2

Geometric programming

In this chapter the reader is introduced to the PLaSM programming by discussing first some examples from number and set theory. Other basic programming examples come from character coding and text processing, and from fundamental algorithms like sorting and merging. Then some elements of geometric programming are discussed, by introducing the reader to the generation of polyhedral approximation of curved manifolds by parametric maps. Also some primitive operators for aggregation of component objects into an assembly are discussed. The aim is both to introduce the reader to basic programming with a very simple but powerful functional language, and to discuss the generation of first non-trivial geometric models.

2.1 Basic programming

In this section some programming problems are discussed of very basic type, with the aim of showing that a PLaSM program is, in most cases, a sort of computational pipeline between the input data and the output results. We begin by computing, in various ways, standard functions on natural numbers like the factorial or the binomial function. Other basic examples are given from set theory. Then we discuss the implementation of simple programs for computing the Cartesian product of two sets and the Cartesian product of several sets, or for computing the power set of a given set, i.e. the collection of all its subsets, and so on. Programming problems of increasing complexity work with characters, strings and the contents of the ASCII character table. This direction is pursued until the implementation of a simple parser of strings. In this section we also discuss the important functions SORT and MERGE, for ordering and merging unordered and ordered sequences, respectively.

A first example The length of a sequence is computed by the predefined function LEN. As a programming exercise, we give here an equivalent function length, defined by the composition of two simpler programs (i.e. functions). In particular, we implement this program by first applying to the input sequence a function AA:(K:1), which transforms each sequence element into the number 1, thus transforming the input sequence into a sequence of ones with the same length. Such a sequence of

Geometric Programming for Computer-Aided Design Alberto Paoluzzi
 ISBN 0-471-89942-9

ones is then summed, generating the actual sequence length. The implementation of such an algorithmic approach, together with an example of computation, are given in Script 2.1.1. Remember that `AA` and `K` are predefined combinators; see Section 1.3.2.

Script 2.1.1

```
DEF length = + ∼ AA:(K:1)

length:<3,*,1,AA>
   ≡ (+ ∼ AA:(K:1)):<3,*,1,AA>
   ≡ +:(AA:(K:1):<3,*,1,AA>)
   ≡ +:<(K:1):3, (K:1):*, (K:1):1, (K:1):AA>
   ≡ +:<1,1,1,1> ≡ 4
```

2.1.1 Some operations on numbers

In several books on basic programming the user is rapidly introduced to writing simple programs to compute basic functions on natural numbers like the factorial function and the binomial function. Here we follow the same approach.

Arithmetic operations A very simple program, called `prog1`, may be given by applying the arithmetic operators of sum, product and division to the same pair of arguments, thus producing a sequence of results. For this purpose the combining form called *construction* is used. As we already know, the operation symbols may be inserted between braces (square parenthesis), and the resulting vector function, having the arithmetic operators as components, may so be applied to data. Notice that the expression `[+, *, /]` is a function.[1] Hence it is meaningful to apply it to some arguments. An example of such a computation is given in the following script.

Script 2.1.2

```
DEF prog1 = [+, *, /];

prog1:<10, 5>
   ≡ [+, *, /]:<10, 5>
   ≡ <+:<10, 5>, *:<10, 5>, /:<10, 5>>
   ≡ <15, 50, 2>
```

Infix expressions It may be interesting to compare the program given in Script 2.1.2 with the one shown in Script 2.1.3. It is worth noting that the "body" of `prog2` is an infix expression of the kind *expr*1 *expr*2 *expr*3, where the parentheses are actually optional and can be safely omitted. In `PLaSM` every infix expression is

[1] Remember the difference between the vector function `[f,g,h]`, whose components must be all functions, and the sequence `<a,b,c>`, that is a useful data structure, whose element may have any type. See Section 1.3.2.

evaluated as follows:

```
expr1 expr2 expr3 ≡ expr2 ∼ [expr1, expr3]
```

when the following conditions apply:

1. both `expr1` and `expr3` evaluate to a function value;
2. `expr2` evaluates to a higher-order binary function value, i.e. to some function which can be applied to two functions and returns a function.

Otherwise, the infix expression is evaluated in the following way:

```
expr1 expr2 expr3 ≡ expr2 : < expr1, expr3 >
```

Script 2.1.3

```
DEF prog2 = (+ * /);

prog2:<10, 5>
   ≡ (+ * /):<10, 5>
   ≡ (* ∼ [+, /]):<10, 5>
   ≡ * : <+:<10, 5>, /:<10, 5>>
   ≡ * : <15, 2>
   ≡ 30
```

Lambda-style and FL-style The lambda calculus, or λ-calculus, from the Greek letter λ, is a formal language introduced in the thirties by the logician Church to study the calculus with functions. All the functional languages can be considered equivalent to some (specialized) λ-calculus.

In the basic λ-calculus a function with parameter x and rule M (usually called *body*), to compute the function values, is denoted as $\lambda x.M$. The *application* of a function f to an argument a is written fa. E.g., the application of the function $\lambda x.x * x$, which returns its squared argument, to the number 8, would be written $(\lambda x.x * x)8$. Such an expression let 8 correspond to 8×8.

In λ-calculus a *term*, or λ-term, is either a *variable* x, or an *application* MN of the M function to the N term, or an *abstraction* $\lambda x.M$. The variable x is said *bound* in M. The variables unbound in M are said to be *free* in M. A *combinator* is a λ-term without free variables. The combinatoric logic, i.e. the use of combinators for the study of functions, started also in the thirties with the work of Schönfinkel and Curry.

We say "λ-style" to indicate the writing of definitions of functions (i.e. the ability to make abstractions) by giving a list of variables, and where the body expression makes explicit reference to such variables. In this book we also call "FL-style" the making of abstractions using only combinators, and therefore without variables. A discussion of pros and cons of both styles is beyond the scope of this book. In order to see a variable-free notation to denote mathematical functions, the reader is referred to Section 5.1.1.

Even and odd numbers Even numbers are, by definitions, those natural numbers (non negative integers) that are divisible by 2, i.e. such that the remainder of their division by 2 is zero. It follows that if a natural is multipied by 2, then the result is *even*. Also, by adding 1 to an even number, a *odd* number is generated.

In Script 2.1.4 two very simple programs `Even` and `Odd` are given, to generate the first n even and odd numbers, respectively. Notice that the first one is built by multiplying times 2 each element of the integer sequence from 0 to $n-1$. Analogously, the second one works by adding one to each element of the sequence generated by the expression `Even:n`.

Script 2.1.4 (Even and odd numbers)

```
DEF Even (n::IsInt) = AA:(C:*:2):(0..(n - 1));
DEF Odd  (n::IsInt) = AA:(C:+:1):(Even:n);

Even:10 ≡ < 0 , 2 , 4 , 6 , 8 , 10 , 12 , 14 , 16 , 18 >
Odd :10 ≡ < 1 , 3 , 5 , 7 , 9 , 11 , 13 , 15 , 17 , 19 >
```

The reader should notice that the *curried* (see Section 1.4.3) function `C:*:2` doubles the value it is applied to, as well as `C:+:1` adds one to its argument.

Factorial function The factorial of n is defined as

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n,$$

with n non-negative integer. In many programming languages, both functional and imperative, such a function is normally computed using either an iterative or a recursive approach. In `PLaSM` (i.e in `FL`) it is more "natural" to compute the function by directly implementing the above definition. The result is hence generated by multiplying each term of the sequence from 1 to n, as shown by Script 2.1.8. Notice that the application `INTSTO:n` generates the sequence of first n positive integers, and that the `*` operator, when applied to a number sequence, returns the product of sequence elements.

Script 2.1.5 (Factorial of a positive number)

```
DEF fact = * ~ INTSTO

fact:n ≡ (* ~ INTSTO):n
       ≡ *:(INTSTO:n)
       ≡ *:<1,2,...,n>
```

The previous definition computes only the factorial of positive numbers. Actually, such a function should also return a value for zero value of the argument, with $0! = 1$ by definition. The implementation of such extended definition is given in Script 2.1.6, by using both a `FL` style and a λ-style of programming, discussed in the above paragraph. Notice that the curried expression (Section 1.4.3)

```
C:EQ:0
```

returns a function such that

```
C:EQ:0:0 ≡ True
C:EQ:0:x ≡ False                    x ≠ 0
```

Notice also that, when using a λ-style of programming, the actual arguments are checked for validity according to the type predicate specified in the list of formal parameters of the function. In this case the type predicate is given as `AND ~ [IsInt, GE:0]`. Such a data typing is called *dynamic typing*, since the type predicate is applied to the input data only at run time. Two equivalent definitions of the `fact` function are given below, using both a proper `FL` style and a λ-style.

Script 2.1.6 (Factorial of a non-negative number)

```
DEF fact = IF:< C:EQ:0, K:1, * ~ INTSTO >;

DEF fact (n::(AND ~ [IsInt, GE:0])) = IF:< C:EQ:0, K:1, * ~ INTSTO >:n;

AA:fact:< 0,1,2,3 > ≡ < 1,1,2,6 >
```

A recursive implementation of the factorial function is given in the Script 2.1.7, according to the definition

$$n! = \begin{array}{ll} 1, & n = 0 \\ n \times (n-1)! & n > 0 \end{array}$$

and using both a proper `FL` style and a λ programming style. In this case let us notice that the "`*`" and "`-`" symbols denote arithmetic operations between functions. The `pred` function (*predecessor*) is defined as `pred` : $Int \rightarrow Int$ such that $n \mapsto n-1$.

Script 2.1.7 (Recursive factorial)

```
DEF pred = ID - K:1;

DEF fact = IF:< C:EQ:0, K:1, ID * fact ~ pred >;

DEF fact (n::(AND ~ [IsInt, GE:0])) =
   IF:< C:EQ:0, K:1, ID * fact ~ pred >:n;
```

To understand the behavior of Script 2.1.7 we should remember that the `IF` combinator must be applied to a triplet of functions. So, when the integer input to the `fact` function is positive, the function `ID * fact ~ pred` is applied to it. Therefore we have, for example:

```
fact:4
   ≡ (IF:< C:EQ:0, K:1, ID * fact ~ pred >): 4
   ≡ (ID * fact ~ pred):4
   ≡ (* ~ [ID, fact ~ pred]): 4
   ≡ *: < 4, fact:(pred:4) >
```

Table 2.1 The Pascal triangle of binomial coefficients $\binom{n}{k}$

n \ k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

```
   ≡ 4 * fact:3
```

and so on, recursively, until we have:

```
fact:4
   ≡ 4 * fact:3
   ≡ 4 * 3 * fact:2
   ≡ 4 * 3 * 2 * fact:1
   ≡ 4 * 3 * 2 * 1 * fact:0
   ≡ 4 * 3 * 2 * 1 * 1 ≡ 24
```

Binomial function The *binomial* function goes from pairs of natural (i.e. non-negative integer) numbers to natural numbers. The binomial number $\binom{n}{k}$ denotes the number of different ways to choose a subset of k elements from a set of n elements. As is well known, the first binomial numbers may be ordered within a lower-triangular matrix, called *Pascal triangle* in English,[2] according to Table 2.1.

A standard formula for the computation of binomial numbers is

$$\binom{n}{k} = \frac{n!}{k!\ (n-k)!}.$$

A direct implementation of this formula is given in Script 2.1.8, where the values of the binomial function, called `choose` in the following, are also shown for some values of the input pair.

Script 2.1.8 (Binomial by factorial)

```
DEF choose (n,k::IsInt) = Fact:n / (Fact:k * Fact:(n-k));

AA:choose:( 4 DISTL (0..4) ) ≡ < 1 , 4 , 6 , 4 , 1 >
```

A recursive function to compute the binomial numbers according to the well-known formula

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, \qquad 0 < k < n \tag{2.1}$$

---

[2] Actually, the triangle of binomial numbers was discovered by the Italian matematician Nicolò Tartaglia (1499–1557), and not by Blaise Pascal (1623–1662). Nicolò Tartaglia also discovered the resolution formula for the algebraic equations of degree 3.

depicted by boxes in Table 2.1 is given in Script 2.1.9. Notice, from Table 2.1, that such formula may only work when $0 < k < n$. Conversely we have, for the binomial numbers on the border of the triangle:

$$\binom{n}{k} = 1, \qquad k = 0, k = n. \tag{2.2}$$

The function `Choose` in Script 2.1.9 is written by using a proper `FL` style, i.e. without formal parameters. According to the `IF` semantics, there are three functions within its argument sequence. The first one is the predicate used to verify if the computation must proceed according to either the basis case of equation (2.2) or the inductive case of equation (2.1). As usual, the arithmetic operators between function-valued expressions denote operations which return a function as value. The predefined selectors `S1` and `S2` are used to extract the first and the second element, respectively, from the input pair $< n, k >$.

Script 2.1.9 (Binomial by recursion)

```
DEF Choose = IF:<
   OR ~ [C:EQ:0 ~ S2, EQ],
   K:1,
   Choose ~ [S1 - K:1, S2 - K:1] + Choose ~ [S1 - K:1, S2]
>;
```

The previous implementation, which makes use of the equations 2.1 and 2.2, is highly inefficient. In fact, the computation of a single binomial number requires the computation of very many such numbers and, even worse, computes most of them again and again many times.

