KENZO

a Symbolic Software
for
Effective Homology Computation

—

Julio Rubio Garcia
Francis Sergeraert
Yvon Siret

—

Institut Fourier
Grenoble, France

—

This document is a user guide for the basic modules of the Effective Homology Software (Kenzo Version 1998).
It is also available by anonymous ftp (**ftp fourier.ujf-grenoble.fr**) in the directory **/pub/KENZO**.

April 26, 1999

FOREWORD

The **Kenzo** program implements the general ideas of the second author about *Effective Homology*[1], mainly around the Serre and Eilenberg–Moore spectral sequences. The first author (re–) discovered the importance of the *Basic Perturbation Lemma* in these questions, already noted by Victor Gugenheim[2] and this program directly implements and directly uses this "lemma" which should be called the *Fundamental Theorem of Algebraic Topology*. The first version of the program, called **EAT**, was written in 1989-90 by the first and the second authors. It has been demonstrated in several universities: France: Grenoble and Montpellier, Belgium: Louvain-la-Neuve, Italy: Genoa and Pisa, Sweden: Stockolm, Japan: Sapporo, Morioka, Urawa, Tokyo, Kyoto, Nara, Osaka and Hiroshima.
In this **Kenzo** version, numerous improvements have been integrated in comparison with **EAT**.

1. Standard CLOS (Common Lisp Object System) techniques.
2. Use of the Zermelo theorem about ordered sets (!): any set can be well ordered and this remark gives important new ideas to improve execution speed.
3. Better memory management.
4. Use of Szczarba's formulas instead of Shih's for implementing the twisted Eilenberg–Zilber theorem.
5. New mathematical objects:
   - Serre spectral sequences.
   - Differentials algebras.
   - Simplicial morphisms.
   - Kan simplicial sets.
   - Simplicial groups.

[1] **Francis Sergeraert.** *The computability problem in algebraic topology.* Advances in Mathematics, 1994, vol. 104, pp 1-29.

[2] **V.K.A.M. Gugenheim.** *On a perturbation theory for the homology of the loop space.* Journal of Pure and Applied Algebra, 1982, vol. 25, pp 197-205.

- Fibrations.
- Classifying spaces.

In particular, using these tools, the first homotopy groups of *arbitrary* simplicial sets with effective homology, are now reachable.

The present documentation is a joint work between the second and third authors.

Chapter 0

Overview

In this overview we try to show, without entering in detailed explanations, various possibilities of `Kenzo` in algebraic topology. The lines ended by the symbol ==> are the commands typed by the user (don't type this symbol!). They are followed by the answer of the program. We have suppressed some extra informations, in particular the ones printed during the computation of homology groups.

Let us begin by the space $Moore(\mathbb{Z}/2\mathbb{Z}, 3)$ described as a simplicial set having only three non–degenerate simplices, namely in dimension 0, 3 and 4. In the representation created by the software, the 0–simplex (base point), the 3–simplex and the 4–simplex are respectively labelled "`*`", `M3` and `N4`. Two faces of the 4-simplex `N4` are identified with the 3-simplex `M3`, the others being contracted on the base point. To create the simplicial set one types simply:

```
(setf m23 (moore 2 3))  ==>
```

The system answers that a `Kenzo` object has been created, with number 1 and type `SIMPLICIAL SET`. This object may be referenced by the symbol `m23`.

```
[K1 Simplicial-Set]
```

We may compute the homology groups of this space, using the underlying chain complex induced by the simplicial set description. Here we compute the H_i from 0 to 4 included. When in the answer the component part is void, it means that the corresponding homology group is null.

```
(homology m23 0 5)  ==>

Homology in dimension 0 :
```

```
Component Z

--done--

Homology in dimension 1 :

--done--

Homology in dimension 2 :

--done--

Homology in dimension 3 :

Component Z/2Z

--done--

Homology in dimension 4 :

---done---
```

As `m23` is a simplicial set, it is possible to create the cartesian product $m23 \times m23$ by the function `crts-prdc`. This is a new simplicial set.

```
(setf m23xm23 (crts-prdc m23 m23))  ==>

[K10 Simplicial-Set]
```

Being a simplicial set, `m23xm23` is also a chain complex object and we may for instance ask for the basis in dimension 6

```
(basis m23xm23 6)  ==>

(<CrPr 1-0 N4 3-2 N4> <CrPr 1-0 N4 4-2 N4> <CrPr 1-0 N4 4-3 N4>
 <CrPr 1-0 N4 4-3-2 M3>  <CrPr 1-0 N4 5-2 N4> <CrPr 1-0 N4 5-3 N4>
 <CrPr 1-0 N4 5-3-2 M3> <CrPr 1-0 N4 5-4 N4>
                              .... ....
 <CrPr 4-1 N4 5-3-2 M3> <CrPr 4-1-0 M3 3-2 N4> <CrPr 4-1-0 M3 5-2 N4>
 <CrPr 4-1-0 M3 5-3 N4> <CrPr 4-1-0 M3 5-3-2 M3> ... ... ...)

(length *)  ==>

230
```

As shown by the last command (* means the previous result), the number of elements of the basis is quite large (230). The user will note that the basis

elements are formed by cartesian products of *degenerated simplices*. In the list, an element like `<CrPr 1-0 N4 5-3-2 M3>` means $\eta_1\eta_0 N4 \times \eta_5\eta_3\eta_2 M3$. We may construct also the tensor product $m23 \otimes m23$ from the underlying chain complex of the simplicial set `m23`. This tensor product is a new chain complex and we see that the basis in dimension 6 has only one element:

```
(setf t2m23 (tnsr-prdc m23 m23)) ==>

[K3 Chain-Complex]

(basis t2m23 6)  ==>

(<TnPr M3 M3>)
```

The Eilenberg-Zilber theorem is used to compute the homology groups of the cartesian product space: as chain complexes, `m23xm23` and `t2m23` have the same homology groups, but the computations using the tensor product are considerably faster.

```
(homology m23xm23 0 8)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :    (meaning: Group null)

Homology in dimension 2 :

Homology in dimension 3 :

Component Z/2Z

Component Z/2Z

Homology in dimension 4 :

Homology in dimension 5 :

Homology in dimension 6 :

Component Z/2Z

Homology in dimension 7 :

Component Z/2Z

---done---
```

Let us consider now the space $K(\mathbb{Z},1)$. This is an Abelian simplicial group created in `Kenzo` by the function `k-z`. In this simplicial group, a simplex in dimension n is mathematically represented by a sequence of integers, known as a *bar* object:

$$[a_1 \mid a_2 \mid \ldots \mid a_n].$$

In `Kenzo`, a non-degenerate simplex of $K(\mathbb{Z},1)$ in dimension n will be simply a list of n non-null integers, for instance: `(2 3 4 5)`. In dimension 0, the only simplex is `NIL` (the base point).

```
(setf kz1 (k-z 1))  ==>

[K38 Abelian-Simplicial-Group]
```

But this object is also a *coalgebra* and an *algebra*, and we may see the effect of the respective induced *coproduct* and *product*:

```
(cprd kz1 4 '(2 3 4 5))  ==>

-------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr NIL (2 3 4 5)>>
<1 * <TnPr (2) (3 4 5)>>
<1 * <TnPr (2 3) (4 5)>>
<1 * <TnPr (2 3 4) (5)>>
<1 * <TnPr (2 3 4 5) NIL>>
-------------------------------------------------------------------------------

(aprd kz1 6 (tnpr 2 '(1 2) 4 '(3 4 5 6)))  ==>

-------------------------------------------------------------------------{CMBN 6}
<1 * (1 2 3 4 5 6)>
<-1 * (1 3 2 4 5 6)>
<1 * (1 3 4 2 5 6)>
<-1 * (1 3 4 5 2 6)>
<1 * (1 3 4 5 6 2)>
<1 * (3 1 2 4 5 6)>
<-1 * (3 1 4 2 5 6)>
<1 * (3 1 4 5 2 6)>
<-1 * (3 1 4 5 6 2)>
<1 * (3 4 1 2 5 6)>
<-1 * (3 4 1 5 2 6)>
<1 * (3 4 1 5 6 2)>
<1 * (3 4 5 1 2 6)>
<-1 * (3 4 5 1 6 2)>
<1 * (3 4 5 6 1 2)>
-------------------------------------------------------------------------------
```

The printed results are the printed representation of *combinations*, i.e. integer linear combinations of generators resulting from the application of the

morphisms. The degree of the combination is indicated by the information: `CMBN` n.
On the same way, we may create the Abelian simplicial groups $K(\mathbb{Z}/2\mathbb{Z}, n)$:

```
(setf k-z2-2 (k-z2 2))  ==>

[K488 Abelian-Simplicial-Group]

(homology k-z2-2 4)  ==>

Homology in dimension 4 :

Component Z/4Z

---done---
```

———-

Let us play now with the sphere S^3 and its loop spaces. S^3 and $\Omega^2 S^3$ are created by respective calls to the functions `sphere` and `loop-space`. Then we compute the H_4 and H_5 of $\Omega^2 S^3$:

```
(setf s3 (sphere 3))  ==>

[K52 Simplicial-Set]

(setf o2s3 (loop-space s3 2))  ==>

[K69 Simplicial-Group]

(homology o2s3 4 6)  ==>

Homology in dimension 4 :

Component Z/3Z

Component Z/2Z

Homology in dimension 5 :

Component Z/3Z

Component Z/2Z

---done---
```

Let us take now the first loop space $\Omega^1 S^3$

```
(setf s3 (sphere 3))  ==>

[K94 Simplicial-Set]

(setf os3 (loop-space s3))  ==>

[K99 Simplicial-Group]
```

In the following instruction, we locate in the symbol `L1` the canonical generator of $\pi_2(\Omega^1 S^3)$, that is the 2–simplex coming from the original sphere. In fact, the object created by the command `(loop3 0 's3 1)` is the "word" $S3^1$ belonging to the Kan simplicial version $G(S^3)$ (a simplicial group) of the loop space ΩS^3.

```
(setf L1  (loop3 0 's3 1))  ==>

<AbSm - <<Loop[S3]>>>
```

Let us consider also the 2–degeneracy of the base point of the loop space. In the printed result, the user will recognize the degeneracy $\eta_1\eta_0$ of the null loop, base point of $\Omega^1 S^3$:

```
(setf null-simp (absm 3 +null-loop+))  ==>

<AbSm 1-0 <<Loop>>>
```

We may build now a new space by pasting a disk $D3$ as indicated by the following call. It means that we "paste" to the space `os3` a 3–simplex named `D3`, the attaching map being described by the list of its faces in dimension 2.

```
(setf dos3 (disk-pasting os3 3 '<D3>
              (list L1 null-simp L1 null-simp)))  ==>

[K212 Simplicial-Set]
```

Let us compute a few homology groups of the new space `dos3`:

```
(homology dos3 2 4)  ==>

Homology in dimension 2 :

Component Z/2Z

Homology in dimension 3 :

---done---
```

But more interesting, let us build the loop space of the object `dos3` and let us compute the homology in dimension 5:

```
(setf odos3 (loop-space dos3))  ==>

[K230 Simplicial-Group]

(homology odos3 5)  ==>

Homology in dimension 5 :

Component Z/2Z

Component Z/2Z

Component Z/2Z

Component Z/2Z

Component Z/2Z

Component Z/2Z

Component Z

--done--
```

———

Let us continue with the Kan theory. First, we check that S^3 is not of type Kan and that ΩS^3 is indeed of type Kan and a non–Abelian simplicial group.

```
(typep s3 'kan)  ==>

NIL

(typep os3 'kan)  ==>

T

(typep os3 'simplicial-group)  ==>

T

(typep os3 'ab-simplicial-group)  ==>

NIL
```

Let us create the word $L2 = (S3)^2$, i.e an object belonging to ΩS^3 and let us apply the product of the underlying algebra upon $L2 \otimes L2$:

```
(setf L2 (loop3 0 's3 2))  ==>

<<Loop[S3\2]>>

(setf square (aprd os3 4 (tnpr 2 L2 2 L2)))  ==>

-------------------------------------------------------------------------{CMBN 4}
<1 * <<Loop[1-0 S3\2][3-2 S3\2]>>>
<-1 * <<Loop[2-0 S3\2][3-1 S3\2]>>>       <-------
<1 * <<Loop[2-1 S3\2][3-0 S3\2]>>>
<1 * <<Loop[3-0 S3\2][2-1 S3\2]>>>
<-1 * <<Loop[3-1 S3\2][2-0 S3\2]>>>
<1 * <<Loop[3-2 S3\2][1-0 S3\2]>>>
-------------------------------------------------------------------------------
```

We see that the result is a linear combination of words composed from degeneracies of `L2`. The following instruction selects the generator part of the second element of the previous combination (noted by the reverse arrow).

```
(setf L4 (gnrt (second (cmbn-list square))))  ==>

<<Loop[2-0 S3\2][3-1 S3\2]>>
```

Let us use the lisp function `mapcar` (one among the various iteration functions of Lisp) to create the list of the faces 1, 2, 3 and 4 of the object `L4`, this list is a "Kan hat".

```
(setf hat (mapcar #'(lambda (i) (face os3 i 4 l4)) '(1 2 3 4)))  ==>

(<AbSm - <<Loop[1 S3\2][2 S3\2]>>> <AbSm - <<Loop[0 S3\2][2 S3\2]>>>
 <AbSm - <<Loop[0 S3\2][1 S3\2]>>> <AbSm 1 <<Loop[S3\2]>>>)
```

The function `kfll` tries to find a filling of this "Kan hat", and we see that the face 2 of the resulting simplex (which is very different from `L4`) is the same as the face 2 of `L4`.

```
(setf kan-simplex (kfll os3 0 4 hat))  ==>

<AbSm - <<Loop[3-1 S3\2][2-1 S3\-2][2-0 S3\2][1-0 S3\-2][2-1 S3\2][3-1 S3\-2]
              [1-0 S3\2][3-1 S3\2][3-0 S3\-2][1-0 S3\-2][3-0 S3\2][2-0 S3\-2]
              [1-0 S3\2][3-0 S3\-2][2-0 S3\2][3-0 S3\2]>>>

(face os3 2 4 kan-simplex)  ==>

<AbSm - <<Loop[0 S3\2][2 S3\2]>>>
```

```
(second hat)  ==>

<AbSm - <<Loop[0 S3\2][2 S3\2]>>>
```

———

Let $\mathcal{G}$ be a simplicial group 0–reduced. ΩS^3 is such a group. The software `Kenzo` allows the construction of the universal bundle $\bar{W}\mathcal{G}$, i.e. the classifying space of $\mathcal{G}$. In our case, as ΩS^3 in non–Abelian, the result is not a simplicial group but only a simplicial set. We verify that the H_4 is null.

```
(setf cls-os3 (classifying-space os3))  ==>

[K598 Simplicial-Set]

(typep cls-os3 'simplicial-group)  ==>

NIL

(homology cls-os3 4)  ==>

Homology in dimension 4 :

---done---
```

———

Let us end this short overview with an example of computation of homotopy groups. The method used in `Kenzo` is the Whitehead tower. In this current version only the case where the first non–null homology group (in non–null dimension) is $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$ can be processed; however if this homology group is a direct sum of several copies of $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$, then the corresponding stage of the Whitehead tower may also be constructed step by step.

We take again $Moore(\mathbb{Z}/2\mathbb{Z}, 3)$ whose H_3 is $\mathbb{Z}/2\mathbb{Z}$. First the fundamental cohomology class is constructed:

```
(setf ch3 (chml-clss m23 3))  ==>

[K729 Cohomology-Class on K1 of degree 3]
```

Then the function `z2-whitehead` is called to build a fibration over the simplicial set `m23` canonically associated to the cohomology class `ch3`.

```
(setf f3 (z2-whitehead m23 ch3))  ==>

[K730 Fibration K1 -> K488]
```

Then the total space of the fibration is built:

```
(setf x4 (fibration-total f3))  ==>

[K736 Simplicial-Set]
```

The H_4 of this total space is the π_4 of $Moore(\mathbb{Z}/2\mathbb{Z})$:

```
(homology x4 3 5)  ==>

Homology in dimension 3 :

---done---

Homology in dimension 4 :

Component Z/2Z

---done---
```

We may now iterate the process, to compute the π_5 of $Moore(\mathbb{Z}/2\mathbb{Z})$:

```
(setf ch4 (chml-clss x4 4))  ==>

[K817 Cohomology-Class on K802 of degree 4]

(setf f4 (z2-whitehead x4 ch4))  ==>

[K832 Fibration K736 -> K818]

(setf x5 (fibration-total f4))  ==>

[K838 Simplicial-Set]

(homology x5 4 6)  ==>

Homology in dimension 4 :

---done---

Homology in dimension 5 :

Component Z/4Z

---done---
```

So $\pi_5(Moore(\mathbb{Z}/2\mathbb{Z}))$ is $\mathbb{Z}/4\mathbb{Z}$.

Contents

Chapter 1

Chain Complexes

1.1 Introduction

A *chain complex*[1] (C_p, d_p) is a collection of free $\mathbb{Z}$–modules (C_p), one for each $p \in \mathbb{Z}$, together with a homomorphism $d_p : C_p \to C_{p-1}$, such that, for all p, $d_{p-1} \circ d_p = 0$.
In the Kenzo program, a *morphism* $f = (f_p)$, of degree k, from a chain complex (C_p, d_p) to another (C'_p, d'_p) is a collection of homomorphisms

$$f_p : C_p \to C'_{p+k}.$$

This is expressed by the following diagram, generally **not assumed commutative**.

$$\begin{array}{ccccccccc}
\cdots & \leftarrow & C_{p-1} & \xleftarrow{d_p} & C_p & \xleftarrow{d_{p+1}} & C_{p+1} & \leftarrow & \cdots \\
 & & \downarrow{\scriptstyle f_{p-1}} & & \downarrow{\scriptstyle f_p} & & \downarrow{\scriptstyle f_{p+1}} & & \\
\cdots & \leftarrow & C'_{p+k-1} & \xleftarrow{d'_{p+k}} & C'_{p+k} & \xleftarrow{d'_{p+k+1}} & C'_{p+k+1} & \leftarrow & \cdots
\end{array}$$

[1] **P.J. Giblin** in *Graph, Surface and Homology*, Chapman and Hall Math. series, 1981.

Three types of morphisms are most generally considered.

1. $k = 0$. If the commutativity relation $d'_p \circ f_p = f_{p-1} \circ d_p$ holds for every p, then the morphism f is an ordinary chain complex morphism or *chain map*.

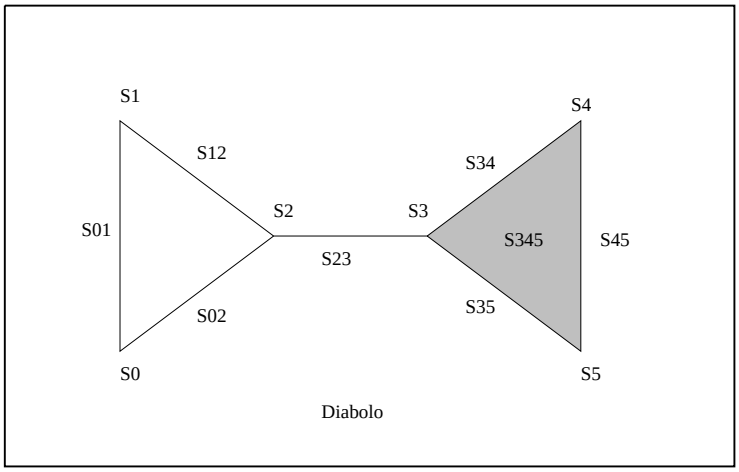
2. $k = -1$. If $(C_p, d_p) = (C'_p, d'_p)$ and $f_p = d_p$, then f is the *differential* of the chain complex (C_p, d_p) and, in fact, this differential is implemented in the `Kenzo` program as a morphism of degree -1.

3. $k = +1$. In this case, f is usually a *homotopy operator*, that is, some relation
$$d'_{p+1} \circ f_p + f_{p-1} \circ d_p = g_p - g'_p$$
is satisfied for two (ordinary) chain complex morphisms g and g'.

For technical reasons, these three types of morphisms have been implemented in the `Kenzo` program in a unique type.

1.2 Generators, terms and combinations

To become familiar to the lisp functions implementing the chain complexes, the best is to begin by an example of chain complex. The *simplicial complexes* are good candidates for this purpose and we shall take as typical example the following simplicial complex.



In this simplicial complex, called here *diabolo*, there are 3 associated chain groups.

- C_0, the free $\mathbb{Z}$–module on the set of vertices $\{s_0, s_1, s_2, s_3, s_4, s_5\}$.
- C_1, the free $\mathbb{Z}$–module on the set of edges $\{s_{01}, s_{02}, s_{12}, s_{23}, s_{34}, s_{35}, s_{45}\}$.
- C_2, the free $\mathbb{Z}$–module on the set of triangles (here a singleton) $\{s_{345}\}$.

The elements of either of those groups C_p are linear integer combinations of the corresponding basis (set of σ_i's), i.e. elements of the form:

$$\sum \lambda_i \sigma_i, \quad \lambda_i \in \mathbb{Z}.$$

An element σ_i of any basis is also called *generator* and in our specific case this generator will be represented by a lisp symbol. For instance, s_{45} will be translated in `s45`. But, the user must know from now, that in the realistic usage of the software, generators may be of any type. A product such as $\lambda_i \sigma_i$ is called a *term* and a sum of terms, a *combination.*

1.2.1 Representation of a combination

A combination is represented internally in the system by a list having the general following form:

(`:cmbn` *degree* $(\lambda_1.\sigma_1)$ ... $(\lambda_k.\sigma_k)$)

and containing

1. The degree of the combination corresponding to the index $p \in \mathbb{Z}$ of the group C_p to which this combination belongs.
2. The list of the internal representation of the terms, namely the list of pairs $(\lambda_i.\sigma_i)$.

This choice of representation implies that only homogeneous combinations will be considered. A type `CMBN` and a printing method have been associated to this internal representation. The external form of a combination is shown in the examples.

1.2.2 Ordering the generators

In order to speed up the execution of algorithms involving combinations, the list of pairs $(\lambda_i.\sigma_i)$ is ordered by an adequate ordering function (ex: the lexicographical ordering on the symbols). For programming convenience, an enumerated type `CMPR` has been defined:

```
(deftype cmpr() '(member :less :equal :greater))
```

A number of macros, functions and methods have been defined on usual sets (symbols, numbers, lists, ...) taking their value in the set `[:less, :equal, :greater]`. Of course, the user may define its own function for a particular case. There exists functions to compare various couples of usual items:

`f-cmpr` *n1 n2* *[Function]*
: Return `:less`, `:equal`, `:greater`, according to the result of the canonical comparison of both integers *n1* and *n2*.

`s-cmpr` *symbol1 symbol2* *[Function]*
: Return `:less`, `:equal`, `:greater`, according to the result of the lisp comparison functions on the strings `(symbol-name` *symbol1*`)` and `(symbol-name` *symbol2*`)`

`l-cmpr` *list1 list2* *[Function]*
: Return `:less`, `:equal`, `:greater`, according to the lexicographical ordering of the two lists *list1* and *list2* representing legal generators in this implementation.

Examples

```
(f-cmpr 123 789)  ==>

:LESS

(s-cmpr 'circulation 'circular)  ==>

:GREATER

(s-cmpr 'qwerty 'qwerty)  ==>

:EQUAL

(l-cmpr '(1 a b) '(1 a))  ==>

:GREATER
```

1.2.3 Functions handling combinations

The software provides a set of functions, methods or macros to create or modify combinations.

`term-cmbn` *dgr cf gnr* *[Macro]*
: Construct the combination of degree *dgr* with unique term $cf * gnr$

`cmbn` *dgr cf1 gnr1 cf2 gnr2 ... cfn gnrn* *[Function]*
: Construct a combination of degree *dgr*, sum of the terms $cf_i * gnr_i$. The sequence of pairs $\{cf_i\ gnr_i\}$ has an undefinite length and may be void. In this case, the combination is a null combination of degree *dgr*.

`cmbn-p` *object* *[Function]*
: Test is *object* is a legal combination.

`cmbn-degr` *cmbn* *[Macro]*
: Get the degree (an integer) of the combination *cmbn*.

`cmbn-list` *cmbn* *[Macro]*
: Get the list of the terms of the combination *cmbn*. Beware: a term is not a `Kenzo` object. One may select the coefficient (an integer) or the generator – a `Kenzo` object – respectively by the macros `cffc` and `gnrt`.

`zero-cmbn` *dgr* *[Function]*
: Create an instance of a null combination in the degree *dgr*.

`zero-pure-dffr` *cmbn* *[Function]*
: Create a null combination of degree $degr(cmbn) - 1$.

`cmbn-zero-p` *cmbn* *[Macro]*
: Test if *cmbn* is a null combination in any degree.

`cmbn-non-zero-p` *cmbn* *[Macro]*
: Test if *cmbn* is a non-null combination in any degree.

`cmbn-opps` *cmbn* *[Function]*
: Create a combination opposite to *cmbn*.

`n-cmbn` *n cmbn* *[Function]*
: Create a combination multiple of *cmbn* by the factor n.

`2cmbn-add` *cmpr cmbn1 cmbn2* *[Function]*
: Create a combination, sum of both combinations *cmbn1* and *cmbn2*. The first argument, *cmpr*, must be a function or macro relevant to compare the generators of the involved combinations, in order to return an ordered combination.

`ncmbn-add` *cmpr cmbn1 cmbn2 ... cmbnk* *[Function]*
Create a combination, sum of the indefinite number of combinations $cmbn_i$. As to the first argument, *cmpr*, see the function `2cmbn-add`.

`2n-2cmbn` *cmpr n1 cmbn1 n2 cmbn2* *[Function]*
Build the combination $n1 * cmbn1 + n2 * cmbn2$. Both integers $n1$ and $n2$ must be non null. As to the first argument, *cmpr*, see the function `2cmbn-add`.

`2cmbn-sbtr` *cmpr cmbn1 cmbn2* *[Function]*
Create a combination, difference of *cmbn1* and *cmbn2*. As to the first argument, *cmpr*, see the function `2cmbn-add`.

Examples

```
(setf comb1 (cmbn 1 1 'u 2 'v 3 'w 4 'z))  ==>
------------------------------------------------------------------------{CMB 1}
<1 * U>
<2 * V>
<3 * W>
<4 * Z>
------------------------------------------------------------------------

(cmbn-non-zero-p comb1)  ==>

T

(cmbn-list comb1)  ==>

((1 . U) (2 . V) (3 . W) (4 . Z))

(setf term3 (third *))  ==>  ;; not a Kenzo object!

(3 . W)

(cffc term3)  ==>

3

(gnrt term3)  ==>

W
```

```
(setf mcomb1 (cmbn-opps comb1))   ==>

--------------------------------------------------------------------------{CMB 1}
<-1 * U>
<-2 * V>
<-3 * W>
<-4 * Z>
------------------------------------------------------------------------------

(setf comb2 (n-cmbn 10 comb1))  ==>

--------------------------------------------------------------------------{CMB 1}
<10 * U>
<20 * V>
<30 * W>
<40 * Z>
------------------------------------------------------------------------------

(setf cmb12 (2cmbn-add #'s-cmpr comb1 comb2))  ==>

--------------------------------------------------------------------------{CMBN 1}
<11 * U>
<22 * V>
<33 * W>
<44 * Z>
------------------------------------------------------------------------------

(2cmbn-sbtr #'s-cmpr comb1 cmb12)  ==>

--------------------------------------------------------------------------{CMBN 1}
<-10 * U>
<-20 * V>
<-30 * W>
<-40 * Z>
------------------------------------------------------------------------------

(ncmbn-add #'s-cmpr
           comb1 comb2 comb1 comb2 comb1 comb2 comb1 comb2  comb1 comb2)  ==>

--------------------------------------------------------------------------{CMBN 1}
<55 * U>
<110 * V>
<165 * W>
<220 * Z>
------------------------------------------------------------------------------
```

1.3 Representation of a chain complex

A chain complex is implemented as an instance of a `CLOS` class, the class `CHAIN-COMPLEX`, whose definition is

```
(DEFCLASS CHAIN-COMPLEX ()
    ((cmpr  :type cmprf :initarg :cmpr  :reader cmpr1)
     (basis :type basis :initarg :basis :reader basis1)
     ;; BaSe GeNerator
     (bsgn :type gnrt :initarg :bsgn :reader bsgn)
     ;; DiFFeRential
     (dffr :type morphism :initarg :dffr :reader dffr1)
     ;; GRound MoDule
     (grmd :type chain-complex :initarg :grmd :reader grmd)
     ;; EFfective HoMology
     (efhm :type homotopy-equivalence :initarg :efhm :reader efhm)
     ;; IDentification NuMber
     (idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
     ;; ORiGiN
     (orgn :type list :initarg :orgn :reader orgn)))
```

This class has 8 slots:

1. `cmpr`, a comparison function or method for generators with a range in the set `[:less, :equal, :greater]`.

   It is very important to note that the generators to be compared are assumed to be of the *same degree*, i.e. they must belong to the same group C_p. As an exception with the general policy of the software, this degree is not explicitly precised. The implementor has chosen to avoid additional tests because in real problems the program spends a lot of time comparing generators.

2. `basis`, a lisp function giving the distinguished **ordered** basis of the free $\mathbb{Z}$–modules[2] (C_p). When some components of the chain complex are not finitely generated, we say that the chain complex is *locally effective*. In this case the value of this slot must be the keyword `locally-effective`.

3. `bsgn`, a lisp object of any type representing a distinguished generator in dimension 0, the base generator.

4. `dffr`, the differential morphism, instance of the class `MORPHISM`, defined hereafter. The pure lisp function corresponding to the differential

[2] Recall that in the software, only free chain complexes are considered.

homomorphism for each p ($d_p : C_p \to C_{p-1}$) is defined in the instance morphism object and not directly in the instance chain complex object.

5. `idnm`, an integer, number plate for this object. This is generated by the system in a sequential way, each time a new `Kenzo` object is created.

6. `orgn`, a list containing a comment to recall to the user the *origin* of the object. This comment must be chosen with care because, when the user creates a new chain complex instance, the system `Kenzo` uses the comment list information to search in a specific list (here `*chcm-list*`) if the object has not been already built. So, one avoids the duplication of instances of the same object.

The two slots `grmd` and `efhm` will be explained later. The accessors of the slots are the functions whose name appears after the specifier `:reader` in the class definition. A printing method has been associated to the class `CHAIN-COMPLEX` and the external representation of a chain complex instance is a string like `[K`n `Chain-Complex]`, where n is the number plate of this `Kenzo` object.

1.3.1 The function build-chcm

To facilitate the construction of instances of the class `CHAIN-COMPLEX` and to free the user to call the standard constructor `make-instance`, the software provides the function

`build-chcm` `:cmpr` *cmpr* `:basis` *basis* `:bsgn` *bsgn* `:intr-dffr` *intr-dffr*
`:strt` *strt* `:orgn` *orgn*

defined with keyword parameters. The returned value is an instance of the class `CHAIN-COMPLEX`. In particular, this function frees the user to build himself the instance of the class `MORPHISM` corresponding to the differential homomorphism. The keyword arguments of `build-chcm` are:

- *cmpr*, the comparison function for generators.
- *basis*, the function defining the distinguished basis of the free $\mathbb{Z}$–modules C_p or the keyword `:locally-effective`.
- *bsgn*, a generator, the base point of the underlying set.
- *intr-dffr*, a **lisp function** defining the differential homomorphism for each p ($d_p : C_p \to C_{p-1}$).

- *strt*, one of the two values: `:gnrt` or `:cmbn`, defining the mapping *strategy* of the differential homomorphism, either by generator or by combination. The default is `:gnrt`. The real connection between the arguments *intr-dffr* and *strt* will be detailed hereafter through typical examples. The general idea is the following: if the strategy is `:gnrt`, the `:intr-dffr` argument function uses two arguments, namely a degree and a generator of this degree and computes the boundary combination of this generator. If the strategy is `:cmbn`, the `:intr-dffr` argument function uses a combination as argument and computes the boundary combination of this argument. We recall that a combination contains its own degree.

- *orgn*, a list containing a relevant and carefully chosen comment about the *origin* of the chain complex. If, during a Lisp session, the user wishes to modify any slot of an existing chain complex, by calling again `build-chcm`, he must change also the comment, otherwise the new version of the object will not be created. This remark is valid for any kind of instantiation of `Kenzo` objects. For avoiding such a constraint, one may use the function `cat-init`, before the redefinition.

After creation of an instance of chain complex, the function `build-chcm` pushes this object in the list of already created chain complexes `*chcm-list*`.

A first example of chain complex

Let us consider our small example *diabolo*. We shall give the same name to the corresponding chain complex instance. Let us define, one by one, the values of the key parameters, though it is possible to put them directly in the `build-chcm` call. First, the function `s-cmpr`, already seen above, is the natural choice to compare generators, which are here, lisp symbols:

```
(setf diabolo-cmpr #'s-cmpr)
```

The function for the basis consists in enumerating the distinguished basis as lisp lists according to the degree:

```
(setf diabolo-basis #'(lambda (dmn)
                        (case dmn
                          (0 '(s0 s1 s2 s3 s4 s5))
                          (1 '(s01 s02 s12 s23 s34 s35 s45))
                          (2 '(s345))
                          (otherwise nil ))))
```

For the base point, we may choose any vertex:

```
(setf diabolo-bspn 's0)
```

The lisp function for the differential homomorphism, also called boundary homomorphism, computes the boundary of each generator, in this case, according to the classical (simplicial) rule:

$$\mathbf{d}[s_0 s_1 \dots s_n] = \sum_{i=0}^{n} (-1)^i s_0 s_1 \dots \widehat{s_i} \dots s_n.$$

It must be noted that our differential uses a predefined choice for the order of the vertices.

```
(setf diabolo-pure-dffr
     #'(lambda (dmn gnr)
        (unless (<= 0 dmn 2)
          (error "Non-correct dimension for diabolo-dp."))
        (case dmn
           (0 (cmbn -1)) ; Note the null combination of degree -1
           (1 (case gnr
                (s01 (cmbn  0 -1 's0 1 's1))
                (s02 (cmbn  0 -1 's0 1 's2))
                (s12 (cmbn  0 -1 's1 1 's2))
                (s23 (cmbn  0 -1 's2 1 's3))
                (s34 (cmbn  0 -1 's3 1 's4))
                (s35 (cmbn  0 -1 's3 1 's5))
                (s45 (cmbn  0 -1 's4 1 's5))))
            (2 (case gnr
                (s345 (cmbn 1 1 's34 -1 's35 1 's45))))
            (otherwise (error "Bad generator for complex diabolo")))
      ))
```

The strategy is by generator and the comment recalls the name of the problem:

```
(setf diabolo-strt :GNRT)

(setf diabolo-orgn '(diabolo-for-example))
```

The effective call to `build-chcm` is now reduced to:

```
(setf diabolo (build-chcm :cmpr diabolo-cmpr :basis diabolo-basis
                          :bsgn diabolo-bspn :intr-dffr diabolo-pure-dffr
                          :strt diabolo-strt :orgn diabolo-orgn))       ==>

[K1 Chain-Complex]
```

The value of the symbol `diabolo` is the `CHAIN-COMPLEX` instance which is here the first created `Kenzo` object. The string `[K1 Chain-Complex]` is printed by the printing method associated to the class.

1.3.2 Simple functions handling chain complexes

`cat-init` *[Function]*

Clear among others, the list `*chcm-list*`, list of user created chain complexes and reset the global counter to 1. The existing objects and in particular here the chain complexes, are not destroyed but they will not enter any more in account during the search process for duplicated objects. This remark is general for the other types of objects saved in specific lists.

`chcm` *n* *[Function]*

Return from the list `*chcm-list*` the chain complex instance whose the Kenzo identification is *n*; if it does not exist, return `NIL`.

`cmpr` *object item1 item2* *[Macro]*

Apply the comparison function associated to the chain complex *object* to the two generators *item1* and *item2*.

`basis` *object n* `:dgnr` *[Macro]*

With only one argument (*object*), get the function attached to the slot `basis` of the chain complex object. With two arguments, get the distinguished basis of the group of degree *n* in the chain complex *object*. If the chain complex is locally effective, this function returns an error because, in some degrees, the corresponding set of generators is probably infinite. With a third argument, the keyword `:dgnr`, get also the *degenerate* elements of the basis in degree *n*.

`dffr` *chcm* `&rest` *[Macro]*

Versatile macro to apply the differential morphism of the chain complex *chcm* either to a combination or a generator with a degree, respectively `(dffr` *chcm cmbn*`)` or `(dffr` *chcm degr gnrt*`)`. The macro `?`, described later, may be used for the same purpose.

`z-chcm` *[Function]*

Build the unit chain complex (see hereafter).

Examples

Let us apply some accessors functions and the simple functions above to the chain complex `diabolo`. First, we see that the list `*chcm-list*` contains only one element, namely the chain complex just created.

```
*chcm-list*  ==>

([K1 Chain-Complex])
```

```
(chcm 1)  ==>

[K1 Chain-Complex]

(orgn diabolo)  ==>

(DIABOLO-FOR-EXAMPLE)

(idnm diabolo)  ==>

1

(basis diabolo 0)  ==>

(S0 S1 S2 S3 S4 S5)

(basis diabolo 1)  ==>

(S01 S02 S12 S23 S34 S35 S45)

(basis diabolo 2)  ==>

(S345)

(basis diabolo 10)  ==>

NIL

(dffr diabolo 2 's345)  ==>

------------------------------------------------------------------------{CMBN 1}
<1 * S34>
<-1 * S35>
<1 * S45>
-------------------------------------------------------------------------------

(dffr diabolo *)  ==>  ;;; (* means the previous result, a combination)

------------------------------------------------------------------------{CMBN 0}
-------------------------------------------------------------------------------
```

An important trivial case: the unit chain complex, $\mathbb{Z}$

The unit chain complex, has a unique non null component, namely a $\mathbb{Z}$–module of degree 0 generated by a unique generator, called here `:Z-gnrt`. It is defined by the following call to `build-chcm`:

```
(setf ZCC
   (the chain-complex
      (build-chcm
         :cmpr #'(lambda (gnrt1 gnrt2) (the cmpr :equal))
         :basis #'(lambda (n)
                     (the list
                        (if (zerop n) '(:Z-gnrt) +empty-list+)))
         :bsgn :Z-gnrt
         :intr-dffr #'(lambda (cmbn)
                         (the cmbn (zero-cmbn (1- (cmbn-degr cmbn)))))
         :strt :cmbn
         :orgn '(zcc-constant))))
```

In this definition,

1. The `:cmpr` keyword argument is a function returning `:equal` on any pair on generators (because there is a unique generator!).

2. The `:basis` keyword argument is a lisp function returning the null basis (the constant `+empty-list+ = ()`) for $p \neq 0$ and the list `(:Z-gnrt)`, for $p = 0$.

3. The base generator is of course `Z-gnrt`.

4. The `:intr-dffr` keyword argument is a lisp function defining the differential which, to any combination of degre p of the chain complex, returns a *null combination* of degree $p - 1$. This simple lisp function is also provided in `Kenzo` and is called `zero-intr-dffr`.

5. The `:strt` keyword argument is the combination strategy (`:cmbn`).

6. The `:orgn` keyword argument is the comment list `(zcc-constant)`.

In the software `Kenzo`, the chain complex instance `ZCC` may be built, when needed, by the lisp statement: `(z-chcm)`. This statement may be used freely each time one needs this chain complex, since the system recognizes if it has already been created.

The chain complex `circle`

On the model of the previous chain complex, one may define a function `circle` for building a chain complex having the same homology as the circle.

```
(defun CIRCLE ()
   (the chain-complex
      (build-chcm
         :cmpr #'(lambda (gnrt1 gnrt2) (the cmpr :equal))
         :basis #'(lambda (dmns)
                     (the list
                       (case dmns (0 '(*)) (1 '(s1))
                                  (otherwise +empty-list+))))
         :bsgn '*
         :intr-dffr #'zero-intr-dffr
         :strt :cmbn
         :orgn '(circle))))
```

1.4 Morphisms

Algebraic Topology uses morphisms between chain complexes and the differential homomorphism may be considered as a particular case of morphism. A morphism is implemented in the system as an instance of the class `MORPHISM`, whose definition is:

```
(DEFCLASS MORPHISM ()
     ;; SOuRCe
    ((sorc :type chain-complex :initarg :sorc :reader sorc)
     ;; TaRGeT
     (trgt :type chain-complex :initarg :trgt :reader trgt)
     ;; DEGRee
     (degr :type fixnum :initarg :degr :reader degr)
     ;; INTeRnal
     (intr :type intr-mrph :initarg :intr :reader intr)
     ;; STRaTegy
     (strt :type strt :initarg :strt :reader strt)
     ;; CaLl NuMber
     (???-clnm :type fixnum :initform 0 :accessor ???-clnm)
     (?-clnm :type fixnum :initform 0 :accessor ?-clnm)
     ;; ReSuLTS
     (rslts :type simple-vector  :reader rslts)
     ;; IDentification NuMber
     (idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
     ;; ORiGiN
     (orgn :type list :initarg :orgn :reader orgn)))
```

This class has 10 slots:

1. `sorc`, an object of the class `CHAIN-COMPLEX`, namely the *source* chain complex of this morphism.

2. `trgt`, an object of the class `CHAIN-COMPLEX`, namely the *target* chain complex of this morphism.

3. `degr`, an integer, the degree of the morphism. A morphism is supposed to associate to any element of degree k of the source chain complex, an element of degree $k + degr$ of the target chain complex. For instance, the differential homomorphism is of degree -1.

4. `intr`, a **pure lisp function** implementing the mathematical algorithm of the morphism and taking in account the strategy (`strt`).

5. `strt`, one of the two symbols `:gnrt`, `:cmbn`. What has already been said about the strategy of the differential morphism is generalized

to any morphism: according to the value of the argument, `:gnrt` or `:cmbn`, the lisp function attached to the keyword just above, works respectively with 2 arguments (a degree and a generator) or only one (a combination) and must return a combination, image of the generator or the combination argument.

6. `???-clnm`, an integer updated by the system for statistics (number of times the morphism has been called on combinations – Internal use)

7. `?-clnm`, analogous to the previous field, but for generators.

8. `rslts`, an array of dimension `*maxdim*` reserved by the system to save computed results in order to avoid re–computing, for instance, the differential of the same generator. – Internal use.

9. `idnm`, an integer, number plate for this object. This is generated by the system.

10. `orgn`, a relevant comment list.

The accessors of the slots are the functions whose name appears after the specifier `:reader` or after the specifier `:accessor` in the class definition. A printing method has been associated to the class `MORPHISM` and the external representation of an instance is a string like `[K`n `Morphism (degree` d`): K`p `-> K`q`]` or `[K`n `Cohomology-Class (degree` d`)]` when the chain complex target is the unit $\mathbb{Z}$. In this string, n is the number plate of the `Kenzo` object, d is the degree of the morphism, `K`p is the `Kenzo` object source of the morphism and `K`q the target. In all the examples of this manual, the last part of the string will not be necessarily printed.

1.4.1 The function build-mrph

To facilitate the construction of instances of the class `MORPHISM` and to free the user to call the standard constructor `make-instance`, the software provides the function

`build-mrph :sorc` *sorc* `:trgt` *trgt* `:degr` *degr* `:intr` *intr* `:strt` *strt*
`:orgn` *orgn*

defined with keyword parameters. The returned value is an instance of the class `MORPHISM`. The keyword arguments of `build-mrph` are:

- *sorc*, the source object, a `CHAIN-COMPLEX` type object.
- *trgt*, the target object, a `CHAIN-COMPLEX` type object.
- *degr*, the degree of the morphism, an integer.
- *intr*, the pure lisp function defining the effective mapping.
- *strt*, the strategy, i.e. `:gnrt` or `:cmbn`.
- *orgn*, a relevant comment list.

After a call to `build-mrph`, the morphism instance is added to a list of previously constructed ones (`*mrph-list*`).

The differential homomorphism in a chain complex instance

In a chain complex instance, the differential homomorphism is defined as a morphism with identical source and target, and degree -1. The user must know that the function `build-chcm` calls internally the function `build-mrph` and passes it the keyword argument `pure-dffr`. The function `build-mrph` builds the morphism instance which will be then inserted into the slot `:dffr` of the chain complex instance to be constructed.

Examples

Let us define respectively a *zero–morphism* of degree -1 and an *identity–morphism* (degree 0) between the *unit chain complex* `ZCC` and itself.

```
(setf ZCC (z-chcm))  ==>

[K1 Chain-Complex]

(setf zero-morphism (build-mrph :sorc ZCC
                                :trgt ZCC
                                :degr -1
                                :intr #'(lambda(comb)
                                          (cmbn (1- (degr comb))))
                                :strt :cmbn
                                :orgn '(zero morphism on ZCC) ))

[K3 Cohomology-Class (degree 1)]
```

```
(setf id-morphism   (build-mrph :sorc ZCC
                                :trgt ZCC
                                :degr 0
                                :intr #'identity
                                :strt :cmbn
                                :orgn '(identity morphism on ZCC) ))

[K4 Cohomology-Class (degree 0)]
```

On the first morphism, we see that the `:intr` keyword argument is a lisp function taking any combination of degree p of the unit chain complex and generating a null combination of degree $p-1$ of the same chain complex. The second morphism uses the lisp function `identity`.

1.4.2 Applying morphisms

To apply an already constructed morphism on a generator, one uses the function `gnrt-?`, the usage of which is described hereafter. On a similar way, to apply a morphism on a combination, one uses the function `cmbn-?`. It is very important to note that these functions can be used with the underlying morphism, **whatever strategy had been decided at creation time by the user for the morphism mapping**. In other words, a morphism defined with the strategy `:gnrt` (resp. `:cmbn`) may be applied to a combination (resp. generator). These functions are mainly used inside the software. For a practical usage, one may use the versatile macro `?`.

`gnrt-?` *mrph degr gnrt* *[Function]*
: Apply the morphism *mrph* on the generator *gnrt* of degree *degr*.

`cmbn-?` *mrph cmbn* *[Function]*
: Apply the morphism *mrph* on the combination *cmbn*.

`?` `&rest` *args* *[Macro]*
: Versatile macro for applying a morphism in both cases above, i.e. indifferently as (`?` *mrph degr gnrt*) or (`?` *mrph cmbn*). If the first argument is a chain complex object, as in (`?` *chcm degr gnrt*) or in (`?` *chcm cmbn*), then the differential morphism of the chain complex *chcm* is applied to the arguments.

1.4.3 Functions defining morphisms

The following functions are useful to work on morphisms, particularly to define new morphisms from already defined ones.

`cat-init` *[Function]*
Clear in particular `*mrph-list*`, the list of user created morphisms and reset the global counter to 1.

`mrph` *n* *[Function]*
Retrieve in the list `*mrph-list*` the morphism instance whose identification is *n*. If it does not exist, return `NIL`.

`zero-mrph` *chcm1* `&optional` *(chcm2 chcm1) (degr 0)* *[Function]*
Construct the null morphism between the chain complexes *chcm1* and *chcm2* of degree *degr*. The parameters *chcm2* and *degr* are optional and if omitted the default values (respectively *chcm1* and *0*) are taken.

`idnt-mrph` *chcm* *[Function]*
Construct the identity morphism (degree 0) between the chain complex *chcm* and itself.

`opps` *mrp* *[Function]*
Construct the opposite morphism of *mrph*, i.e. $-1 \times mrp$; such a function, as well as the following ones, installs the right source and target.

`cmps` *mrph1 mrph2* `&optional` *strt* *[Method]*
Construct the composite of the morphisms, i.e. $mrph_1 \circ mrph_2$. Of course, the target of $mrph_2$ must be the same as the source of $mrph_1$, otherwise the system signals an error. The new instance inherits its source slot from $mrph_2$ and its target slot from $mrph_1$. This function optimizes the compositions in which appear zero morphisms or identity morphisms. Unless the user gives explicitly the strategy (*strt*), the resulting strategy is determined by the respective strategy of the two morphisms.

`cmps` *chcm1 chcm2* `&optional` *strt* *[Method]*
Construct the composite of the differential of the chain complexes *chcm1* and *chcm2*, i.e. $d_1 \circ d_2$.

`cmps` *chcm1 mrph2* `&optional` *strt* *[Method]*
Construct the composite of the differential of the chain complex *chcm1* and the morphism *mrph2*, i.e. $d_1 \circ mrph_2$.

`cmps` *mrph1 chcm2* `&optional` *strt* *[Method]*

Construct the composite of the morphism *mrph1* and the differential of the chain complex *chcm2*, i.e. $mrph_1 \circ d_2$.

`i-cmps` *mrph1 mrph2 ... mrphk* *[Macro]*

Construct the composite of the morphisms, i.e. $mrph_1 \circ mrph2 \circ \cdots \circ mrph_k$. Of course, the target of $mrph_i$ must be the same as the source of $mrph_{i-1}$, otherwise the system signals an error. The new instance inherits its source slot from $mrph_k$ and its target slot from $mrph_1$. This function optimizes the compositions in which appear zero morphisms or identity morphisms.

`add` *mrph1 mrph2* `&optional` *strt* *[Method]*

Construct a morphism, sum of the morphisms $mrph_1$ and $mrph_2$. The result of the mapping of the morphism sum is the sum of the results of the mappings of the morphisms. The respective definitions of the morphisms $mrph_1$ and $mrph_2$ must be coherent, in particular they must have the same source, target and degree. The user may impose its strategy, otherwise it is defined in the program according to the respective strategy of the arguments.

`i-add` *mrph1 mrph2 ... mrphk* *[Macro]*

Construct a morphism, sum of the morphism $mrph_1$, $mrph_2$, ..., $mrph_k$. The result of the mapping of the morphism sum is the sum of the results of the mappings of the $mrph_i$. The respective definitions of $mrph_i$ must be coherent, in particular they must have the same source, target and degree. The macro `i-add` has an undefinite number of arguments. With one argument, the macro returns that argument.

`sbtr` *mrph1 mrph2* `&optional` *strt* *[Method]*

Construct a morphism, difference of the morphisms $mrph_1$ and $mrph_2$. The conditions of validity are similar to those of the method `add`.

`i-sbtr` *mrph1 mrph2 ... mrphk* *[Macro]*

Construct a morphism, difference of the morphisms $mrph_1$, $mrph_2$, ..., $mrph_k$, in the sense $mrph_1 - mrph_2 - \cdots - mrph_k$. The conditions of validity are similar to those of the method `add` for the morphisms. The macro `i-sbtr` must have at least 2 arguments.

`change-sorc-trgt` *mrph* `&key` *sorc trgt* *[Function]*

Build from the morphism *mrph* a new morphism inheriting from *mrph* the slots `:degr` (degree), `:intr` (mapping) and `:strt` (strategy). The source and target slots of this new morphism are given by the key parameters *sorc* and *trgt*. If any key parameter is omitted, the corresponding slot is inherited from *mrph* (default value).

`dstr-change-sorc-trgt` *mrph* `&key` *sorc trgt* *[Function]*

Modify **destructively** the morphism *mrph*. The source and target slots of the first argument are replaced respectively by the key parameters *sorc* and *trgt*.

`add` *chcm perturbation* `&optional` *strt* *[Method]*

Create from the chain complex *chcm* a new chain complex inheriting from *chcm* the slots `cmpr` and `basis`. The boundary morphism attached to this new chain complex is the sum of the boundary morphism d of *chcm* (slot `dffr`) and a perturbation morphism δ represented by the `MORPHISM` instance *perturbation*. Of course, the new boundary operator must verify $(d+\delta)\circ(d+\delta)=0$. The user will note that this method does not create a morphism but a chain complex.

Examples

In the following examples, we first construct a chain complex instance `ccn` where the groups C_p are freely generated by numerical basis taken formally. These basis are sets of 10 numbers or *decades* produced by the fonction `<a-b<`. For instance, in dimension 0, the basis is $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, in dimension 1, $\{10, 11, \ldots, 19\}$ and so on. The differential is defined as follows: for an even dimension p of the group, a generator k is sent to the combination $cmbn(p-1, 1, k-10)$ if k is even and to the null combination of degree $p-1$ if k is odd. The reverse action is taken if p is odd. So, from the programming point of view, it is sufficient to test the parity of $p+k$. Then, we construct two simple morphisms `upper-shift` and `lower-shift` which respectively apply bijectively a decade on the following one and on the previous one. The generators being integers, the comparative function is of course `f-cmpr`.

```
(setf ccn-boundary #'(lambda (dgr gnr)
     (if (evenp (+ dgr gnr))
         (cmbn (1- dgr) 1 (- gnr 10))
         (cmbn (1- dgr)))))

(setf ccn (build-chcm :cmpr #'f-cmpr
                      :basis #'(lambda (n) (<a-b< (* 10 n) (* 10 (1+ n))))
                      :bsgn 0
                      :intr-dffr  ccn-boundary
                      :strt :gnrt
                      :orgn '(ccn) ))  ==>

[K3 Chain-Complex]

(setf upper-shift (build-mrph
                      :sorc ccn :trgt ccn :strt :gnrt :degr +1
                      :intr #'(lambda(d gn) (cmbn (1+ d) 1 (+ gn 10)))
                      :orgn '(ccn shift +10) ))  ==>

[K5 Morphism (degree 1)]

(setf lower-shift (build-mrph
                      :sorc ccn :trgt ccn :strt :gnrt :degr -1
                      :intr #'(lambda(d gn) (cmbn (1- d) 1 (- gn 10)))
                      :orgn '(ccn shift -10) ))  ==>

[K6 Morphism (degree -1)]
```

First, let us test the differential (in particular $d \circ d = 0$).

```
(? ccn 2 22)     ==>
```

```
-------------------------------------------------------------------------{CMB 1}
<1 * 12>
-------------------------------------------------------------------------------

(? ccn *)   ==>     ; (* means the result of the previous command)

-------------------------------------------------------------------------{CMB 0}
-------------------------------------------------------------------------------

(setf combn (cmbn 5 1 50 5 55 9 59))    ==>

-------------------------------------------------------------------------{CMB 5}
<1 * 50>
<5 * 55>
<9 * 59>
-------------------------------------------------------------------------------

(? ccn combn)   ==>

-------------------------------------------------------------------------{CMB 4}
<5 * 45>
<9 * 49>
-------------------------------------------------------------------------------

(? ccn *)  ==>

-------------------------------------------------------------------------{CMB 3}
-------------------------------------------------------------------------------
```

Then, let us test the morphisms on generators and combinations.

```
(? upper-shift 0 6)   ==>

-------------------------------------------------------------------------{CMB 1}
<1 * 16>
-------------------------------------------------------------------------------

(? lower-shift 5 51)  ==>

-------------------------------------------------------------------------{CMB 4}
<1 * 41>
-------------------------------------------------------------------------------
```

We may iterate the mapping upon the previous result (symbol *). Note that now, though `lower-shift` has been constructed with the strategy `:gnrt`, its works also on a combination.

```
(? lower-shift *)   ==>
```

```
-------------------------------------------------------------------------{CMB 3}
<1 * 31>
-------------------------------------------------------------------------------
```

Let us construct new morphisms from `upper-shift` and `lower-shift`. The tests are made upon the degree 1 combination $1*10+2*11+3*12+4*13$.

```
(setf comb1 (cmbn 1 1 10 2 11 3 12 4 13))  ==>

-------------------------------------------------------------------------{CMB 1}
<1 * 10>
<2 * 11>
<3 * 12>
<4 * 13>
-------------------------------------------------------------------------------
```

The composition of the two morphisms must be an identity operation. The degree of `identity?` is in the `degr` slot of the morphism object instance and may be read by the function `degr`:

```
(setf identity? (cmps upper-shift lower-shift))   ==>

[K7 Morphism (degree 0)]

(degr identity?)    ==>

0
```

We see now that `identity?` applied on `comb1` returns a combination mathematically equal to `comb1`. No simple lisp comparison can prove this, nevertheless, their mathematical difference is the null combination, as shown by a call to the function `2cmbn-sbtr` applied to two combinations. Note that the function `2cmbn-sbtr` needs the comparison function of the chain complex `ccn`.

```
(? identity? comb1)    ==>

-------------------------------------------------------------------------{CMB 1}
<1 * 10>
<2 * 11>
<3 * 12>
<4 * 13>
-------------------------------------------------------------------------------

(2cmbr-sbtr (cmpr ccn) comb1 *)  ==>
-------------------------------------------------------------------------{CMB 1}
-------------------------------------------------------------------------------
```

We may compose `upper-shift` with itself:

```
(setf upper2-shift (cmps upper-shift upper-shift))  ==>

[K8 Morphism (degree 2)]

(degr upper2-shift)   ==>

2

(? upper2-shift comb1)   ==>

------------------------------------------------------------------------{CMB 3}
<1 * 30>
<2 * 31>
<3 * 32>
<4 * 33>
------------------------------------------------------------------------
```

Adding `upper-shift` with itself gives a very different result:

```
(setf twice-up-shift (add upper-shift upper-shift))  ==>

[K9 Morphism (degree 1)]

(degr twice-up-shift)  ==>

1

(? twice-up-shift comb1)   ==>

------------------------------------------------------------------------{CMB 2}
<2 * 20>
<4 * 21>
<6 * 22>
<8 * 23>
------------------------------------------------------------------------
```

Let us compose `upper-shift` and the differential in both ways. Recall that the differential is a morphism structure, and may be obtained from the `dffr` slot of the chain complex `ccn` by the reader accessor function `dffr1`. This morphism has been built by `build-chcm` from the lisp function `ccn-boundary`. One can see that the operators do not commute.

```
(setf up-d (cmps upper-shift (dffr1 ccn)))  ==>

[K10 Morphism (degree 1)]

(setf d-up (cmps (dffr1 ccn) upper-shift))  ==>

[K11 Morphism (degree 1)]
```

```
(? up-d 1 11)   ==>

-------------------------------------------------------------------------{CMB 1}
<1 * 11>
-------------------------------------------------------------------------------

(? d-up 1 11)   ==>

-------------------------------------------------------------------------{CMB 1}
-------------------------------------------------------------------------------

(setf comb3 (cmbn 1 1 10 2 11 3 12 4 13 5 14 6 15))   ==>

-------------------------------------------------------------------------{CMB 1}
<1 * 10>
<2 * 11>
<3 * 12>
<4 * 13>
<5 * 14>
<6 * 15>
-------------------------------------------------------------------------------

(? up-d comb3)   ==>

-------------------------------------------------------------------------{CMB 1}
<2 * 11>
<4 * 13>
<6 * 15>
-------------------------------------------------------------------------------

(? d-up comb3)   ==>

-------------------------------------------------------------------------{CMB 1}
<1 * 10>
<3 * 12>
<5 * 14>
-------------------------------------------------------------------------------
```

1.4.4 Accessing Kenzo objects

Up to now, we have seen two kinds of Kenzo objects stored in specific lists and retrievable by a number, namely the chain complexes and the morphisms. The retrieval functions are specific to the object: chcm for a chain complex, mrph for a morphism. The same scheme will be applied for others kinds of objects: reductions, homotopy equivalences, coalgebras, algebras, simplicial morphism, etc, each having its specific list. But, in fact the numbering is independent of the type of the object and is incremented each time an object is created. Three general functions are at the disposal of the user to get information about the n–th Kenzo object: k, kd and kd2. They may be useful for debugging purpose.

k n *[Function]*

Get the n–th Kenzo object.

kd n *[Function]*

Give the type of the Kenzo objet number n and print the comment list (slot :orgn) of the object.

kd2 n *[Function]*

Give the type of the Kenzo objet number n, print the comment list (slot :orgn) of the object and recursively, give the same kind of informations about all the Kenzo objects of the same type in relation with this n–th object. Return the list of numbers of all those objects. See in the following example, the case of composition of morphisms.

Examples

```
(k 1)  ==>

[K1 Chain-Complex]

(kd 1)  ==>

Object: [K1 Chain-Complex]
   Origin: (Z-CHCM)

(k 3)   ==>

[K3 Chain-Complex]

(kd 3)  ==>
```

```
Object: [K3 Chain-Complex]
   Origin: (CIRCLE)

(kd 8)  ==>

Object: [K8 Morphism (degree -1): K5 -> K5]
   Origin: (CCN SHIFT -10)

(kd 5)  ==>

Object: [K5 Chain-Complex]
   Origin: (CCN)

(kd 9)  ==>

Object: [K9 Morphism (degree 0): K5 -> K5]
   Origin: (2MRPH-CMPS [K7 Morphism (degree 1): K5 -> K5]
                       [K8 Morphism (degree -1): K5 -> K5] GNRT)

(kd2 9)  ==>

Object: [K9 Morphism (degree 0): K5 -> K5]
   Origin: (2MRPH-CMPS [K7 Morphism (degree 1): K5 -> K5]
                       [K8 Morphism (degree -1): K5 -> K5] GNRT)


Object: [K8 Morphism (degree -1): K5 -> K5]
   Origin: (CCN SHIFT -10)


Object: [K7 Morphism (degree 1): K5 -> K5]
   Origin: (CCN SHIFT 10)

(9 8 7)
```

Lisp files concerned in this chapter

`combinations.lisp`, `chain-complexes.lisp`, `chcm-elementary-op.lisp`. [`classes.lisp` , `macros.lisp`, `various.lisp`].

Chapter 2

Objects with effective homology

2.1 Introduction

The present chapter describes the set of programming tools for handling *objects with effective homology*. The theoretical material may be found in the paper *Constructive Algebraic Topology* by Julio Rubio Garcia and Francis Sergeraert[1]. The terminology used in this chapter is compatible with this reference.

2.2 Reduction

A *reduction* is a 5–tuple $(\hat{C}, C, f, g, h)$:

$$\begin{array}{c} \hat{C} \xrightarrow{h} {}^{s}\hat{C} \\ f \downarrow\uparrow g \\ C \end{array}$$

where $\hat{C}$ and C are chain complexes, f and g chain complex morphisms and h a homotopy operator. Hereafter, $\hat{C}$ is called the *top chain complex* and C the *bottom chain complex*. ${}^{s}\hat{C}$ is $\hat{C}$ shifted, i.e. h has degree 1. The mappings f, g, h, together with the differential operator d on $\hat{C}$, must verify the following relations:

[1] Available at the web site **http://www-fourier.ujf.grenoble.fr/~ sergerar/**

$$\begin{aligned} f \circ g &= 1_C \\ h \circ d + d \circ h &= 1_{\hat{C}} - g \circ f \\ f \circ h &= 0 \\ h \circ g &= 0 \\ h \circ h &= 0 \end{aligned}$$

The morphisms f and g and the homotopy operator h describe the (big) chain complex $\hat{C}$ as the direct sum

$$\hat{C} = \hat{C}_1 \oplus \hat{C}_2$$

where $\hat{C}_1 = Im(g) \simeq C$ and $\hat{C}_2 = Ker(f)$ ($\hat{C}_2$ is acyclic).

2.2.1 Representation of a reduction

A reduction is implemented as an instance of the `CLOS` class `REDUCTION`, whose definition is:

```
(DEFCLASS REDUCTION ()
     ;; Top Chain Complex
    ((tcc :type chain-complex :initarg :tcc :reader tcc1)
     ;; Bottom Chain Complex
     (bcc :type chain-complex :initarg :bcc :reader bcc1)
     (f :type morphism :initarg :f :reader f1)
     (g :type morphism :initarg :g :reader g1)
     (h :type morphism :initarg :h :reader h1)
     ;; IDentification NuMber
     (idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
     ;; ORiGiN
     (orgn :type list :initarg :orgn :reader orgn)))
```

This class has 7 slots:

1. `tcc`, the object of type `chain-complex` representing the chain complex $\hat{C}$ (top chain complex).

2. `bcc`, the object of type `chain-complex` representing the chain complex C (bottom chain complex).

3. `f`, the object of type `morphism` representing the morphism f.

4. `g`, the object of type `morphism` representing the morphism g.

5. `h`, the object of type `morphism` representing the morphism h.
6. `idmn`, an integer, number plate for the object.
7. `orgn`, a comment list carefully chosen.

The accessors of the slots are the functions whose name appears after the specifier `:reader` in the class definition. A printing method has been associated to the class `REDUCTION` and the external representation of an instance is a string like `[K`n `Reduction]`, where n is the number plate of the Kenzo object.

2.2.2 The function build-rdct

To facilitate the construction of instances of the `REDUCTION` class, the software provides the function `build-rdct`.

`build-rdct :f` f `:g` g `:h` h `:orgn` *orgn*

defined with keyword parameters. The returned value is an instance of the class `REDUCTION`. The keyword arguments are:

- f, the object of type `morphism` representing the morphism f.
- g, the object of type `morphism` representing the morphism g.
- h, the object of type `morphism` representing the morphism h.
- *orgn*, the comment list carefully chosen since the system does not build a new instance of the class if it finds in the list of already built reductions, `*rdct-list*`, a reduction with the same comment list.

The `tcc` slot and the `bcc` slot of the instance are taken respectively from the `sorc` slot (source slot) and the `trgt` slot (target slot) of the morphism f. The function `build-rdc` controls the validity of the degrees of the morphisms and pushes the new created instance on the list `*rdct-list*`.

2.2.3 Useful macros and functions

`cat-init` *[Function]*

Clear among others, the list `*rdct-list*`, list of user created reductions and reset the global counter to 1. See also the description of this function in chapter 1.

`rdct` *n* *[Function]*

Retrieve in the list `*rdct-list*` the reduction instance whose identification (as `Kenzo` object) is *n*. If it does not exist, return `NIL`.

`bcc` *rdct* `&rest` *args* *[Macro]*

With only one argument (a reduction *rdct*) this macro selects the bottom chain complex of the reduction. Otherwise, it applies the differential of the bottom chain complex of the reduction *rdc* on the arguments *args*, (either *degree generator* or *cmb*).

`tcc` *rdct* `&rest` *args* *[Macro]*

With only one argument (a reduction *rdct*) this macro selects the top chain complex of the reduction. Otherwise, it applies the differential of the top chain complex of the reduction *rdc* on the arguments *args*, (either *degree generator* or *cmb*).

`f` *rdct* `&rest` *args* *[Macro]*

With only one argument (a reduction *rdct*) this macro selects the morphism *f* of the reduction. Otherwise, it applies the morphism *f* of the reduction *rdc* on the arguments *args*, (either *degree generator* or *cmb*).

`g` *rdct* `&rest` *args* *[Macro]*

With only one argument (a reduction *rdct*) this macro selects the morphism *g* of the reduction. Otherwise, it applies the morphism *g* of the reduction *rdc* on the arguments *args*, (either *degree generator* or *cmb*).

`h` *rdct* `&rest` *args* *[Macro]*

With only one argument (a reduction *rdct*) this macro selects the morphism *h* of the reduction. Otherwise, it applies the morphism *h* of the reduction *rdc* on the arguments *args*, (either *degree generator* or *cmb*).

`trivial-rdct` *chcm* *[Function]*

Build the trivial reduction involving only the chain complex *chcm*, as shown in the following diagram,

$$\begin{array}{ccc} C & \stackrel{Zero}{\longrightarrow} & {}^{s}C \\ {\scriptstyle Id}\downarrow\uparrow{\scriptstyle Id} & & \\ C & & \end{array}$$

where the morphism *Zero* is the zero morphism of degree 1 in the chain complex C and *Id* is the identity morphism in that chain complex (see the functions `zero-mrph` and `idnt-mrph` in the chain complex chapter).

`cmps` *brdct trdct* *[Method]*

Build a new reduction from the two reductions $r1$ and $r2$ (here, respectively *brdct* and *trdct*). This is done by a call to `build-rdct` with the following parameters:

$$\begin{aligned} f &= f_{r1} \circ f_{r2}, \\ g &= g_{r2} \circ g_{r1}, \\ h &= h_{r2} + g_{r2} \circ h_{r1} \circ f_{r2}. \end{aligned}$$

The compositions and additions of morphisms are realized respectively by the methods `cmps`, `i-cmps` and `add` (see chapter 1). We recall that the `tcc` and the `bcc` slots of this new created reduction are respectively the source and target slots of the new morphim f.

2.2.4 Verification functions

The two following functions are very helpful to verify the coherence of the the various mappings involved in a reduction. Let us recall the diagram of a reduction:

$$\begin{array}{ccc} \hat{C} & \stackrel{h}{\longrightarrow} & {}^{s}\hat{C} \\ {\scriptstyle f}\downarrow\uparrow{\scriptstyle g} & & \\ C & & \end{array}$$

`pre-check-rdct` *rdct* *[Function]*

Assign to the following lisp global variables, the morphism instances computed from the morphisms of the reduction *rdct*, according to the formulas:

`*tdd*` $= d_{\hat{C}} \circ d_{\hat{C}}$,
`*bdd*` $= d_C \circ d_C$,
`*id-fg*` $= f \circ g - Id_C$,
`*id-gf-dh-hd*` $= Id_{\hat{C}} - g \circ f - (d_{\hat{C}} \circ h + h \circ d_{\hat{C}})$,
`*hh*` $= h \circ h$,
`*fh*` $= f \circ h$,
`*hg*` $= h \circ g$,
`*df-fd*` $= d_C \circ f - f \circ d_{\hat{C}}$,
`*dg-gd*` $= d_{\hat{C}} \circ g - g \circ d_C$.

In these formulas d_C and $d_{\hat{C}}$ are the respective boundary operators of the chain complexes C and $\hat{C}$. Id_C and $Id_{\hat{C}}$ are the identity morphisms on the chain complexes C and $\hat{C}$ built by the function `idnt-mrph`

`check-rdct` *[Function]*

Map all the morphisms prepared by the function `pre-check-rdct` on chosen combinations. This function having no parameters, the user must assign to the two lisp global variables: `*tc*` and `*bc*`, valid combinations belonging respectively to $\hat{C}$ and C. The call to this function is simply `(check-rdct)`. This function knows what morphism to apply to the combinations and pauses after each evaluation to allow the user to inspect each result. To resume the execution, the user must enter a blank character. If the morphisms are coherent, the result of each mapping is a null combination.

