**SVG** The *Scalable Vector Graphics* (SVG) is the ISO draft proposal developed by W3C for 2D vector graphics on the web. The exporting into the `out.svg` file of the geometric value of the 2D PLaSM object named `out`, assuming a drawing area with 15 centimeters of width, is obtained by

```
svg: out: 15: 'out.svg';
```

### Saving/restoring geometry

A very useful feature of the current PLaSM version is the ability to save and restore geometric objects into/from external files by using a XML coding. The XML (Extensible Markup Language), can be considered the best method known today for putting structured data in a text file. It is a subset of the text processing international standard SGML (Standard Generalized Markup Language) (ISO 8879), specialized for use on the World-Wide Web.

Thus, large and computing intensive models can be generated only once, stored in external files and directly restored in memory in subsequent work sessions with PLaSM. Even more, the XML coding can be very useful when storing geometric objects in a database or when exchanging geometries across a computer network, within a collaborative and spatially distributed design environment.

The syntax to save/restore the geometric value associated to the `object` symbol is

```
DEF object = ...  ;
save: object:'path/object.xml';
```

in the saving session, whereas in the restoring session the saved `object` must be defined as:

```
DEF object = open:'path/object.xml';
```

## 1.3 Programming at Function Level

PLaSM can be considered a geometry-oriented extension of a subset of the FL language [BWW90, BWW+89], which is a pure functional language based on combinatorial logic. In particular, the FL language makes use of both pre-defined and user-defined *combinators*, i.e. higher-order functions which are applied to functions to produce new functions. The small but very significant FL subset which is used as the base environment of PLaSM is summarized in this section.

Notice that here and in the remainder of this book the infix symbol $\equiv$ is normally used to tell the reader that the *expression* on its left side evaluates to the *value* on its right side. Sometimes this symbol is also used to denote an equivalence between syntactical forms.

### *1.3.1 Elements of* FL *syntax*

Primitive FL *objects* are characters, numbers and truth values. Primitive objects, functions, applications and sequences are *expressions*. *Sequences* are expressions separated by commas and contained within a pair of angle brackets:

```
<5, fun>
```

An *application* expression `exp1:exp2` applies the *function* resulting from the evaluation of `exp1` on the *argument* resulting from the evaluation of `exp2`. Notice that binary functions can also be used in infix form:

```
1 + 3 ≡ +:<1,3> ≡ 4
```

Application associates to left, i.e. a sequence of repeated applications is evaluated from left to right. Notice that this is only possible if all the applications, but possibly the last one, generate a new function to be applied to the next argument:

```
f:g:h ≡ (f:g):h
```

Application binds stronger than composition, i.e. applications are evaluated first before compositions, as is shown in the following example:

```
f:g ~ h ≡ (f:g) ~ h
```

### *1.3.2 Combining forms and functions*

The function level approach to programming of FL emphasizes the definition of new functions by combining existing functions in various ways. The result of this approach is a programming style based on function-valued expressions. Some more important FL *combining forms* and functions follow.

**Construction** The combining form `CONS` allows application of a sequence of functions to an argument producing the sequence of applications:

$$\texttt{CONS:<f}_1\texttt{,...,f}_n\texttt{>:x} \equiv \texttt{[f}_1\texttt{,...,f}_n\texttt{]:x} \equiv \texttt{<f}_1\texttt{:x,...,f}_n\texttt{:x>}$$

A `CONS`ed sequence of functions is a sort of *vector function*, that can be composed with other functions and that can be applied to data. E.g. `cons:<+,->`, written also `[+,-]`, when applied to the argument `<3,2>` returns the sequence of applications

```
[+,-]:<3,2> ≡ <+:<3,2>,-:<3,2>> ≡ <5,1>
```

**Apply-to-all** The combining form `AA` has a symmetric effect, i.e. it applies a function to a sequence of arguments giving a sequence of applications

$$\texttt{AA:f:<x}_1\texttt{,...,x}_n\texttt{>} \equiv \texttt{<f:x}_1\texttt{,...,f:x}_n\texttt{>}$$