A much better solution, given in Script 2.3, is obtained by using the following equation:

$$\binom{n}{k} = \binom{n-1}{k-1} \frac{n}{k}, \qquad 0 < k < n. \tag{2.3}$$

In this case, the function `Choose` makes only one call to itself in the recursive case instead than two. An approach of this kind, where a recursive function calls itself at most one time, is called *linear recursion*. It is easy to see, looking at Table 2.1, that to compute $\binom{n}{k}$ in the worst case needs a total number of function calls which is of the same magnitude of n.

Computer scientists would say that this algorithm has linear complexity, denoted as $O(n)$ (to read as "Order of n"), in the worst case. Conversely, the algorithm implemented by Script 2.1.9 has worst case exponential complexity $O(2^n)$.

Notice that the implementation in Script 2.1.10 uses the infix notation for the function to be applied in the recursive case. The meaning of infix functional expressions like

```
expr * /
```

where `expr` evaluates to some functional value, is discussed in Section 2.1. Notice also the meaning of the function `AA:(C:+:-1)`; when applied to a number sequence, it decrements by one each element.

Script 2.1.10 (Binomial by linear recursion)

```
DEF Choose = IF:<
   OR ~ [C:EQ:0 ~ S2, EQ], K:1, Choose ~ AA:(C:+:-1) * /
>;
```

Example 2.1.1 (Pascal triangle)
A simple script, to generate the Pascal triangle of any size, is step-wise discussed in this example. First, a matrix of index pairs, with both row and column indices ranging from 0 to n, is generated by the function `pairs` when applied to the natural number n. In Script 2.1.11 two further predicates `IsNat` and `IsNum` are given, to test if the input is a natural number (say, it is an integer `AND` is greater or equal to zero) or simply a number (say, it is an integer `OR` a real).

Script 2.1.11 (Pairs of matrix indices)

```
DEF IsNat = AND ~ [IsInt, GE:0];
DEF IsNum = OR ~ [IsInt, IsReal];

DEF pairs (n::IsNat) =
   (AA:DISTL ~ DISTR ~ [ID,ID] ~ FROMTO ~ [K:0, ID]):n ;

pairs:5 ≡ <
   <<0, 0>, <0, 1>, <0, 2>, <0, 3>, <0, 4>, <0, 5>>,
   <<1, 0>, <1, 1>, <1, 2>, <1, 3>, <1, 4>, <1, 5>>,
   <<2, 0>, <2, 1>, <2, 2>, <2, 3>, <2, 4>, <2, 5>>,
   <<3, 0>, <3, 1>, <3, 2>, <3, 3>, <3, 4>, <3, 5>>,
   <<4, 0>, <4, 1>, <4, 2>, <4, 3>, <4, 4>, <4, 5>>,
   <<5, 0>, <5, 1>, <5, 2>, <5, 3>, <5, 4>, <5, 5>>>
```

Then, a matrix of binomial coefficients is generated, by applying the `choose` function to each pair in the lower-triangular submatrix of the matrix generated by `pairs:`n, and by writing 0 in the upper triangular submatrix. In order to penetrate a *double* sequence (a matrix is a sequence of rows, which are also sequences) to reach the pair elements to whom apply either the `choose` or the `K:0` function, the `AA ~ AA` operator is used.

Script 2.1.12 (Pascal's matrix)

```
DEF IsLE (a,b::IsNum) = LE:a:b;
DEF Pascal = (AA ~ AA):(IF:< IsLE, choose, K:0 >);

(Pascal ~ pairs):5 ≡ <
   <1, 0, 0, 0, 0, 0>,
   <1, 1, 0, 0, 0, 0>,
   <1, 2, 1, 0, 0, 0>,
   <1, 3, 3, 1, 0, 0>,
   <1, 4, 6, 4, 1, 0>,
   <1, 5, 10, 10, 5, 1>>
```

Finally, the `filter` function introduced in Script 1.3.1 is applied in Script 2.1.13 to the matrix generated by the function `Pascal ~ pairs`, giving as output only its non-zero elements. The resulting output is the *Pascal's triangle* of any size, organized by rows. Such computation works by applying the function `filter:(C:EQ:0)` is applied to each matrix row.

Script 2.1.13 (Pascal triangle)

```
DEF PascalTriangle (n::IsIntPos) =
   (AA:(filter:(C:EQ:0)) ~ Pascal ~ pairs):n;

PascalTriangle:5 ≡ <
   <1>,
   <1, 1>,
   <1, 2, 1>,
   <1, 3, 3, 1>,
   <1, 4, 6, 4, 1>,
   <1, 5, 10, 10, 5, 1>>
```

It is easy to see, by using the `Choose` implementation given in Script 2.1.10, that the complexity of the computation of the Pascal triangle with n rows is $O(n^3)$. In fact, the number of terms to be computed is $O(n^2)$, and each term is computed with time and space complexity $O(n)$.

Pascal triangle in optimal time The computation of the Pascal triangle with $n+1$ rows can be computed in $O(n^2)$ time by using a proper `FL` style, say, without neither iteration nor recursion, but using a computational pipelining of data across a series of compositions of the same function. A time $O(n^2)$ is optimal, because it is of the same order of magnitude of the output results.

The terse implementation given in Script 2.1.14 relies on the fact that a new triangle row can be obtained by adding termwise two instances of the previous row, with a 0 element appended at the left and at the right, respectively. As a matter of fact, we have:

$$\begin{array}{cccccccc} (0, & 1, & 4, & 6, & 4, & 1) & + \\ (1, & 4, & 6, & 4, & 1, & 0) & = \\ (1, & 5, & 10, & 10, & 5, & 1) & \end{array}$$

The program's building block, to be composed n times with itself, is the function `addrow`, whose effect is to add a $(i+1)$ row to its input sequence with i rows. The computation of the new row according with the above scheme is performed by the `row` function. The `vectsum` function, used for termwise addition of two sequences of numbers, is given in Script 2.1.19. Notice that `C:AL:0` is a *curried* version of the `AL` combinator, that appends a `0` element on the left of its input sequence.

Script 2.1.14 (Pascal triangle (2))

```
DEF pascalTriangle (n::IsIntPos) = (REVERSE ~ COMP:(#:n:addrow)):<<1>>
WHERE
   addrow = AL ~ [row ~ S1, ID],
   row = vectsum ~ [ID, REVERSE] ~ C:AL:0
END;

pascalTriangle:20
```

2.1.2 Set operations

In this section we introduce some basic operations on sets with elements of arbitrary type, including the generation of the Cartesian product set of two or more sets and the power set of a given set. Let notice that the PLaSM representation of any set is the *sequence* of the set elements.

Cartesian product of two sets Given two sets $A = \{a_i\}$ and $B = \{b_j\}$, their *Cartesian product* is defined as the set $A \times B = \{(a_i, b_j) |\ a_i \in A, b_j \in B\}$ of all ordered pairs of elements from A and B.

A PLaSM function, denoted as cart2, which generates the Cartesian product of any two sets, i.e. of any two sequences, is given in Script 2.1.15. Its body is defined as the composition of three functions DISTR, AA:DISTL and CAT, to be orderly applied to the pair of set arguments, as shown in the example of computation given in the script. In particular, the function cart2 is applied to the pair <1, 2, 3> and <'a', 'b'>, generating the sequence of pairs <<1, 'a'>, <1, 'b'>, <2, 'a'>, <2, 'b'>, <3, 'a'>, <3, 'b'>>.

Script 2.1.15 (Binary Cartesian product)

```
DEF cart2 = CAT ~ AA:DISTL ~ DISTR

cart2:<<1, 2, 3>, <'a', 'b'>>
    ≡ <<1, 'a'>, <1, 'b'>, <2, 'a'>,<2, 'b'>, <3, 'a'>, <3, 'b'>>
```

Example 2.1.2 (Debugging by PRINT)
For purpose of debugging, the language offers an useful PRINT function, which is equivalent to ID, with the additional *side effect* of printing in the listener window the value of its input. Hence a new implementation of the cart2 function is given in Script 2.1.16, which allows the intermediate results of the computation to be known.

Cartesian product of several sets The Cartesian product of n sets is defined as the set of ordered n-tuples, obtained by choosing an element from each set in every possible way.

This operation is implemented very simply in Script 2.1.17 by using the proper FL programming style. In particular, we use here the primitive combining form TREE, which allows for recursive application of a *binary* function f to a sequence of n arguments. Such a primitive form is defined as follows:

Script 2.1.16

```
DEF cart2 = CAT ~ PRINT ~ AA:DISTL ~ PRINT ~ DISTR

cart2:<<1, 2, 3>, <'a', 'b'>>
   ≡ (CAT ~ PRINT ~ AA:DISTL):
      <<1, <'a', 'b'>>, <2, <'a', 'b'>>, <3, <'a', 'b'>>>
   ≡ CAT:
      <<<1, 'a'>, <1, 'b'>>, <<2, 'a'>, <2, 'b'>>, <<3, 'a'>, <3, 'b'>>>
   ≡ <<1, 'a'>, <1, 'b'>, <2, 'a'>, <2, 'b'>, <3, 'a'>, <3, 'b'>>
```

$$\texttt{TREE:f:<}x_1, x_2, \ldots, x_n\texttt{>} \equiv \texttt{f:< TREE:f:<}x_1, \ldots, x_k\texttt{>, TREE:f:<}x_{k+1}, \ldots, x_n\texttt{> >} \qquad \text{where } k = \lceil n/2 \rceil,$$

$$\texttt{TREE:f:<}x_1\texttt{>} \equiv x_1\,.$$

We already know two similar combining forms, named `INSR` and `INSL`, respectively. They execute the same reduction by applying the argument function from either the right or the left of the sequence argument, as seen in Section 1.3.2. Conversely, the `TREE` combining form executes a similar recursive reduction, but subdividing the argument sequence in two subsets of nearly equal length. Notice that the `cart2` function is the one defined in Script 2.1.15.

Script 2.1.17 (n-ary Cartesian product)

```
DEF cart = TREE:f2 ~ f1
WHERE
   f2 = AA:CAT ~ cart2,
   f1 = AA:(AA:[ID])
END;

cart:<<1,2>, <3>, <4,5>>
   ≡ <<1,3,4>, <1,3,5>, <2,3,4>, <2,3,5>>
```

Power set An implementation of the `power_set` function, which computes the *power set* of any given set, is discussed here. It is worth remembering that the power set of a set $A = \{a_i\}$ is defined as the collection of all subsets of A, including the empty set and A itself. Notice that if A has n elements, then the power set of A has 2^n elements. For this reason, the notation 2^A is often used to denote this set. Sometimes the notation $\mathcal{P}(A)$ is instead used. The generating function `power_set` is given in Script 2.1.18.

The used algorithm can be summarized as follows:

1. the n input elements a_i are transformed into n pairs `<<`a_i`>, <>>`;
2. the Cartesian product of such pairs is computed, producing 2^n n-tuples;
3. all such n-tuples are catenated, to eliminate the empty elements;

As a further example we can see that:

```
powerSet:< 1, 2, 3, 4 >
   ≡ < <1,2,3,4>, <1,2,3>, <1,2,4>, <1,2>, <1,3,4>, <1,3>, <1,4>, <1>,
       <2,3,4>, <2,3>, <2,4>, <2>, <3,4>, <3>, <4>, <> >
```

Script 2.1.18

```
DEF power_set = AA:CAT ~ cart ~ AA:([[ID], K:<>])
power_set:< 1, ID, 4 >
  ≡ (AA:CAT ~ cart ~ AA:([[ID], K:<>])):<1,ID,4>
  ≡ AA:CAT:(cart:(AA:([[ID], K:<>]):<1,ID,4>))
  ≡ AA:CAT:(cart:<[[ID], K:<>]:1,
                  [[ID], K:<>]:ID,
                  [[ID], K:<>]):4>
  ≡ AA:CAT:(cart:<<<1>,<>>, <<ID>,<>>, <<4>,<>>>)
  ≡ AA:CAT:<<<1>, <ID>, <4>>,
            <<1>, <ID>, <>>,
            <<1>, <>, <4>>,
            <<1>, <>, <>>,
            <<>, <ID>, <4>>,
            <<>, <ID>, <>>,
            <<>, <>, <4>>,
            <<>, <>, <>>>
  ≡ < <1, ID, 4>, <1, ID>, <1, 4>, <1>, <ID, 4>, <ID>, <4>, <> >
```

2.1.3 Vector and matrix operations

Here we introduce the basic vector operations of sum of two or more vectors of arbitrary dimensions, the product of a scalar times a vector and the sum of compatible matrices. A vector is represented as the sequence of its components, and a matrix as the sequence of its row vectors.

Addition element-wise of two sequences A vector $\boldsymbol{v} \in \mathbb{R}^d$ is represented in PLaSM as a sequence of d numbers. The PLaSM program to add two vectors, i.e. to add component-wise two sequences of the same length, can be defined as follows, by first transposing the input sequences, and then by adding each generated pair.