Example

To show an example about the verification functions, we first define a locally effective version of the standard simplex Δ^n. The following function `cdelta` builds the standard simplex in dimension *dmns*. In using the created chain complex, the user must bear in mind that the only valid vertices are the vertices numbered from (0) to (*dmns*).

```
(defun cdelta (dmns)
    (build-chcm
       :cmpr #'l-cmpr
       :basis :locally-effective
       :bsgn '(0)
       :intr-dffr
             #'(lambda (degr gmsm)
                  (make-cmbn
                    :degr (1- degr)
                    :list (do ((rslt +empty-list+
                                (cons (cons sign
                                            (append
                                               (subseq gmsm 0 nark)
                                               (subseq gmsm (1+ nark))))
                                      rslt))
                               (sign 1 (- sign))
                               (nark 0 (1+ nark)))
                              ((> nark degr) rslt))))
       :strt :gnrt
       :orgn `(locally effective version of C_* delta ,dmns)))
```

Now, let us define 3 functions `make-f`, `make-g` and `make-h` which build the following respective morphisms between Δ^m and Δ^n:

- `(make-f` *tdms bdms*`)` builds a projection morphism f from Δ^{tdms} to Δ^{bdms}, $tdms >= bdms$, where the vertices (0) to (dms) of Δ^{tdms} are applied on the vertices of the same number in Δ^{bdms} and the vertices ($bdms + 1$) to ($tdms$) are applied on the vertex ($bdms$) of Δ^{bdms}.

- `(make-g` *tdms bdms*`)` builds the injection morphism g from Δ^{bdms} to Δ^{tdms} (the slot `:intr` is the identity function).

- `(make-h` *tdms bdms*`)` builds a homotopy morphism of degree 1, h, from Δ^{tdms} to itself, connecting w.r.t. the homotopy relation, the chain map $g \circ f$ with the identity morphism Id.

```
(defun make-f (tdmns bdmns)
  (build-mrph
    :sorc (cdelta tdmns) :trgt (cdelta bdmns) :degr 0
    :intr #'(lambda (degr gmsm)
              (let ((pos (position-if #'(lambda (vertex) (>= vertex bdmns))
                                      gmsm)))
                   (if pos
                     (if (< pos degr)
                         (zero-cmbn degr)
                         (cmbn degr 1 (nconc (butlast gmsm) (list bdmns))))
                         (cmbn degr 1 gmsm))))
    :strt :gnrt
    :orgn `(projection delta ,tdmns => delta ,bdmns)))

(defun make-g (tdmns bdmns)
  (build-mrph
    :sorc (cdelta bdmns) :trgt (cdelta tdmns) :degr 0
    :intr #'identity
    :strt :cmbn
    :orgn `(injection delta ,bdmns => delta ,tdmns)))

(defun make-h (tdmns bdmns)
  (build-mrph
    :sorc (cdelta tdmns) :trgt (cdelta tdmns) :degr +1
    :intr #'(lambda (degr gmsm)
             (let ((pos (position-if #'(lambda (vertex) (>= vertex bdmns))
                                     gmsm)))
                  (if pos
                      (if (member bdmns gmsm)
                          (zero-cmbn (1+ degr))
                          (cmbn (1+ degr) (-1-expt-n pos)
                           (append (subseq gmsm 0 pos) (list bdmns)
                                   (subseq gmsm pos))))
                      (zero-cmbn (1+ degr)))))
    :strt :gnrt
    :orgn `(homotopy for delta ,tdmns => ,bdmns)))
```

We may now define a function to build a reduction. One has not to define a priori the standard simplices: this is done in all the functions and we know that there is no duplication, because before creation, the system checks, owing to the comment list, if the instance of a class (here a chain complex) already exists.

```
(defun make-rdct (tdmns bdmns)
      (setf rdct (build-rdct
                  :f (make-f tdmns bdmns)
                  :g (make-g tdmns bdmns)
```

```
                    :h (make-h tdmns bdmns)
                    :orgn `(reduction delta ,tdmns ,bdmns))))
```

We now build with a simple call to the function `make-rdct` the chain complexes corresponding to Δ^6 and Δ^3, the 3 morphisms f, g, h and the reduction. The lisp utility function `inspect` gives an idea of the organisation of the class instance. Then we verify the coherency of the reduction with the functions `pre-check-rdct` and `check-rdct`.

```
(setf rdct (make-rdct 6 3))  ==>

[K8 Reduction]

(inspect rdct)  ==>

REDUCTION @ #x498342 = [K8 Reduction]
   0 Class --------> #<STANDARD-CLASS REDUCTION>
   1 ORGN ---------> (REDUCTION ...), a proper list with 4 elements
   2 IDNM ---------> fixnum 8 [#x00000020]
   3 H ------------> [K7 Morphism (degree 1)]
   4 G ------------> [K6 Morphism (degree 0)]
   5 F ------------> [K5 Morphism (degree 0)]
   6 BCC ----------> [K3 Chain Complex]
   7 TCC ----------> [K1 Chain Complex]

(orgn rdct)  ==>

(REDUCTION DELTA 6 3)

(pre-check-rdct rdct)  ==>

---done---

(setf *tc* (cmbn 2 1 '(0 1 2) 10 '(1 2 3) 100 '(1 2 4) 1000 '(2 3 4)))  ==>

------------------------------------------------------------------------{CMBN 2}
<1 * (0 1 2)>
<10 * (1 2 3)>
<100 * (1 2 4)>
<1000 * (2 3 4)>
------------------------------------------------------------------------

(setf *bc* (cmbn 3 4 '(0 1 2 3)))  ==>

------------------------------------------------------------------------{CMBN 3}
<4 * (0 1 2 3)>
------------------------------------------------------------------------
```

```
(check-rdct)  ==>

*TC* =>
-------------------------------------------------------------------------{CMBN 2}
<1 * (0 1 2)>
<10 * (1 2 3)>
<100 * (1 2 4)>
<1000 * (2 3 4)>
------------------------------------------------------------------------------

*BC* =>
-------------------------------------------------------------------------{CMBN 3}
<4 * (0 1 2 3)>
------------------------------------------------------------------------------

Checking *TDD* = 0
Result:
-------------------------------------------------------------------------{CMBN 0}
------------------------------------------------------------------------------

Checking *BDD* = 0
Result:
-------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

Checking *DF-FD* = 0
Result:
-------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

Checking *DG-GD* = 0
Result:
-------------------------------------------------------------------------{CMBN 2}
------------------------------------------------------------------------------

Checking *ID-FG* = 0
Result:
-------------------------------------------------------------------------{CMBN 3}
------------------------------------------------------------------------------

Checking *ID-GF-DH-HD* = 0
Result:
-------------------------------------------------------------------------{CMBN 2}
------------------------------------------------------------------------------
```

```
Checking *HH* = 0
Result:
------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

Checking *FH* = 0
Result:
------------------------------------------------------------------------{CMBN 3}
-------------------------------------------------------------------------------

Checking *HG* = 0
Result:
------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

---done---
```

We do now the same thing with a more complicated reduction which is the composition of the two following reductions:

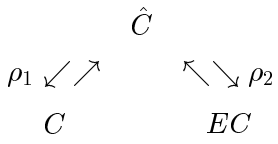
```
(setf trdct (make-rdct 6 4))  ==>

[K40 Reduction]

(setf brdct (make-rdct 4 3))  ==>

[K44 Reduction]

(setf rdct (cmps brdct trdct))  ==>

[K50 Reduction]

(pre-check-rdct rdct)  ==>

---done---

(check-rdct)  ==>

*TC* =>
------------------------------------------------------------------------{CMBN 2}
<1 * (0 1 2)>
<10 * (1 2 3)>
<100 * (1 2 4)>
<1000 * (2 3 4)>
-------------------------------------------------------------------------------
```

```
*BC* =>
------------------------------------------------------------------------{CMBN 3}
<4 * (0 1 2 3)>
------------------------------------------------------------------------------

Checking *TDD* = 0
Result:
------------------------------------------------------------------------{CMBN 0}
------------------------------------------------------------------------------

Checking *BDD* = 0
Result:
------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

Checking *DF-FD* = 0
Result:
------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

Checking *DG-GD* = 0
Result:
------------------------------------------------------------------------{CMBN 2}
------------------------------------------------------------------------------

Checking *ID-FG* = 0
Result:
------------------------------------------------------------------------{CMBN 3}
------------------------------------------------------------------------------

Checking *ID-GF-DH-HD* = 0
Result:
------------------------------------------------------------------------{CMBN 2}
------------------------------------------------------------------------------

Checking *HH* = 0
Result:
------------------------------------------------------------------------{CMBN 4}
------------------------------------------------------------------------------

Checking *FH* = 0
Result:
------------------------------------------------------------------------{CMBN 3}
------------------------------------------------------------------------------
```

```
Checking *HG* = 0
Result:
-------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

---done---
```

2.3 Homotopy equivalence

A *homotopy equivalence* between two chain complexes C and EC is a pair of reductions:



If C and EC are *free* $\mathbb{Z}$-chain complexes, a usual chain equivalence between them can be organized in this way. Frequently the chain complexes C and $\hat{C}$ are *locally effective* and on the contrary, the chain complex EC is *effective*, so that EC can be understood as a description of the homology of C. More precisely, EC is a tool allowing one to compute the homology of C. The chain complexe $\hat{C}$ is only an intermediate object.

2.3.1 Representation of a homotopy equivalence

A homotopy equivalence is implemented as an instance of the `CLOS` class `HOMOTOPY-EQUIVALENCE`, whose definition is

```
(DEFCLASS HOMOTOPY-EQUIVALENCE ()
     ;; Left Bottom Chain Complex
    ((lbcc :type chain-complex :initarg :lbcc :reader lbcc1)
     ;; Top Chain Complex
     (tcc  :type chain-complex :initarg :tcc  :reader tcc1)
     ;; Bottom Right Chain Complex
     (rbcc :type chain-complex :initarg :rbcc :reader rbcc1)
     ;; Left f
     (lf   :type morphism      :initarg :lf   :reader lf1)
     ;; Left g
     (lg   :type morphism      :initarg :lg   :reader lg1)
     ;; Left h
     (lh   :type morphism      :initarg :lh   :reader lh1)
     ;; Right f
     (rf   :type morphism      :initarg :rf   :reader rf1)
     ;; Right g
     (rg   :type morphism      :initarg :rg   :reader rg1)
     ;; Right h
     (rh   :type morphism      :initarg :rh   :reader rh1)
     ;; Left ReDuCTion
     (lrdct :type reduction    :initarg :lrdct :reader lrdct)
```

```
;; Right ReDuCTion
(rrdct :type reduction    :initarg :rrdct :reader rrdct)
;; IDentification NuMber
(idnm :type fixnum :initform (incf *idnm-counter*) :reader idnm)
(orgn :type list          :initarg :orgn :reader orgn)))
```

This class has 13 slots:

1. `lbcc`, the object of type `chain complex` representing the chain complex C (left bottom chain complex on the diagram).
2. `tcc`, the object of type `chain complex` representing the chain complex $\hat{C}$ (top chain complex).
3. `rbcc`, the object of type `chain complex` representing the chain complex EC (right bottom chain complex on the diagram).
4. `lf`, the object of type `morphism` representing the morphism f of the (left) reduction ρ_1.
5. `lg`, the object of type `morphism` representing the morphism g of the (left) reduction ρ_1.
6. `lh`, the object of type `morphism` representing the morphism h of the (left) reduction ρ_1.
7. `rf`, the object of type `morphism` representing the morphism f of the (right) reduction ρ_2.
8. `rg`, the object of type `morphism` representing the morphism g of the (right) reduction ρ_2.
9. `rh`, the object of type `morphism` representing the morphism h of the (right) reduction ρ_2.
10. `lrdc`, the object of type `reduction` representing the (left) reduction ρ_1.
11. `rrdc`, the object of type `reduction` representing the (right) reduction ρ_2.
12. `idn`, an integer, number plate for the object, set by the system.
13. `orgn`, an adequate comment list.

When an instance is created, the printing method associated to the class `HOMOTOPY EQUIVALENCE` prints a string like `[K`*n* `Homotopy-Equivalence]`, where *n* is the number plate of the Kenzo object.

2.3.2 The function build-hmeq

To facilitate the construction of instances of the `HOMOTOPY-EQUIVALENCE` class, the software provides the function, (in fact a method), `build-hmeq` which may be used in the following ways: either with the two reductions ρ_1 and ρ_2 or explicitly with the morphisms. The selection of the adequate method is done by inspecting the first keyword, which is `:lrdct` in the first case and `:lf` in the second one. In both cases the returned value is an instance of the class `HOMOTOPY-EQUIVALENCE`.

1) `build-hmeq :lrdct` *lrdct* `:rrdct` *rrdc* `:orgn` *orgn* *[Method]*

The keyword arguments are:

- *lrdct*, the object of type `reduction` representing the reduction ρ_1.
- *rrdct*, the object of type `reduction` representing the reduction ρ_2.
- *org*, the comment list.

The function `build-heq-from-rdc` calls internally the standard constructor `make-instance`. All the needed information is contained in the two reductions.

2) `build-hmeq :lf` *lf* `:lg` *lg*`:lh` *lh*`:rf` *rf* `:rg` *rg*`:rh` *rh* `:orgn` *orgn* *[Method]*

The keyword arguments are:

- *lf*, the object of type `morphism` representing the morphism f of the reduction ρ_1.
- *lg*, the object of type `morphism` representing the morphism g of the reduction ρ_1.
- *lh*, the object of type `morphism` representing the morphism h of the reduction ρ_1.
- *rf*, the object of type `morphism` representing the morphism f of the reduction ρ_2.

- *rg*, the object of type `morphism` representing the morphism g of the reduction ρ_2.
- *rh*, the object of type `morphism` representing the morphism h of the reduction ρ_2.
- *org*, the comment list.

All the needed information is contained in the morphism structures. In both cases, the method pushes the created instance onto the list `*hmeq-list*`.

2.3.3 Useful macros and functions

`cat-init` *[Function]*

Clear among others, the list `*hmeq-list*`, list of user created homotopy equivalences and reset the global counter to 1.

`hmeq` *n* *[Function]*

Retrieve in the list `*hmeq-list*` the homotopy equivalence instance whose identification is n. If it does not exist, return `NIL`.

`lbcc` *hmeq* `&rest` *args* *[Macro]*

With only one argument (a homotopy equivalence *hmeq*) this macro selects the left bottom chain complex of the homotopy equivalence. Otherwise, it applies the differential of the left bottom chain complex of *hmeq* on the arguments *args*, (either *degree generator* or *cmb*).

`rbcc` *hmeq* `&rest` *args* *[Macro]*

With only one argument (a homotopy equivalence *hmeq*) this macro selects the right bottom chain complex of the homotopy equivalence. Otherwise, it applies the differential of the right bottom chain complex of *hmeq* on the arguments *args*, (either *degree generator* or *cmb*).

`lf` *hmeq* `&rest` *args* *[Macro]*

With only one argument (a homotopy equivalence *hmeq*) this macro selects the morphism f of the left reduction of *hmeq*. Otherwise, it applies the morphism f of the left reduction of *hmeq* on the arguments *args*, (either *degree generator* or *cmb*).

`lg` *hmeq* `&rest` *args* *[Macro]*

With only one argument (a homotopy equivalence *hmeq*) this macro selects the morphism g of the left reduction of *hmeq*. Otherwise,

it applies the morphism g of the left reduction of *hmeq* on the arguments *args*, (either *degree generator* or *cmb*).

`lh` *hmeq* `&rest` *args* *[Macro]*

With only one argument (a homotopy equivalence *hmeq*) this macro selects the morphism h of the left reduction of *hmeq*. Otherwise, it applies the morphism h of the left reduction of *hmeq* on the arguments *args*, (either *degree generator* or *cmb*).

`rf` *hmeq* `&rest` *args* *[Macro]*

With only one argument (a homotopy equivalence *hmeq*) this macro selects the morphism f of the right reduction of *hmeq*. Otherwise, it applies the morphism f of the right reduction of *hmeq* on the arguments *args*, (either *degree generator* or *cmb*).

`rg` *hmeq* `&rest` *args* *[Macro]*

With only one argument (a homotopy equivalence *hmeq*) this macro selects the morphism g of the right reduction of *hmeq*. Otherwise, it applies the morphism g of the right reduction of *hmeq* on the arguments *args*, (either *degree generator* or *cmb*).

`rh` *hmeq* `&rest` *args* *[Macro]*

With only one argument (a homotopy equivalence *hmeq*) this macro selects the morphism h of the right reduction of *hmeq*. Otherwise, it applies the morphism h of the right reduction of *hmeq* on the arguments *args*, (either *degree generator* or *cmb*).

`trivial-hmeq` *chcm* *[Function]*

Build the trivial homotopy equivalence involving only the chain complex *chcm*. In that case, the 3 chain complexes C, EC and $\hat{C}$ are the chain complex *chcm* itself and the reductions ρ_1 and ρ_2 are, of course, the trivial reductions on *chcm* built by the statement `(trivial-rdct` *chcm*`)`.

2.4 The perturbation lemma machinery

For a good understanding of the lisp functions involved in the machinery in question, we recall the *perturbation lemma*[2]. In the basic (resp. trivial) perturbation lemma, the given perturbation concerns the **top** (resp. **bottom**) chain complex.

[2] **Ronald Brown**. *The twisted Eilenberg–Zilber theorem.* Celebrazioni Arch. Secolo XX Simp. Top., 1967, pp 34-37.

Theorem 1 (Basic perturbation lemma) — *Let $\rho = (\hat{C}, C, f, g, h)$ a reduction and $\hat{\delta}$ a* perturbation *of $d_{\hat{C}}$, that is, an operator defined on $\hat{C}$ of degree -1 satisfying the relation $(d_{\hat{C}} + \hat{\delta}) \circ (d_{\hat{C}} + \hat{\delta}) = 0$. Furthermore, the composite function $h \circ \hat{\delta}$ is assumed* locally nilpotent*, that is, $\forall x \in \hat{C}$, $(h \circ \hat{\delta})^n x = 0$, for n sufficiently large. Then a new reduction $\rho' = (\hat{C}', C', f', g', h')$ can be constructed where:*

1) $\hat{C}'$ is the chain complex obtained from C by replacing the old differential $d_{\hat{C}}$ by $(d_{\hat{C}} + \hat{\delta})$,
2) the new chain complex C' is obtained from the chain complex C only by replacing the old differential d_C by $(d_C + \delta)$, with $\delta = f \circ \hat{\delta} \circ \phi \circ g = f \circ \psi \circ \hat{\delta} \circ g$,
3) $f' = f \circ \psi = f \circ (Id - \hat{\delta} \circ \phi \circ h)$,
4) $g' = \phi \circ g$,
5) $h' = \phi \circ h = h \circ \psi$,

with

$$\phi = \sum_{i=0}^{\infty} (-1)^i (h \circ \hat{\delta})^i$$

and

$$\psi = \sum_{i=0}^{\infty} (-1)^i (\hat{\delta} \circ h)^i = Id - \hat{\delta} \circ \phi \circ h,$$

the convergence being ensured by the locally nilpotency of $h \circ \hat{\delta}$ and $\hat{\delta} \circ h$.

Theorem 2 (Trivial perturbation lemma) — *Let $\rho = (\hat{C}, C, f, g, h)$ a reduction and $\check{\delta}$ a* perturbation *of d_C, that is, an operator defined on C of degree -1 satisfying the relation $(d_C + \check{\delta}) \circ (d_C + \check{\delta}) = 0$. Then a new reduction $\rho' = (\hat{C}', C', f', g', h')$ can be constructed where:*

1) $\hat{C}'$ is the chain complex obtained from C by replacing the old differential $d_{\hat{C}}$ by $(d_{\hat{C}} + g \circ \check{\delta} \circ f)$,
2) the new chain complex C' is obtained from the chain complex C only by replacing the old differential d_C by $(d_C + \check{\delta})$,
3) $f' = f$,
4) $g' = g$,
5) $h' = h$.

2.4.1 Useful functions related to the pertubation lemma

The functions implementing the perturbation-lemma are actually the heart of the program. They allow us to implement the effective versions of the classical spectral sequences (Serre, Eilenberg-Moore, ...).

`add` *rdct perturbation* *[Method]*
: Build a new reduction from the object of type `reduction`, *rdct* and the object of type `morphism`, *perturbation*, applying the formulas given in the perturbation lemma. In fact, this method calls one or the other of the two following functions according as the perturbation is on the top or on the bottom chain complex and returns the values of the called function.

`basic-perturbation-lemma` *reduction top-perturbation* *[Function]*
: Return a double value: the perturbed reduction and the computed bottom-perturbation (a morphism) $f \circ \hat{\delta} \circ \phi \circ g$. Of course, to avoid unuseful computations when the morphisms are effectively applied, the function takes in account the simplifications involved for instance when some morphisms are the null morphism.

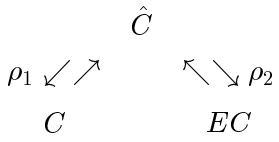
`easy-perturbation-lemma` *reduction bottom-perturbation* *[Function]*
: Return a double value: the perturbed reduction and the computed top-perturbation (a morphism) $g \circ \check{\delta} \circ f$. Of course, to avoid unuseful computation when the morphisms will be applied, the function takes in account the simplifications involved for instance when some morphisms are the null morphism.

`special-bpl` *reduction top-perturbation* *[Function]*
: This function is analogous to `basic-perturbation-lemma` and is used in some special cases when the morphism g is invariant.

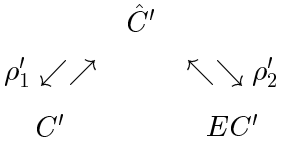
`bpl-*-sigma` *homotopy perturbation* *[Function]*
: Construct the principal series $\phi = \sum_{i=0}^{\infty} (-1)^i (h \circ \hat{\delta})^i$ of the basic perturbation lemma.

`add` *hmeq lb-perturbation* *[Method]*
: Build a homotopy equivalence from the `homotopy-equivalence` *hmeq* and the `morphism` *lb-perturbation*, a perturbation of the dif-

ferential of the left bottom chain complex of *hmeq*. From *hmeq*:



the method `add` builds the new homotopy equivalence:



where the construction steps are the following:

– The reduction ρ'_1 is computed with the method `add` for a reduction with arguments: the left reduction of *hmeq* and the perturbation *lb-perturbation*. The trivial perturbation lemma is applied. As a bonus, this returns also the top perturbation, say δ to be applied to $\hat{C}$.

– The reduction ρ'_2 is computed with the previous method `add` with arguments: the right reduction of *hmeq* and the top perturbation δ just computed. That time, the basic perturbation lemma is applied.

– The function `build-hmeq` is called with parameters ρ'_1 and ρ'_2.

Example

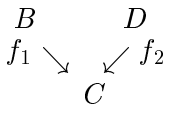
We shall apply these methods and functions in the following chapters. Let us show on a trivial example how this machinery works. First, we build a trivial homotopy equivalence on Δ^4. Then we build a new one by perturbing the differential of the left bottom chain complex with its opposite. The last call shows how this perturbation has been propagated.

```
(setf hmeq (trivial-hmeq (cdelta 4)))  ==>

[K6 Homotopy-Equivalence]
```

```
(setf hmeq (add hmeq (opps (dffr (cdelta 4)))))  ==>

[K14 Homotopy-Equivalence]

(gnrt-? (dffr (rbcc hmeq)) 3 '(0 1 2 3)))  ==>

------------------------------------------------------------------------{CMBN 2}
-------------------------------------------------------------------------------
```

2.5 Bicones

Given 3 chain complexes B, C and D, together with 2 chain homomorphisms f_1 and f_2 of degree 0 (commuting with the differentials), as shown in the following diagram, a bicone on these chain complexes is a chain complex, $BCN(f_1, f_2)$ where the n-th chain group $[BCN(f_1, f_2)]$ is $B_n \oplus C_{n+1} \oplus D_n$.



2.5.1 Representation of a combination in a bicone.

To distinguish to which chain complex belongs a generator in a combination of a bicone, the following convention has been adopted: if `gb` is a generator of any degree of B, it will be represented in the bicone by the list `(:BcnB gb)` and printed as `<BcnB gb>`. The symbols for C and D are respectively `:BcnC` and `:BcnD`. The 3 macros `bcnb`, `bcnc` and `bcnd` may be used to build such a generator. The building function for combinations, `cmbn`, may be used in the following way:

```
(setf comb-bic (cmbn 3 2 (bcnb 'b1) 4 (bcnb 'b2) 6 (bcnb 'b3)
                       3 (bcnc 'c1) 5 (bcnc 'c2) 7 (bcnd 'd1)))

------------------------------------------------------------------------{CMBN 3}
<2 * <BcnB B1>>
<4 * <BcnB B2>>
<6 * <BcnB B3>>
<3 * <BcnC C1>>
<5 * <BcnC C2>>
<7 * <BcnD D1>>
-------------------------------------------------------------------------------

(cmbn-list comb-bic)  ==>

((2 :BCNB . B1) (4 :BCNB . B2) (6 :BCNB . B3)
 (3 :BCNC . C1) (5 :BCNC . C2) (7 :BCND . D1))
```

2.5.2 Useful functions and macros for bicones

`bcnb` *gnrt* *[Macro]*
Build the representation of the generator *gnrt* belonging to the chain complex B.

`bcnc` *gnrt* *[Macro]*
Build the representation of the generator *gnrt* belonging to the chain complex C.

`bcnd` *gnrt* *[Macro]*
Build the representation of the generator *gnrt* belonging to the chain complex D.

`make-bicn-cmbn` *cmbnb cmbnc cmbnd* *[Function]*
Build a bicone combination from the 3 combinations *cmbnb*, *cmbnc* and *cmbnd* belonging respectively and in that order to B, C and D. The degrees of the combinations *cmbnb* and *cmbnd* must be the same, say n. The degree of *cmbnc* must be $n+1$. The degree of the new created combination is n.

`bicn-cmbn-cmbnb` *cmbn* *[Function]*
Extract from the bicone combination of degree n, *cmbn*, the combination relative to B as a legal combination of degree n in B. If there is no B-component, return the null combination of degree n.

`bicn-cmbn-cmbnc` *cmbn* *[Function]*
Extract from the bicone combination of degree n, *cmbn*, the combination relative to C as a legal combination of degree $n+1$ in C. If there is no C-component, return the null combination of degree $n+1$.

`bicn-cmbn-cmbnd` *cmbn* *[Function]*
Extract from the bicone combination of degree n, *cmbn*, the combination relative to D as a legal combination of degree n in D. If there is no D-component, return the null combination of degree n.

`dispatch-bicn-cmbn` *cmbn* *[Function]*
Give the 3-values result constitued by the 3 combinations components of the bicone combination *cmbn*. These combinations are valid combinations in their respective chain complexes.

Examples

```
(setf comb-b (cmbn 3 2 'b1 4 'b2 6 'b3)) ==>

-------------------------------------------------------------------------{CMBN 3}
<2 * B1>
<4 * B2>
<6 * B2>
-------------------------------------------------------------------------------

(setf comb-c (cmbn 4 3 'c1 5 'c2))  ==>

-------------------------------------------------------------------------{CMBN 4}
<3 * C1>
<5 * C2>
-------------------------------------------------------------------------------

(setf comb-d (cmbn 3 7 'd1))  ==>

-------------------------------------------------------------------------{CMBN 3}
<7 * D1>
-------------------------------------------------------------------------------

(setf comb-bic (make-bicn-cmbn comb-b comb-c comb-d))  ==>

-------------------------------------------------------------------------{CMBN 3}
<2 * <BcnB B1>>
<4 * <BcnB B2>>
<6 * <BcnB B3>>
<3 * <BcnC C1>>
<5 * <BcnC C2>>
<7 * <BcnD D1>>
-------------------------------------------------------------------------------

(bicn-cmbn-cmbnb comb-bic)  ==>

-------------------------------------------------------------------------{CMBN 3}
<2 * B1>
<4 * B2>
<6 * B2>
-------------------------------------------------------------------------------

(bicn-cmbn-cmbnc comb-bic)  ==>

-------------------------------------------------------------------------{CMBN 4}
<3 * C1>
<5 * C2>
-------------------------------------------------------------------------------
```

```
(bicn-cmbn-cmbnd comb-bic)  ==>

------------------------------------------------------------------------{CMBN 3}
<7 * D1>
-------------------------------------------------------------------------------

(dispatch-bicn-cmbn comb-bic)  ==>

------------------------------------------------------------------------{CMBN 3}
<2 * B1>
<4 * B2>
<6 * B2>
-------------------------------------------------------------------------------

------------------------------------------------------------------------{CMBN 4}
<3 * C1>
<5 * C2>
-------------------------------------------------------------------------------

------------------------------------------------------------------------{CMBN 3}
<7 * D1>
-------------------------------------------------------------------------------
```

2.6 Construction of a bicone from 2 reductions.

From the slots of the respective chain complexes B, C and D, it is possible to build the bicone chain complex. The 3 essential functions are the following:

`bicone-cmpr` *cmprb cmprc cmprd* *[Function]*
: From the 3 comparison functions *cmprb*, *cmprc* and *cmprd*, build a comparison function adequate to compare the generators as represented in the bicone.

`bicone-basis` *basisb basisc basisd* *[Function]*
: From the 3 basis function *basisb*, *basisc* and *basisd*, build a basis function for the bicone. If at least one of the chain complex component of the bicone is *locally effective*, the function returns the symbol `:locally-effective`.

`bicone-intr-dffr` *cmprc dffrb dffrc dffrd f1 f2* *[Function]*
: Define the differential in the bicone according to the formula:

$$d(cb, cc, cd) = (d_B(cb), f_1(cb) + f_2(cd) - d_C(cc), d_D(cd)),$$

where the notation is self explanatory. One sees that one needs the comparison operator *cmprc* of C, to order correctly the second combination of the resulting triple.

Let us consider now 2 reductions $(\hat{C}_1, C_1, f1, g1, h1)$ and $(\hat{C}_2, C_2, f2, g2, h2)$ in which $C_1 = C_2$. It is possible to define a bicone, with the identification $\hat{C}_1 = B, C_1 = C_2 = C, \hat{C}_2 = D$. This is realised by the following function:

`bicone` *rdct1 rdct2* *[Function]*

Build the bicone chain complex from the objects ot type `reduction` *rdct1* and *rdct2*. The bicone is built with the following identification: B is the top chain complex of *rdct1*, D is the top chain complex of *rdct2*. The bottom chain complexes of both *rdct1* and *rdct2* must be the same and C is that chain complex. Of course, both chain morphisms f_1 and f_2 are respectively the morphism f of the reductions. The construction of the bicone is realised by a call to the building function `build-chcm` using the 3 functions `bicone-cmpr`, `bicone-basis` and `bicone-intr-dffr` with arguments coming from *rdct1* an *rdct2*. The slot `bsgn` (base point) is let undefined and the strategy is by combination.

Example

Let us take again the example with the standard simplex Δ^n. The following function defines an effective version of $\mathcal{C}_*(\Delta^n)$, so we may ask for the basis in some dimensions.

```
(defun cdelta (dmns)
    (build-chcm
       :cmpr #'l-cmpr
       :basis #'(lambda (n)
                   (mapcar #'dlop-int-ext (funcall (delta-n-basis dmns) n)))
       :bsgn '(0)
       :intr-dffr #'(lambda (degr gmsm)
                      (make-cmbn
                         :degr (1- degr)
                         :list (do ((rslt +empty-list+
                                          (cons (cons sign
                                                      (append
                                                        (subseq gmsm 0 nark)
                                                        (subseq gmsm (1+ nark))))
                                                rslt))
                                   (sign 1 (- sign))
                                   (nark 0 (1+ nark)))
```

```
                                    ((> nark degr) rslt))))
        :strt :gnrt
        :orgn `(Effective version of C_* delta ,dmns)))


(setf delta3 (cdelta 3))  ==>

[K1 Chain-Complex]

(basis delta3 0)  ==>

((0) (1) (2) (3))

(basis delta3 1)  ==>

((0 1) (0 2) (1 2) (0 3) (1 3) (2 3))

(basis delta3 2)  ==>

((0 1 2) (0 1 3) (0 2 3) (1 2 3))

(basis delta3 3)  ==>

((0 1 2 3))

(basis delta3 4)  ==>

NIL
```

Using the function `make-rdct` defined in previous example which builds a reduction between Δ^m and Δ^n, we may create a bicone chain complex with the function `bicone` and ask for basis in some dimensions.

```
(setf bic (bicone (make-rdct 3 2)(make-rdct 4 2)))  ==>

[K15 Chain-Complex]

(basis bic 0)  ==>

(<BcnB (0)> <BcnB (1)> <BcnB (2)> <BcnB (3)>
 <BcnC (0 1)> <BcnC (0 2)> <BcnC (1 2)>
 <BcnD (0)> <BcnD (1)> <BcnD (2)> <BcnD (3)> <BcnD (4)>)

(basis bic 1)  ==>

(<BcnB (0 1)> <BcnB (0 2)> <BcnB (1 2)> <BcnB (0 3)> <BcnB (1 3)> <BcnB (2 3)>
 <BcnC (0 1 2)>
 <BcnD (0 1)> <BcnD (0 2)> <BcnD (1 2)> <BcnD (0 3)> <BcnD (1 3)> <BcnD (2 3)>
```

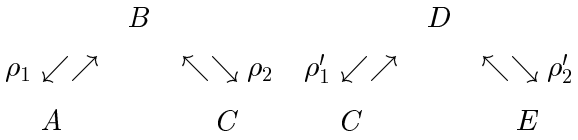
```
 <BcnD (0 4)> <BcnD (1 4)> <BcnD (2 4)> <BcnD (3 4)>)

(basis bic 4)  ==>

(<BcnD (0 1 2 3 4)>)
```

We verify the validity of the differential in the bicone `bic`.

```
(? bic (cmbn 2 3 (bcnb '(0 1 3)) 4 (bcnc '(0 1 2 3)) 5 (bcnd '(0 1 4))))  ==>

-------------------------------------------------------------------------{CMBN 1}
<3 * <BcnB (0 1)>>
<-3 * <BcnB (0 3)>>
<3 * <BcnB (1 3)>>
<12 * <BcnC (0 1 2)>>
<-4 * <BcnC (0 1 3)>>
<4 * <BcnC (0 2 3)>>
<-4 * <BcnC (1 2 3)>>
<5 * <BcnD (0 1)>>
<-5 * <BcnD (0 4)>>
<5 * <BcnD (1 4)>>
-------------------------------------------------------------------------------

(? bic *)  ==>

-------------------------------------------------------------------------{CMBN 0}
-------------------------------------------------------------------------------
```

2.7 Composition of homotopy equivalences.

Let us consider 2 homotopy equivalences, as in the following diagram.



where the right bottom complex of the first one is the same as the left bottom reduction of the second. We may use then the bicone concept to build a bicone chain complex with B, C and D, say BCN, and finally build

a new homotopy equivalence between A, BCN and E.

This is realized by the method `cmps`.

`cmps` *hmeq1 hmeq2* *[Method]*
: Build a homotopy equivalence by composition of both homotopy equivalences *hmeq1*, *hmeq2*. The system controls the validity of the composition.

Example

Starting with a very simple chain complex and the trivial homotopy equivalence on the chain complex, we verify the correctness of various compositions. The chain complex `c` has only one vertex `(a)` in every dimension and the differential is the null morphism.

```
(setf c (build-chcm
             :cmpr #'s-cmpr
             :basis #'(lambda (dmns) '(a))
             :bsgn 'a
             :intr-dffr #'zero-intr-dffr
             :strt :cmbn
             :orgn '(c)))  ==>

[K1 Chain-Complex]
```

We build the trivial homotopy equivalence on `c` and we compose it with itself.

```
(setf h1 (trivial-hmeq c))  ==>

[K6 Homotopy-Equivalence]

(setf h2 (cmps h1 h1))  ==>

[K17 Homotopy-Equivalence]
```

We verify the coherency of the morphisms in the left and right reduction of `h2`. The combination `*tc*` must belong to a bicone whereas `*bc*` belongs to `c`.

```
(pre-check-rdct (lrdct h2))  ==>

---done---

(setf *tc* (cmbn 3 1 (bcnB 'a) 10 (bcnC 'a) 100 (bcnD 'a)))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <BcnB A>>
<10 * <BcnC A>>
<100 * <BcnD A>>
------------------------------------------------------------------------

(setf *bc* (cmbn 3 1 'a))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * A>
------------------------------------------------------------------------

(check-rdct)  ==>

*TC* =>
------------------------------------------------------------------------{CMBN 3}
<1 * <BcnB A>>
<10 * <BcnC A>>
<100 * <BcnD A>>
------------------------------------------------------------------------

*BC* =>
------------------------------------------------------------------------{CMBN 3}
<1 * A>
------------------------------------------------------------------------

Checking *TDD* = 0
Result:
------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------

Checking *BDD* = 0
Result:
------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------

Checking *DF-FD* = 0
Result:
------------------------------------------------------------------------{CMBN 2}
------------------------------------------------------------------------
```

```
Checking *DG-GD* = 0
Result:
-------------------------------------------------------------------------{CMBN 2}
-------------------------------------------------------------------------------

Checking *ID-FG* = 0
Result:
-------------------------------------------------------------------------{CMBN 3}
-------------------------------------------------------------------------------

Checking *ID-GF-DH-HD* = 0
Result:
-------------------------------------------------------------------------{CMBN 3}
-------------------------------------------------------------------------------

Checking *HH* = 0
Result:
-------------------------------------------------------------------------{CMBN 5}
-------------------------------------------------------------------------------

Checking *FH* = 0
Result:
-------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

Checking *HG* = 0
Result:
-------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

---done---

(pre-check-rdct (rrdct h2))  ==>

---done---

(check-rdct)  ==>

*TC* =>
-------------------------------------------------------------------------{CMBN 3}
<1 * <BcnB A>>
<10 * <BcnC A>>
<100 * <BcnD A>>
-------------------------------------------------------------------------------
```

```
*BC* =>
------------------------------------------------------------------------{CMBN 3}
<1 * A>
-------------------------------------------------------------------------------

Checking *TDD* = 0

------------------------------------------------------------------------{CMBN 2}
-------------------------------------------------------------------------------

..... all results NULL.....

Checking *HG* = 0
Result:
------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

---done---
```

We compose now `h2` with itself. We let `*bc*` unchanged, but `*tc*` is a little more complicated because the chain complexes B and D are themselves bicones.

```
(setf h3 (cmps h2 h2))  ==>

[K65 Homotopy-Equivalence]

(setf *tc* (cmbn 3 1 (bcnB (bcnB 'a)) 10 (bcnB (bcnC 'a)) 100 (bcnB (bcnD 'a))
                     1000 (bcnC 'a)
                    10000 (bcnD (bcnB 'a)) 5234 (bcnD (bcnC 'a))
                      223 (bcnD (bcnD 'a))))  ==>
------------------------------------------------------------------------{CMBN 3}
<1 * <BcnB <BcnB A>>>
<10 * <BcnB <BcnC A>>>
<100 * <BcnB <BcnD A>>>
<1000 * <BcnC A>>
<10000 * <BcnD <BcnB A>>>
<5234 * <BcnD <BcnC A>>>
<223 * <BcnD <BcnD A>>>
-------------------------------------------------------------------------------

(pre-check-rdct (lrdct h3))  ==>

---done---

(check-rdct)  ==>

*TC* =>
```

```
------------------------------------------------------------------------{CMBN 3}
<1 * <BcnB <BcnB A>>>
<10 * <BcnB <BcnC A>>>
<100 * <BcnB <BcnD A>>>
<1000 * <BcnC A>>
<10000 * <BcnD <BcnB A>>>
<5234 * <BcnD <BcnC A>>>
<223 * <BcnD <BcnD A>>>
-------------------------------------------------------------------------------

*BC* =>
------------------------------------------------------------------------{CMBN 3}
<1 * A>
-------------------------------------------------------------------------------

Checking *TDD* = 0
Result:
------------------------------------------------------------------------{CMBN 1}
-------------------------------------------------------------------------------

..... all results NULL.....

Checking *HG* = 0
Result:
------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

---done---

(pre-check-rdct (rrdct h3))  ==>

---done---

(check-rdct)  ==>


*TC* =>
------------------------------------------------------------------------{CMBN 3}
<1 * <BcnB <BcnB A>>>
<10 * <BcnB <BcnC A>>>
<100 * <BcnB <BcnD A>>>
<1000 * <BcnC A>>
<10000 * <BcnD <BcnB A>>>
<5234 * <BcnD <BcnC A>>>
<223 * <BcnD <BcnD A>>>
-------------------------------------------------------------------------------
```

```
*BC* =>
------------------------------------------------------------------------{CMBN 3}
<1 * A>
-------------------------------------------------------------------------------

Checking *TDD* = 0
Result:
------------------------------------------------------------------------{CMBN 1}
-------------------------------------------------------------------------------

..... all results NULL.....

Checking *HG* = 0
Result:
------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

---done---
```

Lisp files concerned in this chapter

`effective-homology.lisp`, `cones.lisp`.
[`classes.lisp` , `macros.lisp`, `various.lisp`].

Chapter 3

The Homology module

This chapter is devoted to the description of the functions for computing the homology groups of a chain complex.

3.1 List of functions

`chcm-homology` *chcm dim* *[Function]*

Return a description of the homology group in dimension *dim* of the chain complex *chcm* in terms of *components* of the form $\mathbb{Z}$ or $\mathbb{Z}/n\,\mathbb{Z}$. The desired homology group is the direct sum of these components. No canonical presentation is looked for, so that, if for instance, a homology group is $\mathbb{Z}/6\,\mathbb{Z}$, it can be displayed as one component $\mathbb{Z}/6\,\mathbb{Z}$ or two components $\mathbb{Z}/2\,\mathbb{Z}$ and $\mathbb{Z}/3\,\mathbb{Z}$. On the other hand, if the component part is void, this means the homology group is the null group. The function `chcm-homology` implements the usual algorithms to compute the homology group associated to two integer matrices, the composite of which is null. The current version is verbose: for each group asked for, the program displays the rank of each integer matrix and each generator of the source module. Timing indications are also given. In the examples, only the components are printed.

`chcm-homology-gen` *chcm n* *[Function]*

This function computes the homology group in dimension n of the chain complex *chcm* and prints a generator of degree n in *chcm* for each component of the group (in general a combination of the basis elements of degree n of the chain complex).

`chcm-mat` *chcm n* *[Function]*

Return the the matrix of the linear homomorphism $d_n : C_n \to C_{n-1}$, where C_n is the n–th chain group of the chain complex *chcm*. More precisely, each column of this matrix contains the integer coefficients in the basis of C_{n-1} of the image by d_n of each basis element of C_n. So the number of lines (resp. columns) of the matrix is the number of elements of the basis of C_{n-1} (resp. C_n). The homology group $\mathcal{H}_n = \mathcal{Z}_n/\mathcal{B}_n$ is computed from the two matrices `MZ` = `chcm-mat` *(chcm,n)* and `NB` = `chcm-mat` *(chcm, n+1)*. By well known algorithms on matrix reduction[1], the matrix `MZ` is used to find a basis for the kernel $\mathcal{Z}_n$ and the matrix `NB`, to find in $\mathcal{Z}_n$ a presentation of the group $\mathcal{H}_n = \mathcal{Z}_n/\mathcal{B}_n$ by generators and relations. This is performed by the internal function `homologie` (beware: not `homology`!).

`homology` *chcm degr1* `&optional` *(degr2 (1+ degr1))* *[Method]*

Compute the homology groups from $degr_1$ to $degr_2 - 1$ (default: only $degr_1$, if *degr2* is omitted) of the right chain complex of the homotopy equivalence contained in the slot `:efhm` of the chain complex instance *chcm*. At the creation of the chain complex, this slot is unbound and is set during execution by an adequate CLOS method. If this slot cannot be bound, an error is returned and the homology groups cannot be computed. A more elaborated explanation of the mechanism used by the function `homology` is given in the section: **The general method for computing homology groups**.

Example

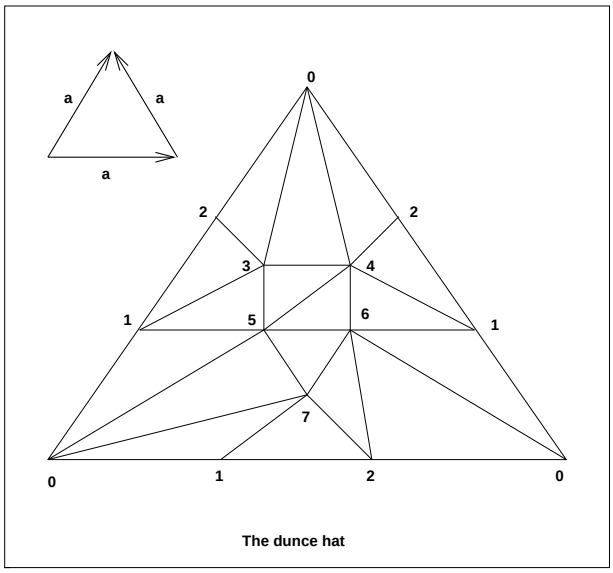
Let us get the homology groups for the example `diabolo` of the chapter 1.

```
(chcm-homology diabolo 0)   ==>

Computing boundary-matrix in dimension 0.
Rank of the source-module : 6.
...

Homology in dimension 0 :

Component Z
```

[1] **S. MacLane & G. Birkhoff**, *Algebra*, The MacMillan Company, 1967

```
(chcm-homology diabolo 1)   ==>

Computing boundary-matrix in dimension 1.
Rank of the source-module : 7.
...

Homology in dimension 1 :

Component Z

(chcm-homology diabolo 2)   ==>

Computing boundary-matrix in dimension 2.
Rank of the source-module : 1.
...

Homology in dimension 2 :

---done---
```

Another simple example is the following 2–chain complex corresponding to the well known *dunce hat*. We shall see later in the chapter `Simplicial Sets` a much more elegant method to describe this object.



The dunce hat

The diagram shows a permissible triangulation. For the generators, we have chosen lists rather symbols: a vertex s_i is represented as `(i)`, an edge $s_i s_j$ as

`(i j)` and a triangle $s_i s_j s_k$ as `(i j k)`. The enumeration of the elements of the basis is a little cumbersome but defining the boundary homomorphism is very easy, bearing in mind the boundary rule:

$$\mathbf{d}[s_0 s_1 \dots s_n] = \sum_{i=0}^{n} (-1)^i s_0 s_1 \dots \widehat{s_i} \dots s_n.$$

As for diabolo, the vertices are implicitly ordered.

```
(setf duncehat-basis #'(lambda(dmn)
  (case dmn
     (0 '((0) (1) (2) (3) (4) (5) (6) (7) ))
     (1 '((0 1)(0 2)(0 3)
          (0 4)(0 5)(0 6)
          (0 7)(1 2)(1 3)
          (1 4)(1 5)(1 6)
          (1 7)(2 3)(2 4)
          (2 6)(2 7)(3 4)
          (3 5)(4 5)(4 6)
          (5 6)(5 7)(6 7) ))
     (2 '((0 1 5)(0 1 6)(0 1 7)
          (0 2 3)(0 2 4)(0 2 6)
          (0 3 4)(0 5 7)(1 2 3)
          (1 2 4)(1 2 7)(1 3 5)
          (1 4 6)(2 6 7)(3 4 5)
          (4 5 6)(5 6 7) ))
     (otherwise nil)) ))

(setf duncehat-df #'(lambda(dmn gnr)
   (case dmn
      (0 (cmbn -1))
      (1 (cmbn 0  -1 (list(first gnr)) 1 (rest gnr)))
      (2 (cmbn 1 1 (list(first gnr) (second gnr))
               -1 (list(first gnr) (third gnr))
                1 (rest gnr) ))
      (otherwise (error "bad generator for dunce hat")))))


(setf duncehat (build-chcm :cmpr #'l-cmpr
                           :basis duncehat-basis
                           :bsgn '(0)
                           :intr-dffr  duncehat-df
                           :strt :gnrt
                           :orgn '(dunce hat)))
```

```
(chcm-homology duncehat 0)   ==>

Computing boundary-matrix in dimension 0.
Rank of the source-module : 9.
...

Homology in dimension 0 :

Component Z

---done---

(chcm-homology duncehat 1)   ==>

Computing boundary-matrix in dimension 1.
Rank of the source-module : 27.
...

Homology in dimension 1 :

---done---

(chcm-homology duncehat 2)  ==>

Computing boundary-matrix in dimension 2.
Rank of the source-module : 19.
...

Homology in dimension 2 :

---done---
```

Let us take again the chain complex `duncehat`. The two matrices of the homomorphisms $d_1 : C_1 \to C_0$ and $d_2 : C_2 \to C_1$ are obtained by calling `chcm-mat`.

```
(setf mz (chcm-mat duncehat 1))  ==>

========== MATRIX 8 lines + 24 columns =====
L1=[C1=-1][C2=-1][C3=-1][C4=-1][C5=-1][C6=-1][C7=-1]
L2=[C1=1][C8=-1][C9=-1][C10=-1][C11=-1][C12=-1][C13=-1]
L3=[C2=1][C8=1][C14=-1][C15=-1][C16=-1][C17=-1]
L4=[C3=1][C9=1][C14=1][C18=-1][C19=-1]
L5=[C4=1][C10=1][C15=1][C18=1][C20=-1][C21=-1]
L6=[C5=1][C11=1][C19=1][C20=1][C22=-1][C23=-1]
L7=[C6=1][C12=1][C16=1][C21=1][C22=1][C24=-1]
L8=[C7=1][C13=1][C17=1][C23=1][C24=1]
========== END-MATRIX
```

```
(setf nb (chcm-mat duncehat 2))  ==>

========== MATRIX 24 lines + 17 columns =====
L1=[C1=1][C2=1][C3=1]
L2=[C4=1][C5=1][C6=1]
L3=[C4=-1][C7=1]
L4=[C5=-1][C7=-1]
L5=[C1=-1][C8=1]
L6=[C2=-1][C6=-1]
L7=[C3=-1][C8=-1]
L8=[C9=1][C10=1][C11=1]
L9=[C9=-1][C12=1]
L10=[C10=-1][C13=1]
L11=[C1=1][C12=-1]
L12=[C2=1][C13=-1]
L13=[C3=1][C11=-1]
L14=[C4=1][C9=1]
L15=[C5=1][C10=1]
L16=[C6=1][C14=1]
L17=[C11=1][C14=-1]
L18=[C7=1][C15=1]
L19=[C12=1][C15=-1]
L20=[C15=1][C16=1]
L21=[C13=1][C16=-1]
L22=[C16=1][C17=1]
L23=[C8=1][C17=-1]
L24=[C14=1][C17=1]
========== END-MATRIX

(homologie mz nb)  ==>

NIL
```

3.2 The general method for computing homology

Among the slots of the instance of an object inheriting the `CHAIN COMPLEX` class, a slot `efhm` has been reserved to point (possibly) to a homotopy equivalence where the right bottom chain complex is effective. The function `homology` is designed to get the right bottom chain complex of the homotopy equivalence value of the slot. If the slot has been bound, then the homology groups are computed by the function `chcm-homology` as shown in the following definition:

```
(DEFMETHOD HOMOLOGY ((chcm chain-complex) degr1 &optional (degr2 (1+ degr1)))
   (do ((degr degr1 (1+ degr)))
       ((>= degr degr2))
 -->   (chcm-homology (rbcc (efhm chcm)) degr)
       (terpri) (clock) (terpri)))
```

But, at the creation of the object, the slot `efhm` is unbound and as soon as the `homology` function tries to get the content of the slot `efhm` via the call `(efchm chcm)`, the *slot-unbound* mechanism of CLOS is triggered, calling at its turn a method `search-efhm` depending on the object. If no method is available for this object, `NIL` is returned. In the following chapters, we shall see that cases have been written for cartesian products, tensor products, suspensions, disk pasting, fibrations, loop spaces, and classifying spaces. The selection of the case is done thanks to the information contained in the comment list of the object (slot `orgn`). In each case, the `search-efhm` method builds – in general with a complex machinery, including a possible recursivity – a homotopy equivalence where the right bottom chain complex is *effective*. The slot `efhm` of the object is then settled and the homology group computation may begin. If `NIL` is returned, then and only at this moment, the slot-unbound mechanism looks if the chain complex is finite (checking if a basis function exists). If this is the case, then the *trivial homotopy equivalence* is built upon the chain complex and this gives the value of the slot `efhm`. If there is no basis function, meaning that probably the chain complex is locally effective, an error is returned.

Remark. The user may wonder why one does not look first if the given object is effective. We recall simply that even in the case of an effective chain complex, it is sometimes possible to find another effective chain complex, homotopic to the first and whose number of basis elements, in any dimensions, is considerably smaller in comparison with the first one. A striking case will be shown in a further chapter, showing the application of the Eilenberg-Zilber theorem to a cartesian product.

Example

The reading of this subsection may be postponed until a full reading of this user's guide. The following examples show how are filled the slots `efhm` of the various objects. Starting from the sphere S^2, we verify first that the `efhm` slot is unbound. Then we ask for $\mathcal{H}_1$ and verify that, though there is a finite basis, `Kenzo` has nevertheless built a trivial homotopy equivalence on this object.

```
(setf s2 (sphere 2))  ==>

[K1 Simplicial-Set]

(inspect s2)  ==>

SIMPLICIAL-SET @ #x39a1e2 = [K1 Simplicial-Set]
   0 Class --------> #<STANDARD-CLASS SIMPLICIAL-SET>
   1 ORGN ---------> (SPHERE 2), a proper list with 2 elements
   2 IDNM ---------> fixnum 1 [#x00000004]
-->3 EFHM ---------> The symbol :--UNBOUND--
   4 GRMD ---------> [K1 Simplicial-Set]
   5 DFFR ---------> [K2 Morphism (degree -1)]
   6 BSGN ---------> The symbol *
-->7 BASIS --------> #<Closure (FLET SPHERE-BASIS RSLT) @ #x39a14a>
   8 CMPR ---------> #<Function SPHERE-CMPR>
   9 CPRD ---------> [K5 Morphism (degree 0)]
  10 FACE ---------> #<Closure (FLET SPHERE-FACE RSLT) @ #x39a172>

(homology s2 1)  ==>

Homology in dimension 1 :

---done---

(inspect s2)  ==>

SIMPLICIAL-SET @ #x410372 = [K1 Simplicial-Set]

   ........................

-->3 EFHM ---------> [K9 Homotopy-Equivalence]

   ........................

(orgn (hmeq 9))  ==>

(TRIVIAL-HMEQ [K1 Simplicial-Set])
```

Now, we create $\Omega^1(S^2)$. Getting the value of the slot `efhm` by a call to the

accessor function **efhm**, triggers the search-efhm method for a loop space. A homotopy equivalence is built and the slot is set.

```
(setf os2 (loop-space s2))  ==>

[K10 Simplicial-Group]

(inspect os2)  ==>

SIMPLICIAL-GROUP @ #x4a208a = [K10 Simplicial-Group]

   ......................

-->3 EFHM ---------> The symbol :--UNBOUND--

   ......................

(orgn os2)  ==>

(LOOP-SPACE [K1 Simplicial-Set])

(efhm os2)  ==>

[K118 Homotopy-Equivalence]

(inspect os2)  ==>

SIMPLICIAL-GROUP @ #x30943a = [K10 Simplicial-Group]

   ......................

   3 EFHM ---------> [K118 Homotopy-Equivalence]

   ......................
```

The following example shows the recursion mechanism when one wants to get the value of the slot **efhm** of an iterated loop space, namely $\Omega^3(S^4)$.

```
(setf s4 (sphere 4))  ==>

[K119 Simplicial-Set]

(inspect s4)  ==>

SIMPLICIAL-SET @ #x38647a = [K119 Simplicial-Set]

   ......................
```

```
   3 EFHM ---------> The symbol :--UNBOUND--

   ........................

(setf ooos4 (loop-space (loop-space (loop-space s4))))  ==>

[K148 Simplicial-Group]

(orgn ooos4)  ==>

(LOOP-SPACE [K136 Simplicial-Group])

(orgn (second *))  ==>

(LOOP-SPACE [K124 Simplicial-Group])

(orgn (second *))  ==>

(LOOP-SPACE [K119 Simplicial-Set])

(orgn (second *))  ==>

(SPHERE 4)

(inspect (smgr 136))  ==>

SIMPLICIAL-GROUP @ #x463312 = [K136 Simplicial-Group]

   ........................

   3 EFHM ---------> The symbol :--UNBOUND--

   ........................

(inspect (smgr 124))  ==>

SIMPLICIAL-GROUP @ #x460efa = [K124 Simplicial-Group]

   ........................

   3 EFHM ---------> The symbol :--UNBOUND--

   ........................

(inspect (smst 119))  ==>
```

```
SIMPLICIAL-SET @ #x4102fa = [K119 Simplicial-Set]
   0 Class --------> #<STANDARD-CLASS SIMPLICIAL-SET>
   1 ORGN ---------> (SPHERE 4), a proper list with 2 elements

   ..........................

   3 EFHM ---------> The symbol :--UNBOUND--

   ..........................

(efhm ooos4)  ==>

[K522 Homotopy-Equivalence]

(inspect (smgr 136))  ==>

SIMPLICIAL-GROUP @ #x410e62 = [K136 Simplicial-Group]

   ..........................

   3 EFHM ---------> [K474 Homotopy-Equivalence]

   ..........................

(inspect (smgr 124))  ==>

SIMPLICIAL-GROUP @ #x41066a = [K124 Simplicial-Group]

   ..........................

   3 EFHM ---------> [K426 Homotopy-Equivalence]

   ..........................

(inspect (smst 119))  ==>

SIMPLICIAL-SET @ #x4191d2 = [K119 Simplicial-Set]

   ..........................

   3 EFHM ---------> [K412 Homotopy-Equivalence]

   ..........................

(inspect s4)  ==>

SIMPLICIAL-SET @ #x4191d2 = [K119 Simplicial-Set]
```

```
.........................

3 EFHM ---------> [K412 Homotopy-Equivalence]

.........................
```

Lisp files concerned in this chapter

`homology-groups.lisp`, `searching-homology` and files containing a `search-efhm` method.

Chapter 4

Tensor product of chain complexes

4.1 Introduction

One knows that the homology groups of a cartesian product of two spaces K and L may be obtained by considering a chain complex derived from respective chain complexes of K and L by taking their *tensor product*. But this is only one of the numerous uses of tensor products of chain complexes in algebraic topology, so the `Kenzo` software provides the handling of such an important tool.
Let us recall that chain complexes are free $\mathbb{Z}$–modules with distinguished basis. A tensor product of chain complexes is itself a free $\mathbb{Z}$–module with a natural basis formed by the tensor product of the generators of the chain complex factors. The program conforms to that rule.

4.2 Tensor product of generators and combinations

An elementary tensor product of two generators is represented, in the software, by a structured list in which we may find 4 items for the description of the two generators together with their respective degree. The internal representation of $gnrt1 \otimes gnrt2$ has the form:

(:tnpr (*degr1*.*gnrt1*).(*degr2*.*gnrt2*))

where,

1. *degr1* is an integer, the degree of the generator *gnrt1*.
2. *gnrt1* is the first generator of the pair.
3. *degr2* is an integer, the degree of the generator *gnrt2*.
4. *gnrt2* is the second generator of the pair.

To construct such an object, one may use the macro `tnpr`. The corresponding type is `TNPR`. The printing method prints the tensor product under the form:

`<TnPr` *gnrt1 gnrt2*`>` or `<TnPr` *degr1 gnrt1 degr2 gnrt2*`>`

according to the boolean value, `NIL` (default) or `T` of the system variable `*tnpr-with-degrees*` (see the examples).

4.2.1 Simple functions for the tensor product

`tnpr` *degr1 gnrt1 degr2 gnrt2* *[Macro]*
Build a tensor product $gnrt1 \otimes gnrt2$.

`tnpr-p` *object* *[Predicate]*
Test if *object* is of type `TNPR`.

`degr1` *tnpr* *[Macro]*
Select the degree of the first generator in the tensor product *tnpr*.

`gnrt1` *tnpr* *[Macro]*
Select the first generator from the object *tnpr*.

`degr2` *tnpr* *[Macro]*
Select the degree of the second generator in the tensor product *tnpr*.

`gnrt2` *tnpr* *[Macro]*
Select the second generator from the object *tnpr*.

`2cmbn-tnpr` *cmbn1 cmbn2* *[Function]*
Create, from two combinations *cmbn1* and *cmbn2* with respective degree *degr1* and *degr2*, a combination of degree $dgr1 + dgr2$ by applying the tensorial distributive law on the two sums of terms of the combinations.

Example

```
(tnpr 1 'a 2 'b)  ==>

<TnPr A B>

(tnpr-p *)  ==>

T

(tnpr-p (cmbn 0 1 'a 2 'b))  ==>

NIL

(setf *tnpr-with-degrees* t)  ==>

T

(2cmbn-tnpr (cmbn 2 3 'a 4 'b -5 'c) (cmbn 3 4 'x -3 'y 2 'z))  ==>

------------------------------------------------------------------------{CMBN 5}
<12 * <TnPr 2 A 3 X>>
<-9 * <TnPr 2 A 3 Y>>
<6 * <TnPr 2 A 3 Z>>
<16 * <TnPr 2 B 3 X>>
<-12 * <TnPr 2 B 3 Y>>
<8 * <TnPr 2 B 3 Z>>
<-20 * <TnPr 2 C 3 X>>
<15 * <TnPr 2 C 3 Y>>
<-10 * <TnPr 2 C 3 Z>>
------------------------------------------------------------------------------

(setf *tnpr-with-degrees* nil)  ==>

NIL

**  ==>      ;;; ** is the last but one result
------------------------------------------------------------------------{CMBN 5}
<12 * <TnPr A X>>
<-9 * <TnPr A Y>>
<6 * <TnPr A Z>>
<16 * <TnPr B X>>
<-12 * <TnPr B Y>>
<8 * <TnPr B Z>>
<-20 * <TnPr C X>>
<15 * <TnPr C Y>>
<-10 * <TnPr C Z>>
------------------------------------------------------------------------------
```

4.3 Tensor product of chain complexes

The software implements the tensor product of chain complexes according to the classical following definition. Let C and C' two chain complexes. The tensor product $C \otimes C'$ is the chain complex D such that:

$$D_p = \bigoplus_{m+n=p} C_m \otimes C'_n,$$

$C_m \otimes C'_n$ being the tensor product of the two modules C_m and C'_n. A basis for D_p is the union of the basis of $C_m \otimes C'_n$, with $m + n = p$.
The boundary operator $d^{\otimes}$ is defined, according to the Koszul rule, by:

$$d^{\otimes}(c_m \otimes c'_n) = d(c_m) \otimes c'_n + (-1)^m c_m \otimes d'(c'_n),$$

with $c_m \in C_m$, $c'_n \in C'_n$ and d, d' being the respective boundary operators of C and C'.
In the software, this is realized by the function `tnsr-prdc`.

`tnsr-prdc` *chcm1 chcm2* *[Method]*

Build a chain complex, tensor product of the two chain complexes *chcm1* and *chcm2*. The elements of this new chain complex are integer combinations of generators of `TNPR` type. The creation of this new chain complex is done by a call to the function `build-chcm` with actual parameters defined from the constituting elements of *chcm1* and *chcm2*, according to the mathematical definitions above. If both arguments are *effective*, the function constructs an *effective* chain complex. On the other hand, if at least one of the chain complex is *locally effective*, the tensor product is also *locally effective*. In fact, the construction is correct only if both chain complexes are null in negative degrees, otherwise the result is undefined.

Examples

Let us take the standard 2–simplex, Δ^2 and let us build $C_*(\Delta^2) \otimes C_*(\Delta^2)$. To build the corresponding chain complex, we use the function `cdelta`, defined in a previous chapter.

```
(setf triangle (cdelta 2))  ==>

[K1 Chain-Complex]
```

```
(basis triangle 1)  ==>

((0 1) (0 2) (1 2))

(setf tpr-triangles (tnsr-prdc triangle triangle))  ==>

[K3 Chain-Complex]
```

Let us inspect some basis of this newly created chain complex.

```
(basis tpr-triangles 0)  ==>

(<TnPr (0) (0)> <TnPr (0) (1)> <TnPr (0) (2)> <TnPr (1) (0)> <TnPr (1) (1)>
 <TnPr (1) (2)> <TnPr (2) (0)> <TnPr (2) (1)> <TnPr (2) (2)>)

(basis tpr-triangles 1)  ==>

(<TnPr (0) (0 1)> <TnPr (0) (0 2)> <TnPr (0) (1 2)> <TnPr (1) (0 1)>
 <TnPr (1) (0 2)> <TnPr (1) (1 2)> <TnPr (2) (0 1)> <TnPr (2) (0 2)>
 <TnPr (2) (1 2)> <TnPr (0 1) (0)> <TnPr (0 1) (1)> <TnPr (0 1) (2)>
 <TnPr (0 2) (0)> <TnPr (0 2) (1)> <TnPr (0 2) (2)> <TnPr (1 2) (0)>
 <TnPr (1 2) (1)> <TnPr (1 2) (2)>)

(basis tpr-triangles 2)  ==>

(<TnPr (0) (0 1 2)> <TnPr (1) (0 1 2)> <TnPr (2) (0 1 2)> <TnPr (0 1) (0 1)>
 <TnPr (0 1) (0 2)> <TnPr (0 1) (1 2)> <TnPr (0 2) (0 1)> <TnPr (0 2) (0 2)>
 <TnPr (0 2) (1 2)> <TnPr (1 2) (0 1)> <TnPr (1 2) (0 2)> <TnPr (1 2) (1 2)>
 <TnPr (0 1 2) (0)> <TnPr (0 1 2) (1)> <TnPr (0 1 2) (2)>)

(basis tpr-triangles 3)  ==>

(<TnPr (0 1) (0 1 2)> <TnPr (0 2) (0 1 2)> <TnPr (1 2) (0 1 2)>
 <TnPr (0 1 2) (0 1)> <TnPr (0 1 2) (0 2)> <TnPr (0 1 2) (1 2)>)

(basis tpr-triangles 4)  ==>

(<TnPr (0 1 2) (0 1 2)>)
```

Let us consider now the chain complex, *ccn*, that we used in the chain complex chapter. The basis in any degree are decades produced by the function `<a-b<`. We build $ccn \otimes ccn$ and we verify the fundamental property of the associated boundary operator:

$$d^{\otimes} \circ d^{\otimes} = 0.$$

```
(setf ccn-boundary #'(lambda (dgr gnr)
     (if (evenp (+ dgr gnr))
         (cmbn (1- dgr) 1 (- gnr 10))
         (cmbn (1- dgr)))))

(setf ccn (build-chcm :cmpr #'f-cmpr
                      :basis #'(lambda (n) (<a-b< (* 10 n) (* 10 (1+ n))))
                      :intr-dffr  ccn-boundary
                      :strt :gnrt
                      :orgn '(ccn) ))  ==>

[K5 Chain-Complex]

(basis ccn 3)  ==>

(30 31 32 33 34 35 36 37 38 39)

(setf tpr-ccn-ccn (tnsr-prdc ccn ccn))   ==>

[K7 Chain-Complex]

(setf comb2 (cmbn 2 1 21 5 25 9 29))   ==>

-------------------------------------------------------------------------{CMBN 2}
<1 * 21>
<5 * 25>
<9 * 29>
-------------------------------------------------------------------------------

(setf comb3 (cmbn 3 2 32 3 33 -4 34 -6 36))  ==>

-------------------------------------------------------------------------{CMBN 3}
<2 * 32>
<3 * 33>
<-4 * 34>
<-6 * 36>
-------------------------------------------------------------------------------

(setf tcmb (2cmbn-tnpr comb2 comb3))  ==>

-------------------------------------------------------------------------{CMBN 5}
<2 * <TnPr 21 32>>
<3 * <TnPr 21 33>>
<-4 * <TnPr 21 34>>
<-6 * <TnPr 21 36>>
<10 * <TnPr 25 32>>
<15 * <TnPr 25 33>>
```

```
<-20 * <TnPr 25 34>>
<-30 * <TnPr 25 36>>
<18 * <TnPr 29 32>>
<27 * <TnPr 29 33>>
<-36 * <TnPr 29 34>>
<-54 * <TnPr 29 36>>
--------------------------------------------------------------------------------

(? tpr-ccn-ccn *)  ==>

------------------------------------------------------------------------{CMBN 4}
<3 * <TnPr 21 23>>
<15 * <TnPr 25 23>>
<27 * <TnPr 29 23>>
--------------------------------------------------------------------------------

(? tpr-ccn-ccn *)  ==>

------------------------------------------------------------------------{CMBN 3}
--------------------------------------------------------------------------------
```

4.4 Tensor product of morphisms, reductions, homotopy equivalences.

Let $f : A_1 \longrightarrow A_2$ and $g : B_1 \longrightarrow B_2$ two morphisms between two chain complexes. The tensor product $f \otimes g$ is the morphism

$$f \otimes g : A_1 \otimes B_1 \longrightarrow A_2 \otimes B_2$$

defined by the following formula respecting the Koszul rule:

$$(f \otimes g)(a_1 \otimes b_1) = (-1)^{deg(g)*deg(a1)} f(a_1) \otimes g(b_1).$$

The method `tnsr-prdc` already defined for chain complexes, may be also used for that purpose.

`tnsr-prdc` *mrph1 mrph2* *[Method]*
: Return the morphism, tensor product of the two morphisms *mrph1* and *mrph2*. The source and target of this new morphism are respectively the tensor product of the chain complexes source and target of *mrph1* and *mrph2*. The degree is the sum of the degrees of the parameters and the lisp function (the `:pure` keyword argument of the function `build-mrph`) conforms to the mathematical definition above. The strategy is by generator, i.e. the morphism is designed to work with 2 arguments: a degree and a generator which must be an object of type `TNPR`.

`tnsr-prdc` *rdct1 rdct2* *[Method]*
: Return the reduction, tensor product of the two reductions *rdct1* and *rdct2*. The algorithm consists essentially in defining

$$f = f1 \otimes f2,\ g = g1 \otimes g2,\ h = h1 \otimes (g2 \circ g2) + Id_{tcc1} \otimes h2,$$

where Id_{tcc1} is the identity morphism in the top chain complex of the reduction *rdct1*. The returned reduction is then built by a call to the method `build-rdct`. This defines completely the chain complexes involved in the reduction.

