClassCastException e) {
          throw new RuntimeException("Wrong type of parameter: "+param,e); }
     return(this);
 }

 // Constructor in case of Formal/Virtual Procedure Call
 public P(RTObject$ SL$) {
     super(SL$,2); // Expecting 2 parameters
 }

 // Normal Constructor
 public P(RTObject$ SL$,int sp$p1,float sp$p2) {
     super(SL$);
     // Parameter assignment to locals
     this.p$p1 = sp$p1;
     this.p$p2 = sp$p2;
     BBLK();
     // Declaration Code
     int[] TA$LB=new int[1]; int[] TA$UB=new int[1];
     TA$LB[0]=1; TA$UB[0]=40;
     BOUND_CHECK$(TA$LB[0],TA$UB[0]);
     TA=new ARRAY$<TXT$[]>(new TXT$[TA$UB[0]-TA$LB[0]+1],TA$LB,TA$UB);
     STM$();
 }

 // Procedure Statements
 public P STM$() {
     RESULT$=p$p1;
     i=6;
     EBLK();
     return(this);
 }
}
```

2.5.1 Procedure Calls

Ordinary procedure-call are similar to class object generation:

```
P(4,3.14);          ==>   new P(CUR$,4,3.14f);
x.P(4,3.14);        ==>   new P(x,4,3.14f);
Inspect x do P(4,3.14);       ==>   if(x!=null) new P(x,4,3.14f);
```

However, we need to pick up the result of <type>procedures.

```
i:=P(4,3.14);     ==>   i=new P(CUR$,4,3.14f).RESULT$;
i:=x.P(4,3.14);   ==>   i=new P(x,4,3.14f).RESULT$;
Inspect x do i:=P(4,3.14);    ==>   if(x!=null) i=new P(x,4,3.14f).RESULT$;
```

Formal Procedures

A call on a formal Procedure; however; will look quite different. Suppose that F is a procedure quantity representing a formal parameter procedure.
Then a call like `i:=F(7,9);` will generate code like this:

```
i = intValue(F.CPF()
       .setPar(new NAME$<Integer>() { public Integer get() { return (7); }})
       .setPar(new NAME$<Integer>() { public Integer get() { return (9); }})
       .ENT$().RESULT$());
```

The parameters are transmitted by repeated calls on the compiler-generated method 'setpar'. Since we neither know the transfer mode nor the type, the parameter is transferred 'by name' and post-processed by 'setpar'.

Virtual Procedures

A Simula virtual procedure is represented by a Java virtual member method in the Java class. The method returns a Procedure Quantity (PRCQNT$) pointing to the matching Simula Procedure Declaration. We use Java's virtual concept to implement Simula's virtual procedures. When a procedure redefinition occur in a subclass, an additional member method is generated by the compiler.

If there is no matching procedure a default method is generated by the compiler:

```
public PRCQNT$ V(){ throw new RuntimeException("No Virtual Match: P"); }
```

If a matching procedure is declared the following method is generated:

```
public PRCQNT$ V() { return(new PRCQNT$(this,V.class)); }
```

In this way, we ensure that the correct virtual match is called.

A call on a virtual Procedure uses the V() method to select the virtual match. Apart from that a call on a virtual procedure will follow the same pattern as formal procedures. However, if the virtual procedure is specified, a call like `i:=x.V(4,3.14);`
will generate this simplified code:

```
i = intValue(V().CPF().setPar(4).setPar(3.14f).ENT$().RESULT$());
```

In this case transmission 'by name' is not necessary.

2.6 Switch Declaration

Switch declarations are treated as LABQNT$ Procedures without any optimization whatsoever.

Consider the following example:

```
switch S2:=L1,L2,S(3),S( expr );
```

This code will be generated:

```
public final class S2 extends PROC$ {
    // Declare return value as attribute
    public LABQNT$ RESULT$;
    public Object RESULT$() { return(RESULT$); }
    // Declare parameters as attributes
    public int p$$SW;

    // Parameter Transmission in case of Formal/Virtual Procedure Call
    public S2 setPar(Object param) {
        try {
            switch($nParLeft--) {
                case 1: p$$SW=intValue(param); break;
                default: throw new RuntimeException("Too many parameters");
            }
        }
        catch(ClassCastException e) {
             throw new RuntimeException("Wrong type of parameter: "+param,e); }
        return(this);
    }

    // Constructor in case of Formal/Virtual Procedure Call
    public S2(RTObject$ SL$) {
        super(SL$,1); // Expecting 1 parameters
    }

    // Normal Constructor
    public S2(RTObject$ SL$,int sp$$SW) {
        super(SL$);
        // Parameter assignment to locals
        this.p$$SW = sp$$SW;
        BBLK();
        // Declaration Code
        STM$();
    }

    // Switch Body
    public S2 STM$() {
        switch(p$$SW-1) {
            case 0: RESULT$= enc.L1; break;
            case 1: RESULT$= enc.L2; break;
            case 2: RESULT$=new S(CUR$.SL$,3).RESULT$; break;
            case 3: RESULT$=new S(CUR$.SL$,expr).RESULT$; break;
            default: throw new RuntimeException("Illegal switch index: "+p$$SW);
        }
        EBLK();
        return(this);
    }
}
```

2.6.1 Switch Designators

Ordinary switch designators are similar to call on <type>procedures.

```
Goto P(4); ==>   GOTO$(new S(CUR$,4).RESULT$);
```

*Switches are implicit specified **protected**, thus goto a remote or connected switch is illegal.*

Formal Switches

Formal switches follows the pattern of procedures.

Suppose that F is a procedure quantity representing a formal switch.
Then the statement: goto *F(4);* will generate code like this:

```
GOTO$((LABQNT$)(F.CPF()
           .setPar(new NAME$<Integer>(){ public Integer get() {return(4);}})
           .ENT$().RESULT$()));
```

Virtual Switches

A virtual switches follow the pattern of virtual procedures. It is represented by a Java virtual member method in the Java class. The method returns a Procedure Quantity (PRCQNT$) pointing to the matching Simula Switch Declaration.

3. Representation of Quantities.

Within an object, each quantity local to that object is represented. Local quantities are e.g. Declared variables in a block (sub-block, procedure body or prefixed block), or the attribute of a class. Quantities may be of value types, object reference type, text descriptors, array quantities, procedure quantities, label quantities or switch quantities..

3.1 Value type variables.

A variable of value type is represented in Java as follows:

Boolean	boolean
Character	char
Short integer	int *)
Integer	int
Real	float
Long real	double

*) Simula Standard 2.1.1 states; " An implementation may choose to implement short integer exactly as integer, i.e. ignoring the keyword short "

3. 2 Object Reference Types.

Object reference variables are represented in Java as such.

ref(ClassIdentifier)	ClassIdentifier
none	**null**

3. 3 Text Reference Types.

Text reference variables is a composite structure represented by a TXTREF Java Object.

Text	TXT$

where TXT$ has the following definition:

```
public class TXT$ extends RTObject {
   TEXTOBJ OBJ; // Reference to the text object.
   int START;   // Start index of OBJ.MAIN[], counting from zero.
                // Note this differ from Simula Definition.
   int LENGTH;
   int POS;     // Current index of OBJ.MAIN[], counting from zero.
                // Note this differ from Simula Definition.
   ...
}
```

The special text constant **notext** is represented by Java **null**.

The object TEXTOBJ is defined as:

```
public class TEXTOBJ extends RTObject {
   int SIZE;       // Number of characters in the text object.
   boolean CONST;  // True: Indicates a text constant
   char[] MAIN;    // The characters
   ...
}
```

3.4 Array Quantities.

A local array is represented by a reference to an array object which is a subclass of _ARRAY:

```
public abstract class _ARRAY {
	public  final _BOUNDS[] BOUNDS;
	public  final int SIZE;
	private final int BASE;
	final private int[] DOPE;

	public _ARRAY(final _BOUNDS... BOUNDS) {
		this.BOUNDS = BOUNDS;
		DOPE=new int[BOUNDS.length];
		DOPE[0]=1;
		int SIZE=BOUNDS[0].SIZE;
		int BASE=BOUNDS[0].LB * DOPE[0];
		for(int i=1;i<BOUNDS.length;i++) {
			DOPE[i] = BOUNDS[i-1].SIZE * DOPE[i-1];
			BASE=BASE + BOUNDS[i].LB * DOPE[i];
			SIZE=SIZE * BOUNDS[i].SIZE; }
		this.SIZE=SIZE;
		this.BASE=BASE;
	}

	public int nDim() { return (BOUNDS.length); }
	public int size() { return (SIZE); }
	public int lowerBound(int i) { return(BOUNDS[i].LB); }
	public int upperBound(int i) { return(BOUNDS[i].LB + BOUNDS[i].SIZE - 1); }

	public abstract _ARRAY COPY();

	public int index(int... x) {
		int idx=0;
		for(int k=0;k<x.length;k++) {
			if(x[k] < lowerBound(k) || x[k] > upperBound(k))
				throw new _SimulaRuntimeError("Array index("+(k+1)+") = "+x[k]
					+" is outside bounds "+BOUNDS[k]);
			idx=idx+(x[k] * DOPE[k]);
		}
		return(idx - BASE);
	}
}
```

The implementation technique used is called 'dope vector indexing'. The dope vector is calculated in the constructor based on the array bound pairs. A fictitious base is also Calculated. This base is the relative address of the array element (0,0, ... 0).

Indexing is done by using this data in the 'index' method.

A parameter Array is represented by the _ARRAY quantity.

Arrays of different types are declared <type> array, for example:

```
public final class _INT_ARRAY extends _ARRAY {
      final private int[] ELTS;
      public _INT_ARRAY(final _BOUNDS... BOUNDS) {
            super(BOUNDS); ELTS=new int[SIZE]; }
      public int putELEMENT(int ix,int val) {
            ELTS[ix]=val; return(val); }
      public int getELEMENT(int... x) {
            return(ELTS[index(x)]); }
      public _INT_ARRAY COPY() {
            _INT_ARRAY copy = new _INT_ARRAY(BOUNDS);
            System.arraycopy(ELTS, 0, copy.ELTS, 0, SIZE);
            return(copy); }
}
```

Example: A one-dimensional *Integer Array Table(6:56)* is declared by:

```
table=new _INT_ARRAY(new _BOUNDS(6,56));
```

And we access elements like this:

```
i=table.getELEMENT(x);
table.putELEMENT(table.index(x),i);
```

And, a muliti-dimesional t*ext Array TA(6:56,13:34,-74:-30, ...)* is coded like:

```
ta=new _TEXT_ARRAY(new _BOUNDS(6,56),new _BOUNDS(13,34)
                  ,new _BOUNDS(-74,-30), ...);
```

And we access elements like this:

```
t=ta.getELEMENT(45,21,-60, ...);
ta.putELEMENT(ta.index(45,21,-60, ...),t);
```

3.5 Procedure Quantities.

A procedure is represented as a composite structure as shown below. Procedure quantities are used to represent formal procedure parameters. Local procedures, however, are not represented as quantities within an object.

```
public class $PRCQNT
{ RTObject$ staticLink;
  Class<?> procedure;

  // Constructor
  public $PRCQNT(RTObject$ staticLink,Class<?> procedure)
  { this.staticLink=staticLink; this.procedure=procedure; }

  public PROC$ CPF()
  { try
    { Constructor<?> constr = procedure.getConstructor(new Class[]{RTObject$.class});
      Object obj=constr.newInstance(staticLink);
      return((PROC$)obj);
    }
    catch(Throwable e) { throw new RuntimeException("Internal Error",e);     }
  }

}
```

The attribute 'staticLink' is a pointer to the block-instance in which the procedure is declared.

In traditional Simula implementations, a prototype is generated to hold basic informations of block-instances, including procedures. In this implementation no Prototype is generated. Instead, we use Java's reflection mechanisms to create procedure objects.

The attribute 'procedure' is a pointer to Java's 'prototype' i.e. a pointer to a Java Object containing all information of the class representing the Simula procedure.

The CPF method is used to create procedure object when generating call to formal or virtual procedures. Further details in a later section.

Suppose we have a Simula procedure 'P'. To create a procedure quantity at runtime we use a constructs like this:

```
new PRCQNT$(CUR$,P.class);     // P is local
new PRCQNT$(CUR$.SL$,P.class); // P is declared at an outer block
```

3.6 Label Quantities.

Label quantities are explained in chp. 6.1.1.

3.7 Switch Quantities.

This implementation treats a Simula Switch as a LABQNT$ Procedure.

4. Parameter Transmission.

4.1 Parameter Transmission 'by Name'.

The basic principle is to establish an object within the calling scope. This object will have two attributt methods; 'get' og 'put' to read the value of the actual parameter, or, if legal, to write into it. The following Java-class is used to perform such parameter transmissions:

```
abstract class $NAME<T> {
      abstract T get();
      void put(T x) { error("Illegal ..."); }
}
```

Note that we both use abstract Java classes and 'generics' i.e. the actual type is a parameter. Also note that the 'put' method has a default definition producing an error. This enables redefinition of the 'put' method to be dropped for expression as actual parameters.

Suppose the Simula Procedure:

***procedure** P(k); **name** k; **integer** k; k:=k+1;*

It will be translated to something like this Java method:

```
void P($NAME<Integer> k) {
      k.put(k.get() + 1); // E.g: k=k+1
}
```

In the calling place, in practice in the actual parameter list, we create an object of a specific subclass of *$NAME<T>* by specifying the Integer type and defining the get and put methods. Eg. If the current parameter is a variable 'q', then the actual parameter will be coded as follows:

```
 new $NAME<Integer>() {
      Integer get() { return (q); }
      void put(Integer x) { q = (int) x;  }
}
```

However, if the actual parameter is an expression like (j + m * n) then it will be coded as follows:

```
new $NAME<Integer>() {
      Integer get() { return (j + m * n);       }
}
```

Here we see that the 'put' method is not redefined so that any attempt to assign a new value to this name parameter will result in an error message.

A classic example of name parameters is taken from Wikipedia's article about *Jensen's Device*. We change it slightly because Simula does not allow the control variable in a for-statement to be a formal parameter transferred by name.