Script 2.1.19

```
DEF vectSum = AA:+ ~ TRANS;

vectSum:<<1, 2, 3, 4>, <11, 12, 13, 14>>
   ≡ (AA:+ ~ TRANS):<<1, 2, 3, 4>, <11, 12, 13, 14>>
   ≡ (AA:+:(TRANS:<<1, 2, 3, 4>, <11, 12, 13, 14>>)
   ≡ AA:+:<<1, 11>, <2, 12>, <3, 13>, <4, 14>>
   ≡ <+:<1, 11>, +:<2, 12>, +:<3, 13>, +:<4, 14>>
   ≡ <12, 14, 16, 18>
```

It might be interesting to notice, as a further example of the amazing power of the combinatorial logic underlying FL, that the same vectSum function may also be used to add *any set* of vectors in the same vector space, i.e. with the same number of components. E.g.

```
vectSum:<<1,2,3,4>, <11,12,13,14>, <21,22,23,24>, <31,32,33,34>>
   ≡ <64, 68, 72, 76>
```

Product of a scalar times a vector The product of a scalar times a vector, represented as a sequence of numbers, returns a sequence of the same length where each element is the product of the scalar times an element of the original sequence. The realization of a PLaSM function for this operation is very easy. First, the scalar is left-distributed on the sequence; then, the product operation is applied to all the pairs so generated.

Script 2.1.20

```
DEF scalarVectProd = AA:* ~ DISTL

scalarVectProd:<2, <1, 2, 3, 4>>
   ≡ (AA:* ~ DISTL):<2, <1, 2, 3, 4>>
   ≡ AA:*:( DISTL:<2, <1, 2, 3, 4>> )
   ≡ AA:* :<<2, 1>, <2, 2>, <2, 3>, <2, 4>>
   ≡ <*:<2, 1>, *:<2, 2>, *:<2, 3>, *:<2, 4>>
   ≡ <2, 4, 6, 8>
```

The scalarVectProd operator given in Script 2.1.20 may be only used with an ordered pair <*scalar*, *vector*>. A useful generalization of such operation, which is also able to manage the converse case <*vector*, *scalar*>, is given in Script 2.1.21.

Script 2.1.21

```
DEF scalarVectProd = AA:* ~ IF:< IsReal ~ S1, DISTL, DISTR >

scalarVectProd:<2, <1, 2, 3, 4>>
   ≡ scalarVectProd:<<1, 2, 3, 4>, 2>
   ≡ <2, 4, 6, 8>
```

Addition of compatible matrices Two matrices of numbers are said to be *compatible* when they have the same number of rows and column, i.e. when they belong to the same space $\mathcal{M}_n^m$. The *sum* of two compatible matrices $A = (a_{ij})$ and $B = (b_{ij})$ is the matrix $A + B = (a_{ij} + b_{ij})$ obtained by component-wise addition of matrix elements.

In PLaSM a matrix is represented as a *sequence of rows*, in turn represented as sequences of elements. Notice that all the rows *must* have the same length, which equates the number of matrix columns. The type of the matrix is that of its elements.

A function matSum to add two compatible matrices is given in Script 2.1.22. To add two compatible matrices we must start by producing a sequence of pairs of corresponding rows, then transpose all such (row) pairs, thus producing sequences of element pairs, and finally applying the sum operation to all such pairs. This last step is performed by the operation (AA ~ AA):+, since two levels of parentheses must be penetrated.

Script 2.1.22

```
DEF matSum = (AA ∼ AA):+ ∼ AA:TRANS ∼ TRANS;

matSum:<<<1,2,3>,<4,5,6>>,
        <<11,12,13>,<14,15,16>>>
   ≡ <<12,14,16>,<18,20,22>>
```

2.1.4 String and character operations

In PLaSM there are some predefined *string operations*, which allow for manipulation of strings as sequences of characters. In particular:

1. ISSTRING predicate tests if its input is a string;
2. ISCHAR predicate tests if its input is a string of unit length, i.e. constituted by a single character;
3. STRING function maps a sequence of characters into a string;
4. CHARSEQ function maps a string into a sequence of characters;
5. ORD function maps an ASCII character into its ordinal value, i.e. into the integer associated to the character in the ASCII table;
6. CHAR function maps an integer from the interval $[0, 255]$ into the corresponding ASCII character;
7. comparison predicates LT (less than), LE (less or equal), EQ (equal), GE (greater or equal) and GT (greater than) are used to compare characters and strings in *lexicographic order*, as discussed in a following paragraph, as well as to compare numbers.

Example 2.1.3 (Printable ASCII)
It is very easy to transform any string into the sequence of ordinal values of its characters, and vice versa. The following Script 2.1.23 shows the PLaSM generation of the subset of printable ASCII characters, with ordinal value in the interval $[32, 126]$. Let us notice that the expression `(STRING ∼ AA:CHAR):(32..126)` maps the sequence of numbers into a unique string.

Example 2.1.4 (Natural numbers as binary strings)
An useful function nat2string is given in Script 2.1.24 to transform a natural number to a binary string, i.e. to a string containing only the two symbols '0' and '1', corresponding to the binary representation of the number. The binary string is generated by the standard algorithm of successive divisions by 2. In particular, the recursive function dec2binSeq generates a nested sequence of binary digits. For example:

```
dec2binSeq:10 ≡ <<<1, 0>, 1>, 0>
```

The output of dec2binSeq is then modified by the recursive function flatten:

```
(flatten ∼ dec2binSeq):10 ≡ <1, 0, 1, 0>
```

Such number sequence is transformed into the sequence of ordinal values of characters '0' and '1' by adding 48 to each element via the curried + operator:

Script 2.1.23 (ASCII **table**)

```
AA:[ID, CHAR]:(32..126) ≡ <
< 32, ′␣′>, < 33, ′!′>, < 34, ′"′>, < 35, ′#′>, < 36, ′/′>,
< 37, ′%′>, < 38, ′&′>, < 39, ′’′>, < 40, ′(′>, < 41, ′)′>,
< 42, ′*′>, < 43, ′+′>, < 44, ′,′>, < 45, ′-′>, < 46, ′.′>,
< 47, ′/′>, < 48, ′0′>, < 49, ′1′>, < 50, ′2′>, < 51, ′3′>,
< 52, ′4′>, < 53, ′5′>, < 54, ′6′>, < 55, ′7′>, < 56, ′8′>,
< 57, ′9′>, < 58, ′:′>, < 59, ′;′>, < 60, ′<′>, < 61, ′=′>,
< 62, ′>′>, < 63, ′?′>, < 64, ′@′>, < 65, ′A′>, < 66, ′B′>,
< 67, ′C′>, < 68, ′D′>, < 69, ′E′>, < 70, ′F′>, < 71, ′G′>,
< 72, ′H′>, < 73, ′I′>, < 74, ′J′>, < 75, ′K′>, < 76, ′L′>,
< 77, ′M′>, < 78, ′N′>, < 79, ′O′>, < 80, ′P′>, < 81, ′Q′>,
< 82, ′R′>, < 83, ′S′>, < 84, ′T′>, < 85, ′U′>, < 86, ′V′>,
< 87, ′W′>, < 88, ′X′>, < 89, ′Y′>, < 90, ′Z′>, < 91, ′[′>,
< 92, ′\′>, < 93, ′]′>, < 94, ′^′>, < 95, ′_′>, < 96, ′‘′>,
< 97, ′a′>, < 98, ′b′>, < 99, ′c′>, <100, ′d′>, <101, ′e′>,
<102, ′f′>, <103, ′g′>, <104, ′h′>, <105, ′i′>, <106, ′j′>,
<107, ′k′>, <108, ′l′>, <109, ′m′>, <110, ′n′>, <111, ′o′>,
<112, ′p′>, <113, ′q′>, <114, ′r′>, <115, ′s′>, <116, ′t′>,
<117, ′u′>, <118, ′v′>, <119, ′w′>, <120, ′x′>, <121, ′y′>,
<122, ′z′>, <123, ′{′>, <124, ′|′>, <125, ′}′>, <126, ′~′>>
```

```
(AA:(C:+:48) ∼ flatten ∼ dec2binSeq):10 ≡ <49, 48, 49, 48>
```

and this one is mapped to a sequence of characters:

```
(AA:(CHAR ∼ C:+:48) ∼ flatten ∼ dec2binSeq):10 ≡ <′1′, ′0′, ′1′, ′0′>
```

The last step is quite obvious:

```
(STRING ∼ AA:(CHAR ∼ C:+:48) ∼ flatten ∼ dec2binSeq):10 ≡ ′1010′
```

Notice that the `MOD` function given in Script 2.1.24 must be applied to a pair of integers and returns the remainder of their division. E.g. `MOD:<7,2>` ≡ `1`. The `IsNat` predicate, to test if the input of the `nat2string` function is a natural number, is defined in Script 2.1.11.

Example 2.1.5 (String to tokens)
In this example a second order function `StringTokens` is given, which is first applied to a sequence of separators and then to a string. It returns the sequence of so-called *tokens* contained in the input string, i.e. the sequence of substrings separated by the elements of the given set of separators. The example is aimed at discussing a non-trivial exercise of non-geometric `PLaSM` programming.

In order to reach our goal we start by defining a `IN` predicate used to test the *set-membership* property for a generic set and a generic element. Clearly this predicate will return `TRUE` if the element is contained in the set and will return `FALSE` otherwise.

The algorithm used in implementing the `StringTokens` function is very simple (and quite inefficient). In particular, the value of type `STRING` contained in the formal parameter `input` is preliminarily transformed into a `CHAR` sequence. Such a sequence is then transformed into a sequence of pairs, where each character is coupled with its

Script 2.1.24 (Number to binary string)

```
DEF MOD = - ~ [S1, S2 * FLOOR ~ /];

DEF flatten =
   IF:< IsSeq, IF:< IsSeq ~ S1, AR, ID > ~ [flatten ~ FIRST, LAST], ID >;

DEF dec2binSeq =
   IF:< GE:2, [dec2binSeq ~ FLOOR ~ /, MOD] ~ [ID, K:2], ID >;

DEF nat2string (n::IsNat) = ( STRING
   ~ AA:(CHAR ~ C:+:48)
   ~ IF:<IsSeq, ID, [ID]>
   ~ flatten
   ~ dec2binSeq ):n;

AA:nat2string:< 0,1,10,14,255,256 >
   ≡ < '0' , '1' , '1010' , '1110' , '11111111' , '100000000' >
```

position in the `input` string. Then all the positions of separators constituted by a single character are selected. Such positions are used to build the pairs of indices associated with each sub-string. The pairs then properly generate the selector functions used to extract the various substrings.

Finally, an internal `filter` function is applied to eliminate the empty sequences and the substrings possibly contained in the separator set `separators`, which may contain both single characters and substrings. It is worth noting that such a formal variable is defined of type `IsSeqOf:IsString`.

Notice also that the `TT` predefined symbol denotes a predicate which returns `TRUE` for every input, and is used to leave *undefined* the type of a formal parameter. It is actually equivalent to the function `K:TRUE`.

Example 2.1.6 (String parsing)
Some examples of application of the token extractor `StringTokens` are given in Script 2.1.26. In particular, in the first `PLaSM` expression, the only defined separator is the *blank* character '␣'; in the second expression three separators are given, corresponding to the characters '␣', ',' and to the word 'and'.

Example 2.1.7 (Parsing of source code)
A further and more interesting example with the `StringTokens` operator is shown in Script 2.1.27, where tokens are extracted from a fragment of source code, given as the input string of the `StringTokens` function.

In this case it is useful to insert in the set of separators both the language keywords, and some single and double separators. Notice also that the ASCII character denoted as CR (*Carriage Return*) is inserted in such a set, by using its generating expression `CHAR:13`. The second part of Script 2.1.27 shows the output generated by applying the `StringTokens` operation to the fragment of source code where the `IN` function is defined.

Script 2.1.25 (Tokens extraction from a string)

```
DEF IN (set::IsSeq)(element::TT) = (OR ~ AA:EQ ~ DISTR):< set, element >;

DEF StringTokens (separators::IsSeqOf:IsString) (input::IsString) =
   ( filter ~ AA:(IF:< NOT ~ ISNULL, [STRING], ID >) ~ APPLY
   ~ [ CONS
     ~ AA:CONS
     ~ (AA ~ AA):SEL
     ~ AA:FROMTO
     ~ TRANS
     ~ start_stop_pos, ID
     ] ~ CHARSEQ ):input
WHERE
   separator_pos = CAT ~ AA:(IF:< IN:separators ~ S1, [S2], K:<> >)
     ~ TRANS ~ [ID, INTSTO ~ LEN],
   start_stop_pos = [
     AA:(ID + K:1) ~ AL ~ [K:0, separator_pos],
     AA:(ID - K:1) ~ AR ~ [separator_pos, LEN + K:1]
   ],
   filter = CAT ~ AA:(IF:< IN:separators, K:<>, [ID] >) ~ CAT
END;
```

Script 2.1.26 (String parsing)

```
StringTokens:<'␣'>:'hello PLaSM world' ≡ <'hello', 'PLaSM', 'world'>

StringTokens:< '␣', 'and', ',' >:'Fred, Wilma, Barney and Lucy' ≡
   < 'Fred', 'Wilma', 'Barney', 'Lucy' >
```

2.1.5 Other examples

Some further examples of basic data structures and algorithms are given here, including fundamental list operations like sorting and merging, the construction and quick search of associative lists, and the set operations of union, intersection and difference.