`tnsr-prdc` *hmeq1 hmeq2* *[Method]*
: Return the homotopy equivalence, tensor product of the two homotopy equivalence *heqm1* and *heqm2*. This is a homotopy equivalence where the new reductions are the tensor products of the arguments reductions (See the lisp definition just below).

4.4.1 Searching homology for tensor products

The comment list of a tensor product of two chain complexes has the form (`TNSR-PRDC` *chcm1 chcm2*). The `search-efhm` method applied to a tensor product, looks for the value of the respective `efhm` slots of the chain complexes *chcm1* and *chcm2*, (i.e. two homotopy equivalences) or tries to settle these slots if they are still unbound. Then it builds the tensor product of both homotopy equivalences, as shown in the following lisp listing. At its turn, the resulting homotopy equivalence will become the value of the `efhm` slot of the initial tensor product chain complex.

```
(defmethod SEARCH-EFHM (chcm (orgn (eql 'tnsr-prdc)))
  (declare (type chain-complex chcm))
  (the homotopy-equivalence
       (tnsr-prdc (efhm (second (orgn chcm)))
                  (efhm (third (orgn chcm))))))

(defmethod TNSR-PRDC ((hmeq1 homotopy-equivalence)
      (hmeq2 homotopy-equivalence))
  (the homotopy-equivalence
     (build-hmeq
        :lrdct (tnsr-prdc (lrdct hmeq1) (lrdct hmeq2))
        :rrdct (tnsr-prdc (rrdct hmeq1) (rrdct hmeq2))
        :orgn `(tnsr-prdc ,hmeq1 ,hmeq2))))
```

Lisp file concerned in this chapter

`tensor-products.lisp`, `searching-homology.lisp`.

Chapter 5

Coalgebras and cobars

5.1 Introduction

The implementation of coalgebras and cobars follows closely the section 6 of the paper *Effective Homology: a survey*[1].

5.2 Coalgebras

A coalgebra is a pair (C, χ) where

1. C is a chain complex.
2. χ (the *coproduct*) is a chain complex morphism

$$\chi : C \to C \otimes C.$$

These components must satisfy the usual structural properties of the coalgebras[2].

5.2.1 Implementation of a coalgebra

A coalgebra is represented as an instance of the CLOS class `COALGEBRA`, subclass of the `CHAIN COMPLEX` class. The definition of this new class is simply:

```
(DEFCLASS COALGEBRA (chain-complex)
    ((cprd :type morphism :initarg :cprd :reader cprd1)))   ;; coproduct
```

[1] Available at the web site **http://www-fourier.ujf.grenoble.fr/~ sergerar/**

[2] **Mac Lane**, Homology, section VI-9. Springer 1970.

So, this new class inherits the slots of the `CHAIN COMPLEX` class and has a slot of its own, namely `cprd`, a morphism object representing the coproduct. The user will note the important following fact that, **once a coalgebra has been defined, one may use on it any function or method applicable to a chain complex**.

5.2.2 The function build-clgb

To facilitate the construction of instances of the `COALGEBRA` class, the software provides the function `build-clgb`,

`build-clgb` `:cmpr` *cmpr* `:basis` *basis* `:bsgn` *bsgn* `:intr-dffr` *intr-dffr*
`:dffr-strt` *dffr-strt* `:intr-cprd` *intr-cprd*
`:cprd-strt` *cprd-strt* `:orgn` *orgn*

defined with keyword parameters. The returned value is an instance of type `COALGEBRA`. The keyword arguments are:

- *cmpr*, a comparison function for generators.
- *basis*, the basis function for the underlying chain complex.
- *bsgn*, the base point, a generator of any type.
- *intr-dffr*, the differential lisp function for the chain complex.
- *dffr-strt*, the strategy (`:gnrt` or `:cmbn`) for the differential.
- *intr-cprd*, the lisp function for the coproduct of the coalgebra.
- *cprd-strt*, the strategy (`:gnrt` or `:cmbn`) for the coproduct.
- *orgn*, an adequate comment list.

The function `build-clgb` calls the function `build-chcm` and with the help of the function `build-mrph` builds the coproduct morphism of degree 0, to settle the slot `crpd` of the instance. According to the definition, the source slot of the coproduct morphism is the underlying chain complex and the target slot is the tensor product of this chain complex with itself. As to the other objects, an identification number n (slot `idnm`) is assigned to this Kenzo object and the `COALGEBRA` instance is pushed onto the list `*clgb-list*`. The associated printing method prints a string like `[K`*n* `Coalgebra]`.

5.2.3 Miscellaneous functions and macros for coalgebras

`cat-init` *[Function]*
Clear among others, the list `*clgb-list*`, list of user created coalgebras and reset the global counter to 1.

`clgb` *n* *[Function]*
Retrieve in the list `*clgb-list*` the coalgebra instance whose identification is *n*. If it does not exist, return `NIL`.

`cprd` *clgb* `&rest` *[Macro]*
Versatile macro relative to the coproduct of a coalgebra. The first argument is a `COALGEBRA` instance. With only one argument, this macro returns the coproduct morphism of the coalgebra. With more arguments, it applies the coproduct morphism on a combination (`(cprd` *clgb cmbn*`)`) or on a pair *degree, generator* (`(cprd` *clgb degr gnrt*`)`).

`change-chcm-to-clgb` *chcm* `:intr-cprd` *intr-cprd* `:cprd-strt` *cprd-strt* `:orgn` *orgn*
Build a coalgebra instance from the already created chain complex *chcm*. The user must give via key parameters a lisp function for the coproduct, the strategy and a comment list.

5.3 Cobar of a coalgebra

Let $\mathcal{A}$ be a coalgebra, $\mathcal{M}$ a right $\mathcal{A}$-comodule, $\mathcal{N}$ a left $\mathcal{A}$-comodule. It means that there exist coproducts $\chi_{\mathcal{M}}$ and $\chi_{\mathcal{N}}$:

$$\chi_{\mathcal{M}} : \mathcal{M} \longrightarrow \mathcal{M} \otimes \mathcal{A}, \quad \chi_{\mathcal{N}} : \mathcal{N} \longrightarrow \mathcal{A} \otimes \mathcal{N}.$$

It is possible to build a chain complex denoted $Cobar^{\mathcal{A}}(\mathcal{M},\mathcal{N})$,

$$Cobar^{\mathcal{A}}(\mathcal{M},\mathcal{N}) = \bigoplus_p \mathcal{M} \otimes \bar{\mathcal{A}}^{\otimes p} \otimes \mathcal{N}.$$

This section is devoted to the particular case $\mathcal{M} = \mathcal{N} = \mathbb{Z}$.
So, let $\mathcal{A}$ be a coalgebra , assumed 1–connected, i.e. $\mathcal{A}_1 = 0$ and $\mathcal{A}_0 \simeq \mathbb{Z}$, we consider the chain complex, $Cobar^{\mathcal{A}}(\mathbb{Z},\mathbb{Z})$, noted more simply $Cobar(\mathcal{A})$, whose n-th component is:

$$[Cobar(\mathcal{A})]_n = \bigoplus(\bar{\mathcal{A}}_{i1} \otimes \cdots \otimes \bar{\mathcal{A}}_{ik}), \quad i_j > 0, \quad i1 + \cdots + ik = n + k.$$

A tensor product $g_1 \otimes \cdots \otimes g_k$, where g_i is a generator of $\bar{\mathcal{A}}_i$ is called, in the software `Kenzo`, an **algebraic loop**. The reason is the following:

if X is a 1–connected simplicial set, we shall see in a next chapter that the program `Kenzo` constructs a homotopy equivalence between the chain complex of the loop space $\mathcal{C}_*(GX)$ and $Cobar(\mathcal{C}_*(X)) = Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathbb{Z})$, so that a generator of the last chain complex is a kind of "algebraic" version of a generator of a loop space. The definition of differential of the Cobar is recalled hereafter.

5.3.1 Representation of an algebraic loop

An algebraic loop is represented by a lisp object of the form:

$$\texttt{(:ALLP } (i_1 \,.\, g_1) \ \ldots (i_k \,.\, g_k)\texttt{)}$$

where the i_j are the degrees **in the cobar chain complex** of the generators g_j. In the original coalgebra $\mathcal{A}$, the generators g_j had the degree $i_j + 1$. The corresponding type is `ALLP`. The function to build such an object is also called `allp`

`allp` *degr1 gnr1 degr2 gnr2 ... degrk gnrk* *[Function]*
: Construct an algebraic loop, i.e. a tensorial product of degree $\sum degr_i$. The sequence of pairs $\{degr_i, gnr_i\}$ has an undefinite length and may be void. In this case, the algebraic loop is the null algebraic loop, identified in the system by the constant `+null-allp+`. The function `allp` accepts also as unique argument a list of the form (*degr1 gnr1 degr2 gnr2 ... degrk gnrk*)

`allp-p` *object* *[Function]*
: Test if *object* is an algebraic loop.

The associated printing method prints the object under the form:

$$\texttt{<<AlLp [}i_1\ g_1\texttt{] ... [}i_k\ g_k\texttt{]>>}$$

The user will have noted that in the Cobar, each generator g_i appears with its own degree in the underlying chain complex, **lowered by** 1.

5.3.2 Definition of the chain complex Cobar

The definition of the differential in the Cobar, will be done in three steps. This is better understood, if we consider the following diagram. We recall that the elements of $(\bar{\mathcal{A}}_*^{\otimes_p})_q$ are by convention of degree $q - p$ in the Cobar.

- A chain complex called `vertical-cobar` is defined with the *vertical differential* d_v. In this case, only the underlying chain complex structure is used.
- A horizontal differential d_h is defined. This uses the coproduct structure.
- The final chain complex `cobar` is created with differential $d_v + d_h$.

$$\begin{array}{ccccccc} & & \downarrow & & \downarrow & & \\ \cdots & \to & (\bar{\mathcal{A}}_*^{\otimes p})_q & \xrightarrow{d_h} & (\bar{\mathcal{A}}_*^{\otimes p+1})_q & \to & \cdots \\ & & \downarrow d_v & & \downarrow d_v & & \\ \cdots & \to & (\bar{\mathcal{A}}_*^{\otimes p})_{q-1} & \xrightarrow{d_h} & (\bar{\mathcal{A}}_*^{\otimes p+1})_{q-1} & \to & \cdots \\ & & \downarrow & & \downarrow & & \end{array}$$

Definition of the vertical cobar

To define the vertical cobar chain complex from a chain complex $\mathcal{C}$, the following functions are provided:

`cobar-cmpr` *cmpr* *[Function]*
: From the comparison function *cmpr*, build a comparison function for objects of algebraic loop type.

`cobar-basis` *basis* *[Function]*
: From the function *basis* of a chain complex $\mathcal{C}$, build a basis function for the vertical cobar chain complex defined on $\mathcal{C}$. In dimension 0, there is only one basis element, namely the null algebraic loop. If $\mathcal{C}$ is locally effective, this function returns the symbol `:locally-effective`.

`cobar-intr-vrtc-dffr` *dffr* *[Function]*
: From the `lisp` function *dffr* of a chain complex $\mathcal{C}$, build a lisp function for the differential of the vertical cobar chain complex,

according to the formula

$$d_v(g_1 \otimes \cdots \otimes g_n) = \sum_{i=1}^{n} (-1)^{n-i+1+\sum_{j=1}^{i-1} |g_j|} g_1 \otimes \cdots \otimes g_{i-1} \otimes \mathbf{dg_i} \otimes g_{i+1} \otimes \cdots \otimes g_n,$$

where dg_i is the differential of the generator g_i in the original chain complex (function *dffr*) and $\mid g_i \mid$ is the degree of the generator g_i *in the cobar chain complex*. In general, the differential dg_i is a sum of tensor products. So, applying the distribution law, the right member is a sum of tensor products, represented by a combination.

`vrtc-cobar` *chcm* *[Method]*

Build the vertical cobar chain complex with underlying chain complex *chcm*. This is simply done by the following call to `build-chcm`:

```
(build-chcm
   :cmpr (cobar-cmpr cmpr)
   :basis (cobar-basis basis)
   :bsgn +null-allp+
   :intr-dffr (cobar-intr-vrtc-dffr dffr)
   :strt :gnrt
   :orgn `(vrtc-cobar ,chcm))
```

where, *cmpr*, *basis* and *dffr* are extracted from the chain complex *chcm*. Note that the base generator is the null algebraic loop.

Example

We are going to build a coalgebra which will be reconsidered in a next chapter (Simplicial sets). It is the coalgebra canonicaly associated to the simplicial set $\Delta^{\mathbb{N}}_2$. This simplicial set is the quotient of $\Delta^{\mathbb{N}}$ – the standard simplex spanned by the non–negative integers – by the relation identifying, firstly all the vertices with a unique one and then any 1–simplex of $\Delta^{\mathbb{N}}$ with the unique degeneracy of this unique vertex. In other words, $\Delta^{\mathbb{N}}_2$ is 1–reduced: 1 vertex, no edges.

Our implementation represents any non–degenerate n–simplex as an integer increasing list of length $n+1$. In dimension 0, for every non–negative m and n, the vertices `(m)` and `(n)` are identified; in dimension 1, the list $(m\,n)$ is always an **illegal** 1–simplex. This coalgebra will be located through the symbol `smp-deltab2`. The lisp definition uses the function `build-smst` that will be described in the simplicial set chapter. For the moment, it is sufficient to know, that the function `build-smst` allows us to create a coalgebra suitable for the tests. The definition of $\Delta^{\mathbb{N}}_2$ given here, is only for

pedagogical purpose, an efficient definition using another representation for the simplices is provided in the software Kenzo.

```
(setf smp-deltab2
   (build-smst
        :cmpr #'(lambda(gsm1 gsm2)
                       (if(rest gsm1) (l-cmpr gsm1 gsm2) :equal))
        :basis :locally-effective
        :bspn '(0)
        :face #'(lambda (i dmn gsm)
                        (case dmn
                         (0 (error ``No face in dimension 0''))
                         (1 (error ``No non-degenerate simplex in dimension 1''))
                         (2 (absm 1 '(0)))
                         (otherwise (absm 0 (append (subseq gsm 0 i)
                                                    (subseq gsm (1+ i))))) ))
        :orgn '(simple-deltab2)))
```

The following statements show the identification of vertices and that the boundary of a 2–simplex is null.

```
(cmpr smp-deltab2 '(5) '(0))  ==>

:EQUAL

(? smp-deltab2 2 '(0 1 2))  ==>

-------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

(? smp-deltab2 3 '(0 1 2 3))  ==>

-------------------------------------------------------------------------{CMBN 2}
<-1 * (0 1 2)>
<1 * (0 1 3)>
<-1 * (0 2 3)>
<1 * (1 2 3)>
------------------------------------------------------------------------------
```

Let us build the vertical cobar on the coalgebra smp-deltab2 and apply the comparison function and the differential on some algebraic loops.

```
(setf precobar (vrtc-cobar smp-deltab2))  ==>

[K6 Chain-Complex]

(bsgn precobar)  ==>
```

```
(<<AlLp>>)

(setf loop-1 (allp 2 '(0 1 2 3)))  ==>

<<AlLp[2 (0 1 2 3)]>>

(setf loop-2 (allp '(2 (1 2 3 4))))  ==>

<<AlLp[2 (1 2 3 4)]>>

(cmpr precobar loop-1 loop-2)  ==>

:LESS

(? precobar 2 loop-1)  ==>

------------------------------------------------------------------------{CMBN 1}
<1 * <<AlLp[1 (0 1 2)]>>>
<-1 * <<AlLp[1 (0 1 3)]>>>
<1 * <<AlLp[1 (0 2 3)]>>>
<-1 * <<AlLp[1 (1 2 3)]>>>
--------------------------------------------------------------------------------

(? precobar *)  ==>

------------------------------------------------------------------------{CMBN 0}
--------------------------------------------------------------------------------

(setf loop-3 (allp 3 '(0 1 2 3 4) 3 '(1 3 5 7 9)))  ==>

<<AlLp[3 (0 1 2 3 4)][3 (1 3 5 7 9)]>>

(? precobar 6 loop-3)  ==>

------------------------------------------------------------------------{CMBN 5}
<1 * <<AlLp[2 (0 1 2 3)][3 (1 3 5 7 9)]>>>
<-1 * <<AlLp[2 (0 1 2 4)][3 (1 3 5 7 9)]>>>
<1 * <<AlLp[2 (0 1 3 4)][3 (1 3 5 7 9)]>>>
<-1 * <<AlLp[2 (0 2 3 4)][3 (1 3 5 7 9)]>>>
<1 * <<AlLp[2 (1 2 3 4)][3 (1 3 5 7 9)]>>>
<1 * <<AlLp[3 (0 1 2 3 4)][2 (1 3 5 7)]>>>
<-1 * <<AlLp[3 (0 1 2 3 4)][2 (1 3 5 9)]>>>
<1 * <<AlLp[3 (0 1 2 3 4)][2 (1 3 7 9)]>>>
<-1 * <<AlLp[3 (0 1 2 3 4)][2 (1 5 7 9)]>>>
<1 * <<AlLp[3 (0 1 2 3 4)][2 (3 5 7 9)]>>>
--------------------------------------------------------------------------------
```

```
(? precobar *)  ==>

------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

(setf loop-4 (allp 3 '(0 1 4 5 6) 4 '(2 3 4 5 6 8) 4 '(0 4 5 6 7 8)))  ==>

<<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>

(? precobar 11 loop-4)  ==>

-----------------------------------------------------------------------{CMBN 10}
<-1 * <<AlLp[2 (0 1 4 5)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[2 (0 1 4 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[2 (0 1 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[2 (0 4 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[2 (1 4 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][3 (2 3 4 5 6)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][3 (2 3 4 5 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][3 (2 3 4 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][3 (2 3 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][3 (2 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][3 (3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 4 5 6 7)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 4 5 6 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 4 5 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 4 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (4 5 6 7 8)]>>>
-------------------------------------------------------------------------------

(? precobar *)  ==>

------------------------------------------------------------------------{CMBN 9}
-------------------------------------------------------------------------------
```

Definition of the horizontal differential

So far, the coalgebra structure has not been used but now it will be the main ingredient. If we make the convention that the elements of $(\bar{\mathcal{A}}_*^{\otimes_p})_q$ are of degree $q-p$, we know that exists a canonical "horizontal" differential d_h,

$$d_h : (\bar{\mathcal{A}}_*^{\otimes p})_q \longrightarrow (\bar{\mathcal{A}}_*^{\otimes p+1})_q,$$

$$\begin{aligned} d_h(g_1 \otimes \cdots \otimes g_p) &= (-1)^p \Delta g_1 \otimes g_2 \otimes \cdots \otimes g_p \\ &\quad +(-1)^{p-1} g_1 \otimes \Delta g_2 \otimes \cdots \otimes g_p \\ &\quad \cdots \\ &\quad -g_1 \otimes g_2 \otimes \cdots \otimes g_{p-1} \otimes \Delta g_p, \end{aligned}$$

where Δ is the coproduct of the coalgebra. Two functions are designed for that:

`cobar-intr-hrzn-dffr` *cprd* *[Function]*

From the coproduct morphism *cprd*, build the lisp function with 2 arguments: a degree and an algebraic loop (i.e. a generator) implementing the algorithm for d_h.

`cobar-hrzn-dffr` *clbg* *[Function]*

Build the horizontal differential morphism, using the slot coproduct *cprd* of the coalgebra *clbg* and the previous function. Note that when called, this function creates, if not already created, the vertical cobar chain complex on *clbg*:

```
(build-mrph
  :sorc (vrtc-cobar clgb)
  :trgt (vrtc-cobar clgb)
  :degr -1
  :intr (cobar-intr-hrzn-dffr cprd)
  :strt :gnrt
  :orgn `(cobar-hrzn-dffr ,clgb))
```

Example

We still use our simple coalgebra `smp-deltab2`. The horizontal differential uses the coproduct of the coalgebra. The following examples show the application of the coproduct (note that there are no 1-simplices in the tensor products). We verify also the property of the horizontal differential.

```
(cprd smp-deltab2 2 '(0 1 2))  ==>
```

```
------------------------------------------------------------------------{CMBN 2}
<1 * <TnPr (0) (0 1 2)>>
<1 * <TnPr (0 1 2) (0)>>
------------------------------------------------------------------------------

(cprd smp-deltab2 3 '(1 2 3 4))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr (0) (1 2 3 4)>>
<1 * <TnPr (1 2 3 4) (0)>>
------------------------------------------------------------------------------

(cprd smp-deltab2 4 '(0 1 2 3 4))  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr (0) (0 1 2 3 4)>>
<1 * <TnPr (0 1 2) (2 3 4)>>
<1 * <TnPr (0 1 2 3 4) (0)>>
------------------------------------------------------------------------------

(setf dh-mrph (cobar-hrzn-dffr smp-deltab2))  ==>

[K8 Morphism (degree -1)]

loop-1  ==>

<<AlLp[2 (0 1 2 3)]>>

(? dh-mrph 2 loop-1)  ==>

------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

loop-3  ==>

<<AlLp[3 (0 1 2 3 4)][3 (1 3 5 7 9)]>>

(? dh-mrph 6 loop-3)  ==>

------------------------------------------------------------------------{CMBN 5}
<1 * <<AlLp[1 (0 1 2)][1 (2 3 4)][3 (1 3 5 7 9)]>>>
<-1 * <<AlLp[3 (0 1 2 3 4)][1 (1 3 5)][1 (5 7 9)]>>>
------------------------------------------------------------------------------
```

```
(? dh-mrph *)  ==>

-------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

loop-4  ==>

<<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>

(? dh-mrph 11 loop-4)  ==>

------------------------------------------------------------------------{CMBN 10}
<-1 * <<AlLp[1 (0 1 4)][1 (4 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][1 (2 3 4)][2 (4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][2 (2 3 4 5)][1 (5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][1 (0 4 5)][2 (5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][2 (0 4 5 6)][1 (6 7 8)]>>>
-------------------------------------------------------------------------------

(? dh-mrph *)  ==>

-------------------------------------------------------------------------{CMBN 9}
-------------------------------------------------------------------------------
```

Final definition of the cobar of a coalgebra

To define completely the cobar chain complex, we have to provide the lisp function for the differential $d_v + d_h$ and, once created the vertical cobar chain complex, to call the building function `build-chcm` with adequate arguments.

`cobar-intr-dffr` *vrtc-dffr hrzn-dffr* *[Function]*
: Define the lisp function for the differential $d_v + d_h$ from the two morphisms *vrtc-dffr* and *hrzn-dffr*. For efficiency reasons, the implementor has chosen to define this new function rather than to use simply the addition of two morphisms.

`cobar` *coalgebra* *[Method]*
: Build first the vertical cobar chain complex on the coalgebra. Then return a chain complex with the same slots as the vertical cobar chain complex but with new `:intr-dffr` and `:orgn` slots, as shown in the following definition:

```
(defmethod COBAR ((coalgebra coalgebra))
   (let ((vrtc-cobar (vrtc-cobar coalgebra))
         (cobar-hrzn-dffr (cobar-hrzn-dffr coalgebra)))
   (declare (type chain-complex vrtc-coabr hrzn-cobar))
   (the chain-complex
       (build-chcm
          :cmpr (cmpr vrtc-cobar)
          :basis (basis vrtc-cobar)
          :bsgn +null-allp+
          :intr-dffr (cobar-intr-dffr (dffr vrtc-cobar)
                                      cobar-hrzn-dffr)
          :strt :gnrt
          :orgn `(add ,vrtc-cobar ,cobar-hrzn-dffr)))))
```

Example

We may now build the cobar of `smp-deltab2` and test the differential.

```
(setf cobar-deltab2 (cobar smp-deltab2))  ==>

[K9 Chain-Complex]

loop-3  ==>

<<AlLp[3 (0 1 2 3 4)][3 (1 3 5 7 9)]>>


(? cobar-deltab2 6 loop-3)  ==>

------------------------------------------------------------------------{CMBN 5}
<1 * <<AlLp[2 (0 1 2 3)][3 (1 3 5 7 9)]>>>
<-1 * <<AlLp[2 (0 1 2 4)][3 (1 3 5 7 9)]>>>
<1 * <<AlLp[2 (0 1 3 4)][3 (1 3 5 7 9)]>>>
<-1 * <<AlLp[2 (0 2 3 4)][3 (1 3 5 7 9)]>>>
<1 * <<AlLp[2 (1 2 3 4)][3 (1 3 5 7 9)]>>>
<1 * <<AlLp[3 (0 1 2 3 4)][2 (1 3 5 7)]>>>
<-1 * <<AlLp[3 (0 1 2 3 4)][2 (1 3 5 9)]>>>
<1 * <<AlLp[3 (0 1 2 3 4)][2 (1 3 7 9)]>>>
<-1 * <<AlLp[3 (0 1 2 3 4)][2 (1 5 7 9)]>>>
<1 * <<AlLp[3 (0 1 2 3 4)][2 (3 5 7 9)]>>>
<1 * <<AlLp[1 (0 1 2)][1 (2 3 4)][3 (1 3 5 7 9)]>>>
<-1 * <<AlLp[3 (0 1 2 3 4)][1 (1 3 5)][1 (5 7 9)]>>>
------------------------------------------------------------------------------

(? cobar-deltab2 *)  ==>

------------------------------------------------------------------------{CMBN 4}
------------------------------------------------------------------------------
```

```
loop-4  ==>

<<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>

(? cobar-deltab2 11 loop-4)  ==>

-------------------------------------------------------------------------{CMBN 10}
<-1 * <<AlLp[2 (0 1 4 5)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[2 (0 1 4 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[2 (0 1 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[2 (0 4 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[2 (1 4 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][3 (2 3 4 5 6)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][3 (2 3 4 5 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][3 (2 3 4 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][3 (2 3 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][3 (2 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][3 (3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 4 5 6 7)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 4 5 6 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 4 5 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 4 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (0 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][3 (4 5 6 7 8)]>>>
<-1 * <<AlLp[1 (0 1 4)][1 (4 5 6)][4 (2 3 4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][1 (2 3 4)][2 (4 5 6 8)][4 (0 4 5 6 7 8)]>>>
<1 * <<AlLp[3 (0 1 4 5 6)][2 (2 3 4 5)][1 (5 6 8)][4 (0 4 5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][1 (0 4 5)][2 (5 6 7 8)]>>>
<-1 * <<AlLp[3 (0 1 4 5 6)][4 (2 3 4 5 6 8)][2 (0 4 5 6)][1 (6 7 8)]>>>
-------------------------------------------------------------------------------

(? cobar-deltab2 *)  ==>

-------------------------------------------------------------------------{CMBN 9}
-------------------------------------------------------------------------------
```

5.4 Other functions for the cobar construction

The vertical cobar construction is *natural* and therefore defines a functor. Consequently, the `Kenzo` program contains the following methods.

`vrtc-cobar` *mrph* *[Method]*
: From the morphism *mrph*, build a new morphism between the vertical-cobar of the source of *mrph* and the vertical cobar of the target of *mrph*, the morphism itself being induced in a natural way by the underlying morphism.

`vrtc-cobar` *rdct* *[Method]*
: From the reduction *rdct*, build a new reduction, by applying the vertical cobar method to the functions f and g. The new homotopy cannot be obtained so simply, being a function of the f, g and h of *rdct*. Nevertheless, the source and target of the new homotopy are the vertical cobar of the underlying homotopy.

`vrtc-cobar` *hmeq* *[Method]*
: Build a homotopy equivalence by applying the vertical cobar construction to the left and right reductions of the homotopy equivalence *hmeq*.

`cobar` *hmeq* *[Method]*
: Build a homotopy equivalence where the cobar construction is applied to the underlying homotopy equivalence in the following way. First this construction is limited to the case where the left bottom chain complex of *hmeq* is a **coalgebra**. Then the vertical cobar construction is applied to *hmeq* and the horizontal differential of the left bottom chain complex is propagated upon the new homotopy equivalence with the help of the method `add`. In other words, starting from the left bottom chain complex, firstly the easy pertubation lemma is applied to obtain a new differential on the top chain complex; then the basic perturbation lemma is applied to the right reduction. If *hmeq* is a trivial homotopy equivalence, this function returns a trivial one, built on the cobar of the left bottom chain complex of *hmeq*.

Lisp files concerned in this chapter

`coalgebras.lisp`, `cobar.lisp`.
[`classes.lisp`, `macros.lisp`, `various.lisp`].

Chapter 6

Algebras and Bars

6.1 Algebra

An algebra is a pair (A, ϖ) where

1. A is a chain complex.
2. ϖ (the *product*) is a chain complex morphism

$$\varpi : A \otimes A \to A.$$

These components must satisfy the usual structural properties of the algebras.

6.1.1 Implementation of an algebra

An algebra is represented as an instance of the CLOS class `ALGEBRA`, subclass of the `CHAIN COMPLEX` class. The definition of this new class is simply:

```
(DEFCLASS ALGEBRA (chain-complex)
    ((aprd :type morphism :initarg :aprd :reader aprd1)))   ;; product
```

So, this new class inherits the slots of the `CHAIN COMPLEX` class and has a slot of its own, namely `aprd`, an object of type `MORPHISM` representing the product. Like in a coalgebra, the user will note the important following fact that, **once an algebra has been defined, one may use on it any function or method applicable to a chain complex**.

6.1.2 The function build-algb

To facilitate the construction of instances of the `ALGEBRA` class, the software **Kenzo** provides the function `build-algb`:

`build-algb` `:cmpr` *cmpr* `:basis` *basis* `:bsgn` *bsgn* `:intr-dffr` *intr-dffr*
`:dffr-strt` *dffr-strt* `:intr-aprd` *intr-aprd*
`:aprd-strt` *aprd-strt* `:orgn` *orgn*

defined with keyword parameters. The returned value is an instance of type `ALGEBRA`. The keyword arguments are:

- *cmpr*, a comparison function for generators.
- *basis*, the basis function for the underlying chain complex.
- *bsgn*, the base point, a generator of any type.
- *intr-dffr*, the differential lisp function for the chain complex.
- *dffr-strt*, the strategy (`:gnrt` or `:cmbn`) for the differential.
- *intr-aprd*, the lisp function for the product of the algebra.
- *aprd-strt*, the strategy (`:gnrt` or `:cmbn`) for the coproduct.
- *orgn*, an adequate comment list.

The function `build-algb` calls the function `build-chcm` and with the help of the function `build-mrph` builds the product morphism of degree 0 to settle the slot `aprd` of the instance. According to the definition, the source slot of the product morphism is the tensor product of this chain complex with itself and the target slot is the underlying chain complex. An identification number n (slot `idnm`) is assigned to this Kenzo object and the `ALGEBRA` instance is pushed onto the list `*algb-list*`. The associated printing method prints a string like `[K`n `Algebra]`.

6.1.3 Miscellaneous functions and macros for algebras

`cat-init` *[Function]*
Clear among others, the list `*algb-list*`, list of user created algebras and reset the global counter to 1.

`algb` *n* *[Function]*
Retrieve in the list `*algb-list*` the algebra instance whose identification is *n*. If it does not exist, return `NIL`.

`aprd` *algb* `&rest` *[Macro]*
Versatile macro relative to the product of an algebra. The first argument is an `ALGEBRA` instance. With only one argument, this macro returns the product morphism of the algebra. With more arguments, it applies the product morphism on a combination (`(aprd` *algb cmbn*`)`) or on a pair *degree, tensor-product* (`(aprd` *algb degr tensor-product*`)`).

`change-chcm-to-algb` *chcm* `:intr-aprd` *intr-aprd* `:aprd-strt` *aprd-strt* `:orgn` *orgn* *[Function]*
Build an algebra instance from the already created chain complex *chcm*. The user must give via key parameters a lisp function for the product, its strategy and a comment list.

6.2 Hopf Algebra

In **Kenzo**, a Hopf Algebra is an instance of the class `HOPF-ALGEBRA`, the definition of which being simply:

```
(DEFCLASS HOPF-ALGEBRA (coalgebra algebra)
  ())
```

So, this class (multi–) inherits the slots of the `COALGEBRA` and `ALGEBRA` classes.

The associated printing method prints a string like `[K`*n* `Hopf-Algebra]`, and the function `hopf` may be used to retrieve the Hopf instance, in the list `*hopf-list*`.

6.2.1 Example of algebra

We shall see later important examples of algebras. Let us content ourself for the moment to define the trivial algebra on a given chain complex *having only one generator in degree* 0. If this chain complex *chcm* already exists, we use simply the function `change-chcm-to-algb` and we have to provide for the keyword `:intr-aprd`, a lisp function for the product morphism. We recall

that the argument of the product in the algebra is a tensor product. This function must obey the rule of the multiplication by the unit and otherwise will return a null combination of degree the sum of the two degrees in the tensor product.

```
(defun trivial-algebra (chcm)
   (change-chcm-to-algb
      chcm
      :intr-aprd #'(lambda (degr tnpr)
                     (with-tnpr (degr1 gnrt1 degr2 gnrt2) tnpr
                         (if (zerop degr1)
                             (cmbn degr2 1 gnrt2)
                             (if (zerop degr2)
                                 (cmbn degr1 1 gnrt1)
                                 (zero-cmbn (+ degr1 degr2))))))
      :aprd-strt :gnrt
      :orgn `(trivial-algebra ,chcm)))
```

A good example of a chain complex to be given as argument to the previous function, is the simplicial set `smp-deltab2` that we used as simple example in the coalgebra chapter:

```
(setf smp-deltab2
   (build-smst
        :cmpr #'(lambda(gsm1 gsm2)
                       (if(rest gsm1) (l-cmpr gsm1 gsm2) :equal))
        :basis :locally-effective
        :bspn '(0)
        :face #'(lambda (i dmn gsm)
                        (case dmn
                         (0 (error "No face in dimension 0"))
                         (1 (error "No non-degenerate simplex in dimension 1"))
                         (2 (absm 1 '(0)))
                         (otherwise (absm 0 (append (subseq gsm 0 i)
                                                    (subseq gsm (1+ i))))) ))
        :orgn '(simple-deltab2)))

[K1 Simplicial-Set]

(trivial-algebra smp-deltab2)  ==>

[K1 Algebra]
```

Note that now `smp-deltab2` is of type `ALGEBRA` and has kept its Kenzo numbering.

```
smp-deltab2  ==>

[K1 Algebra]
```

```
(inspect smp-deltab2)  ==>

ALGEBRA @ #x3065ea = [K1 Algebra]
   0 Class --------> #<STANDARD-CLASS ALGEBRA>
   1 ORGN ---------> (SIMPLE-DELTAB2), a proper list with 1 element
   2 IDNM ---------> fixnum 1 [#x00000004]
   3 EFHM ---------> The symbol :--UNBOUND--
   4 GRMD ---------> [K1 Algebra]
   5 DFFR ---------> [K2 Morphism (degree -1)]
   6 BSGN ---------> (0), a proper list with 1 element
   7 BASIS --------> The symbol :LOCALLY-EFFECTIVE
   8 CMPR ---------> #<Interpreted Function (unnamed) @ #x306652>
   9 APRD ---------> [K6 Morphism (degree 0)]
```

Let us test now the product in this trivial algebra:

```
(aprd smp-deltab2 3 (tnpr 0 '(0) 3 '(1 2 3 4)))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * (1 2 3 4)>
-------------------------------------------------------------------------------

(aprd smp-deltab2 3 (tnpr  3 '(1 2 3 4) 0 '(0)))

------------------------------------------------------------------------{CMBN 3}
<1 * (1 2 3 4)>
-------------------------------------------------------------------------------

(aprd smp-deltab2 5 (tnpr 2 '(0 1 2) 3 '(6 7 8 9)))  ==>

------------------------------------------------------------------------{CMBN 5}
-------------------------------------------------------------------------------
```

6.3 The Bar construction

Let $\mathcal{A}$ be an associative algebra, assumed connected and $\mathcal{A}_0 \simeq \mathbb{Z}$. Furthermore let us suppose that $\mathcal{A}$ is a free $\mathbb{Z}$–module. Then it is possible to define a chain complex, $Bar^{\mathcal{A}}(\mathbb{Z},\mathbb{Z})$ – simply denoted here by $Bar(\mathcal{A})$ – whose n-th component is the free $\mathbb{Z}$–module generated by the "bars":

$$[Bar(\mathcal{A})]_n = \{[g_1 \mid g_2 \mid \ldots \mid g_k]\}, \quad \sum_{j=1}^{k}[deg(g_j)+1] = n.$$

The object noted $[g_1 \mid g_2 \mid \ldots \mid g_k]$ with $\sum_{j=1}^{k}[deg(g_j)+1] = n$, is traditionally called a *bar* and is in fact, an element of the n–th iterated suspension

of $\mathcal{A}^{\otimes n}$. The integer n is the *total degree*. The structure of the bar of the algebra $\mathcal{A}$ is recalled in the following figure, where $\bar{\mathcal{A}}$ is $\mathcal{A}$ without its component of degree 0. In the vertical sense, we have the *tensorial degree*, whereas in the horizontal one, we have the *simplicial degree*. The *total degree* n is the sum of both degrees.

$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
0	$\bar{\mathcal{A}}_3$	$(\bar{\mathcal{A}} \otimes \bar{\mathcal{A}})_3$	$(\bar{\mathcal{A}} \otimes \bar{\mathcal{A}} \otimes \bar{\mathcal{A}})_3$	$\cdots$
0	$\bar{\mathcal{A}}_2$	$(\bar{\mathcal{A}} \otimes \bar{\mathcal{A}})_2$	0	$\cdots$
0	$\bar{\mathcal{A}}_1$	0	0	$\cdots$
$\mathbb{Z}$	0	0	0	$\cdots$
$\bar{\mathcal{A}}^{\otimes 0}$	$\bar{\mathcal{A}}^{\otimes 1}$	$\bar{\mathcal{A}}^{\otimes 2}$	$\bar{\mathcal{A}}^{\otimes 3}$	$\cdots$

6.3.1 Representation of a bar object

An elementary bar object – not to be confused with the chain complex $Bar(\mathcal{A})$ – is represented in `Kenzo` by a lisp object of the form:

`(:ABAR` $(i_1 \,.\, a_1)$ $\ldots$ $(i_k \,.\, a_k)$`)`

where the i_j are the degrees **in the bar chain complex** of the generators a_j. In the original algebra $\mathcal{A}$, the generators a_j had the degree $i_j - 1$. The corresponding type is `ABAR`. The function building such an object is also called `abar`

`abar` *degr1 gnr1 degr2 gnr2 ... degrk gnrk* *[Function]*
: Construct an elementary bar object, i.e. a "suspended tensorial product" of degree $\sum degr_i$. The sequence of pairs $\{degr_i, gnr_i\}$ has an undefinite length and may be void. In this case, the bar is the null bar object, also located in the system through the constant `+null-abar+`. The function `abar` accepts also as unique argument a list of the form (*degr1 gnr1 degr2 gnr2 ... degrk gnrk*).

`abar-p` *object* *[Function]*
: Test if *object* is an elementary bar object.

The associated printing method prints the object under the form:

`<<Abar [`i_1 a_1`] ... [`i_k a_k`] >>`

Examples

These simple commands show the two different ways to create bar objects from elements of an algebra (or a chain complex).

```
(abar 2 'a 3 'b 5 'c)  ==>

<<Abar[2 A][3 B][5 C]>>

(abar '(2 a 3 b 5 c))  ==>

<<Abar[2 A][3 B][5 C]>>

(abar)  ==>

<<Abar>>
```

6.3.2 Definition of the chain complex Bar

The definition of the differential in the bar chain complex, will be done in three steps. This is better understood, if we consider the following diagram.

$$\begin{array}{ccccccc}
 & & \downarrow & & \downarrow & & \\
\cdots & \leftarrow & (\bar{\mathcal{A}}_*^{\otimes p})_q & \xleftarrow{d_h} & (\bar{\mathcal{A}}_*^{\otimes p+1})_q & \leftarrow & \cdots \\
 & & \downarrow d_v & & \downarrow d_v & & \\
\cdots & \leftarrow & (\bar{\mathcal{A}}_*^{\otimes p})_{q-1} & \xleftarrow{d_h} & (\bar{\mathcal{A}}_*^{\otimes p+1})_{q-1} & \leftarrow & \cdots \\
 & & \downarrow & & \downarrow & &
\end{array}$$

- A chain complex called `vertical-bar` is defined with the *vertical differential* d_v. In this case, only the underlying chain complex structure is used.
- A horizontal differential d_h is defined. This uses the product structure.
- The final chain complex `bar` is created with differential $d_v + d_h$.

Definition of the vertical bar

To define the vertical bar chain complex from the chain complex $\mathcal{A}$, the following functions are provided:

`bar-cmpr` *cmpr* *[Function]*
: From the comparison function *cmpr*, build a comparison function for objects of type `abar`.

`bar-basis` *basis* *[Function]*
: From the function *basis* of a chain complex $\mathcal{A}$, build a basis function for the vertical bar chain complex defined on $\mathcal{A}$. In dimension 0, there is only one basis element, namely the null abar object. If $\mathcal{A}$ is locally effective, this function returns the symbol `:locally-effective`.

`bar-intr-vrtc-dffr` *dffr* *[Function]*
: From the `lisp` function *dffr* of a chain complex $\mathcal{A}$, build a lisp function for the differential of the vertical bar chain complex, according to the formula:

$$d_v[a_1 \mid \cdots \mid a_n] = -\sum_{i=1}^{n} (-1)^{\sum_{j=1}^{i-1} |a_j|} [a_1 \mid \cdots \mid a_{i-1} \mid \mathbf{da_i} \mid a_{i+1} \mid \cdots \mid a_n],$$

: where da_i is the differential of the generator a_i in the original chain complex (function *dffr*) and $\mid a_i \mid$ is the degree of the generator a_i *in the bar chain complex*. In general, the differential da_i is a sum of objects. So, applying the distributive law, the right member is a sum of bar objects, represented by a combination.

`vrtc-bar` *chcm* *[Method]*
: Build the vertical bar chain complex from the underlying chain complex *chcm*. This is simply done by the call to `build-chcm`:

```
(build-chcm
   :cmpr  (bar-cmpr cmpr)
   :basis (bar-basis basis)
   :bsgn +null-abar+
   :intr-dffr (bar-intr-vrtc-dffr dffr)
   :strt :gnrt
   :orgn `(vrtc-bar ,chcm))
```

: where, *cmpr*, *basis* and *dffr* are extracted from the chain complex *chcm*. Note that the base generator is the null bar object.

Examples

First, let us test the functions `bar-cmpr` and `bar-basis`. With the first one, applied to the elementary comparison function `s-cmpr`, we create a comparison function suitable for algebras where the generators are symbols.

```
(setf cmpr-for-bar (bar-cmpr #'s-cmpr))

(funcall cmpr-for-bar (abar)(abar))  ==>

:EQUAL

(funcall cmpr-for-bar (abar 3 'a 7 'q) (abar))  ==>

:GREATER

(funcall cmpr-for-bar (abar '(1 x 2 y 3 z) )(abar 1 'x 2 'y 3 'z))  ==>

:EQUAL
```

Let us suppose now that the function `simple-basis` is the basis function for a certain algebra (chain complex): in degree i, the basis is only the list (a_i).

```
 (setf simple-basis #'(lambda(degr)
    (list (intern(format nil "A~D" degr)))))

(funcall simple-basis 5)  ==>

(A5)
```

The function `bar-basis` creates the function for the basis of the corresponding bar chain complex. Note that when this new function is applied, it returns a priori the null abar in dimension 0 and a null basis in dimension 1. Note also that the basis elements of the underlying algebra (the a_i's) are suspended.

```
(setf basis-for-bar (bar-basis simple-basis))

(dotimes (i 7)
   (format t "~%~D ~A" i (funcall basis-for-bar i)))  ==>

0 (<<Abar>>)
1 NIL
2 (<<Abar[2 A1]>>)
3 (<<Abar[3 A2]>>)
4 (<<Abar[4 A3]>> <<Abar[2 A1][2 A1]>>)
5 (<<Abar[5 A4]>> <<Abar[2 A1][3 A2]>> <<Abar[3 A2][2 A1]>>)
```

```
6 (<<Abar[6 A5]>> <<Abar[2 A1][4 A3]>> <<Abar[3 A2][3 A2]>>
   <<Abar[4 A3][2 A1]>> <<Abar[2 A1][2 A1][2 A1]>>)
```

To test the vertical differential, let us take the simplicial set `smp-deltab2` which is also a chain complex. We may use it for testing the function `vrtc-bar` which defines a chain complex independently of the product of the algebra. We recall that, in dimension n, the elements of the chain complex `smp-deltab2` are represented by lists of increasing $n+1$ integers. So, in the abar objects, the degree of such elements are $n+1$.

```
(setf vrt-bar (vrtc-bar smp-deltab2))  ==>

[K6 Chain-Complex]

(? vrt-bar 9 (abar 4 '(0 1 2 3) 5 '(4 5 6 7 8)))  ==>

-------------------------------------------------------------------------{CMBN 8}
<-1 * <<Abar[3 (0 1 2)][5 (4 5 6 7 8)]>>>
<1 * <<Abar[3 (0 1 3)][5 (4 5 6 7 8)]>>>
<-1 * <<Abar[3 (0 2 3)][5 (4 5 6 7 8)]>>>
<1 * <<Abar[3 (1 2 3)][5 (4 5 6 7 8)]>>>
<-1 * <<Abar[4 (0 1 2 3)][4 (4 5 6 7)]>>>
<1 * <<Abar[4 (0 1 2 3)][4 (4 5 6 8)]>>>
<-1 * <<Abar[4 (0 1 2 3)][4 (4 5 7 8)]>>>
<1 * <<Abar[4 (0 1 2 3)][4 (4 6 7 8)]>>>
<-1 * <<Abar[4 (0 1 2 3)][4 (5 6 7 8)]>>>
-------------------------------------------------------------------------------

(? vrt-bar *)  ==>

-------------------------------------------------------------------------{CMBN 7}
-------------------------------------------------------------------------------
```

Definition of the horizontal differential

So far, the algebra structure has not been used but now it will be the main ingredient. The canonical "horizontal" differential d_h is defined as follows:

$$d_h : (\bar{\mathcal{A}}_*^{\otimes p})_q \longrightarrow (\bar{\mathcal{A}}_*^{\otimes p-1})_q,$$

$$d_h[a_1 \mid \cdots \mid a_k] = \sum_{i=2}^{n} (-1)^{\sum_{j=1}^{i-1} |a_j|} [a_1 \mid \cdots \mid a_{i-2} \mid \mathbf{a_{i-1}\, a_i} \mid a_{i+1} \mid \cdots \mid a_k]$$

where the product in the algebra is noted simply by the concatenation and $\mid a_i \mid$ is the degree of a_i in $Bar(\mathcal{A})$. In `Kenzo`, two functions are designed for building the horizontal differential:

`bar-intr-hrzn-dffr` *aprd* *[Function]*

From the product morphism *aprd*, build the lisp function with 2 arguments: a degree and an `abar` object (i.e. a generator) realizing the algorithm for d_h.

`bar-hrzn-dffr` *algb* *[Function]*

Build the horizontal differential morphism, using the slot product *aprd* of the algebra *algb* and the previous function. Note that when called, this function creates, if not already created, the vertical bar chain complex on *algb*:

```
(build-mrph
  :sorc (vrtc-bar algb)
  :trgt (vrtc-bar algb)
  :degr -1
  :intr (bar-intr-hrzn-dffr aprd)
  :strt :gnrt
  :orgn `(bar-hrzn-dffr ,algb))
```

Final definition of the bar of an algebra

To define completely the bar chain complex, we have to provide the lisp function for the differential $d_v + d_h$ and, once created the vertical bar chain complex, we have to call the building function `build-chcm` with adequate arguments.

`bar-intr-dffr` *vrtc-dffr hrzn-dffr* *[Function]*

Define the lisp function for the differential $d_v + d_h$ from the two morphisms *vrtc-dffr* and *hrzn-dffr*. For efficiency reasons, the implementor has chosen to define this new function rather than to use simply the addition of two morphisms.

`bar` *algebra* *[Method]*

Build first the vertical bar chain complex on the algebra. Then return a chain complex with the same slots as the vertical bar chain complex but with new `:intr-dffr` and `:orgn` slots, as shown in the following definition:

```
(defmethod BAR ((algebra algebra))
   (let ((vrtc-bar (vrtc-bar algebra))
         (bar-hrzn-dffr (bar-hrzn-dffr algebra)))
   (declare (type chain-complex vrtc-abr hrzn-bar))
   (the chain-complex
       (build-chcm
          :cmpr (cmpr vrtc-bar)
```

```
:basis (basis vrtc-bar)
:bsgn +null-abar+
:intr-dffr (bar-intr-dffr (dffr vrtc-bar)
                          bar-hrzn-dffr)
:strt :gnrt
:orgn `(add ,vrtc-bar ,bar-hrzn-dffr)))))
```

We postpone the examples for the bar of an algebra, up to the chapter on simplicial groups where we shall build interesting examples of algebras.

6.4 Other functions for the bar construction

The vertical bar construction is *natural* and therefore defines a functor. Consequently, the `Kenzo` program contains the following methods.

`vrtc-bar` *mrph* *[Method]*
: From the morphism *mrph*, build a new morphism between the vertical-bar of the source of *mrph* and the vertical bar of the target of *mrph*, the morphism itself being induced in a natural way by the underlying morphism.

`vrtc-bar` *rdct* *[Method]*
: From the reduction *rdct*, build a new reduction, by applying the vertical bar method to the functions f and g. The new homotopy cannot be obtained so simply, being a function of the f, g and h of *rdct*. Nevertheless, the source and target of the new homotopy are the vertical bar of the underlying homotopy.

`vrtc-bar` *hmeq* *[Method]*
: Build a homotopy equivalence by applying the vertical bar construction to the left and right reductions of the homotopy equivalence *hmeq*.

`bar` *hmeq* *[Method]*
: Build a homotopy equivalence where the bar construction is applied to the underlying homotopy equivalence in the following way. First this construction is limited to the case where the left bottom chain complex of *hmeq* is an **algebra**. Then the vertical bar construction is applied to *hmeq* and the horizontal differential of the left bottom chain complex is propagated upon the new homotopy equivalence with the help of the method `add`. In other words, starting from the left bottom chain complex, firstly the easy pertubation lemma is applied to obtain a new differential on the top chain complex; then the basic perturbation lemma is applied to the right reduction. If *hmeq* is a trivial homotopy equivalence, this function returns a trivial one, built on the bar of the left bottom chain complex of *hmeq*.

Lisp files concerned in this chapter

`algebras.lisp`, `bar.lisp`.
[`classes.lisp`, `macros.lisp`, `various.lisp`].

Chapter 7

Simplicial Sets

7.1 Introduction

A *simplicial set*[1] K is a union $K = \bigcup_{q \geq 0} K^q$, where the K^q are disjoints sets, together with functions

$$\partial_i^q : K^q \longrightarrow K^{q-1}, \quad q > 0, \quad i = 0, \ldots, q,$$
$$\eta_i^q : K^q \longrightarrow K^{q+1}, \quad q \geq 0, \quad i = 0, \ldots, q,$$

subject to the relations

$$\begin{aligned}
\partial_i^{q-1}\partial_j^q &= \partial_{j-1}^{q-1}\partial_i^q, & i < j \\
\eta_i^{q+1}\eta_j^q &= \eta_j^{q+1}\eta_{i-1}^q, & i > j \\
\partial_i^{q+1}\eta_j^q &= \eta_{j-1}^{q-1}\partial_i^q, & i < j \\
\partial_i^{q+1}\eta_i^q &= \partial_{i+1}^{q+1}\eta_i^q & Identity, \\
\partial_i^{q+1}\eta_j^q &= \eta_j^{q-1}\partial_{i-1}^q, & i > j+1.
\end{aligned}$$

An element of K^q is an (*abstract*) q–simplex of K and the functions ∂ and η are respectively the *face operators* and the *degeneracy operators*. Formally,

[1] **Hilton & Wylie**, *Homology theory*, Cambridge University Press, 1967

their action on a simplex is very simple. For instance, if a q–simplex is an ordered set of vertices, like $\{v_0, v_1, \ldots, v_i, \ldots, v_q\}$, the rules are the following (we omit the indication of the dimension of the simplex):

$$\begin{aligned} \partial_i\{v_0, v_1, \ldots, v_i, \ldots, v_q\} &= \{v_0, v_1, \ldots, \mathbf{v_{i-1}}, \mathbf{v_{i+1}}, \ldots, v_q\}, \\ \eta_i\{v_0, v_1, \ldots, v_i, \ldots, v_q\} &= \{v_0, v_1, \ldots, \mathbf{v_i}, \mathbf{v_i}, \ldots, v_q\}. \end{aligned}$$

The operators ∂_i^q will be used hereafter to define the boundary operator d^q for the q–component of the associated chain complex:

$$d^q = \sum_{i=0}^{q} (-1)^i \partial_i^q.$$

The image of a simplex under some η is called *degenerate*, because it is not directly implied in the realization of the simplicial set.

Example

Let us take the small example *diabolo* from the chapter 1. For a better understanding, it is convenient to use a list representation for the simplices. So the set K^0 of 0–simplices (vertices) is:

$$\{(0), (1), \ldots (5)\}.$$

The set K^1 of 1–simplices contains the set of regular simplices:

$$\{(0\ 1), (0\ 2), (1\ 2), (2\ 3), (3\ 4), (3\ 5), (4\ 5)\},$$

but also the degenerate 1– simplices:

$$\{(0\ 0), (1\ 1), (2\ 2), \ldots, (5\ 5)\}.$$

The set K^2 of 2–simplices contains the regular singleton:

$$\{(3\ 4\ 5)\}$$

but also the degenerate 2–simplices:

$$\{(0\ 0\ 0), \ldots, (5\ 5\ 5), (0\ 0\ 1), (0\ 0\ 2), (1\ 1\ 2), \ldots, (4\ 5\ 5)\}.$$

The sets K^q $(q > 2)$ contain only degenerate simplices. With this naming of the elements of a q–simplex, the action of ∂ and η is now clear. For instance

$$\partial_0(0\ 1) = (1),\ \partial_1(0\ 1) = (0),$$

$$\eta_0(0\ 1) = (0\ 0\ 1),\ \eta_1(1\ 2) = (1\ 2\ 2),\ \eta_2(3\ 4\ 5) = (3\ 4\ 5\ 5).$$

By the way, we see that a simplex is non-degenerate if the list is strictly increasing.

7.2 Notion of abstract simplex in this Lisp implementation

In this implementation, on account of the essential following property: *any simplex may be expressed in an unique way as a possibly iterated degeneracy of a non–degenerate simplex*, we call an **abstract simplex**, (in short **absm**), a pair consisting of:

- a (possibly iterated) degeneracy operator,
- a "geometric" simplex, i.e. a non–degenerate simplex.

For instance, if σ is a non–degenerate simplex, and σ' is the degenerate simplex $\eta_3\eta_1\sigma$, the corresponding abstract simplices are respectively $\{\emptyset, \sigma\}$ and $\{\eta_3\eta_1, \sigma\}$.
An abstract simplex is represented internally in the system by the following lisp object:

`(:absm` *dgop* `.` *gmsm*`)`

where,

1. *dgop* is a non-negative integer coding a *strictly decreasing integer list*. This strictly decreasing list of integers represents a sequence of η operators and is coded as a **unique** integer with the following convention: a number i in the sequence is the $i+1$–th binary bit of a machine word. The null list is coded as the number 0 whereas the list `(0)`, i.e. the η_0 operator, is coded as the integer `1`, etc. Two functions are provided to help the user for the transformation in both directions.
2. *gmsm* is a geometric simplex, i.e. any kind lisp object modelizing a non–degenerate simplex. For practical reason, a type `GMSM`, (any kind of lisp object), has been defined.

The simplest constructor is the macro `(absm (dgop-ext-int` *dgop-list*`)` *gmsm*`)`, where *dgop-list* is a strictly decreasing integer list. A type `ABSM` has been defined. The associated printing method prints such an object under the form:

`<AbSm` *ext-dgop gmsm*`>`

where *ext-dgop* shows clearly the sequence of the operators η_i (see the examples). The accessor functions for the components are the macros `dgop` and `gmsm`.

7.2.1 Simple functions for abstract simplices

`dgop-ext-int` *ext-dgop* *[Function]*
: Code on an integer the valid list representing the degeneracy operator *ext-dgop*.

`dgop-int-ext` *dgop* *[Function]*
: Give the list representing a degeneracy operator from the integer *dgop*.

`absm` *dgop gmsm* *[Macro]*
: Create an abstract simplex, using directly the coded representation, *dgop*, of the degeneracy operator.

`dgop` *absm* *[Macro]*
: Get the integer code of the internal representation of the degeneracy operator of the abstract simplex *absm*.

`gmsm` *absm* *[Macro]*
: Get the basic non-degenerate simplex part of the abstract simplex `absm`.

`absm-p` *object* *[Predicate]*
: Test if *object* is an abstract simplex.

Examples

Let us suppose that `:sigma` "is" a geometric simplex, we may construct the abstract simplices of the beginning of the section.

```
(dgop-ext-int '(0) )  ==>

1

(dgop-ext-int '(4 0) )  ==>

17

(dgop-int-ext 63)  ==>

(5 4 3 2 1 0)

(dgop-ext-int '(2 2))  ==>

Error: In DGOP-EXT-INT, the external dgop (2 2) is not decreasing.

(setf asigma (absm 0 :sigma)) ==>

<AbSm - SIGMA>
```

```
(setf asigma-prime (absm (dgop-ext-int '(3 1)) :sigma)) ==>

<AbSm 3-1 SIGMA>

(absm-p asigma-prime)  ==>

T
```

The following command gives the sixth degeneracy of the vertex 0 represented as `(0)`.

```
(setf deg-6 (absm (dgop-ext-int '(5 4 3 2 1 0)) '(0)))  ==>

<AbSm 5-4-3-2-1-0 (0)>

(dgop deg-6)  ==>

63

(gmsm deg-6)  ==>

(0)
```

7.3 Representation of a simplicial set

A simplicial set is implemented as an instance of the class `SIMPLICIAL-SET`, subclass of the class `COALGEBRA`.

```
(DEFCLASS SIMPLICIAL-SET (coalgebra)
    ((face :type face :initarg :face :reader face1)))
```

This class inherits also from the class `CHAIN-COMPLEX` and has one slot of its own:

`face`, a lisp function computing any face of any geometric simplex of the simplicial set. This is a function with 3 arguments: a face index (a non–negative integer, `indx`), a simplex dimension (a positive integer, `dmns`) and a geometric simplex (`gmsm`), the *indx*–th face of which is looked for. Whatever the face simplex is, degenerate or not, the simplex returned by this function must be an **abstract** simplex, i.e. an `ABSM` object.

A printing method has been associated to the class `SIMPLICIAL-SET` and the external representation of an instance is a string like `[K`*n* `Simplicial-Set]`, where *n* is the number plate of the Kenzo object.

7.4 The function build-smst

To facilitate the construction of instances of the class `SIMPLICIAL-SET`, the software `Kenzo` provides the function `build-smst` defined with keyword parameters:

`build-smst`
`:cmpr` *cmpr* `basis` *basis* `:bspn` *bspn* `:face` *face*
`:face*` *face** `:intr-bndr` *intr-bndr* `:bndr-strt` *bndr-strt*
`:intr-dgnl` *intr-dgnl* `:dgnl-strt` *dgnl-strt* `:orgn` *orgn*

The returned value is an object of type `SIMPLICIAL-SET`. This function assigns an identification number `idnm` to the Kenzo object instance in a sequential way and pushes this object on the list of already created simplicial sets instances, `*smst-list*`.
The keywords parameters are:

- *cmpr*, a comparison function for generators.
- *basis*, if the simplicial set is effective, this argument must be the lisp function returning the list of non-degenerate simplices of the simplicial set, in a given dimension; if the simplicial set is locally effective, this argument must be the keyword `:locally-effective`.
- *bspn*, the lisp representation of the base point (an item of type `GMSM`). This is in fact the slot `bsgn` of the associated chain complex.
- *face*, a lisp function for the face operators.
- *face**, a lisp function for the face operators, returning the symbol `:degenerate` if the corresponding face is degenerate.
- *intr-bndr*, a lisp function for the differential of the associated chain complex.
- *bndr-strt*, the strategy (`:cmbn` or `:gnrt`) for the function *intr-bndr*.
- *intr-dgnl*, a lisp function for the coproduct of the coalgebra.
- *dngl-strt*, the strategy (`:cmbn` or `:gnrt`) for the function *intr-dgnl*.
- *org*, the comment list, adequately chosen.

The three arguments *face**, *intr-bndr* and *intr-dgnl* are not mandatory. Only the function *face* is necessary to define precisely the simplicial set. From the function *face*, we know that we may construct:

- The differential for the associated chain complex, according to the formula :
$$d^q = \sum_{i=0}^{q} (-1)^i \partial_i^q.$$
- The coproduct of the underlying coalgebra (the Alexander-Whitney diagonal, Δ) according to the formula:
$$\Delta(\sigma) = \sum_{i=0}^{n} \tilde{\delta}^{(i)}\sigma \otimes \delta_0^{(i)}\sigma,$$
where the power (i) denotes the i-th composition of the operator and $\tilde{\delta}$ is the *last face* operator, so that
$$\tilde{\delta}^{(i)} = \delta_{n-i+1} \circ \cdots \circ \delta_{n-1} \circ \delta_n.$$

But, as the differential ignore degenerate faces, it may be interesting for the user to provide a face function, weaker than the function *face* but sufficient and more efficient to compute the boundary of a simplex. It is the raison d'être of the parameter *face** which may be used as an alternative to *face*. In both cases, the strategy is set to `:gnrt` by the system. It may also arrive that the user knows a particularly simple form for the differential, he may then use the parameter *intr-bndr*. In the computation of differentials in the associated chain complex, this function will be used instead of the canonical differential computed from the face function. In this last case, the user is required to provide the strategy. Same remark for the parameter *intr-dgnl*.

Examples

As first example, we give the construction of the simplicial set Δ^n corresponding to the standard n-simplex. This is given only as example. The program `Kenzo` provides a function for the same purpose which will be described later. For every simplex, we shall use a list representation as in our first example `diabolo`. The various keyword arguments for the call to `build-smst`, are:

1. *bspn*, the list `(0)`, i.e. the representation of the vertex 0.

2. *cmpr*, the lisp function `#'l-cmpr`, since this function is adequate to compare the lists representing the simplices.

3. *basis*, the lisp function returning the list of all possible non–degenerate simplices in some dimension. Here, this is the combinatory function returning the set of all the lists of $p+1$ objects taken among $n+1$. Such a function is for instance the function (`delta-inj` p n) whose definition is given hereafter. So, the `:basis` keyword argument is simply:

   ```
   #'(lambda(dmn) (delta-inj dmn n)).
   ```

   Note that here, `n` is a global parameter.

4. *face*, the lisp function for the face operators (`face-delta-n`). It consists simply in removing the i–th element of the list describing a simplex (i starting at 0).

5. *org* will be a comment.

```
(defun delta-inj (m n)
  (declare (type fixnum m n))
  (cond ((> m n) +empty-list+)
        ((zerop m) (mapcar #'list (<a-b> 0 n)))
        (t (mapcan #'(lambda (list)
                       (declare (type list list))
                       (mapcar #'(lambda (k)
                                   (declare (type fixnum k))
                                   (cons k list))
                                (<a-b< 0 (first list))))
                   (delta-inj (1- m) n)))))

(delta-inj 0 3) ==>

((0) (1) (2) (3))

(delta-inj 1 3)  ==>

((0 1) (0 2) (0 3) (1 2) (1 3) (2 3))

(delta-inj 3 3)  ==>

((0 1 2 3))
```

```
(setf face-delta-n
  #'(lambda(i dmn gsm)
      (declare(ignore dmn))
      (absm 0 (append (subseq gsm 0 i)
                      (subseq gsm (1+ i))) )))
```

To create Δ^n for any order n, we embed the call of `build-smst` in a function having n as parameter.

```
(defun delta-n(n)
  (declare (type fixnum n))
        (build-smst :cmpr #'l-cmpr
                    :basis #'(lambda(dmn)(delta-inj dmn n))
                    :bspn '(0)
                    :face face-delta-n
                    :orgn `(delta-n ,n)))    ==>

DELTA-N

(setf delta4 (delta-n 4))  ==>

[K3 Simplicial-Set]

(bspn delta4)  ==>

(0)

(basis delta4 0)  ==>

((0) (1) (2) (3) (4))

(basis delta4 3)  ==>

((0 1 2 3) (0 1 2 4) (0 1 3 4) (0 2 3 4) (1 2 3 4))
```

A second example is the contruction of the simplicial complex $\Delta^{\mathbb{N}}$ freely generated by the positive integers. The construction of the `SIMPLICIAL-SET` instance to implement this simplicial set is quite similar to that of Δ^n, but in this case the `:basis` keyword argument is the keyword `:locally-effective`, since the sets K^q are infinite. Whence, the definition of $\Delta^{\mathbb{N}}$:

```
(setf delta-infty
        (build-smst :cmpr #'l-cmpr
                    :basis :locally-effective
                    :bspn '(0)
                    :face face-delta-n
                    :orgn `(delta-infinity)))
```

A third example is the construction of the simplicial set corresponding to `diabolo`. This simplicial set is a sub–complex of Δ^2. The `:face` function is exactly the same as in the examples above and the `:basis` function returns the sub–simplices of `diabolo`, by enumeration.

```
(setf diabolo-ss
        (build-smst :cmpr #'l-cmpr
                    :basis  #'(lambda (dmn)
                                (case dmn
                                      (0 '((0)(1)(2)(3)(4)(5)))
                                      (1 '((0 1)(0 2)(1 2)(2 3)(3 4)(3 5)(4 5)))
                                      (2 '((3 4 5)))))
                    :bspn '(0)
                    :face face-delta-n
                    :orgn '(simplicial set for diabolo)))
```

7.5 A first set of helpful functions on simplicial sets

We give now a first set of functions, methods or macros acting on simplicial set instances.

`cat-init` *[Function]*
Clear in particular `*smst-list*`, the list of user created simplicial sets and reset the global counter to 1.

`smst` *n* *[Function]*
Retrieve in the list `*smst-list*` the simplicial set instance whose the Kenzo identification is *n*. If it does not exist, return `NIL`.

`cmpr` *smst absm1 absm2* *[Macro]*
Compare the abstract simplices *absm1* and *absm2* with the comparison function associated with the simplicial set *smst*.

`bspn` *smst* *[Macro]*
Return the base point of the simplicial set *smst*.

`bndr` *smst* `&rest` *[Macro]*
Identical to the macro `dffr` (see the chain complex chapter). For simplicial sets, it is more traditional to use the term *boundary*.

`dgnl` *smst* `&rest` *[Macro]*
Identical to the macro `cprd` (see the coalgebra chapter). For simplicial sets, this recalls the Alexander-Witney diagonal.

`face` *smst i k gmsm-or-absm* *[Macro]*
Versatile macro to apply the face operator ∂_i^k on the simplex *gmsm-or-absm* of the simplicial set *smst*. The name of the simplex parameter shows clearly that it may be either a geometric simplex or an abstract simplex. With only one argument (*smst*), return the function *face*. In any other case, we recall that this application returns always an object of type `ABSM`.

`degenerate-p` *absm* *[Macro]*
Return `t` if the abstract simplex *asm* is degenerate, `nil` otherwise.

`non-degenerate-p` *absm* *[Macro]*
Return `t` if the abstract simplex *asm* is non-degenerate, `nil` otherwise.

`check-smst` *smst dmns1* `&optional` *dmns2 (1+ dmns1)* *[Function]*
Return `t` if the fundamental relations between the face operators ∂_i^k with $k = dmns1, \ldots, dmns2 - 1$, applied on the effective simplicial

set *smst*, are satisfied, otherwise `nil`. If the optional parameter *dmns2* is omitted, the test is done only for $k = dmns1$.