```
long real procedure Sum(k, lower, upper, ak);
    value lower, upper; name k, ak;
integer k, lower, upper; long real ak;
        begin long real s;
                s := 0.0;
                k := lower;
                while k <= upper do
                begin
                        s := s + ak;
                        k := k + 1;
                end while;
                Sum := s
        end Sum;
```

Translated to Java, this will be:

```
public double Sum($NAME<Integer> k,
            int lower, int upper,
            $NAME<Double> ak) {
      public double s;
      s = ((double) (0.0));
      k.put(lower);
      while (k.get() <= upper) {
            s = s + ak.get();
            k.put(k.get() + 1);
      }
      return (s);
}
```

At procedure call like this:

```
integer i;
long real array A[0:99];
long real result;

resultat=Sum(i,10,60,A[i]);
```

Is translated to:

```
public int control;
public double[] A=new double[100];
public double result;


result = Sum(
  new $NAME<Integer>() {
      Integer get() {   return(i); }
      void put(Integer x) { i = (int)x; }
  },
  10,
  60,
  new $NAME<Double>() {
      Double get() { return (A[i]); }
      void put(Double x) { A[i] = (double) x; }
  });
```

4.2 Array as actual parameter

A procedure call with array parameter *i:=P(A);* where the parameter A is specified 'value' will generate code like:

```
i=new P(CUR$,A.COPY()).RESULT$;
```

Where the method 'COPY' is defined in the runtime system.
See: https://coderanch.com/t/378421/java/Multidimensional-array-copy

When the parameter is specified 'name' and the actual parameter expression is:
i:=P(if cond then A else B); the following code is generated:

```
i=new P(CUR$,new NAME$<ARRAY$<?>>() {
   public ARRAY$<?> get() { return(cond?A:B); } }).RESULT$;
```

And, finally, in default transmission mode the code will be:

```
i=new P(CUR$,A).RESULT$;
```

In all cases it is assumed that the procedure P is declared in the same block as the procedure call. If not, substitute '*CUR$*' with '*CUR$.SL$...*' to refer the static environment of the procedure.

4.3 Procedure identifier as actual parameter to formal procedure

In this case, we don't know if it is the name or value P to be transferred. In all cases the parameter is represented by a NAME$<PRCQNT$> which can be used to obtain a reference to the actual parameter procedure. Which in turn can be used to evaluate the value of P.

ActualParameter is visible procedure

A procedure call with procedure parameter *i:=Q(P);* where the parameter P is specified 'by name' and the actual parameter is an ordinary visible procedure will generate code like:

```
i = new Q(CUR$, new NAME$<PRCQNT$>() {
      public PRCQNT$ get() { return (new PRCQNT$(CUR$, P.class)); }
}).RESULT$;
```

ActualParameter is remote procedure

A procedure call with procedure parameter *i:=Q(x.P);* where the parameter is specified 'by name' and the actual parameter is an remote procedure will generate code like:

```
i = new Q(CUR$, new NAME$<PRCQNT$>() {
      public PRCQNT$ get() { return (new PRCQNT$(x, P.class)); }
}).RESULT$;
```

ActualParameter is connected procedure

A procedure call with procedure parameter *inspect x do r := Q(P);* where the parameter is specified 'by name' and the actual parameter is an connected procedure will generate code like:

```
if (x != null) // INSPECT x
       r = new Q(CUR$, new NAME$<PRCQNT$>() {
              public PRCQNT$ get() { return (new PRCQNT$(x, P.class)); }
       }).RESULT$;
```

ActualParameter is virtual procedure

A procedure call with procedure parameter *i:=Q(V);* where the parameter V is specified 'by name' and the actual parameter is an virtual procedure will generate code like:

```
s = new Q(CUR$.SL$..., new NAME$<PRCQNT$>() {
       public PRCQNT$ get() { return V(); }
}).RESULT$;
```

Where V() is the PRCQNT$ method used to evaluate the virtual match.
See also 5.1.5 Virtual Procedures.

ActualParameter is remote virtual procedure

A procedure call with procedure parameter *i:=Q(x.V);* where the parameter V is specified 'by name' and the actual parameter is a remote virtual procedure will Generate code like:

```
s = new Q(CUR$.SL$..., new NAME$<PRCQNT$>() {
       public PRCQNT$ get() { return x.V(); }
}).RESULT$;
```

Where V() is the PRCQNT$ method used to evaluate the virtual match.

ActualParameter is connected virtual procedure

A procedure call with procedure parameter *inspect x do r := Q(V);* where the parameter V is specified 'by name' and the actual parameter is a connected virtual procedure will generate code like:

```
if (x != null) // INSPECT x
       s = new Q(CUR$.SL$..., new NAME$<PRCQNT$>() {
              public PRCQNT$ get() { return x.V(); }
       }).RESULT$;
```

Where V() is the PRCQNT$ method used to evaluate the virtual match.

5. Some Expressions

5.1 Object Relation IS:

b := ***x is C***;

will be translated to:

```
b = IS$(x,B.class);
```

where the method *IS$* is defined by:

```
public boolean IS$(final Object obj,final Class<?> cls) {
	return((obj == null)?false:(obj.getClass() == cls));
}
```

5.2 Object Relation IN:

b := ***x in C***;

will be translated to:

```
b = x instanceof C;
```

5.3 QualifiedObject QUA:

x qua B.p3 := 3.14;

will be translated to:

```
((B)x).p3=3.14;
```

6. Some Statements

6.1 Goto Statement

Java does not support labels like Simula. The Java Virtual Machine (JVM), however, has labels. A JVM-label is simply a relative byte-address within the byte-code of a method. We will use Java's exception handling together with *byte code engineering* to re-introduce goto in the Java Language. This is done by generating Java-code which is prepared for Byte Code Engineering.

Suppose a Simula program containing labels and goto like this:

```
Begin
  L: ...
     goto L;
     ...
End;
```

This will be coded as:

```
1. public final class adHoc00 extends BASICIO$ {
2.     public int prefixLevel() { return(0); }
3.     final LABQNT$ L=new LABQNT$(this,1); // Local Label #1=L
4.     public adHoc00(RTObject$ staticLink) {
5.         super(staticLink);
6.         BBLK();
7.         BPRG("adHoc00");
8.     } // End of Constructor
9.
10.    // SimulaProgram Statements
11.    public RTObject$ STM() {
12.        adHoc00 THIS$=(adHoc00)CUR$;
13.        LOOP$:while(JTX$>=0) {
14.            try {
15.                JUMPTABLE$(JTX$); // For ByteCode Engineering
16.                // Statements ....
17.                LABEL$(1); // L
18.                // Statements ....
19.                throw(L); // GOTO EVALUATED LABEL
20.                // Statements ....
21.                break LOOP$;
22.            }
23.            catch(LABQNT$ q) {
24.                CUR$=THIS$;
25.                if(q.SL$!=CUR$) {
26.                    CUR$.STATE$=OperationalState.terminated;
27.                    throw(q); // Re-throw exception for non-local Label
28.                }
29.                JTX$=q.index; continue LOOP$; // GOTO Local L
30.            }
31.        }
32.        EBLK();
33.        return(null);
34.    }
35. }
```

6.1.1 Label Quantities.

At source line 3. the label 'L' is declared like this:

```
final LABQNT$ L=new LABQNT$(this,1); // Local Label #1=L
```

Where LABQNT$ is defined by:

```
public final class LABQNT$ extends RuntimeException
{ public RTObject$ SL$; // Static link, block in which the label is defined.
  public int index; // Ordinal number of Label within its Scope(staticLink).

  public LABQNT$(RTObject$ SL$,int index)
  { this.SL$=SL$; this.index=index; }
}
```

A goto-statement is simply coded as:

```
throw(L); // GOTO EVALUATED LABEL
```

And this exception is caught and tested (lines 23 - 38) throughout the operating chain. If the label does not belong to this block instance the exception is rethrown.

In the event of no matching block instances the exception is caught by an UncaughtExceptionHandler like this:

```
public void uncaughtException(Thread thread, Throwable e) {
    if(e instanceof LABQNT$) {
        // POSSIBLE GOTO OUT OF COMPONENT
        RTObject$ DL=obj.DL$;
        if(DL!=null && DL!=CTX$) {
            DL.PENDING_EXCEPTION$=(RuntimeException)e;
            DL.CORUT$.run();
        } else {
            ERROR("Illegal GOTO "+e);
            ...
        } else ...
 }
```

Thus, when a QPS-component is left we raise the PENDING_EXCEPTION flag and resume next operating component. The resume-operations will retrow the exception within its Thread.

For further details see the source code of RTObject.java

6.1.2 Byte Code Engineering.

The ObjectWeb ASM Java bytecode engineering library from the OW2 Consortium is used to modify the byte code to allow very local goto statements.

They say:

ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or to dynamically generate classes, directly in binary form. ASM provides some common bytecode transformations and analysis algorithms from which custom complex transformations and code analysis tools can be built. ASM offers similar functionality as other Java bytecode frameworks, but is focused on performance. Because it was designed and implemented to be as small and as fast as possible, it is well suited for use in dynamic systems (but can of course be used in a static way too, e.g. in compilers).

More info at: https://asm.ow2.io and https://www.ow2.org/

To easily modify the code, the Simula Compiler generates certain method call in the .java file:

- `LABEL$(n); // Label #n`

  This method-call is used to signal the occurrence of a Simula Label. The bytecode address is collected and some instruction are removed. The parameter 'n' is the label's ordinal number.

  I.e. Try to locate the instruction sequence:

  ```
  PREV-INSTRUCTION
  ICONST n
  INVOKESTATIC LABEL$
  NEXT-INSTRUCTION
  ```

  Pick up the number 'n', remember address and remove the two middle instruction.

- `JUMPTABLE$(JTX$); // For ByteCode Engineering`

  This method-call is a placeholder for where to put in a Jump-Table.

  Try to locate the instruction sequence:

  ```
  PREV-INSTRUCTION
  GETFIELD JTX$
  INVOKESTATIC JUMPTABLE$
  NEXT-INSTRUCTION
  ```

  And replace it by the instruction sequence:

  ```
  PREV-INSTRUCTION
  GETFIELD JTX$
  TABLESWITCH ... uses the addresses collected for labels.
  NEXT-INSTRUCTION
  ```

For further details see the source code of ByteCodeEngineering.java

6.2 For-Statement

The Implementation of the for-statement is a bit tricky. The basic idea is to create a ForList iterator that iterates over a set of ForElt iterators. The following subclasses of ForElt are defined:

- SingleElt<T> for basic types T control variable
- SingleTValElt for Text type control variable
- StepUntil for numeric types
- WhileElt<T> for basic types T control variable
- WhileTValElt representing For t:= <TextExpr> while <Cond>
  With text value assignment

Each of which deliver a boolean value 'CB' used to indicate whether this for-element is exhausted. All parameters to these classes are transferred 'by name'. This is done to ensure that all expressions are evaluated in the right order. The assignment to the 'control variable' is done within the various for-elements when the 'next' method is invoked. To get a full overview of all the details you are encouraged to study the generated code together with the 'FRAMEWORK for for-list iteration' found in the runtime class RTObject$.

Example, the following for-statement:

```
for i:=1,6,13 step 6 until 66,i+1 while i<80 do j:=j+i;
```

Is compiled to:

```
for(boolean CB:new ForList(
    new SingleElt<Number>(...)
   ,new SingleElt<Number>(...)
   ,new StepUntil(...)
   ,new WhileElt<Number>(...)
)) { if(!CB) continue;
     j=j+i;
   }
```

Another example with control variable of type Text:

```
for t:="one",other while k < 7 do  <statement>
```

Where 'other' is a text procedure, is compiled to:

```
for(boolean CB:new ForList(
    new SingleTValElt(...)
   ,new WhileTValElt(...)
 )) { if(!CB) continue;
       … // Statement
    }
```

Optimized For-Statement

However; most of the for-statements with only one for-list element are optimized.

Single for step-until statements are optimized when the step-expression is constant. I.e. the following for-statements:

for i:=<expr-1> step 1 until <expr-2> do <statements>
for i:=<expr-1> step -1 until <expr-2> do <statements>

for i:=<expr-1> step 6 until <expr-2> do <statements>
for i:=<expr-1> step -6 until <expr-2> do <statements>

are compiled to:

```
for(i = <expr-1>; r <= <expr-2>; r++) { <statements> }
for(i = <expr-1>; r >= <expr-2>; r--) { <statements> }

for(i = <expr-1>; r <= <expr-2>; r=r+6) { <statements> }
for(i = <expr-1>; r >= <expr-2>; r=r-6) { <statements> }
```

The other kinds of single elements are optimized in these ways:

for i:=<expr> do <statements>

for i:=<expr> while <cond> do <statements>

are compiled to:

```
i = <expr>; { <statements> }

i = <expr>;
While( <cond> ) {
       <statements>;
       i = <expr>;
}
```

6.3 Connection Statement

The connection statement is implemented using Java's **instanceof** operator and the **if** statement. For example, the connection statement:

inspect x **do** image:-t;

Where 'x' is declared as a reference to an ImageFile, is compiled to:

```
if(x!=null) x.image=t;
```

Other examples that also use 'ref(Imagefile) x' may be:

1) **inspect** x **do** image:-t **otherwise** t:-**notext**;

2) **inspect** x
   when infile **do** t:-intext(12)
   when outfile **do** outtext(t);

3) **inspect** x
   when infile **do** t:-intext(12)
   when outfile **do** outtext(t)
   otherwise t:-**notext**;

These examples are compiled to:

1)
```
if(x!=null) x.image=t; else t=null;
```

2)
```
if(x instanceof InFile$ infile) t=inFile.intext(12);
else if(x instanceof OutFile$ outfile) outfile.outtext(t);
```

3)
```
if(x instanceof InFile$ in) t=in.intext(12);
else if(x instanceof OutFile$ out) out.outtext(t);
else t=null;
```

6.4 Switch Statement

The Switch Statement is an extension inherited from S-Port Simula.

Syntax:

Switch-statement
= **SWITCH** (lowKey : hiKey) switchKey **BEGIN** { switch-case } [none-case] **END**

switch-case = **WHEN** <caseKey-list> **do** <statement> ;
none-case = **WHEN NONE do** <statement> ;
<caseKey-list> = caseKey { , caseKey }
caseKey = caseConstant | caseConstant : caseConstant
lowKey = integer-or-character-expression
hiKey = integer-or-character-expression
switchKey = integer-or-character-expression
caseConstant = integer-or-character-constant

Translated into a Java Switch Statement with no break after each <statement>.

For example, the following Switch Statement::

```
switch(lowkey:hikey) key
begin
        when 1 do begin
                !Statements;
                goto AFTER;
        end;

        when 2 do begin
                !Statements;
                goto AFTER;
        end;

        when NONE do begin
                !Statements;
                goto AFTER;
        end;
end switch;
AFTER:
```

Is compiled to: (lowkey and hikey are ignored)