For example, we may apply the `SIN` function to all the elements of a list of numeric expressions:

```
AA:SIN:< 0, PI/3, PI/6, PI/2 >
   ≡ < SIN:0, SIN:(PI/3), SIN:(PI/6), SIN:(PI/2) >
   ≡ < 0 , 0.8660254037844382 , 0.49999999999999956 , 1.0 >;
```

The reader should notice that numeric computations often introduce round-off and approximation errors. Just remember that $\pi$ is an irrational number and cannot be represented exactly by using finite precision arithmetic. Also, functions like `SIN` are computed by using some truncated series expansion.

**Identity** The function ID returns its argument unchanged

```
ID:x ≡ x
```

In other words, the application of the identity function to *any* argument, gives back the same argument:

```
ID:0.5 ≡ 0.5
ID:SIN ≡ SIN
ID:SIN:0 ≡ SIN:0 ≡ 0
ID:'out.wrl' ≡ 'out.wrl'
```

**Constant** The combining form K is evaluated as follows, for whatever $x_2$:

```
K:x1:x2 ≡ x1;
```

In other words, the first application returns a constant function of value $x_1$, i.e. such that when applied to *any* argument $x_2$, *always* returns $x_1$. Some concrete examples follow:

```
K:0.5 ≡ Anonymous-Function
K:0.5:10 ≡ 0.5
K:0.5:100 ≡ 0.5
K:0.5:SIN ≡ 0.5
```

**Composition** The binary composition of functions, denoted by the symbol "∼", is defined in the standard mathematical way:

```
(f ∼ g):x ≡ f:(g:x)
```

$n$-ary composition of functions is also allowed:

```
COMP:< f, g, h >:x ≡ (f ∼ g ∼ h):x ≡ f:(g:(h:x))
```

In this case we have, where PI, COS and ACOS are the PLaSM denotations for $\pi$, cos and arccos, respectively:

```
(ACOS ∼ COS):PI ≡ ACOS:(COS:PI) ≡ ACOS:-1 ≡ 3.141592653589793
(COS ∼ ACOS):-1 ≡ COS:(ACOS:-1) ≡ COS:3.141592653589793 ≡ -1
(ACOS ∼ COS ∼ ACOS):-1 ≡ (ACOS:(COS:(ACOS:-1))) ≡ 3.141592653589793
```

**Conditional** The conditional form IF:< p, f, g > is evaluated as follows:

```
IF:< p, f, g >:x ≡ f:x   if  p:x ≡ TRUE
                 ≡ g:x   if  p:x ≡ FALSE
```

Notice that both the predicate[9] p, as well as f and g, to be alternatively executed depending on the truth value of the expression p:x, must be all *functions*. E.g., we have:

```
IF:< IsIntPos, K:True, K:False >:1000 ≡ True
IF:< IsIntPos, K:True, K:False >:-1000 ≡ False
```

where IsIntPos is a predefined predicate that returns True when applied to some positive integer.

---

[9] A *predicate* is a function p : Dom → { True, False }, where Dom is any set. Both True and False are called *truth values*, and also *Boolean* values.

**Insert Right/Left** The combining forms INSR and INSL allow the user to apply a *binary* function f, with signature[10] f : $D \times D \rightarrow D$, on a sequence of arguments of *any* length $n$. Notice that in the right-hand expressions below, f is applied to a *pair* of arguments:

INSR:f:< $x_1$, $x_2$, ... , $x_n$ > ≡ f:< $x_1$, INSR:f:< $x_2$, ... , $x_n$ > >
INSL:f:< $x_1$, ... , $x_{n-1}$, $x_n$ > ≡ f:< INSL:f:< $x_1$, ... , $x_{n-1}$ >, $x_n$ >,

where INSR:f:<x> ≡ INSL:f:<x> ≡ x.