List operation The function `butFirstButLast` given in Script 2.1.28 may only operate on sequences. In particular such a program returns the input sequence but without the first and last element. The implementation makes use of the primitive operations `TAIL` and `REVERSE`. The first one returns the input sequence without the

Script 2.1.27 (Parsing)

```
DEF separators = < 'DEF', 'WHERE', 'END',
   '(', ')', '::', ':', '␣', '=', ' ~ ', ',', ';', '<', '>', CHAR:13 >;

StringTokens:separators: 'DEF IN (set::IsSeq) (element::K:TRUE) =
   (OR ~ AA:EQ ~ DISTR):< set, element >;'

≡ < 'IN', 'set', 'IsSeq', 'element', 'K', 'TRUE', 'OR',
    'AA', 'EQ', 'DISTR', 'set', 'element' >
```

first element; the second one reverses the order of its input.

Script 2.1.28

```
DEF butFirstButLast = REVERSE ∼ TAIL ∼ REVERSE ∼ TAIL

butFirstButLast:<10, 30, 50, 70> = <30, 50>
butFirstButLast:(15..20) = <16, 17, 18, 19>
butFirstButLast:<ID, *, AA, K, (ID * ID)> = <*, AA, K>
```

Merge of ordered sequences A primitive FL combining form is used to merge two ordered sequences x = <x_1, ... ,x_n> and y = <y_1, ... ,y_m>. As usual, the combining form MERGE is defined in a very general way, i.e. with respect to every binary predicate f:

MERGE:f:<x,y> =
   x *or* y *if either* y = <> *or* x = <>
   AL:<x_1, MERGE:f:<TAIL:x,y>> *if* f:<x_1,y_1> ≡ *true*
   AL:<y_1, MERGE:f:<x,TAIL:y>> *if* f:<x_1,y_1> ≡ *false*

The MERGE operator was conceived to work with binary predicates of the kind op:<x_i,y_j>, which return a truth value when applied to the pair of their arguments. Actually, the current implementation of PLaSM uses a curried version of the comparison predicates GE, GT, LE and LT, making impossible their direct use with the MERGE operator.

In order to use the MERGE combining form with the comparison operators it is necessary to transform expressions like GT:a:b into expressions like IsGT:<a,b>. Hence a complete set of un-curried comparison operators is also given.

Such a transformation can be executed by some very simple function like the one given in Script 2.1.29, so that we can finally write, to merge two or more ordered sequences:

Script 2.1.29 (Merge of number sequences)

```
DEF IsGT (a,b::TT) = GT:a:b;
DEF IsLT (a,b::TT) = LT:a:b;
DEF IsGE (a,b::TT) = GE:a:b;
DEF IsLE (a,b::TT) = LE:a:b;

MERGE:IsGT: <<1,3,4,5,5.15,7,9,10>,<2,4,6,8>>
   ≡ <1, 2, 3, 4, 4, 5, 5.15, 6, 7, 8, 9, 10>

(INSL:(MERGE:IsGT)): <<1,3,4,5,5.15,7,9,10>,<2,4,6,8>,<-4,8.2,20,61>>
   ≡ <-4, 1, 2, 3, 4, 4, 5, 5.15, 6, 7, 8, 8.2, 9, 10, 20, 61>
```

Sort of unordered sequences By using few FL combining forms and the predefined operator MERGE it is possible to give an amazingly simple implementation of the well-

known *merge-sort* algorithm (see, e.g. [AU92]), which executes the sorting of unordered sequences by recursively merging ordered sub-sequences.

The function `SORT`, given in Script 2.1.30, depends also on a `predicate` used to compare a pair of elements to be ordered. By using either the predicate `IsGT` or the predicate `IsLT`, both given in Script 2.1.29, the output sequence will contain an increasing or decreasing ordering, respectively.

Script 2.1.30 (Sort of numbers)

```
DEF SORT (predicate::IsFun) = TREE:(MERGE:predicate) ~ AA:[ID];

SORT:IsGT:<8, 2, 4, 2, 3, 11, -5> ≡ <-5, 2, 2, 3, 4, 8, 11>
SORT:IsLT:<8, 2, 4, 2, 3, 11, -5> ≡ <11, 8, 4, 3, 2, 2, -5>
```

The algorithm used by `SORT` works as follows:

1. First, the input number sequence is transformed into a sequence of sequences with only one element by the `AA:[ID]` function.
2. Then, the `TREE` operator[3] applies *recursively* the `MERGE:predicate` operator to both half sub-sequences of its unorderd input, and merges their ordered results;
3. The recursive application of `MERGE:predicate` continues until a sub-sequence is split into two parts of length either two or one;
   (a) in the former case each sub-sequence is ordered (since it contains only one element), and the `MERGE` operator can be applied, returning an ordered sequence;
   (b) in the latter case the unique sub-sequence with one element is already ordered, and is directly returned.

Lexicographic ordering The same function `SORT` may be used to compute the ordering of a sequence of strings. We only need a predicate able to compare any pair of strings and answer the question if the pair is either ordered or not.

The ordering normally used for strings is said to be *lexicographic*. It is used, e.g., to order the user names in a phone directory. In such ordering, given any two strings, considered as arrays of characters:

$$a = a[i] \quad \text{and} \quad b = b[j], \qquad 1 \le i \le m,\ 1 \le j \le n,$$

we say that $a < b$ if either

$$\mathsf{ord}(a[1]) < \mathsf{ord}(b[1])$$

or

$$\mathsf{ord}(a[k]) < \mathsf{ord}(b[k]) \quad \text{and} \quad \mathsf{ord}(a[j]) = \mathsf{ord}(b[j]), \qquad 1 \le j < k.$$

Where the value of the function ord applied to a character is the corresponding index in the ASCII table. For example, $\mathsf{ord}(\text{'a'}) = 97$. The ord value of a character is also

[3] See Section 2.1.2.

called its *ordinal value*. The ordinal values of the printable subset of ASCII characters are given by Script 2.1.23.

The standard PLaSM comparison predicates LT (less than), LE (less or equal), EQ (equal), GE (greater or equal) and GT (greater than) are applicable to a pair of strings a and b, and return TRUE if either a pred b in lexicographic ordering, or FALSE otherwise. Some examples follows. Remember that all such predicates but EQ are second-order functions, i.e. must be applied twice to their arguments. The function EQ can by used at the same way if curried:

```
LT: 'Barney': 'Fred' ≡   FALSE
LE: 'Barney': 'Fred' ≡   FALSE
C:EQ: 'Barney': 'Fred' ≡ FALSE
GE: 'Barney': 'Fred' ≡   TRUE
GT: 'Barney': 'Fred' ≡   TRUE
```

Example 2.1.8 (Sort and merge of strings)
Some examples of sorting unordered sequences and merging ordered sequences of strings are given in the following. Notice that both the SORT and the MERGE functions are the same for sorting and merging numbers, respectively.

```
SORT:IsGT:< 'Fred', 'Wilma', 'Barney', 'Lucy' > ≡
   <'Barney', 'Fred', 'Lucy', 'Wilma'>

SORT:IsGT:< 'Homer', 'Margie', 'Bart', 'Lisa', 'Maggie' > ≡
   <'Bart', 'Homer', 'Lisa', 'Maggie', 'Margie'>

MERGE:IsGT: < <'Barney', 'Fred', 'Lucy', 'Wilma'>,
   <'Bart', 'Homer', 'Lisa', 'Margie'> > ≡
   <'Barney', 'Bart', 'Fred', 'Homer', 'Lisa', 'Lucy',
    'Maggie', 'Margie', 'Wilma'>
```

Associative lists An associative list is a data structure often used in programming languages. It is defined as a sequence of pairs <key, data>, where the first element is an integer and the second element may be in our case any language expression. It is used to execute a quick search of some information depending on the value of the *key*.

In Script 2.1.31 we give a function alias used to return the data value paired to an integer key in an associative list. In particular it returns the data value if the key value is found in the list; otherwise it returns the null value, i.e. the empty sequence. This function is implemented using the function assoc, which returns the nearest pair, say, the pair whose key has smallest distance from the input key. It is supposed that the pairs are mantained in increasing order of the key. In case of multiple pairs with the same key, the last associated pair is returned by alloc.

Both the expressions assoc:n and alias:n, where n is an integer, return a function that must be applied to an ordered sequence of pairs, where the first element is an integer, and the second one may be any PLaSM expression, as in the following examples:

```
alias:0:
   <<-1,35>, <2,1..3>, <5,41>, <7,43>, <8,44>> ≡ <>;
```

Script 2.1.31 (`alias` **and** `alloc`)

```
DEF nearest (key::IsInt)(p1,p2::IsPair) =
   IF:< K:(GT:d1:d2), S1, S2 >:<p1,p2>
WHERE
   d1 = ABS:(key - S1:p1),
   d2 = ABS:(key - S1:p2)
END;

DEF assoc (key::IsInt) = TREE:(nearest:key);
DEF alias (key::IsInt) = IF:< C:EQ:key ~ S1, S2, K:<> > ~ assoc:key;
```

```
alias:2:
   <<-1,35>, <2,1..3>, <5,41>, <7,43>, <18,44>> ≡ <1,2,3>;

alias:3:
   <<1,'dog'>, <2,37>, <3,'cat'>, <4,ID>, <6,<11,2>>> ≡ 'cat';
```

Set operations A quite natural representation of sets is given in `PLaSM` by using sequences. In Script 2.1.25 we have given an implementation of the set-membership predicate, which allows testing if an element e belongs to a set S, by returning `TRUE` if and only if $e \in S$, and returning `FALSE` otherwise. In the following Script 2.1.32 the four operations of set union, intersection, difference and symmetric difference are given, respectively denoted as `setOR`, `setAND`, `setDIFF` and `setXOR`.

The algorithm used is quadratic, since we stress here the code simplicity and compactness, and is similar for each operation. Hence the common part of the code is abstracted as an external function `InSet`. For example, the expression

```
InSet:NOT:set_b: set_a
```

returns the subset of `set_a` whose elements are not contained in `set_b`.

Script 2.1.32 (Set operations)

```
DEF InSet (p::IsFun)(set::IsSeq) = CAT ~ AA:(IF:<p ~ IN:set, [ID], K:<>>);

DEF setOR (set_a, set_b::IsSEq) = InSet:NOT:set_b: set_a CAT set_b;
DEF setAND (set_a, set_b::IsSEq) = InSet:ID:set_b: set_a;
DEF setDIFF (set_a, set_b::IsSEq) = InSet:NOT:set_b: set_a;
DEF setXOR = CAT ~ AA:setDIFF ~ [[S1,S2], [S2,S1]];
```

Example 2.1.9

The four set operations defined above are contemporarily applied to two sequences of strings, numbers and functions, using an infix style. The sequence of results of the operations is given below.

```
DEF A = <ID, 11, 'Lucy', 12, 'Bart', 'Albert'>;
DEF B = <'Bart', 'Homer', 11, ID>

A setOR   B ≡ <'Lucy', 12, 'Albert', 'Bart', 'Homer', 11, ID>,
```

```
A setAND  B ≡ <ID, 11, 'Bart'>,
A setDIFF B ≡ <'Lucy', 12, 'Albert'>,
A setXOR  B ≡ <'Lucy', 12, 'Albert', 'Homer'>
```

2.1.6 Pipeline paradigm

In this book we will often use the algorithmic scheme[4] based on the *pipelining* of data across repeated instances of a function with equal domain and codomain, so that the computational machinery

$$\texttt{prog} : \texttt{Dom} \to \texttt{Dom},$$

is applied n times to its own output. For example, this paradigm was used in the optimal time generation of the Pascal triangle in Script 2.1.14, in the following computation of the `ProgressiveSum` function, as well as for generating the *fractal simplex* given in Script 4.5.3. Such pipeline paradigm is depicted in Figure 2.1, where `input` $\in$ `Dom`.

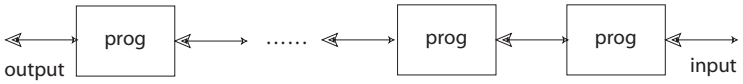

Figure 2.1 Pictorial representation of the pipeline paradigm

Progressive sums As our last example of `FL` computation with numbers, we give a function `ProgressiveSum` from sequences of reals to sequences of reals of the same length, which returns for each x_k input value the sum of the first k terms of the sequence. Formally we can write:

$$\texttt{ProgressiveSum} : \mathbb{R}^d \to \mathbb{R}^d : (x_k) \mapsto (\textstyle\sum_{i=1}^{k} x_i), \qquad 1 \leq d$$

A good example of pipelined computation is given in Script 2.1.33, where we have a computational `pipeline` made by `n` repeated applications of the `block` function, followed by a `finish` function to perform some final housekeeping. The behavior of the `block` function is represented in Figure 2.2.