```
switch(key) {
        case 1:
                // Statements …
                GOTO$(AFTER$); // GOTO EVALUATED LABEL
        case 2:
                // Statements ...
                GOTO$(AFTER$); // GOTO EVALUATED LABEL
        default:
                // Statements ...
                GOTO$(AFTER$); // GOTO EVALUATED LABEL
 }
LABEL$(1,"AFTER$");
```

6.5 Activation Statement

The activation statement is defined by the procedure ACTIVAT in Simula Standard.
In this implementation we use a set of methods for the same purpose:

activate x;	==>	`ActivateDirect(false,x);`
activate x delay 1.34;	==>	`ActivateDelay(false,x,1.34f,false);`
activate x delay 1.34 prior;	==>	`ActivateDelay(false,x,1.34f,true);`
activate x at 13.7;	==>	`ActivateAt(false,x,13.7f,false);`
activate x at 13.7 prior;	==>	`ActivateAt(false,x,13.7f,true);`
activate x before y;	==>	`ActivateBefore(false,x,y);`
activate x after y;	==>	`ActivateAfter(false,x,y);`
reactivate x;	==>	`ActivateDirect(true,x);`
reactivate x delay 1.34;	==>	`ActivateDelay(true,x,1.34f,false);`
reactivate x delay 1.34 prior;	==>	`ActivateDelay(true,x,1.34f,true);`
reactivate x at 13.7;	==>	`ActivateAt(true,x,13.7f,false);`
reactivate x at 13.7 prior;	==>	`ActivateAt(true,x,13.7f,true);`
reactivate x before y;	==>	`ActivateBefore(true,x,y);`
reactivate x after y;	==>	`ActivateAfter(true,x,y);`

See runtime module Simulation$ for details.

The sequencing set is implemented by java.util.TreeSet.
See runtime module SequencingSet$ for details.

7. Sequencing

7.1 Co-routines: detach - call

Simula has three primitives to express Sequencing:

1) obj.Detach – Suspends the execution, saves a reactivation point and returns.
2) Call(obj) – Restarts a detached object at the saved reactivation point.
3) Resume(obj) – Suspend and restart another object.

The (detach,call) pair constitutes co-routines in Simula while the (detach,resume) pair establish Symmetric component sequencing used to implement discrete event simulation. For details on the implementation, see the runtime system code.

A small Java class 'Coroutine' is used to implement the qps primitives.

```
public class Coroutine implements Runnable {
    public Coroutine(Runnable target)
    public static Coroutine getCurrentCoroutine()
    public final void run()
    public static void detach()
    public boolean isDone()
    ....
}
```

A coroutine is a program object representing a computation that may be suspended and resumed. The implementation use Threads to constitute separate stacks.

To illustrate the transition of the pointers when executing the qps primitives we will use the following program example.