An interesting example of use of the INSL combinator is given below, where the function bigger : Num × Num → Num is defined, being the syntax of definitions explained in detail in the next section. Notice that IsNum is a predicate used to check at run-time if the arguments of function application are of the correct type. The bigger function returns the one of its *two* arguments with maximum value; the biggest function does the same from a list of arguments of *arbitrary length*:

```
DEF bigger  (a,b::IsNum) = IF:< GT:a, K:b, K:a >:b;
DEF biggest = INSL:bigger;

bigger:<-10, 0> ≡ 0
biggest:<-10, 0, -100, 4, 22, -3, 88, 11 > ≡ 88
```

**Catenate** The CAT function appends any number of input sequences creating a single output sequence:

```
CAT:<<a,b,c>,<d,e>,...,<x,y,z>> ≡ <a,b,c,d,e,...,x,y,z>
```

A pair of concrete examples of how the CAT function is used follows. The second one is quite interesting: it gives a *filter* function used to select the non-negative elements of a number sequence:

```
CAT:<<1,2,3>,<11,12,13>,<21,22,23>> ≡ <1,2,3,11,12,13,21,22,23>

(CAT ~ AA:(IF:< LT:0, K:<>, [ID] >)):< -101, 23, -37.02, 0.1, 84 >
≡ CAT:<<>, <23>, <>, <0.1>, <84>>
≡ <23, 0.1, 84>
```

It may be very useful to *abstract* a filter function with respect to a generic predicate and to a generic argument sequence, by giving a function *definition*:

---

**Script 1.3.1 (Filter function)**

```
DEF filter (predicate::IsFun) (sequence::IsSeq) =
   (CAT ~ AA:(IF:< predicate, K:<>, [ID] >)):sequence;
```

---

Two examples of application of the filter function to actual arguments follow. Notice that the two applications respectively return the positive and the negative elements of the input sequence. Remember that a sequence of applications is *left-associative*:

---

[10] The *signature* of a function $f$ from a *domain* $A$ to a *codomain* $B$ is the ordered pair of sets $(A, B)$. It is normally associated to $f$ by writing $f : A \rightarrow B$.

```
filter:(LE:0):< -101, 23, 0, -37.02, 0.1, 84 >
   ≡ (filter:(LE:0)):< -101, 23, 0, -37.02, 0.1, 84 >
   ≡ Anonymous-Function:< -101, 23, 0, -37.02, 0.1, 84 >
   ≡ <23, 0.1, 84>

filter:(GE:0):< -101, 23, 0, -37.02, 0.1, 84 >
   ≡ (filter:(GE:0)):< -101, 23, 0, -37.02, 0.1, 84 >
   ≡ Anonymous-Function:< -101, 23, 0, -37.02, 0.1, 84 >
   ≡ <-101, -37.02>
```

**Distribute Right/Left** The functions DISTR and DISTL are defined as:

```
DISTR:<<a,b,c>, x> ≡ <<a,x>, <b,x>, <c,x>>
DISTL:<x, <a,b,c>> ≡ <<x,a>, <x,b>, <x,c>>
```

They accordingly transform a *pair*, constituted by an arbitrary expression x and by an arbitrary sequence, into a *sequence of pairs.*

Let us give an example of use. The Euler number $e$ is defined as the sum of a series of numbers. In particular:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!} + \cdots$$