Examples

After having created instances of the standard simplicial sets Δ^2 and Δ^3 ($\Delta^{\mathbb{N}}$ has been defined previously), let us test some simple functions. Note that, since the class of simplicial sets is a sub-class of the class of coalgebras, itself subclass of chain complexes, the functions defined on these classes are available.

```
(setf delta2 (delta-n 2))  ==>

[K8 Simplicial-Set]

(setf delta3 (delta-n 3))  ==>

[K9 Simplicial-Set]

(degenerate-p (absm 0 '(1 2 3)))  ==>

NIL

(degenerate-p (absm 5 '(0 1 2 3 4)))  ==>

T

(basis delta3 2)  ==>

((0 1 2) (0 1 3) (0 2 3) (1 2 3))

(basis delta2 0)  ==>

((0) (1) (2))
```

The set of simplices in any dimension for the simplicial set $\Delta^{\mathbb{N}}$ is infinite:

```
(basis delta-infty 1)  ==>

Error: The object [K3 Simplicial-Set] is locally-effective.
```

Let us test the face function. We see that, though `delta-infty` is locally effective, nevertheless, we may work with its simplices:

```
(face delta-infty 2 4 '(0 1 2 3 4))  ==>

<AbSm - (0 1 3 4)>
```

```
(face delta-infty 2 4 (absm 0 '(0 1 2 3 4)))  ==>

<AbSm - (0 1 3 4)>

(face delta-infty 2 5 (absm (dgop-ext-int '(1)) '(0 1 2 3 4)))  ==>

<AbSm - (0 1 2 3 4)>

(face delta-infty 2 5 (absm (dgop-ext-int '(3)) '(0 1 2 3 4)))  ==>

<AbSm 2 (0 1 3 4)>

(bspn delta-infty)  ==>

(0)
```

The two following statement show that `delta-infty` may be considered also as a coalgebra and a chain complex.

```
(cprd delta-infty 4 '(0 1 2 3 4))  ==>

-------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr (0) (0 1 2 3 4)>>
<1 * <TnPr (0 1) (1 2 3 4)>>
<1 * <TnPr (0 1 2) (2 3 4)>>
<1 * <TnPr (0 1 2 3) (3 4)>>
<1 * <TnPr (0 1 2 3 4) (4)>>
-------------------------------------------------------------------------------

(? delta-infty 4 '(0 1 2 3 4))  ==>

-------------------------------------------------------------------------{CMBN 3}
<1 * (0 1 2 3)>
<-1 * (0 1 2 4)>
<1 * (0 1 3 4)>
<-1 * (0 2 3 4)>
<1 * (1 2 3 4)>
-------------------------------------------------------------------------------
```

The simplicial set `diabolo-ss`, has also all the properties of a coalgebra and of a chain complex.

```
(basis diabolo-ss 1)  ==>

((0 1) (0 2) (1 2) (2 3) (3 4) (3 5) (4 5))

(basis diabolo-ss 2)  ==>
```

```
((3 4 5))

(? diabolo-ss 2 (first *))  ==>

-------------------------------------------------------------------------{CMB 1}
<1 * (3 4)>
<-1 * (3 5)>
<1 * (4 5)>
-------------------------------------------------------------------------------

(? diabolo-ss *)  ==>

-------------------------------------------------------------------------{CMBN 0}
-------------------------------------------------------------------------------

(dgnl diabolo-ss **)  ==>

-------------------------------------------------------------------------{CMBN 1}
<1 * <TnPr (3) (3 4)>>
<-1 * <TnPr (3) (3 5)>>
<1 * <TnPr (4) (4 5)>>
<1 * <TnPr (3 4) (4)>>
<-1 * <TnPr (3 5) (5)>>
<1 * <TnPr (4 5) (5)>>
-------------------------------------------------------------------------------
```

7.6 The function build-finite-ss

The function `build-finite-ss` is a powerful function simplifying the creation of a finite simplicial set. From the lisp point of view, it accepts only one argument, the structured list of the simplices and their faces. In this list, every non–degenerate simplex must be coded as a symbol given by the user. The structure of the list is the following:

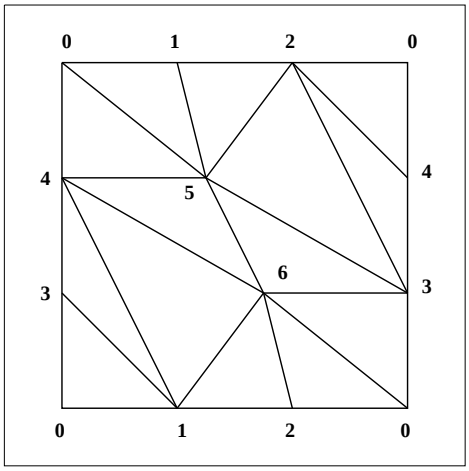
- **0** (in fact, optional)
- <Sequence of the vertices, the first item being the base point>,
- **1**
- <Sequence of description of simplices in dimension 1>,
- **2**
- <Sequence of description of simplices in dimension 2>,
- ... ...
- **n**
- <Sequence of description of simplices in dimension n>.

where <Sequence of description of simplices in dimension k> is a sequence of the following items:

1. A symbol, the symbolic name of the simplex.
2. A list of faces of this simplex, describing the faces of the simplex. A face can be described in two ways:
   - a symbol, the symbolic name of a **non-degenerate** simplex composing itself the face.
   - or a list of the form ($i_k\ i_{k-1}\ \ldots\ i_1\ i_0$ `symbol`) meaning that the degeneracy $\eta_{i_k}\eta_{i_{k-1}}\ldots\eta_{i_1}\eta_{i_0}$ is to be applied to the non-degenerate simplex of name `symbol` to produce the actual (possibly degenerate) face. If the simplex face has dimension 0 (a vertex), the name of this vertex is sufficient in the face description because the program computes itself the unique possible degeneracy of the vertex, i.e. $\eta_{k-2}\ldots\eta_0$.

Examples

Let us give some examples to clarify the use of `build-finite-ss`. In the first example, we use `build-finite-ss` to create a simplicial complex, corresponding to a triangulation of the torus. As the faces are never degenerated, the description of the faces is of simplified form.



The vertices will be named `v0`, `v1`, ..., `v6`, `v0` being the base point. The edges will be named `e01`, `e02`, ..., `e56`, the list of faces of the edge e_{ij} being $(v_j\ v_i)$. The triangles will be named `t013`, `t015`, ..., `t456`, the list of faces of the triangle t_{ijk} being $(e_{jk}\ e_{ik}\ e_{ij})$. With this convention of describing the simplices, the call to `build-finite-ss` is in simplified form:

```
(setf torus2
    (build-finite-ss
       '(v0  v1  v2  v3  v4  v5  v6

         1 e01  (v1 v0)   e02  (v2 v0)   e03  (v3 v0)
           e04  (v4 v0)   e05  (v5 v0)   e06  (v6 v0)
           e12  (v2 v1)   e13  (v3 v1)   e14  (v4 v1)
           e15  (v5 v1)   e16  (v6 v1)   e23  (v3 v2)
           e24  (v4 v2)   e25  (v5 v2)   e26  (v6 v2)
           e34  (v4 v3)   e35  (v5 v3)   e36  (v6 v3)
           e45  (v5 v4)   e46  (v6 v4)   e56  (v6 v5)

         2 t013  (e13 e03 e01)  t015  (e15 e05 e01)  t024  (e24 e04 e02)
           t026  (e26 e06 e02)  t036  (e36 e06 e03)  t045  (e45 e05 e04)
           t125  (e25 e15 e12)  t126  (e26 e16 e12)  t134  (e34 e14 e13)
           t146  (e46 e16 e14)  t234  (e34 e24 e23)  t235  (e35 e25 e23)
           t356  (e56 e36 e35)  t456  (e56 e46 e45)  )))
```

```
Checking the 0-simplices...
Checking the 1-simplices...
Checking the 2-simplices...
[K1 Simplicial-Set]
```

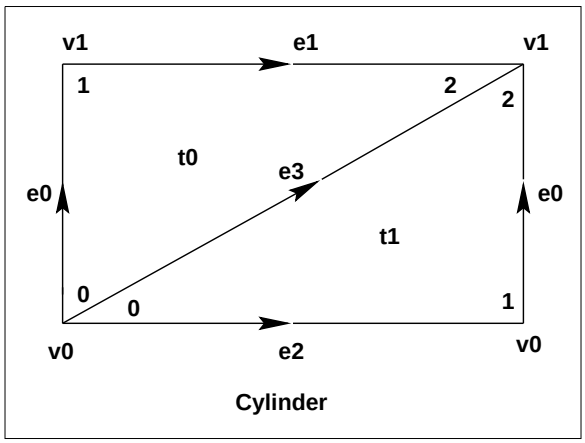
After verification of the coherence of the description of the faces, the function `build-finite-ss` calls adequately the function `build-smst` to create an instance of the class `SIMPLICIAL SET` subclass of the class `CHAIN COMPLEX`. So that, it is easy for the user to obtain the homology groups of the torus, for instance in dimension 1:

```
(chcm-homology  torus2 1)  ==>

Homology in dimension 1:

Component Z

Component Z
```

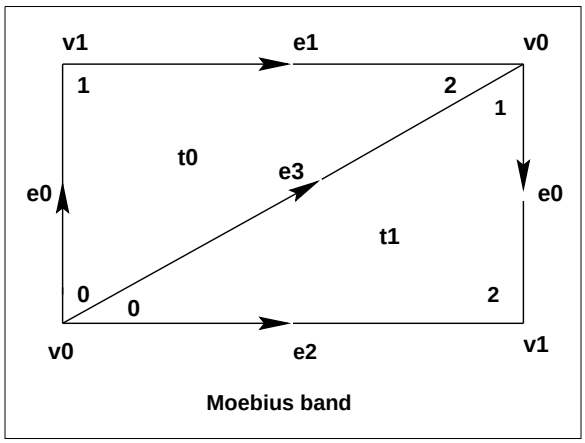
The function `build-finite-ss` calls internally some verification functions mentioned above to verify the coherence of the description of the simplicial set and gives an error message if the description of the simplicial set is incorrect. For instance:

```
(setf mm (build-finite-ss
   '(s0 s1 s2 s3
     1 0-1 (s1 s0)  1-2 (s2 s1)  2-3 (s3 s2)
     2 0-1-2 (2-3 1-2 0-1))
  ))

Checking the 0-simplices...
Checking the 1-simplices...
Checking the 2-simplices...
Error: Noncoherent boundary operators detected by CHECK-FACES :
Simplex => <AbSm - 0-1-2>
del_0 o del_0 => <AbSm - S3>
del_0 o del_1 => <AbSm - S2>
```

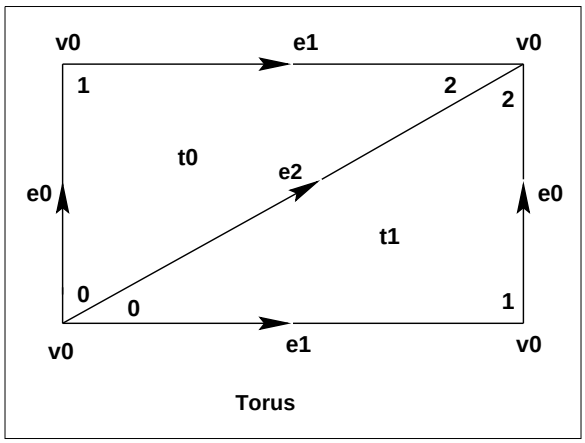
In the following examples, we show the power of `build-finite-ss` for the description of classical surfaces as simplicial sets, and in each case we verify the well known homology groups of those surfaces. The examples of the torus or the dunce-hat must be compared, as to the simplicity of the definition, to the respective ones in the previous page and in the Homology chapter.



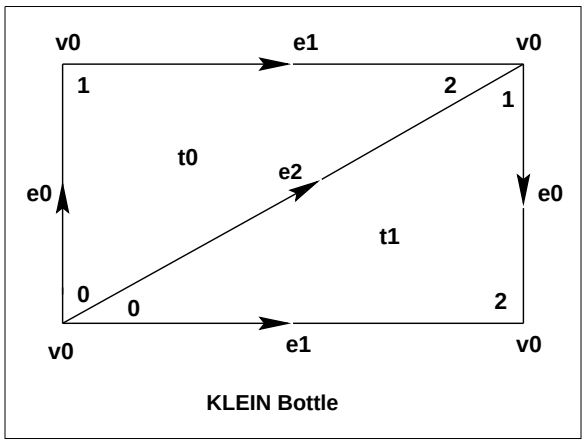

```
(setf cylinss (build-finite-ss
    '(v0 v1
      1 e0 (v1 v0) e1 (v1 v1) e2 (v0 v0) e3 (v1 v0)
      2 t0 (e1 e3 e0) t1 (e0 e3 e2)) ))  ==>

[K2 Simplicial-Set]

(dotimes (i 3) (chcm-homology cylinss i)) ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z

Homology in dimension 2 :

---done---
```



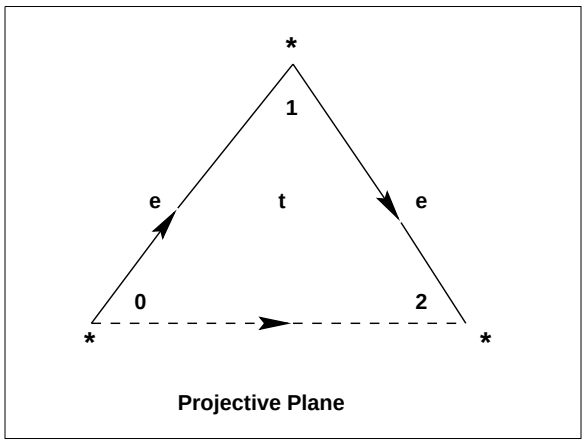

```
(setf moebiuss (build-finite-ss
  '(v0 v1
    1 e0 (v1 v0) e1 (v0 v1) e2 (v1 v0) e3 (v0 v0)
    2 t0 (e1 e3 e0) t1 (e0 e2 e3)) ))  ==>

[K3 Simplicial-Set]

(dotimes (i 3) (chcm-homology moebiuss i)) ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z

Homology in dimension 2 :

---done---
```




```
(setf toruss (build-finite-ss
  '(v0
    1 e0 (v0 v0) e1 (v0 v0) e2 (v0 v0)
    2 t0 (e1 e2 e0) t1 (e0 e2 e1)) ))           ==>

[K4 Simplicial-Set]

(dotimes (i 3) (chcm-homology toruss i))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z

Component Z

Homology in dimension 2 :

Component Z

---done---
```

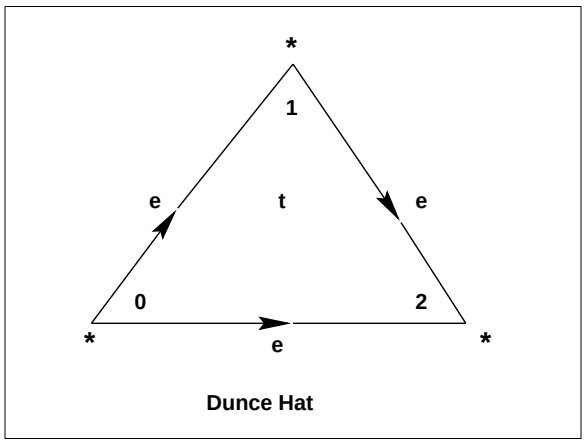



```
(setf bottless (build-finite-ss
  '(v0
    1 e0 (v0 v0) e1 (v0 v0) e2 (v0 v0)
    2 t0 (e1 e2 e0) t1 (e0 e1 e2)) ))           ==>

[K5 Simplicial-Set]

(dotimes (i 3) (chcm-homology bottless i))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Component Z

Homology in dimension 2 :

---done---
```




```
(setf ppss (build-finite-ss
  '( *
     1 e (* *)
     2 t (e * e)) ))             ==>

[K6 Simplicial-Set]

(dotimes (i 3) (chcm-homology ppss i))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :

---done---
```

The user will note that the list of faces for the 2–simplex `t`, is in simplified form. In particular, as the face 1 of `t` is the 0–degeneracy of the base point "$*$", it is sufficient to code the face 1 of `t` by "`*`" instead of the complete form `(0 *)`.



Dunce Hat

```
(setf duncess (build-finite-ss
  '( *
     1 e (* *)
     2 t (e e e)) ))  ==>

[K7 Simplicial-Set]

(dotimes (i 3) (chcm-homology duncess i))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

---done---
```

7.7 Special simplicial sets

The system provides useful functions to create interesting simplicial sets of constant usage. In particular, the user is advised from now to use the following version of the standard simplex implemented in `Kenzo` and not the one given previously for simplicity reason.

7.7.1 The standard simplex

`delta` *dmns* *[Function]*

Create the standard simplex Δ^n with vertices $0, 1, \ldots, n$, where *dmns* is the parameter for n. This simplicial set is so important in the applications that the implementor has decided to code the simplices in a way similar to the coding of the degeneracy operators. An **increasing** sequence of non-negative integers describing a simplex, is coded on a binary integer with the following convention: a number i representing the vertex i is the $(i+1)$–th binary bit of a machine word. The base point 0 is the bit 1 in position 1 of the word. This representation is very efficient for saving memory space but somehow awkward to read.

`delta-infinity` *[Function]*

Create the locally effective standard simplex $\Delta^{\mathbb{N}}$ freely generated by the positive integers.

`deltab` *[Function]*

Create the locally effective **reduced** simplicial set $\bar{\Delta}$, built from $\Delta^{\mathbb{N}}$ by identifying all the vertices to the base point.

`deltab2` *[Function]*

Create the locally effective **1–reduced** simplicial set $\Delta_2^{\mathbb{N}}$, obtained from the above $\bar{\Delta}$, by identification of all the edges with the base point. This is the efficient version of the coalgebra that is used as example in the cobar chapter.

`dlop-ext-int` *ext-dlop* *[Function]*

Code on an integer the valid list (increasing order) representing the simplex *ext-dlop* of the standard simplex.

`dlop-int-ext` *dgop* *[Function]*

Give the list representing a simplex of the standard simplex from the integer *dgop*.

`vertex-i` *absm i* *[Function]*
: Give the *i*–th vertex of the (coded, degenerate or not) abstract simplex *absm* belonging to a simplicial set of the Δ family.

`absm-ext-int` *vlist* *[Function]*
: Create a valid abstract simplex of the Δ family from the simplex *vlist* written as a non-decreasing list of non–negative integers (i.e coding of any degenerate or non–degenerate simplex of the Δ family).

`absm-int-ext` *absm* *[Function]*
: From an internal coded form of an abstract simplex of the Δ family, create the non-decreasing list of non–negative integers representing the canonical external form of such a simplex.

`soft-delta` *dmns* *[Function]*
: Create another version of the standard simplex Δ^n designed for a better clarity in the printing of the results at testing time. More precisely, the simplices are represented internally by a list of the form `(:delt` *binary-code*`)`. With this representation, the system is able to recognize a coded simplex and to print it in a readable form. For the user, the only requirement is to write a simplex coded n as `(d` n`)`, where `d` is the macro building the internal representation (see the examples). Attention: the above conversion functions do not work with this representation.

`soft-delta-infinity` *[Function]*
: Create another version of the locally effective standard simplex $\Delta^{\mathbb{N}}$ (see the function `soft-delta` above).

Examples

```
(setf d3 (delta 3))  ==>

[K8 Simplicial-Set]
```

The simplices 2, 4 and 8 are the coded representation of the vertices 1, 2 and 3:

```
(cmpr d3 2 4)  ==>

:LESS

(cmpr d3 4 4)  ==>

:EQUAL
```

```
(cmpr d3 8 4)  ==>

:GREATER
```

The list returned by the following statement getting the basis of Δ^3 in dimension 2, namely `(7 11 13 14)` is in fact the coded list of the following simplices `((0 1 2) (0 1 3) (0 2 3) (1 2 3))`.

```
(basis d3 2)  ==>

(7 11 13 14)
```

In the following statement, the integer 21 represents the simplex in dimension 2, `(0 1 4)`, 1 represents the operator ∂_1 and the face is therefore `(0 4)` represented as the integer 17.

```
(face d3 1 2 21)  ==>

<AbSm - 17>
```

We may obtain the same result, using:

```
(face d3 1 2 (dlop-ext-int '(0 1 4)))  ==>

<AbSm - 17>
```

Let us test the conversion functions.

```
(vertex-i (absm 0 1) 0)  ==>

0

(vertex-i (absm 1 1) 0)  ==>

0

(vertex-i (absm 1 1) 1)  ==>

0

(vertex-i (absm 0 7) 2)  ==>

2

(absm-ext-int '(0 0 0 1 2 3 3 3))  ==>

<AbSm 6-5-1-0 15>

(absm-ext-int '(0 1 1 1 2))  ==>
```

```
<AbSm 2-1 7>

(absm-int-ext (absm-ext-int '(0 0 0 1 2 3 3 3)))  ==>

(0 0 0 1 2 3 3 3)
```

The following call to the macro `?` returns the boundary of the simplex `(0 2 3)` of Δ^3, the integers 5, 9 and 12 representing respectively the simplices `(0 2)`, `(0 3)` and `(2 3)`. The application of the coproduct `dgnl` to the simplex `(0 1 2 3)`, coded 15, is also easily interpreted:

```
(? d3 2 13)

------------------------------------------------------------------------{CMB 1}
<1 * 5>
<-1 * 9>
<1 * 12>
-------------------------------------------------------------------------------

(dgnl d3 3 15)  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 1 15>>
<1 * <TnPr 3 14>>
<1 * <TnPr 7 12>>
<1 * <TnPr 15 8>>
-------------------------------------------------------------------------------
```

We may show now some examples with the *soft* version of the standard simplex. Using the macro `d` and the function `dlop-ext-int` the user may work with a readable form of the simplices and the degeneracy operators.

```
(setf d3 (soft-delta 3))  ==>

[K10 Simplicial-Set]

(cmpr d3 (d 2) (d 4))  ==>

:LESS

(basis d3 1)  ==>

(0-1 0-2 1-2 0-3 1-3 2-3)
```

```
(dgnl d3 3 (d (dlop-ext-int '(0 1 2 3))))  ==>

-------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 0 0-1-2-3>>
<1 * <TnPr 0-1 1-2-3>>
<1 * <TnPr 0-1-2 2-3>>
<1 * <TnPr 0-1-2-3 3>>
-------------------------------------------------------------------------------

(face d3 1 2 (d (dlop-ext-int '(0 2 4))))  ==>

<AbSm - 0-4>

(? d3 2 (d (dlop-ext-int '(0 2 3))))  ==>

-------------------------------------------------------------------------{CMBN 1}
<1 * 0-2>
<-1 * 0-3>
<1 * 2-3>
-------------------------------------------------------------------------------
```

7.7.2 Spheres, Moore spaces and projectives spaces

`sphere` *n* *[Function]*

Create a simplicial set, a model for the sphere of dimension n, ($n \geq 0$). This is a typical example where the differential is known a priori to be null, so the `:intr-bndr` keyword parameter is set to function `zero-pure-dffr`. This function generates a name `S`n for the **unique** simplex of dimension n whose faces are the degeneracies of the base point labelled "`*`".

`sphere-wedge` *dmns1 ... dmnsn* *[Function]*

Create a simplicial set for the wedge of spheres. Here, the $dmns_i$ are integers, namely the dimensions of the spheres to be wedged. The differential is null. In the representation created by the software, the 0–simplex (base point) is labelled "`*`" and in dimension p the simplexes are labelled `S`*p-1*, ..., `S`*p-s*, where s is the number of spheres of dimension p in the wedge. (See the example).

`moore` *p n* *[Function]*

Construct a simplicial set, a model for Moore($\mathbb{Z}/p\mathbb{Z}, n$). The integers p and n must satisfy the conditions $p > 1,\ n > 2p-4$, otherwise the result is undefined. Moore($\mathbb{Z}/p\mathbb{Z}, n$) is a space, whose the only non–null homology groups are $H_0 = \mathbb{Z}$ and $H_n = \mathbb{Z}/p\mathbb{Z}$. A Moore space has only three non–degenerate simplices, namely in dimension 0, n and $n+1$. A number p of faces of the $(n+1)$–simplex are identified with the n–simplex, the others faces being contracted on the base point. In the representation created by the software, the 0–simplex (base point), the n–simplex and the $(n+1)$–simplex are respectively labelled "`*`", `M`n and `N`n', where $n' = n+1$.

`R-proj-space &optional` *k l* *[Function]*

If $k = 1$ or omitted, build a simplicial set model of $K(\mathbb{Z}_2, 1) = P^\infty\mathbb{R}$. In dimension n, this simplicial set has only one non-degenerate simplex, namely the integer n. The faces of this non-degenerate simplex n are given by the following formulas: $\partial_0 n = \partial_n n = n-1$ and for $i \neq 0$ and $i \neq n$, $\partial_i n = \eta_{i-1}(n-2)$. If $k > 1$, build an analogous simplicial set but with no simplices in dimensions $1 \leq m < k$. If in addition to k the argument l, ($l \geq k$), is provided, build an analogous simplicial set but with no simplices in dimensions $m \geq l$.

Examples

Let us define first, an auxiliary function `show-structure` with 2 arguments *ss* and *dmn*, to show the structure (i.e. generators and faces) of the simplicial set *ss*, from the dimension 0 up to the dimension *dmn* included.

```
(defun show-structure (ss dmn)
 (dotimes (i (1+ dmn))
   (format t "~2%Dimension = ~D :" i)
   (case i
     (0 (format t "~2%~8TVertices : ~8T~A" (basis ss 0)))
     (otherwise
        (dolist (s (basis ss i))
            (format t "~2%~8TSimplex : ~A~2%~16TFaces : ~A"
                    s (mapcar #'(lambda (j) (face ss j i s))
                              (<a-b> 0 i)))))
  )))
```

In these elementary examples, we show the structure of some simplicial sets built from the functions above. Let us begin with S^2.

```
(setf s2 (sphere 2))  ==>

[K11 Simplicial-Set]

(bspn s2)  ==>

*
```

Applying the function `basis`, we see that the only non-null simplices are in dimension 0 and 2.

```
(dotimes (i 4) (print (basis s2 i)))  ==>

(*)
NIL
(S2)
NIL
```

In dimension 2, the 3 faces of the simplicial set are all the degeneracy η_0 of the base point.

```
(mapcar #'(lambda(i)(face s2 i 2 's2)) '(0 1 2))  ==>

(<AbSm 0 *> <AbSm 0 *> <AbSm 0 *>)

(face s2 5 7 (absm (dgop-ext-int '(5 3 1 0)) 's2))  ==>

<AbSm 3-1-0 S2>
```

```
(face s2 2 7 (absm (dgop-ext-int '(5 3 1 0)) 's2))  ==>

<AbSm 4-2-0 S2>
```

The differential of the simplex `s2` in dimension 2 is of course the null combination of degree 1:

```
(? s2 2 's2)  ==>

-------------------------------------------------------------------------{CMB 1}
--------------------------------------------------------------------------------
```

Now, let us see the stucture of S^3, using our auxiliary function `show-structure`.

```
(setf s3 (sphere 3))  ==>

[K12 Simplicial-Set]

(show-structure s3 3)  ==>

Dimension = 0 :

        Vertices :  (*)

Dimension = 1 :

Dimension = 2 :

Dimension = 3 :

        Simplex : S3

                Faces : (<AbSm 1-0 *> <AbSm 1-0 *> <AbSm 1-0 *> <AbSm 1-0 *>)
```

The space Moore$(2,1)$ is the projective plane:

```
(setf p2 (moore 2 1))  ==>

[K13 Simplicial-Set]

(show-structure p2 2)  ==>

Dimension = 0 :

        Vertices :  (*)
```

```
Dimension = 1 :

        Simplex : M1

                Faces : (<AbSm - *> <AbSm - *>)

Dimension = 2 :

        Simplex : N2

                Faces : (<AbSm - M1> <AbSm 0 *> <AbSm - M1>)
```

In the following example, note the identification of p faces of the $(n+1)$–simplex with the n–simplex.

```
(setf sp2r (moore 2 2))  ==>

[K15 Simplicial-Set]

(show-structure sp2r 3)  ==>

Dimension = 0 :

        Vertices :  (*)

Dimension = 1 :

Dimension = 2 :

        Simplex : M2

                Faces : (<AbSm 0 *> <AbSm 0 *> <AbSm 0 *>)

Dimension = 3 :

        Simplex : N3

                Faces : (<AbSm - M2> <AbSm 1-0 *> <AbSm - M2>
                         <AbSm 1-0 *>)
```

Let us see an example of a wedge of spheres:

```
(setf w (sphere-wedge 3 2 3))  ==>

[K16 Simplicial-Set]

(show-structure w 5)  ==>
```

```
Dimension = 0 :

        Vertices :  (*)

Dimension = 1 :

Dimension = 2 :

        Simplex : S2-1

                Faces : (<AbSm 0 *> <AbSm 0 *> <AbSm 0 *>)

Dimension = 3 :

        Simplex : S3-1

                Faces : (<AbSm 1-0 *> <AbSm 1-0 *> <AbSm 1-0 *>
                         <AbSm 1-0 *>)

        Simplex : S3-2

                Faces : (<AbSm 1-0 *> <AbSm 1-0 *> <AbSm 1-0 *>
                         <AbSm 1-0 *>)

Dimension = 4 :

Dimension = 5 :

(cmpr w 's3-1 's3-2)  ==>

:LESS

(face w 2 3 's3-1)  ==>

<AbSm 1-0 *>

(? w 3 's3-2)  ==>

-------------------------------------------------------------------------{CMB 2}
-------------------------------------------------------------------------------
```

Let us show now some examples with the simplicial sets generated by the function `R-proj-space`.

```
(setf p1 (R-proj-space))  ==>

[K20 Simplicial-Set]

(dotimes (i 7)(print(basis p1 i)))  ==>

(0)
(1)
(2)
(3)
(4)
(5)
(6)

(show-structure p1 5)  ==>

Dimension = 0 :

        Vertices :  (0)

Dimension = 1 :

        Simplex : 1

                Faces : (<AbSm - 0> <AbSm - 0>)

Dimension = 2 :

        Simplex : 2

                Faces : (<AbSm - 1> <AbSm 0 0> <AbSm - 1>)

Dimension = 3 :

        Simplex : 3

                Faces : (<AbSm - 2> <AbSm 0 1> <AbSm 1 1> <AbSm - 2>)

Dimension = 4 :

        Simplex : 4

                Faces : (<AbSm - 3> <AbSm 0 2> <AbSm 1 2> <AbSm 2 2> <AbSm - 3>)

Dimension = 5 :
```

```
        Simplex : 5

                Faces : (<AbSm - 4> <AbSm 0 3> <AbSm 1 3> <AbSm 2 3> <AbSm 3 3>
                         <AbSm - 4>)

(dotimes (i 5)(chcm-homology p1 i)) ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :

---done---

Homology in dimension 3 :

Component Z/2Z

Homology in dimension 4 :

---done---

(setf p2 (R-proj-space 2))  ==>

[K21 Simplicial-Set]

(show-structure p2 5)  ==>

Dimension = 0 :

        Vertices :  (0)

Dimension = 1 :

Dimension = 2 :

        Simplex : 2

                Faces : (<AbSm 0 0> <AbSm 0 0> <AbSm 0 0>)

Dimension = 3 :
```

```
        Simplex : 3

                Faces : (<AbSm - 2> <AbSm 1-0 0> <AbSm 1-0 0> <AbSm - 2>)

Dimension = 4 :

        Simplex : 4

                Faces : (<AbSm - 3> <AbSm 0 2> <AbSm 1 2> <AbSm 2 2> <AbSm - 3>)

Dimension = 5 :

        Simplex : 5

                Faces : (<AbSm - 4> <AbSm 0 3> <AbSm 1 3> <AbSm 2 3> <AbSm 3 3>
                         <AbSm - 4>)

(dotimes (i 5)(chcm-homology p2 i))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

---done---

Homology in dimension 2 :

Component Z

Homology in dimension 3 :

Component Z/2Z

Homology in dimension 4 :

---done---

(setf pr3 (R-proj-space 3))  ==>

[K22 Simplicial-Set]

(show-structure pr3 4)  ==>

Dimension = 0 :
```

```
        Vertices :  (0)

Dimension = 1 :

Dimension = 2 :

Dimension = 3 :

        Simplex : 3

                Faces : (<AbSm 1-0 0> <AbSm 1-0 0> <AbSm 1-0 0> <AbSm 1-0 0>)

Dimension = 4 :

        Simplex : 4

                Faces : (<AbSm - 3> <AbSm 2-1-0 0> <AbSm 2-1-0 0>
                         <AbSm 2-1-0 0> <AbSm - 3>)

(dotimes (i 7)  (print (? pr3 i i)))  ==>

-------------------------------------------------------------------------{CMBN -1}
------------------------------------------------------------------------------

-------------------------------------------------------------------------{CMBN 0}
------------------------------------------------------------------------------

-------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

-------------------------------------------------------------------------{CMBN 2}
------------------------------------------------------------------------------

-------------------------------------------------------------------------{CMBN 3}
<2 * 3>
------------------------------------------------------------------------------

-------------------------------------------------------------------------{CMBN 4}
------------------------------------------------------------------------------

-------------------------------------------------------------------------{CMBN 5}
<2 * 5>
------------------------------------------------------------------------------
```

Let us use now the parameter l, allowing the truncation in the upper dimensions.

```
(setf p12 (R-proj-space 1 2))  ==>
```

```
[K23 Simplicial-Set]

(show-structure p12 2)  ==>

Dimension = 0 :

        Vertices :  (0)

Dimension = 1 :

        Simplex : 1

                Faces : (<AbSm - 0> <AbSm - 0>)

Dimension = 2 :
```

Setting $k = l$ does not creates something amazing!

```
(setf p22 (R-proj-space 2 2))  ==>

[K28 Simplicial-Set]

(show-structure p22 2)  ==>

Dimension = 0 :

        Vertices :  (0)

Dimension = 1 :

Dimension = 2 :

(setf p47 (R-proj-space 4 7))  ==>

[K33 Simplicial-Set]

(show-structure p47 8)  ==>

Dimension = 0 :

        Vertices :  (0)

Dimension = 1 :

Dimension = 2 :

Dimension = 3 :
```

```
Dimension = 4 :

        Simplex : 4

                Faces : (<AbSm 2-1-0 0> <AbSm 2-1-0 0> <AbSm 2-1-0 0>
                         <AbSm 2-1-0 0> <AbSm 2-1-0 0>)

Dimension = 5 :

        Simplex : 5

                Faces : (<AbSm - 4> <AbSm 3-2-1-0 0> <AbSm 3-2-1-0 0>
                         <AbSm 3-2-1-0 0> <AbSm 3-2-1-0 0> <AbSm - 4>)

Dimension = 6 :

        Simplex : 6

                Faces : (<AbSm - 5> <AbSm 0 4> <AbSm 1 4> <AbSm 2 4>
                         <AbSm 3 4> <AbSm 4 4> <AbSm - 5>)

Dimension = 7 :

Dimension = 8 :
```

7.8 Cartesian product of simplicials sets

Let X and Y be two simplicial sets, the construction of $X \times Y$ is based on the very definition $(X \times Y)_n = X_n \times Y_n$ where X_n, Y_n and $(X \times Y)_n$ are the *possibly degenerate* simplices of X, Y and $X \times Y$ respectively. A simplex of the product $X \times Y$ is characterized by its projections on the factors X and Y.
The *non–degenerate* simplices of $X \times Y$ are represented internally in the system by a lisp object of the form:

`(:crpr (`*dgop1*`.`*gmsm1*`).(`*dgop2*`.`*gmsm2*`))`

where,

1. `dgop1` is an integer representing a coded degeneracy operator.
2. `gmsm1` is a non-degenerate simplex of X, to which is applied the degeneracy operator `dgop1`.
3. `dgop2` is an integer representing a coded degeneracy operator.
4. `gmsm2` is a non-degenerate simplex of Y, to which is applied the degeneracy operator `dgop2`.

This object must be a *non–degenerate* simplex of $X \times Y$, that is to say, the degeneracy operators `dgop1` and `dgop2` must not have a common η_j, and therefore their list representations must have a void intersection. The coded representation by binary bit positions, has the same property. The corresponding type is `CRPR`. To construct such an object, one may use the macro `crpr`. As usual, a printing method has been defined to reflect the structure of the product under the form:

`<CrPr` *ext-dgop1 gmsm1 ext-dgop2 gmsm2*`>`

where the sequence of the operators η_i is printed in explicit form (this is the meaning of *ext-dgop*). If this sequence of η_i is void, i.e. if the simplex *gmsm* is not degenerate, the symbol `-` is printed instead.

7.8.1 Functions and macros for the product of simplicial sets

`crpr` *dgop1 gmsm1 dgop2 gmsm2* *[Macro]*
Build an object of type `CRPR`, using directly the integer coding for the degeneracy operators. The arguments *gmsm1* and *gmsm2* are non-degenerate simplices.

`crpr` *absm1 absm2* *[Macro]*
Build an object of type `CRPR`, using two abstract simplices *absm1* and *absm2*. If these abstract simplices are degenerate, the degeneracy operators must verify the condition of the definition, i.e. no common η_i.

`2absm-acrpr` *absm1 absm2* *[Function]*
Build an abstract simplex, i.e. object of type `ABSM`, cartesian product of both abstract simplices *absm1* and *absm2*. In contrast with the previous function, there is no condition upon the degeneracy operators of the abstract simplices. Of course, the function `2absm-acrpr` returns a legal abstract simplex.

`crpr-p` *object* *[Predicate]*
Test if *object* is of type `CRPR`.

`dgop1` *crpr* *[Macro]*
Select the degeneracy operator *dgop1* from the object `crpr`.

`gmsm1` *crpr* *[Macro]*
Select the geometric simplex *gmsm1* from the object `crpr`.

`dgop2` *crpr* *[Macro]*
Select the degeneracy operator *dgop2* from the object `crpr`.

`gmsm2` *crpr* *[Macro]*
Select the geometric simplex *gmsm2* from the object `crpr`.

`absm1` *crpr* *[Macro]*
Build the abstract simplex `(absm` *dgop1 gmsm1*`)` from *crpr*.

`absm2` *crpr* *[Macro]*
Build the abstract simplex `(absm` *dgop2 gmsm2*`)` from *crpr*.

`crts-prdc` *smst1 smst2* *[Function]*
Build the simplicial set, cartesian product $smst1 \times smts2$.

Examples

In the first example, note that the coded representations of η_1 and η_2 are respectively 2 and 4, but these degeneracy operators appear clearly as 1 and 2 in the printed result. In the second statement, the integer 28 is the coded representation of the degeneracy `(4 3 2)`.

```
(crpr 2 'a 4 'b)  ==>

<CrPr 1 A 2 B>

(crpr 0 '(0 1 2 3 4 5) 28 '(0 1 2))   ==>

<CrPr - (0 1 2 3 4 5) 4-3-2 (0 1 2)>
```

For more clarity, one may also write:

```
(crpr 0 '(0 1 2 3 4 5) (dgop-ext-int '(4 3 2)) '(0 1 2))   ==>

<CrPr - (0 1 2 3 4 5) 4-3-2 (0 1 2)>
```

In the following example, we see that both functions `crpr` and `2absm-acrpr` return the same geometric simplex,

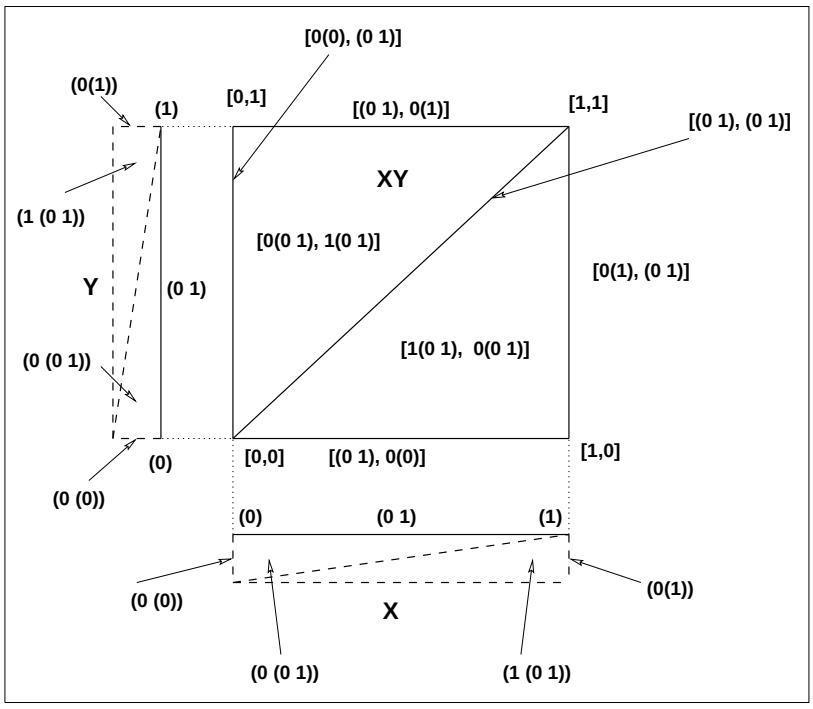
```
(crpr (absm 4 'a)(absm 3 'b))  ==>

<CrPr 2 A 1-0 B>

(2absm-acrpr (absm 4 'a)(absm 3 'b))  ==>

<AbSm - <CrPr 2 A 1-0 B>>
```

whereas, in the following call to `crpr`, the condition upon the degeneracy operator is not respected and the result is an illegal cartesian product. The function `2absm-acrpr` returns the correct answer.

```
(crpr (absm 5 'a)(absm 3 'b))  ==>

<CrPr 2-0 A 1-0 B>     ;;; ILLEGAL!

(2absm-acrpr (absm 5 'a) (absm 3 'b))  ==>

<AbSm 0 <CrPr 1 A 0 B>>
```

The following example is particularly instructive. The simplicial set whose the realization is homeomorphic to the square is built by using the product of two segments. A call like `(delta n)` builds the standard n–simplex. So first, we generate two copies `X` and `Y` of the standard 1–simplex and we list the abstract simplices up to dimension 2 for a better understanding of the components of the product `XY`.

```
(setf X (delta 1))  ==>

[K4 Simplicial-Set]

(setf Y X)  ==>

[K4 Simplicial-Set]
```



In list representation (non-coded simplices), the abstract simplices of X up to the dimension 2, including the degenerate ones are:

- Dimension 0: `<AbSm - (0)>`, `<AbSm - (1)>`.
- Dimension 1: `<AbSm 0 (0)>`, `<AbSm 0 (1)>`, `<AbSm - (0 1)>`.
- Dimension 2: `<AbSm 1-0 (0)>`, `<AbSm 1-0 (1)>`, `<AbSm 0 (0 1)>`, `<AbSm 1 (0 1)>`.

Let us look first at the non-degenerate simplices of the simplicial set $X \times Y$. Those simplices are objects of type `CRPR`. In dimension 1, we see that some projections are degeneracies of vertices and, in dimension 2, the projections are all degeneracies of the 1–simplex `(0 1)` coded as 3. Nevertheless, all the listed simplices are *geometric* simplices (i.e. non-degenerate ones) of the product `XY`. The diagram above shows the relationship between the 3 simplicial sets.

```
(setf XY (crts-prdc X Y))  ==>

[K5 Simplicial-Set]
```

```
(show-structure XY 2)  ==>

Dimension = 0 :

        Vertices :  (<CrPr - 1 - 1> <CrPr - 1 - 2> <CrPr - 2 - 1>
                     <CrPr - 2 - 2>)

Dimension = 1 :

        Simplex : <CrPr - 3 - 3>

                Faces : (<AbSm - <CrPr - 2 - 2>>
                         <AbSm - <CrPr - 1 - 1>>)

        Simplex : <CrPr - 3 0 1>

                Faces : (<AbSm - <CrPr - 2 - 1>>
                         <AbSm - <CrPr - 1 - 1>>)

        Simplex : <CrPr - 3 0 2>

                Faces : (<AbSm - <CrPr - 2 - 2>>
                         <AbSm - <CrPr - 1 - 2>>)

        Simplex : <CrPr 0 1 - 3>

                Faces : (<AbSm - <CrPr - 1 - 2>>
                         <AbSm - <CrPr - 1 - 1>>)

        Simplex : <CrPr 0 2 - 3>

                Faces : (<AbSm - <CrPr - 2 - 2>>
                         <AbSm - <CrPr - 2 - 1>>)

Dimension = 2 :

        Simplex : <CrPr 0 3 1 3>

                Faces : (<AbSm - <CrPr - 3 0 2>>
                         <AbSm - <CrPr - 3 - 3>>
                         <AbSm - <CrPr 0 1 - 3>>)

        Simplex : <CrPr 1 3 0 3>

                Faces : (<AbSm - <CrPr 0 2 - 3>>
                         <AbSm - <CrPr - 3 - 3>>
                         <AbSm - <CrPr - 3 0 1>>)
```

To be more readable we may translate the coded representations of the degeneracy operators and of the simplices in the list representation:

```
Dimension 0 :

        Vertices : (<CrPr - (0) - (0)>
                    <CrPr - (0) - (1)>
                    <CrPr - (1) - (0)>
                    <CrPr - (1) - (1)>)

Dimension 1 :

        Simplex : <CrPr 0 (0) - (0 1)>

                Faces : (<AbSm - <CrPr - (0) - (1)> >
                         <AbSm - <CrPr - (0) - (0)> >)

        Simplex : <CrPr 0 (1) - (0 1)>

                Faces : (<AbSm - <CrPr - (1) - (1)> >
                         <AbSm - <CrPr - (1) - (0)> >)

        Simplex : <CrPr - (0 1) 0 (0)>

                Faces : (<AbSm - <CrPr - (1) - (0)> >
                         <AbSm - <CrPr - (0) - (0)> >)

        Simplex : <CrPr - (0 1) 0 (1)>

                Faces : (<AbSm - <CrPr - (1) - (1)> >
                         <AbSm - <CrPr - (0) - (1)> >)

        Simplex : <CrPr - (0 1) - (0 1)>

                Faces : (<AbSm - <CrPr - (1) - (1)> >
                         <AbSm - <CrPr - (0) - (0)> >)

Dimension 2 :

        Simplex : <CrPr 0 (0 1) 1 (0 1)>

                Faces : (<AbSm - <CrPr - (0 1) 0 (1)> >
                         <AbSm - <CrPr - (0 1) - (0 1)> >
                         <AbSm - <CrPr 0 (0) - (0 1)> >)

        Simplex : <CrPr 1 (0 1) 0 (0 1)>
```

```
                Faces : (<AbSm - <CrPr 0 (1) - (0 1)> >
                         <AbSm - <CrPr - (0 1) - (0 1)> >
                         <AbSm - <CrPr - (0 1) 0 (0)> >)
```

In an analogous way, let us build the torus as the cartesian product of two 1–spheres. We recall that the base point of the sphere is named *.

```
(setf s1xs1 (crts-prdc (sphere 1) (sphere 1)))  ==>

[K15 Simplicial-Set]

(show-structure s1xs1 2)  ==>

Dimension = 0 :

        Vertices :  (<CrPr - * - *>)

Dimension = 1 :

        Simplex : <CrPr - S1 - S1>

                Faces : (<AbSm - <CrPr - * - *>>
                         <AbSm - <CrPr - * - *>>)

        Simplex : <CrPr - S1 0 *>

                Faces : (<AbSm - <CrPr - * - *>>
                         <AbSm - <CrPr - * - *>>)

        Simplex : <CrPr 0 * - S1>

                Faces : (<AbSm - <CrPr - * - *>>
                         <AbSm - <CrPr - * - *>>)

Dimension = 2 :

        Simplex : <CrPr 0 S1 1 S1>

                Faces : (<AbSm - <CrPr - S1 0 *>>
                         <AbSm - <CrPr - S1 - S1>>
                         <AbSm - <CrPr 0 * - S1>>)

        Simplex : <CrPr 1 S1 0 S1>

                Faces : (<AbSm - <CrPr 0 * - S1>>
                         <AbSm - <CrPr - S1 - S1>>
                         <AbSm - <CrPr - S1 0 *>>)
```

Let us verify that $S^1 \times S^1$ has the same homology groups as the torus.

```
(dotimes (i 3)(chcm-homology s1xs1 i)) ==>

Homology in dimension 0 :

 Component Z

Homology in dimension 1 :

 Component Z

 Component Z

Homology in dimension 2 :

 Component Z
```

Of course, we may generate more complex products, whose basis are composed of rather long lists of simplices as shown below. The user will also pay attention to the fact that the composition of the macro `crpr` is not associative. In the following example, the simplicial sets located respectively by `p1` and `p2` are different objects, though being canonically isomorphic.

```
(setf p1 (crts-prdc (sphere 2) (crts-prdc (sphere 2) (sphere 3))))  ==>

[K20 Simplicial-Set]

(setf p2 (crts-prdc (crts-prdc (sphere 2) (sphere 2)) (sphere 3)))  ==>

[K21 Simplicial-Set]

(dotimes (i 8) (print (length (basis p1 i))))  ==>

1
0
3
22
138
390
480
210
```

Lisp files concerned in this chapter

`simplicial-sets.lisp`, `delta.lisp`, `specials-smsts.lisp`, `cartesian-products.lisp`.
[`classes.lisp`, `macros.lisp`, `various.lisp`].

Chapter 8

Simplicial morphisms

The software Kenzo implements simplicial morphisms in a way analogous to chain complex morphisms.

8.1 Representation of a simplicial morphism

A simplicial morphism is an instance of the class SIMPLICIAL-MRPH, subclass of the class MORPHISM.

```
(DEFCLASS SIMPLICIAL-MRPH (morphism)
  ((sintr :type sintr :initarg :sintr :reader sintr)))
```

It has one slot of its own:

sintr, an object of type SINTR, in fact a lisp function defining the morphism between the source and target simplicial sets. It must have 2 parameters: a dimension (an integer) and a geometric simplex of this dimension (a generator of any type) . It must return an abstract simplex, image in the target simplicial set of this geometric simplex.

A printing method has been associated to the class SIMPLICIAL-MRPH. A string like [Kn Simplicial-Morphism] is the external representation of an instance of such a class, where n is the number plate of the Kenzo object.

8.2 The function build-smmr

To facilitate the construction of instances of the class SIMPLICIAL-MRPH and to free the user to call the standard constructor make-instance, the software provides the function

`build-smmr` `:sorc` *sorc* `:trgt` *trgt* `:degr` *degr* `:sintr` *sintr* `:intr` *intr*
`:strt` *strt* `:orgn` *orgn*

defined with keyword parameters and returning an instance of the class `SIMPLICIAL-MRPH`. The keyword arguments of `build-smmr` are:

- *sorc*, the source object, an object of type `SIMPLICIAL-SET`.
- *trgt*, the target object, an object of type `SIMPLICIAL-SET`.
- *degr*, the degree of the morphism, an integer. In this chapter, we consider only the 0 degree case (the usual one). The case -1 is particularly important: it allows to implement the notion of twisting operator defining a fibration (See the chapter Fibration).
- *sintr*, the internal lisp function defining the effective mapping between simplicial sets. If the integer *degr* is 0 and if the following keyword argument *intr* is omitted, then the function `build-smmr` builds a lisp function implementing the induced mapping between the underlying source and target chain complexes. This function is installed in the slot `intr`. The strategy is then set to `:gnrt`.
- *intr*, a user defined morphism for the underlying chain complexes. This argument is optional and taken in account only if the degree is 0 and in this case, supersedes the previous derived mapping. The strategy is then mandatory. If the degree is not null, the implementor has decided to set the corresponding slot to `NIL`.
- *strt*, the strategy, i.e. `:gnrt` or `:cmbn` attached to the previous function.
- *orgn*, a relevant comment list.

After a call to `build-smmr`, the simplicial morphism instance is added to a list of previously constructed ones (`*smmr-list*`). As the other similar lists, the list `*smmr-list*` may be cleared by the function `cat-init`.
The effective application of a simplicial morphism upon arguments, is realized with the macro `?` which calls the adequate method defined for this kind of objects.

`?` `&rest` *args* *[Macro]*

Versatile macro for applying a simplicial morphism indifferently as (`?` *smmr dmns absm-or-gmsm*) or (`?` *smmr cmbn*). In the first case, the third argument is either an abstract simplex or a geometric one. In the second case, if the second argument is a combination, *smmr* is then considered as a chain complex morphism and it is the function in the slot `intr` which is applied, so that it makes sense only if the degree of the mapping is 0.

Examples

In the following examples, we work with Δ_3. We define three simplicial morphisms, `sm1`, `sm2` and `sm3`. In `sm1`, the mapping is the identity mapping and we can see that this identity mapping has been propagated on the underlying chain complex.

```
(setf d3 (delta 3))  ==>

[K1 Simplicial-Set]

(setf sm1 (build-smmr
            :sorc d3 :trgt d3 :degr 0
            :sintr #'(lambda (dmns gmsm)
                       (absm 0 gmsm))
            :orgn '(identity delta-3)))   ==>

[K6 Simplicial-Morphism]

(? sm1 2 7)  ==>

<AbSm - 7>

(? sm1 1 (absm 1 1))  ==>

<AbSm 0 1>

(? sm1 (cmbn 2 1 7))  ==>

------------------------------------------------------------------------{CMBN 2}
<1 * 7>
-------------------------------------------------------------------------------
```

In the simplicial morphism `sm2`, the mapping is the "null" mapping. In fact, it consists in applying any abstract simplex in dimension n upon the n–degenerate base point. Of course, in terms of chain complex, the corresponding mapping applies any chain complex element upon the null combination of same degree.

```
(setf sm2 (build-smmr
            :sorc d3 :trgt d3 :degr 0
            :sintr #'(lambda (dmns gmsm)
                       (absm (mask dmns) 1))    ;;; mask(n)=2^n - 1
            :orgn '(null delta-3)))  ==>

[K7 Simplicial-Morphism]

(? sm2 0 4)  ==>

<AbSm - 1>

(? sm2 2 7)  ==>

<AbSm 1-0 1>

(? sm2 1 (absm 1 1))  ==>

<AbSm 0 1>

(? sm2 3 15)  ==>

<AbSm 2-1-0 1>

(? sm2 (cmbn 3 2 15))  ==>

------------------------------------------------------------------------{CMBN 3}
-------------------------------------------------------------------------------
```

In the simplicial morphism sm3, we keep the same "null" mapping as in sm2 but we choose as mapping for the underlying chain complex the opposite of any chain complex element.

```
(setf sm3 (build-smmr
            :sorc d3 :trgt d3 :degr 0
            :sintr #'(lambda (dmns gmsm)
                       (absm (mask dmns) 1))
            :intr #'cmbn-opps
            :strt :cmbn
            :orgn '(null and opposite delta-3)))  ==>

[K8 Simplicial-Morphism]

(? sm3 3 15)  ==>

<AbSm 2-1-0 1>

(? sm3 (cmbn 3 7 15))  ==>
```

```
-------------------------------------------------------------------------{CMBN 3}
<-7 * 15>
-------------------------------------------------------------------------------

(? sm3 (cmbn 3 11 (dlop-ext-int '(0 1 2 3))))  ==>

-------------------------------------------------------------------------{CMBN 3}
<-11 * 15>
-------------------------------------------------------------------------------
```

Lisp files concerned in this chapter

`simplicial-mrphs.lisp`.
[`classes.lisp`, `macros.lisp`, `various.lisp`].

Chapter 9

The Eilenberg-Zilber module

The functions of this module are related to the chain complex of the product of two simplicial sets. They implement the Eilenberg-Mac Lane homomorphism (function `eml`) and the Alexander-Whitney homomorphism (function `aw`).

`eml` *ssx ssy* *[Function]*

build the Eilenberg-Mac Lane homomorphism $\mathcal{EML}$, also called the *shuffle homomorphism*[1]. The arguments *ssx* and *ssy* represent some simplicial sets X and Y. The homomorphism

$$\mathcal{EML} : \mathcal{C}_*(X) \otimes \mathcal{C}_*(Y) \longrightarrow \mathcal{C}_*(X \times Y)$$

is defined by:

$$\mathcal{EML}(\alpha \otimes \beta) = \sum \varepsilon(\varpi)(\eta_{j_q} \ldots \eta_{j_1}\alpha,\ \eta_{i_p} \ldots \eta_{i_1}\beta),$$

where $X \times Y$ is the simplicial cartesian product of the two simplicial sets X and Y, α (resp. β) is a geometric simplex of X (resp. Y), η_k is the k^{th} degeneracy operator, the sum is over all permutations $\varpi = (i_1 \ldots i_p j_1 \ldots j_q)$ of $(0 \ldots p+q-1)$ such that

$$i_1 < i_2 < \cdots < i_p, \quad j_1 < j_2 < \cdots < j_q, \quad p+q = n$$

and $\varepsilon(\varpi)$ is the signature of the corresponding permutation. The sum at the right hand part determines a decomposition of the geometric product $\alpha \times \beta$ into a combination of generators of the simplicial cartesian product (see the simplicial sets chapter). The combination takes in account the orientation of every simplex. These

[1] **Marvin Greenberg**, *Lectures on Algebraic Topology*, W.A. Benjamin Inc, 1967.

generators are non-degenerate simplices of the cartesian product but their projections are in general degenerate simplices of X and Y (see the examples).

`aw` *ssx ssy* *[Function]*

build the Alexander-Whitney chain homomorphism. The arguments *ssx* and *ssy* represent some simplicial sets X and Y. The homomorphism

$$\mathcal{AW} : \mathcal{C}_*(X \times Y) \longrightarrow \mathcal{C}_*(X) \otimes \mathcal{C}_*(Y)$$

is defined by the following rule: let σ be an n-simplex and let us define two operators, λ_p and ρ_q:

$$\lambda_p : \mathcal{C}_n(X) \longrightarrow \mathcal{C}_{n-p}(X), \quad (n \geq p),$$

$$\rho_q : \mathcal{C}_n(Y) \longrightarrow \mathcal{C}_{n-q}(X), \quad (n \geq q),$$

respectively by

$$\lambda_p(\sigma) = \partial_{p+1} \ldots \partial_n \sigma$$

and

$$\rho_q(\sigma) = \partial_0 \ldots \partial_{n-q} \sigma.$$

Now, for the generator (α, β) of degree n in $\mathcal{C}_*(X \times Y)$, the Alexander-Whitney homomorphism is defined by:

$$\mathcal{AW}(\alpha, \beta) = \sum_{p=0}^{n} \lambda_p(\alpha) \otimes \rho_{n-p}(\beta).$$

This sum can be interpreted as a decomposition, up to a homotopy, of the simplex $\alpha \times \beta$ (α **and** β can be degenerate even if $\alpha \times \beta$ is not) in terms of (tensor) products of non-degenerate simplices.

`phi` *ssx ssy* *[Function]*

build the homotopy chain morphism

$$\Phi : \mathcal{C}_*(X \times Y) \longrightarrow \mathcal{C}_*(X \times Y)$$

satisfying the formula

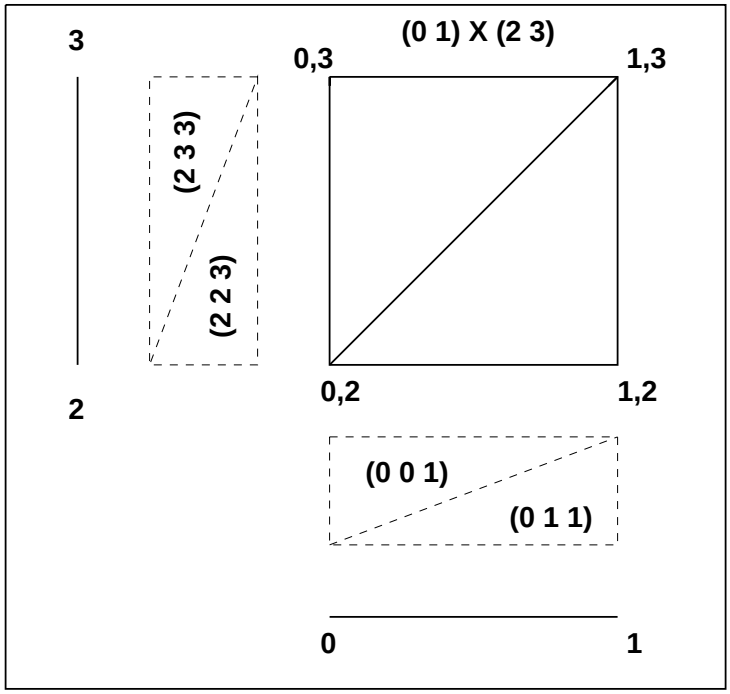
$$d \circ \Phi + \Phi \circ d = Id - \mathcal{EML} \circ \mathcal{AW}.$$

`ez` *ssx ssy* *[Function]*

build the Eilenberg–Zilber reduction as shown by the following diagram, where $X = ssx$ and $Y = ssy$.

$$\begin{array}{ccc} \mathcal{C}_*(X \times Y) & \xrightarrow{\Phi} & {}^s\mathcal{C}_*(X \times Y) \\ \mathcal{AW} \downarrow\uparrow \mathcal{EML} & & \\ \mathcal{C}_*(X) \otimes \mathcal{C}_*(Y) & & \end{array}$$

Examples



As simplicial sets X and Y, let us use two copies of $\Delta^{\mathbb{N}}$, that we may build by the lisp statement `(soft-delta-infinity)` (Though we repeat this statement in the following line, there will be only a simplicial set created). We have chosen the *soft* version, to inspect the results more easily.

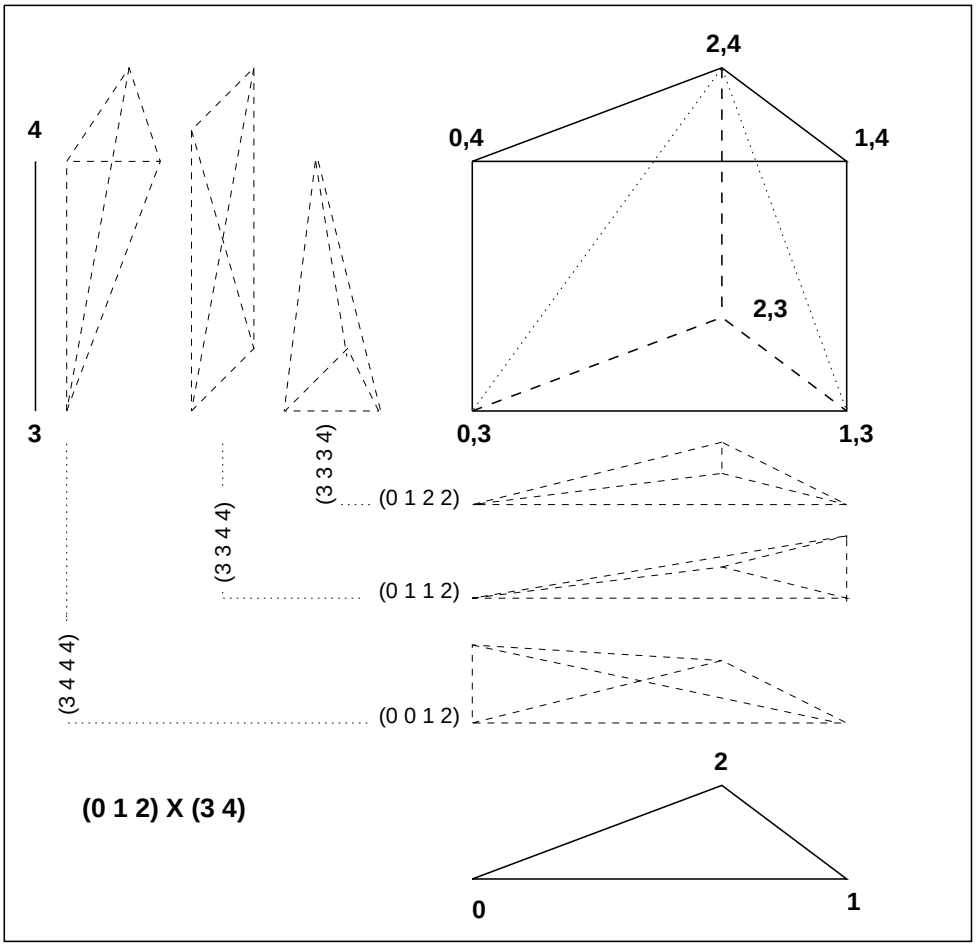
```
(setf eml-mrp (eml (soft-delta-infinity)(soft-delta-infinity)))  ==>

[K11 Morphism (degree 0)]
```

Let us apply the Eilenberg-Mac Lane homomorphism to the tensor product $(0\,1) \otimes (2\,3)$:

```
(? eml-mrp 2 (tnpr 1 (d(dlop-ext-int '(0 1)))
                   1 (d(dlop-ext-int '(2 3)))))   ==>

------------------------------------------------------------------------{CMBN 2}
<-1 * <CrPr 0 0-1 1 2-3>>
<1 * <CrPr 1 0-1 0 2-3>>
------------------------------------------------------------------------------
```

The decomposition is illustrated by the previous geometric diagram, where for instance a degenerate simplex like $\eta_1(0\,1)$ is noted $(0\,1\,1)$.
For the tensor product $(0\,1\,2)\otimes(3\,4)$, an intuitive description is still possible, as shown in the following diagram, but not in higher dimensions.
As the input of simplices is sometimes cumbersome, we have defined a macro called `code` to facilitate the input of the simplices.

```
(defmacro code (arg) `(d (dlop-ext-int ,arg)))  ==>

CODE

(? eml-mrp 3 (tnpr 2 (code '(0 1 2)) 1 (code '(3 4)) ))  ==>

-------------------------------------------------------------------------{CMBN 3}
<1 * <CrPr 0 0-1-2 2-1 3-4>>
<-1 * <CrPr 1 0-1-2 2-0 3-4>>
<1 * <CrPr 2 0-1-2 1-0 3-4>>
-------------------------------------------------------------------------------

(? eml-mrp 6 (tnpr 2 (code '(0 3 5)) 4 (code '(0 1 2 4 5)) ))  ==>

-------------------------------------------------------------------------{CMBN 6}
<1 * <CrPr 3-2-1-0 0-3-5 5-4 0-1-2-4-5>>
<-1 * <CrPr 4-2-1-0 0-3-5 5-3 0-1-2-4-5>>
<1 * <CrPr 4-3-1-0 0-3-5 5-2 0-1-2-4-5>>
<-1 * <CrPr 4-3-2-0 0-3-5 5-1 0-1-2-4-5>>
<1 * <CrPr 4-3-2-1 0-3-5 5-0 0-1-2-4-5>>
<1 * <CrPr 5-2-1-0 0-3-5 4-3 0-1-2-4-5>>
<-1 * <CrPr 5-3-1-0 0-3-5 4-2 0-1-2-4-5>>
<1 * <CrPr 5-3-2-0 0-3-5 4-1 0-1-2-4-5>>
<-1 * <CrPr 5-3-2-1 0-3-5 4-0 0-1-2-4-5>>
<1 * <CrPr 5-4-1-0 0-3-5 3-2 0-1-2-4-5>>
<-1 * <CrPr 5-4-2-0 0-3-5 3-1 0-1-2-4-5>>
<1 * <CrPr 5-4-2-1 0-3-5 3-0 0-1-2-4-5>>
<1 * <CrPr 5-4-3-0 0-3-5 2-1 0-1-2-4-5>>
<-1 * <CrPr 5-4-3-1 0-3-5 2-0 0-1-2-4-5>>
<1 * <CrPr 5-4-3-2 0-3-5 1-0 0-1-2-4-5>>
-------------------------------------------------------------------------------
```



The Alexander-Whitney homomorphism works in the reverse sense:

```
(setf aw-mrp (aw (soft-delta-infinity)(soft-delta-infinity)))  ==>

[K12 Morphism (degree 0)]
```

The 2-simplices of the cartesian product $(0\,1\,2) \times (3\,4)$ are (here, the degeneracy operators are in clear):

```
<CrPr 0 (0 1) 1 (3 4)>
<CrPr 1 (0 1) 0 (3 4)>
<CrPr 0 (0 2) 1 (3 4)>
<CrPr 1 (0 2) 0 (3 4)>
<CrPr 0 (1 2) 1 (3 4)>
<CrPr 1 (1 2) 0 (3 4)>
```

```
      <CrPr * (0 1 2) 1-0 (3)>
      <CrPr * (0 1 2) 1-0 (4)>
 -->  <CrPr * (0 1 2) 0 (3 4)>
      <CrPr * (0 1 2) 1 (3 4)>
```

The last but one line of this list describes the triangle shown in dashed lines in the prism in the previous picture, joining the points $(0,3),(1,3),(2,4)$. Let us apply the homomorphism $\mathcal{AW}$:

```
(? aw-mrp 2 (crpr 0 (code '(0 1 2)) 1 (code '(3 4)) ))  ==>

-------------------------------------------------------------------------{CMBN 2}
<1 * <TnPr 0-1 3-4>>
<1 * <TnPr 0-1-2 4>>
------------------------------------------------------------------------------
```

We see that we obtain a kind of decomposition, up to a homotopy, of this triangle in terms of the two tensor products $(0\,1\,2)\otimes(4)$ (the "upper" triangle of the prism) and $(0\,1)\otimes(3\,4)$ (the "front" rectangle of the prism).