```
System begin ref(C) x;
        procedure P;
        begin x:-new C();
                …
                call(x);
        end;
        class Component;
        begin
                procedure Q;
                begin ... detach; ... end;
                ...
                Q;
                ...
        end Component;
        ...
        P;
        ...
end System;
```

The following pages show the transitions in detail.

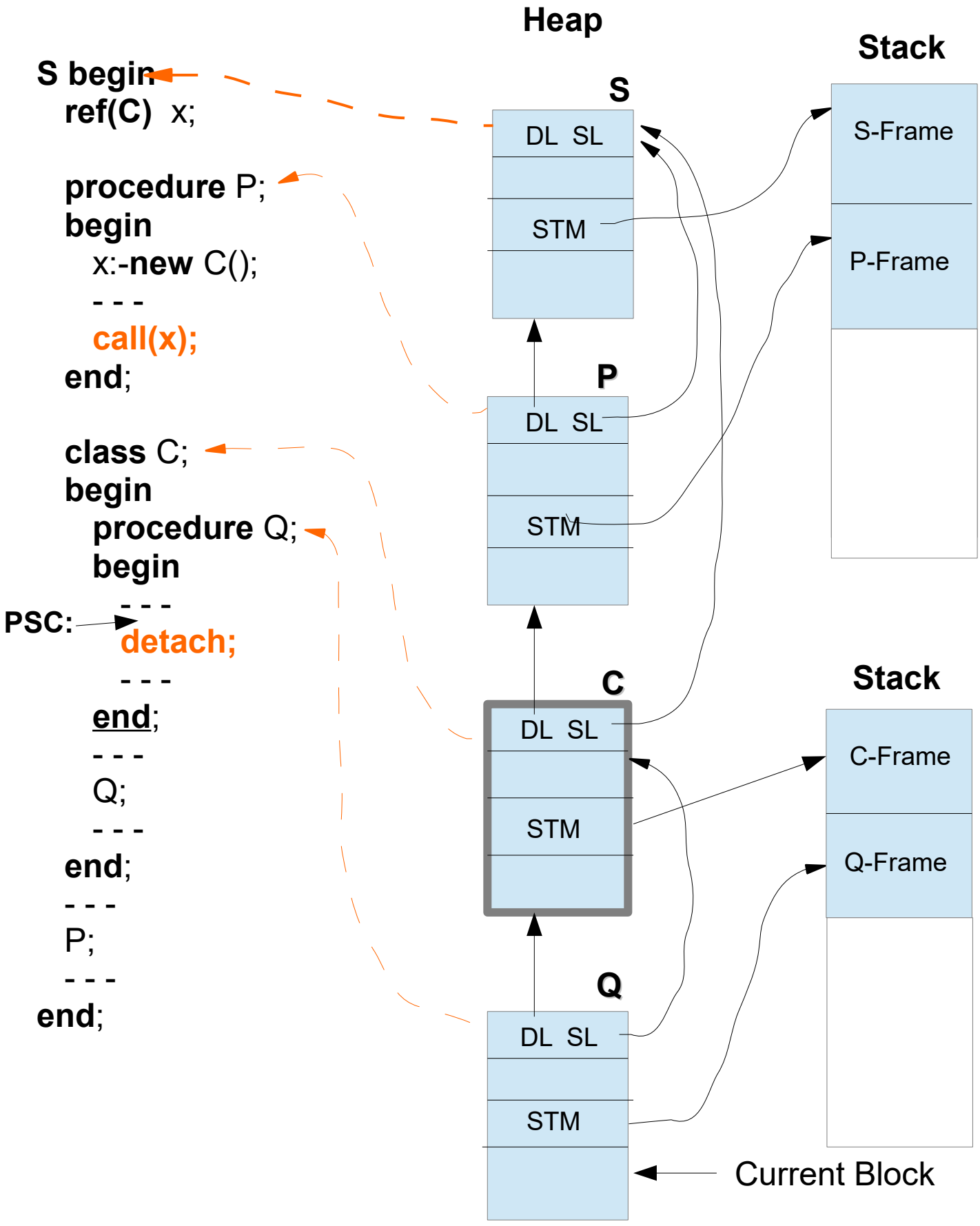

Fig. 2: Just before Detach

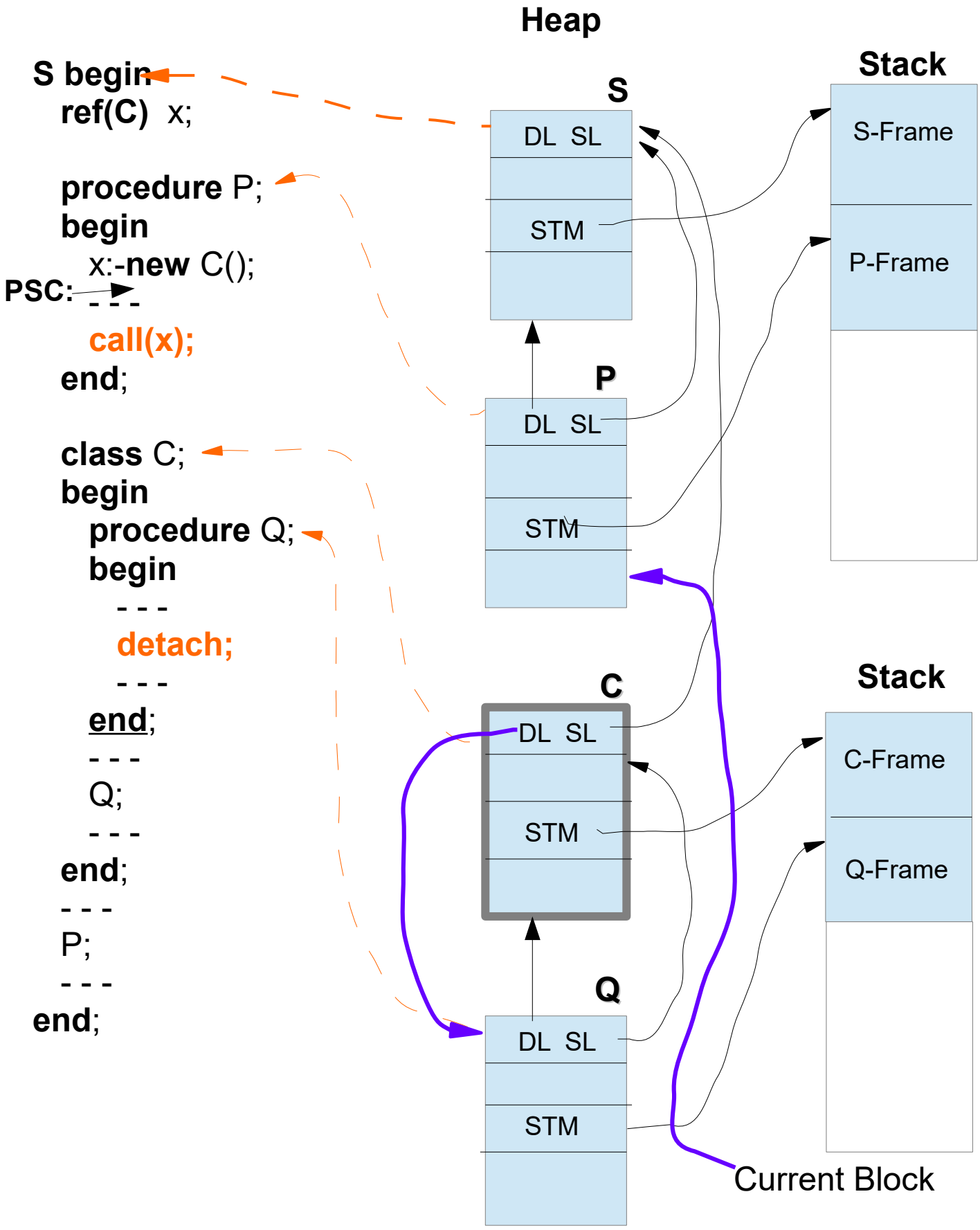

Fig. 3: Just after Detach – C-Object is dismounted

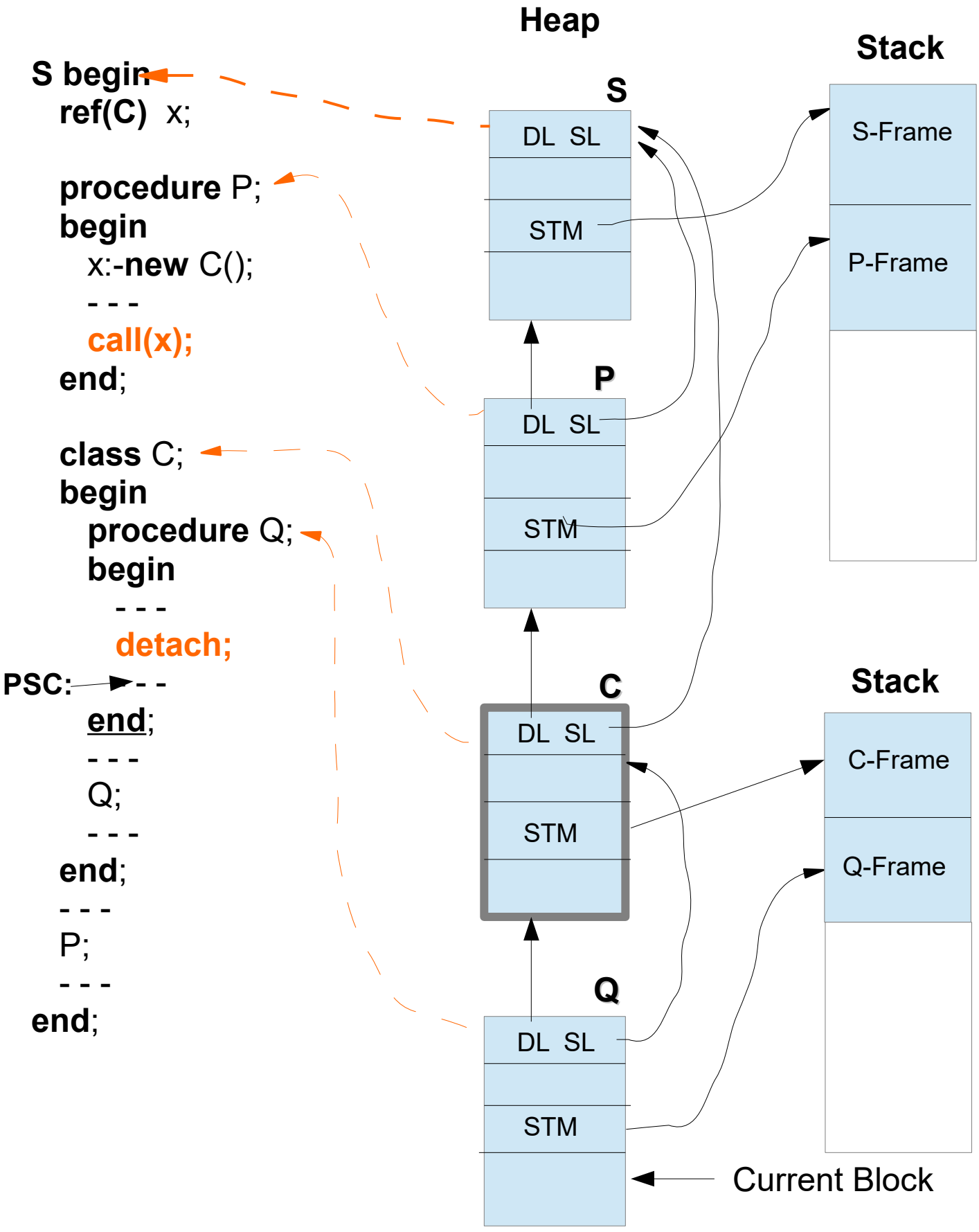

Fig. 4: Just after Call – C-Object is re-mounted

7.2 Symmetric component sequencing: detach - resume

Symmetric component sequencing, the (detach,resume) pair, is used to implement discrete event simulation. This is a complicated implementation that was largely inherited from Simula I. It is believed that this should have been implemented using ordinary coroutines.

For details on the implementation, see the runtime system code.

To illustrate the transition of the stack when executing the qps primitives we will use the following program example.