We compute an *approximation* of $e$, named **euler**, as the sum of the first 21 terms of the series above. The definition of the factorial function **fact** is given in Script 2.1.6. Notice that the + operator may be applied to a sequence of numeric arguments and remember that $0! = 1$.

```
DEF euler = (+ ~ AA:/ ~ DISTL):< 1.0, AA:fact:(0..20) >;

euler ≡ (+ ~ AA:/ ~ DISTL):< 1.0, AA:fact:(0..20) >
   ≡ (+ ~ AA:/ ~ DISTL):< 1.0, AA:fact:< 0, 1, 2, ... , 8, 9 > >
   ≡ (+ ~ AA:/ ~ DISTL):< 1.0, < fact:0, fact:1, ... , fact:8, fact:9 > >
   ≡ (+ ~ AA:/ ~ DISTL):< 1.0, < 1, 1, 2, 6, ... , 40320, 362880 > >
   ≡ (+ ~ AA:/): (DISTL:< 1.0, < 1, 1, 2, 6, ... , 40320, 362880 > >)
   ≡ (+ :(AA:/: < <1.0, 1>, <1.0, 1>, ... , <1.0, 40320>, <1.0, 362880> >))
   ≡ + : < 1.0/1, 1.0/1, 1.0/2, 1.0/6, ... , 1.0/40320, 1/362880 >
   ≡ 2.7182815255731922
```

Above we have seen our first important example of FL computation as a sequence of expression transformations using the defining rules of the combinators. The order of transformations is induced by the parenthesis included into an expression. The sub-expressions nested more deeply are transformed first.

A simpler and more elegant implementation of the Euler number is given in Script 1.3.2, where C is the currifier combinator discussed in Section 1.4.3.

---

**Script 1.3.2 (Euler number)**

```
DEF euler = (+ ~ AA:(C:/:1.0 ~ fact)):(0..20)
```

---

**Example 1.3.1 (Conditional operator)**
As we have seen, the conditional form `IF:<p,f,g>:data` has the following semantic: "IF the predicate `p` applied to `data` is true, THEN apply `f` to `data`; ELSE apply `g` to `data`". This construct is very useful when it is necessary to apply different actions to input data depending on the value of some predicate evaluated on them, and is probably more "natural" than the conditional statements available in other languages.

From a syntactical viewpoint, notice that the `IF` operator is a higher-order function that *must* be applied to a *triplet of functions* in order to return a function which is in turn applied to the input data.

Such a syntax and semantics of the `IF` operator can be demonstrated by the following code, where a string is generated depending on the truth value of a simple predicate. The reader should notice that the result of evaluating both the expressions `K:'True'` and `K:'False'` is a (constant) function.

```
IF:< EQ, K:'True', K:'False' >:<10, 20> ≡ 'False'
IF:< EQ, K:'True', K:'False' >:<20, 20> ≡ 'True'
```

## 1.4 Basics of PLaSM programming

The PLaSM language was designed to introduce a well-founded programming approach to *generative modeling*, where geometric objects are generated by invoking the generating functions with *geometric expressions* as actual parameters. This is achieved by allowing for a sort of *geometric calculus* over embedded polyhedra. For this purpose the language contains in its kernel a *dimension-independent* approach to the representation of geometric data structures and algorithms.

The programming approach to geometric design enforced by PLaSM makes it possible to manage a sort of extended *variational geometry* [LG82], where classes of objects with varying topology and shape are parametrized by some language function and handled and combined as a whole. The language can be considered a *geometry-oriented extension* of a quite small subset of the functional language FL, where the *validity* of geometry is always syntactically guaranteed. In other words it is guaranteed that any well-formed language expression which generates a polyhedrally-typed data object always corresponds to some valid internal data structure.

Such a functional and dimension-independent approach to geometric design achieves a *representation domain* broader than usually done by standard solid modelers, so that points, wire-frames, surfaces, solids and even higher dimensional manifolds can be suitably combined or blended altogether.

### *1.4.1 Expressions*

The syntax of PLaSM is very similar to the syntax of FL (see Section 1.3.1). For the reader experienced in FL we may say that the differences mainly concern the meaning of few symbols (composition and constant operators) and the lack of pattern matching in the definition of type predicates. In particular, the PLaSM language:

1. uses a case-insensitive alphabet;
2. allows for overloading of some (pre-defined) operators;
3. evaluates expressions which return a polyhedral complex as value;