The composition $\mathcal{AW}\circ\mathcal{EML}$ is the identity in the tensor product of the two chain complexes.

```
(setf g (tnpr 2 (code '(0 1 2)) 3 (code '(2 3 4 5)) ))  ==>

<TnPr 0-1-2 2-3-4-5>

(? eml-mrp 5 g)  ==>

-------------------------------------------------------------------------{CMBN 5}
<1 * <CrPr 2-1-0 0-1-2 4-3 2-3-4-5>>
<-1 * <CrPr 3-1-0 0-1-2 4-2 2-3-4-5>>
<1 * <CrPr 3-2-0 0-1-2 4-1 2-3-4-5>>
<-1 * <CrPr 3-2-1 0-1-2 4-0 2-3-4-5>>
<1 * <CrPr 4-1-0 0-1-2 3-2 2-3-4-5>>
<-1 * <CrPr 4-2-0 0-1-2 3-1 2-3-4-5>>
<1 * <CrPr 4-2-1 0-1-2 3-0 2-3-4-5>>
<1 * <CrPr 4-3-0 0-1-2 2-1 2-3-4-5>>
<-1 * <CrPr 4-3-1 0-1-2 2-0 2-3-4-5>>
<1 * <CrPr 4-3-2 0-1-2 1-0 2-3-4-5>>
------------------------------------------------------------------------------

(? aw-mrp *)

-------------------------------------------------------------------------{CMBN 5}
<1 * <TnPr 0-1-2 2-3-4-5>>
------------------------------------------------------------------------------
```

```
(setf *tnpr-with-degrees* t)  ==>

T

**

------------------------------------------------------------------------{CMBN 5}
<1 * <TnPr 2 0-1-2 3 2-3-4-5>>
--------------------------------------------------------------------------------
```

But the composition $\mathcal{EML} \circ \mathcal{AW}$ is by no means the identity. This composition is related to the identity via the homotopy morphism Φ. Let us consider the two following simple simplicial sets `delta-0-1` and `delta-2-3`, where the second is the simplicial set, model of the segment [2, 3]. To build `delta-2-3`, we have just changed the basis function, the base point and the comment in the system function `soft-delta`.

```
(setf delta-0-1 (soft-delta 1))

[K13 Simplicial-Set]

(setf delta-2-3 (build-smst
                  :cmpr #'soft-delta-cmpr
                  :basis #'(lambda(dmn)
                             (case dmn
                                  (0 (list (d 4)(d 8)))
                                  (1 (list(d 12)))))
                  :bspn (d 4)
                  :face #'soft-delta-face
                  :intr-dgnl #'soft-delta-dgnl :dgnl-strt :gnrt
                  :intr-bndr #'soft-delta-bndr :bndr-strt :gnrt
                  :orgn '(my-soft-delta-2-3)))

[K18 Simplicial-Set]

(show-structure delta-0-1 1)  ==>

Dimension = 0 :

        Vertices :  (0 1)

Dimension = 1 :

        Simplex : 0-1

                Faces : (<AbSm - 1> <AbSm - 0>)

(show-structure delta-2-3 1)  ==>
```

```
Dimension = 0 :

        Vertices :  (2 3)

Dimension = 1 :

        Simplex : 2-3

                Faces : (<AbSm - 3> <AbSm - 2>)
```

Let us build the cartesian product of these two simplicial sets and let us list, in particular the 0-simplices and the 1-simplices of this new simplicial set:

```
(setf carre (crts-prdc delta-0-1 delta-2-3))  ==>

[K23 Simplicial-Set]

(show-structure carre 2)  ==>


Dimension = 0 :

        Vertices :  (<CrPr - 0 - 2> <CrPr - 0 - 3> <CrPr - 1 - 2>
                     <CrPr - 1 - 3>)

Dimension = 1 :

        Simplex : <CrPr - 0-1 - 2-3>

                Faces : (<AbSm - <CrPr - 1 - 3>> <AbSm - <CrPr - 0 - 2>>)

        Simplex : <CrPr - 0-1 0 2>

                Faces : (<AbSm - <CrPr - 1 - 2>> <AbSm - <CrPr - 0 - 2>>)

        Simplex : <CrPr - 0-1 0 3>

                Faces : (<AbSm - <CrPr - 1 - 3>> <AbSm - <CrPr - 0 - 3>>)

        Simplex : <CrPr 0 0 - 2-3>

                Faces : (<AbSm - <CrPr - 0 - 3>> <AbSm - <CrPr - 0 - 2>>)

        Simplex : <CrPr 0 1 - 2-3>

                Faces : (<AbSm - <CrPr - 1 - 3>> <AbSm - <CrPr - 1 - 2>>)
```

```
Dimension = 2 :

        Simplex : <CrPr 0 0-1 1 2-3>

                Faces : (<AbSm - <CrPr - 0-1 0 3>>
                         <AbSm - <CrPr - 0-1 - 2-3>>
                         <AbSm - <CrPr 0 0 - 2-3>>)

        Simplex : <CrPr 1 0-1 0 2-3>

                Faces : (<AbSm - <CrPr 0 1 - 2-3>>
                         <AbSm - <CrPr - 0-1 - 2-3>>
                         <AbSm - <CrPr - 0-1 0 2>>)
```

Let us take the first 1-simplex which corresponds to the diagonal of the square (realisation of the cartesian product) and let us build the three homomorphisms $\mathcal{AW}$, $\mathcal{EML}$ and Φ:

```
(setf diagonal (crpr 0 (code '(0 1)) 0 (code '(2 3))))  ==>

<CrPr - 0-1 - 2-3>

(setf aw-mrp (aw delta-0-1 delta-2-3))  ==>

[K30 Morphism (degree 0)]

(setf eml-mrp (eml delta-0-1 delta-2-3))  ==>

[K31 Morphism (degree 0)]

(setf phi-hmy (phi delta-0-1 delta-2-3))  ==>

[K32 Morphism (degree 1)]
```

We see that

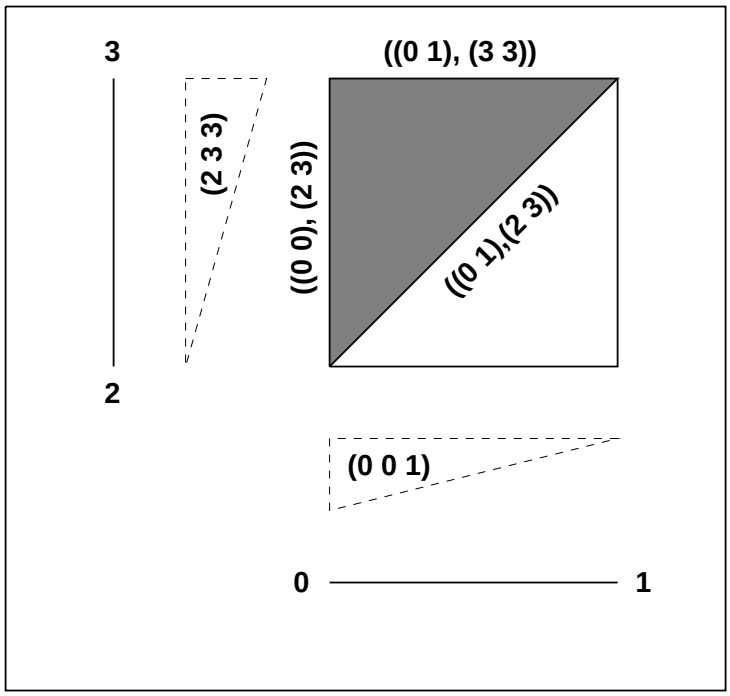
$$\mathcal{AW}((0\,1),(2\,3)) = (0\,1)\otimes(3) + (0)\otimes(2\,3),$$

$$\mathcal{EML}((0\,1)\otimes(3) + (0)\otimes(2\,3)) = ((0\,1),\eta_0(3)) + (\eta_0(0),(2\,3))$$

and that Φ applied to the diagonal returns the 2-simplex compatible with the homotopy.

$$\Phi((0\,1),(2\,3)) = (\eta_1(2\,3),\eta_0(0\,1)).$$

This is illustrated by the following diagram (the 2-simplex is the shaded triangle).




```
(? aw-mrp 1 diagonal)  ==>

--------------------------------------------------------------------------{CMBN 1}
<1 * <TnPr 0 0 1 2-3>>
<1 * <TnPr 1 0-1 0 3>>
-------------------------------------------------------------------------------

(? eml-mrp *)  ==>

--------------------------------------------------------------------------{CMBN 1}
<1 * <CrPr - 0-1 0 3>>
<1 * <CrPr 0 0 - 2-3>>
-------------------------------------------------------------------------------

(? phi-hmy 1 diagonal)  ==>

--------------------------------------------------------------------------{CMBN 2}
<-1 * <CrPr 0 0-1 1 2-3>>
-------------------------------------------------------------------------------
```

Let us inspect now the reduction generated by the function `ez` and verify on reasonably general combinations that the involved morphisms are coherent.

```
(setf ez-rdc (ez (soft-delta-infinity)(soft-delta-infinity)))  ==>
```

```
[K34 Reduction]

(inspect *)  ==>

REDUCTION @ #x36a55a = [K34 Reduction]
   0 Class --------> #<STANDARD-CLASS REDUCTION>
   1 ORGN ---------> <...>, a proper list with 3 elements
   2 IDNM ---------> fixnum 34 [#x00000088]
   3 H ------------> [K33 Morphism (degree 1)]
   4 G ------------> [K11 Morphism (degree 0)]
   5 F ------------> [K12 Morphism (degree 0)]
   6 BCC ----------> [K3 Chain-Complex]
   7 TCC ----------> [K6 Simplicial-Set]
```

With the comment slots we see that the morphism f, g and h are respectively *aw*, *eml* and *phi*:

```
(orgn ez-rdc)  ==>

(EILENBERG-ZILBER [K1 Simplicial-Set][K1 Simplicial-Set])

(orgn (f ez-rdc)) ==>

(AW [K1 Simplicial-Set][K1 Simplicial-Set])

(orgn (g ez-rdc))  ==>

(EML [K1 Simplicial-Set][K1 Simplicial-Set])

(orgn (h ez-rdc))  ==>

(PHI [K1 Simplicial-Set][K1 Simplicial-Set])

(setf *bc* (cmbn 3 1 (tnpr 0 (code '(0)) 3 (code '(1 2 3 4)))
                  10 (tnpr 1 (code '(5 6)) 2 (code '(7 8 9)))
                 100 (tnpr 2 (code '(10 11 12)) 1 (code '(13 14)))
                1000 (tnpr 3 (code '(15 16 17 18)) 0 (code '(19)))))  ==>

--------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 0 1-2-3-4>>
<10 * <TnPr 5-6 7-8-9>>
<100 * <TnPr 10-11-12 13-14>>
<1000 * <TnPr 15-16-17-18 19>>
--------------------------------------------------------------------------
```

```
(setf *tc*
     (cmbn 3 1 (crpr 0 (code '(0 1 2 3)) 0 (code '(5 6 7 8)))
           10 (crpr 0 (code '(0 1 2 3)) (dgop-ext-int '(2 0)) (code '(5 6)))
          100 (crpr (dgop-ext-int '(2 1)) (code '(0 1)) 0 (code '(5 6 7 8)))
         1000 (crpr (dgop-ext-int '(2 1)) (code '(0 1)) 1 (code '(5 6 7)))))

--------------------------------------------------------------------------{CMBN 3}
<1 * <CrPr - 0-1-2-3 - 5-6-7-8>>
<10 * <CrPr - 0-1-2-3 2-0 5-6>>
<100 * <CrPr 2-1 0-1 - 5-6-7-8>>
<1000 * <CrPr 2-1 0-1 0 5-6-7>>
------------------------------------------------------------------------------

(pre-check-rdct ez-rdc)  ==>

---done---

(check-rdct)  ==>

*TC* =>
--------------------------------------------------------------------------{CMBN 3}
<1 * <CrPr - 0-1-2-3 - 5-6-7-8>>
<10 * <CrPr - 0-1-2-3 2-0 5-6>>
<100 * <CrPr 2-1 0-1 - 5-6-7-8>>
<1000 * <CrPr 2-1 0-1 0 5-6-7>>
------------------------------------------------------------------------------

*BC* =>
--------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 0 1-2-3-4>>
<10 * <TnPr 5-6 7-8-9>>
<100 * <TnPr 10-11-12 13-14>>
<1000 * <TnPr 15-16-17-18 19>>
------------------------------------------------------------------------------

Checking *TDD* = 0
Result:
--------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

Checking *BDD* = 0
Result:
--------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

Checking *DF-FD* = 0
Result:
```

```
------------------------------------------------------------------------{CMBN 2}
-------------------------------------------------------------------------------

Checking *DG-GD* = 0
Result:
------------------------------------------------------------------------{CMBN 2}
-------------------------------------------------------------------------------

Checking *ID-FG* = 0
Result:
------------------------------------------------------------------------{CMBN 3}
-------------------------------------------------------------------------------

Checking *ID-GF-DH-HD* = 0
Result:
------------------------------------------------------------------------{CMBN 3}
-------------------------------------------------------------------------------

Checking *HH* = 0
Result:
------------------------------------------------------------------------{CMBN 5}
-------------------------------------------------------------------------------

Checking *FH* = 0
Result:
------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

Checking *HG* = 0
Result:
------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------

---done---
```

9.1 Application to Homology

Let us illustrate the Eilenberg-Zilber theorem by the following example. Let us consider the manifold $P^2\mathbb{R} \times S^3$. The theorem says that

$$H_n(\mathcal{C}_*(P^2\mathbb{R} \times S^3)) \cong H_n(\mathcal{C}_*(P^2\mathbb{R}) \otimes \mathcal{C}_*(S^3)).$$

```
(setf p2 (moore 2 1))  ==>      ;; Moore(2,1)   generates the projectif plane.

[K1 Simplicial-Set]
```

```
(setf s3 (sphere 3))  ==>

[K6 Simplicial-Set]
```

The simplicial set `p2-X-s3` is obtained from the cartesian product of the two simplicial sets `p2` and `s3`.

```
(setf p2-X-s3 (crts-prdc p2 s3))  ==>

[K11 Simplicial-Set]
```

Now, the chain complex `p2-T-s3` is obtained from the tensor product of the two chain complexes associated to both simplicial sets `p2` and `s3` (don't forget that the class `SIMPLICIAL SET` is a subclass of the class `CHAIN COMPLEX`).

```
(setf p2-T-s3 (tnsr-prdc p2 s3))  ==>

[K16 Chain-Complex]
```

Applying successively the function `chcm-homology` (which computes directly the homology groups without using a homotopy equivalence) on these chain complexes, shows that the homology groups are effectively isomorphic, but the method of the tensor product is much faster than the method of the simple cartesian product, the number of generators being much more smaller.

```
(time(dotimes (i 6)(chcm-homology p2-X-s3 i))) ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

Component Z/2Z

Homology in dimension 5 :

---done---
```

```
; cpu time (non-gc) 880 msec user, 170 msec system
; cpu time (gc)     250 msec user, 0 msec system
; cpu time (total)  1,130 msec user, 170 msec system
; real time  2,563 msec
; space allocation:
;  43,812 cons cells, 64 symbols, 162,904 other bytes

(time(dotimes (i 6)(chcm-homology p2-T-s3 i)))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

Component Z/2Z

Homology in dimension 5 :

---done---

; cpu time (non-gc) 130 msec user, 20 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  130 msec user, 20 msec system
; real time  326 msec
; space allocation:
;  1,897 cons cells, 0 symbols, 11,232 other bytes
```

The following example shows clearly the discrepancy between the length of the basis of two objects built respectively by cartesian product and tensor product and having the same homology groups.

```
(setf s2 (sphere 2))  ==>

[K1 Simplicial-Set]

(setf s3 (sphere 3))  ==>
```

```
[K6 Simplicial-Set]

(setf s2Xs2Xs3 (crts-prdc (crts-prdc s2 s2) s3))  ==>

[K16 Simplicial-Set]

(orgn s2Xs2Xs3)  ==>

(CRTS-PRDC [K23 Simplicial-Set] [K6 Simplicial-Set])

(setf s2Ts2Ts3 (tnsr-prdc (tnsr-prdc s2 s2) s3))   ==>

[K21 Chain-Complex]

(dotimes (i 7) (print(length(basis s2Xs2Xs3 i))))  ==>

1
0
3
22
138
390
480

(dotimes (i 7) (print(length(basis s2Ts2Ts3 i))))  ==>

1
0
2
1
1
2
0
```

9.1.1 Searching homology process for cartesian products

When the `search-efhm` method recognizes a cartesian product, by means of the comment list (slot `orgn`) of the object, it builds a homotopy equivalence where the right bottom chain complex is created by the function `ez`. The process may be recursif as shown by the very definition of the method:

```
(defun LEFT-CRTS-PRDC-EFHM (smst1 smst2)
  (declare (type simplicial-set smst1 smst2))
  (the homotopy-equivalence
     (build-hmeq
       :lrdct (trivial-rdct (crts-prdc smst1 smst2))
       :rrdct (ez smst1 smst2))))

(defmethod SEARCH-EFHM (smst (orgn (eql 'crts-prdc)))
  (declare (type simplicial-set smst))
  (the homotopy-equivalence
    (cmps
      (left-crts-prdc-efhm (second (orgn smst))
                           (third  (orgn smst)))
      (tnsr-prdc (efhm (second (orgn smst)))
                 (efhm (third  (orgn smst)))))))
```

Lisp files concerned in this chapter

`eilenberg-zilber.lisp`, `searching-homology.lisp`.

Chapter 10

Programming the Kan theory

10.1 Introduction

Let us recall some definitions in relation with the Kan simplicial sets.

Definition 1. A Kan "hat" of dimension n and index i is a collection of n simplices of dimension $(n-1)$, noted $(\sigma_0, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_n)$, such that the following conditions are fulfilled:

$$\partial_j \sigma_k = \partial_k \sigma_{j+1}, \quad k \neq i, \quad j{+}1 \neq i, \quad 0 \leq j \leq n{-}1, \quad 0 \leq k \leq n{-}1, \quad k \leq j.$$

Note that there is no σ_i term in the list above; in a sense, the Kan process consists in constructing the missing σ_i to be considered as a "composition" of the given σ_j's.

Example

We are going to work with the simplices of $\Delta^{\mathbb{N}}$. The set (σ_1, σ_2) = `((0 2), (0 1))` is a Kan hat of dimension 2 and index 0. In effect, $\delta_1\sigma_1 = \delta_1\sigma_2 =$ `(0)`. Likewise, the two sets `((1 2), (0 1))` and `((1 2), (0 2))` are Kan hats of dimension 2 and of respective index 1 and 2. Of course, this definition applies equally if the simplices are degenerate.

Definition 2. A *filling* of a Kan hat, $(\sigma_0, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_n)$, is an n–simplex $\tilde{\sigma}$, such that for every $j \neq i$, $\partial_j \tilde{\sigma} = \sigma_j$.

Example

A filling of `((0 2), (0 1))` is the 2-simplex `(0 1 2)`. A filling of the Kan hat `((1 1), (0 1))` is the degenerate 2-simplex `(0 1 1)`. A filling of the vertex `(0)` is the degenerate 1–simplex `(0 0)`.

Definition 3. A simplicial set is a Kan simplicial set if for every hat there exists a filling.

10.1.1 Representation of a Kan simplicial set

A Kan simplicial set is implemented as an instance of the class `KAN`, subclass of the class `SIMPLICIAL-SET`.

```
(DEFCLASS KAN (simplicial-set)
  ((kfll :type kfll :initarg :kfll :reader kfll1)))
```

We recall that this new class inherits also from the class `CHAIN-COMPLEX`. It has one slot of its own:

`kfll`, an object of type `KFLL`, in fact a lisp function with 3 parameters describing a Kan hat: an index (an integer), a dimension (an integer) and the Kan hat (a list of abstract simplices). This function must return a filling of the Kan hat argument, i.e. an abstract simplex satisfying the theoritical definition.

A printing method has been associated to the class `KAN` and the external representation of an instance is a string like `[K`*n* `Kan-Simplicial-Set]`, where *n* is the number plate of the Kenzo object.

10.1.2 Helpful functions on Kan simplicial sets

`cat-init` *[Function]*

Clear in particular `*Kan-list*`, the list of user created Kan simplicial sets and reset the global counter to 1.

`Kan` *n* *[Function]*

Retrieve in the list `*Kan-list*` the Kan object instance whose the Kenzo identification is *n*. If it does not exist, return `NIL`.

`kfll &rest` *args* *[Macro]*

With only one argument (a Kan instance) return the slot `kfll` of this instance. With 4 arguments like (`kfll` *kan indx dmns hat*), return a filling of the Kan hat *hat*, of dimension *dmns* and of index *indx* by applying the filling function value of the slot `kfll` of the Kan instance *kan*.

`smst-kan` *smst kfll* *[Function]*

With the filling lisp function *kfll*, **transform** the simplicial set *smst* in an object of type `KAN` (in other word, *smst* is `modified`). This is the easiest way to build a Kan simplicial set.

`check-hat` *kan indx dmns hat* *[Function]*

Useful verification function to check if the collection of simplices *hat* is a valid Kan hat of dimension *dmns* and of index *indx* in the Kan simplicial set *kan*. In fact this works equally if *kan* is a general simplicial set.

`check-kan` *kan indx dmns hat* *[Function]*

Useful verification function to check if the collection of simplices *hat* is a valid Kan hat of dimension *dmns* and of index *indx* in the Kan simplicial set *kan*. This verification function applies the filling function of the instance *kan* to the argument *hat* and perform the verification of the faces relations upon the resulting *dmns*–simplex.

Examples

Let us take again the small examples of the introduction. First we define a function `dkfll`, a filling function suitable for a Kan hat in $\Delta^{\mathbb{N}}$. The user will note that in the abstract simplices the degeneracy operators and the geometric simplices are coded in binary.

```
(defun dkfll (indx dmns hat)
     (cond ((= 1 dmns)
            (absm 1 (gmsm (first hat))))
           ((= 0 indx)
            (let ((del-1 (absm-int-ext (first hat)))
                  (del-2 (absm-int-ext (second hat))))
              (absm-ext-int
                 (cons (first del-1)
                       (cons (second del-2) (rest del-1))))))
           ((= 1 indx)
            (let ((del-0 (absm-int-ext (first hat)))
                  (del-2 (absm-int-ext (second hat))))
              (absm-ext-int
                 (cons (first del-2) del-0))))
           (t
            (let ((del-0 (absm-int-ext (first hat)))
                  (del-1 (absm-int-ext (second hat))))
              (absm-ext-int
                 (cons (first del-1) del-0))))))

DKFLL
```

```
(setf d (delta-infinity))  ==>

[K5 Simplicial-Set]

(smst-kan d #'dkfll)  ==>

[K5 Kan-Simplicial-Set]

(kfll d 0 1 (list (absm 0 1)))  ==>

<AbSm 0 1>

(kfll d 0 2 (list (absm 0 5) (absm 0 3)))  ==>

<AbSm - 7>

(kfll d 1 2 (list (absm 0 6) (absm 0 3)))  ==>

<AbSm - 7>

(kfll d 2 2 (list (absm 0 6) (absm 0 5)))  ==>

<AbSm - 7>

(kfll d 0 2 (list (absm 0 3) (absm 0 3)))  ==>

<AbSm 1 3>

(kfll d 1 2 (list (absm 1 2) (absm 0 3)))  ==>

<AbSm 1 3>

(check-hat d 1 2 (list (absm 1 2) (absm 0 3)))  ==>

T

(check-hat d 0 1 (list (absm 0 1)))  ==>

T

(check-kan d 0 1 (list (absm 0 1)))  ==>

---done---
```

More elaborate examples with Kan simplicial sets will be given later in the loop spaces chapter.

Lisp files concerned in this chapter

`kan.lisp`.
[`classes.lisp` , `macros.lisp`, `various.lisp`].

Chapter 11

Simplicial Groups

11.1 Representation of a simplicial group

A simplicial group is an instance of the class `SIMPLICIAL-GROUP`, subclass of the classes `KAN` and `HOPF-ALGEBRA`.

```
(DEFCLASS SIMPLICIAL-GROUP (kan hopf-algebra)
   ((grml :type simplicial-mrph :initarg :grml :reader grml1)
    (grin :type simplicial-mrph :initarg :grin :reader grin1)))
```

We recall that this new class (multi–) inherits from the following classes: `KAN`, `SIMPLICIAL-SET`, `COALGEBRA`, `ALGEBRA` and `CHAIN-COMPLEX`. It has two slots of its own:

`grml`, an object of type `SIMPLICIAL-MRPH`, defining in the underlying simplicial set S, the group operation as a simplicial morphism from $S \times S$ onto S, compatible with the face and degeneracy operators in S, i.e. the ∂'s and η's operators are also group morphisms. In dimension n, the neutral element is assumed to be the n–th degeneracy of the base point of the underlying simplicial set.

`grin`, an object of type `SIMPLICIAL-MRPH`, defining the inverse of an element of the Kan simplicial set w.r.t. the preceding group law.

A printing method has been associated to the class `SIMPLICIAL GROUP` and the external representation of an instance is a string like `[K`n `Simplicial-Group]`, where n is the number plate of the Kenzo object.

11.2 Representation of an Abelian simplicial group

The class of Abelian simplicial group, `AB-SIMPLICIAL-GROUP`, inherits all the properties of the class `SIMPLICIAL-GROUP` and has not slots of its own, as shown by:

```
(DEFCLASS AB-SIMPLICIAL-GROUP (simplicial-group) ())
```

The difference is purely mathematical: it is up to the user to provide as slot *grml*, a simplicial morphism which is **commutative**. As expected, the external representation of an instance of this class, is a string like `[K`*n* `Abelian-Simplicial-Group]`, where *n* is the number plate of the Kenzo object.

11.3 The functions build-smgr and build-ab-smrg

To facilitate the construction of instances of the classes `SIMPLICIAL-GROUP` and `AB-SIMPLICIAL-GROUP` and to free the user to call the standard constructor `make-instance`, the software provides the functions *build-smrg* and *build-ab-smrg*.

`build-smgr`
`:cmpr` *cmpr* `basis` *basis* `:bspn` *bspn* `:face` *face* `:face*` *face** `:intr-bndr` *intr-bndr* `:bndr-strt` *bndr-strt* `:intr-dgnl` *intr-dgnl* `:dgnl-strt` *dgnl-strt* `:sintr-grml` *sintr-grml* `:sintr-grin` *sintr-grin* `:orgn` *orgn*

defined with keyword parameters and returning an instance of the class `SIMPLICIAL-GROUP`. The keyword arguments *cmpr*, *basis*, *bspn*, *face*, *face**, *intr-bndr*, *bndr-strt*, *intr-dgnl* and *dgnl-strt* are the arguments provided to build the underlying simplicial set using the function `build-smst`. Let us call it S. As usual, an adequate comment list must be provided for the parameter `:orgn`. The only two new arguments are:

- *sintr-grml*, a lisp function defining the group operation $S \times S \longrightarrow S$. The function `build-smgr` uses this lisp function to build a simplicial morphism of degree 0 between the cartesian product of the newly created simplicial set S and itself. This new simplicial morphism is the value of the slot `grml` of the simplicial group instance.

- *sintr-grin*, a lisp function defining the inverse w.r.t the preceding group law of an element of S. The function `build-smgr` uses this lisp function

to build a simplicial morphism of degree 0 between S and itself. This new simplicial morphism is the value of the slot `grin` of the simplicial group instance.

After a call to `build-smgr`, the simplicial group instance is pushed onto the list of previously constructed ones (`*smgr-list*`). The simplicial group with Kenzo identification n, may be retrieved in the `*smgr-list*`, by a call to the function `sgmr` as `(sgmr` n`)`. The list `*smgr-list*` may be cleared by the function `cat-init`.

Up to now the slot `kfll` has not been filled. As long as the user does not make use of the filling function, directly or indirectly, this slot remains unbound. But as soon as it is needed, the *slot-unbound* mecanism of CLOS enters in action and the filling function is defined by the system with the help of the functions `face`, `sintr-grml` and `sintr-grin`. The automatic creation of the filling function is possible because mathematically, a simplicial group is always a Kan simplicial set. This is realized by the internal function `smgr-kfll-intr`.
For the user, the two simplicial morphisms, values of the specific slots of a `SIMPLICIAL-GROUP` are applied via the two following macros:

`grml` *smgr dmns crpr* *[Macro]*
: Apply the simplicial morphism, value of the slot `grml` of the simplicial group instance *smgr* to the cartesian product *crpr* in dimension *dmns*. Mathematically, this realizes the group operation. In principle, the simplicial morphism must accept the *crpr* argument on both forms: either a geometric cartesian product or an abstract simplex, the geometric part of which is a geometric cartesian product (see the examples). With only one argument (*smgr*) the macro selects the simplicial morphism instance.

`grin` *smgr dmns absm-or-gsm* *[Macro]*
: Apply the simplicial morphism, value of the slot `grin` of the simplicial group instance *smgr* to the element *absm-or-gsm* in dimension *dmns*. Mathematically this gives the inverse in the group of the element. As suggested by the name of the parameter, this must work with geometric simplices as well as abstract simplices. With only one argument (*smgr*) the macro selects the simplicial morphism.

`build-ab-smgr`
`:cmpr` *cmpr* `basis` *basis* `:bspn` *bspn* `:face` *face* `:face*` *face**
`:intr-bndr` *intr-bndr* `:bndr-strt` *bndr-strt* `:intr-dgnl` *intr-dgnl*
`:dgnl-strt` *dgnl-strt* `:sintr-grml` *sintr-grml* `:sintr-grin` *sintr-grin*
`:orgn` *orgn*

The description of the parameters is exactly the same as in the previous function, but in this case, the group operation given by the user, (argument *sintr-grml*), must be commutative. The lisp definition of `build-ab-smgr` is simply:

```
(DEFMACRO BUILD-AB-SMGR (&rest rest)
  `(change-class (build-smgr ,@rest) 'ab-simplicial-group))
```

11.4 Two important Abelian simplicial groups: $K(\mathbb{Z},1)$ and $K(\mathbb{Z}_2,1)$

Let us recall the notion of classifying space in the framework of simplicial sets and in the particular case of discrete groups. Given a discrete group $\mathcal{G}$, we know that we may build a simplicial set $K(\mathcal{G},1)$; in dimension n, we have $(K(\mathcal{G},1))_n = \mathcal{G}^n$, the simplices or *bar objects* being conventionally represented as sequences of elements of $\mathcal{G}$:

$$[g_1 \mid g_2 \mid \ldots \mid g_n].$$

The base point is the void bar object [ ]. Let us note the group law multiplicatively.
The face operators are defined as follows:

$$\begin{aligned}
\partial_0[g_1 \mid g_2 \mid \ldots \mid g_n] &= [g_2 \mid g_3 \mid \ldots \mid g_n],\\
\partial_1[g_1 \mid g_2 \mid \ldots \mid g_n] &= [\mathbf{g_1 g_2} \mid g_3 \mid \ldots \mid g_n],\\
\partial_2[g_1 \mid g_2 \mid \ldots \mid g_n] &= [g_1 \mid \mathbf{g_2 g_3} \mid \ldots \mid g_n],\\
\ldots\ldots &= \ldots\ldots\\
\partial_{n-1}[g_1 \mid g_2 \mid \ldots \mid g_n] &= [g_1 \mid g_2 \mid \ldots \mid g_{n-2} \mid \mathbf{g_{n-1} g_n}],\\
\partial_n[g_1 \mid g_2 \mid \ldots \mid g_n] &= [g_1 \mid g_2 \mid \ldots \mid g_{n-1}].
\end{aligned}$$

The degeneracy operators are defined as follows:

$$\eta_i[g_1 \mid \ldots \mid g_{i-1} \mid g_i \mid g_{i+1} \mid \ldots \mid g_n] = [g_1 \mid \ldots \mid g_{i-1} \mid g_i \mid \mathbf{e} \mid g_{i+1} \mid \ldots \mid g_n],$$

where e is the neutral element of $\mathcal{G}$.

If, in addition $\mathcal{G}$ is abelian, then $K(\mathcal{G},1)$ is itself a *simplicial group*, with the following group law, in dimension n:

$$[a_1 \mid \cdots \mid a_n].[b_1 \mid \cdots \mid b_n] = [a_1 + b_1 \mid \cdots \mid a_n + b_n],$$

$$[a_1 \mid \cdots \mid a_n]^{-1} = [-a_1 \mid \cdots \mid -a_n], \quad a, b \in \mathcal{G}$$

In dimension n, the neutral element is the bar object constitued by a sequence of n times the neutral e of G; this is also the n–th degeneracy of the base point [ ] of $K(\mathcal{G},1)$.
The two important simplicial groups $K(\mathbb{Z},1)$ and $K(\mathbb{Z}_2,1)$ may be constructed in `Kenzo`.

`k-z-1` *[Function]*

Build the simplicial group $K(\mathbb{Z},1)$. In this simplicial group, a simplex in dimension n is mathematically represented by a sequence of integers, known as a *bar* object:

$$[a_1 \mid a_2 \mid \ldots \mid a_n].$$

The group operation inherits the additive law of $\mathbb{Z}$:

$$[a_1 \mid \cdots \mid a_n] + [b_1 \mid \cdots \mid b_n] = [a_1 + b_1 \mid \cdots \mid a_n + b_n]$$

and the inverse of $[a_1 \mid \ldots \mid a_n]$ is $[-a_1 \mid \ldots \mid -a_n]$. In `Kenzo`, a non-degenerate simplex of $K(\mathbb{Z},1)$ in dimension n will be simply a list of n non-null integers. The underlying simplicial set is locally effective and its base point is `NIL`, i.e the void bar object [ ]. To be coeherent with the general policy of `Kenzo`, a degenerate simplex must be represented by an abstract simplex of the form `(:absm` *dgop* . *gmsm*`)` where *gmsm* is a non-degenerate simplex, here a list of non-null integers. But, for the computations in the simplicial group, the general form of a bar object, including 0, is more natural and more convenient. Consequently, two conversion functions are provided (see below).

`k-z2-1` *[Function]*

Build the simplicial group $K(\mathbb{Z}_2,1)$. Mathematically, in dimension n, the only non-degenerate simplex is a sequence of n 1's. But one may show that as simplicial set, $K(\mathbb{Z}_2,1)$ is isomorphic to $P^\infty\mathbb{R}$ which may be built by the statement `(R-proj-space)`. This is the representation adopted in `Kenzo`. In dimension n, this simplicial set

has only one non-degenerate simplex, namely the integer n. The formulas for the faces of this non-degenerate simplex n collapse to:

$$\begin{aligned} \partial_0 n &= n-1, \\ \partial_i n &= \eta_{i-1}(n-2), \quad i \neq 0, i \neq n, \\ \partial_n n &= n-1. \end{aligned}$$

As a group, in dimension n, the non–degenerate element n is its own inverse and the neutral is the n-th degeneracy of 0. As in $K(\mathbb{Z},1)$ a degenerate simplex must be represented by a valid abstract simplex, but as the computations are more convenient if the general bar objects are represented by a sequence of mixed 0's and 1's, two conversion functions are provided.

`z-bar-absm` *bar* *[Function]*

Transform a general mathematical bar object *bar* of $K(\mathbb{Z},1)$, represented by a list of integers (including 0's), into a valid abstract simplex (an object of type `ABSM`), (`:absm` *dgop* . *gmsm*), where *gmsm* is a sequence of non-null integers representing a non-degenerate bar object of $K(\mathbb{Z},1)$ and *dgop* a coded sequence of degeneracy operators η_i's.

`z-absm-bar` *absm* *[Function]*

Transform the abstract simplex *absm*, (`:absm` *dgop* . *gmsm*), where *gmsm* is a sequence of non-null integers representing a non-degenerate bar object of $K(\mathbb{Z},1)$ and *dgop* a coded sequence of degeneracy operators, into a list of integers representing a bar object, degenerate or not, of $K(\mathbb{Z},1)$, for more convenience in the internal computations or for more clarity in external printing.

`z2-bar-absm` *bar* *[Function]*

Function analogous to `z-bar-absm` but for $K(\mathbb{Z}_2,1)$.

`z2-absm-bar` *absm* *[Function]*

Function analogous to `z-absm-bar` but for $K(\mathbb{Z}_2,1)$.

Examples

```
(setf KZ1 (k-z-1))  ==>

[K10 Abelian-Simplicial-Group]

(setf simplex '(1 10 100))  ==>
```

```
(1 10 100)

(face KZ1 0 3 simplex)  ==>

<AbSm - (10 100)>
```

Let us build the list of the four faces and by suppressing successively, each face one after the other in this list, let us verify that we have always a Kan hat (we recall that the lisp function **remove**, works on a copy of the list, i.e. the argument **hat** is not modified). In the forth statement, we verify the property of minimality of this special simplicial group.

```
(setf hat (mapcar #'(lambda (i) (face KZ1 i 3 simplex)) (<a-b> 0 3)))  ==>

(<AbSm - (10 100)> <AbSm - (11 100)> <AbSm - (1 110)> <AbSm - (1 10)>)

(dotimes (i 4)
    (check-kan k i 3 (remove (nth i hat) hat)))  ==>

---done---
---done---
---done---
---done---

(setf hat1 (remove (nth 1 hat) hat))  ==>

(<AbSm - (10 100)> <AbSm - (1 110)> <AbSm - (1 10)>)

(kfll KZ1 1 3 hat1)

<AbSm - (1 10 100)>
```

We test now the two macros **grml** and **grin** related to the group operation.

```
(grml KZ1 3 (crpr 0 simplex 0 simplex))  ==>

<AbSm - (2 20 200)>

(setf invsimplex (grin KZ1 3 simplex))  ==>

<Absm - (-1 -10 -100)>

(grin KZ1 3 *)  ==>

<Absm - (1 10 100)>

(grml KZ1 3 (crpr 0 simplex 0 (gmsm invsimplex))))  ==>
```

```
<AbSm 2-1-0 NIL>

(2absm-acrpr (absm 0 simplex ) (grin KZ1 3 simplex)) ==>

<AbSm - <CrPr - (1 10 100) - (-1 -10 -100)>>

(grml KZ1 3 *)  ==>

<AbSm 2-1-0 NIL>

(setf KZ2 (k-z2-1))  ==>

[K22 Abelian-Simplicial-Group]

(grml KZ2 3 (crpr 0 3 0 3))  ==>

<AbSm 2-1-0 0>

(grin KZ2 4 4)  ==>

<AbSm - 4>

(grml KZ2 4 (crpr 0 * 0 *))  ==>

<AbSm 3-2-1-0 0>
```

Let us show some examples of conversions, between general bar objects and valid abstract simplices:

```
(z-absm-bar (absm 0 '()))  ==>

NIL

(z-absm-bar (absm 1 '()))  ==>

(0)

(z-absm-bar (absm 0 '(2)))  ==>

(2)

(z2-absm-bar (absm 7 7))  ==>

(0 0 0 1 1 1 1 1 1)

(z2-absm-bar (absm 0 7))  ==>

(1 1 1 1 1 1 1)
```

Let us suppose that the value of the symbol `labsms` is a list of abstract simplices built from the simplex `(3 6))` in dimension 1 of $K(\mathbb{Z}, 1)$. The application of the function `z-absm-bar` on this list gives the representation of the simplices under the classical mathematical representation. Applying then `z-bar-absm` returns the original list. A similar example is given with the symbol `labsm2` in dimension 0.

```
labsms ==>

(<AbSm - (3 6)> <AbSm 0 (3 6)> <AbSm 1 (3 6)>
 <AbSm 1-0 (3 6)> <AbSm 2 (3 6)> <AbSm 2-0 (3 6)>
 <AbSm 2-1 (3 6)> <AbSm 2-1-0 (3 6)>)

(mapcar #'z-absm-bar labsms)  ==>

((3 6) (0 3 6) (3 0 6) (0 0 3 6) (3 6 0) (0 3 0 6) (3 0 0 6) (0 0 0 3 6))

(mapcar #'z-bar-absm *)  ==>

(<AbSm - (3 6)> <AbSm 0 (3 6)> <AbSm 1 (3 6)>
 <AbSm 1-0 (3 6)> <AbSm 2 (3 6)> <AbSm 2-0 (3 6)>
 <AbSm 2-1 (3 6)> <AbSm 2-1-0 (3 6)>)

labsms2   ==>

(<AbSm - 0> <AbSm - 1> <AbSm - 2> <AbSm - 3>
 <AbSm 0 0> <AbSm 0 1> <AbSm 0 2> <AbSm 0 3>
 <AbSm 1-0 0> <AbSm 1-0 1> <AbSm 1-0 2> <AbSm 1-0 3>)

(mapcar #'z2-absm-bar labsms2)  ==>

(NIL (1) (1 1) (1 1 1) (0) (0 1) (0 1 1) (0 1 1 1) (0 0) (0 0 1) (0 0 1 1)
 (0 0 1 1 1))

(mapcar #'z2-bar-absm *)  ==>

(<AbSm - 0> <AbSm - 1> <AbSm - 2> <AbSm - 3>
 <AbSm 0 0> <AbSm 0 1> <AbSm 0 2> <AbSm 0 3>
 <AbSm 1-0 0> <AbSm 1-0 1> <AbSm 1-0 2> <AbSm 1-0 3>)
```

11.5 Simplicial Groups as Algebras

If we consider a simplicial group $\mathcal{G}$, with group operation τ, there is a canonical product for the algebra

$$\varpi : C_*(\mathcal{G}) \otimes C_*(\mathcal{G}) \longrightarrow C_*(\mathcal{G}),$$

defined as the composition $\tau \circ \mathcal{EML}$, where $\mathcal{EML}$ is the Eilenberg-Mac Lane homomorphism or the *shuffle homomorphism* (see the Eilenberg-Zilber chapter):

$$C_*(\mathcal{G}) \otimes C_*(\mathcal{G}) \stackrel{\mathcal{EML}}{\longrightarrow} C_*(\mathcal{G} \times \mathcal{G}) \stackrel{\tau}{\longrightarrow} C_*(\mathcal{G}).$$

For instance let us look at the simplicial group $K(\mathbb{Z}, 1)$:

```
(setf kz1 (k-z-1))  ==>

[K10 Abelian-Simplicial-Group]

(inspect kz1)  ==>

AB-SIMPLICIAL-GROUP @ #x38c902 = [K10 Abelian-Simplicial-Group]
   0 Class --------> #<STANDARD-CLASS AB-SIMPLICIAL-GROUP>
   1 ORGN ---------> (K-Z-1), a proper list with 1 element
   2 IDNM ---------> fixnum 10 [#x00000028]
   3 EFHM ---------> The symbol :--UNBOUND--
   4 GRMD ---------> [K10 Abelian-Simplicial-Group]
   5 DFFR ---------> [K11 Morphism (degree -1)]
   6 BSGN ---------> The symbol NIL
   7 BASIS --------> The symbol :LOCALLY-EFFECTIVE
   8 CMPR ---------> #<Function K-Z-1-CMPR>
-->9 APRD ---------> The symbol :--UNBOUND--
  10 CPRD ---------> [K14 Morphism (degree 0)]
  11 FACE ---------> #<Function K-Z-1-FACE>
  12 KFLL ---------> The symbol :--UNBOUND--
  13 GRIN ---------> [K21 Simplicial-Morphism]
  14 GRML ---------> [K20 Simplicial-Morphism]
```

We see that the slot `aprd` has not been filled. As long as the user does not need, directly or indirectly, the product of the algebra, this slot remains unbound. But as soon as it is needed, the *slot-unbound* mecanism of CLOS enters in action, the canonical algebra product morphism is defined by the system, and the slot `aprd` is set.
For instance, let us take the tensor product of two elements of $K(\mathbb{Z}, 1)$ and apply the product in the algebra:

```
(setf tnsrp (tnpr 2 '(1 2) 3 '(7 8 9)))  ==>

<TnPr (1 2) (7 8 9)>

(aprd kz1 5 tnsrp)  ==>

------------------------------------------------------------------------{CMBN 5}
<1 * (1 2 7 8 9)>
<-1 * (1 7 2 8 9)>
<1 * (1 7 8 2 9)>
<-1 * (1 7 8 9 2)>
<1 * (7 1 2 8 9)>
<-1 * (7 1 8 2 9)>
<1 * (7 1 8 9 2)>
<1 * (7 8 1 2 9)>
<-1 * (7 8 1 9 2)>
<1 * (7 8 9 1 2)>
------------------------------------------------------------------------------
```

The system `Kenzo` has automatically created the needed algebra product morphism and this is shown in the slot instance `aprd`:

```
(inspect kz1)  ==>

AB-SIMPLICIAL-GROUP @ #x419422 = [K10 Abelian-Simplicial-Group]
   0 Class --------> #<STANDARD-CLASS AB-SIMPLICIAL-GROUP>
   1 ORGN ---------> (K-Z-1), a proper list with 1 element
   2 IDNM ---------> fixnum 10 [#x00000028]
   3 EFHM ---------> The symbol :--UNBOUND--
   4 GRMD ---------> [K10 Abelian-Simplicial-Group]
   5 DFFR ---------> [K11 Morphism (degree -1)]
   6 BSGN ---------> The symbol NIL
   7 BASIS --------> The symbol :LOCALLY-EFFECTIVE
   8 CMPR ---------> #<Function K-Z-1-CMPR>
-->9 APRD ---------> [K35 Morphism (degree 0)]
  10 CPRD ---------> [K14 Morphism (degree 0)]
  11 FACE ---------> #<Function K-Z-1-FACE>
  12 KFLL ---------> The symbol :--UNBOUND--
  13 GRIN ---------> [K21 Simplicial-Morphism]
  14 GRML ---------> [K20 Simplicial-Morphism]
```

An interesting example is to show the associativity of the canonical product of an algebra, by defining in `Kenzo` the morphisms of the following diagram and verifying that this diagram is commutative. In the diagram, ∇ is the algebra product of $\mathcal{A}$, 1 is the identity morphism on $\mathcal{A}$ and *assoc*, the morphism

$$assoc : (\mathcal{A} \otimes \mathcal{A}) \otimes \mathcal{A} \longrightarrow \mathcal{A} \otimes (\mathcal{A} \otimes \mathcal{A}),$$

with:

$$assoc((a \otimes b) \otimes c) = a \otimes (b \otimes c).$$

$$\begin{array}{ccc} (\mathcal{A} \otimes \mathcal{A}) \otimes \mathcal{A} & \stackrel{assoc}{\longrightarrow} & \mathcal{A} \otimes (\mathcal{A} \otimes \mathcal{A}) \\ \nabla \otimes 1 \downarrow & & \downarrow 1 \otimes \nabla \\ \mathcal{A} \otimes \mathcal{A} & & \mathcal{A} \otimes \mathcal{A} \\ \nabla \searrow & & \swarrow \nabla \\ & \mathcal{A} & \end{array}$$

Let us define two chain complexes with the two ways to compose two tensorial products:

```
(setf kz1 (k-z-1))  ==>

[K10 Abelian-Simplicial-Group]

(setf 3-left (tnsr-prdc (tnsr-prdc kz1 kz1) kz1))  ==>

[K13 Chain-Complex]

(setf 3-right (tnsr-prdc kz1 (tnsr-prdc kz1 kz1)))  ==>

[K15 Chain Complex]
```

Then we define the morphism `assoc` and test it:

```
(setf assoc (build-mrph
               :sorc 3-left
               :trgt 3-right
               :degr 0
               :intr #'(lambda (degr a2-a)
                         (with-tnpr (degra2 gnrta2 degra gnrta) a2-a
                            (with-tnpr (degr1 gnrt1 degr2 gnrt2) gnrta2
                               (cmbn (+ degr1 degr2 degra)
                                     1 (tnpr degr1
                                             gnrt1
                                             (+ degr2 degra)
                                             (tnpr degr2 gnrt2 degra gnrta))))))
               :strt :gnrt
               :orgn '(assoc-double-tensor-product)))  ==>
```

```
[K17 Morphism (degree 0)]

(setf *tnpr-with-degrees* t)  ==>

T

(? assoc 7 (tnpr 4 (tnpr 2 '(1 2) 2 '(2 3)) 3 '(4 5 6)))  ==>

------------------------------------------------------------------------{CMBN 7}
<1 * <TnPr 2 (1 2) 5 <TnPr 2 (2 3) 3 (4 5 6)>>>
-------------------------------------------------------------------------------
```

We define now the other morphisms shown in the diagram:

```
(setf nabla (aprd kz1))  ==>

[K35 Morphism (degree 0)]

(setf idnt (idnt-mrph kz1)  ==>

[K36 Morphism (degree 0)]

(setf 1-t-nabla (tnsr-prdc idnt nabla))  ==>

[K39 Morphism (degree 0)]

(setf nabla-t-1 (tnsr-prdc nabla idnt))  ==>

[K42 Morphism (degree 0)]
```

Now, if the diagram is commutative, the difference morphism, noted `zero`, between the following morphisms `right` and `left`, must give the null combination if applied to an element of $(K(\mathbb{Z},1)\otimes K(\mathbb{Z},1))\otimes K(\mathbb{Z},1)$.

```
(setf left (cmps nabla nabla-t-1))  ==>

[K43 Morphism (degree 0)]

(setf right (i-cmps nabla 1-t-nabla assoc))  ==>

[K45 Morphism (degree 0)]
```

```
(setf zero (sbtr left right))  ==>

[K46 Morphism (degree 0)]

(? zero  7 (tnpr 4 (tnpr 2 '(1 2) 2 '(2 3)) 3 '(4 5 6)))  ==>

--------------------------------------------------------------------------{CMBN 7}
------------------------------------------------------------------------------

(? zero 9 (tnpr 7 (tnpr 3 '(2 3 4) 4 '(5 6 7 8)) 2 '(11 12)))

--------------------------------------------------------------------------{CMBN 9}
------------------------------------------------------------------------------
```

The resulting combination of the rather simple double tensor product has yet 140 terms:

```
(? right  7 (tnpr 4 (tnpr 2 '(1 2) 2 '(2 3)) 3 '(4 5 6)))  ==>

--------------------------------------------------------------------------{CMBN 7}
<1 * (1 2 3 2 4 5 6)>
<-1 * (1 2 3 4 2 5 6)>
<1 * (1 2 3 4 5 2 6)>
<-1 * (1 2 3 4 5 6 2)>
<1 * (1 2 4 3 2 5 6)>
<-1 * (1 2 4 3 5 2 6)>
<1 * (1 2 4 3 5 6 2)>
<1 * (1 2 4 5 3 2 6)>
<-1 * (1 2 4 5 3 6 2)>
<1 * (1 2 4 5 6 3 2)>
<-1 * (1 4 2 3 2 5 6)>
<1 * (1 4 2 3 5 2 6)>
<-1 * (1 4 2 3 5 6 2)>
<-1 * (1 4 2 5 3 2 6)>
... ... ... ...
------------------------------------------------------------------------------

(length (cmbn-list *))  ==>

140
```

11.6 Coming back to the Bar of an algebra

The existence of a canonical product for the underlying algebra of a simplicial group allows us now to generate the bar of this algebra. Let us use again $K(\mathbb{Z}, 1)$. We begin to test separately the vertical and the horizontal differential.

```
(setf kz1 (k-z-1))  ==>

[K10 Abelian-Simplicial-Group]

(setf d-vert (bar-intr-vrtc-dffr (dffr kz1)))

(funcall d-vert 0 (abar))

-------------------------------------------------------------------------{CMBN -1}
-------------------------------------------------------------------------------

(setf abar0 (abar 3 '(1 2) 3 '(-1 -2) 3 '(3 4)))  ==>

<<Abar[3 (1 2)][3 (-1 -2)][3 (3 4)]>>

(funcall d-vert 9 abar0)  ==>

-------------------------------------------------------------------------{CMBN 8}
<-1 * <<Abar[2 (1)][3 (-1 -2)][3 (3 4)]>>>
<-1 * <<Abar[2 (2)][3 (-1 -2)][3 (3 4)]>>>
<1 * <<Abar[2 (3)][3 (-1 -2)][3 (3 4)]>>>
<1 * <<Abar[3 (1 2)][2 (-3)][3 (3 4)]>>>
<-1 * <<Abar[3 (1 2)][2 (-2)][3 (3 4)]>>>
<-1 * <<Abar[3 (1 2)][2 (-1)][3 (3 4)]>>>
<-1 * <<Abar[3 (1 2)][3 (-1 -2)][2 (3)]>>>
<-1 * <<Abar[3 (1 2)][3 (-1 -2)][2 (4)]>>>
<1 * <<Abar[3 (1 2)][3 (-1 -2)][2 (7)]>>>
-------------------------------------------------------------------------------

(setf d-hrz  (bar-intr-hrzn-dffr (aprd kz1)))

(funcall d-hrz 9 abar0)  ==>

-------------------------------------------------------------------------{CMBN 8}
<-1 * <<Abar[3 (1 2)][5 (-1 -2 3 4)]>>>
<1 * <<Abar[3 (1 2)][5 (-1 3 -2 4)]>>>
<-1 * <<Abar[3 (1 2)][5 (-1 3 4 -2)]>>>
<-1 * <<Abar[3 (1 2)][5 (3 -1 -2 4)]>>>
<1 * <<Abar[3 (1 2)][5 (3 -1 4 -2)]>>>
<-1 * <<Abar[3 (1 2)][5 (3 4 -1 -2)]>>>
<1 * <<Abar[5 (-1 -2 1 2)][3 (3 4)]>>>
```

```
<-1 * <<Abar[5 (-1 1 -2 2)][3 (3 4)]>>>
<1 * <<Abar[5 (-1 1 2 -2)][3 (3 4)]>>>
<1 * <<Abar[5 (1 -1 -2 2)][3 (3 4)]>>>
<-1 * <<Abar[5 (1 -1 2 -2)][3 (3 4)]>>>
<1 * <<Abar[5 (1 2 -1 -2)][3 (3 4)]>>>
-------------------------------------------------------------------------------
```

We may now construct the bar chain complex of the algebra `kz1` and test the differential, sum of both previous ones:

```
(setf bara (bar kz1))  ==>

[K18 Chain-Complex]

(? bara 9 abar0)

-------------------------------------------------------------------------{CMBN 8}
<-1 * <<Abar[3 (1 2)][5 (-1 -2 3 4)]>>>
<1 * <<Abar[3 (1 2)][5 (-1 3 -2 4)]>>>
<-1 * <<Abar[3 (1 2)][5 (-1 3 4 -2)]>>>
<-1 * <<Abar[3 (1 2)][5 (3 -1 -2 4)]>>>
<1 * <<Abar[3 (1 2)][5 (3 -1 4 -2)]>>>
<-1 * <<Abar[3 (1 2)][5 (3 4 -1 -2)]>>>
<1 * <<Abar[5 (-1 -2 1 2)][3 (3 4)]>>>
<-1 * <<Abar[5 (-1 1 -2 2)][3 (3 4)]>>>
<1 * <<Abar[5 (-1 1 2 -2)][3 (3 4)]>>>
<1 * <<Abar[5 (1 -1 -2 2)][3 (3 4)]>>>
<-1 * <<Abar[5 (1 -1 2 -2)][3 (3 4)]>>>
<1 * <<Abar[5 (1 2 -1 -2)][3 (3 4)]>>>
<-1 * <<Abar[2 (1)][3 (-1 -2)][3 (3 4)]>>>
<-1 * <<Abar[2 (2)][3 (-1 -2)][3 (3 4)]>>>
<1 * <<Abar[2 (3)][3 (-1 -2)][3 (3 4)]>>>
<1 * <<Abar[3 (1 2)][2 (-3)][3 (3 4)]>>>
<-1 * <<Abar[3 (1 2)][2 (-2)][3 (3 4)]>>>
<-1 * <<Abar[3 (1 2)][2 (-1)][3 (3 4)]>>>
<-1 * <<Abar[3 (1 2)][3 (-1 -2)][2 (3)]>>>
<-1 * <<Abar[3 (1 2)][3 (-1 -2)][2 (4)]>>>
<1 * <<Abar[3 (1 2)][3 (-1 -2)][2 (7)]>>>
-------------------------------------------------------------------------------

(? bara *)  ==>

-------------------------------------------------------------------------{CMBN 7}
-------------------------------------------------------------------------------

(setf abar1 (abar '(5 (7 11  4 -9) 4 (8 3 2) 5 (-1 -2 -3 -4))))  ==>
```

```
<<Abar[5 (7 11 4 -9)][4 (8 3 2)][5 (-1 -2 -3 -4)]>>

(? bara 14 abar1)  ==>

-------------------------------------------------------------------------{CMBN 13}
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 -3 -4 8 3 2)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 -3 8 -4 3 2)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 -3 8 3 -4 2)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 -3 8 3 2 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 8 -3 -4 3 2)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 8 -3 3 -4 2)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 8 -3 3 2 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 8 3 -3 -4 2)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 8 3 -3 2 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 -2 8 3 2 -3 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 -2 -3 -4 3 2)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 -2 -3 3 -4 2)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 -2 -3 3 2 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 -2 3 -3 -4 2)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 -2 3 -3 2 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 -2 3 2 -3 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 3 -2 -3 -4 2)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 3 -2 -3 2 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 3 -2 2 -3 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (-1 8 3 2 -2 -3 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 -2 -3 -4 3 2)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 -2 -3 3 -4 2)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 -2 -3 3 2 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 -2 3 -3 -4 2)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 -2 3 -3 2 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 -2 3 2 -3 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 3 -2 -3 -4 2)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 3 -2 -3 2 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 3 -2 2 -3 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (8 -1 3 2 -2 -3 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (8 3 -1 -2 -3 -4 2)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (8 3 -1 -2 -3 2 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (8 3 -1 -2 2 -3 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][8 (8 3 -1 2 -2 -3 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][8 (8 3 2 -1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (7 8 3 2 11 4 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 8 3 11 2 4 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 8 3 11 4 -9 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (7 8 3 11 4 2 -9)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (7 8 11 3 2 4 -9)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (7 8 11 3 4 -9 2)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 8 11 3 4 2 -9)][5 (-1 -2 -3 -4)]>>>
```

```
<-1 * <<Abar[8 (7 8 11 4 -9 3 2)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 8 11 4 3 -9 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (7 8 11 4 3 2 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 11 4 -9 8 3 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (7 11 4 8 -9 3 2)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 11 4 8 3 -9 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (7 11 4 8 3 2 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 11 8 3 2 4 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 11 8 3 4 -9 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (7 11 8 3 4 2 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 11 8 4 -9 3 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (7 11 8 4 3 -9 2)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (7 11 8 4 3 2 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (8 3 2 7 11 4 -9)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (8 3 7 2 11 4 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (8 3 7 11 2 4 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (8 3 7 11 4 -9 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (8 3 7 11 4 2 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (8 7 3 2 11 4 -9)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (8 7 3 11 2 4 -9)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (8 7 3 11 4 -9 2)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (8 7 3 11 4 2 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (8 7 11 3 2 4 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (8 7 11 3 4 -9 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (8 7 11 3 4 2 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (8 7 11 4 -9 3 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[8 (8 7 11 4 3 -9 2)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[8 (8 7 11 4 3 2 -9)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[4 (7 11 -5)][4 (8 3 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[4 (7 11 4)][4 (8 3 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[4 (7 15 -9)][4 (8 3 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[4 (11 4 -9)][4 (8 3 2)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[4 (18 4 -9)][4 (8 3 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][3 (3 2)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][3 (8 3)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][3 (8 5)][5 (-1 -2 -3 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][3 (11 2)][5 (-1 -2 -3 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][4 (8 3 2)][4 (-3 -3 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][4 (8 3 2)][4 (-2 -3 -4)]>>>
<1 * <<Abar[5 (7 11 4 -9)][4 (8 3 2)][4 (-1 -5 -4)]>>>
<-1 * <<Abar[5 (7 11 4 -9)][4 (8 3 2)][4 (-1 -2 -7)]>>>
<1 * <<Abar[5 (7 11 4 -9)][4 (8 3 2)][4 (-1 -2 -3)]>>>
--------------------------------------------------------------------------------
```

```
(? bara *)  ==>

------------------------------------------------------------------------{CMBN 12}
-------------------------------------------------------------------------------
```

Lisp files concerned in this chapter

`simplicial-groups.lisp`, `k-pi-n.lisp`.
[`classes.lisp` , `macros.lisp`, `various.lisp`].

Chapter 12

Fibrations

In tt Kenzo, the fibration theory is applied in the simplicial frame and limited to the *simplicial principal fiber spaces.*

12.1 Notion of fibration

Let B a simplicial set and $\mathcal{G}$ a simplicial group. A *(right–hand) simplicial fibration* is defined by a **twisting operator**, $\tau : B \longrightarrow \mathcal{G}$, of degree -1. For any dimension n and for any element $b \in B_n$, the face and degeneracy operators must satisfy the following relations:

$$\begin{aligned} \partial_i(\tau b) &= \tau(\partial_i b), \qquad i < n-1, \\ \partial_{n-1}(\tau b) &= [\tau(\partial_n b)]^{-1}.\tau(\partial_{n-1} b), \\ \eta_i(\tau b) &= \tau(\eta_i b), \qquad i \leq n-1, \\ e_n &= \tau(\eta_n b). \end{aligned}$$

where e_n is the neutral element of $\mathcal{G}_n$.

Now, the twisting operator τ defines a fibration of base space B, of fiber space $\mathcal{G}$ and of total space $B \times_\tau \mathcal{G}$, whose simplicial components are:

$$(B \times_\tau \mathcal{G})_n = (B \times \mathcal{G})_n = (B_n \times \mathcal{G}_n),$$

and whose face operators are defined, in any dimension n and for any pair (b, g), $b \in B_n$, $g \in \mathcal{G}_n$ by:

$$\begin{aligned} \partial_i(b, g) &= (\partial_i b, \partial_i g), \quad i < n, \\ \partial_n(b, g) &= (\partial_n b, \tau(b).\partial_n g). \end{aligned}$$

As it is well known, there are 4 natural choices for such a definition. The action of the group can be from the right or from the left and the critical index can be the last one (n) or the first one (0). In `Kenzo`, the implementor has chosen the non–usual one (i.e. action on the right hand and critical index, the last one), leading to the Szczarba formulas, the best choice in the case of the loop spaces.

12.2 Building a twisting operator

To build a twisting operator, one uses the function `build-smmr` designed as a convenient tool to build a simplicial morphism. The source (keyword `:sorc`) is the base space, a **reduced** simplicial set, the target (keyword `:trgt`) is the fiber space, a simplicial group. The degree is -1 and the morphism (keyword `:sintr`) is the internal lisp function implementing τ. See the example below, where τ is a simple twisting operator, applying in particular the only (geometrical) simplex of degree 2 of S^2 upon the bar object $[1] \in (K(\mathbb{Z}_2, 1))_1$.
In `Kenzo`, the type `FIBRATION` is defined. An object of this type must be of type `SIMPLICIAL-MRPH`, the degree of the corresponding morphism being -1. To apply the morphism, one uses the well known macro `?` or the special macro `tw-a-sintr3`.

`?` *twop dmns gmsm* *[Macro]*
: Apply the twisting operator *twop* to the geometrical simplex *gmsm* in dimension *dmns*. Of course, `?` works as well with a combination of geometric simplices as in `(? twop cmbn)`.

`tw-a-sintr3` *sintr dmns absm bspn* *[Macro]*
: Apply the internal lisp function *sintr* implementing a twisting operator to the abstract simplex *absm* in dimension *dmns*. The base point *bspn* of the target simplicial set (a simplicial group) must be given, since the function may return the neutral element in dimension $dmns - 1$.

12.3 Constructing the total space

The total space associated to a twisting operator is constructed by the function `fibration-total`.

`fibration-total` *fibration* *[Function]*

Build the simplicial set, total space of the fibration. This function extracts from the simplicial morphism *fibration* (a previously defined twisting operator), the base space (a simplicial set) and the fiber space (a simplicial group) and constructs the twisted cartesian product of these simplicial sets. The face function is built using the internal function `fibration-total-face`. The non–degenerate simplices of the total space are coded exactly as those of the non–twisted product ($B \times \mathcal{G}$) obtained by the lisp call (`crts-prdc` *base fiber*). This non-twisted cartesian product is saved into the slot `grmd` (graded module) of the instance object. If the base space and the fiber space are both of type `KAN`, then the filling function is computed by the internal function `fibration-kfll` and the function `fibration-total` returns an object of type `KAN`.

Examples

```
(setf s2 (sphere 2))  ==>

[K1 Simplicial-Set]

(setf kz2 (k-z2-1))  ==>

[K6 Abelian-Simplicial-Group]

(setf tw2 (build-smmr
             :sorc s2
             :trgt kz2
             :degr -1
             :sintr #'(lambda (dmns gmsm) (absm 0 1))
             :orgn '(s2-tw-kz2)))   ==>

[K18 Fibration]
```

```
(inspect tw2)  ==>

SIMPLICIAL-MRPH @ #x4a54c2 = [K18 Fibration]
   0 Class --------> #<STANDARD-CLASS SIMPLICIAL-MRPH>
   1 ORGN ---------> (S2-TW-KZ2), a proper list with 1 element
   2 IDNM ---------> fixnum 18 [#x00000048]
   3 RSLTS --------> simple T vector (15) = #(#() #() #() ...)
   4 ?-CLNM -------> fixnum 0 [#x00000000]
   5 ???-CLNM -----> fixnum 0 [#x00000000]
   6 STRT ---------> The symbol :GNRT
   7 INTR ---------> The symbol NIL
   8 DEGR ---------> fixnum -1 [#xffffffc]
   9 TRGT ---------> [K6 Abelian-Simplicial-Group]
  10 SORC ---------> [K1 Simplicial-Set]
  11 SINTR --------> #<Interpreted Function (unnamed) @ #x4a545a>
```

The following statement show how to apply the twisting operator to a *geometrical* simplex. But if we have an abstract simplex, one must use the special function `tw-a-sintr3`.

```
(? tw2 2 's2)  ==>

<AbSm - 1>

(tw-a-sintr3 (sintr tw2) 3 (absm 1 's2) nil)  ==>

<AbSm 0 1>

(tw-a-sintr3 (sintr tw2) 3 (absm 7 '*) nil)  ==>

<AbSm 1-0 NIL>

(dotimes (i 3) (print (tw-a-sintr3 (sintr tw2) 3
                                   (absm i 's2) nil)))  ==>

<AbSm - 1>
<AbSm 0 1>
<AbSm 1 1>
```

Let us build the total space of the fibration (a simplicial set). We show some basis and compute some homology groups. The method to compute the homology groups will be explained in a following chapter.

```
(setf tot-spc2 (fibration-total tw2))  ==>

[K24 Simplicial-Set]

(inspect tot-spc2)  ==>
```

```
SIMPLICIAL-SET @ #x4a87f2 = [K24 Simplicial-Set]
   0 Class --------> #<STANDARD-CLASS SIMPLICIAL-SET>
   1 ORGN ---------> <...>, a proper list with 2 elements
   2 IDNM ---------> fixnum 24 [#x00000060]
   3 EFHM ---------> The symbol :--UNBOUND--
   4 GRMD ---------> [K19 Simplicial-Set]
   5 DFFR ---------> [K25 Morphism (degree -1)]
   6 BSGN ---------> <CrPr - * - 0>, a dotted list with 3 elements
   7 BASIS --------> #<Closure (FLET CRTS-PRDC-BASIS RSLT) @ #x4a735a>
   8 CMPR ---------> #<Closure (FLET CRTS-PRDC-CMPR RSLT) [#'SPHERE-CMPR] @ #x4a732a>
   9 CPRD ---------> [K28 Morphism (degree 0)]
  10 FACE ---------> #<Closure (FLET FIBRATION-TOTAL-FACE RSLT) @ #x4a8752>

(basis tot-spc2 0)  ==>

(<CrPr - * - 0>)

(basis tot-spc2 1)  ==>

(<CrPr 0 * - 1>)

(basis tot-spc2 2)  ==>

(<CrPr - S2 - 2> <CrPr - S2 0 1> <CrPr - S2 1 1> <CrPr - S2 1-0 0> <CrPr 1-0 * - 2>)

(basis tot-spc2 3)  ==>

(<CrPr 0 S2 - 3> <CrPr 0 S2 1 2> <CrPr 0 S2 2 2> <CrPr 0 S2 2-1 1> <CrPr 1 S2 - 3>
 <CrPr 1 S2 0 2> <CrPr 1 S2 2 2> <CrPr 1 S2 2-0 1> <CrPr 2 S2 - 3> <CrPr 2 S2 0 2>
 <CrPr 2 S2 1 2> <CrPr 2 S2 1-0 1> <CrPr 2-1-0 * - 3>)

(homology tot-spc2 0 8) ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

---done---

Homology in dimension 2 :

Component Z

Homology in dimension 3 :
```

```
Component Z/4Z

Homology in dimension 4 :

---done---

Homology in dimension 5 :

Component Z/4Z

Homology in dimension 6 :

---done---

Homology in dimension 7 :

Component Z/4Z
```

In the following example, we take $K(\mathbb{Z},1)$ and build a similar twisting operator. Note that in $(K(\mathbb{Z},1))_1$, the bar object [1] is represented as `(1)` whereas in $K(\mathbb{Z}_2,1)$ it is `1`. In this case, it is known that the total space is a model of the sphere S^3 (Hopf fibration - see below).

```
(setf s2 (sphere 2))  ==>

[K1 Simplicial-Set]

(setf kz1 (k-z-1))  ==>

[K6 Abelian-Simplicial-Group]

(setf tw1 (build-smmr
             :sorc s2
             :trgt kz1
             :degr -1
             :sintr #'(lambda (dmns gmsm) (absm 0 (list 1)))
             :orgn '(s2-tw-kz1)))  ==>

[K18 Fibration]

(setf tot-spc1 (fibration-total tw1))  ==>

[K24 Simplicial-Set]

(homology tot-spc1 0 6)  ==>

Homology in dimension 0 :
```

```
Component Z

Homology in dimension 1 :

---done---

Homology in dimension 2 :

---done---

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

---done---

Homology in dimension 5 :

---done---
```

We may also obtain the boundary and faces of elements of the total space:

```
(setf elem (crpr 0 's2 0 '(1 2)))  ==>

<CrPr - S2 - (1 2)>

(? tot-spc1 2 elem)  ==>

-------------------------------------------------------------------------{CMBN 1}
<2 * <CrPr 0 * - (2)>>
<-1 * <CrPr 0 * - (3)>>
-------------------------------------------------------------------------------

(? tot-spc1 *)  ==>

-------------------------------------------------------------------------{CMBN 0}
-------------------------------------------------------------------------------

(dotimes (i 3) (print (face tot-spc1 i 2 elem)))  ==>

<AbSm - <CrPr 0 * - (2)>>
<AbSm - <CrPr 0 * - (3)>>
<AbSm - <CrPr 0 * - (2)>>
```

The system `Kenzo` provides the function `hopf` to build Hopf fibrations. The lisp definition is the following:

```
(defun HOPF (n)
  (declare (fixnum n))
  (the simplicial-mrph
     (build-smmr
        :sorc (sphere 2)
        :trgt (k-z-1)
        :degr -1
        :sintr (hopf-sintr n)
        :orgn `(hopf ,n))))

(defun HOPF-SINTR (n)
  (declare (fixnum n))
  (flet ((rslt (dmns gmsm)
          (declare (ignore dmns gmsm))
          (if (zerop n)
              (absm 1 +empty-list+)
              (absm 0 (list n)))))
          (the sintr #'rslt)))
```

In the definition of the lisp function for the slot `sintr`, we see that the simplicial morphism applies in particular the only geometric simplex in dimension 2 of S^2 onto the bar object $[n] \in (K(\mathbb{Z},1))_1$. If $n = 0$, this is $\eta_0[\,]$, i.e. the 0–degeneracy of the base point `NIL`. We give some examples of some fibrations and of the corresponding total spaces in function of the parameter `n`. First we generate 4 fibrations and apply the twisting operator on the geometrical simplex `s2`.

```
(setf h1 (hopf 1) h2 (hopf 2) h3 (hopf 3) h10 (hopf 10))

(mapcar #'(lambda (h) (funcall (sintr (eval h)) 2 's2)) '(h1 h2 h3 h10)) ==>

(<AbSm - (1)> <AbSm - (2)> <AbSm - (3)> <AbSm - (10)>)
```

Then we generate the corresponding total space and get some homology groups.

```
(setf t1 (fibration-total h1) t2  (fibration-total h2)
      t3 (fibration-total h3) t10 (fibration-total h10))

(homology t1 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :
```

```
---done---

Homology in dimension 2 :

---done---

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

---done---

......... all next groups null .....

(homology t2 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :

---done---

Homology in dimension 3 :

Component Z

---done---

......... all next groups null .....

(homology t3 0 4)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :
```

```
Component Z/3Z

Homology in dimension 2 :

---done---

Homology in dimension 3 :

Component Z

(homology t10 1)  ==>

Homology in dimension 1 :

Component Z/10Z
```

Lisp file concerned in this chapter

`fibrations.lisp`.

Chapter 13

Loop Spaces

13.1 Introduction

Let us recall the definition of the *free group* generated by a set $\mathcal{A}$. Consider the set $\mathcal{A} \times \{+1, -1\}$ and make, as long as there is no ambiguity, the identification $a_i = (a_i, 1)$ and $a_i^{-1} = (a_i, -1)$ for all $a_i \in \mathcal{A}$.

Definition. *a_i and its formal inverse a_i^{-1} are* letters.