```
System begin ref(C) x;
        procedure P;
        begin x:-new C();
                …
                resume(x);
        end;
        class Component;
        begin
                procedure Q;
                begin
                        while … do
                        begin ... detach; … end;
                end;
                ...
                Q;
                ...
        end Component;
        ...
        P;
        ...
end System;
```

The following pages show the transitions in detail.

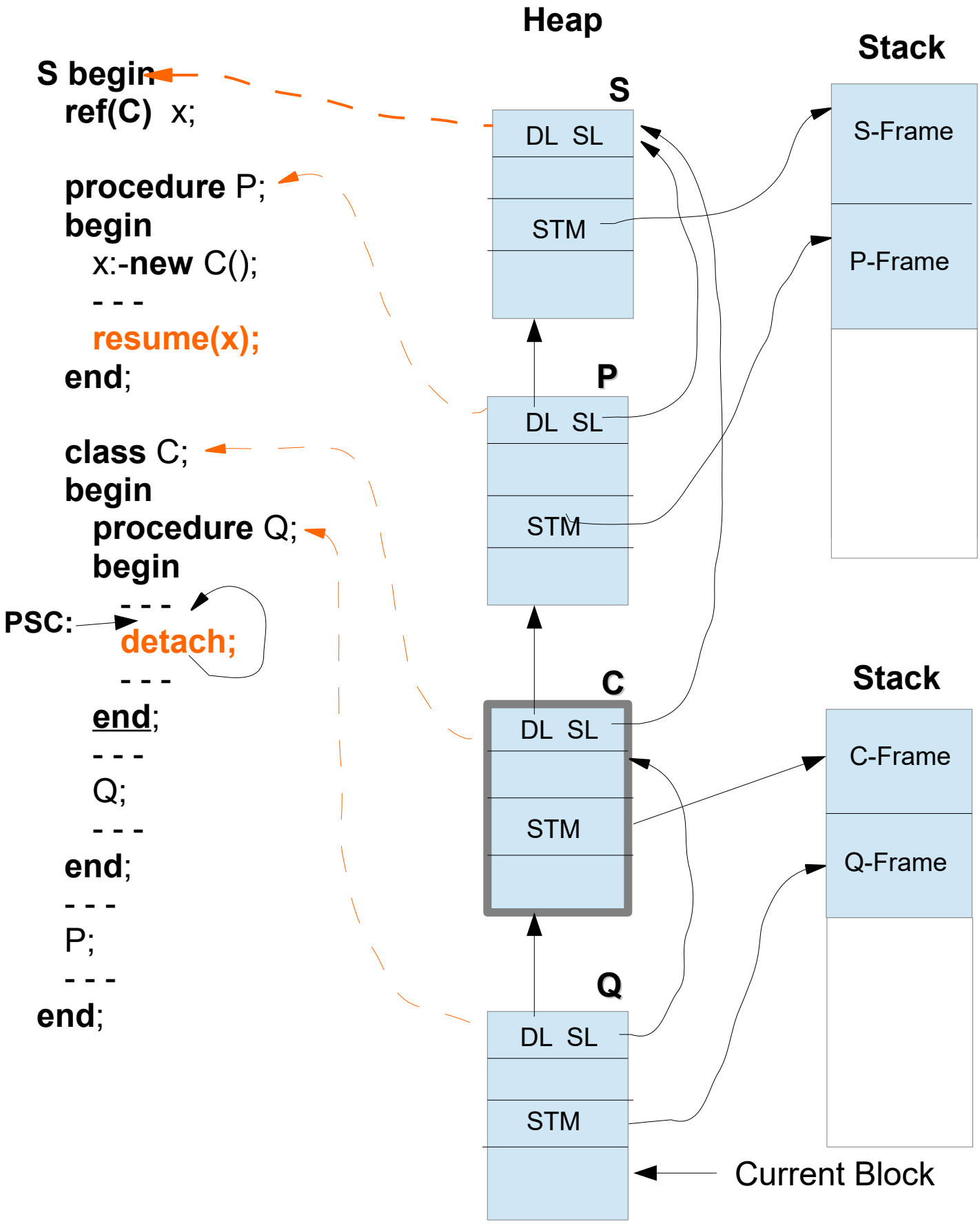

Fig. 5: Just before 1. Detach

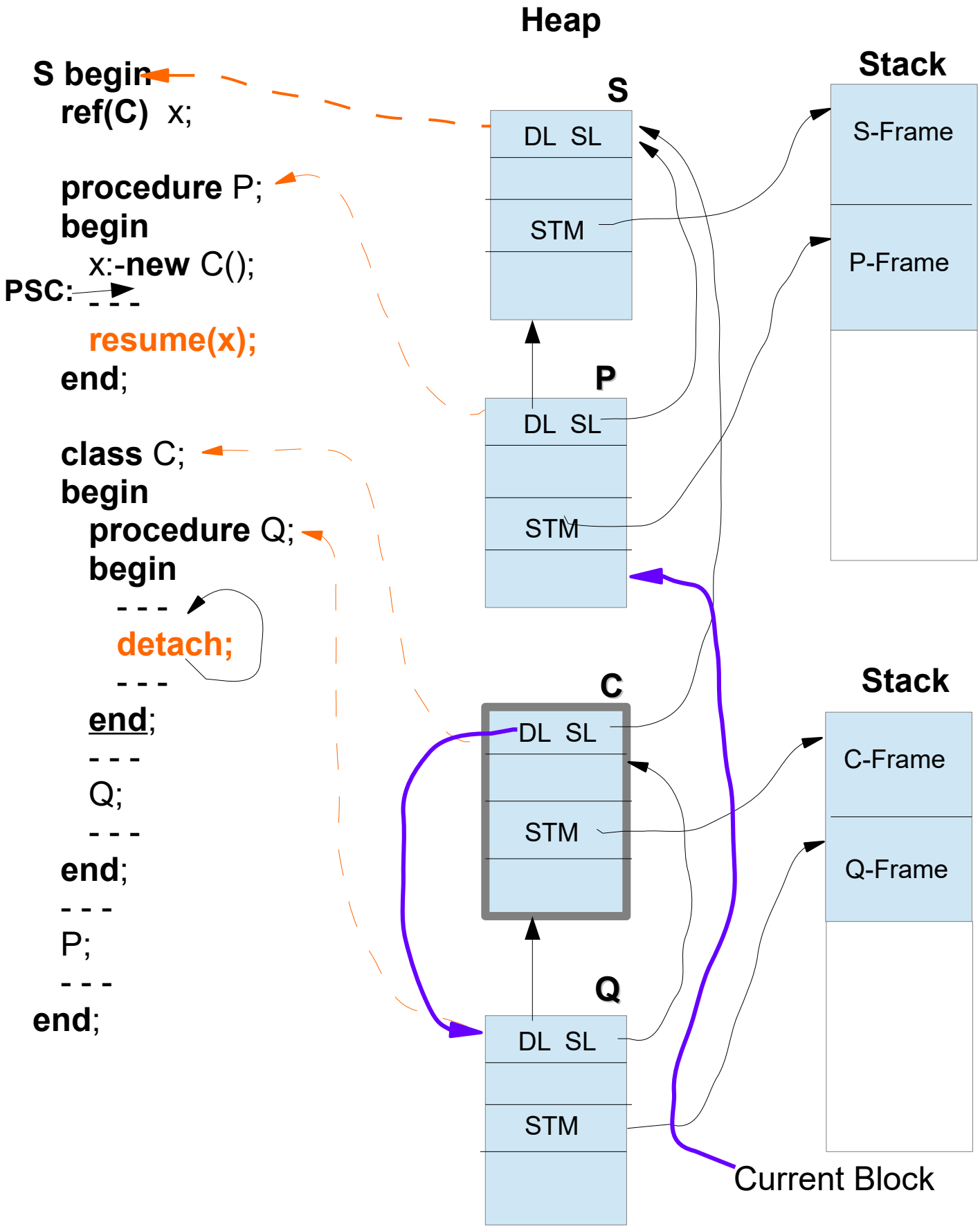

Fig. 6 Just after 1.Detach and before Resume

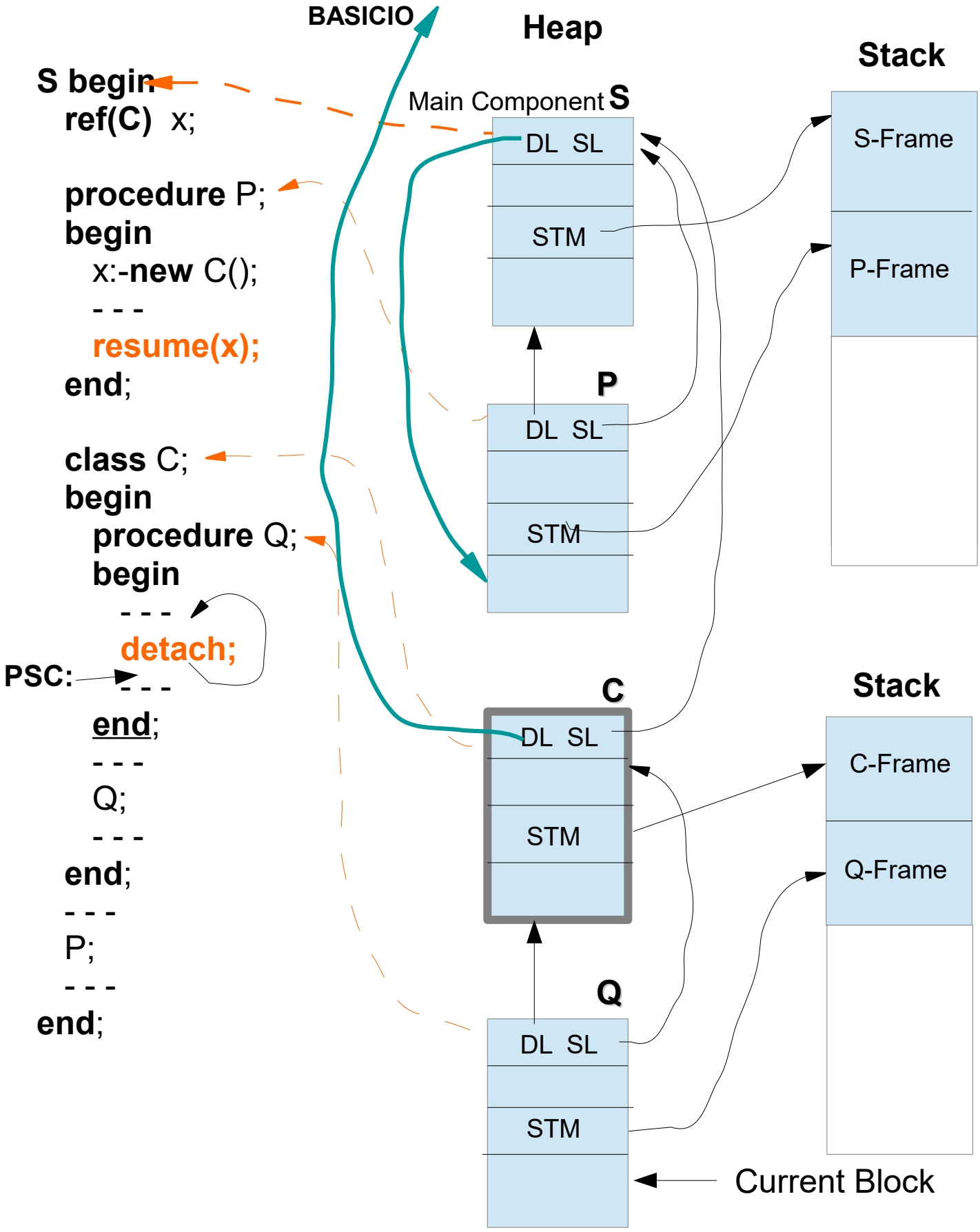

Fig. 7: Just after Resume

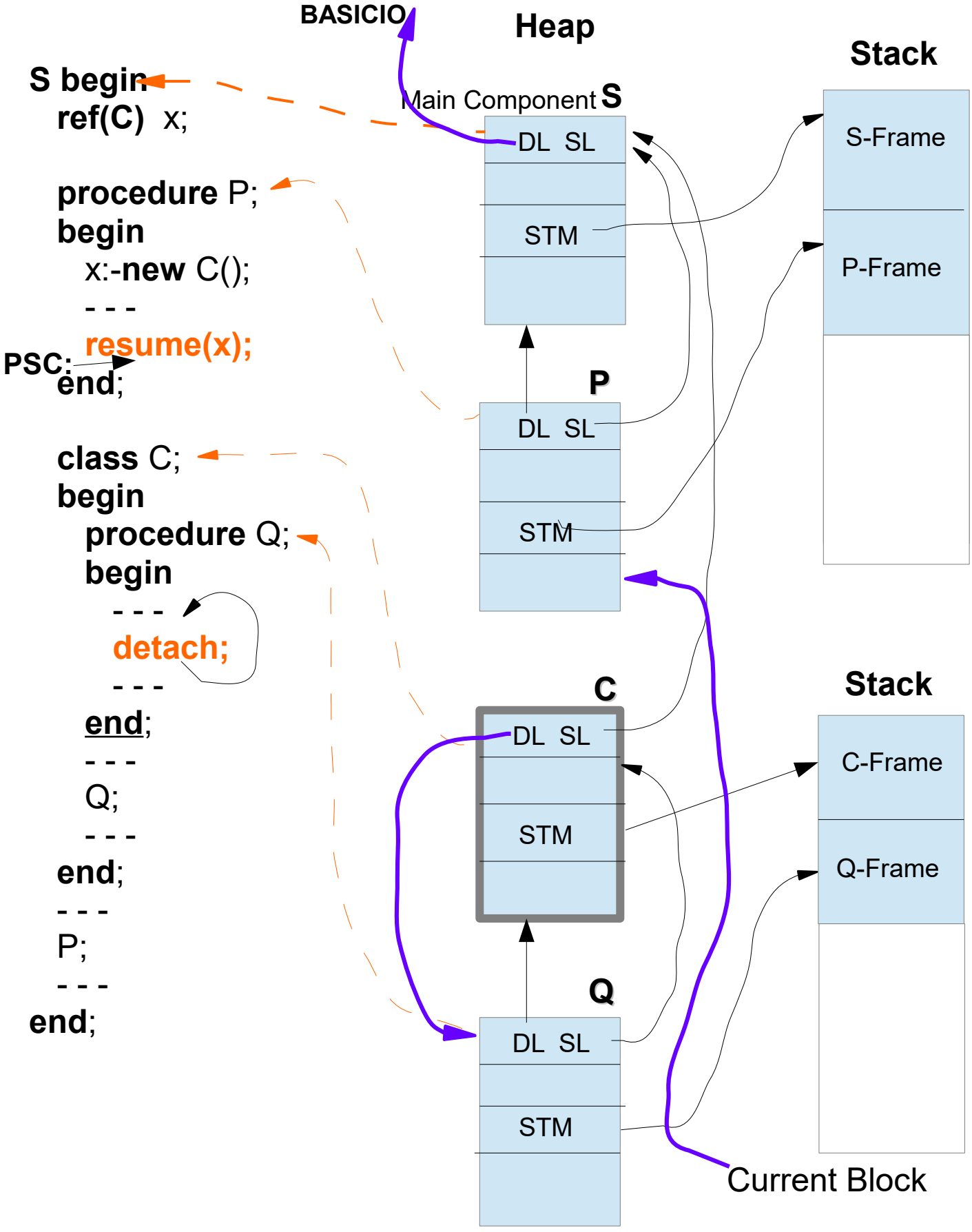

Fig. 8: Just after 2. Detach