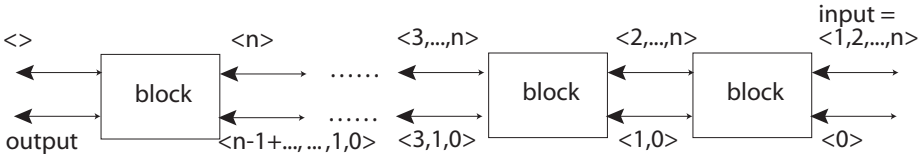

Figure 2.2 Computational pipeline of `ProgressiveSum` function

[4] Also called *paradigm*.

Script 2.1.33 (Pipeline paradigm example)

```
DEF ProgressiveSum (input::IsSeqOf:isNum) = (finish ~ pipeline):<input,<0>>
WHERE
   n = LEN:input,
   block = [TAIL ~ S1, AL ~ [FIRST ~ S1 + FIRST ~ S2, S2]],
   pipeline = (COMP ~ #:n):block,
   finish = TAIL ~ REVERSE ~ S2
END;

ProgressiveSum:<1,2,3,4,5,6,7,8,9,10> ≡ <1,3,6,10,15,21,28,36,45,55>
ProgressiveSum:<1,-1,0,1.5,2.5,10,0.5> ≡ <1,0,0,1.5,4.0,14.0,14.5>
```

Example 2.1.10 (Squares of first integers)
It may be interesting to notice that the sequence of squares of the first n integers can be computed by evaluating the `ProgressiveSum` function on the sequence of the first n *odd* numbers. E.g.:

```
ProgressiveSum:<1,3,5,7,9,11,13,15,17,19> ≡ <1,4,9,16,25,36,49,64,81,100>
```

2.2 Basic geometric programming

In this section we systematically present the geometric primitives of the `PLaSM` language. We begin by showing the very few constructors of primitive shapes, and continue by discussing the powerful `MAP` construct, used to generate any user-defined parametric map, which allow building of circumferences, spheres, toruses, as well as any kind of manifold of dimension 1, 2, 3 and even of higher dimension. The basic geometric concepts used here are discussed with much greater detail in Chapter 4.

2.2.1 Primitive shapes

Only three primitive shape constructors `SIMPLEX`, `CUBOID` and `CYLINDER` are predefined in `PLaSM`, for building dimension-independent simplices and cuboids and for 3D cylinders, respectively. The very powerful basic generator `MKPOL` of polyhedral complexes is also discussed here.

Simplex The *convex hull* of a set of points of $\mathbb{E}^n$ is the smallest convex set of $\mathbb{E}^n$ which contains all the points.

The *simplex* of dimension d, or *d-simplex*, is defined as the *convex hull* of $d+1$ *affinely independent* points of Euclidean space of dimension n, with $d \leq n$. Two such points are affinely independent when they are non-coincident, three points are a.i. when non-aligned, four points are a.i. when non-coplanar, and so on.

In particular, in $\mathbb{E}^3$, 0-simplex is a single point, 1-simplex is a segment, 2-simplex is a triangle, 3-simplex is a tetrahedron.

A primitive function `SIMPLEX` is available in `PLaSM`, to be applied to an integer parameter d, which returns the *standard* d-simplex of Euclidean d-space. This one is the convex hull generated by the origin $\boldsymbol{o}$ and by the d points $\boldsymbol{o} + \boldsymbol{e}_i$, where $(\boldsymbol{o}, \{\boldsymbol{e}_i\})$ is a Cartesian frame of $\mathbb{E}^d$. In Figure 2.3 the show the standard d-simplices of $\mathbb{E}^3$, with $d = 0, 1, 2, 3$.

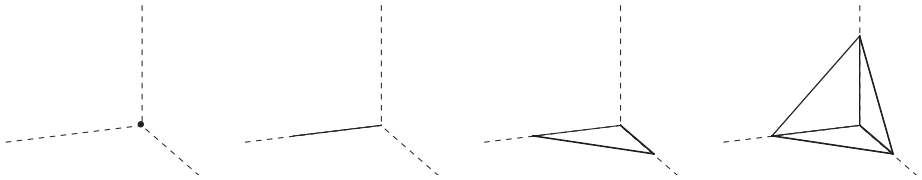

Figure 2.3 Standard d-simplices of dimension $0, 1, 2$ and 3 embedded in $\mathbb{E}^3$

Cuboid A primitive PLaSM function named CUBOID is used to generate the geometric model of hyper-parallelopipeds of dimension n, depending on the number of arguments passed when invoking the function. This function allows generation of line segments, plane rectangles, space parallelepipeds, and even hyper-parallelopipeds of intrinsic dimension higher than three.

Script 2.2.1 (Unit hyper-parallelopipeds)

```
DEF Segment   = CUBOID:<1>;
DEF Square    = CUBOID:<1,1>;
DEF Cube      = CUBOID:<1,1,1>;
DEF Hypercube = CUBOID:<1,1,1,1>;
```

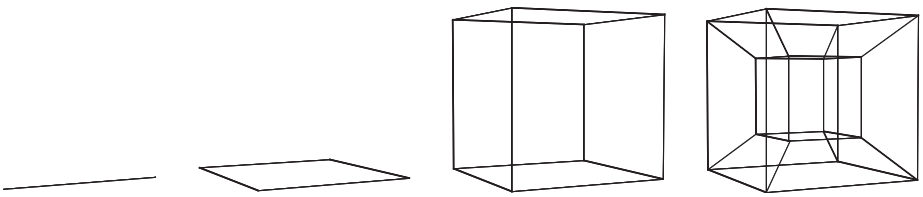

Figure 2.4 Unit segment, unit square, unit cube, unit 4D hypercube projected in 3D space

Example 2.2.1 (d-dimensional cuboid)
A function to generate 1D segments, 2D rectangles, 3D parallelepipeds and higher dimensional hypercubes, shown in Figure 2.4, is quite easy to define in PLaSM. It is sufficient to transform the sequence of numeric arguments into a sequence of 1D polyhedra, and then to apply to this sequence — via the INSR combining form — the binary product operation between polyhedra defined in [BFPP93], and implemented as * operation in PLaSM. Both such user-specified definition of the cuboid function and an example of computation are given in the following.

Cylinder A predefined function CYLINDER generates 3D cylinder of variable radius and height, polyhedrally approximated with a variable number of side facets. The CYLINDER function has two real arguments and one integer arguments.

Remember that the geometric kernel of the language is able to work only with linear polyhedra, i.e. geometric object with no curved faces.

Script 2.2.2

```
DEF cuboid = (INSR:*) ~ (AA:(QUOTE ~ [ID]))

cuboid:<x1, x2, ..., xd>
   = (INSR:* ~ AA:(QUOTE ~ [ID])):<x1, ..., xd>
   = (INSR:*):((AA:(QUOTE ~ [ID])):<x1, ..., xd> )
   = (INSR:*):< (QUOTE ~ [ID]):x1, ..., (QUOTE ~ [ID]):xd >
   = (INSR:*):< QUOTE:<x1>, ..., QUOTE:<xd> >
   = (INSR:*):< p1, ..., pd >
```

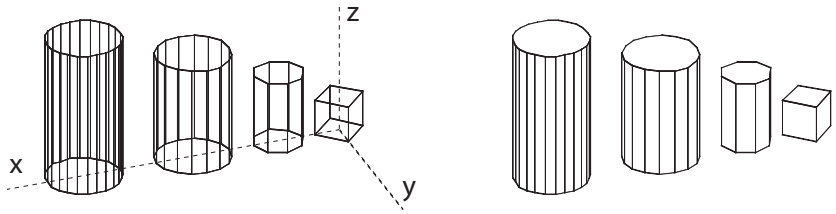

Figure 2.5 Cylinders with different radius, height and number of side faces

The cylinders in Figure 2.5 are generated by the following PLaSM expression, where each cylinder is moved by a translation along the x axis.

```
STRUCT:<
   CYLINDER:<1, 1>:4, T:1:2.5,
   CYLINDER:<1, 2>:8, T:1:3.5,
   CYLINDER:<1.5, 3>:16, T:1:4.5,
   CYLINDER:<1.5, 4>:24
>;
```

Basic polyhedral constructor The PLaSM language contains only one basic constructor, named MKPOL, of geometric objects. The type of geometric objects is called *polyhedral complex*. As we discuss in Chapter 4, this kind of complex can be seen as a set of convex cells, grouped into subsets called polyhedral cells. Each cell is obtained as the convex hull of a discrete subset of points. Therefore there are three ingredients in a *polyhedral complex*:

1. a discrete set of points, called *vertices*;
2. a discrete set of convex cells, called *cells*;
3. a discrete set of collections of cells, called *polyhedra*.

All vertices must have the same number of coordinates, i.e. must belong to the same Euclidean space. Depending on the number of coordinates, the resulting object may be either 1D, or 2D, or 3D or even higher-dimensional.

Each cells is defined as the convex hull of a subset of vertices. The various cells give a cell *covering* of the resulting object. The cells may have any intrinsic dimension, i.e. may be topologically equivalent to either a point, a curve, a surface, a solid and so on.

Each polyhedron in a complex is in turn defined as the union of a subset of convex cells, possibly embedded in different affine subspaces of the Euclidean space. When

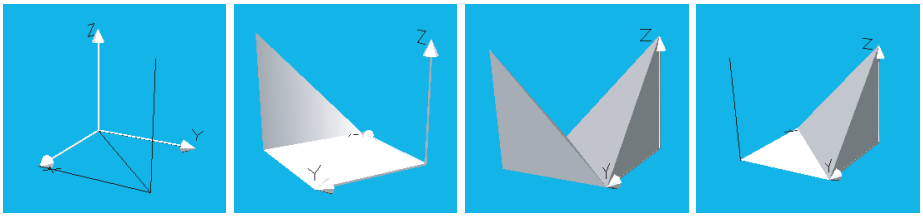

Figure 2.6 Polyhedral complexes of dimension 1,2,3 and of mixed dimensionality, drawn with a model of reference frame

the affine hulls of all cells of a polyhedron are coincident, the polyhedron is said to be *regular* or homogeneously dimensional. When all the polyhedra in a complex are regular and have the same dimension of the embedding space, the complex is *solid*.

A polyhedral complex is produced in `PLaSM` by applying the basic constructor `MKPOL` to a triplet of *vertices*, *cells* and *polyhedra*. Vertices are specified as sequences of coordinates, cells and polyhedra are given as sequences of indices of vertices and cells, respectively.

Example 2.2.2 (Polyhedral complexes)
Some complexes of different dimensionality are defined in Script 2.2.3 and are shown in Figure 2.6. All examples rely on the same set, named `verts`, of vertices in $\mathbb{E}^3$. In particular, the object named `pol_1D` generates a 1D complex with three convex cells (segments) defined by pairs of vertices with indices `<1,4>`, `<2,4>` and `<4,8>`, respectively. Analogously, the `pol_2D` object has two convex cells with four and three vertices, respectively. The `pol_3D` object is a solid complex with two convex cells of four vertices. Finally, the `mixed_D` object, of mixed dimensionality, contains three cells, of dimension 3, 2 and 1, respectively. The generated objects are combined with the model of the reference frame given by the symbol `MKframe`, that is defined in Script 6.5.3.

Script 2.2.3

```
DEF verts =
   <<0,0,0>,<1,0,0>,<0,1,0>,<1,1,0>,<0,0,1>,<1,0,1>, <0,1,1>,<1,1,1>>;

DEF pol_1D = MKPOL:< verts, cells, pols > STRUCT MKframe
WHERE
   cells = <<1,4>,<2,4>,<4,8>>,
   pols = <<1,2,3>>
END;

DEF pol_2D = MKPOL:<verts, <<1,2,3,4>,<2,4,8>>, <<1>,<2>>> STRUCT MKframe
DEF pol_3D = MKPOL:<verts, <<1,2,3,5>,<2,3,4,8>>, <<1,2>>> STRUCT MKframe
DEF mixed_D = MKPOL:<verts, <<1,2,3,5>,<2,3,4>,<4,8>>, <<1,2,3>> >
   STRUCT MKframe
```

2.2.2 Non-primitive shapes (simplicial maps)

As we already said, the geometric kernel of the language gives support only for modeling linear polyhedra. Curved objects are always approximated — with user-defined precision — by linear polyhedra. This is actually the standard approach in computer graphics, where every curved line is drawn as a polygonal line made of many small segments, and every surface is rendered as faceted by a great number of small triangles. Analogously, a curved solid may be approximated by a number of small tetrahedra.

All such approximations can be seen as resulting from the application of a so-called *simplicial mapping* to some properly decomposed *domain*. The domain image in such a mapping gives a polyhedral approximation of the curved object. The finer the domain decomposition, the better is the linear approximation of the object. A well-known example is the circle approximation by regular polygons with increasing number of sides.