Definition. *A word is a* finite *sequence of letters, possibly repeated, without any pair of the form $a_j a_j^{-1}$ and $a_k^{-1} a_k$. The word is said* reduced.

Now, let $\mathbb{Z}^{*\mathcal{A}}$ the set of *words* on $\mathcal{A}$. It is easy to show that if we define in $\mathbb{Z}^{*\mathcal{A}}$ an internal law of composition as the formal *concatenation* of two words, on which the cancelation rule is applied, i.e. any pair of the form $a_j a_j^{-1}$ and $a_k^{-1} a_k$ is replaced by the *void sequence*, then $\mathbb{Z}^{*\mathcal{A}}$ is a group (non-commutative in general). The void word is the neutral element for that law. If we consider the concatenation as a *product*, then, as usual in group theory, a sequence of k times the same letter a will be written a^k and a sequence of k times the same letter a^{-1}, a^{-k}. So a word formed from letters by the concatenation product together with the cancellation rule, will be written under the reduced form:

$$a_1^{n_1} a_2^{n_2} \dots a_k^{n_k},\ n_i \in \mathbb{Z}^*,\ a_i \in \mathcal{A},\ a_i \neq a_{i+1}$$

and its inverse is

$$a_k^{-n_k} \dots a_2^{-n_2} a_1^{-n_1}.$$

After this résumé, let X be a reduced simplicial set (i.e. having only one 0–simplex, namely its base point). The Kan simplicial version GX of the loop space $\Omega(|X|)$ is defined as follows. The set of n–simplices GX_n is the free group generated by the $(n+1)$–simplices $\sigma \in X_{n+1}$ except those which are n–degenerate. In other words:

$$GX_n = \mathbb{Z}^{*X^+_{n+1}},\ X^+_{n+1} = X_{n+1} - \eta_n X_n.$$

In fact, from a strictly mathematical point of view, the right definition is

$$GX_n = \mathbb{Z}^{*X^+_{n+1}}/\overline{\eta_n X_n},$$

i.e. the quotient group of $\mathbb{Z}^{*X^+_{n+1}}$, the free group generated by **all** the $n+1$–simplices, the n–degenerate included, by the normal subgroup $\overline{\eta_n X_n}$ generated by the n–degenerate simplices. Both definitions are equivalent, the second is mathematicaly correct but only the first one is visible in the program.

For a better understanding, we may distinguish carefully a generator σ belonging to X_{n+1} and the corresponding letter $\tau(\sigma) \in GX_n$. So a word, element of GX_n will be written

$$\tau(\sigma_1)^{\epsilon_1}\tau(\sigma_2)^{\epsilon_2}\ldots\tau(\sigma_p)^{\epsilon_p}.$$

Now, let us define two homomorphisms:

$$\overline{\partial}^n_i : GX_n \longrightarrow GX_{n-1},\ \overline{\eta}^n_i : GX_n \longrightarrow GX_{n+1},$$

by the following relations, keeping in mind that ∂_i and η_i are respectively the face and degeneracy operators acting on the simplices of X:

$$\begin{aligned}
\overline{\partial}_i(\tau(\sigma)) &= \tau(\partial_i\sigma), \qquad 0 \le i < n,\\
\overline{\partial}_n(\tau(\sigma)) &= \tau(\partial_{n+1}\sigma)^{-1}\tau(\partial_n\sigma),\\
\overline{\eta}_i(\tau(\sigma)) &= \tau(\eta_i\sigma), \qquad 0 \le i \le n.
\end{aligned}$$

When applied to words, the two homomorphisms, $\overline{\partial}$ and $\overline{\eta}$, satisfy (for clarity, we omit here the operator τ):

$$\begin{aligned}
\overline{\partial}(a_1^{n_1}\ldots a_k^{n_k}) &= (\overline{\partial}(a_1))^{n_1}\ldots(\overline{\partial}(a_k))^{n_k},\\
\overline{\eta}(a_1^{n_1}\ldots a_k^{n_k}) &= (\overline{\eta}(a_1))^{n_1}\ldots(\overline{\eta}(a_k))^{n_k}.
\end{aligned}$$

It is known that the $\overline{\partial}_i$ and the $\overline{\eta}_i$ satisfy the fundamental relations between face and degeneracy operators (see the chapter on simplicial sets). So, GX is itself a simplicial set; each GX_n is a group, each face and degeneracy operator is an homomorphism, so that GX is a *simplicial group*. The importance of the simplicial group GX is emphasized by the Kan theorem, stating that if we consider the realization $|X|$ of X and the loop space $\Omega(|X|, x_0)$ at the base point $x_0 \in |X|$, then $\Omega(|X|, x_0)$ and the realization $|GX|$ of GX have the same homotopy type.

13.2 Representation of letters and words

In the system, the chosen representation for letters and words follows the very definition of the free group $\mathbb{Z}^{*X^+_{n+1}}$. Here, a letter is a generator of GX_n i.e. an **abstract simplex** σ of X^+_{n+1}. A word will be implemented as a sequence of such letters to a certain power (including 1). The chosen terminology is the following:

- A term like σ^p is a **power**.
- A word ${\sigma_1}^{\epsilon_1}{\sigma_2}^{\epsilon_2}\ldots{\sigma_p}^{\epsilon_p}$ is a **loop**.

Note that, in this framework the operator τ is not visible.

13.2.1 Representation of a power

A power is internally represented by lisp object of type `apowr`. This has the form:

(*dgop gmsm . expn*)

in which we may recognize:

1. the components of the abstract simplex: *dgop* and *gmsm*,
2. the exponent of the letter: *expn*, an integer.

The associated constructor is the macro `apowr`.

`apowr` *dgop gmsm expn* *[Macro]*

Build an object of type `apowr`. The accessor macros are respectively `apdgop`, `apgmsm` and `apexpn`. There is no special printing function since the real object interesting for the user is not the power but rather the loop.

13.2.2 Representation of a loop

A loop is represented by a lisp object of the form:

(:loop $apower_1$ ... $apower_k$)

The corresponding type is `LOOP`. The constructor is the macro `make-loop` but the pratical one is `loop3` and the associated printing function prints the object under the form:

<<Loop [*ext-dgop1 gmsm1 expn1*] ... [*ext-dgopk gmsmk expnk*]>>

where the *ext-dgop*'s are under the form of a readable sequence of η operators.

Notion of normalized loop

Normalizing a loop is a matter of factoring the degeneracy operators components (the *dgop*'s) of the *apowr*'s and building an object of type `ABSM` belonging to the simplicial set GX. For instance, the normalized null-loop in dimension 5 is `<AbSm 4-3-2-1-0 <<Loop>>>`. For non-null loops, the following examples show the transformation. We have written the loops under the form printed by the system.

```
<<Loop [1-0 A\2]>> ===>  <AbSm 1-0 <<Loop [A\2]>>>

<<Loop [3-2-1 A\2] [4-2 B\3]>>  ===>  <AbSm 2 <<Loop [2-1 A\2] [3 B\3]>>>.

<<Loop [2-1 A\2] [2-0 B\2] [1-0 A\3]>> ===>

       <AbSm - <<Loop [2-1 A\2] [2-0 B\2] [1-0 A\3]>>>

<<Loop [2-1-0 A\2] [2-1-0 B\2] [2-1-0 A\3]>> ===>

       <AbSm 2-1-0 <<Loop [A\2] [B\2] [A\3]>>>
```

13.3 A set of functions for loops

To facilitate the construction of objects of `LOOP` type, a set of useful functions is provided by the system.

`loop3` *dgop1 gmsm1 pwr1 ... dgopk gmsmk pwrk* *[Function]*
: Construct a loop (a word) corresponding to the product of the abstract simplices asm_i, the components of which being $dgop_i$ and $gmsm_i$, to the power pwr_i. The function constructs itself the objects of type `apowr` from the arguments. It accepts an indefinite

number of arguments, including none. If there is no argument, the null loop is created. It exists in fact in the system the constant `+null-loop+` which is the base point of the underlying simplicial group. **Warning**: the function `loop3` does return neither a normalized loop nor a reduced word (no automatic simplification), so it is up to the user to give a correct sequence of arguments corresponding to a reduced word.

`loop-space-cmpr` *cmpr* *[Function]*

From the comparison function *cmpr* suitable for comparing the geometrical simplices belonging to the underlying simplicial set, build a comparison function for objects of type `LOOP`.

`loop-space-face` *cmpr face* *[Function]*

From the comparison function *cmpr* and the face function *face*, build a face function for an object of type `LOOP` returning the face of such an object as an `ABSM` type object (a normalized loop).

`loop-space-face*` *cmpr face* *[Function]*

From the comparison function *cmpr* and the face function *face*, build a face* function. See the description of the slot `face*` in the simplicial sets chapter.

`loop-space-grml-sintr` *cmpr* *[Function]*

From the comparison function *cmpr*, build a function for the group operation of the loop space (a simplicial group).

`loop-space-grin-sintr` *dmns loop* *[Function]*

Build the inverse of the object *loop* in the simplicial group. The first argument *dmns* is mandatory but ignored.

`loop-space` *smst* `&optional (n 1)` *[Function]*

If the second argument is `1` or omitted, construct, from the **reduced** simplicial set X, here the argument *smst*, the corresponding simplicial group GX. The new created simplicial set is of course locally effective and its base point is the null loop. For any other positive integer value n of the second argument, build the n-th iterated loop space of X. The basic construction is given by the following call to the function `build-smgr`:

```
(build-smgr   :cmpr (loop-space-cmpr cmpr)
              :basis :locally-effective
              :bspn +null-loop+
              :face (loop-space-face cmpr face)
              :face* (loop-space-face* cmpr face)
              :sintr-grml (loop-space-grml-sintr cmpr)
              :sintr-grin #'loop-space-grin-sintr
              :orgn `(loop-space ,smst))
```

`gdeltab` *[Function]*

Build $\Omega^1(\bar{\Delta})$. This is simply `(loop-space(deltab))`. We recall that $\bar{\Delta}$ is the reduced and locally effective simplicial set obtained from $\Delta^{\mathbb{N}}$, by identifying all the vertices to the base point. $\Delta^{\mathbb{N}}$ is the locally effective simplicial set freely generated by the positive integers.

Examples

```
(loop3)  ==>

<<Loop>>

(loop3 0 'a 2 20 'b -3 12 'c 1)  ==>

<<Loop [A\2] [4-2 B\-3] [3-2 C]>>

(loop3 0 'a 2 20 'b -3 12 'c 1 0 'a 3)  ==>

<<Loop [A\2] [4-2 B\-3] [3-2 C] [A\3]>>

(loop3 (dgop-ext-int '(2 1 0)) 'p 2
       (dgop-ext-int '(5 4 3 2 1)) 'q 5)  ==>

<<Loop [2-1-0 P\2] [5-4-3-2-1 Q\5]>>
```

Let us consider now a true reduced simplicial set, namely $Moore(\mathbb{Z}/2\mathbb{Z}, 3)$, whose structure is shown using the function `show-structure` defined in the Simplicial Sets chapter.

```
(setf moore23 (moore 2 3)) ==>

[K7 Simplicial-Set]

(show-structure moore23 4)  ==>


Dimension = 0 :

        Vertices :  (*)

Dimension = 1 :

Dimension = 2 :

Dimension = 3 :
```

```
        Simplex : M3

                Faces : (<AbSm 1-0 *> <AbSm 1-0 *>
                         <AbSm 1-0 *> <AbSm 1-0 *>)

Dimension = 4 :

        Simplex : N4

                Faces : (<AbSm - M3> <AbSm 2-1-0 *>
                         <AbSm - M3> <AbSm 2-1-0 *> <AbSm 2-1-0 *>)
```

Let us build now $\Omega^1(Moore(\mathbb{Z}/2\mathbb{Z}, 3))$ and ask for faces of some loops. The user will note that the function face admits either `LOOP` type objects or `ABSM` type objects (normalized loop) but always returns normalized loops, i.e. an abstract simplex, according to the normal behaviour of the face function in a simplicial set.

```
(setf o-moore23 (loop-space moore23))  ==>

[K11 Simplicial-Group]

(face o-moore23 0 4 +null-loop+)  ==>

<AbSm 2-1-0 <<Loop>>>

(face o-moore23 0 3 (loop3 0 'N4 2))  ==>

<AbSm - <<Loop [M3\2]>>>

(face o-moore23 1 3 (loop3 0 'N4 2))  ==>

<AbSm 1-0 <<Loop>>>

(face o-moore23 1 4 (loop3 1 'N4 2))  ==>

<AbSm - <<Loop [N4\2]>>>

(face o-moore23 0 2 (loop3 0 'M3 2))  ==>

<AbSm 0 <<Loop>>>
```

Let us do the same with the following reduced simplicial set (`ss2`) and with $\Omega^1(\bar{\Delta})$.

```
(setf ss2 (build-finite-ss '( *
                              2 s2 (* * *)
                              3 s3 (s2 s2 s2 s2))))  ==>

Checking the 0-simplices...
Checking the 1-simplices...
Checking the 2-simplices...
Checking the 3-simplices...
[K18 Simplicial-Set]

(setf o-ss2 (loop-space ss2))  ==>

[K23 Simplicial-Group]

(face o-ss2 0 1 (loop3 0 's2 1))  ==>

<AbSm - <<Loop>>>

(face o-ss2 0 2 (loop3 0 's3 2))  ==>

<AbSm - <<Loop [S2\2]>>>

(face o-ss2 1 2 (loop3 0 's3 2))  ==>

<AbSm - <<Loop [S2\2]>>>

(face o-ss2 1 2 (loop3 0 's3 2))  ==>

<AbSm - <<Loop [S2\2]>>>

(setf db (gdeltab))  ==>

[K40 Simplicial-Group]

(face db 3 3 (loop3 12 7 3))  ==>

<AbSm 1-0 <<Loop>>>

(face db 3 3 (loop3 5 7 3))  ==>

<AbSm 0 <<Loop [7\3]>>>

(face db 3 3 (loop3 6 7 3 5 7 3 3 7 3))  ==>

<AbSm - <<Loop [1 7\3] [0 7\3] [1-0 3\-1] [1-0 5] [1-0 3\-1]
```

```
               [1-0 5] [1-0 3\-1] [1-0 5]>>>

(face db 3 2 *)  ==>

<AbSm - <<Loop [7\3] [0 3\-1] [0 5] [0 3\-1] [0 5] [0 3\-1]
               [0 5] [0 3\-1] [0 5] [0 3\-1] [0 5] [0 3\-1] [0 5]>>>
```

The last statement shows that the boundary operator attached to the chain complex derived from all this machinery satisfies its fundamental property ($d \circ d = 0$).

```
(? db (? db 3 (loop3 (dgop-ext-int '(3 2)) (dlop-ext-int '(0 1 2)) 3
                     (dgop-ext-int '(3 1)) (dlop-ext-int '(0 1 2)) 3
                     (dgop-ext-int '(2 1)) (dlop-ext-int '(0 1 2)) 3)))  ==>

-------------------------------------------------------------------------{CMBN 1}
-------------------------------------------------------------------------------
```

The loop spaces of reduced simplicial sets, provide us with new examples of algebras. So, we may apply the functor `bar`.

```
(setf moore-22 (moore 2 2))  ==>

[K1 Simplicial-Set]

(dotimes (i 4) (print (basis moore-22  i :dgnr)))  ==>

(<AbSm - *>)
(<AbSm 0 *>)
(<AbSm - M2> <AbSm 1-0 *>)
(<AbSm - N3> <AbSm 0 M2> <AbSm 1 M2> <AbSm 2 M2> <AbSm 2-1-0 *>)

(setf o-moore-22 (loop-space moore-22))  ==>

[K6 Simplicial-Group]

(aprd o-moore-22 3 (tnpr 1 (loop3 0 'm2 1) 2 (loop3 0 'n3 1)))  ==>

-------------------------------------------------------------------------{CMBN 3}
<1 * <<Loop[1-0 M2][2 N3]>>>
<-1 * <<Loop[2-0 M2][1 N3]>>>
<1 * <<Loop[2-1 M2][0 N3]>>>
-------------------------------------------------------------------------------

(setf bar-om (bar o-moore-22))  ==>

[K36 Chain-Complex]
```

```
(? bar-om 5 (abar 2 (loop3 0 'm2 1) 3 (loop3 0 'n3 1)))  ==>

-----------------------------------------------------------------------{CMBN 4}
<1 * <<Abar[4 <<Loop[1-0 M2][2 N3]>>]>>>
<-1 * <<Abar[4 <<Loop[2-0 M2][1 N3]>>]>>>
<1 * <<Abar[4 <<Loop[2-1 M2][0 N3]>>]>>>
<-2 * <<Abar[2 <<Loop[M2]>>][2 <<Loop[M2]>>]>>>
-------------------------------------------------------------------------------
```

Lisp file concerned in this chapter

`loop-spaces.lisp`.

Chapter 14

Disk pasting and suspension

14.1 Introduction

Let us recall the technique of space construction by *attaching maps*[1]. Let A be a subspace of a space X and a map f of A into a space Y. In the space $X \coprod Y$, let us identify each x in A with its image $f(x)$ in Y. The quotient space $Z = X \bigcup_f Y$ of the space $X \coprod Y$ by the equivalence relation that the identification above determines, is called the *adjunction space* of the system $(X \supset A), (Y \supset f(A))$. The quotient map $g : X \coprod Y \to Z$ sends Y onto a subspace of Z.
In this section, we shall take as pair (X, A) the pair (D^n, S^{n-1}), where D^n is the unit disc of $\mathbb{R}^n$ and S^{n-1} is the sphere of dimension $n-1$, boundary of D^n. In this case, the space $Z = D^n \bigcup_f Y$ is said to be obtained from Y by *attaching an n–disc via f*, or *disk pasting*.

14.2 The functions for disk pasting

The **Kenzo** program implements the previous technique of space construction, via simplicial sets. We know that D^n is homeomorphic to the standard simplex Δ^n. The method used by the implementor is the following:
Starting from a simplicial set *ss* representing Y, build a new one having the same simplices as *ss* and in dimension n, a new added simplex described by the user. This description consists of:
– a name (a symbol) for this new generator,
– the list of all its $n+1$ faces.

[1] **Marvin Greenberg** in *Lectures on Algebraic Topology.* Benjamin Inc, 1967.

`disk-pasting` *smst dmns new faces* *[Function]*

Construct, from the simplicial set *smst*, a new simplicial set by attaching the unit disc of dimension *dmns*. The argument *faces* is the list of $dmn+1$ simplices corresponding to the faces of the new simplex attached to *smst*. These simplices are, in principle, written as abstract simplices (*absm*), nevertheless it is possible to write them as geometric simplices (*gmsm*). In this case, the corresponding geometric simplices must have the correct dimension. The program will transform the list of *gmsm*'s into a list of *absm*'s. The argument *new* allows the user to name this new added simplex. The program verifies the coherence of the addition of the new simplex by calling internally the function `check-faces`.

`chcm-disk-pasting` *chcm dmns new bndr* *[Function]*

Construct, from the chain complex *chcm* (chain groups $\mathcal{C}_*$), a new chain complex (chain groups $\mathcal{C}'_*$), where:

$$C'_p = C_p, \quad \forall p \neq dmn,$$

$$C'_{dmn} = C_{dmn} \oplus \mathbb{Z}\, new.$$

The symbol `new` is the name given by the user to a new generator in dimension *dmns* and *bndr* is a combination of degree $dmns - 1$ representing the boundary of the generator *new*. Of course, one must have $d_{dmns-1}(bndr) = 0$.

`hmeq-disk-pasting` *hmeq dmns new bndr* `&key` *new-lbcc* *[Function]*

Construct a homotopy equivalence by attaching the generator *new* of dimension *dmns* with boundary *bndr* to the left bottom chain complex of the homotopy equivalence *hmeq*. This modification of the left bottom chain complex is propagated along the whole new homotopy equivalence. The key parameter *new-lbcc* is for internal use.

Example

Let us begin by a trivial example corresponding to the well known following result: if f maps S^{n-1} onto a point Y, then by disk pasting we obtain a space homeomorphic to S^n. Here, the point Y will be represented by the trivial simplicial set:

```
 (setf s0 (build-finite-ss '(*)))  ==>

Checking the 0-simplices...
[K1 Simplicial-Set]
```

In the identification process, the 1-simplices boundaries of the 2-simplex Δ^2 are applied on the 0–degeneracy of the base point $*$.

```
(setf s2 (disk-pasting s0
                       2
                      's2
                      (list (absm 1 '*) (absm 1 '*) (absm 1 '*))  ))

[K6 Simplicial-Set]

(show-structure s2 2)  ==>

Dimension = 0 :

        Vertices :  (*)

Dimension = 1 :

Dimension = 2 :

        Simplex : S2

                Faces : (<AbSm 0 *> <AbSm 0 *> <AbSm 0 *>)
```

More interesting is the construction of the successive projectives spaces $P^i\mathbb{R}$. Let us begin by $P^1\mathbb{R}$. We start from a point (the simplicial set `s0`) and we identify the two boundary points of Δ^1 to the base point of `s0`. The new simplex to be added to `s0` is of dimension 1 and its 2 faces are the base point itself. It is well known that the resulting corresponding space is homeomorphic to the circle S^1.

```
(setf p1r (disk-pasting s0 1 'd1 (list (absm 0 '*)(absm 0 '*))))  ==>

[K34 Simplicial-Set]
```

```
(show-structure p1r 1)  ==>

Dimension = 0 :

        Vertices :  (*)

Dimension = 1 :

        Simplex : D1

                Faces : (<AbSm - *> <AbSm - *>)

(dotimes (i 3) (homology p1r i))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z

Homology in dimension 2 :

---done---
```

The projective spaces $P^n\mathbb{R}$ of higher degree may be constructed by the following iterative rule. Suppose we have built a simplicial set corresponding to $P^{n-1}\mathbb{R}$ and that we have named the additional simplex d_{n-1}. Suppose also that d_{n-2} has been previously defined. (d_0 is the base point). Then the simplicial set corresponding to $P^n\mathbb{R}$ is obtained by pasting Δ^n with the following description of the additional simplex:

- the name of the new simplex is d_n,
- the faces of d_n are:

$$\begin{aligned}
\partial_0 d_n &= d_{n-1}, \\
\partial_1 d_n &= \eta_0 d_{n-2}, \\
\partial_2 d_n &= \eta_1 d_{n-2}, \\
\cdots &= \cdots, \\
\partial_{n-1} d_n &= \eta_{n-2} d_{n-2}, \\
\partial_n d_n &= d_{n-1}.
\end{aligned}$$

So, applying this rule, we may construct a model for $P^i\mathbb{R}$, $i = 2, 3, 4$ and verify the well known results about their respective homology groups.

```
(setf p2r (disk-pasting p1r 2 'd2
                        (list (absm 0 'd1)
                              (absm 1 '*)
                              (absm 0 'd1)) )) ==>

[K42 Simplicial-Set]

(show-structure p2r 2)  ==>

Dimension = 0 :

        Vertices :  (*)

Dimension = 1 :

        Simplex : D1

                Faces : (<AbSm - *> <AbSm - *>)

Dimension = 2 :

        Simplex : D2

                Faces : (<AbSm - D1> <AbSm 0 *> <AbSm - D1>)

(dotimes (i 3) (homology p2r i))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :

---done---

(setf p3r (disk-pasting p2r 3 'd3
                        (list (absm 0 'd2)
                              (absm 1 'd1)
                              (absm 2 'd1)
                              (absm 0 'd2))))  ==>
```

```
[K56 Simplicial-Set]

(show-structure p3r 3)  ==>

Dimension = 0 :

        Vertices :  (*)

Dimension = 1 :

        Simplex : D1

                Faces : (<AbSm - *> <AbSm - *>)

Dimension = 2 :

        Simplex : D2

                Faces : (<AbSm - D1> <AbSm 0 *> <AbSm - D1>)

Dimension = 3 :

        Simplex : D3

                Faces : (<AbSm - D2> <AbSm 0 D1> <AbSm 1 D1> <AbSm - D2>)

(dotimes (i 4)(homology p3r i))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :

Homology in dimension 3 :

Component Z


(setf p4r (disk-pasting p3r 4 'd4
                        (list (absm 0 'd3)
                              (absm 1 'd2)
                              (absm 2 'd2)
```

```
                              (absm 4 'd2)
                              (absm 0 'd3))))  ==>

[K70 Simplicial-Set]

(dotimes (i 5) (homology p4r i))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :

Homology in dimension 3 :

Component Z/2Z

Homology in dimension 4 :

---done---
```

Let us give now some examples with the function `chcm-disk-pasting`. We are going to work with the unit chain complex (obtained by the function `z-chcm`, see chapter 1), having a unique non null component, namely a $\mathbb{Z}$–module of degree 0 generated by the unique generator `:Z-gnrt`.

```
(setf *ccz* (z-chcm))  ==>

[K84 Chain-Complex]
```

Using the function `chcm-disk-pasting`, let us construct from `*ccz*` a new chain complex having in dimension 1 a unique new generator `gen-1` whose boundary is 5 times the unique generator in dimension 0. In this new chain complex the homology group in dimension 0 is $\mathbb{Z}/5\,\mathbb{Z}$.

```
(setf newcc-0 (chcm-disk-pasting *ccz* 1 'gen-1 (cmbn 0 5 :Z-gnrt)))  ==>

[K86 Chain-Complex]

(chcm-homology-gen newcc-0 0)  ==>

Homology in dimension 0 :

Component Z/5Z
```

```
Generator :

1     *  :Z-GNRT

(chcm-homology-gen newcc-0 1)  ==>

Homology in dimension 1 :

---done---
```

Let us do the same with a generator whose boundary is 165 times the generator `:Z-gnrt`:

```
(setf newcc-1 (chcm-disk-pasting *ccz* 1 'gen-1 (cmbn 0 165 :Z-gnrt))) ==>

[K88 Chain-Complex]

(chcm-homology-gen newcc-1 0)  ==>

Homology in dimension 0 :

Component Z/165Z

Generator :

1     *  :Z-GNRT
```

Now from the chain complex `newcc-1`, let us build a new one, `newcc-2`, by adding in dimension 1 a second generator whose boundary is 182 times the generator `:Z-gnrt`. The homology group in dimension 0 of this new chain complex is the null group, since 165 and 182 are relatively prime integers.

```
(setf newcc-2 (chcm-disk-pasting newcc-1 1 'gen-2 (cmbn 0 182 :Z-gnrt)))  ==>

[K92 Chain-Complex]

(chcm-homology-gen newcc-2 0)  ==>

Homology in dimension 0 :

---done---

(chcm-homology-gen newcc-2 1)  ==>

Homology in dimension 1 :

Component Z

Generator :
```

```
-165  *  GEN-2
182   *  GEN-1
```

As a second example, let us start from $\Omega^1(S^3)$.

```
(setf s3 (sphere 3))  ==>

[K94 Simplicial-Set]

(setf os3 (loop-space s3))  ==>

[K99 Simplicial-Group]
```

In the following instruction, we locate in the symbol `fund-simp` the canonical generator of $\pi_2(\Omega^1 S^3)$, that is the 2–simplex coming from the original sphere:

```
(setf fund-simp (absm 0 (loop3 0 's3 1)))  ==>

<AbSm - <<Loop[S3]>>>
```

We need also the 2–degeneracy of the base point of the loop space:

```
(setf null-simp (absm 3 +null-loop+))  ==>

<AbSm 1-0 <<Loop>>>
```

We may build now a new object by pasting a disk D^3 as indicated by the following call:

```
(setf dos3 (disk-pasting os3 3 '<D3>
                        (list fund-simp null-simp fund-simp null-simp)))  ==>

[K212 Simplicial-Set]

(homology dos3 2)  ==>

Homology in dimension 2 :

Component Z/2Z

(homology dos3 3)  ==>

Homology in dimension 3 :

---done---
```

Then, let us build the loop space of the object `dos3` and show the homology in dimension 5:

```
(setf odos3 (loop-space dos3))  ==>

[K230 Simplicial-Group]

(homology odos3 5)  ==>

Homology in dimension 5 :

Component Z/2Z

Component Z/2Z

Component Z/2Z

Component Z/2Z

Component Z/2Z

Component Z/2Z

Component Z
```

14.2.1 Searching Homology for objects created by disk pasting

The comment list of an object created by disk pasting contains various informations as shown by the simple following example

```
(setf s0 (build-finite-ss '(*)))  ==>

Checking the 0-simplices...

[K1 Simplicial-Set]

(setf s2 (disk-pasting s0
                       2
                      's2
                      (list (absm 1 '*) (absm 1 '*) (absm 1 '*))  ))  ==>

[K6 Simplicial-Set]

(orgn s2)  ==>

(DISK-PASTING [K1 Simplicial-Set] 2 S2 (<AbSm 0 *> <AbSm 0 *> <AbSm 0 *>))
```

In fact, the comment list contains the very arguments used to realize the attaching. The method retrieves those arguments, gets the homotopy equivalence value of the `efhm` slot of the old object –creating it if necessary – then

build a new homotopy equivalence, by calling the function `hmeq-disk-pasting` *hmeq* which takes in account the attaching.

```
(DEFMETHOD SEARCH-EFHM (smst (orgn (eql 'disk-pasting)))
  (declare (type simplicial-set smst))
  (the homotopy-equivalence
   (destructuring-bind (old-smst dmns new faces) (rest (orgn smst))
       (declare
          (type simplicial-set old-smst)
          (fixnum dmns)
          (symbol new)
          (ignore faces))
       (hmeq-disk-pasting
               (efhm old-smst)
               dmns new (? smst dmns new)
               :new-lbcc smst))))
```

14.3 The functions for the suspension

In **Kenzo** the suspension process is realized by the function `suspension`. This function has one argument, a reduced object (chain complex, simplicial set, etc.). **If the object is not reduced, the result is undefined**. The CLOS mechanism is used to apply an adequate method for building the suspension of the object. For a chain complex, a new base point is created with name `:s-bsgn` and the degree of the generators is increased by 1. If the suspension process is applied to a homotopy equivalence, then firstly, the left bottom chain complex is suspended and secondly, the modification is propagated along the whole homotopy equivalence.

`suspension-cmpr` *cmpr* *[Function]*
: From the comparison function *cmpr*, build a comparison function to compare two generators of a suspension.

`suspension-basis` *basis* *[Function]*
: From the function *basis* of a chain complex, build a basis function for the suspension of this chain complex. In degree 0, the only generator is `:s-bsgn`, in degree 1, the basis is void and in degree k, $k \geq 2$, the elements of the basis of the suspension are the elements of the initial chain complex in degree $k-1$.

`suspension-intr-dffr` *dffr* *[Function]*
: From the lisp differential function *dffr* of a chain complex, build the lisp differential function for the suspension of this chain complex.

`suspension-intr-cprd` *cmbn* *[Function]*
: Return the result of the application of the coproduct of a suspension (considered as a coalgebra) upon the combination *cmbn*. Note that this function does not need any other argument than the combination.

`suspension-face` *face* *[Function]*
: From the function face of a simplicial set, build the face function for the suspension of this simplicial set.

`suspension-intr` *mrph* *[Function]*
: From a **Kenzo** morphism *mrph*, build an internal function corresponding to the suspension of the initial morphism. This function will be applied to a combination assumed to belong to a suspension of a chain complex.

`suspension` *obj* `&optional` *(n 1)* *[Function]*

Construct a new object, the suspension of the **reduced** object *obj*. The (geometrical) base point of the suspension of a chain complex (or simplicial set) is named `:s-bsgn`. The optional parameter *n* (default: 1) allows to create in one instruction iterated suspensions. This function calls the adequate method (`suspension-1`) for the object *obj*.

`suspension-1` *chcm* *[Method]*

Build the chain complex suspension of the chain complex *chcm*, using the basic functions above, as shown in the following call to `build-chcm`:

```
(build-chcm
  :cmpr (suspension-cmpr cmpr)
  :basis (suspension-basis basis)
  :bsgn :s-bsgn
  :intr-dffr (suspension-intr-dffr dffr)
  :strt :cmbn
  :orgn `(suspension ,chcm))
```

`suspension-1` *clbg* *[Method]*

Return the coalgebra associated to the suspension of the coalgebra *clbg*. This calls internally the method for the suspension of the chain complex *clbg* then assigns the function `suspension-intr-cprd` to the value of the slot `cprd`.

`suspension-1` *smst* *[Method]*

Return the simplicial set, suspension of the simplicial set *smst*. This function uses the previous basic function `suspension-face`. The resulting object is also a coalgebra with a coproduct defined by the function `suspension-intr-cprd`.

`suspension-1` *mrph* *[Method]*

Build the suspension of the morphism *mrph*. This is a morphism of the same degree as *mrph*. If *mrph* is the zero morphism, return the zero morphism between the suspensions of the source and target of *mrph*. If *mrph* is the identity morphim, return the identity morphism upon the suspension of the source of *mrph*. Otherwise, return the morphism built by the following call to `build-mrph`:

```
(build-mrph
   :sorc (suspension (sorc mrph))
   :trgt (suspension (trgt mrph))
   :degr (degr mrph)
   :intr (suspension-intr mrph)
   :strt :cmbn
   :orgn `(suspension ,mrph))
```

`suspension-1` *rdct* *[Method]*

Build the suspension of the reduction *rdct*. If *rdct* is the trivial reduction upon a chain complex $\mathcal{C}$, return the trivial reduction upon the suspension of $\mathcal{C}$. Otherwise, return the reduction built by the following call to the function `build-rdct`:

```
(build-rdct
   :f (suspension (f rdct))
   :g (suspension (g rdct))
   :h (suspension (h rdct))
   :orgn `(suspension ,rdct))
```

`suspension-1` *hmeq* *[Method]*

Build the suspension of the homotopy equivalence *hmeq*. If *hmeq* is the trivial homotopy equivalence upon a chain complex $\mathcal{C}$, return the trivial homotopy equivalence upon the suspension of $\mathcal{C}$. Otherwise, return the homotopy equivalence built by the following call to the function `build-hmeq`:

```
(build-hmeq
   :lrdct (suspension (lrdct hmeq))
   :rrdct (suspension (rrdct hmeq))
   :orgn `(suspension ,hmeq))
```

Examples

Firstly we test the coproduct of the suspension of $\bar{\Delta}$. In the third statement, the generators 7 and 11 are the binary coded representation of the simplices `(0 1 2)` and `(0 1 3)` in dimension 3 of $\mathcal{S}\bar{\Delta}$,

```
(suspension-intr-cprd (cmbn 0))  ==>

--------------------------------------------------------------------------{CMBN 0}
--------------------------------------------------------------------------------

(suspension-intr-cprd (cmbn 0 5 :s-bsgn))  ==>

--------------------------------------------------------------------------{CMBN 0}
<5 * <TnPr S-BSGN S-BSGN>>
--------------------------------------------------------------------------------

(suspension-intr-cprd (cmbn 3 4 7 5 11))  ==>

--------------------------------------------------------------------------{CMBN 3}
<4 * <TnPr S-BSGN 7>>
<5 * <TnPr S-BSGN 11>>
<4 * <TnPr 7 S-BSGN>>
<5 * <TnPr 11 S-BSGN>>
--------------------------------------------------------------------------------
```

Let us verify now the classical result $\mathcal{S}(S^n) = S^{n+1}$.

```
(setf s1 (sphere 1))  ==>

[K1 Simplicial-Set]

(setf ss1 (suspension s1))  ==>

[K6 Simplicial-Set]

(orgn ss1)  ==>

(SUSPENSION [K1 Simplicial-Set])

(show-structure ss1 2)  ==>

Dimension = 0 :

        Vertices :  (S-BSGN)

Dimension = 1 :

Dimension = 2 :

        Simplex : S1

                Faces : (<AbSm 0 S-BSGN> <AbSm 0 S-BSGN> <AbSm 0 S-BSGN>)
```

For comparison, let us recall the simplicial set model of S^2:

```
(show-structure (sphere 2) 2)  ==>

Dimension = 0 :

        Vertices : (*)

Dimension = 1 :

Dimension = 2 :

        Simplex : S2

                Faces : (<AbSm 0 *> <AbSm 0 *> <AbSm 0 *>)
```

Let us iterate twice the operation of suspension on S^2:

```
(setf ss22 (suspension (sphere 2) 2))  ==>

[K21 Simplicial-Set]
```

```
(show-structure ss22 4)  ==>

Dimension = 0 :

        Vertices :  (S-BSGN)

Dimension = 1 :

Dimension = 2 :

Dimension = 3 :

Dimension = 4 :

        Simplex : S2

                Faces : (<AbSm 2-1-0 S-BSGN> <AbSm 2-1-0 S-BSGN>
                         <AbSm 2-1-0 S-BSGN> <AbSm 2-1-0 S-BSGN>
                         <AbSm 2-1-0 S-BSGN>)
```

To test the suspension of a less simple chain complex, let us take a Moore space that we have seen in the simplicial set chapter. We know that the corresponding chain complex is connected. Let us compare the basis elements, in various degrees, of this chain complex and its suspension:

```
(setf m23 (moore 2 3))  ==>

[K26 Simplicial-Set]

(dotimes (i 7) (print (basis m23 i)))

(*)
NIL
NIL
(M3)
(N4)
NIL
NIL
NIL

(setf sm23 (suspension m23))  ==>

[K31 Simplicial-Set]

(dotimes (i 7) (print (basis sm23 i)))

(:S-BSGN)
```

```
NIL
NIL
NIL
(M3)
(N4)
NIL
NIL
```

Let us apply now the function `suspension` to objects built from S^2, namely $\Omega^1(S^2)$ and $\Omega^2(S^2)$ and let us have a look upon the homology groups . For S^2, we know that the only non–null homology groups are in dimension 0 and 2:

```
(homology (sphere 2) 0 5)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Component Z

Homology in dimension 3 :

Homology in dimension 4 :

---done---
```

For $\mathcal{S}(S^2)$, the non–null homology groups are of course in dimension 0 and 3:

```
(setf ss2 (suspension (sphere 2)))  ==>

[K16 Simplicial-Set]

(homology ss2 0 5)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :
```

```
Homology in dimension 3 :

Component Z

Homology in dimension 4 :

---done---
```

Let us verify now that, for the space $\Omega(S^2)$, the homology groups are organized as the tensor algebra over one generator in degree 1:

```
(setf s2 (sphere 2))  ==>

[K11 Simplicial-Set]

(setf os2 (loop-space s2))  ==>

[K44 Simplicial-Group]

(homology os2 0 5)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z

Homology in dimension 2 :

Component Z

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

Component Z
```

And, in the suspension of the previous space, we verify that the homology groups are suspended (in particular, there is no homology in dimension 1)

```
(setf sos2 (suspension os2))  ==>

[K153 Simplicial-Set]
```

```
(homology sos2 0 5)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Component Z

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

Component Z
```

At last, taking the loop space of the previous suspension, we recognize the tensor algebra over the previous tensor algebra, $\mathcal{H}_*\Omega S^2$. In each dimension, the number of generators is 2^n:

```
(setf osos2 (loop-space sos2))  ==>

[K171 Simplicial-Group]

(homology osos2 0 5)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z

Homology in dimension 2 :

Component Z

Component Z

Homology in dimension 3 :

Component Z
```

```
Component Z

Component Z

Component Z

Homology in dimension 4 :

Component Z

Component Z

Component Z

Component Z

Component Z

Component Z

Component Z

Component Z
```

14.3.1 Searching homology for suspensions

The comment list of a suspension has the form `(SUSPENSION` *suspended-object*`)`. The `search-efhm` method applied to a suspension, looks for the value of the `efhm` slot of the *suspended-object*, (i.e. a homotopy equivalence), then builds the suspension of this homotopy equivalence (see the method `suspension` applied to a homotopy equivalence). Of course, this may imply a recursive process.

```
(DEFMETHOD SEARCH-EFHM (suspension (orgn (eql 'suspension)))
  (declare (type chain-complex suspension))
  (suspension (efhm (second (orgn suspension)))))
```

Lisp files concerned in this chapter

`disk-pasting.lisp`, `suspensions.lisp`, `searching-homology.lisp`

Chapter 15

Loop spaces fibrations

15.1 The canonical loop space fibration

Let X be a simplicial set. The software `Kenzo` implements the **canonical** twisted cartesian product of X by its loop space GX, denoted $X \times_\tau GX$, giving the Kan[1] model of the contractible path space of X. The *canonical fibration*

$$GX \quad \hookrightarrow \quad X \times_\tau GX \quad \longrightarrow\!\!\!\!\!\rightarrow \quad X,$$

is defined by the trivial application τ, where $\tau(x)$ is the *letter* x in GX. This twisted cartesian product has the same simplices as the non-twisted one. So both associated chain complexes are equal as graded modules. The essential difference resides in the differential morphism. More precisely, the face operator ∂_n behaves differently in the twisted cartesian product. Let $(x, g) \in X \times_\tau GX$, the rule for the face operators are:

$$\begin{aligned} \partial_i(x,g) &= (\partial_i x, \partial_i g), \qquad i < n, \\ \partial_n(x,g) &= (\partial_n x, \tau(x).\partial_n g), \qquad i = n, \end{aligned}$$

where $\tau(x)$ is the *letter* x in GX and '.' is the group product in GX. Let us recall that, from the mathematical definition of GX, if x is n–degenerate, for instance $x = \eta_n x'$, the $\tau(x)$ is the null element of GX_n. The behaviour of ∂_n induces a change in the differential morphism of the chain complex associated to the twisted cartesian product.

[1] **Daniel M. Kan**. *A combinatorial definition of homotopy groups*, Ann. of Math., 1958, vol. 67, pp 282–312.

15.2 The associated tensor twisted product

We recall that around the cartesian product $X \times GX$, the Eilenberg–Zilber theorem allows to build the reduction (see the function `ez`):

$$\begin{array}{ccc} \mathcal{C}_*(X \times GX) & \stackrel{h}{\longrightarrow} & {}^s\mathcal{C}_*(X \times GX) \\ f \downarrow\uparrow g & & \\ \mathcal{C}_*(X) \otimes \mathcal{C}_*(GX) & & \end{array}$$

On the other hand, the differential morphism d_τ of the chain complex $\mathcal{C}_*(X \times_\tau GX)$ may be considered as a perturbed differential morphism d of the chain complex $\mathcal{C}_*(X \times GX)$, the perturbation δ being: $\delta = d_\tau - d$. Using the basic perturbation lemma, we may construct a new reduction:

$$\begin{array}{ccc} \mathcal{C}_*(X \times_\tau GX) & \stackrel{h}{\longrightarrow} & {}^s\mathcal{C}_*(X \times_\tau GX) \\ f \downarrow\uparrow g & & \\ \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX) & & \end{array}$$

In fact, the interesting object produced by this machinery is the bottom chain complex $\mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX)$ which is nothing but the *twisted tensor product*[2].

15.3 Main functions for the twisted products

`twisted-crts-prdc` *space* *[Function]*

From the simplicial set X, here the argument *space*, return the twisted cartesian product $X \times_\tau GX$ (a simplicial set). This function calls internally the functions `loop-space` and `crts-prdc` described in preceding chapters. The slot `grmd` of the resulting instance is set with the ordinary cartesian product $X \times GX$.

[2] **Edgar H. Brown Jr.** *Twisted tensor products, I*, Ann. of Math., 1959, vol. 69, pp. 223-246.

`dtau-d` *space* *[Function]*

From the simplicial set X, here the argument *space*, build the morphism perturbation (degree -1)

$$d_\tau - d : X \times GX \longrightarrow X \times GX.$$

The work is essentialy done by the lisp function `dtau-d-intr` which computes the difference between the differential morphisms taking in account the discrepancy between the last face operator (∂_n) in both chain complexes, $X \times_\tau GX$ and $X \times GX$.

`szczarba` *space* *[Function]*

Starting from the simplicial set X, (the argument *space*), return two values:

1- The reduction

$$\begin{array}{ccc} \mathcal{C}_*(X \times_\tau GX) & \xrightarrow{h} & {}^s\mathcal{C}_*(X \times_\tau GX) \\ f \downarrow\uparrow g & & \\ \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX) & & \end{array}$$

obtained by perturbing by $d_\tau - d$ the top chain complex of the reduction

$$\begin{array}{ccc} \mathcal{C}_*(X \times GX) & \xrightarrow{h} & {}^s\mathcal{C}_*(X \times GX) \\ f \downarrow\uparrow g & & \\ \mathcal{C}_*(X) \otimes \mathcal{C}_*(GX) & & \end{array} .$$

2- The morphism corresponding to the perturbation of the bottom chain complex induced by the perturbation $d_\tau - d$ upon the top chain complex of the above reduction.

This algorithm implements in particular in a very efficient way the formulas of Szczarba[3].

[3] **R.H. Szczarba.** *The homology of twisted cartesian products.* Transactions of the American Math. Society, 1961, vol. 100, pp. 197-216

`twisted-tnsr-prdc` *space* *[Function]*

Return the twisted tensor product (a chain complex) $X \otimes_t GX$. This is done simply by extraction of the bottom chain complex from the reduction (`szczarba` *space*).

Examples

For testing the functions above, we shall use a "soft" version of $\bar{\Delta}$ analogous to `soft-delta` or `soft-delta-infinity`. The input of the simplices will be done in clear with the help of the macro `code`. As this version does not exists in `Kenzo`, we have modified some functions of `soft-delta`, as indicated.

```
(DEFUN SOFT-DELTAB-CMPR (gmsm1 gmsm2)
    (if (= 1 (logcount (cdr gmsm1)))
        :EQUAL
        (f-cmpr (cdr gmsm1)(cdr gmsm2)) ))

(DEFUN SOFT-DELTAB-BNDR (dmns gmsm)
   (declare
      (fixnum dmns)
      (type soft-dlop gmsm))
   (the cmbn
      (if (< dmns 2)
         (zero-cmbn (1- dmns))
         (make-cmbn
         :degr (1- dmns)
         :list (mapcar #'(lambda (term)
                         (with-term (cffc gmsm) term
                            (term cffc (d gmsm))))
               (cmbn-list (delta-bndr dmns (cdr gmsm))))))))

(DEFUN SOFT-DELTAB ()
   (the simplicial-set
      (build-smst
         :cmpr #'soft-deltab-cmpr
         :basis :locally-effective
         :bspn (d 1)
         :face #'soft-delta-face
;;;      :intr-dgnl #'soft-delta-dgnl :dgnl-strt :gnrt
         :intr-bndr #'soft-deltab-bndr :bndr-strt :gnrt
         :orgn `(soft-deltab))))


(defmacro code (arg) `(d (dlop-ext-int ,arg)))
```

```
(setf db (soft-deltab))  ==>

[K1 Simplicial-Set]

(setf gdb (loop-space db))  ==>

[K6 Simplicial-Group]
```

Let us compare the faces of a cartesian product in $\bar{\Delta} \times \Omega(\bar{\Delta})$ and the faces of the same object in $\bar{\Delta} \times_\tau \Omega(\bar{\Delta})$:

```
(setf crts-pr (crts-prdc db gdb))  ==>

[K18 Simplicial-Set]

(setf tw-crts-pr (twisted-crts-prdc db))  ==>

[K23 Simplicial-Set]

(setf smpx (crpr 0 (code '(0 1 2 3 4))
                 0 (loop3 0 (code '(0 1 2 3 4 5)) 1)))  ==>

<CrPr - 0-1-2-3-4 - <<Loop[0-1-2-3-4-5]>>>

(dotimes (j 5)
         (print (face crts-pr j 4 smpx)))  ==>

<AbSm - <CrPr - 1-2-3-4 - <<Loop[1-2-3-4-5]>>>>
<AbSm - <CrPr - 0-2-3-4 - <<Loop[0-2-3-4-5]>>>>
<AbSm - <CrPr - 0-1-3-4 - <<Loop[0-1-3-4-5]>>>>
<AbSm - <CrPr - 0-1-2-4 - <<Loop[0-1-2-4-5]>>>>
<AbSm - <CrPr - 0-1-2-3 - <<Loop[0-1-2-3-4\-1][0-1-2-3-5]>>>>

(dotimes (j 5)
         (print (face tw-crts-pr j 4 smpx)))  ==>

<AbSm - <CrPr - 1-2-3-4 - <<Loop[1-2-3-4-5]>>>>
<AbSm - <CrPr - 0-2-3-4 - <<Loop[0-2-3-4-5]>>>>
<AbSm - <CrPr - 0-1-3-4 - <<Loop[0-1-3-4-5]>>>>
<AbSm - <CrPr - 0-1-2-4 - <<Loop[0-1-2-4-5]>>>>
<AbSm - <CrPr - 0-1-2-3 - <<Loop[0-1-2-3-5]>>>>
```

```
(setf smpx2 (crpr 0 (code '(1 2 3 4 5))
                  0 (loop3 0 (code '(5 6 7 8 9 10)) 1)))  ==>

<CrPr - 1-2-3-4-5 - <<Loop[5-6-7-8-9-10]>>>

(dotimes (j 5)
         (print (face crts-pr j 4 smpx2)))  ==>

<AbSm - <CrPr - 2-3-4-5 - <<Loop[6-7-8-9-10]>>>>
<AbSm - <CrPr - 1-3-4-5 - <<Loop[5-7-8-9-10]>>>>
<AbSm - <CrPr - 1-2-4-5 - <<Loop[5-6-8-9-10]>>>>
<AbSm - <CrPr - 1-2-3-5 - <<Loop[5-6-7-9-10]>>>>
<AbSm - <CrPr - 1-2-3-4 - <<Loop[5-6-7-8-9\-1][5-6-7-8-10]>>>>

(dotimes (j 5)
         (print (face tw-crts-pr j 4 smpx2)))  ==>

<AbSm - <CrPr - 2-3-4-5 - <<Loop[6-7-8-9-10]>>>>
<AbSm - <CrPr - 1-3-4-5 - <<Loop[5-7-8-9-10]>>>>
<AbSm - <CrPr - 1-2-4-5 - <<Loop[5-6-8-9-10]>>>>
<AbSm - <CrPr - 1-2-3-5 - <<Loop[5-6-7-9-10]>>>>
<AbSm - <CrPr - 1-2-3-4 - <<Loop[1-2-3-4-5][5-6-7-8-9\-1][5-6-7-8-10]>>>>
```

The user will have noted that we may obtain the same twisted cartesian product by the fibration theory. Let us define a general function `loop-fbr` with a reduced simplicial set, *space*, as argument. This function uses the function `fibration-total` that we have seen in the chapter about fibrations. Here, we see that the lisp function for the simplicial morphism, (keyword `:sintr`), implements straightforwardly the canonical twisting operator which applies a simplex of X to the corresponding 1–letter word of GX. Note that the execution time by the fibration function is a little longer than for the direct twisted cartesian product and it requires more memory.

```
(defun loop-fbr (space)
            (fibration-total
               (build-smmr
                 :sorc space
                 :trgt (loop-space space)
                 :degr -1
                 :sintr #'(lambda (dmns gmsm)
                             (absm 0 (loop3 0 gmsm 1)))
                 :orgn `(total-fibration ,space))))


(setf fb-tt-db (loop-fbr db))  ==>

[K46 Simplicial-Set]
```

```
(dotimes (j 5)
         (print(face fb-tt-db j 4 smpx2)))

<AbSm - <CrPr - 2-3-4-5 - <<Loop[6-7-8-9-10]>>>>
<AbSm - <CrPr - 1-3-4-5 - <<Loop[5-7-8-9-10]>>>>
<AbSm - <CrPr - 1-2-4-5 - <<Loop[5-6-8-9-10]>>>>
<AbSm - <CrPr - 1-2-3-5 - <<Loop[5-6-7-9-10]>>>>
<AbSm - <CrPr - 1-2-3-4 - <<Loop[1-2-3-4-5][5-6-7-8-9\-1][5-6-7-8-10]>>>>

(time (dotimes (j 1000) (face tw-crts-pr 4 4 smpx2)))  ==>

; cpu time (non-gc) 143,480 msec (00:02:23.480) user, 40 msec system
; cpu time (gc)     2,140 msec user, 40 msec system
; cpu time (total)  145,620 msec (00:02:25.620) user, 80 msec system
; real time  146,899 msec (00:02:26.899)
; space allocation:
;  16,122,082 cons cells, 11,000 symbols, 6,226,992 other bytes

(time (dotimes (j 1000) (face fb-tt-db 4 4 smpx2)))  ==>

; cpu time (non-gc) 215,190 msec (00:03:35.190) user, 40 msec system
; cpu time (gc)     3,630 msec user, 50 msec system
; cpu time (total)  218,820 msec (00:03:38.820) user, 90 msec system
; real time  221,946 msec (00:03:41.946)
; space allocation:
;  23,859,004 cons cells, 17,000 symbols, 10,536,128 other bytes
```

The four following statements show the relationship between the differentials:

```
(setf perturb (dtau-d db))  ==>

[K51 Morphism (degree -1)]

(? perturb 4 smpx2)  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <CrPr - 1-2-3-4 - <<Loop[1-2-3-4-5][5-6-7-8-9\-1][5-6-7-8-10]>>>>
<-1 * <CrPr - 1-2-3-4 - <<Loop[5-6-7-8-9\-1][5-6-7-8-10]>>>>
------------------------------------------------------------------------------
```

```
(? crts-pr 4 smpx2)  ==>

-------------------------------------------------------------------------{CMBN 3}
<1 * <CrPr - 1-2-3-4 - <<Loop[5-6-7-8-9\-1][5-6-7-8-10]>>>>
<-1 * <CrPr - 1-2-3-5 - <<Loop[5-6-7-9-10]>>>>
<1 * <CrPr - 1-2-4-5 - <<Loop[5-6-8-9-10]>>>>
<-1 * <CrPr - 1-3-4-5 - <<Loop[5-7-8-9-10]>>>>
<1 * <CrPr - 2-3-4-5 - <<Loop[6-7-8-9-10]>>>>
-------------------------------------------------------------------------------

(? tw-crts-pr 4 smpx2)  ==>

-------------------------------------------------------------------------{CMBN 3}
<1 * <CrPr - 1-2-3-4 - <<Loop[1-2-3-4-5][5-6-7-8-9\-1][5-6-7-8-10]>>>>
<-1 * <CrPr - 1-2-3-5 - <<Loop[5-6-7-9-10]>>>>
<1 * <CrPr - 1-2-4-5 - <<Loop[5-6-8-9-10]>>>>
<-1 * <CrPr - 1-3-4-5 - <<Loop[5-7-8-9-10]>>>>
<1 * <CrPr - 2-3-4-5 - <<Loop[6-7-8-9-10]>>>>
-------------------------------------------------------------------------------
```

The same type of tests may be done with another reduced simplicial set, namely $P^3\mathbb{R}$.

```
(setf pri3 (R-proj-space 3))  ==>

[K52 Simplicial-Set]

(setf tw-pri3 (twisted-crts-prdc pri3))  ==>

[K74 Simplicial-Set]

(basis tw-pri3)  ==>

:LOCALLY-EFFECTIVE

(setf crt-pri3 (crts-prdc pri3 (loop-space pri3)))  ==>

[K69 Simplicial-Set]

(setf s (crpr 0 4 0 (loop3 0 5 1)))  ==>

<CrPr - 4 - <<Loop[5]>>>
```

```
(dotimes (i 5) (print (face crt-pri3 i 4 s)))  ==>

<AbSm - <CrPr - 3 - <<Loop[4]>>>>
<AbSm 0 <CrPr 1-0 0 - <<Loop[3]>>>>
<AbSm 1 <CrPr 1-0 0 - <<Loop[3]>>>>
<AbSm 2 <CrPr 1-0 0 - <<Loop[3]>>>>
<AbSm - <CrPr - 3 - <<Loop[4\-1]>>>>

(dotimes (i 5) (print (face tw-pri3 i 4 s)))  ==>

<AbSm - <CrPr - 3 - <<Loop[4]>>>>
<AbSm 0 <CrPr 1-0 0 - <<Loop[3]>>>>
<AbSm 1 <CrPr 1-0 0 - <<Loop[3]>>>>
<AbSm 2 <CrPr 1-0 0 - <<Loop[3]>>>>
<AbSm - <CrPr - 3 2-1-0 <<Loop>>>>

(setf fb-tt-pri3 (loop-fbr pri3))  ==>

[K80 Simplicial-Set]

(dotimes (j 5) (print (face fb-tt-pri3 j 4 s)))

<AbSm - <CrPr - 3 - <<Loop[4]>>>>
<AbSm 0 <CrPr 1-0 0 - <<Loop[3]>>>>
<AbSm 1 <CrPr 1-0 0 - <<Loop[3]>>>>
<AbSm 2 <CrPr 1-0 0 - <<Loop[3]>>>>
<AbSm - <CrPr - 3 2-1-0 <<Loop>>>>
```

Let us check now the Szczarba reduction, with the simplicial set $\bar{\Delta}$.

```
(setf szc-reduc (szczarba db))  ==>

[K107 Reduction]

(pre-check-rdct szc-reduc)  ==>

---done---

(setf *tc* (cmbn 2 1 (crpr 0 (code '(0 1 2))
                           0 (loop3 0 (code '(5 6 7 8)) 1))))

------------------------------------------------------------------------{CMBN 2}
<1 * <CrPr - 0-1-2 - <<Loop[5-6-7-8]>>>>
------------------------------------------------------------------------

(setf *bc* (cmbn 2 1 (tnpr 0 (code '(0)) 2 (loop3 0 (code '(5 6 7 8)) 1))
                 10 (tnpr 1 (code '(10 11)) 1 (loop3 0 (code '(0 1 2)) 1))
                100 (tnpr 2 (code '(0 1 2)) 0 (loop3 0 (code '(10 11)) 1))))
```

```
-------------------------------------------------------------------------{CMBN 2}
<1 * <TnPr 0 <<Loop[5-6-7-8]>>>>
<10 * <TnPr 10-11 <<Loop[0-1-2]>>>>
<100 * <TnPr 0-1-2 <<Loop[10-11]>>>>
-------------------------------------------------------------------------------

(pre-check-rdct szc-reduc)  ==>

---done---

(check-rdct)  ==>

*TC* =>
-------------------------------------------------------------------------{CMBN 2}
<1 * <CrPr - 0-1-2 - <<Loop[5-6-7-8]>>>>
-------------------------------------------------------------------------------

*BC* =>
-------------------------------------------------------------------------{CMBN 2}
<1 * <TnPr 0 <<Loop[5-6-7-8]>>>>
<10 * <TnPr 10-11 <<Loop[0-1-2]>>>>
<100 * <TnPr 0-1-2 <<Loop[10-11]>>>>
-------------------------------------------------------------------------------

Checking *TDD* = 0
Result:
-------------------------------------------------------------------------{CMBN 0}
-------------------------------------------------------------------------------

              ....... All results null ......

Checking *HG* = 0
Result:
-------------------------------------------------------------------------{CMBN 3}
-------------------------------------------------------------------------------

---done---
```

15.4 Exercice with the twisting cochain.

Let B be a reduced simplicial set and its loop space GB (a simplicial group). In the following examples, we shall use the efficient `Kenzo` version of the simplicial set $\bar{\Delta}$, i.e. the simplices are input and printed under their binary form coding.

15.4.1 The twisting cochain.

The twisting cochain t is the morphism

$$t : aug \circ d_{\otimes t} \circ coaug$$

where,

1. $coaug$ is the morphism

$$coaug : B \to B \otimes GB,$$

$$coaug(b) = b \otimes 1_{GB}.$$

2. $d_{\otimes t}$ is the differential morphism of the twisted tensor product $B \otimes_t GB$.

3. aug is the morphism

$$aug : B \otimes GB \longrightarrow GB,$$

$$\begin{aligned} aug(b_i \otimes g_j) &= g_j, \qquad b_i \in B_0, \\ aug(b_i \otimes g_j) &= 0, \qquad b_i \in B_k,\ k \neq 0. \end{aligned}$$

We recall that, as graded modules, $B \otimes GB$ and $B \otimes_t GB$ are identical. First, we create $\bar{\Delta}$ and $G\bar{\Delta}$. The function `twisted-tnsr-prdc` extracts the twisted tensor product $\bar{\Delta} \otimes_t G\bar{\Delta}$ from the reduction obtained by a call to the function `szczarba`.

```
(setf d (deltab)) ==>

[K1 Simplicial-Set]

(setf gd (loop-space d))  ==>

[K6 Simplicial-Group]

(setf d-twtp-gd (twisted-tnsr-prdc d))  ==>

[K49 Chain-Complex]
```

Let us define the coaugmentation and augmentation morphisms. As usual, it is sufficient to define the algorithm on a generator.

```
(setf coaug (build-mrph
               :sorc d
               :trgt d-twtp-gd
               :degr 0
               :intr #'(lambda (degr gnrt)
                         (term-cmbn degr 1 (tnpr degr gnrt 0 +null-loop+)))
               :strt :gnrt
               :orgn '(coaug. d --> d-twtp-gd)))  ==>

[K52 Morphism (degree 0)]

(? coaug 3 15)  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 15 <<Loop>>>>
------------------------------------------------------------------------

(? coaug (cmbn 4 1 31 1 62 1 124))  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr 31 <<Loop>>>>
<1 * <TnPr 62 <<Loop>>>>
<1 * <TnPr 124 <<Loop>>>>
------------------------------------------------------------------------

(setf aug (build-mrph
             :sorc d-twtp-gd
             :trgt gd
             :degr 0
             :intr #'(lambda (degr tnpr)
                       (with-tnpr (degr1 gmsm1 degr2 loop2) tnpr
                           (if (zerop degr1)
                               (term-cmbn degr 1 loop2)
                               (zero-cmbn degr))))
             :strt :gnrt
             :orgn '(aug. of d-twtp-gd)))  ==>

[K53 Morphism (degree 0)]

(? aug (cmbn 2 9 (tnpr 0 1 2 (loop3 0 15 1))
               3 (tnpr 1 3 1 (loop3 0 7 1))))  ==>

------------------------------------------------------------------------{CMBN 2}
<9 * <<Loop[15]>>>
------------------------------------------------------------------------
```

The morphism `cochain` is simply a composition of morphisms. We apply it upon the simplices `(0)`, `(0 1 2)`, `(0 1 2 3)`, ..., coded as the integers $1, 7, 15, \ldots$.

```
(setf cochain (i-cmps aug (dffr d-twtp-gd) coaug))  ==>

[K55 Morphism (degree -1)]

(? cochain 0 1)  ==>

-------------------------------------------------------------------------{CMBN -1}
-------------------------------------------------------------------------------

(? cochain 2 7)  ==>

-------------------------------------------------------------------------{CMBN 1}
<1 * <<Loop[0 3][7]>>>
-------------------------------------------------------------------------------

(? cochain 3 15)  ==>

-------------------------------------------------------------------------{CMBN 2}
<1 * <<Loop[1-0 3][0 7][1 13]>>>
<-1 * <<Loop[1-0 3][1 7][15]>>>
-------------------------------------------------------------------------------

(? cochain 4 31)  ==>

-------------------------------------------------------------------------{CMBN 3}
<-1 * <<Loop[2-1-0 3][1-0 7][2-0 13][2-1 25]>>>
<1 * <<Loop[2-1-0 3][1-0 7][2-1 13][2 29]>>>
<1 * <<Loop[2-1-0 3][2-0 7][0 15][2-1 25]>>>
<-1 * <<Loop[2-1-0 3][2-0 7][2-1 13][1 29]>>>
<-1 * <<Loop[2-1-0 3][2-1 7][1 15][2 27]>>>
<1 * <<Loop[2-1-0 3][2-1 7][2 15][31]>>>
-------------------------------------------------------------------------------

(? cochain 5 63)  ==>

-------------------------------------------------------------------------{CMBN 4}
<-1 * <<Loop[3-2-1-0 3][2-1-0 7][3-1-0 13][3-2-0 25][3-2-1 49]>>>
<1 * <<Loop[3-2-1-0 3][2-1-0 7][3-1-0 13][3-2-1 25][3-2 57]>>>
<1 * <<Loop[3-2-1-0 3][2-1-0 7][3-2-0 13][3-0 29][3-2-1 49]>>>
<-1 * <<Loop[3-2-1-0 3][2-1-0 7][3-2-0 13][3-2-1 25][3-1 57]>>>
<-1 * <<Loop[3-2-1-0 3][2-1-0 7][3-2-1 13][3-1 29][3-2 53]>>>
<1 * <<Loop[3-2-1-0 3][2-1-0 7][3-2-1 13][3-2 29][3 61]>>>
<1 * <<Loop[3-2-1-0 3][3-1-0 7][1-0 15][3-2-0 25][3-2-1 49]>>>
<-1 * <<Loop[3-2-1-0 3][3-1-0 7][1-0 15][3-2-1 25][3-2 57]>>>
```

```
<-1 * <<Loop[3-2-1-0 3][3-1-0 7][3-2-0 13][2-0 29][3-2-1 49]>>>
<1 * <<Loop[3-2-1-0 3][3-1-0 7][3-2-0 13][3-2-1 25][2-1 57]>>>
<1 * <<Loop[3-2-1-0 3][3-1-0 7][3-2-1 13][2-1 29][3-2 53]>>>
<-1 * <<Loop[3-2-1-0 3][3-1-0 7][3-2-1 13][3-2 29][2 61]>>>
<-1 * <<Loop[3-2-1-0 3][3-2-0 7][2-0 15][3-0 27][3-2-1 49]>>>
<1 * <<Loop[3-2-1-0 3][3-2-0 7][2-0 15][3-2-1 25][3-1 57]>>>
<1 * <<Loop[3-2-1-0 3][3-2-0 7][3-0 15][0 31][3-2-1 49]>>>
<-1 * <<Loop[3-2-1-0 3][3-2-0 7][3-0 15][3-2-1 25][2-1 57]>>>
<-1 * <<Loop[3-2-1-0 3][3-2-0 7][3-2-1 13][2-1 29][3-1 53]>>>
<1 * <<Loop[3-2-1-0 3][3-2-0 7][3-2-1 13][3-1 29][1 61]>>>
<1 * <<Loop[3-2-1-0 3][3-2-1 7][2-1 15][3-1 27][3-2 51]>>>
<-1 * <<Loop[3-2-1-0 3][3-2-1 7][2-1 15][3-2 27][3 59]>>>
<-1 * <<Loop[3-2-1-0 3][3-2-1 7][3-1 15][1 31][3-2 51]>>>
<1 * <<Loop[3-2-1-0 3][3-2-1 7][3-1 15][3-2 27][2 59]>>>
<1 * <<Loop[3-2-1-0 3][3-2-1 7][3-2 15][2 31][3 55]>>>
<-1 * <<Loop[3-2-1-0 3][3-2-1 7][3-2 15][3 31][63]>>>
--------------------------------------------------------------------------------
```

We verify, that for a simplex in dimension n, the length of the resulting combination is $(n-1)!$.

```
(length(cmbn-list *))  ==>

24

(? cochain 6 127)  ==>

--------------------------------------------------------------------------{CMBN 5}
<1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-2-1-0 13][4-3-1-0 25][4-3-2-0 49][4-3-2-1 97]>>>
<-1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-2-1-0 13][4-3-1-0 25][4-3-2-1 49][4-3-2 113]>>>
<-1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-2-1-0 13][4-3-2-0 25][4-3-0 57][4-3-2-1 97]>>>
<1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-2-1-0 13][4-3-2-0 25][4-3-2-1 49][4-3-1 113]>>>
<1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-2-1-0 13][4-3-2-1 25][4-3-1 57][4-3-2 105]>>>
<-1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-2-1-0 13][4-3-2-1 25][4-3-2 57][4-3 121]>>>
<-1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-3-1-0 13][4-1-0 29][4-3-2-0 49][4-3-2-1 97]>>>
<1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-3-1-0 13][4-1-0 29][4-3-2-1 49][4-3-2 113]>>>
<1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-3-1-0 13][4-3-2-0 25][4-2-0 57][4-3-2-1 97]>>>
<-1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-3-1-0 13][4-3-2-0 25][4-3-2-1 49][4-2-1 113]>>>
<-1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-3-1-0 13][4-3-2-1 25][4-2-1 57][4-3-2 105]>>>
<1 * <<Loop[4-3-2-1-0 3][3-2-1-0 7][4-3-1-0 13][4-3-2-1 25][4-3-2 57][4-2 121]>>>
... ...
--------------------------------------------------------------------------------

(length(cmbn-list *))  ==>

120
```

15.4.2 Expression of the differential $d_{\otimes t}$

The differential $d_{\otimes t}$ of the twisted tensor product $B \otimes_t GB$ may be expressed as the sum of the differential of the simple tensor product $B \otimes GB$ and the following morphism of degree -1,

$$\nabla : B \otimes GB \longrightarrow B \otimes GB,$$

where ∇ is the composition of the morphisms of the following sequence:

$$B \otimes GB \stackrel{\Delta \otimes 1_{GB}}{\longrightarrow} (B \otimes B) \otimes GB \stackrel{assoc}{\longrightarrow} B \otimes (B \otimes GB) \stackrel{1_B \otimes (t \otimes 1_{GB})}{\longrightarrow} B \otimes (GB \otimes GB) \stackrel{1_B \otimes \varpi}{\longrightarrow} B \otimes GB.$$

In the previous formula, Δ is the coproduct in the coalgebra B, ϖ is the product in the algebra GB, 1_B is the identity morphism in B and 1_{GB} the identity morphism in GB. We need also, as in a previous example, an "assoc" morphism for the compound tensor product. The user will note the easiness for coding in `Kenzo` such a long composition of morphisms.

```
(setf 3-left (tnsr-prdc (tnsr-prdc d d) gd))  ==>

[K56 Chain-Complex]

(setf 3-right (tnsr-prdc d (tnsr-prdc d gd)))  ==>

[K58 Chain-Complex]

(setf assoc (build-mrph
               :sorc 3-left
               :trgt 3-right
               :degr 0
               :intr #'(lambda (degr a2-a)
                         (with-tnpr (degra2 gnrta2 degra gnrta) a2-a
                            (with-tnpr (degr1 gnrt1 degr2 gnrt2) gnrta2
                               (cmbn (+ degr1 degr2 degra)
                                     1 (tnpr degr1
                                             gnrt1
                                             (+ degr2 degra)
                                             (tnpr degr2 gnrt2 degra gnrta))))))
               :strt :gnrt
               :orgn '(assoc-double-tensor-product)))   ==>

[K60 Morphism (degree 0)]

(setf id-d (idnt-mrph d))  ==>

[K61 Morphism (degree 0)]
```

```
(setf id-gd (idnt-mrph gd))  ==>

[K62 Morphism (degree 0)]

(setf nabla
 (add (dffr (tnsr-prdc d gd))
      (i-cmps (tnsr-prdc id-d (aprd gd))
              (tnsr-prdc id-d (tnsr-prdc cochain id-gd))
              assoc
              (tnsr-prdc (cprd d) id-gd))))  ==>

[K74 Morphism (degree -1)]
```

We verify that the morphism `nabla` and the differential of the twisted tensor product (accessible through the symbol `d-twtp-gd`) return the same results:

```
(? (dffr d-twtp-gd) 2 (tnpr 1 3 1 (loop3 0 7 1)))  ==>

-------------------------------------------------------------------------{CMBN 1}
<1 * <TnPr 1 <<Loop[7]>>>>
<-1 * <TnPr 1 <<Loop[0 3][7]>>>>
<1 * <TnPr 3 <<Loop[3\-1][5]>>>>
<-1 * <TnPr 3 <<Loop[6]>>>>
-------------------------------------------------------------------------------

(? nabla 2 (tnpr 1 3 1 (loop3 0 7 1)))  ==>

-------------------------------------------------------------------------{CMBN 1}
<1 * <TnPr 1 <<Loop[7]>>>>
<-1 * <TnPr 1 <<Loop[0 3][7]>>>>
<1 * <TnPr 3 <<Loop[3\-1][5]>>>>
<-1 * <TnPr 3 <<Loop[6]>>>>
-------------------------------------------------------------------------------

(setf zero (sbtr nabla (dffr d-twtp-gd)))  ==>

[K75 Morphism (degree -1)]

(? zero 2 (tnpr 1 3 1 (loop3 0 7 1)))  ==>

-------------------------------------------------------------------------{CMBN 1}
-------------------------------------------------------------------------------

(? zero 5 (tnpr  3 15 2 (loop3 0 15 1)))  ==>

-------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------
```

15.4.3 The cup product of the twisting cochain

The cup product $t \sqcup t$ of the twisting cochain t,

$$t \sqcup t : B \longrightarrow GB$$

is defined by the following composition of morphisms:

$$B \xrightarrow{\Delta} B \otimes B \xrightarrow{t \otimes t} GB \otimes GB \xrightarrow{\varpi} GB.$$

We may now verify the formula

$$d_{GB} \circ t + t \circ d_B + t \sqcup t = 0,$$

where d_B and d_{GB} are the respective differential morphisms in B and GB.

```
(setf t-cup-t (i-cmps (aprd gd) (tnsr-prdc cochain cochain) (cprd d)))  ==>

[K78 Morphism (degree -2)]

(setf dt (cmps (dffr gd) cochain))  ==>

[K79 Morphism (degree -2)]

(setf td (cmps cochain (dffr d)))  ==>

[K80 Morphism (degree -2)]

(setf zero-2 (i-add dt td t-cup-t))  ==>

[K82 Morphism (degree -2)]

(? zero-2 3 15)  ==>

------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

(? zero-2 4 31)  ==>

------------------------------------------------------------------------{CMBN 2}
------------------------------------------------------------------------------

(? zero-2 5 63)  ==>

------------------------------------------------------------------------{CMBN 3}
------------------------------------------------------------------------------
```

```
(? zero-2 6 127)  ==>

-------------------------------------------------------------------------{CMBN 4}
-------------------------------------------------------------------------------
```

So the user has easily verified the *cochain condition* for the twisting cochain t.