Parametric `MAP` operator The very useful primitive operator `MAP` allows for generating curves and surfaces — as well as higher-dimensional manifolds — via parametric maps from a simplicial decomposition of a polyhedral complex. The predefined operator `MAP` is used at this purpose as

```
MAP:f:domain
```

where

- the mapping `f` is either the *construction* `[f1, f2, ... , fn]` of a number n of *coordinate functions* or is their *sequence* `<f1, f2, ... , fn>`. Notice that n is the dimension of the *target space* of the mapping, where the `domain` is embedded and (usually) incurved, and
- `domain` is a *polyhedral complex*. A commonly enforced requirement is that the intrinsic dimension of `domain` is either less or equal than the dimension of the target space of the mapping.

The operational semantics of the `MAP` operator can be briefly described as follows:

1. compute a simplicial decomposition of the convex cells of the polyhedral `domain`;
2. apply to each vertex of such a decomposition the coordinate functions `f1, f2, ... , fn`, in order to generate the vertex image in the target space $\mathbb{E}^n$.

Notice that each *coordinate function* will produce one of the n coordinates of vertices in target space, by properly combining vertex coordinates in domain space. Since vertices are internally seen as sequences of coordinates, these *must* be extracted, in the formal definition of the coordinate functions, by using the standard `FL` selector functions `S1`, `S2`, ... , `Sn`. This is a strict *semantic requirement* when using the `MAP` operator. We hope the examples will clarify this point.

Circumference A polygonal line with n segments which approximates the boundary of the unit circle can easily be generated as discussed in the following.

Our aim is to implement the trigonometric parametrization of circumference with unit radius given by equations:

$$\begin{aligned} x(u) &= \cos u \\ y(u) &= \sin u \end{aligned}$$

where $u \in [0, 2\pi]$.

First, we decompose the $[0, 2\pi]$ interval into n parts of equal size. For this purpose we generate the sequence of n numbers $\frac{2\pi}{n}$. This sequence is then transformed into a 1D polyhedron by the `PLaSM` primitive `QUOTE`. Finally, on such domain decomposition we apply the trigonometric mapping `f:` $\mathbb{R} \to \mathbb{R}^2$, where

```
f = [cos, sin] ~ S1
```

As we already noticed, the primitive selector `S1` must be used to extract the first (and unique) coordinate from vertices of the domain decomposition into small segments. In particular, a polygonal approximation with 24 line segments of the circumference is obtained as done in Script 2.2.4 and shown in Figure 2.7. Notice that `Circumference` is a function in proper `FL` style, that returns a polyhedral complex when applied to some positive integer.

Script 2.2.4 (Circumference)

```
DEF Domain (a::IsReal)(n::IsIntPos) = (QUOTE ~ #:n):(a/n);
DEF Circumference = MAP:([COS, SIN] ~ S1) ~ Domain:(2*PI);

Circumference:24
```

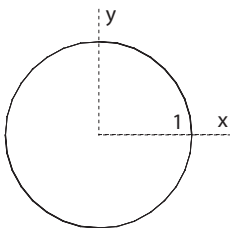

Figure 2.7 Unit circumference centered in the origin, as generated by the function `Circumference`

Circle A polyhedral approximation of the unit circle can be easily obtained as the `JOIN` of the circle boundary. We remember that this function generates the convex hull of its polyhedral argument:

```
(JOIN ~ Circumference:1):24;
```

Actually, a finer decomposition of the circle of radius r can be obtained by using the parametric equations

$$\begin{aligned} x(u) &= v \cos u \\ y(u) &= v \sin u \end{aligned}$$

where $\leq u \leq 2\pi,\ 0 \leq v \leq r$, or better, $(u, v) \in [0, 2\pi] \times [0, r]$. Hence the polyhedral approximation of the circle is obtained by the map $\gamma : \mathbb{R}^2 \to \mathbb{R}^2$ such that

$$\gamma = [\texttt{S2 * COS} \sim \texttt{S1}, \texttt{S2 * SIN} \sim \texttt{S1}]$$

Notice, according to Script 2.2.5, that the γ function will be applied by the `MAP` operator to all vertices of the simplicial decomposition of the complex generated by the function-valued expression `Domain2D:<2*PI,1>`. Notice also that `Domain2D` is a second-order function which produces a decomposition into `n`×`m` sub-intervals of the 2D domain produced by Cartesian product of the intervals $[0, \texttt{a}]$ and $[0, \texttt{b}]$.

Script 2.2.5 (Circle)

```
DEF Domain2D (a,b::IsReal)(n,m::IsIntPos) = Domain:a:n * Domain:b:m;
DEF Circle (r::IsReal) =
   MAP:[S2 * COS ~ S1, S2 * SIN ~ S1] ~ Domain2D:<2*PI,r>;

Domain2D:<2*PI,1>:<24,4>;
Circle:1:<24,4>;
```

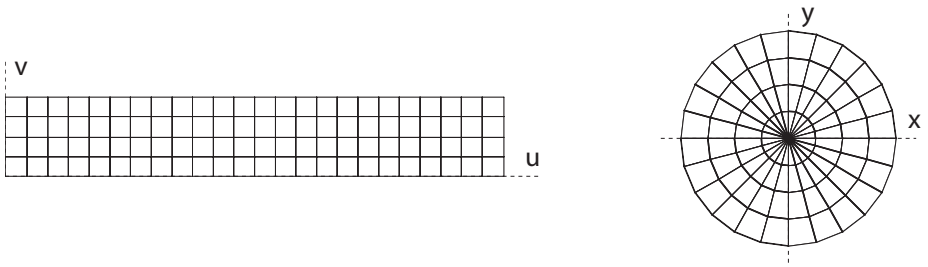

Figure 2.8 Domain and range decomposition associated with `Circle:1:<24,4>`

In Figure 2.8 there are actually represented the polyhedral domains generated by evaluating the following `PLaSM` expressions, where the `@1` operator executes the extraction of the 1-skeleton of the the resulting object, i.e. the set of its 1-cells (see Chapter 4 for a definition):

```
(@1 ~ Domain2D:<2*PI,1>):<24,4>;
(@1 ~ Circle:1):<24,4>
```

The generating code can be further specialized to produce circles with a hole, called *rings*, and circular segments with either a filled or an empty hole, by just modifying the limits of the domain interval in parameter space. The definition of a function `Ring` in proper `FL` style is given in Script 2.2.6. The `Ring` function actually depends also (implicitly) on two integer parameters, needed by the partially specified function `Interval2D` in its body. A picture of the object generated by the last expression of the Script is shown in Figure 2.9.

Script 2.2.6

```
DEF Interval (x1,x2::IsReal)(n::IsIntPos) =
   (T:1:x1 ~ QUOTE ~ #:n):((x2-x1)/n);

DEF Interval2D (a1,a2,b1,b2::IsReal)(n,m::IsIntPos) =
   Interval:<a1,a2>:n * Interval:<b1,b2>:m;

DEF Ring (r1,r2::IsReal) =
   MAP:[s2 * cos ~ S1, s2 * sin ~ S1] ~ Interval2D:<0,2*PI, r1,r2>;

(@1 ~ Ring:<0.5,1>):<24,4>;
```

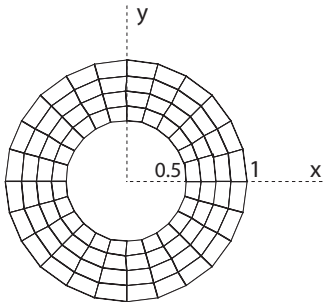

Figure 2.9 Polyhedral approximation of the circular ring by a decomposition of domain directions into 24 and 4 parts

Sphere The trigonometric parametrization of the spherical surface with radius r and center in the origin has equations:

$$\begin{aligned} x(u,v) &= -r\cos u \sin v, \\ y(u,v) &= r\cos u \cos v, \\ z(u,v) &= r\sin u, \end{aligned}$$

where $(u,v) \in [-\frac{\pi}{2}, \frac{\pi}{2}] \times [0, 2\pi]$.

A second-order function `Sphere` is given in Script 2.2.7, which must be applied first to a real positive `radius` parameter. The resulting partial function must then be applied to a pair of integers `n` and `m`, used to specify the "grain" of the polyhedral approximation of the surface, i.e. of its domain decomposition.

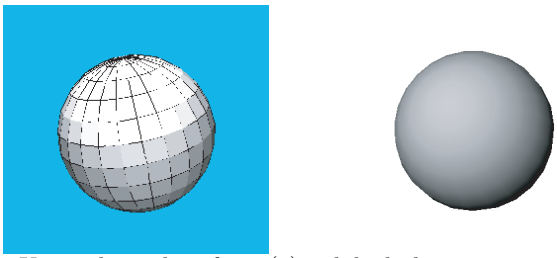

Figure 2.10 Unit spherical surface: (a) polyhedral approximation of the surface (b) smooth rendering via color shading

Script 2.2.7 (Spherical surface)

```
DEF Sphere (radius::IsRealPos)(n,m::IsIntPos) = MAP:[fx,fy,fz]:domain
WHERE
   fx = K:radius * - ~ SIN ~ S2 * COS ~ S1,
   fy = K:radius * COS ~ S1 * COS ~ S2,
   fz = K:radius * SIN ~ S1,
   domain = Interval:<PI/-2,PI/2>:n * Interval:<0,2*PI>:m
END;

(STRUCT ~ [ID, @1] ~ Sphere:1):<12,24>;
Sphere:1:<12,24> CREASE (PI/2);
```

The result of the evaluation of the last two expressions of Script 2.2.7 is shown in Figures 2.10a and 2.10b. Notice that both the spherical surface and its 1-skeleton are aggregated in the resulting object of Figure 2.10a, in order to better highlight the facets of the approximating polyhedron. Conversely, a *color shading* is used when browsing the VRML file generated when exporting the geometric object produced by the last `PLaSM` expression. In particular, the surface facets are rendered on the screen by using a color interpolation tecnique, according with the value of the *crease angle* between adjacent facets. This smooth rendering technique, which is very frequently used in computer graphics, is discussed in Section 10.6. The reader should notice that *both* images are produced by using the *same* (polyhedral) geometric model.

Cone A very different constructive method is used in Script 2.2.8 to generate a solid model of the 3D cone with assigned `radius` and `height`. In particular we generate the cone as the *convex hull* of the `basis` and the `apex`, represented as polyhedra of dimension 2 and 0, respectively. The predefined `PLaSM` operator `JOIN` is used at this purpose. Notice that, once more, the generated object is a polyhedral approximation of the curved solid.

The function `basis` is the composition of two functions. The `Circle:radius` function generates a 2D circle of proper `radius` centered in the origin, whereas the `EMBED:1` function embed this circle in 3D, actually putting it in the coordinate subspace $z = 0$. The `apex` object is a 0D polyhedron constituted by a single point, which is embedded in 3D by the function `EMBED:3` and then is translated along the z coordinate by the function `T:3:height`.

Notice that the integer parameter of the `EMBED:`d' unary operator must be the *codimension* (see Section 3.1) of the geometric object returned by the operator, with $d+d' = n$, where d and n are the *intrinsic* and the *embedding* dimensions, respectively.

Script 2.2.8

```
DEF Cone (radius, height::IsReal)(n::IsInt) = JOIN:< basis:<n,1>, apex >
WHERE
   basis = EMBED:1 ~ Circle:radius,
   apex = (T:3:height ~ EMBED:3 ~ SIMPLEX):0
END;
```

An equivalent definition for the `apex` object, as a 0D polyhedral complex made by a single cell, in turn consisting in only one 3D point, could be given as

```
apex = MKPOL:< <<0,0,height>>, <<1>>, <<1>> >
```

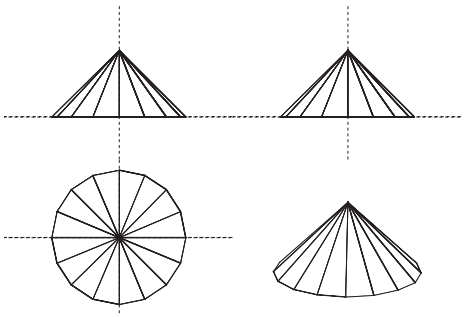

Figure 2.11 Orthographic and dimetric projections of the faceted cone with unit radius and height generated by `Cone:<1,1>:16`

Cylinder and tube A primitive `CYLINDER` generator is predefined in `PLaSM`. A possible implementation is given by the `Cylnder` function in Script 2.2.9. It works by computing the Cartesian product of a circle of radius r times a 1D segment of length h. The same approach is used to compute the empty cylinder with the function `Tube`, as a Cartesian product of a 2D object generated by the `Ring` function, times a 1D segment of proper length. In both cases the formal parameter `n` denotes the number of facets in the polyhedral approximation of the side surface. Pictures of the object generated by last expressions of Script 2.2.9 are given in Figure 2.12. Notice the graphical smoothing of adjacent facets is performed only when their common angle is *greater* than $\frac{\pi}{2}$.

Script 2.2.9

```
DEF Cylnder (r,h::IsReal)(n::IsInt) = Circle:r:<n,1> * QUOTE:<h>;
DEF Tube (r1,r2,h::IsReal)(n::IsInt) = Ring:<r1,r2>:<n,1> * QUOTE:<h>;

Tube:<0.8, 1, 2>:24
Tube:<0.8, 1, 2>:24 CREASE (PI/2)
```

Torus A *torus* is a 3D surface with the characteristic shape of a doughnut. Such a surface can be thought of as being generated by rotating with angle 2π a circumference about an axis of its plane, providing that the rotation axis does not intersect the circumference.

In particular, the torus surface may be generated by rotating, about the z axis, a circle on the plane $y = 0$, with center $(0, 0, R)$ and radius r. In this case the parametric equations of the surface are:

Figure 2.12 Geometric object generated by expression `Tube:<0.8,1,2>:24` (a) standard rendering (b) smooth rendering via color shading

$$\begin{aligned} x(u,v) &= (r\cos u + R)\cos v, \\ y(u,v) &= (r\cos u + R)\sin v, \qquad (u,v) \in [0,2\pi]^2 \\ z(u,v) &= r\sin u \end{aligned} \tag{2.4}$$

Such equations are directly translated in `PLaSM` to the function `torus` given in Script 2.2.10. In this case the symbol `domain2D` is redefined locally, to take into account the new domain limits of u and v parameters. Notice once more that the coordinate functions `fx`, `fy` and `fz` are written by using the `FL` selectors `S1` and `S2`. Their aim is to properly select the coordinates of vertices of the simplicial decomposition of the mapping domain generated by the `MAP` operator.

Script 2.2.10

```
DEF torus (r1,R2::IsReal) (n,m::IsIntPos) = MAP:[fx,fy,fz]:domain
WHERE
   fx = (K:r1 * COS ~ S1 + K:R2) * COS ~ S2,
   fy = (K:r1 * COS ~ S1 + K:R2) * SIN ~ S2,
   fz = (K:r1 * SIN ~ S1),
   domain = Interval:<0,2*PI>:n * Interval:<0,2*PI>:m
END;

torus:<1,3>:<12,24>
torus:<1,3>:<12,24> CREASE PI/2
```

A different method of generating the torus as the manifold product of 2D circles will be given in Example 5.4.2.

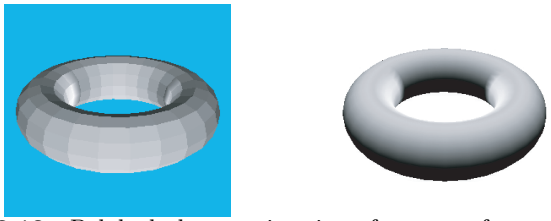

Figure 2.13 Polyhedral approximation of torus surface produced by `torus:<1,3>:<12,24>` (a) standard rendering (b) smooth rendering

2.3 Assembling shapes

In geometric modeling of complex assemblies the geometric and graphics programmer makes wide use of the so-called *hierarchical scene graphs* [WO94, SRD00] or hierarchical structures [Gas92], where sub-assemblies are defined in local coordinates, and are transformed into the coordinate frame of the calling assembly by explicitly using proper coordinate transformations. Such "modeling transformations" are left entirely to the programmer's responsibility. To this topic we dedicate the whole of Chapter 8. In the present section we conversely discuss some simplified assembly operators, where the coordinate transformations are computed automatically by the language.

2.3.1 Primitive alignments

Measuring A primitive function `SIZE` of second order provides support for computing the size of the projection of any polyhedral complex along any coordinate subspace. The result of a double application of the `SIZE` function to either a coordinate index or a sequence of indices, followed by the application to the polyhedral argument, will accordingly return either a single positive number or a sequence of positive numbers. For example:

```
(SIZE:1 ∼ T:1:PI ∼ CYLINDER):<1,1,24> ≡ 2.0
(SIZE:<1,2,3> ∼ CYLINDER):<1,1,24> ≡ <2.0, 2.0, 1.0>
(SIZE:<1,2,3> ∼ R:<2,3>:(PI/2) ∼ CYLINDER):<1,1,24> ≡ <2.0, 1.0, 2.0>
```

Pointing Three specialized functions `MIN`, `MED` and `MAX` may be used to compute the corresponding extreme (or middle) values within the coordinate interval projected by a given polyhedral complex on each coordinate axis. For example:

```
DEF TheCylinder = CYLINDER:<1,2>:24

MIN:3:TheCylinder ≡ 0.0;
MED:3:TheCylinder ≡ 1.0;
MAX:3:TheCylinder ≡ 2.0;

MIN:<1,2>:TheCylinder ≡ <-1.0, -1.0>;
MED:<1,2>:TheCylinder ≡ <0.0, 0.0>;
MAX:<1,2>:TheCylinder ≡ <1.0, 1.0>;
```

Boxing Similarly, a predefined function `BOX` of second order returns the containment box of a proper projection of the polyhedral argument. The result of a double application of the `BOX` function to a sequence of indices, followed by the application to a polyhedral complex, is the minimal hyper-parallelepiped parallel to the reference frame which contains the polyhedral argument in its interior. The results of the evaluation of the three examples given in the following are shown in Figure 2.14.

```
(@1 ∼ STRUCT ∼ [ID, BOX:<1,2>] ∼ CYLINDER):<1,1,24>;
(@1 ∼ STRUCT ∼ [ID, BOX:<1,2,3>] ∼ CYLINDER):<1,1,24>;
(STRUCT ∼ [ID, @1 ∼ BOX:<1,2,3>] ∼ CYLINDER):<1,1,24>;
```

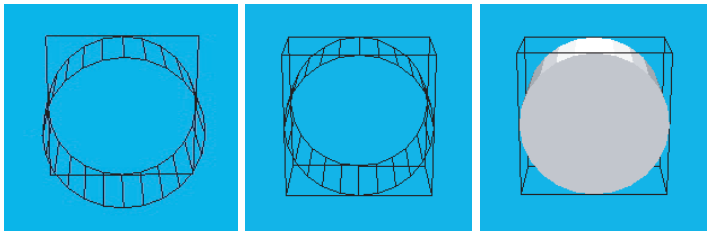

Figure 2.14 (a) Containment box of the cylinder projection in a coordinate plane (b, c) Containment box of the cylinder

Aligning Every pair of polyhedral complexes may be aligned along any given subset of coordinates by using the primitive binary function `ALIGN`. Such a second-order operator must be applied to a sequence of triples which define a specialized behavior for each affected coordinate. The resulting specialized operator is then applied to a pair of polyhedral complex, and returns a single polyhedral complex. Each alignment directive along a coordinate must belong to the set

$$\{1, \ldots, n\} \times \{\texttt{MIN, MED, MAX, K:}\alpha\} \times \{\texttt{MIN, MED, MAX, K:}\alpha\}$$

where n is the dimension of the embedding space of the two polyhedral arguments, and $\alpha \in \mathbb{R}$.

Example 2.3.1 (Diagram)
The simple diagram of Figure 2.15 is produced by Script 2.3.1. The *circle*, *square* and *segment* symbols are named `a`, `b` and `l`, respectively. Then the sequence of symbols `< a,l,b,l,a >` is pairwise aligned according to the directives `<<1,MAX,MIN>,<2,MED,MED>>`. Therefore, *each* consecutive pair, say `<p1,p2>`, is aligned so that:

1. the `MAX` value of coord 1 of `p1` coincides with the `MIN` value of coord 1 of `p2`;
2. the `MED` value of coord 2 of `p1` coincides with the `MED` value of coord 2 of `p2`.

Script 2.3.1

```
DEF a = Circle:0.75:<24,1>;
DEF b = CUBOID:< 1,1 >;
DEF l = (EMBED:1 ~ CUBOID):< 1 >;

INSL:(ALIGN:<<1,MAX,MIN>,<2,MED,MED>>):< a,l,b,l,a >;
```

Example 2.3.2 (Histogram)
A very simple implementation of various kind of histogram generating functions is given in Script 2.3.2. The three definitions correspond to a 2D histogram and to a 3D histogram of a single sequence of data, and to a 3D histogram of a double sequence of data, respectively. The histograms of the number series given in the script are shown in Figure 2.16.

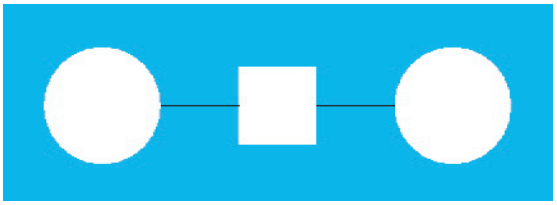

Figure 2.15 Simple diagram generated by alignment operators

Script 2.3.2 (Histograms)

```
DEF histogram1 = INSR:(ALIGN:<<1,MAX,MIN>>) ~ AA:(CUBOID ~ [K:1,ID]);
DEF histogram2 = INSR:(ALIGN:<<1,MAX,MIN>>) ~ AA:(CUBOID ~ [K:1,K:1,ID]);
DEF histogram3 = INSR:(ALIGN:<<2,MAX,MIN>>) ~ AA:histogram2;

histogram1:<3,5,4,6.5,2.34,6,7>;
histogram2:<3,5,4,6.5,2.34,6,7>;
histogram3:<
   <3,5,5,6.5,2.34,6,7>,
   <2,5,4,3.6,3,4,2>,
   <4,5,6,1.5,4,1,0.5>,
   <1,5,3,4.2,2.5,5,1>,
   <0.5,3,2,3.5,1,6,2>,
   <0.1,5,4,1.5,2.4,1,4> >;
```

2.3.2 Relative arrangements

The PLaSM language has some predefined binary functions used to relatively locate two complexes with respect each other. In particular, the second argument of such functions will be positioned either TOP the first one, or ABOVE, or LEFT, or RIGHT, or UP, or DOWN, respectively, according to the alignment semantics given in Script 2.3.3, where the definitions of such primitive functions are reported.

As we show in this section, such relative positioning operator allows for the easy construction of quite complex assemblies, without taking explicitly into account the local coordinate frames where the sub-assemblies and the elementary parts are defined. With an approach of this kind we avoid applying affine transformations explicitly parametrized on the dimensions of the arguments objects, as it is conversely needed in hierarchical scene graphs.

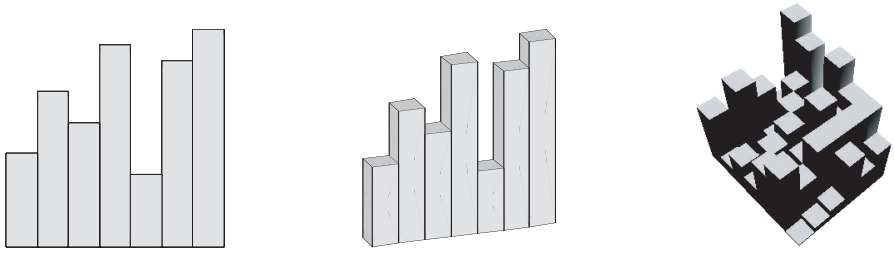

Figure 2.16 (a) 2D histogram (b) 3D histogram (c) 3D histogram of a double sequence of values

Binary placements The predefined binary function `TOP`, given in Script 2.3.3, locates the second argument on top of the first, by making also coincident the centers of their projection extents on the xy plane. In particular, a proper translation is applied to the second object, with translation vector (see Section 6.2.1) produced by the difference of points generated by the expressions

`[MED,MED,MIN]:pol2` and `[MED,MED,MAX]:pol1`

The resulting complex is defined in the local frame of the first argument (i.e. `pol1`). Similar placements are produced by the other operators `BOTTOM`, `LEFT`, `RIGHT`, `UP` and `DOWN`.

Script 2.3.3 (Relative arrangements)

```
DEF TOP (pol1, pol2 ::IsPol) =
   ALIGN:<<3,MAX,MIN>,<1,MED,MED>,<2,MED,MED>>:<pol1, pol2>

DEF BOTTOM (pol1, pol2 ::IsPol) =
   ALIGN:<<3,MIN,MAX>,<1,MED,MED>,<2,MED,MED>>:<pol1, pol2>

DEF LEFT (pol1, pol2 ::IsPol) =
   ALIGN:<<1,MIN,MAX>,<3,MIN,MIN>>:<pol1, pol2>

DEF RIGHT (pol1, pol2 ::IsPol) =
   ALIGN:<<1,MAX,MIN>,<3,MIN,MIN>>:<pol1, pol2>

DEF UP (pol1, pol2 ::IsPol) =
   ALIGN:<<2,MAX,MIN>,<3,MIN,MIN>>:<pol1, pol2>

DEF DOWN (pol1, pol2 ::IsPol) =
   ALIGN:<<2,MIN,MAX>,<3,MIN,MIN>>:<pol1, pol2>
```

Example 2.3.3 (Single placement)
Let us consider the standard unit cube and the cylinder with unit radius and height generated by `PLaSM` expressions

```
DEF Cube = CUBOID:<1,1,1>;
DEF Cyl = CYLINDER:<1,1>:16;
```

The result of the relative placement produced by the expression

```
Cube TOP Cyl;
```

is graphically shown in Figure 2.17.