15.5 The essential contraction

It is known that if B is a reduced simplicial set, the space $B \times_\tau GB$ is contractible. So it is possible to build a reduction of this space over $\mathbb{Z}$. This reduction depends of the important contraction

$$\chi_{\times\tau} : \mathcal{C}_*(B) \times_\tau \mathcal{C}_*(GB) \longrightarrow \mathcal{C}_*(B) \times_\tau \mathcal{C}_*(GB),$$

which is a homotopy operator (degree: $+1$) satisfying the relation

$$d \circ \chi_{\times\tau} + \chi_{\times\tau} \circ d = 1.$$

Now, from the reduction

$$\begin{array}{ccc} \mathcal{C}_*(X \times_\tau GX) & \stackrel{h}{\longrightarrow} & {}^s\mathcal{C}_*(X \times_\tau GX) \\ f \downarrow\uparrow g & & \\ \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX) & & \end{array}$$

there is also an induced contraction

$$\chi_{\otimes t} : \mathcal{C}_*(B) \otimes_t \mathcal{C}_*(GB) \longrightarrow \mathcal{C}_*(B) \otimes_t \mathcal{C}_*(GB),$$

defined by $\chi_{\otimes t} = f \circ \chi_{\times\tau} \circ g$. This implies that the twisted tensor product is also contractible over $\mathbb{Z}$.
The two functions of `Kenzo` for this construction are:

`crts-contraction` *space* *[Function]*
Return the homotopy morphism corresponding to the contraction $\chi_{\times\tau}$. The lisp definition is:

```
(defun CRTS-CONTRACTION
   (space
      &aux (twisted-crts-prdc (twisted-crts-prdc space)))
   (the morphism
      (build-mrph
         :sorc twisted-crts-prdc
         :trgt twisted-crts-prdc
         :degr +1
         :intr (crts-contraction-intr
                    (cmpr space)
```

```
                        (bspn space)
                        (face space)
                        (cmpr twisted-crts-prdc))
              :strt :gnrt
              :orgn `(crts-contraction ,space))))
```

At execution time, the work is essentialy done by the function put in the slot `:intr`.

`tnpr-contraction` *space* *[Function]*
Return the induced morphism $\chi_{\otimes t} = f \circ \chi_{\times \tau} \circ g$, as shown by the lisp definition:

```
(defun TNPR-CONTRACTION
   (space
      &aux (szczarba (szczarba space))
           (f (f szczarba))
           (g (g szczarba))
           (crts-contraction (crts-contraction space)))

   (the morphism
      (i-cmps f crts-contraction g)))
```

Examples

Let us use again our familiar simplicial set $\bar{\Delta}$. First we apply the homotopy operator $\chi_{\times t}$ upon various generators. Note that the binary coding of the simplices allows normal arithmetic operators to generate them.

```
(setf delta (deltab))  ==>

[K1 Simplicial-Set]

(setf chi-x (crts-contraction delta))  ==>

[K28 Morphism (degree 1): K23 -> K23]

(? chi-x 0 (crpr 0 1 0 +null-loop+))  ==>

------------------------------------------------------------------------{CMBN 1}
------------------------------------------------------------------------------

(? chi-x 0 (crpr 0 1 0 (loop3 0 96 1)))  ==>

------------------------------------------------------------------------{CMBN 1}
<-1 * <CrPr - 96 0 <<Loop>>>>
------------------------------------------------------------------------------
```

```
(? chi-x 0 (crpr 0 1 0 (loop3 0 96 1 0 (+ 256 128) 1)))  ==>

-------------------------------------------------------------------------{CMBN 1}
<-1 * <CrPr - 96 0 <<Loop[384]>>>>
<-1 * <CrPr - 384 0 <<Loop>>>>
-------------------------------------------------------------------------------

(? chi-x 0 (crpr 0 1 0 (loop3 0 96 1 0 (+ 256 128) 1 0 (+ 512 1024) 1)))  ==>

-------------------------------------------------------------------------{CMBN 1}
<-1 * <CrPr - 96 0 <<Loop[384][1536]>>>>
<-1 * <CrPr - 384 0 <<Loop[1536]>>>>
<-1 * <CrPr - 1536 0 <<Loop>>>>
-------------------------------------------------------------------------------

(? chi-x 0 (crpr 0 1 0 (loop3 0 96 2)))  ==>

-------------------------------------------------------------------------{CMBN 1}
<-1 * <CrPr - 96 0 <<Loop>>>>
<-1 * <CrPr - 96 0 <<Loop[96]>>>>
-------------------------------------------------------------------------------

(? chi-x 1 (crpr 0 3 0 (loop3 0 (+ 32 64 128) 1)))  ==>

-------------------------------------------------------------------------{CMBN 2}
<1 * <CrPr - 224 1-0 <<Loop>>>>
<-1 * <CrPr 1 3 0 <<Loop[224]>>>>
<1 * <CrPr 1 96 0 <<Loop[224]>>>>
-------------------------------------------------------------------------------

(? chi-x 2 (crpr 0 7 0 (loop3 0 (+ 32 64 128 256) 1)))  ==>

-------------------------------------------------------------------------{CMBN 3}
<-1 * <CrPr - 480 2-1-0 <<Loop>>>>
<1 * <CrPr 2 7 1 <<Loop[480]>>>>
<-1 * <CrPr 2 224 1 <<Loop[480]>>>>
<-1 * <CrPr 2-1 5 0 <<Loop[480]>>>>
<1 * <CrPr 2-1 160 0 <<Loop[480]>>>>
-------------------------------------------------------------------------------

(? chi-x 3 (crpr 2 7 1 (loop3 2 (+ 32 64 128) 2 4 (+ 32 64 128) -2)))  ==>

-------------------------------------------------------------------------{CMBN 4}
<-1 * <CrPr 3-1 7 2-0 <<Loop[1 224\2][2 224\-2]>>>>
<1 * <CrPr 3-2 7 1-0 <<Loop[1 224\2][2 224\-2]>>>>
-------------------------------------------------------------------------------
```

```
(? chi-x 3 (crpr 2 7 4 (loop3 2 (+ 32 64 128) 2 1 (+ 32 64 128) -2)))  ==>

------------------------------------------------------------------------{CMBN 4}
------------------------------------------------------------------------------

(? chi-x 3 (crpr 2 7 1 (loop3 2 (+ 32 64 128) -2 4 (+ 32 64 128) 2)))  ==>

------------------------------------------------------------------------{CMBN 4}
<-1 * <CrPr 3-1 7 2-0 <<Loop[1 224\-2][2 224\2]>>>>
<1 * <CrPr 3-2 7 1-0 <<Loop[1 224\-2][2 224\2]>>>>
------------------------------------------------------------------------------

(? chi-x 3 (crpr 2 7 4 (loop3 2 (+ 32 64 128) -2 1 (+ 32 64 128) 2)))  ==>

------------------------------------------------------------------------{CMBN 4}
------------------------------------------------------------------------------

(? chi-x 3 (crpr 1 7 2 (loop3 2 (+ 32 64 128) 2 4 (+ 32 64 128) -2)))  ==>

------------------------------------------------------------------------{CMBN 4}
<-1 * <CrPr 3-0 7 2-1 <<Loop[1 224\2][2 224\-2]>>>>
------------------------------------------------------------------------------

(? chi-x 3 (crpr 4 7 2 (loop3 2 (+ 32 64 128) 2 1 (+ 32 64 128) -2)))  ==>

------------------------------------------------------------------------{CMBN 4}
------------------------------------------------------------------------------

(? chi-x 3 (crpr 1 7 2 (loop3 2 (+ 32 64 128) -2 4 (+ 32 64 128) 2)))  ==>

------------------------------------------------------------------------{CMBN 4}
<-1 * <CrPr 3-0 7 2-1 <<Loop[1 224\-2][2 224\2]>>>>
------------------------------------------------------------------------------

(? chi-x 3 (crpr 4 7 2 (loop3 2 (+ 32 64 128) -2 1 (+ 32 64 128) 2)))  ==>

------------------------------------------------------------------------{CMBN 4}
------------------------------------------------------------------------------
```

Then, for testing $\chi_{\otimes t}$ we use the image of all those generators by the function

f of the Brown reduction.

$$\begin{array}{ccc} \mathcal{C}_*(X \times_\tau GX) & \xrightarrow{h} & {}^s\mathcal{C}_*(X \times_\tau GX) \\ f \downarrow\uparrow g & & \\ \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX) & & \end{array}$$

```
(setf *tnpr-with-degrees* t)  ==>

T

(setf chi-t (tnpr-contraction delta))  ==>

[K54 Morphism (degree 1): K50 -> K50]

(setf reduc (szczarba delta))  ==>

[K52 Reduction]

(setf f (f reduc))  ==>

[K42 Morphism (degree 0): K23 -> K50]

(? f 0 (crpr 0 1 0 +null-loop+))  ==>

------------------------------------------------------------------------{CMBN 0}
<1 * <TnPr 0 1 0 <<Loop>>>>
-------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 1}
-------------------------------------------------------------------------------

(? f 0 (crpr 0 1 0 (loop3 0 96 1))) ==>

------------------------------------------------------------------------{CMBN 0}
<1 * <TnPr 0 1 0 <<Loop[96]>>>>
-------------------------------------------------------------------------------
```

```
(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 1}
<-1 * <TnPr 1 96 0 <<Loop>>>>
-------------------------------------------------------------------------------

(? f 0 (crpr 0 1 0 (loop3 0 96 1 0 (+ 256 128) 1)))  ==>

------------------------------------------------------------------------{CMBN 0}
<1 * <TnPr 0 1 0 <<Loop[96][384]>>>>
-------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 1}
<-1 * <TnPr 1 96 0 <<Loop[384]>>>>
<-1 * <TnPr 1 384 0 <<Loop>>>>
-------------------------------------------------------------------------------

(? f 0 (crpr 0 1 0 (loop3 0 96 1 0 (+ 256 128) 1 0 (+ 512 1024) 1)))  ==>

------------------------------------------------------------------------{CMBN 0}
<1 * <TnPr 0 1 0 <<Loop[96][384][1536]>>>>
-------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 1}
<-1 * <TnPr 1 96 0 <<Loop[384][1536]>>>>
<-1 * <TnPr 1 384 0 <<Loop[1536]>>>>
<-1 * <TnPr 1 1536 0 <<Loop>>>>
-------------------------------------------------------------------------------

(? f 0 (crpr 0 1 0 (loop3 0 96 2)))  ==>

------------------------------------------------------------------------{CMBN 0}
<1 * <TnPr 0 1 0 <<Loop[96\2]>>>>
-------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 1}
<-1 * <TnPr 1 96 0 <<Loop>>>>
<-1 * <TnPr 1 96 0 <<Loop[96]>>>>
-------------------------------------------------------------------------------
```

```
(? f 1 (crpr 0 3 0 (loop3 0 (+ 32 64 128) 1)))  ==>

--------------------------------------------------------------------------{CMBN 1}
<1 * <TnPr 0 1 1 <<Loop[0 3][224]>>>>
<1 * <TnPr 1 3 0 <<Loop[192]>>>>
-------------------------------------------------------------------------------

(? chi-t *)  ==>

--------------------------------------------------------------------------{CMBN 2}
<-1 * <TnPr 1 3 1 <<Loop[224]>>>>
<1 * <TnPr 1 96 1 <<Loop[224]>>>>
<1 * <TnPr 2 224 0 <<Loop>>>>
-------------------------------------------------------------------------------

(? f 2 (crpr 0 7 0 (loop3 0 (+ 32 64 128 256) 1)))  ==>

--------------------------------------------------------------------------{CMBN 2}
<-1 * <TnPr 0 1 2 <<Loop[1-0 3][0 7][1 416]>>>>
<1 * <TnPr 0 1 2 <<Loop[1-0 3][1 7][480]>>>>
<1 * <TnPr 1 3 1 <<Loop[0 6][448]>>>>
<1 * <TnPr 2 7 0 <<Loop[384]>>>>
-------------------------------------------------------------------------------

(? chi-t *)  ==>

--------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 0 1 3 <<Loop[2-1-0 3][1-0 7][2 480]>>>>
<-1 * <TnPr 0 1 3 <<Loop[2-1-0 3][2-0 7][1 480]>>>>
<-1 * <TnPr 0 32 3 <<Loop[2-1-0 96][1-0 224][2 480]>>>>
<1 * <TnPr 0 32 3 <<Loop[2-1-0 96][2-0 224][1 480]>>>>
<-1 * <TnPr 1 5 2 <<Loop[480]>>>>
<1 * <TnPr 1 160 2 <<Loop[480]>>>>
<1 * <TnPr 2 7 1 <<Loop[448]>>>>
<-1 * <TnPr 2 224 1 <<Loop[448]>>>>
<-1 * <TnPr 3 480 0 <<Loop>>>>
-------------------------------------------------------------------------------

(? f 3 (crpr 2 7 1 (loop3 2 (+ 32 64 128) 2 4 (+ 32 64 128) -2)))  ==>

--------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 0 1 3 <<Loop[2-1-0 3][2-1 7][2-0 224\2][3-0 224\-2]>>>>
-------------------------------------------------------------------------------
```

```
(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr 2 7 2 <<Loop[1 224\2][2 224\-2]>>>>
------------------------------------------------------------------------------

(? f 3 (crpr 2 7 4 (loop3 2 (+ 32 64 128) 2 1 (+ 32 64 128) -2)))  ==>

------------------------------------------------------------------------{CMBN 3}
------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 4}
------------------------------------------------------------------------------

(? f 3 (crpr 2 7 1 (loop3 2 (+ 32 64 128) -2 4 (+ 32 64 128) 2)))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 0 1 3 <<Loop[2-1-0 3][2-1 7][2-0 224\-2][3-0 224\2]>>>>
------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr 2 7 2 <<Loop[1 224\-2][2 224\2]>>>>
------------------------------------------------------------------------------

(? f 3 (crpr 2 7 4 (loop3 2 (+ 32 64 128) -2 1 (+ 32 64 128) 2)))  ==>

------------------------------------------------------------------------{CMBN 3}
------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 4}
------------------------------------------------------------------------------

(? f 3 (crpr 1 7 2 (loop3 2 (+ 32 64 128) 2 4 (+ 32 64 128) -2)))  ==>

------------------------------------------------------------------------{CMBN 3}
<-1 * <TnPr 0 1 3 <<Loop[2-1-0 3][1-0 7][2-1 224\2][3-2 224\-2]>>>>
<1 * <TnPr 0 1 3 <<Loop[2-1-0 3][2-0 7][2-1 224\2][3-1 224\-2]>>>>
------------------------------------------------------------------------------

(? chi-t *)  ==>
```

```
------------------------------------------------------------------------{CMBN 4}
--------------------------------------------------------------------------------

(? f 3 (crpr 4 7 2 (loop3 2 (+ 32 64 128) 2 1 (+ 32 64 128) -2)))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 0 1 3 <<Loop[2-1-0 3][1-0 7][2-1 224\2][2-0 224\-2]>>>>
<-1 * <TnPr 0 1 3 <<Loop[2-1-0 3][2-0 7][2-1 224\2][1-0 224\-2]>>>>
<1 * <TnPr 2 7 1 <<Loop[0 192\2][224\-2]>>>>
--------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 4}
--------------------------------------------------------------------------------

(? f 3 (crpr 1 7 2 (loop3 2 (+ 32 64 128) -2 4 (+ 32 64 128) 2)))  ==>

------------------------------------------------------------------------{CMBN 3}
<-1 * <TnPr 0 1 3 <<Loop[2-1-0 3][1-0 7][2-1 224\-2][3-2 224\2]>>>>
<1 * <TnPr 0 1 3 <<Loop[2-1-0 3][2-0 7][2-1 224\-2][3-1 224\2]>>>>
--------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 4}
--------------------------------------------------------------------------------

(? f 3 (crpr 4 7 2 (loop3 2 (+ 32 64 128) -2 1 (+ 32 64 128) 2)))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 0 1 3 <<Loop[2-1-0 3][1-0 7][2-1 224\-2][2-0 224\2]>>>>
<-1 * <TnPr 0 1 3 <<Loop[2-1-0 3][2-0 7][2-1 224\-2][1-0 224\2]>>>>
<1 * <TnPr 2 7 1 <<Loop[0 192\-2][224\2]>>>>
--------------------------------------------------------------------------------

(? chi-t *)  ==>

------------------------------------------------------------------------{CMBN 4}
--------------------------------------------------------------------------------
```

We verify now the required property of the homotopy operator $\chi_{\times\tau}$. As usual, we build by composition a morphism `zero`.

```
(setf dfr (bndr (twisted-crts-prdc delta)))   ==>

[K24 Morphism (degree -1)]
```

```
(setf zero (i-sbtr (idnt-mrph (sorc dfr)) (cmps dfr chi-x)
                   (cmps chi-x dfr)))   ==>

[K33 Morphism (degree 0)]

(setf simplx (crpr 4 7 2 (loop3 2 (+ 32 64 128) -2 1 (+ 32 64 128) 2)))  ==>

<CrPr 2 7 1 <<Loop[1 224\-2][0 224\2]>>>

(? zero 3 simplx)

-------------------------------------------------------------------------{CMBN 3}
-------------------------------------------------------------------------------
```

Then, we do the same test for the homotopy operator $\chi_{\otimes t}$.

```
(setf tw (twisted-tnsr-prdc d))  ==>

[K55 Chain-Complex]

(setf zero-2  (i-sbtr (idnt-mrph tw) (cmps tw chi-t) (cmps chi-t tw)))  ==>

[K64 Morphism (degree 0)]

(setf simpx2 (tnpr 0 1 3 (loop3 0 (mask 5) 2 0 (* 32 (mask 5)) -1)))  ==>

<TnPr 1 <<Loop[31\2][992\-1]>>>

(time (? zero-2 3 simpx2))  ==>

; cpu time (non-gc) 6,050 msec user, 260 msec system
; cpu time (gc)     450 msec user, 0 msec system
; cpu time (total)  6,500 msec user, 260 msec system
; real time  6,903 msec
; space allocation:
;  947,980 cons cells, 0 symbols, 71,344 other bytes

-------------------------------------------------------------------------{CMBN 3}
-------------------------------------------------------------------------------
```

Note nevertheless that the relation is not satisfied for the base generator $* \otimes *$, namely the tensor product of both base generators. This is due to the fact that both contractions, $\chi_{\times t}$ and $\chi_{\otimes t}$, correspond to reductions over $\mathbb{Z}$ and not on 0.

```
(setf gnrt-0 (tnpr 0 1 0 +null-loop+))  ==>

<TnPr 1 <<Loop>>>
```

```
(? zero-2 0 gnrt-0)  ==>

-------------------------------------------------------------------------{CMBN 0}
<1 * <TnPr 1 <<Loop>>>>
------------------------------------------------------------------------------
```

Let us build, for instance, the reduction of the twisted tensor product over $\mathbb{Z}$ and check it. In that case, the two homomorphisms f and g are respectively the *augmentation* and *coaugmentation* morphisms.

$$\begin{array}{ccc} \mathcal{C}_*(B) \otimes_t \mathcal{C}_*(GB) & \xrightarrow{\chi \otimes t} & {}^s\mathcal{C}_*(B) \otimes_t \mathcal{C}_*(GB) \\ {\scriptstyle aug} \downarrow \uparrow {\scriptstyle coaug} & & \\ \mathcal{C}_*(\mathbb{Z}) & & \end{array}$$

We recall that, the unit chain complex corresponding to $\mathbb{Z}$ is built by the function `z-chcm`. The unique generator in degree 0 is `:z-gnrt`.

```
(setf aug (build-mrph
            :sorc tw
            :trgt (z-chcm)
            :degr 0
            :intr #'(lambda (degr gnrt)
                      (if (zerop degr)
                          (term-cmbn 0 1 :z-gnrt)
                          (zero-cmbn degr)))
            :strt :gnrt
            :orgn '(aug tw)))  ==>

[K67 Cohomology-Class (degree 0)]

(setf coaug (build-mrph
              :sorc (z-chcm)
              :trgt tw
              :degr 0
              :intr #'(lambda (degr gnrt)
                        (if (zerop degr)
                            (term-cmbn 0 1 (tnpr 0 1 0 +null-loop+))
                            (zero-cmbn degr)))
              :strt :gnrt
              :orgn '(coaug tw)))  ==>

[K68 Morphism (degree 0)]
```

```
(setf rdct (build-rdct :f aug
                       :g coaug
                       :h Chi-t
                       :orgn '(reduction-tw-Z)))   ==>

[K69 Reduction]

(pre-check-rdct rdct)  ==>

---done---

(setf *tc* (cmbn 3 1 (tnpr 1 3 2 (loop3 0 15 2))))  ==>

-------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 3 <<Loop[15\2]>>>>
-------------------------------------------------------------------------------

(setf *bc* (cmbn 0 1 :z-gnrt))  ==>

-------------------------------------------------------------------------{CMBN 0}
<1 * Z-GNRT>
-------------------------------------------------------------------------------

(check-rdct)  ==>

*TC* =>
-------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr 3 <<Loop[15\2]>>>>
-------------------------------------------------------------------------------

*BC* =>
-------------------------------------------------------------------------{CMBN 0}
<1 * Z-GNRT>
-------------------------------------------------------------------------------

Checking *TDD* = 0
Result:
-------------------------------------------------------------------------{CMBN 1}
-------------------------------------------------------------------------------

              ....... All results null ......

Checking *HG* = 0
Result:
-------------------------------------------------------------------------{CMBN 1}
-------------------------------------------------------------------------------
```

15.6 An unproved property of the algebraic Kan contraction

Since the early version of the software (1990), and through a host of examples, an amazing algebraic property of the Kan contraction $\chi_{\otimes t}$ has been experimentally discovered. Many computations have been done of $\chi_{\otimes t}(b \otimes g)$, for various choices of $b \otimes g$ and in all these cases, as soon as $degree(b) > 0$, then the result is null. In other words, $\chi_{\otimes t}$ *is null outside the base fiber.* Up to now, no actual proof of this result has yet been given.
We show that, using the "soft" version of $\bar{\Delta}$.

```
(setf db (soft-deltab))  ==>

[K1 Simplicial-Set]

(setf tt (tnpr-contraction db))  ==>

[K54 Morphism (degree 1)]

(setf gnrt1 (tnpr 3 (code '(0 1 2 3 4))
                4 (loop3 0 (code '(0 1 2 3 4 5)) 2)))  ==>

<TnPr 3 0-1-2-3-4 4 <<Loop[0-1-2-3-4-5\2]>>>

(? tt 7 gnrt1)  ==>

-------------------------------------------------------------------------{CMBN 8}
------------------------------------------------------------------------------

(setf gnrt2 (tnpr 4 (code '(2 4 6 8 10 12))
                  5 (loop3 0 (code '(1 3 5 7 9 11 13)) 5)))  ==>

<TnPr 4 2-4-6-8-10-12  5 <<Loop[1-3-5-7-9-11-13\5]>>>

(? tt 9 gnrt2)  ==>

-------------------------------------------------------------------------{CMBN 10}
------------------------------------------------------------------------------
```

But if $degree(b) = 0$, then the result is non–null in general.

```
(? tt 2 (tnpr 0 (code '(0)) 2 (loop3 0 (code '(0 1 2 3)) 1)))  ==>

------------------------------------------------------------------------{CMBN 3}
<-1 * <TnPr 0 <<Loop[2-1-0 0-1][1-0 0-1-2][2 0-1-2-3]>>>>
<1 * <TnPr 0 <<Loop[2-1-0 0-1][2-0 0-1-2][1 0-1-2-3]>>>>
<1 * <TnPr 0-2 <<Loop[0-1-2-3]>>>>
<-1 * <TnPr 0-1-2 <<Loop[1-2-3]>>>>
<-1 * <TnPr 0-1-2-3 <<Loop>>>>
------------------------------------------------------------------------------

(? tt 4 (tnpr 0 (code '(0)) 4 (loop3 0 (code '(1 2 3 4 5 6)) 3)))  ==>

------------------------------------------------------------------------{CMBN 5}
<1 * <TnPr 1 <<Loop[4-3-2-1-0 1-2][3-2-1-0 1-2-3][4-2-1-0 1-3-4][4-3-1-0 1-4-5]
                   [4-3-2 1-4-5-6]>>>>
<1 * <TnPr 1 <<Loop[4-3-2-1-0 1-2][3-2-1-0 1-2-3][4-2-1-0 1-3-4][4-3-1-0 1-4-5]
                   [4-3-2 1-4-5-6\2]>>>>
... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...
------------------------------------------------------------------------------

(length *)  ==>

537
```

Lisp file concerned in this chapter

`ls-twisted-products.lisp`.

Chapter 16

Eilenberg-Moore spectral sequence I

16.1 Introduction

This chapter is devoted to the effective homology version of the spectral sequence of Eilenberg-Moore, in the particular case of loop spaces. More precisely, let X be a 1-**reduced** simplicial set with effective homology, the software `Kenzo` constructs a 0-reduced simplicial set GX which is also with effective homology. In particular, if X is m-reduced, this process may be iterated m times, producing an effective homology version of G^kX, $k \leq m$. So, the software builds an object which contains a complete solution of the Adams problem about the iteration of the cobar construction. The user is advised to consult the paper *Stable homotopy and iterated loop spaces, pp. 545* by **Gunnar Carlsson** and **R.James Milgram** in *Handbook of Algebraic Topology, pp 545* edited by *I.M.James, North-Holland, 1995.*

16.2 The detailed construction

Let $\mathcal{C}_*(X)$ be a coalgebra with effective homology. This means that a homotopy equivalence:

$$\begin{array}{ccc} & \hat{C}_* & \\ \swarrow\nearrow & & \searrow\nwarrow \\ \mathcal{C}_*(X) & & EC_* \end{array}$$

is provided, where the chain complex EC_* is effective and must be considered as describing the homology of $\mathcal{C}_*(X)$. This scheme includes the case where $\mathcal{C}_*(X)$ is itself effective; without any other information, the program constructs automatically a trivial homotopy equivalence.
Now, if we apply the *Cobar* functor to this homotopy equivalence we obtain the homotopy equivalence H_R:

$$\begin{array}{ccc} & \widetilde{Cobar}^{\hat{C}_*}(\mathbb{Z},\mathbb{Z}) & \\ \swarrow\nearrow & & \searrow\nwarrow \\ Cobar^{\mathcal{C}_*(X)}(\mathbb{Z},\mathbb{Z}) & & \widetilde{Cobar}^{EC_*}(\mathbb{Z},\mathbb{Z}) \end{array}$$

in which the $\widetilde{Cobar}$'s are cobars construction with respect to the A_∞-coalgebras structure on $\hat{C}_*$ and EC_* defined by the initial homotopy equivalence. Secondly, Julio Rubio[1] has shown that it is possible to construct another

[1] **J.J. Rubio-Garcia**. *Homologie effective des espaces de lacets itérés: un logiciel*, Thèse, Institut Fourier, 1991.

homotopy equivalence, H_L:

$$\begin{array}{ccc} & Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX)) & \\ \swarrow\nearrow & & \searrow\nwarrow \\ \mathcal{C}_*(GX) & & Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathbb{Z}) \end{array}$$

Both reductions of H_L are obtained by the basic perturbation lemma, as explained in the following section. The composition of both homotopy equivalences, H_L and H_R, makes the link between GX, i.e. the loop space of X and the effective right bottom chain complex of H_R.

16.2.1 Obtaining the left reduction H_L

The homotopy equivalence H_L is obtained by a sequence of intermediate constructions based mainly upon two applications of the basic perturbation lemma. We are led to consider the two following comodules:

- $\mathcal{C}_*(X)$ comodule on itself, with the canonical coproduct.

$$\mathcal{C}_*(X) \xrightarrow{\Delta} \mathcal{C}_*(X) \otimes \mathcal{C}_*(X).$$

- $\check{\mathcal{C}}_*(X)$, comodule on $\mathcal{C}_*(X)$, with a "trivial" coproduct

$$\check{\mathcal{C}}_*(X) \xrightarrow{\check{\Delta}} \mathcal{C}_*(X) \otimes \check{\mathcal{C}}_*(X), \quad \sigma \to 1 \otimes \sigma,\ \sigma \in \check{\mathcal{C}}_*(X).$$

Now, we consider the set of the followings cobars (where u means *"untwisted"* and t *"twisted"*):

$$\begin{aligned} \mathrm{Hat}_{uu} &= Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \check{\mathcal{C}}_*(X) \otimes \mathcal{C}_*(GX)), \\ \mathrm{Hat}_{ut} &= Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \check{\mathcal{C}}_*(X) \otimes_t \mathcal{C}_*(GX)), \\ \mathrm{Hat}_{tu} &= Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes \mathcal{C}_*(GX)), \\ \mathrm{Hat}_{tt} &= Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX)). \end{aligned}$$

One may always say that Hat_{ut} is obtained from Hat_{uu} by a perturbation δ_r (r: right) induced by the twisted tensor product $\otimes_t$ and that Hat_{tu} is obtained from Hat_{uu} by a perturbation δ_l (l: left) induced by the discrepancy

between the coproducts in $\check{\mathcal{C}}_*(X)$ and $\mathcal{C}_*(X)$. After that, Hat_{tt} is obtained from Hat_{tu} by the perturbation δ_r as well as, by commutativity, from Hat_{ut} by the perturbation δ_l.
This is shown in the following diagram (here, the arrows are not reductions, but denote the perturbations between the differential morphisms):

$$\begin{array}{ccc} & \mathrm{Hat}_{uu} & \\ \delta_l \swarrow & & \searrow \delta_r \\ \mathrm{Hat}_{tu} & & \mathrm{Hat}_{ut} \\ \delta_r \searrow & & \swarrow \delta_l \\ & \mathrm{Hat}_{tt} & \end{array}$$

The underlying graded modules Hat_{uu}, Hat_{ut}, Hat_{tu} and Hat_{tt} are the same and the program keeps Hat_{uu} as the underlying graded module for all the chain complexes. The differential perturbations are given by the formulas:

$$\begin{aligned} \delta_r(\bar{c}_1 \otimes \cdots \otimes \bar{c}_n \otimes c \otimes g) &= [\bar{c}_1 \otimes \cdots \otimes \bar{c}_n] \otimes [d_{\otimes t}(c \otimes g) - d_{\otimes}(c \otimes g)], \\ \delta_l(\bar{c}_1 \otimes \cdots \otimes \bar{c}_n \otimes c \otimes g) &= \bar{c}_1 \otimes \cdots \otimes \bar{c}_n \otimes \bar{\Delta} c \otimes g, \end{aligned}$$

where $\bar{c}_i, c \in \mathcal{C}_*(X)$, $g \in \mathcal{C}_*(GX))$ and $\bar{\Delta} = \Delta - \check{\Delta}$.

Now, on one hand, we know that there exists a reduction

$$\mathrm{Hat}_{tu} = Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes \mathcal{C}_*(GX)) \Longrightarrow \mathcal{C}_*(GX),$$

so, perturbing this reduction by δ_r, one obtains the Rubio reduction

$$\mathrm{Hat}_{tt} = Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX)) \Longrightarrow \mathcal{C}_*(GX).$$

On the other hand, we know also that there exists a reduction

$$\mathrm{Hat}_{ut} = Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \check{\mathcal{C}}_*(X) \otimes_t \mathcal{C}_*(GX)) \Longrightarrow Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathbb{Z}),$$

so, perturbing this reduction by δ_l, one obtains the reduction

$$\mathrm{Hat}_{tt} = Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX)) \Longrightarrow Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathbb{Z}).$$

Finally, we have obtained the wished left homotopy equivalence H_L:

$$
\begin{array}{ccc}
 & Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX)) & \\
\swarrow\nearrow & & \searrow\nwarrow \\
\mathcal{C}_*(GX) & & Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathbb{Z})
\end{array}
$$

16.2.2 The useful functions

For the applications, the only function that the user must know is the function `loop-space-efhm` which builds the final homotopy equivalence. But for the interested user, we give nevertheless a short description of all the functions involved in the described process.

`loop-space-efhm` *space* *[Function]*
: From the space X (1-reduced) with effective homology (here the argument *space*), build a homotopy equivalence giving an effective homology version of the space GX. This homotopy equivalence will be used by the homology function to compute the homology groups. In fact, due to the slot-unbound mechanism of CLOS, this function will be automatically called, as soon as the user requires a homology group of a loop-space. If X is n-reduced and if the wished loop space is $\Omega^k X$, $k \leq n$, then the process will be automatically iterated.

`ls-hat-u-u` *space* *[Function]*
: Return the chain complex $Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \check{\mathcal{C}}_*(X) \otimes \mathcal{C}_*(GX))$. Because of the particular structure of $\check{\mathcal{C}}_*(X)$, this chain complex is nothing but $Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathbb{Z}) \otimes \check{\mathcal{C}}_*(X) \otimes \mathcal{C}_*(GX)$.

`ls-hat-left-perturbation` *space* *[Function]*
: Return the morphism corresponding to the differential perturbation δ_l, induced by the discrepancy between the coproducts of $\check{\mathcal{C}}_*(X)$ and $\mathcal{C}_*(X)$.

`ls-hat-t-u` *space* *[Function]*
: Return the chain complex $Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes \mathcal{C}_*(GX))$ by applying the differential perturbation `hat-left-perturbation` upon the chain complex built by the function `hat-u-u`. This is realized by the method `add`.

`ls-hat-u-t` *space* *[Function]*

Return the chain complex $Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \check{\mathcal{C}}_*(X) \otimes_t \mathcal{C}_*(GX))$. Because of the particular structure of $\check{\mathcal{C}}_*(X)$, this chain complex is nothing but $Cobar^{\mathcal{C}_*(X)}(\mathbb{Z},\mathbb{Z}) \otimes [\check{\mathcal{C}}_*(X) \otimes_t \mathcal{C}_*(GX)]$, where the twisted tensor product is obtained by a call to `twisted-tnsr-prdc` (see loop space fibrations chapter).

`ls-hat-right-perturbation` *space* *[Function]*

Return the morphism corresponding to the differential perturbation δ_r induced by the twisted tensor product. This morphism is nothing but the tensor product of two morphisms: the identity morphism on $Cobar^{\mathcal{C}_*(X)}(\mathbb{Z},\mathbb{Z})$ and the perturbation morphism induced by the twisted tensor product. This last morphism is a by-product of the function `szczarba`.

`ls-left-hmeq-hat` *space* *[Function]*

Return the chain complex $\mathrm{Hat}_{tt} = Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX))$ by perturbing the chain complex Hat_{ut} by the perturbation δ_l.

`ls-pre-left-hmeq-left-reduction` *space* *[Function]*

Build the reduction

$$\mathrm{Hat}_{tu} = Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes \mathcal{C}_*(GX)) \Longrightarrow \mathcal{C}_*(GX).$$

`ls-pre-left-hmeq-right-reduction` *space* *[Function]*

Build the reduction

$$\mathrm{Hat}_{ut} = Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \check{\mathcal{C}}_*(X) \otimes_t \mathcal{C}_*(GX)) \Longrightarrow Cobar^{\mathcal{C}_*(X)}(\mathbb{Z},\mathbb{Z}).$$

`ls-left-hmeq-left-reduction` *space* *[Function]*

Build the Rubio reduction

$$Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX)) \Longrightarrow \mathcal{C}_*(GX)$$

by perturbing by the perturbation δ_r the reduction

$$\mathrm{Hat}_{tu} \Longrightarrow \mathcal{C}_*(GX)$$

obtained by the function `pre-left-hmeq-left-reduction`.

`ls-left-hmeq-right-reduction` *space* *[Function]*

Build the reduction

$$Cobar^{\mathcal{C}_*(X)}(\mathbb{Z}, \mathcal{C}_*(X) \otimes_t \mathcal{C}_*(GX)) \Longrightarrow Cobar^{\mathcal{C}_*(X)}(\mathbb{Z},\mathbb{Z})$$

by perturbing by the perturbation δ_l the reduction

$$\mathrm{Hat}_{ut} \Longrightarrow Cobar^{\mathcal{C}_*(X)}(\mathbb{Z},\mathbb{Z})$$

obtained by the function `pre-left-hmeq-right-reduction`.

`ls-left-hmeq` *space* *[Function]*

Build the homotopy equivalence

$$\begin{array}{ccc} & Cobar^{\mathcal{C}_*(X)}(\mathbb{Z},\mathcal{C}_*(X)\otimes_t \mathcal{C}_*(GX)) & \\ \swarrow\nearrow & & \searrow\nwarrow \\ \mathcal{C}_*(GX) & & Cobar^{\mathcal{C}_*(X)}(\mathbb{Z},\mathbb{Z}) \end{array}$$

from the above reductions. The function `loop-space-efhm` composes this left homotopy equivalence H_L with a homotopy equivalence H_R which is the cobar of a pre-existing homotopy equivalence (possibly the trivial one) between the coalgebra X and an effective version of it, as described at the beginning of this chapter. The Lisp definition is given in the next subsection.

16.2.3 Searching homology for loop spaces

The origin list of a loop space object has the form (`LOOP-SPACE` *space*). The `search-efhm` method applied to a loop space object consists essentially in a call to the function `loop-space-efhm` described in this chapter.

```
(defmethod SEARCH-EFHM (loop-space (orgn (eql 'loop-space)))
  (declare (type simplicial-set loop-space))
  (loop-space-efhm (second (orgn loop-space))))
```

The lisp definition of the function `loop-space-efhm` shows clearly the possible recursivity of the process is *space* is itself a loop space.

```
(defun LOOP-SPACE-EFHM (space)
  (declare (type simplicial-set space))
  (let ((left-hmeq (ls-left-hmeq space))
        (right-hmeq (cobar (efhm space))))
    (declare (type homotopy-equivalence left-hmeq right-hmeq))
    (cmps left-hmeq right-hmeq)))
```

Examples

Let us show first how to get the homology groups of some known loop spaces, $\Omega\mathcal{S}^2$, $\Omega^3\mathcal{S}^4$, $\Omega^2\mathrm{Moore}(\mathbb{Z}/2\mathbb{Z}, 3)$.

```
(setf s2 (sphere 2))  ==>

[K1 Simplicial-Set]

(setf os2 (loop-space s2))  ==>

[K6 Simplicial-Group]

(dotimes (k 10) (homology os2 k))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z

Homology in dimension 2 :

Component Z

Homology in dimension 3 :
```

```
Component Z

Homology in dimension 4 :

Component Z

Homology in dimension 5 :

Component Z

Homology in dimension 6 :

Component Z

Homology in dimension 7 :

Component Z

Homology in dimension 8 :

Component Z

Homology in dimension 9 :

Component Z

---done---

(cat-init)

(setf s4 (sphere 4))  ==>

[K1 Simplicial-Set]

(setf ooos4 (loop-space s4 3))  ==>

[K30 Simplicial-Group]

(dotimes (k 6) (homology ooos4 k))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z
```

```
Homology in dimension 2 :

Component Z/2Z

Homology in dimension 3 :

Component Z/2Z

Homology in dimension 4 :

Component Z/3Z

Component Z/2Z

Component Z

Homology in dimension 5 :

Component Z/3Z

Component Z/2Z

Component Z

---done---

(setf moore-23 (moore 2 3))  ==>

[K42 Simplicial-Set]

(setf oo-moore-23 (loop-space moore-23 2))  ==>

[K59 Simplicial-Group]

(dotimes (k 6) (homology oo-moore-23 k))  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :
```

```
Component Z/2Z

Homology in dimension 3 :

Component Z/4Z

Component Z/2Z

Homology in dimension 4 :

Component Z/2Z

Component Z/2Z

Component Z/2Z

Homology in dimension 5 :

Component Z/2Z

Component Z/2Z

Component Z/2Z

Component Z/2Z

---done---
```

16.3 The solution of the Adams and Carlsson problem

Let X an n-reduced simplicial set which is not a suspension. The Adams and Carlsson problem[2] asks for a finite CW-complex modelizing the iterated loop space $\Omega^n X$. We show in this section how the `Kenzo` program builds the relevant solution.
Let us consider for instance the truncated projective space $X = P^\infty\mathbb{R}/P^3\mathbb{R}$ (a non-suspended space) and let us build $\Omega^3 X$.

```
(setf p4 (R-proj-space 4))  ==>

[K1 Simplicial-Set]

(setf o3p4 (loop-space p4 3))  ==>

[K30 Simplicial-Group]

(setf ecc (rbcc (efhm o3p4)))  ==>

[K390 Chain-Complex]
```

The cellular homology of $\Omega^3 X$ is now known through the symbol `ecc`. Note that 360 objects $(390 - 30)$ have been built to get the final complex `ecc`. From the printing of the length of the basis in dimensions 0 to 5,

```
(dotimes (i 6) (print (length (basis ecc i))))  ==>

1
1
2
5
13
33
```

we see that the required CW-complex can be constructed using 1 0-cell, 1 1-cell, 2 2-cells, 5 3-cells, 13 4-cells and 33 5-cells. Let us print the incidence matrix between the 5 and the 4-cells:

[2] **Gunnar Carlsson, R.James Milgram.** *Stable homotopy and iterated loop spaces* in *Handbook of Algebraic Topology, pp 545* edited by *I.M.James, North-Holland, 1995.*

```
(chcm-mat ecc 5)  ==>

========== MATRIX 13 lines + 33 columns =====

L1=[C1=-2]
L2=[C1=-1]
L3=[C1=-4][C2=1][C3=-1][C4=-2]
L4=[C2=1][C3=-1][C6=2]
L5=[C1=6][C4=1][C6=1]
L6=[C1=4][C4=4][C6=4][C7=3]
L7=[C1=4][C12=-2][C14=2]
L8=[C1=6][C4=1][C6=1]
L9=[C1=4][C4=4][C6=4][C7=3]
L10=[C8=4][C10=1][C11=-1][C14=-4][C15=-2][C20=-2]
L11=[C1=4][C8=4][C10=1][C11=-1][C16=-4][C18=-1][C19=1][C23=-2]
L12=[C12=4][C13=2][C16=-4][C18=-1][C19=1][C27=-2]
L13=[C1=-1][C20=4][C21=2][C23=-4][C24=-2][C27=4][C28=2]

========== END-MATRIX
---done---
```

For instance, we see that the first 5-cell,

```
(first (basis ecc 5))  ==>

<<AlLp[5 <<AlLp[6 <<AlLp[7 8]>>]>>]>>
```

is glued to the 4-cells #5 and #8 by attaching maps of degree 6. The 5-cell #4 is also glued to the 4-cell #5 by an attaching map of degree 1, etc.

Lisp files concerned in this chapter

`lp-space-efhm.lisp`, `searching-homology.lisp`.

Chapter 17

Classifying Spaces

17.1 Introduction

Let $\mathcal{G}$ be a simplicial group 0–reduced. The software `Kenzo` allows the construction of the universal bundle $\bar{W}\mathcal{G}$, called the classifying space of $\mathcal{G}$. It is known that this is the base space of a principal bundle such that the total space is contractible. We know also that if $\mathcal{G}$ is an Abelian simplicial group then $\bar{W}\mathcal{G}$ is also an Abelian simplicial group, otherwise if $\mathcal{G}$ is non–Abelian, then $\bar{W}\mathcal{G}$ is only a simplicial set. In dimension n the simplices of the classifying space are elements of the product:

$$\mathcal{G}_{n-1} \times \mathcal{G}_{n-2} \times \cdots \times \mathcal{G}_0,$$

where $\mathcal{G}_k$ is the set of the k–simplices of the given simplicial group.
A simplex of $\bar{W}\mathcal{G}$ is called **gbar** (*geometric* bar) and may be seen as an n–uple

$$(g_{n-1}, g_{n-2}, \ldots, g_1, g_0),$$

where the g_i's are elements of $\mathcal{G}_i$, possibly *degenerated*. In `Kenzo` a gbar is represented externally under the form:

`<<GBar <` $[\eta g]_{n-1}$`><`$[\eta g]_{n-2}$ `> ... < `$[\eta g]_1$ `> < - `g_0`> >>`

where a term such as $[\eta g]_i$ is a simplex of $\mathcal{G}_i$. Let us recall that our group $\mathcal{G}$ is assumed 0–reduced; as a consequence, the g_0 compound of a gbar is necessarily the identity element of $\mathcal{G}$.
In the symbolic expression $[\eta g]_i$, the operator η represents a sequence (possibly the void one) of degeneracy operators applied coherently upon some element of $\mathcal{G}_k$, for $k \leq i$. For instance $[\eta g]_1$ can be $\eta_0 g_0$, i.e. the 0–degeneracy

of the base point and $[\eta g]_2$ can be can be $\eta_1\eta_0 g_0$. To be more explicit, let us take the Abelian simplicial group $K(\mathbb{Z}_2, 1)$ (see the simplicial groups chapter), then some elements of the basis in dimension 4 of its classifying space are, for instance:

```
<<GBar<- 3><- 2><- 1><- 0>>>
<<GBar<- 3><1-0 0><0 0><- 0>>>
<<GBar<1 2><- 2><0 0><- 0>>>
<<GBar<1 2><0 1><0 0><- 0>>>
<<GBar<2 2><1 1><0 0><- 0>>>
<<GBar<2-0 1><1 1><0 0><- 0>>>
<<GBar<2-1 1><1 1><- 1><- 0>>>
<<GBar<2-1 1><0 1><0 0><- 0>>>
```

We recall that the symbol -, in front of an element means "no degeneracy".

17.2 Face, degeneracy and group operations

Let us denote a gbar under the following generic form:

$$g_{bar} = (g_{n-1}, g_{n-2}, \ldots, \mathbf{g_j}, \ldots, g_1, g_0).$$

In this symbolism, g_0 is also the base point of the initial simplicial group. The rule for the face operators is the following:

- If $j = n$, $\delta_n g_{bar} = (g_{n-2}, g_{n-3}, \ldots, g_1, g_0)$.
- If $j \neq n$, $\delta_j g_{bar} = (\delta_j g_{n-1}, \ldots, \delta_{\mathbf{j}}\mathbf{g_{j+1}}, \mathbf{g_{j-1}}.\delta_{\mathbf{j}}\mathbf{g_j}, \mathbf{g_{j-1}}, \ldots, g_1, g_0)$, where in the term $\mathbf{g_{j-1}}.\delta_{\mathbf{j}}\mathbf{g_j}$, the dot represents the group composition.
- If $j = 0$, $\delta_0 g_{bar} = (\delta_0 g_{n-1}, \ldots, \delta_0 g_1)$.

The rule for the degeneracy operators is:

$$\eta_j g_{bar} = (\eta_j g_{n-1}, \ldots, \eta_{\mathbf{j}}\mathbf{g_{j+1}}, \eta_{\mathbf{j-1}}\eta_{\mathbf{j-2}}\ldots\eta_{\mathbf{1}}\eta_{\mathbf{0}}\mathbf{g_0}, \mathbf{g_{j-1}}, \ldots, g_1, g_0).$$

If $\mathcal{G}$ is Abelian then $\bar{\mathcal{W}}\mathcal{G}$ is also Abelian and its composition law is simply:

$$(g_{n-1}, \ldots, g_1, g_0).(g'_{n-1}, g'_1, g'_0) = (g_{n-1}.g'_{n-1}, \ldots, g_1.g'_1, g_0.g'_0).$$

It is well known that if the initial group is not Abelian then, in general, the previous composition law is not compatible with the simplicial structure.

17.3 The functions for the classifying spaces

`gbar` *dmns dop-1 elem-1 ... dop-n elem-n* *[Macro]*

Build a gbar in dimension *dmns*. The sequence *dop-1 elem-1 ... dop-n elem-n*, is a sequence of pairs (*dop-k, elem-k*), where *dop-k* is a **binary coded** degeneracy operator and *elem-k* a simplex of the initial simplicial group. The ordering of the pairs in the sequence must conform to the theoritical definition of a gbar. If the sequence of pairs is void, the special call `(gbar 0)`, creates the base point of the classifying space externally represented by `<<GBAR>>`. This base point can be reached through the system symbol `+null-gbar+`.

`classifying-space-cmpr` *cmpr* *[Function]*

From a comparison function *cmpr* defined for a simplicial group $\mathcal{G}$, build a comparison function for the classifying space of $\mathcal{G}$.

`classifying-space-basis` *basis* *[Function]*

From the basis function *basis* of a simplicial group $\mathcal{G}$, build a basis function for the classifying space of $\mathcal{G}$. If $\mathcal{G}$ is locally effective, this function returns the symbol `:locally-effective`.

`classifying-space-face` *face sintr-grml* *[Function]*

From the face function *face* and the lisp function for the group law, *sintr-grml*, of a simplicial group $\mathcal{G}$, build a face function for the classifying space of $\mathcal{G}$.

`classifying-space` *smgr* *[Method]*

Build the simplicial set, classifying space of the simplicial group *smgr*. The construction uses the three preceding auxiliary functions. The base point of the returned simplicial set is the null gbar `+null-gbar+`.

`classifying-space` *A-smgr* *[Method]*

Build the Abelian simplicial group, classifying space of the Abelian simplicial group *A-smgr*. The group law and its inverse (slots `:sintr-grml` and `:sintr-grin`) are built by the two auxiliary functions:

`classifying-space-grml-sintr` and
`classifying-space-grin-sintr`,

defining the laws of this new group from the corresponding laws of *A-smgr*.

17.4 Eilenberg-Mac Lane spaces $K(\pi, n)$

We have already described $K(\mathbb{Z}, 1)$ and $K(\mathbb{Z}_2, 1)$ in the chapter on simplicial groups. This section gives some complements about the construction of $K(\mathbb{Z}, n)$ and $K(\mathbb{Z}_2, n)$. The trivial case for $n = 0$ is not implemented in `Kenzo`. For $n > 1$, these simplicial groups are built recursively as the classifying spaces of $K(\mathbb{Z}, n-1)$ and $K(\mathbb{Z}_2, n-1)$, respectively, as shown in the following description:

`k-z` n *[Function]*
Build $K(\mathbb{Z}, n)$ recursively from $K(\mathbb{Z}, n-1)$. If $n = 1$, return $K(\mathbb{Z}, 1)$

```
(defun K-Z (n)
   (the ab-simplicial-group
      (if (= n 1)
          (k-z-1)
          (classifying-space (k-z (1- n))))))
```

`k-z2` n *[Function]*
Build $K(\mathbb{Z}_2, n)$ recursively from $K(\mathbb{Z}_2, n-1)$. If $n = 1$, return $K(\mathbb{Z}_2, 1)$

```
(defun K-Z2 (n)
   (the ab-simplicial-group
      (if (= n 1)
          (k-z2-1)
          (classifying-space (k-z2 (1- n))))))
```

These spaces have an effective homology which is computed via a *search-efhm* method (described in a next chapter). For the case $n = 1$, we may recall here that $K(\mathbb{Z}, 1)$, though being locally effective, has the homology of S^1. The *search-efhm* method applied to this chain complex does the following: a reduction between $K(\mathbb{Z}, 1)$ and the chain complex `circle` (described in the chain complex chapter) is built using the function `kz1-rdct` and the homotopy equivalence, value of the slot `efhm`, is built between this reduction and the trivial reduction of $K(\mathbb{Z}, 1)$ upon itself.
On the other hand, $K(\mathbb{Z}, 2)$ has a very simple finite basis in every dimension (see $\mathbb{R}$-projective spaces), so its homology is computed directly from its basis. The homotopy equivalence, value of the slot `efhm`, is the trivial homotopy equivalence of $K(\mathbb{Z}, 2)$ upon itself.

Examples

Let us begin by showing some examples of gbars built from various initial simplicial groups. In the second statement, the initial simplicial group is $K(\mathbb{Z}_2, 1)$, in the third, it is $\Omega(Moore(2,2))$. We recall that in the call of the function `gbar`, the degeneracy operators appear as integers, according to the general rule in `Kenzo`, but the printed result shows in clear the sequence of η_i's.

```
(gbar 2 1 'a 2 'b)  ==>

<<GBar<0 A><1 B>>>

(gbar 4 0 3 3 0 1 0 0 0)  ==>

<<GBar<- 3><1-0 0><0 0><- 0>>>

(setf gbar-mr22 (gbar 4 0 (loop3 3 'm2 1 4 'n3 1)
                        0 (loop3 0 'n3 1)
                        0 (loop3 0 'm2 1)
                        0 +null-loop+))   ==>

<<GBar<- <<Loop[1-0 M2][2 N3]>>><- <<Loop[N3]>>><- <<Loop[M2]>>>
      <- <<Loop>>>>>
```

Let us test the face function upon this simplex. The user will note that the face function returns an abstract simplex.

```
(setf om (loop-space (moore 2 2)))  ==>

[K18 Simplicial-Group]

(setf face (classifying-space-face (face om) (sintr (grml om))))  ==>

#<Closure (FLET CLASSIFYING-SPACE-FACE RSLT) @ #x49af32>

(dotimes (i 5) (print (funcall face i 4 gbar-m22)))  ==>

<AbSm - <<GBar<- <<Loop[0 M2][1 M2]>>><- <<Loop[M2]>>><- <<Loop>>>>>>
<AbSm 0 <<GBar<- <<Loop[M2]>>><- <<Loop>>>>>>
<AbSm - <<GBar<- <<Loop[0 M2][N3]>>><- <<Loop[M2\2]>>><- <<Loop>>>>>>
<AbSm - <<GBar<- <<Loop[N3\2]>>><- <<Loop[M2]>>><- <<Loop>>>>>>
<AbSm - <<GBar<- <<Loop[N3]>>><- <<Loop[M2]>>><- <<Loop>>>>>>
```

Let us work now, directly with the classifying space of $K(\mathbb{Z}_2, 1)$.

```
(setf cs-kz21 (classifying-space (k-z2-1)))  ==>

[K13 Abelian-Simplicial-Group]
```

```
(orgn cs-kz21)  ==>

(CLASSIFYING-SPACE [K1 Abelian-Simplicial-Group])

(setf elem-1 (first (basis cs-kz21 4)))  ==>

<<GBar<- 3><- 2><- 1><- 0>>>

(? cs-kz21 4 elem-1)  ==>

-------------------------------------------------------------------------{CMBN 3}
<2 * <<GBar<- 2><- 1><- 0>>>>
-------------------------------------------------------------------------------

(? cs-kz21 *)  ==>

-------------------------------------------------------------------------{CMBN 2}
-------------------------------------------------------------------------------

(cprd cs-kz21  4 elem-1)  ==>

-------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr <<GBar>> <<GBar<- 3><- 2><- 1><- 0>>>>>
<1 * <TnPr <<GBar<- 1><- 0>>> <<GBar<- 1><- 0>>>>>
<1 * <TnPr <<GBar<- 3><- 2><- 1><- 0>>> <<GBar>>>>
-------------------------------------------------------------------------------

(dotimes (i 5)
    (print (face cs-kz21 i 4 elem-1)))  ==>

<AbSm - <<GBar<- 2><- 1><- 0>>>>
<AbSm 0 <<GBar<- 1><- 0>>>>
<AbSm 1 <<GBar<- 1><- 0>>>>
<AbSm 2 <<GBar<- 1><- 0>>>>
<AbSm - <<GBar<- 2><- 1><- 0>>>>
```

Let us test the law group with $K(\mathbb{Z}, 1)$. We recall the simplices of this Abelian simplicial group are represented as lists of integers. The functions `grml` and `grin` accept also the gbars under the form of abstract simplices (see the last statement).

```
(setf cs-kz1 (classifying-space (k-z-1)))  ==>

[K13 Abelian-Simplicial-Group]

(grml cs-kz1 3 (crpr 0 (gbar 3 0 '(1 2) 0 '(3) 0 '())
                     0 (gbar 3 0 '(-1 -2) 0 '(-3) 0 '())))  ==>
```

```
<AbSm 2-1-0 <<GBar>>>

(grml cs-kz1 3 (crpr 0 (gbar 3 0 '(1 2) 0 '(3) 0 '())
                     4 (gbar 2 0 '(-3) 0 '())))            ==>

<AbSm - <<GBar<- (1 2)><0 NIL><- NIL>>>>

(grml cs-kz1 3 (crpr 0 (gbar 3 0 '(1 2) 0 '(3) 0 '())
                     1 (gbar 2 0 '(-3) 0 '())))            ==>

<AbSm - <<GBar<- (1 -1)><- (3)><- NIL>>>>

(grin cs-kz1 3 (gbar 3 0 '(1 2) 1 '() 0 '()))  ==>

<AbSm - <<GBar<- (-1 -2)><0 NIL><- NIL>>>>

(grin cs-kz1 3 *)  ==>

<AbSm - <<GBar<- (1 2)><0 NIL><- NIL>>>>
```

As $K(\mathbb{Z},1)$ and $K(\mathbb{Z}_2,1)$ are Abelian simplicial groups, we may iterate the classifying space construction and retrieve some known results.

```
(cat-init)    ;; re-initialization

(setf k-z-3 (k-z 3))  ==>

[K25 Abelian-Simplicial-Group]

(homology k-z-3 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

Homology in dimension 5 :

Component Z/2Z
```

```
Homology in dimension 6 :

Homology in dimension 7 :

Component Z/3Z

Homology in dimension 8 :

Component Z/2Z

Homology in dimension 9 :

Component Z/2Z

(setf k-z2-5 (k-z2 5))  ==>

[K342 Abelian-Simplicial-Group]

(homology k-z2-5 0 7)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Homology in dimension 4 :

Homology in dimension 5 :

Component Z/2Z

Homology in dimension 6 :

--done--
```

Lisp files concerned in this chapter

`classifying-space.lisp`, `k-pi-n.lisp`.

Chapter 18

Serre spectral sequence

18.1 Introduction

This chapter is devoted to the `Kenzo` implementation of the Serre spectral sequence with the aim to compute the homology groups of the total space of a fibration. From the programming point of view, this method will be used according to our general scheme of searching homology (generic function `search-efhm`) applied to an object of type `fibration-total`.

18.2 The topological problem

Let $\mathcal{G}$ be a simplicial group (the fiber space), B, a 1–reduced simplicial set (the base space) and τ (the fibration), a simplicial morphism of degree -1, $\tau : B \longrightarrow \mathcal{G}$. We know, from a previous chapter (fibrations) that the software `Kenzo` knows how to build the total space of the fibration: $B \times_\tau \mathcal{G}$, which has the same simplices as $B \times \mathcal{G}$ and the same face operators except the last one, given by:

$$\partial_n(b, g) = (\partial_n b, \tau(b).\partial_n g), \quad b \in B, g \in \mathcal{G},$$

where the '.' denotes the group operation in $\mathcal{G}$. This discrepancy between the respective face operators ∂_n induces a discrepancy between the differential operators. We suppose now, that the base and fiber spaces are objects

of effective homology type, i.e. there exists homotopy equivalences

$$\begin{array}{ccccccc} & & \widehat{B} & & & & \widehat{\mathcal{G}} \\ & \rho_1 \swarrow & & \searrow \rho_2 & & \rho_1' \swarrow & & \searrow \rho_2' \\ B & & & EB & & \mathcal{G} & & EG \end{array}$$

where the chain complexes EB and EG are effective. The problem consists in finding a homotopy equivalence between $B \times_\tau \mathcal{G}$ and an effective chain complex noted $EB \widetilde{\otimes} EG$; the underlying graded module of $EB \widetilde{\otimes} EG$ is $EB \otimes EG$ but the differential is twisted in a rather complicated way. In fact, the chain complex $EB \widetilde{\otimes} EG$ is nothing but the Hirsh complex[1].

18.2.1 The Ronald Brown reduction

We start from the Eilenberg-Zilber reduction (built in `Kenzo` by the function `ez`).

$$\begin{array}{ccc} \mathcal{C}_*(B \times \mathcal{G}) & \xrightarrow{h} & {}^s\mathcal{C}_*(B \times \mathcal{G}) \\ f \downarrow\uparrow g & & \\ \mathcal{C}_*(B) \otimes \mathcal{C}_*(\mathcal{G}) & & \end{array} .$$

But, in fact we are interested in the following one (the Ronald Brown reduction):

$$\begin{array}{ccc} \mathcal{C}_*(B \times_\tau \mathcal{G}) & \xrightarrow{h} & {}^s\mathcal{C}_*(B \times_\tau \mathcal{G}) \\ f \downarrow\uparrow g & & \\ \mathcal{C}_*(B) \otimes_t \mathcal{C}_*(\mathcal{G}) & & \end{array}$$

[1] **Guy Hirsh.** *Sur les groupes d'homologie des espaces fibrés*, Bulletin de la Société Mathématique de Belgique, 1954, vol. 6, pp. 79-96.

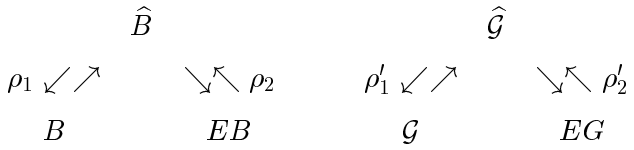
where the symbol $\otimes_t$ represents a twisted tensor product, induced by the twisting operator τ; this twisting operator is nothing but the Shih twisting cochain[2] determined by Szczarba[3]. Here, to obtain the Szczarba cochain, we apply the perturbation lemma (Ronnie Brown). In effect, writing

$$d_\tau = d + (d_\tau - d),$$

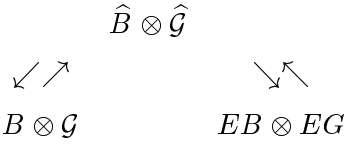
one may consider that the differential d_τ of $B \times_\tau \mathcal{G}$ is the differential d of $B \times \mathcal{G}$ modified by the perturbation $d_\tau - d$. The perturbation morphism $d_\tau - d$ is deduced from the difference between both ∂_n face operators. This being done, a perturbation may possibly be propagated along the Eilenberg-Zilber reduction (see the method `add` applied to a reduction) and the perturbation lemma gives also, as a bonus, the perturbation to be applied to the differential of $B \otimes \mathcal{G}$ to obtain the differential of $B \otimes_t \mathcal{G}$.

18.2.2 The Gugenheim algorithm

On the other hand, from the homotopy equivalences



it is possible to build their tensor product, i.e. the homotopy equivalence:



[2] **Gugenheim.**, *On the chain complex of a fibration.* Illinois Journal of Mathematics, 1972, vol. 16, pp. 398-414.

[3] **R. H. Szczarba.** *The homology of twisted cartesian products.* Transactions of the American Mathematical Society, 1961, vol. 100, pp. 197-216.

This is done in `Kenzo` by the method `tnsr-prdc`. In fact, we are interested in the following one:

$$\begin{array}{ccc} & \widehat{B}\widetilde{\otimes}\widehat{\mathcal{G}} & \\ \swarrow\nearrow & & \searrow\nwarrow \\ B \otimes_t \mathcal{G} & & EB\widetilde{\otimes}EG \end{array}$$

where $\widehat{B}\widetilde{\otimes}\widehat{\mathcal{G}}$ and $EB\widetilde{\otimes}EG$ are appropriate strongly twisted tensor products determined by the perturbation lemma: this last homotopy equivalence is obtained by propagating over the previous one, the perturbation to be applied to the differential of $B \otimes \mathcal{G}$ to obtain the differential of $B \otimes_t \mathcal{G}$. The homotopy equivalence so obtained is called in `Kenzo` the *right Serre homotopy equivalence*.

18.2.3 Assembling the puzzle

Let us build now the so-called *left Serre homotopy equivalence*,

$$\begin{array}{ccc} & B \times_\tau \mathcal{G} & \\ \swarrow\nearrow & & \searrow\nwarrow \\ B \times_\tau \mathcal{G} & & B \otimes_t \mathcal{G} \end{array}$$

where the left reduction is the trivial one and the right reduction is the Brown reduction. We may then compose both Serre homotopy equivalences to obtain what we were looking for:

$$\begin{array}{ccccccc} & B \times_\tau \mathcal{G} & & & & \widehat{B} \otimes_t \widehat{\mathcal{G}} & \\ \rho_1 \swarrow\nearrow & & \searrow\nwarrow \rho_2 & & \rho_1' \swarrow\nearrow & & \searrow\nwarrow \rho_2' \\ B \times_\tau \mathcal{G} & & B \otimes_t \mathcal{G} & & B \otimes_t \mathcal{G} & & EB \otimes_t EG \end{array}$$

18.3 The functions for the Serre spectral sequence

The following functions are given for information purpose.

`fibration-dtau-d-intr` *fibration* *[Function]*
: Build the lisp function corresponding to the difference $d_\tau - d$, where d is the differential in $B \times \mathcal{G}$ and d_τ the differential in $B \times_\tau \mathcal{G}$. This is an internal function and is used in the following one.

`fibration-dtau-d` *fibration* *[Function]*
: Build the morphism $d_\tau - d$, which is the perturbation morphism to be added to the differential d of $B \times \mathcal{G}$ to obtain d_τ, the differential in $B \times_\tau \mathcal{G}$. We recall that both simplicial sets have the same simplices. The source and the target of this morphism is the total space $B \times_\tau \mathcal{G}$ and the degree is -1.

`Brown-reduction` *fibration* *[Function]*
: Return two values: a) the Brown reduction and b) the perturbation to be applied to the differential of the tensor product $B \otimes \mathcal{G}$ to obtain the differential of the twisted tensor product $B \otimes_t \mathcal{G}$.

`left-Serre-efhm` *fibration* *[Function]*
: Build the left Serre homotopy equivalence from the argument *fibration*.

`right-Serre-efhm` *fibration* *[Function]*
: Build the right Serre homotopy equivalence from the argument *fibration*.