Example 2.3.4 (Multiple placement)
The `TOP` operator may be used more times in the same expression, both with and without parentheses, since it is left- and right-associative. Anyway, the reader should remember that every multiple infix expression in `PLaSM` is left-associative. The result of each expression in Script 2.3.4 is shown in Figure 2.18a.

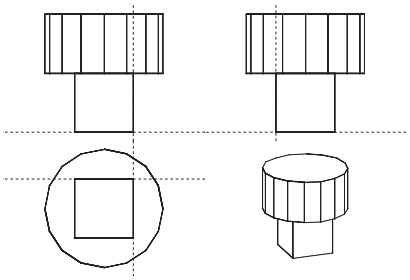

Figure 2.17 Monge's projections and dimetric projection of the result of the expression `Cube TOP Cyl`

Script 2.3.4 (Associative property)

```
(Cube TOP Cyl) TOP Simplex:3; ≡
Cube TOP (Cyl TOP Simplex:3); ≡
Cube TOP Cyl TOP Simplex:3;
```

The aggregation by `TOP` of several instances of two objects may be defined by exploiting the combinatorial features of the language. For example, four instances of the pair `<Cube, Cyl>` may be vertically aggregated as shown in Script 2.3.5.

where the binary operator `TOP` is applied to a sequence of eight arguments by using the `FL` combinator `INSL`. The result of the evaluation of this expression is shown in Figure 2.18b.

2.4 Examples

Two examples of assembly by relative location operators are given in this section. The first one is a mechanical example where a parametrized assembly of two different families of nuts is discussed. In the second example we generate a simplified model of a Greek temple. A programming example concludes this section, aiming at showing the great difference in making geometric programming by using the standard iteration of functional programming, based on recursion, and the proper `FL` approach based on combinators.

2.4.1 Parametric nut stack

We define here a parametric model of a mechanical assembly constituted by alternate cylindric and cubic nuts, with a variable number of such parts, each one with a variable width. In particular our aim is to write a generating function depending on the numeric sequence of nut widths.

Let us start by giving two generating functions `Nut_1` and `Nut_2` which produce a squared and hexagonal nut model, respectively, when applied to the `h` parameter.

Notice that both the `Nut_1` and the `Nut_2` partial function must be further applied

Script 2.3.5 (n-ary usage of the operator)

```
(INSL:TOP ∼ ##:4):<Cube, Cyl>
```

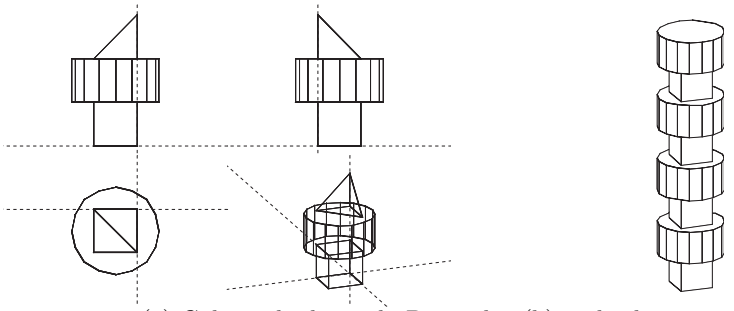

Figure 2.18 (a) Cube, cylinder and 3D simplex (b) multiple aggregation

Script 2.4.1 (Nut stack)

```
DEF Nut (radius::IsReal; nstep::IsIntPos) (h::IsReal) =
   CYLINDER:<radius,h>:nstep;

DEF Nut_1 = Nut:<0.5, 4>;
DEF Nut_2 = Nut:<0.5, 6>;

(INSR:TOP ~ AA:APPLY ~ TRANS):
   < ##:4:< Nut_1, Nut_2 >, <0.1, 1, 0.3, 0.3, 1, 0.4, 2, 0.2> >
```

to a real `h` parameter to generate a geometrical object.

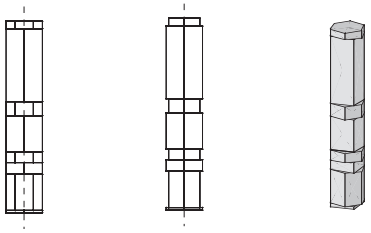

Figure 2.19 Composite nut assembled from parts of various heights

The following functions are orderly applied to the pair constituted by the function sequence `##:4:< Nut_1, Nut_2 >` and by the sequence of numbers:

1. the function `TRANS` returns a sequence of pairs <*function*, *number*>;
2. to each such pairs is applied the primitive `FL` function `APPLY`, thus generating a sequence of nuts, each one defined in a local coordinate system;
3. the function `INSR:TOP` is finally applied to the nut sequence, so producing the geometric model of the whole assembly, which is shown in Figure 2.19.

2.4.2 Hierarchical temple

In this section we discuss the hierarchical modeling of a simplified Greek temple. The construction will proceed bottom-up by generating:

1. the single column with a beam;
2. a row of columns;

3. the gable;
4. a row of columns with gable;
5. a first temple version;
6. the arrangement of secondary beams;
7. the final temple version.

Column definition A simplified column model may be defined as shown in Figure 2.20, where a parallelepiped `basis` and `capital` are separated by the cylinder `trunk` of the column, approximated by 12 rectangular facets. On the column top is centered a `beam` obtained by scaling three times in the direction of the first coordinate the parallelepiped model of the capital. The model generated by the `column` function is parametrized by positive real formal parameters `r` and `h`, corresponding to the `trunk` radius and to the vertical size of the aggregation of `basis`, `trunk` and `capital`. Both `basis` and `capital` are defined with side `1.2 * r`.

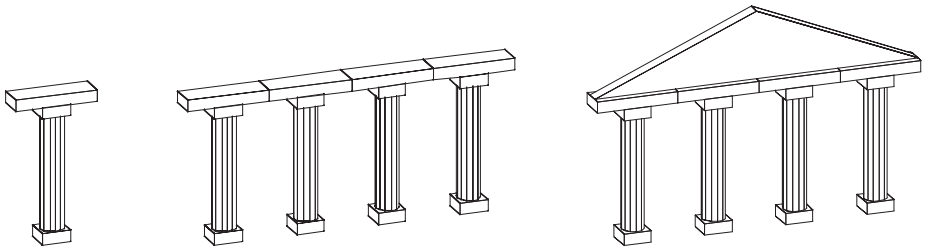

Figure 2.20 Projections of the models generated by `Column:<1,12>`, by `ColRow:4` and by `ColRowAndGable`, respectively

Script 2.4.2 (Column)

```
DEF Column (r,h::IsRealPos) = basis TOP trunk TOP capital TOP beam
WHERE
   basis = CUBOID:< 2*r*1.2, 2*r*1.2, h/12 >,
   trunk = CYLINDER:< r, (10/12)*h >:12,
   capital = basis,
   beam = S:1:3:capital
END;

DEF col = Column:<1, 12>
```

For the sake of computational efficiency it is useful to invoke the `Column` function through the definition of a constant symbol, say, the symbol `col` in Script 2.4.2. When such a symbol is evaluated the first time, its value is computed by evaluating the *body* of its definition and it is stored as associated to the symbol. In every subsequent invocation of the symbol such value is just referred to, and is neither computed nor stored again.

Column row A row of n columns, with n positive integer, is generated by the `ColRow` function of Script 2.4.3, where the `column` value memorized in the constant `col` is instanced n times and located each time on the `RIGHT` of the previous instancing. The model generated by `ColRow:4` expression is shown in Figure 2.20b.

Script 2.4.3

```
DEF ColRow (n::IsIntPos) = (INSR:RIGHT ~ #:n):col;

ColRow:4
```

Gable A simplified temple's `Gable` is defined here. Later on it will be positioned on top of first and last column rows. This is not historically correct, but helps to give a "temple" aspect to our simplified model. The shape chosen for the gable is that of a vertical solid triangle. Such geometric model can be generated in several ways. We choose one geometric construction which uses only geometric operators that we already encountered in this chapter. In particular we generate the `Gable` of Script 2.4.4 by `JOIN` of `basis` and `apex` of the solid triangle. Notice that `apex` is given as a 3D segment in its proper position.

Script 2.4.4 (Gable)

```
DEF Gable (radius,h::IsReal; n::IsInt) = JOIN:<basis, apex>
WHERE
   basis = (EMBED:1 ~ CUBOID):<l, w>,
   apex = MKPOL:<<<l/2,0,h/2>,<l/2,w,h/2>>,<<1,2>>,<<1>>>,
   l = 3*n*(2*w),
   w = 1.2*radius
END;

DEF ColRowAndGable = ColRow:4 TOP Gable:<1,12,4>;
```

Column row and gable The location of the gable over the row of four columns is produced by using the relative positioning operator `TOP`. The result of the evaluation of the `ColRowAndGable` symbol, defined in Script 2.4.4, is shown in Figure 2.20c.

First temple version At this point we are able to generate the first quite complete version of the temple model, by aggregation of four rows of columns without gable with two `ColRowAndGable` positioned at model extremes in the direction of second coordinate axis.

For this purpose we use, in Script 2.4.5, the predefined function `STRUCT`, which allows for hierarchical *aggregation* of sub-assemblies. In particular we catenate three partial sequences: (a) a first `ColRowAndGable` instance, together with the translation function which is applied to the subsequent column rows; (b) four catenated pairs `<`*row*`,` *translation*`>`; and (c) the final `ColRowAndGable` instance. The actual mechanism of

composition of sub-assemblies given in local coordinate frames by means of intermixed affine transformations will be discussed in Chapter 8.

Script 2.4.5

```
DEF myColRow = ColRow:4

DEF Temple1 = (STRUCT ~ CAT):<
   <ColRowAndGable, T:2:6>,##:4:<myColRow, T:2:6>,<ColRowAndGable> >
```

In Figure 2.21 we can see a projection of the geometric model generated by evaluation of the symbol `Temple1`.

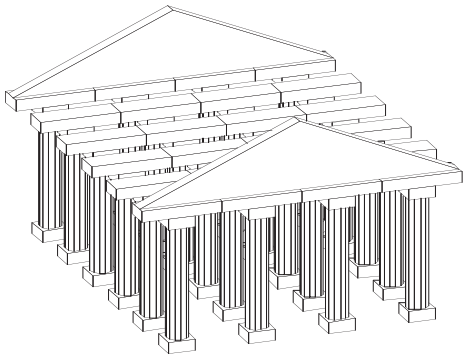

Figure 2.21 Dimetric projection of the polyhedral complex `Temple1`

Secondary beams The geometric model of the grid of secondary beams is defined in a very compact way by making use of the Cartesian product operator uniquely provided [BFPP93, PFP96] by the PLaSM language.

Depending on the regularity of repetition of secondary beams in the three coordinate directions, we can build three 1D polyhedral complexes, orderly named `Xsizes`, `Ysizes` and `Zsizes` in Script 2.4.6, and compute their Cartesian product, so generating a lattice of 3D parallelepipeds.

The 1D polyhedral arguments are generated by applying the primitive operator `QUOTE` to three non-empty sequences of non-zero reals. In such sequences, positive numbers correspond to the material cells, whereas negative numbers correspond to the empty space between cells. Notice that negative parameters, to be interpreted as *displacements*, are also used here to locate the complex in the coordinate frame of the temple.

Final model The complete geometric model associated in Script 2.4.6 to the `FinalTemple` symbol is just the aggregation of `Temple1` and `SecondaryBeams` in the same reference frame. In Figure 2.22. three views of the VRML file generated by PLaSM are reported. It would also be possible to easily apply some texture mapping on such a generated model, as discussed in Chapter 10.

Script 2.4.6

```
DEF SecondaryBeams = Xsizes * Ysizes * Zsizes
WHERE
   Xsizes = QUOTE:( ##:14:<0.6,-1.2> ),
   Ysizes = QUOTE:( AL:< -0.7, ##:5:<-1,5> > ),
   Zsizes = QUOTE:< -13,0.6 >
END;

DEF FinalTemple = STRUCT: < Temple1, SecondaryBeams >
```

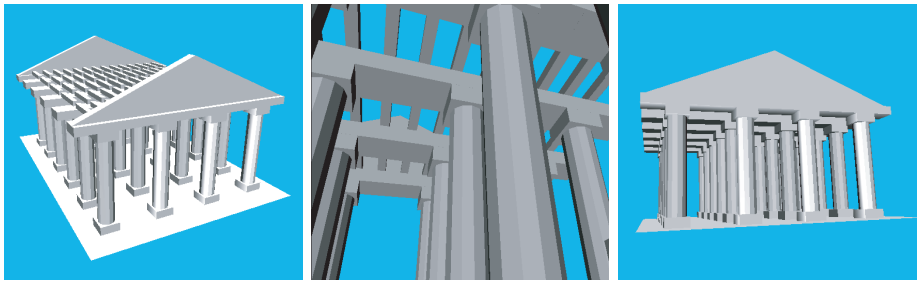

Figure 2.22 Three views from the VRML file of the temple generated by PLaSM