18.4 Searching homology for fibration total spaces

The comment list of a `Kenzo` object which is the total space of a fibration has the form (`fibration-total` *fibration*). The `search-efhm` method applied to a simplicial set having this kind of comment list, builds the composition of the left and right Serre homotopy equivalences. The right one needs the homotopy equivalences attached respectively to the base and fiber spaces. This may involve a recursive call of `search-efhm`. If the base space or the fiber space are locally effective, the total space is locally effective and the method cannot compute the homology.

```
(defmethod SEARCH-EFHM (smst (orgn (eql 'fibration-total)))
  (declare
     (type simplicial-set smst))
  (the homotopy-equivalence
     (fibration-total-efhm (second (orgn smst)))))
```

```
(defun FIBRATION-TOTAL-EFHM (fibration)
  (declare (type fibration fibration))
  (the homotopy-equivalence
       (cmps (left-serre-efhm fibration)
             (right-serre-efhm fibration))))
```

Examples

We begin by some examples similar to examples that we have seen in the fibrations chapter. For the user, the delicate point is to write in Lisp a correct twisting operator. There is no check in this version of `Kenzo` for the coherency of the twisting operator. In the first example we define a fibration $\tau : S^2 \to K(\mathbb{Z}, 1)$ with $\tau(s2) = (2)$, then we build the total space of the fibration, namely $P^3\mathbb{R}$, and we compute some known homology groups.

```
(setf sph2 (sphere 2))  ==>

[K1 Simplicial-Set]

(setf kz1 (k-z-1))  ==>

[K6 Abelian-Simplicial-Group]

(setf tw2 (build-smmr
             :sorc sph2
             :trgt kz1
             :degr -1
             :sintr #'(lambda (dmns gmsm)
                        (unless (= dmns 2)
                                (error "Dimension error for
                                        the twisting operator S2-->KZ1"))
                        (absm 0 (list 2)))
     :orgn '(s2-tw-kz1)))            ==>

[K18 Fibration]

(? tw2 2 's2)  ==>

<AbSm - (1)>

(? tw2 0 '*)  ==>

Error: Dimension error for the twisting operator S2-->KZ1

(setf p3r (fibration-total tw2))  ==>

[K24 Simplicial-Set]
```

```
(homology p3r 0 10)   ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/2Z

Homology in dimension 2 :

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

Homology in dimension 5 :

Homology in dimension 5 :

Homology in dimension 7 :

Homology in dimension 8 :

Homology in dimension 9 :

--done--
```

Let us do the same kind of computations with $\tau(s2) = (5)$ and $\tau(s2) = (17)$.

```
(setf tw5 (build-smmr
             :sorc sph2
             :trgt kz1
             :degr -1
             :sintr #'(lambda (dmns gmsm)
                        (absm 0 (list 5)))
             :orgn '(s2-tw-kz1-5)))         ==>

[K110 Fibration]

(setf total-5 (fibration-total tw5))  ==>

[K111 Simplicial-Set]

(homology total-5 0 5)
```

```
Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/5Z

Homology in dimension 2 :

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

(setf tw17 (build-smmr
             :sorc sph2
             :trgt kz1
             :degr -1
             :sintr #'(lambda (dmms gmsm)
                        (absm 0 (list 17)))
             :orgn '(s2-tw-kz1-17)))         ==>

[K165 Fibration]

(setf total-17 (fibration-total tw17))  ==>

[K166 Simplicial-Set]

(homology total-17 0 4)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z/17Z

Homology in dimension 2 :

Homology in dimension 3 :

Component Z
```

Using the last total space (`total-17`), it is instructive to enter in the details of the resulting homotopy equivalence that the system builds to compute the homology groups. First, the function `echcm` selects the **effective** chain complex of the homotopy equivalence, value of the slot `efhm` of the simplicial set `total-17`, and we may print some basis of this chain complex. In fact, all basis in dimension above 3 are null. The presence of the symbol `S1` is due to the fact that the effective chain complex used for the homology of $K(\mathbb{Z},1)$ is the circle (see the classifying spaces chapter).

```
(setf echcm (echcm total-17))  ==>

[K205 Chain-Complex]

(dotimes (i 6) (print (basis echcm i)))  ==>

(<TnPr * *>)
(<TnPr * S1>)
(<TnPr S2 *>)
(<TnPr S2 S1>)
NIL
NIL
```

The resulting homotopy equivalence built by the system, may be summarized by the diagram:

$$\begin{array}{ccc} & \widehat{S^2 \times_\tau K(\mathbb{Z},1)} & \\ f \swarrow\nearrow & & \searrow\nwarrow g \\ S^2 \times_\tau K(\mathbb{Z},1) & & [K205\, Chain\, Complex] \end{array}$$

where f and g are respectively the *descending* and *ascending* morphisms of the left and right reductions of the homotopy equivalence. We recall that these morphisms are obtained respectively by the functions `lf` and `rg`. We may compose these morphisms to create a morphism `fg` from $[K205]$ to $S^2 \times_\tau K(\mathbb{Z},1)$ and see how it acts on the basis elements in dimension 1 and 2:

```
(setf fg (cmps (lf (efhm total-17)) (rg (efhm total-17))))  ==>

[K220 Morphism (degree 0)]

(setf gen1 (first (basis echcm 1)))  ==>
```

```
<TnPr * S1>

(? fg 1 gen1)  ==>

-------------------------------------------------------------------------{CMBN 1}
<-1 * <CrPr 0 * - (1)>>
-------------------------------------------------------------------------------

(setf gen2 (first (basis echcm 2)))  ==>

<TnPr S2 *>

(setf fg-gen2 (? fg 2 gen2))  ==>

-------------------------------------------------------------------------{CMBN 2}
<-1 * <CrPr - S2 1-0 NIL>>
<-1 * <CrPr 1-0 * - (1 1)>>
<-1 * <CrPr 1-0 * - (1 2)>>
<-1 * <CrPr 1-0 * - (1 3)>>
<-1 * <CrPr 1-0 * - (1 4)>>
<-1 * <CrPr 1-0 * - (1 5)>>
<-1 * <CrPr 1-0 * - (1 6)>>
<-1 * <CrPr 1-0 * - (1 7)>>
<-1 * <CrPr 1-0 * - (1 8)>>
<-1 * <CrPr 1-0 * - (1 9)>>
<-1 * <CrPr 1-0 * - (1 10)>>
<-1 * <CrPr 1-0 * - (1 11)>>
<-1 * <CrPr 1-0 * - (1 12)>>
<-1 * <CrPr 1-0 * - (1 13)>>
<-1 * <CrPr 1-0 * - (1 14)>>
<-1 * <CrPr 1-0 * - (1 15)>>
<-1 * <CrPr 1-0 * - (1 16)>>
-------------------------------------------------------------------------------
```

The boundary of this combination in $S^2 \times_\tau K(\mathbb{Z}, 1)$ is

```
(? total-17 *)

-------------------------------------------------------------------------{CMBN 1}
<-17 * <CrPr 0 * - (1)>>
-------------------------------------------------------------------------------
```

to which corresponds in the effective chain complex, the boundary of the basis element in dimension 2, $gen2 = s2 \otimes *$:

```
(? echcm 2 gen2)  ==>

-------------------------------------------------------------------------{CMBN 1}
<17 * <TnPr * S1>>
-------------------------------------------------------------------------------
```

The printing of the 3 faces of the 17 simplices of the combination `fg-gen2` shows the geometrical organisation of the twisted product:

```
(dolist(iterm (cmbn-list fg-gen2))
       (dotimes (i 3)(print (face total-17 i 2 (gnrt iterm))))
                     (terpri))  ==>

<AbSm 0 <CrPr - * - NIL>>
<AbSm 0 <CrPr - * - NIL>>
<AbSm - <CrPr 0 * - (17)>>

<AbSm - <CrPr 0 * - (1)>>
<AbSm - <CrPr 0 * - (2)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (2)>>
<AbSm - <CrPr 0 * - (3)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (3)>>
<AbSm - <CrPr 0 * - (4)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (4)>>
<AbSm - <CrPr 0 * - (5)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (5)>>
<AbSm - <CrPr 0 * - (6)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (6)>>
<AbSm - <CrPr 0 * - (7)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (7)>>
<AbSm - <CrPr 0 * - (8)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (8)>>
<AbSm - <CrPr 0 * - (9)>>
<AbSm - <CrPr 0 * - (1)>>
```

```
<AbSm - <CrPr 0 * - (9)>>
<AbSm - <CrPr 0 * - (10)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (10)>>
<AbSm - <CrPr 0 * - (11)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (11)>>
<AbSm - <CrPr 0 * - (12)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (12)>>
<AbSm - <CrPr 0 * - (13)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (13)>>
<AbSm - <CrPr 0 * - (14)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (14)>>
<AbSm - <CrPr 0 * - (15)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (15)>>
<AbSm - <CrPr 0 * - (16)>>
<AbSm - <CrPr 0 * - (1)>>

<AbSm - <CrPr 0 * - (16)>>
<AbSm - <CrPr 0 * - (17)>>
<AbSm - <CrPr 0 * - (1)>>
```

———

We have seen that for the special case of loop spaces, there is a canonical fibration and that the function `twisted-crts-prdc` builds the total space $X \times_\tau \Omega X$.

```
(setf sph3 (sphere 3))  ==>

[K221 Simplicial-Set]

(setf tw3 (twisted-crts-prdc sph3))  ==>

[K243 Simplicial-Set]

(face tw3 3 3 (crpr 0 's3 7 +null-loop+))  ==>
```

```
<AbSm - <CrPr 1-0 * - <<Loop[S3]>>>>
```

It is of course possible to study, any other fibration, provided one defines a correct fibration τ. In the following example the fibration $\tau : S^3 \longrightarrow \Omega S^3$, is defined by $\tau(s3) = s3^{-2}$, where $s3^{-2}$ is a word in the Kan simplicial version of the first loop space of S^3. In Kenzo, this object is created by the statement:

```
(absm 0 (loop3 0 's3 -2))  ==>

<AbSm - <<Loop[S3\-2]>>>

(setf tws3 (build-smmr   :sorc sph3
                         :trgt (loop-space sph3)
                         :degr -1
                         :sintr #'(lambda (dmns gmsm) (absm 0 (loop3 0 's3 -2)))
                         :orgn '(s3-tw-lps3)))     ==>

[K248 Fibration]

(setf total (fibration-total tws3))  ==>

[K249 Simplicial-Set]

(homology total 0 7)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Component Z/2Z

Homology in dimension 3 :

Homology in dimension 4 :

Component Z/2Z

Homology in dimension 5 :

Homology in dimension 6 :

Component Z/2Z
```

Lisp files concerned in this chapter

`serre.lisp`, `searching-homology.lisp`.

Chapter 19

Simplicial groups fibrations

19.1 Introduction

This chapter is analogous to the Loop spaces fibrations chapter where algorithms have been specially designed for loop spaces. In the present case, we have at our disposal all the tools developped for the fibrations. Let $\mathcal{G}$ be a 0–reduced simplicial group and $\bar{W}\mathcal{G}$, its classifying space. We recall that if $\mathcal{G}$ is an Abelian simplicial group then $\bar{W}\mathcal{G}$ is also an Abelian simplicial group, otherwise if $\mathcal{G}$ is non–Abelian, then $\bar{W}\mathcal{G}$ is only a simplicial set. The software `Kenzo` implements the **canonical** twisted cartesian product of $\bar{W}\mathcal{G}$ by $\mathcal{G}$, denoted $\mathcal{WG} = \bar{W}\mathcal{G} \times_\tau \mathcal{G}$. The *canonical fibration*

$$\mathcal{G} \quad \hookrightarrow \quad \mathcal{WG} \quad \twoheadrightarrow \quad \bar{W}\mathcal{G},$$

is defined by the following twisting operator:

$$\tau[(g_{n-1}, g_{n-2}, \ldots, g_0)] = g_{n-1}.$$

The total space $\mathcal{WG} = \bar{W}\mathcal{G} \times_\tau \mathcal{G}$ is acyclic.

19.2 The function for the canonical fibration

`smgr-fibration` *smgr* *[Function]*

Build the simplicial morphism of degree -1 corresponding to the canonical fibration. The source is the classifying space of the simplicial group *smgr* (built internally by the function), the target is *smgr* and the internal lisp function value of the slot `:sintr` implements the twisting operator. The returned object is a fibration.

Examples

Let us take the Abelian simplicial group $K(\mathbb{Z}, 1)$ and build the fibration morphism.

```
(setf k1 (k-z-1))  ==>

[K1 Abelian-Simplicial-Group]

(setf tw (smgr-fibration k1))  ==>

[K25 Fibration]
```

If needed, the classifying space may be found in the slot `:sorc` of the fibration:

```
(setf k2 (sorc tw))  ==>

[K13 Abelian-Simplicial-Group]
```

We may build now the total space of the fibration. We see that `Kenzo` returns a Kan simplicial set because the base and fiber spaces are also of Kan type. Then we test the face and differential operator upon a simplex of degree 4 of the total space `tt`.

```
(setf tt (fibration-total tw))  ==>

[K31 Kan-Simplicial-Set]

(setf gmsm (crpr 0 (gbar 4 0 '(10 11 12) 0 '(20 21) 0 '(30) 0 '())
                 0 '(2 4 6 8)))  ==>

<CrPr - <<GBar<- (10 11 12)><- (20 21)><- (30)><- NIL>>> - (2 4 6 8)>

(dotimes (i 5) (print (face tt i 4 gmsm)))  ==>

<AbSm - <CrPr - <<GBar<- (11 12)><- (21)><- NIL>>> - (4 6 8)>>
<AbSm - <CrPr - <<GBar<- (21 12)><- (41)><- NIL>>> - (6 6 8)>>
<AbSm - <CrPr - <<GBar<- (10 23)><- (50)><- NIL>>> - (2 10 8)>>
<AbSm - <CrPr - <<GBar<- (30 32)><- (30)><- NIL>>> - (2 4 14)>>
<AbSm - <CrPr - <<GBar<- (20 21)><- (30)><- NIL>>> - (12 15 18)>>

(? tt 4 gmsm)  ==>
```

```
------------------------------------------------------------------------{CMBN 3}
<1 * <CrPr - <<GBar<- (10 23)><- (50)><- NIL>>> - (2 10 8)>>
<1 * <CrPr - <<GBar<- (11 12)><- (21)><- NIL>>> - (4 6 8)>>
<1 * <CrPr - <<GBar<- (20 21)><- (30)><- NIL>>> - (12 15 18)>>
<-1 * <CrPr - <<GBar<- (21 12)><- (41)><- NIL>>> - (6 6 8)>>
<-1 * <CrPr - <<GBar<- (30 32)><- (30)><- NIL>>> - (2 4 14)>>
------------------------------------------------------------------------------

(? tt *)  ==>

------------------------------------------------------------------------{CMBN 2}
------------------------------------------------------------------------------
```

We may also build the Ronald Brown reduction and test the differential upon a simplex belonging to the twisted tensor product (slot `:bcc` of the reduction).

```
(setf br (brown-reduction tw))  ==>

[K59 Reduction]

(setf tw-pr (bcc br))  ==>

[K57 Chain-Complex]

(? tw-pr 4 (tnpr 4 (gbar 4 0 '(1 10 100) 0 '(1000 10000) 0 '(100000) 0 '())
                           0 '()))

------------------------------------------------------------------------{CMBN 3}
<-1 * <TnPr <<GBar>> (111 11000 100000)>>
<1 * <TnPr <<GBar>> (111 101000 10000)>>
<1 * <TnPr <<GBar>> (11011 100 100000)>>
<-1 * <TnPr <<GBar>> (11011 100000 100)>>
<-1 * <TnPr <<GBar>> (101001 110 10000)>>
<1 * <TnPr <<GBar>> (101001 10010 100)>>
<1 * <TnPr <<GBar<- (100000)><- NIL>>> (100)>>
<1 * <TnPr <<GBar<- (1 110)><- (101000)><- NIL>>> NIL>>
<1 * <TnPr <<GBar<- (10 100)><- (10000)><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<- (11 100)><- (11000)><- NIL>>> NIL>>
<1 * <TnPr <<GBar<- (1000 10000)><- (100000)><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<- (1001 10010)><- (100000)><- NIL>>> NIL>>
------------------------------------------------------------------------------

(? tw-pr *)  ==>

------------------------------------------------------------------------{CMBN 2}
------------------------------------------------------------------------------
```

19.3 The essential contraction for simplicial groups fibrations

It is known that if $\mathcal{G}$ is a 0–reduced simplicial group, the space $\mathcal{WG} = \bar{\mathcal{W}}\mathcal{G} \times_\tau \mathcal{G}$ is contractible. So it is possible to build a reduction of this space over $\mathbb{Z}$. This reduction depends on the important contraction

$$\chi_{\times\tau} : \bar{\mathcal{W}}\mathcal{G} \times_\tau \mathcal{G} \longrightarrow \bar{\mathcal{W}}\mathcal{G} \times_\tau \mathcal{G},$$

defined by:

$$\chi_{\times\tau}[(g_{n-1}, \ldots, g_0), g_n] = (-1)^{n+1}(g_n, g_{n-1}, \ldots, g_0, *),$$

where $g_n \in \mathcal{G}$, $(g_{n-1}, \ldots, g_0)$ is a gbar in $\bar{\mathcal{W}}\mathcal{G}$, and $*$ is the neutral element of $\mathcal{G}_{n+1}$ (in fact the n–th degeneracy of the base point of $\mathcal{G}$). We recall also that $\chi_{\times\tau}$ is a homotopy operator (degree: $+1$) satisfying the relation

$$d \circ \chi_{\times\tau} + \chi_{\times\tau} \circ d = 1.$$

In addition, if we consider the Brown reduction

$$\begin{array}{ccc} \bar{\mathcal{W}}\mathcal{G} \times_\tau \mathcal{G} & \stackrel{h}{\longrightarrow} & {}^s\bar{\mathcal{W}}\mathcal{G} \times_\tau \mathcal{G} \\ {\scriptstyle f}\downarrow\uparrow{\scriptstyle g} & & \\ \bar{\mathcal{W}}\mathcal{G} \otimes_\tau \mathcal{G} & & \end{array}$$

there is an induced contraction

$$\chi_{\otimes t} : \bar{\mathcal{W}}\mathcal{G} \otimes_\tau \mathcal{G} \longrightarrow \bar{\mathcal{W}}\mathcal{G} \otimes_\tau \mathcal{G},$$

defined by $\chi_{\otimes t} = f \circ \chi_{\times\tau} \circ g$. This implies that the twisted tensor product is also contractible over $\mathbb{Z}$.
The two functions of `Kenzo` for this construction are:

`smgr-crts-contraction` *smgr* *[Function]*
Return the homotopy morphism corresponding to the contraction $\chi_{\times\tau}$. The lisp definition is:

```
(defun SMGR-CRTS-CONTRACTION (smgr)
    (the morphism
      (build-mrph
       :sorc (fibration-total (smgr-fibration smgr))
       :trgt (fibration-total (smgr-fibration smgr))
       :degr +1
       :intr (smgr-crts-contraction-intr (bspn smgr))
       :strt :gnrt
       :orgn `(smgr-crts-contraction ,smgr))))
```

At execution time, the work is essentialy done by the function put in the slot `:intr`. This function is itself built by the internal function `smgr-crts-contraction-intr` requiring as argument the base point of the simplicial group.

`smgr-tnpr-contraction` *smgr* *[Function]*

Return the induced morphism $\chi_{\otimes t} = f \circ \chi_{\times\tau} \circ g$, where f and g are built by the Brown reduction. The lisp definition is:

```
(defun SMGR-TNPR-CONTRACTION (smgr
    &aux (fibration (smgr-fibration smgr))
         (brown (brown-reduction fibration))
         (f (f brown))
         (g (g brown))
         (chi (smgr-crts-contraction smgr)))

  (the morphism
     (i-cmps f chi g)))
```

Examples

It is an instructive exercise to build and check the reduction over $\mathbb{Z}$ of the space $\mathcal{WG} = \bar{\mathcal{W}}\mathcal{G} \times_\tau \mathcal{G}$ as we did in the loop spaces fibrations chapter.

$$\begin{array}{ccc} \bar{\mathcal{W}}\mathcal{G} \times_\tau \mathcal{G} & \xrightarrow{\chi_{\times t}} & {}^s\bar{\mathcal{W}}\mathcal{G} \times_\tau \mathcal{G} \\ {\scriptstyle aug}\downarrow\uparrow{\scriptstyle coaug} & & \\ \mathcal{C}_*(\mathbb{Z}) & & \end{array}$$

In this case, the two homomorphisms f and g of the reduction, are respectively the *augmentation* and *coaugmentation* morphisms. We define both morphisms according to our previous example based on $K(\mathbb{Z}, 1)$. The unit chain complex corresponding to $\mathbb{Z}$ is built by the function `z-chcm`; its unique generator in degree 0 is `:z-gnrt`.

```
(setf k1 (k-z-1))  ==>

[K1 Abelian-Simplicial-Group]

(setf tw (smgr-fibration k1))  ==>

[K25 Fibration]

(setf tt (fibration-total tw))  ==>

[K31 Kan-Simplicial-Set]
```

The base point of the total space, needed for the coaugmentation, may be obtained by either of the following statements:

```
(bspn tt)  ==>

<CrPr - <<GBar>> - NIL>

(crpr 0 +null-gbar+ 0 nil)  ==>

<CrPr - <<GBar>> - NIL>

(setf aug (build-mrph
              :sorc (fibration-total(smgr-fibration(k-z-1)))
              :trgt (z-chcm)
              :degr 0
              :intr #'(lambda (degr gnrt)
                        (if (zerop degr)
                            (term-cmbn 0 1 :z-gnrt)
                            (zero-cmbn degr)))
              :strt :gnrt
              :orgn '(aug-fibr-tot-smgr-fibr-k-z-1)  ))  ==>

[K38 Cohomology-Class (degree 0)]

(setf coaug (build-mrph
              :sorc (z-chcm)
              :trgt (fibration-total(smgr-fibration(k-z-1)))
              :degr 0
              :intr #'(lambda (degr gnrt)
                           (if (zerop degr)
                               (term-cmbn 0 1 (crpr 0 +null-gbar+ 0 '()))
                               (zero-cmbn degr)))
              :strt :gnrt
              :orgn '(coaug-fibr-tot-smgr-fibr-k-z-1) ))  ==>

[K39 Morphism (degree 0)]
```

```
(? aug 0 (bspn tt))  ==>

------------------------------------------------------------------------{CMBN 0}
<1 * Z-GNRT>
-------------------------------------------------------------------------------

(? coaug 0 :z-gnrt)  ==>

------------------------------------------------------------------------{CMBN 0}
<1 * <CrPr - <<GBar>> - NIL>>
-------------------------------------------------------------------------------

(setf chi-x-tau (smgr-crts-contraction (k-z-1)))  ==>

[K40 Morphism (degree 1)]
```

Let us apply the contraction morphism upon some simplices:

```
(setf *tc* (cmbn 0 1 (crpr 0 +null-gbar+ 0 '())))  ==>

------------------------------------------------------------------------{CMBN 0}
<1 * <CrPr - <<GBar>> - NIL>>
-------------------------------------------------------------------------------

(? chi-x-tau *)  ==>

------------------------------------------------------------------------{CMBN 1}
-------------------------------------------------------------------------------

(setf *tc* (cmbn 4 1 (crpr 0 (gbar 4 0 '(10 11 12) 0 '(20 21) 0 '(30) 0 '())
                           0 '(2 4 6 8))))  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <CrPr - <<GBar<- (10 11 12)><- (20 21)><- (30)><- NIL>>> - (2 4 6 8)>>
-------------------------------------------------------------------------------

(? chi-x-tau *)  ==>

------------------------------------------------------------------------{CMBN 5}
<-1 * <CrPr - <<GBar<- (2 4 6 8)><- (10 11 12)><- (20 21)><- (30)><- NIL>>>
              4-3-2-1-0 NIL>>
-------------------------------------------------------------------------------
```

```
(setf *tc* (cmbn 3 1 (crpr 0 (gbar 3 0 '(10 11) 0 '(20) 0 '())
                           0 '(2 4 6))))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <CrPr - <<GBar<- (10 11)><- (20)><- NIL>>> - (2 4 6)>>
-------------------------------------------------------------------------------

(? chi-x-tau *)  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <CrPr - <<GBar<- (2 4 6)><- (10 11)><- (20)><- NIL>>> 3-2-1-0 NIL>>
-------------------------------------------------------------------------------
```

We may build now the reduction over $\mathbb{Z}$ and test it upon the various simplices above:

```
(setf rdct (build-rdct :f aug
                       :g coaug
                       :h chi-x-tau
                       :orgn '(reduction-tt-z)))   ==>

[K41 Reduction]

(pre-check-rdct rdct)  ==>

---done---

(setf *tc* (cmbn 0 1 (crpr 0 +null-gbar+ 0 '())))  ==>

------------------------------------------------------------------------{CMBN 0}
<1 * <CrPr - <<GBar>> - NIL>>
-------------------------------------------------------------------------------

(setf *bc* (cmbn 0 1 :z-gnrt))  ==>

------------------------------------------------------------------------{CMBN 0}
<1 * Z-GNRT>
-------------------------------------------------------------------------------

(check-rdct)  ==>

*TC* =>
------------------------------------------------------------------------{CMBN 0}
<1 * <CrPr - <<GBar>> - NIL>>
-------------------------------------------------------------------------------
```

```
*BC* =>
------------------------------------------------------------------------{CMBN 0}
<1 * Z-GNRT>
-------------------------------------------------------------------------------

               .......  All results null ........

---done---

(setf *tc* (cmbn 4 1 (crpr 0 (gbar 4 0 '(10 11 12) 0 '(20 21) 0 '(30) 0 '())
                           0 '(2 4 6 8))))  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <CrPr - <<GBar<- (10 11 12)><- (20 21)><- (30)><- NIL>>> - (2 4 6 8)>>
-------------------------------------------------------------------------------

(check-rdct)  ==>

               .......  All results null ........

---done---

(setf *tc* (cmbn 3 1 (crpr 0 (gbar 3 0 '(10 11) 0 '(20) 0 '())
                             0 '(2 4 6))))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <CrPr - <<GBar<- (10 11)><- (20)><- NIL>>> - (2 4 6)>>
-------------------------------------------------------------------------------

(check-rdct)  ==>

               .......  All results null ........

---done---
```

We build now the reduction over $\mathbb{Z}$ of the twisted tensor product. The contraction is obtained by a call to the function `smgr-tnpr-contraction`.

$$\begin{array}{ccc} \bar{\mathcal{W}}\mathcal{G} \otimes_\tau \mathcal{G} & \xrightarrow{\chi \otimes t} & {}^s\bar{\mathcal{W}}\mathcal{G} \otimes_\tau \mathcal{G} \\ {\scriptstyle aug}\downarrow\uparrow{\scriptstyle coaug} & & \\ \mathcal{C}_*(\mathbb{Z}) & & \end{array}$$

Of course, we have to define the augmentation and coaugmentation morphisms adapted to this new example. The user will note that the twisted tensor product chain complex may be obtained from the slot `:sorc` of the contration $\chi_{\otimes t}$.

```
(setf chi-t-tau (smgr-tnpr-contraction (k-z-1)))  ==>

[K100 Morphism (degree 1)]

(setf aug-t (build-mrph
              :sorc (sorc chi-t-tau)
              :trgt (z-chcm)
              :degr 0
              :intr #'(lambda (degr gnrt)
                        (if (zerop degr)
                            (term-cmbn 0 1 :z-gnrt)
                            (zero-cmbn degr)))
              :strt :gnrt
              :orgn '(aug-t-fibr-tot-smgr-fibr-k-z-1)  ))   ==>

[K209 Cohomology-Class (degree 0)]

(setf coaug-t (build-mrph
                :sorc (z-chcm)
                :trgt (sorc chi-t-tau)
                :degr 0
                :intr #'(lambda (degr gnrt)
                          (if (zerop degr)
                              (term-cmbn 0 1 (tnpr 0 +null-gbar+ 0 '()))
                              (zero-cmbn degr)))
                :strt :gnrt
                :orgn '(coaug-t-fibr-tot-smgr-fibr-k-z-1) ))  ==>

[K210 Morphism (degree 0)]

(setf rdct-t (build-rdct :f aug-t
                         :g coaug-t
                         :h chi-t-tau
                         :orgn '(reduction-t-tt-z)))   ==>

[K211 Reduction]

(pre-check-rdct rdct-t)

---done---

(setf *tc* (cmbn 0 1 (tnpr 0 +null-gbar+ 0 '())))  ==>
```

```
-------------------------------------------------------------------------{CMBN 0}
<1 * <TnPr <<GBar>> NIL>>
-------------------------------------------------------------------------------

(setf *bc* (cmbn 0 1 :z-gnrt))  ==>

-------------------------------------------------------------------------{CMBN 0}
<1 * Z-GNRT>
-------------------------------------------------------------------------------

(check-rdct)  ==>

.......  All results null .....

---done---
```

The check is validated also for the following simplices. We see also that, if we apply the contraction $\chi_{\otimes t}$ upon various simplices outside the base fiber, the result is in general non–null. The nullity of this contraction outside the base fiber, which we have verified experimentally in many cases in loop spaces fibration is not verified in simplicial group fibrations.

```
(setf *tc* (cmbn 3 1 (tnpr 0 +null-gbar+ 3 '(1 10 100))))  ==>

-------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr <<GBar>> (1 10 100)>>
-------------------------------------------------------------------------------

(? chi-t-tau *tc*)  ==>

-------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr <<GBar<- (1 10 100)><1-0 NIL><0 NIL><- NIL>>> NIL>>
-------------------------------------------------------------------------------

(setf *tc* (cmbn 3 1 (tnpr 2 (gbar 2 0 '(1) 0 '()) 1 '(10))))  ==>

-------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr <<GBar<- (1)><- NIL>>> (10)>>
-------------------------------------------------------------------------------

(? chi-t-tau *tc*)  ==>

-------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr <<GBar<1-0 (10)><1-0 NIL><- (1)><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<2-0 (10)><1 (1)><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<2-1 (10)><0 (1)><0 NIL><- NIL>>> NIL>>
-------------------------------------------------------------------------------
```

```
(setf *tc* (cmbn 3 1 (tnpr 3 (gbar 3 0 '(1 10) 0 '(100) 0 '()) 0 '())))  ==>

------------------------------------------------------------------------{CMBN 3}
<1 * <TnPr <<GBar<- (1 10)><- (100)><- NIL>>> NIL>>
-------------------------------------------------------------------------------

(? chi-t-tau *tc*)  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr <<GBar<2 (1 10)><1 (100)><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<2-1 (11)><0 (100)><0 NIL><- NIL>>> NIL>>
-------------------------------------------------------------------------------

(setf *tc* (cmbn 4 1 (tnpr 0 +null-gbar+ 4 '(1 10 100 1000))))  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr <<GBar>> (1 10 100 1000)>>
-------------------------------------------------------------------------------

(? chi-t-tau *tc*)  ==>

------------------------------------------------------------------------{CMBN 5}
<-1 * <TnPr <<GBar<- (1 10 100 1000)><2-1-0 NIL><1-0 NIL><0 NIL><- NIL>>> NIL>>
-------------------------------------------------------------------------------

(setf *tc* (cmbn 4 1 (tnpr 2 (gbar 2 0 '(1) 0 '()) 2 '(10 100))))  ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr <<GBar<- (1)><- NIL>>> (10 100)>>
-------------------------------------------------------------------------------

(? chi-t-tau *tc*)  ==>

------------------------------------------------------------------------{CMBN 5}
<-1 * <TnPr <<GBar<1-0 (10 100)><2-1-0 NIL><1-0 NIL><- (1)><- NIL>>> NIL>>
<1 * <TnPr <<GBar<2-0 (10 100)><2-1-0 NIL><1 (1)><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<2-1 (10 100)><2-1-0 NIL><0 (1)><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3-0 (10 100)><2-1 (1)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3-1 (10 100)><2-0 (1)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3-2 (10 100)><1-0 (1)><1-0 NIL><0 NIL><- NIL>>> NIL>>
-------------------------------------------------------------------------------

(setf *tc* (cmbn 4 1 (tnpr 3 (gbar 3 0 '(1 10) 0 '(100) 0 '())
                           1 '(1000))))   ==>
```

```
------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr <<GBar<- (1 10)><- (100)><- NIL>>> (1000)>>
------------------------------------------------------------------------------

(? chi-t-tau *tc*)  ==>

------------------------------------------------------------------------{CMBN 5}
<-1 * <TnPr <<GBar<2 (1 10 1000)><2-1-0 NIL><1 (100)><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<2-1 (11 1000)><2-1-0 NIL><0 (100)><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<2-1-0 (1000)><2-1-0 NIL><- (1 10)><- (100)><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3 (1 10 1000)><2-1 (100)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3 (1 1000 10)><2-1 (100)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3 (1000 1 10)><2-0 (100)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3-1 (11 1000)><2-0 (100)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3-1-0 (1000)><2 (1 10)><1-0 NIL><- (100)><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3-2 (11 1000)><1-0 (100)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3-2 (1000 11)><1-0 (100)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3-2-0 (1000)><1 (1 10)><1 (100)><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3-2-1 (1000)><0 (1 10)><0 (100)><0 NIL><- NIL>>> NIL>>
------------------------------------------------------------------------------

(setf *tc* (cmbn 4 1 (tnpr 4 (gbar 4 0 '(1 10 100)
                                    0 '(1000 10000)
                                    0 '(100000)
                                    0 '())
                               0 '()))))     ==>

------------------------------------------------------------------------{CMBN 4}
<1 * <TnPr <<GBar<- (1 10 100)><- (1000 10000)><- (100000)><- NIL>>> NIL>>
------------------------------------------------------------------------------

(? chi-t-tau *tc*)  ==>

------------------------------------------------------------------------{CMBN 5}
<1 * <TnPr <<GBar<2 (1001 10010 100)><2-1-0 NIL><1 (100000)><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<2-1 (11011 100)><2-1-0 NIL><0 (100000)><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3 (1 10 100)><2 (1000 10000)><1-0 NIL><- (100000)><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3 (111 1000 10000)><2-0 (100000)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3 (1001 110 10000)><2-1 (100000)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3 (1001 10010 100)><2-1 (100000)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3-1 (11011 100)><2-0 (100000)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3-2 (1 110)><1 (1000 10000)><1 (100000)><0 NIL><- NIL>>> NIL>>
<1 * <TnPr <<GBar<3-2 (111 11000)><1-0 (100000)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3-2 (11011 100)><1-0 (100000)><1-0 NIL><0 NIL><- NIL>>> NIL>>
<-1 * <TnPr <<GBar<3-2-1 (111)><0 (1000 10000)><0 (100000)><0 NIL><- NIL>>> NIL>>
------------------------------------------------------------------------------
```

Let us verify once again the unproved conjecture signaled in the loop spaces fibrations chapter. In fact, if the conjecture is true, it is sufficient to prove it with $\bar{\Delta}$. But the following is also a good test of verifying all the involved machinery. If we take $\mathcal{G} = K(\mathbb{Z}, 1)$, the space $\mathcal{WG} = \bar{\mathcal{W}}\mathcal{G} \times_\tau \mathcal{G}$ is 0–reduced. Let us consider its loop space and the induced contraction $\chi_{\otimes t}$ over $\mathcal{WG} \otimes_\tau \Omega\mathcal{WG}$. $\mathcal{WG}$ and the contraction $\chi_{\otimes t}$ are built by the two following statements:

```
(setf wgkz1 (fibration-total (smgr-fibration(k-z-1))))  ==>

[K31 Kan-Simplicial-Set]

(setf chi-t-tau (tnpr-contraction wgkz1))  ==>

[K84 Morphism (degree 1)]
```

Let us apply successively the contraction upon a simplex belonging to the base fiber (`gnrt0`) and a simplex out of the base fiber (`gnrt1`). We see that in the second case we obtain a null result. We verify also that $\chi_{\otimes t} \circ \chi_{\otimes t} = 0$.

```
(setf gnrt0 (tnpr 0 (crpr 0 +null-gbar+  0 nil)
                  0 (loop3 0 (crpr 0 (gbar 2 0 '(1) 0 '(0) ) 0 '(1)) -3)))  ==>

<TnPr <CrPr - <<GBar>> - NIL> <<Loop[<CrPr - <<GBar<- (1)><- (0)>>> - (1)>\-3]>>>

(? chi-t-tau 0 gnrt0)  ==>

-------------------------------------------------------------------------{CMBN 1}
<1 * <TnPr <CrPr - <<GBar<- (1)><- (0)>>> - (1)>
           <<Loop[<CrPr - <<GBar<- (1)><- (0)>>> - (1)>\-3]>>>>
<1 * <TnPr <CrPr - <<GBar<- (1)><- (0)>>> - (1)>
           <<Loop[<CrPr - <<GBar<- (1)><- (0)>>> - (1)>\-2]>>>>
<1 * <TnPr <CrPr - <<GBar<- (1)><- (0)>>> - (1)>
           <<Loop[<CrPr - <<GBar<- (1)><- (0)>>> - (1)>\-1]>>>>
-------------------------------------------------------------------------------

(? chi-t-tau *)  ==>

-------------------------------------------------------------------------{CMBN 2}
-------------------------------------------------------------------------------

(setf gnrt1 (tnpr 4 (crpr 0 (gbar 4 0 '(10 11 12) 0 '(20 21) 0 '(30) 0 '())
                          0 '(2 4 6 8))
                  0 (loop3 0 (crpr 0 (gbar 2 0 '(1) 0 '(0) ) 0 '(1)) 2)))  ==>

<TnPr <CrPr - <<GBar<- (10 11 12)><- (20 21)><- (30)><- NIL>>> - (2 4 6 8)>
      <<Loop[<CrPr - <<GBar<- (1)><- (0)>>> - (1)>\2]>>>
```

```
(? chi-t-tau 4 gnrt1)  ==>

-------------------------------------------------------------------------{CMBN 5}
-------------------------------------------------------------------------------
```

At last we verify that the total space of the fibration of $K(\mathbb{Z},1)$ is contractible.

```
(homology (fibration-total(smgr-fibration (k-z-1))) 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Homology in dimension 4 :

Homology in dimension 5 :

Homology in dimension 6 :

Homology in dimension 7 :

Homology in dimension 8 :

Homology in dimension 9 :

---done---
```

Another example is the following: we build the canonical fibration of $\bar{W}^2K(\mathbb{Z}_2,1)$ and verify that its fibration total space is contractible:

```
(setf k22 (classifying-space(classifying-space(k-z2-1))))  ==>

[K306 Abelian-Simplicial-Group]

(setf fb (smgr-fibration k22))  ==>

[K836 Fibration]

(setf tt2 (fibration-total fb))  ==>

[K837 Kan-Simplicial-Set]
```

```
(homology tt2 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Homology in dimension 4 :

Homology in dimension 5 :

Homology in dimension 6 :

Homology in dimension 7 :

Homology in dimension 8 :

Homology in dimension 9 :

--done--
```

Lisp file concerned in this chapter

`cs-twisted-products.lisp`.

Chapter 20

Eilenberg-Moore spectral sequence II

20.1 Introduction

This chapter is devoted to the effective homology version of the spectral sequence of Eilenberg-Moore, in the particular case of classifying spaces. More precisely, let $\mathcal{G}$ be a simplicial group; then its classifying space $\bar{\mathcal{W}}\mathcal{G}$ is a simplicial set canonically defined[1]. Furthermore, if $\mathcal{G}$ is an Abelian simplicial group, then $\bar{\mathcal{W}}\mathcal{G}$ is again an Abelian simplicial group with natural structure, so that the $\bar{\mathcal{W}}$ construction can be iterated. The $\bar{\mathcal{W}}$ construction is implemented in `Kenzo` only if $\mathcal{G}$ is reduced ($\mathcal{G}_0$ has only one element). In particular if $\mathcal{G}$ is a simplicial group with effective homology, then `Kenzo` constructs a version of $\bar{\mathcal{W}}\mathcal{G}$ also with effective homology.

[1] **J. Peter May**. *Simplicial objects in algebraic topology*, Van Nostrand, 1967.

20.2 The detailed construction

Let $\mathcal{G}$ be a simplicial group with effective homology. This means that a homotopy equivalence:

$$\begin{array}{ccc} & \hat{C}_* & \\ \swarrow & \nearrow & \searrow \quad \nwarrow \\ \mathcal{C}_*(\mathcal{G}) & & EG_* \end{array}$$

is provided, where the chain complex EG_* is effective and must be considered as describing the homology of $\mathcal{C}_*(\mathcal{G})$. This scheme includes the case where $\mathcal{C}_*(\mathcal{G})$ itself is effective; without any other information, the program constructs automatically a trivial homotopy equivalence.
Now, if we apply the *Bar* functor to this homotopy equivalence we obtain the homotopy equivalence H_R:

$$\begin{array}{ccc} & \widetilde{Bar}^{\hat{C}_*} & \\ \swarrow & \nearrow & \searrow \quad \nwarrow \\ Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z},\mathbb{Z}) & & \widetilde{Bar}^{EG_*}(\mathbb{Z},\mathbb{Z}) \end{array}$$

in which the $\widetilde{Bar}$'s are Bars constructions with respect to the A_∞ structure on $\hat{C}_*$ and EG_* defined by the initial homotopy equivalence.
By analogy with the result of Julio Rubio[2] one may show that it is possible to construct another homotopy equivalence, H_L:

$$\begin{array}{ccc} & Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \mathcal{C}_*(\mathcal{G}),\mathbb{Z}) & \\ \swarrow & \nearrow & \searrow \quad \nwarrow \\ \mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) & & Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z},\mathbb{Z}) \end{array}$$

[2] **J.J. Rubio-Garcia**. *Homologie effective des espaces de lacets itérés: un logiciel*, Thèse, Institut Fourier, 1991.

Both reductions of H_L are obtained by the basic perturbation lemma, as explained in the following section. The composition of both homotopy equivalences, H_L and H_R, makes the link between $\bar{\mathcal{W}}\mathcal{G}$, the classifying space of the simplicial group $\mathcal{G}$ and the effective right bottom chain complex of H_R.

20.2.1 Obtaining the left reduction H_L

The homotopy equivalence H_L is obtained by a sequence of intermediate constructions based mainly upon two applications of the basic perturbation lemma. We are led to consider the two following modules:

- $\mathcal{C}_*(\mathcal{G})$ module on itself, with the canonical product.

$$\mathcal{C}_*(\mathcal{G}) \otimes \mathcal{C}_*(\mathcal{G}) \xrightarrow{\Pi} \mathcal{C}_*(\mathcal{G}).$$

- $\check{\mathcal{C}}_*(\mathcal{G})$, module on $\mathcal{C}_*(\mathcal{G})$, with a "trivial" product

$$\check{\mathcal{C}}_*(\mathcal{G}) \otimes \mathcal{C}_*(\mathcal{G}) \xrightarrow{\check{\Pi}} \check{\mathcal{C}}_*(\mathcal{G})$$

  defined by

$$\sigma \otimes \tau \longmapsto \sigma.\varepsilon(\tau),$$

  where ε is the traditional augmentation of $\mathcal{C}_*(\mathcal{G})$.

Now, we consider the set of the followings Bars (where u means *"untwisted"* and t *"twisted"*):

$$\begin{aligned}
\mathrm{Hat}_{uu} &= Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes \check{\mathcal{C}}_*(\mathcal{G}), \mathbb{Z}),\\
\mathrm{Hat}_{ut} &= Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes \mathcal{C}_*(\mathcal{G}), \mathbb{Z}),\\
\mathrm{Hat}_{tu} &= Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \check{\mathcal{C}}_*(\mathcal{G}), \mathbb{Z}),\\
\mathrm{Hat}_{tt} &= Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \mathcal{C}_*(\mathcal{G}), \mathbb{Z}).
\end{aligned}$$

One may always say that Hat_{ut} is obtained from Hat_{uu} by a perturbation δ_r (r: right) induced by the discrepancy between the products in $\check{\mathcal{C}}_*(\mathcal{G})$ and $\mathcal{C}_*(\mathcal{G})$ and that Hat_{tu} is obtained from Hat_{uu} by a perturbation δ_l (l: left) induced by the twisted tensor product $\otimes_t$. After that, Hat_{tt} is obtained from Hat_{tu} by the perturbation δ_r as well as, by commutativity, from Hat_{ut} by the perturbation δ_l.

This is shown in the following diagram (here, the arrows are not reductions, but denote the perturbations between the differential morphisms):

$$\begin{array}{ccc} & \mathrm{Hat}_{uu} & \\ \delta_l \swarrow & & \searrow \delta_r \\ \mathrm{Hat}_{tu} & & \mathrm{Hat}_{ut} \\ \delta_r \searrow & & \swarrow \delta_l \\ & \mathrm{Hat}_{tt} & \end{array}$$

The underlying graded modules Hat_{uu}, Hat_{ut}, Hat_{tu} and Hat_{tt} are the same and the program keeps Hat_{uu} as the underlying graded module for all the chain complexes. The differential perturbations are given by the formulas:

$$\begin{aligned} \delta_l[(w \otimes \check{g}_0) \otimes (g_1 \otimes \cdots \otimes g_n)] &= [d_{\otimes t}(w \otimes \check{g}_0) - d_{\otimes}(w \otimes \check{g}_0)] \otimes (g_1 \otimes \cdots \otimes g_n), \\ \delta_r[(w \otimes \check{g}_0) \otimes (g_1 \otimes \cdots \otimes g_n)] &= w \otimes (g_0.g_1) \otimes g_2 \otimes \cdots \otimes g_n, \end{aligned}$$

where $\check{g}_0 \in \check{\mathcal{G}}_0$, $g_i \in \mathcal{G}$, $w \in \bar{\mathcal{W}}\mathcal{G}$.
Now, on the other hand, we know that there exists a reduction

$$\mathrm{Hat}_{ut} = Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes \mathcal{C}_*(\mathcal{G}), \mathbb{Z}) \Longrightarrow \mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}),$$

so, perturbing this reduction by δ_l, one obtains the Rubio reduction

$$\mathrm{Hat}_{tt} = Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \mathcal{C}_*(\mathcal{G}), \mathbb{Z}) \Longrightarrow \mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}).$$

On the other hand, we know also that there exists a reduction

$$\mathrm{Hat}_{tu} = Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \check{\mathcal{C}}_*(\mathcal{G}), \mathbb{Z}) \Longrightarrow Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z}),$$

so, perturbing this reduction by δ_r, one obtains the reduction

$$\mathrm{Hat}_{tt} = Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \mathcal{C}_*(\mathcal{G}), \mathbb{Z}) \Longrightarrow Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z}).$$

Finally, we have obtained the wished left homotopy equivalence H_L:

$$\begin{array}{ccc} & Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \mathcal{C}_*(\mathcal{G}), \mathbb{Z}) & \\ \swarrow \nearrow & & \searrow \nwarrow \\ \mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) & & Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z}) \end{array}$$

20.2.2 The useful functions

For the applications, the only function that the user must know is the function `classifying-space-efhm` which builds the final homotopy equivalence. But for the interested user, we give nevertheless a short description of all the functions involved in the described process.

`classifying-space-efhm` *smgr* *[Function]*
: From the simplicial group $\mathcal{G}$ (0-reduced) with effective homology (here the argument *smgr*), build a homotopy equivalence giving an effective homology version of the classifying space $\bar{\mathcal{W}}\mathcal{G}$ of $\mathcal{G}$. This homotopy equivalence will be used by the homology function to compute the homology groups. In fact, due to the slot-unbound mechanism of CLOS, this function will be automatically called, as soon as the user requires a homology group for a classifying space.

`cs-hat-u-u` *smgr* *[Function]*
: Return the chain complex $Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes \check{\mathcal{C}}_*(\mathcal{G}), \mathbb{Z})$. Because of the particular structure of $\check{\mathcal{C}}_*(\mathcal{G})$, this chain complex is nothing but $[\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes \check{\mathcal{C}}_*(\mathcal{G})] \otimes Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z})$.

`cs-hat-right-perturbation` *smgr* *[Function]*
: Return the morphism corresponding to the differential perturbation δ_r, induced by the discrepancy between the respective products in $\check{\mathcal{C}}_*(\mathcal{G})$ and in $\mathcal{C}_*(\mathcal{G})$.

`cs-hat-u-t` *smgr* *[Function]*
: Return the chain complex $Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes \mathcal{C}_*(\mathcal{G}), \mathbb{Z})$ by applying the differential perturbation `hat-right-perturbation` upon the chain complex `hat-u-u` *smgr*. This is realized by the method `add`.

`cs-hat-t-u` *smgr* *[Function]*
: Return the chain complex $Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \check{\mathcal{C}}_*(\mathcal{G}), \mathbb{Z})$. Because of the particular structure of $\check{\mathcal{C}}_*(\mathcal{G})$, this chain complex is nothing but $[\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \check{\mathcal{C}}_*(\mathcal{G})] \otimes Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z})$, where the twisted tensor product is the botton chain complex of the Brown reduction of the fibration of the simplicial group.

`cs-hat-left-perturbation` *smgr* *[Function]*
: Return the morphism corresponding to the differential perturbation δ_l induced by the twisted tensor product. This morphism is nothing but the tensor product of two morphisms: the perturbation morphism by-product of the Brown reduction of the fibration of the simplicial group and the identity morphism on $Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z})$.

`cs-left-hmeq-hat` *smgr* *[Function]*

Return the chain complex $\mathrm{Hat}_{tt} = Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \mathcal{C}_*(\mathcal{G}), \mathbb{Z})$ by perturbing the chain complex Hat_{ut} by the perturbation δ_l.

`cs-pre-left-hmeq-left-reduction` *smgr* *[Function]*

Build the reduction

$$\mathrm{Hat}_{tu} = Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes \mathcal{C}_*(\mathcal{G}), \mathbb{Z}) \Longrightarrow \mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}).$$

`cs-pre-left-hmeq-right-reduction` *smgr* *[Function]*

Build the reduction

$$\mathrm{Hat}_{ut} = Bar^{\mathcal{C}_*(\mathcal{G})}(\check{\mathcal{C}}_*(\mathcal{G}) \otimes_t \mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}), \mathbb{Z}) \Longrightarrow Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z}).$$

`cs-left-hmeq-left-reduction` *smgr* *[Function]*

Build the Rubio reduction

$$Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \mathcal{C}_*(\mathcal{G}), \mathbb{Z}) \Longrightarrow \mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G})$$

by perturbing by the perturbation δ_r the reduction

$$\mathrm{Hat}_{tu} \Longrightarrow \mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G})$$

obtained by the function `pre-left-hmeq-left-reduction`.

`cs-left-hmeq-right-reduction` *smgr* *[Function]*

Build the reduction

$$Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \mathcal{C}_*(\mathcal{G}), \mathbb{Z}) \Longrightarrow Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z})$$

by perturbing by the perturbation δ_l the reduction

$$\mathrm{Hat}_{ut} \Longrightarrow Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z})$$

obtained by the function `cs-pre-left-hmeq-right-reduction`.

`cs-left-hmeq` *smgr* *[Function]*

Build the homotopy equivalence

$$\begin{array}{ccc} & Bar^{\mathcal{C}_*(\mathcal{G})}(\mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) \otimes_t \mathcal{C}_*(\mathcal{G}), \mathbb{Z}) & \\ \swarrow\nearrow & & \searrow\nwarrow \\ \mathcal{C}_*(\bar{\mathcal{W}}\mathcal{G}) & & Bar^{\mathcal{C}_*(\mathcal{G})}(\mathbb{Z}, \mathbb{Z}) \end{array}$$

from the above reductions. The function `classifying-space-efhm` composes this left homotopy equivalence H_L with the homotopy equivalence H_R which is the Bar of a pre-existing homotopy equivalence (possibly the trivial one) between the module $\mathcal{G}$ and an effective version of it, as described at the beginning of this chapter.

20.2.3 Searching homology for classifying spaces

The origin list of a classifying space object has the form `(CLASSIFYING-SPACE` *smgr*`)`. The `search-efhm` method applied to a classifying space object consists essentially in a call to the function `classifying-space-efhm` described just above.

```
(defmethod SEARCH-EFHM (classifying-space (orgn (eql 'classifying-space)))
  (declare (type simplicial-set classifying-space))
  (classifying-space-efhm (second (orgn classifying-space))))
```

The following Lisp definition of the function `classifying-space-efhm` shows that the process may be recursif if it happens that *smrg* is itself a classifying space:

```
(defun CLASSIFYING-SPACE-EFHM (smgr)
  (declare (type simplicial-group smgr))
  (let ((left-hmeq (cs-left-hmeq smgr))
        (right-hmeq (bar (efhm smgr))))
        (declare (type homotopy-equivalence left-hmeq right-hmeq))
        (cmps left-hmeq right-hmeq)))
```

Examples

```
(setf kz1 (k-z-1))  ==>

[K1 Abelian-Simplicial-Group]

(homology kz1 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z

Homology in dimension 2 :

Homology in dimension 3 :
```

```
Homology in dimension 4 :

Homology in dimension 5 :

Homology in dimension 6 :

Homology in dimension 7 :

Homology in dimension 8 :

Homology in dimension 9 :

---done---

(setf bkz1 (classifying-space kz1))  ==>

[K23 Abelian-Simplicial-Group]

(homology bkz1 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Component Z

Homology in dimension 3 :

Homology in dimension 4 :

Component Z

Homology in dimension 5 :

Homology in dimension 6 :

Component Z

Homology in dimension 7 :

Homology in dimension 8 :
```

```
Component Z

Homology in dimension 9 :

---done---

(setf obkz1 (loop-space bkz1))  ==>

[K154 Simplicial-Group]

(homology obkz1 0 6)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Component Z

Homology in dimension 2 :

Homology in dimension 3 :

Homology in dimension 4 :

Homology in dimension 5 :

---done---
```

Lisp files concerned in this chapter

`cs-space-efhm.lisp`, `searching-homology.lisp`.

Chapter 21

Computing homotopy groups

21.1 Mathematical aspects

The method used here to compute the homotopy groups is known as the **Whitehead tower**. We recall some mathematical points:

1) The Hurewicz theorem.

Theorem. *Let X be a 1–connected space such that all its homology groups $H_r(X)$ are null for $0 < r < n$, then $\pi_n(X) \simeq H_n(X)$.*

2) The properties of the classifying spaces.

Let X such that $\pi_r(X) = 0$ for $0 < r < n$. From now on, we shall denote $\pi_n(X)$ simply by π. Let us consider the universal coefficients exact sequence:

$$0 \longrightarrow Ext(H_{n-1}(X), \pi) \longrightarrow H^n(X, \pi) \longrightarrow Hom(H_n(X), \pi) \longrightarrow 0.$$

Here $H_{n-1}(X) = 0$, so that a canonical isomorphism is given:

$$H^n(X, \pi) \xrightarrow{\simeq} Hom(H_n(X), \pi),$$

but $H_n(X) \simeq \pi$ and this gives us a canonical element $h \in H^n(X, \pi)$ which is the fundamental n–th cohomology class in this context. A cocycle χ representing h is constructed as follows. In the chain complex

$$\cdots \to C_{n+1} \xrightarrow{d_{n+1}} C_n \xrightarrow{d_n} C_{n-1} \longrightarrow \cdots,$$

the kernel Z_n of d_n is a summand of $C_n(X)$:

$$C_n = Z_n \oplus S$$

and such a decomposition induces a projection $p : C_n(X) \longrightarrow Z_n$. Finally, we have also the canonical projection $p' : Z_n \longrightarrow H_n = \pi$. Then, χ can be defined by $\chi = p' \circ p$ and χ *is a cocycle*. If another p is chosen, another χ is obtained but the cohomology class is the same.
Let us fix now a choice for the cocycle $\chi : C_n(X) \longrightarrow \pi$. This cocycle induces in turn a simplicial map:

$$\varphi_\chi : C_n(X) \longrightarrow K(\pi, n),$$

where $K(\pi, n)$ is the classifying group $K(\pi_n(X), n)$ of the group $\pi_n(X) = \pi$. If χ' is another choice for the cocycle then $\varphi_{\chi'}$ is homotopic to φ_χ. We recall that the space $K(\pi, n)$ has the following properties:

1. $H_r(K(\pi, n)) = 0$ for $0 < r < n$,
2. $\pi_r(K(\pi, n)) = 0$ for $r \neq n$,
3. $\pi_n(K(\pi, n)) = H_n(K(\pi, n) = \pi$.

On the other hand, a canonical fibration of base space $K(\pi, n)$ and fibre space $K(\pi, n-1)$ is deduced from a canonical twisting operator

$$\tau : K(\pi, n) \longrightarrow K(\pi, n-1)$$

$$\begin{array}{c} K(\pi, n-1) \\ \downarrow \\ K(\pi, n) \times_\tau K(\pi, n-1) \\ \downarrow \\ K(\pi, n) \end{array}$$

So, if a simplicial map $\varphi_\chi : X \longrightarrow K(\pi, n)$ is given, it is then possible to construct a fibration of base space X and fibre space $K(\pi, n-1)$, the twisting

operator being then defined by: $\tau_\chi = \tau \circ \varphi_\chi$, according to the diagram:

$$\begin{array}{ccc} K(\pi, n-1) & \stackrel{=}{\longrightarrow} & K(\pi, n-1) \\ \downarrow & & \downarrow \\ X' = X \times_{\tau_\chi} K(\pi, n-1) & \stackrel{\psi_\chi}{\longrightarrow} & K(\pi, n) \times_\tau K(\pi, n-1) \\ \downarrow & & \downarrow \\ X & \stackrel{\varphi_\chi}{\longrightarrow} & K(\pi, n) \end{array}$$

3) The Serre exact sequence of the homotopy groups of fibrations.

Let us consider the fibration

$$F \quad \hookrightarrow \quad T = X \otimes_\tau F \quad \twoheadrightarrow \quad X,$$

The Serre exact sequence establishes a connection between the homotopy groups of the 3 spaces, in any dimension:

$$\cdots \longrightarrow \pi_{n+1}(F) \longrightarrow \pi_{n+1}(T) \longrightarrow \pi_{n+1}(X) \longrightarrow \pi_n(F) \longrightarrow$$

$$\longrightarrow \pi_n(T) \longrightarrow \pi_n(X) \longrightarrow \pi_{n-1}(F) \longrightarrow \cdots$$

In our special case, where the fibration is:

$$K(\pi, n-1) \quad \hookrightarrow \quad X' \quad \twoheadrightarrow \quad X,$$

the Serre sequence may be written:

$$\cdots \longrightarrow \pi_{n+1}(K(\pi, n-1)) \longrightarrow \pi_{n+1}(X') \longrightarrow \pi_{n+1}(X) \longrightarrow$$

$$\longrightarrow \pi_n(K(\pi, n-1)) \longrightarrow \pi_n(X') \longrightarrow \pi_n(X) \longrightarrow \pi_{n-1}(K(\pi, n-1)) \longrightarrow 0.$$

But we know that $\pi_i(K(\pi, n)) = 0$ for $i \neq n$ and that $\pi_n(K(\pi, n)) = \pi$, so the exact sequence may be re-written:

$$0 \to \pi_{n+1}(X') \to \pi_{n+1}(X) \to 0 \to \pi_n(X') \to \pi_n(X) \to \pi \to 0.$$

The subsequence:

$$0 \longrightarrow \pi_n(X') \longrightarrow \pi_n(X) \longrightarrow \pi \longrightarrow 0.$$

is exact; furthermore, the cocycle χ is such that the connection morphism $\partial : \pi_n(X) \to \pi$ is the identity; so that we deduce that $\pi_n(X') = 0$.
On the other hand, from the exactness of the subsequence

$$0 \longrightarrow \pi_{n+1}(X') \longrightarrow \pi_{n+1}(X) \longrightarrow 0$$

we deduce that $\pi_{n+1}(X') \simeq \pi_{n+1}(X)$ and more generally $\pi_r(X') \simeq \pi_r(X)$ for $r \neq n$. In particular, $\pi_r(X') = 0$ for $r \leq n+1$ and the Hurewicz theorem gives again:

$$H_{n+1}(X') \simeq \pi_{n+1}(X') \simeq \pi_{n+1}(X).$$

So, if we know how to compute $H_{n+1}(X')$ then we have obtained $\pi_{n+1}(X)$.

4) The Whitehead tower

Let us denote for a reason which will be clear in a moment the space X' by $X^{(n+1)}$. Due to the properties of $X^{(n+1)}$, we may iterate the process, namely, build the following fibrations, where $\pi_{n+1}(X^{(n+1)})$ is denoted simply by π':

$$\begin{array}{ccc}
K(\pi', n) & \xrightarrow{=} & K(\pi', n) \\
\downarrow & & \downarrow \\
X^{(n+2)} = X^{(n+1)} \times_{\tau_\chi} K(\pi', n) & \xrightarrow{\psi_\chi} & K(\pi', n+1) \times_\tau K(\pi', n) \\
\downarrow & & \downarrow \\
X^{(n+1)} & \xrightarrow{\varphi_\chi} & K(\pi', n+1)
\end{array}$$

where $X^{(n+2)} = X^{(n+1)} \times_{\tau_\chi} K(\pi', n)$ has the property

$$H_{n+2}(X^{(n+2)}) \simeq \pi_{n+2}(X^{(n+2)}) \simeq \pi_{n+2}(X^{(n+1)}) \simeq \pi_{n+2}(X).$$

This construction is known as the Whitehead tower.

21.2 The functions for computing homotopy groups

An important remark

In this version of Kenzo, only the case where the first non–null homology group (in non–null dimension) is $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$ can be processed; however if this homology group is a direct sum of several copies of $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$, then the corresponding stage of the Whitehead tower may also be constructed step by step.

`chml-clss` *chcm first* *[Function]*

Return a "fundamental" cohomology class, more precisely a cocycle χ defining it. The first argument *chcm* must be a chain complex C_* with effective homology; in particular, an effective chain complex EC_* is a by–product of the machine object C_*. The second argument "*first*" is an integer n, namely the first non–null dimension from which the chain complex *chcm* has a non–null homology group, which **must be isomorphic to** $\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z}$. The reader may be amazed that this argument "*first*" must be provided, since it is a consequence of the given C_*. But in fact, the same function `chml-class` may also be used in different contexts, for example, the Postnikov tower; in such a case, the argument *first* is not redundant. The returned cocycle χ is in any case a chain complex morphism $\chi : EC_* \longrightarrow \mathbb{Z}$, where $\mathbb{Z}$ is the unit chain complex created by the function `z-chcm`. The degree of the morphism is $-n$. It is important to note that the chain complex involved in the source of the morphism is the **effective** chain complex of the homotopy equivalence value of the slot `efhm` of the object *chcm*. See the section *The general method for computing homology* in the Homology chapter. Finally, if $H_n(C_*) = \mathbb{Z}/2\mathbb{Z}$, the actual cohomology class hoped by the user is the composition $p \circ \chi$, where p is the canonical projection $\mathbb{Z} \longrightarrow \mathbb{Z}/2\mathbb{Z}$. But nevertheless, the `chml-clss` lisp function returns χ and not $p \circ \chi$.

`z-whitehead` *smst chml-clss* *[Function]*

Return a fibration over the simplicial set *smst* (the first argument), canonically associated to the "cohomology class" *chml-clss* (the second argument). The simplicial set X, i.e. *smst*, is reduced; its homotopy groups $\pi_r(X)$ are null for $0 \leq r \leq n-1$. The first non null homotopy group $\pi_n(X)$ is assumed to be $\mathbb{Z}$, i.e.

$\pi_n(X) = H_n(X) = \mathbb{Z}$. The previous function `chml-clss`, in this situation, returns a cocycle χ, which may be used as the second argument *chml-clss* of the function `z-whitehead`. The integer n is determined by the absolute value of the degree of the cohomology class *chcm-clss*. As explained in the previous section, a canonical fibration is induced by χ and it is this fibration which is returned by `z-whitehead`. The slot `sintr` of the fibration is set by the internal function `z-whitehead-sintr` which builds in an efficient way the lisp function implementing the twisting operator $\tau \circ \varphi_\chi$.

`z2-whitehead` *smst chcm-clss* *[Function]*

Return a fibration over the simplicial set *smst* (the first argument), canonically associated to the "cohomology class" *chml-clss* (the second argument). The simplicial set X, i.e. *smst*, is reduced; its homotopy groups $\pi_r(X)$ are null for $0 \leq r \leq n-1$. The first non null homotopy group $\pi_n(X)$ is assumed to be $\mathbb{Z}/2\mathbb{Z}$, i.e. $\pi_n(X) = H_n(X) = \mathbb{Z}/2\mathbb{Z}$. The previous function `chml-clss`, in this situation, returns a "cocycle" χ, which may be used as the second argument *chml-clss* of the function `z2-whitehead`. The integer n is determined by the absolute value of the degree of the cohomology class *chcm-clss*. In this $\mathbb{Z}/2\mathbb{Z}$ case, χ is even not a cocycle. The actual cocycle is obtained by the composition $p \circ \chi$, p being the canonical projection $\mathbb{Z} \longrightarrow \mathbb{Z}/2\mathbb{Z}$. But the user is not concerned by these technicalities, because the function `z2-whitehead` makes itself the necessary conversion. The slot `sintr` of the fibration is set by the internal function `z2-whitehead-sintr` which builds in an efficient way the lisp function implementing the twisting operator $\tau \circ \varphi_\chi$.

Examples

Let us retrieve some known facts about S^3, in particular $\pi_4(S^3) \simeq \mathbb{Z}/2\mathbb{Z}$. We follow the theoritical method exposed above, namely build the fibration

$$K(\mathbb{Z},2) \quad \hookrightarrow \quad S^3 \times_{\tau_\chi} K(\mathbb{Z},2) \quad \twoheadrightarrow \quad S^3,$$

and get the homology group in dimension 4 of the total space.

```
(setf s3 (sphere 3))  ==>

[K1 Simplicial-Set]
```

```
(homology s3 0 4)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Component Z

---done---

(setf s3-chml-clss (chml-clss s3 3))  ==>

[K12 Cohomology-Class (degree 3)]

(setf s3-fibr (z-whitehead s3 s3-chml-clss))  ==>

[K37 Fibration]

(setf s3-total (fibration-total  s3-fibr))  ==>

[K43 Simplicial-Set]

(homology s3-total 0 6)

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Homology in dimension 4 :

Component Z/2Z

Homology in dimension 5 :

---done---
```

Let us show a similar example with the space $Moore(2,4)$. As $H_4 \simeq \mathbb{Z}/2\mathbb{Z}$, the function for building the fibration is `z2-whitehead`. We verify that $\pi_5(Moore(2,4)) \simeq \mathbb{Z}/2\mathbb{Z}$

```
(setf m24 (moore 2 4)) ==>

[K1 Simplicial-Set]

(show-structure m24 6)  ==>

Dimension = 0 :

        Vertices :  (*)

Dimension = 1 :

Dimension = 2 :

Dimension = 3 :

Dimension = 4 :

        Simplex : M4

                Faces : (<AbSm 2-1-0 *> <AbSm 2-1-0 *> <AbSm 2-1-0 *>
                         <AbSm 2-1-0 *> <AbSm 2-1-0 *>)
Dimension = 5 :

        Simplex : N5

                Faces : (<AbSm - M4> <AbSm 3-2-1-0 *> <AbSm - M4> <AbSm 3-2-1-0 *>
                         <AbSm 3-2-1-0 *> <AbSm 3-2-1-0 *>)

Dimension = 6 :

NIL

(homology m24 0 5)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :
```

```
Homology in dimension 4 :

Component Z/2Z

---done---

(setf m24-chml-clss (chml-clss m24 4))  ==>

[K12 Cohomology-Class (degree 4)]

(setf m24-fibr (z2-whitehead m24 m24-chml-clss))  ==>

[K49 Fibration]

(setf m24-total (fibration-total m24-fibr))  ==>

[K55 Simplicial-Set]

(homology m24-total 0 6)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Homology in dimension 4 :

Homology in dimension 5 :

Component Z/2Z

---done---
```

We may even verify that, up to 9, the homotopy groups of the classifying group $K(\mathbb{Z}, 5)$, for instance, are null except π_5. This is a far-fetched method to verify this well known result, but it proves that the software is coherent.

```
(setf k5 (k-z 5))  ==>

[K49 Abelian-Simplicial-Group]

(setf k5-chml-clss (chml-clss k5 5))  ==>
```

```
[K576 Cohomology-Class (degree 5)]

(setf k5-fibr (z-whitehead k5 k5-chml-clss))  ==>

[K579 Fibration]

(setf k5-total (fibration-total k5-fibr))  ==>

[K580 Kan-Simplicial-Set]

(homology k5-total 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Homology in dimension 4 :

Homology in dimension 5 :

Homology in dimension 6 :

Homology in dimension 7 :

Homology in dimension 8 :

Homology in dimension 9 :

---done---
```

Let us show now the iteration of the process, to get for instance $\pi_5(S^3)$ and $\pi_6(S^3)$:

```
(setf s3 (sphere 3))  ==>

[K1 Simplicial-Set]

(setf s3-chml-clss (chml-clss s3 3))  ==>

[K12 Cohomology-Class (degree 3)]
```

```
(setf s3-fibration (z-whitehead s3 s3-chml-clss))  ==>

[K37 Fibration]

(setf s3-4 (fibration-total s3-fibration))  ==>

[K43 Simplicial-Set]

(homology s3-4 4)  ==>

Homology in dimension 4 :

Component Z/2Z

---done---

(setf s3-4-chml-clss (chml-clss s3-4 4))  ==>

[K253 Cohomology-Class (degree 4)]

(setf s3-4-fibration (z2-whitehead s3-4 s3-4-chml-clss))  ==>

[K292 Fibration]

(setf s3-5 (fibration-total s3-4-fibration))  ==>

[K298 Simplicial-Set]

(homology s3-5 5)  ==>

Homology in dimension 5 :

Component Z/2Z

---done---

(setf s3-5-chml-clss (chml-clss s3-5 5))  ==>

[K609 Cohomology-Class (degree 5)]

(setf s3-5-fibration (z2-whitehead s3-5 s3-5-chml-clss))  ==>

[K624 Fibration]

(setf s3-6 (fibration-total s3-5-fibration))  ==>

[K630 Simplicial-Set]
```

```
(homology s3-6 6)  ==>

Component Z/12Z
```

An interesting example is given by the real projective space $P^\infty\mathbb{R}/P^1\mathbb{R}$ which may be built in `Kenzo` by the function `r-proj-space`. We list its first homotopy groups.

```
(setf x (r-proj-space 2))  ==>

[K1 Simplicial-Set]

(homology x 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Component Z

Homology in dimension 3 :

Component Z/2Z

Homology in dimension 4 :

Homology in dimension 5 :

Component Z/2Z

Homology in dimension 6 :

Homology in dimension 7 :

Component Z/2Z

Homology in dimension 8 :

Homology in dimension 9 :

Component Z/2Z

---done---
```

Using the machinery described in this chapter, we find that the total space of the fibration has the same homology groups as S^3. It has been effectively proved that this space is homotopic to S^3.

```
(setf ch (chml-clss x 2))  ==>

[K12 Cohomology-Class (degree 2)]

(setf f2 (z-whitehead x ch))  ==>

[K25 Fibration]

(setf x3 (fibration-total f2))  ==>

[K31 Simplicial-Set]

(homology x3 0 10)  ==>

Homology in dimension 0 :

Component Z

Homology in dimension 1 :

Homology in dimension 2 :

Homology in dimension 3 :

Component Z

Homology in dimension 4 :

Homology in dimension 5 :

Homology in dimension 6 :

Homology in dimension 7 :

---done---
```

Lisp files concerned in this chapter

`whitehead.lisp`, `smith.lisp`.

Index

Index