SetlX-Quickreference

`asdf` Element-Placeholder
`…` continual Element-List
`asdf` optional Element

`asdf` exemplary Implementation

variables	`a;`	→ Name has to start with a lowercase letter. Apart from that it can contain any letters, numbers and _ → They can't get statically typed
assignment	`a := b;`	→ if the number contains **.** it will be automatically recognized as a real number
strings	`"asdf"`	→ can be concatenated via + → the modifiers for lists (see below) are also useable for them
literal strings	`'asdf'`	→ turns of all processing, e.g. `'\n'` will be saved as `\n` in characters, rather than getting processed into a newline-character
undefined Ω	`om`	
placeholder	`_`	→ use it if you have to provide a variable for a call because of its syntax but actually don't need this variable

comments	`// asdf`
	`/* multiple-line`
	`asdf */`	

output	`print(asdf, asdf, …);`	→ you can insert expressions like $3+2$ which will be evaluated when printing the output-string
input	`a := read("asdf");`	→ Prints the argument into the prompt and returns the user-input

rational numbers:

→ they work without overflows and in theory indefinitely accurate because they are stored as fractions
→ `1/3 + 1/2` would return `5/6`

different types of functions:

procedure	`asdf := procedure(v1, v2, …) {`
	`...`
	`return r;`
	`};`	

cached / memorized procedure	`asdf := cachedProcedure(v1, …) {`
	`...`
	`return r;`
	`};`
		→ speeds up computation by saving results of the function in-memory in a lookup-table
		→ only allowed for $pure\ functions$
		a pure functions **always** returns the same output if it is called with the same input

closure	`asdf := closure(v1, v2, …) {`
	`...`
	`r := extVar * 2;`
	`return r;`
	`};`
		→ works like a procedure
		→ additionally you are able to access variables which are defined outside the function

mathematical function	`f := x |-> def;`
	`f := x |-> 1.0/(1+x);`
	`a := f(2);` *a = 1/3*
		→ equals to $f\colon x \to def$ which equals to $f(x) = expr$
		→ useable via `a := f(n);`

lambda closure-definition	`f := in |=> expression;` `f := [in1, in2] |=> expression;`	→ equivalent to `f := closure(in) { return expression; }` → equivalent to `f := closure(in1, in2) { return expression; }`
call	`f(arg);`	

control structures:

If-branching	`if (test1) {`
	`body1;`
	`} else if (test2) {`
	`body2;`
	`} else {`
	`body3;`
	`}`
		→ the brackets are always necessary!

switch-branching	`switch {`
	`case test1 : body1;`
	`case test2 : body2;`
		→ only one body gets executed

	... default : body3; }	

while-loop	while (test) {
	body;
	}	

for-loop	for (i in m) {
	body;
	}
		→ iterates through the elements of the set/list like `m[i]`

abort one iteration	continue;	
abort the loop completely	break;	

predefined real ("reelle") functions:

trigonometric	sin(x)		
	asin(x)	equals to $sin^{-1}(x)$	
	sinh(x)	sinus hyperbolises	
	cos(x)		
	acos(x)	equals to $cos^{-1}(x)$	
	cosh(x)	cosine hyperbolises	
	tan(x)		
	atan(x)	equals to $tan^{-1}(x)$	
	tanh(x)	tangent hyperbolises	
exponential	exp(x)		equals to e^x
	x ** a		equals to x^a
logarithmic	log(x)		equals to $\ln(x)$ (natural logarithmic)
	log10(x)		equals to $log_{10}(x)$
absolute value	abs(x)		equals to $\|x\|$
sign	signum(x)		
square root	sqrt(x)		
3rd-root	cbrt(x)		
round up	ceil(x)		rounds up to the next integral number
round down	floor(x)		rounds down to the next integral number
round to nearest	round(x)		also known in german as "kaufmännisches Runden"

sets:

definition by enumeration	{start .. stop}	→ equals to $\{x \in \mathbb{Z} \mid start \leq x \wedge x \leq stop\}$ → any element is only contained once and elements are ordered by their value
definition by step-enumeration	{start, second .. stop}	→ equals to $\{start + n * step \mid n \in \mathbb{N}_0 \wedge start + n * step \leq stop\}$ with $step = second - start$
definition by iterators	{definition : ranges} {n * m : n in {2..10}, m in {2..10}};	→ the set then contains the non-trivial Solutions for the condition which meet the ranges for their elements → eqauls to $\{n * m \mid n \in \mathbb{N} \wedge 2 \leq n \wedge n \leq 10 \wedge 2 \leq m \wedge m \leq 10\} =$ $\{4, 6, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 25, 27, 28, 30, 32, 35, 36, 40, 42, 45, 48, 49, 50, 54, 56, 60, 63, 64, 70, 72, 80, 81, 90, 100\}$
additionally: selection	{definition | condition}	→ only elements which fulfill the additional condition are added to the set
summation	+/m	→ returns the sum of all elements in the set M

product	`*/m`	→ returns the product of all elements in the set M
element-count	`#(m)`	→ returns the number of elements contained in the set
union $a \cup b$	`a + b`	
intersection $a \cap b$	`a * b`	
difference a / b	`a - b`	
power 2^a	`2 ** a`	
Cartesian product $A \times B$	`a >< b`	
powerset	`pow(m)`	→ returns the set which contains all possible subsets of `m`

is a subset $a \subseteq b$	`a <= b`	
is an element $a \in M$	`a in m`	
get the element with the highest value	`max(m)`	
get the element with the lowest value	`min(m)`	
take a (not pre-defined) element	`from(m)`	→ Returns a kind of random element from the set: At first, you don't know which one it will be. But if you run the program again, the order of the returned elements is exactly the same. → Removes the element from the set!
get a (not pre-defined) element	`arb(m)`	→ works like `from`, but doesn't remove the element from the set
get a (pseudo-) random element	`rnd(m)` `rnd(5)`	→ bad for debugging → computes a random natural number less or equal then 5, via the implicated call `rnd([1..5])`

general tuples / lists:

→ They can be defined and used just like sets.
→ {} in the definition then become []
they are definable through enumeration, Iterators and selection
→ e.g. a pair $\langle x, y \rangle$ is definable through `[x, y]`
→ differences to sets: elements are not ordered and can be contained multiple times

reverse it	`reverse(l)`	
sort it	`sort(l)`	→ sorts the elements in the list in ascending order of their values
element-reference	`m[i]`	→ returns the i^{th} element out of the set (ordered ascending by value) `m[-1]` returns the last, `m[-2]` the pre-last element and so on → the counting of elements starts at 1!
Subset-reference	`m[a..b];`	→ returns the sub-set of `m` starting at index `a` and ending on index `b` → one of the limits can be omitted
append it to itself	`n * l`	
concatenate them as a string	`join(l, s)` `join([1,2,3, "*")` 🖹 *"1*2*3"*	→ converts the elements of `l` into strings and concatenates them using the string `s` as a separator

relations:

definition of relations	`{[pair-Def] : Condition}` `{[n, n**2] : n in {1..10}};`	→ eqauls to the Function $x \to x^2$ on the set $\{n \in \mathbb{N} \mid 1 \leq n \wedge n \leq 10\}$ 🖹 {[1, 1], *[2, 4], [3, 9], [4, 16], [5, 25], [6, 36], [7, 49], [8, 64], [9, 81], [10, 100]}*
domain	`domain(m)`	
range	`range(m)`	

logical expressions:

boolean test-operators	`==`	
	`!=`	

	`<`	
	`<=`	also checks ⊆ at sets
	`>`	
	`>=`	
	`in`	
test-junctures	`!` equals to ¬ strongest bind `&&` equals to ∧ `||` equals to ∨ weakest bind	
all-quantifier	`forall(x in m | cond)`	→ equal to $\forall x \in m : cond$
exists-quantifier	`exists(x in m | cond)`	→ equal to $\exists x \in m : cond$
implication	`a => b`	
equivalence	`a <==> b`	
antivalence	`a <!=> b`	
convert strings	`eval(expr)`	→ the string expr has to be a string which can be parsed as a SetlX-Expression → the result of the evaluation of the represented expressions is them returned

terms:

symbolic Programs = programs/procedures which take functions (contained in strings) and manipulate them

a program which takes strings like "x*3" and finds the derivate of them

function-symbols	`A`	→ the name has to start with a uppercase letter. Apart from that it can contain any letters, numbers and _
	`^Asdf`	→ used internally to define operators like +
	`@asdf`	→ used to case (lowercase) built-in functions into a function-symbol
terms	`funcSymbol(value1, value2, ...)` `Adresse("Musterstr 1", 23456, "Musterstadt")`	→ Terms are never evaluated! They are only used to store data.
undefined	`Nil();`	
get the function-symbol	`fct(Asdf(value))`	
get the values/ argument-list	`args(Asdf(value))`	

matching:

match-branches	`match (Term0) {`
	`  case pattern1 : body1;`
	`  case pattern2 : body2;`
	`  ...`
	`  default : body3;`
	`}`
	`match (Pl(3, 5)) {`
	`  case Pl(t1, t2) : return"$t1+t2$ +";`
	`  case Mi(t1, t2) : return"$t1-t2$ -";`
	`}`
	`→ "8 via +"`
		→ Instead of a Term, a String or a List can also be matched
		→ The patterns have to contain placeholders for variables
		→ At the evaluation, SetlX tries to insert the values of `Term0` into these placeholders to create a (new) term which is equal to `Term0`.
		If this succeeds, the corresponding body gets executed (with the specified variables filled accordingly)..
		If not, the default-body gets executed.

vectors:

definition	`v1 := la_vector([1, 1/2, 1/3]);`
	`v2 := <<1 1/2 1/3>>;`
	`→ v1 == v2 == <<1.0 0.5 0.3333333333333>>`
	`note: 1/3 is only printed rounded`
		→ only real valued vectors, but with any dimension, are supported
		→ all vectors are column-vectors, via concept

accessor	`v[i]`	→ gives the $i\text{-}th$ element of the vector back

addition / subtraction	`v := v1 + v2;`
	`v := v1 - v2;`
	`v += v1;`
	`v -= v2;`	

scalar *multiplication*	`v := <<1 1/2 1/3>> * (1/2);` `v *= (1/3);`	→ * is commutative
scalar product	`v := v1 * v2;`	
cross product	`v := v1 >< v2;`	→ only defined for three-dimensional products

matrices:

definition	`m1 := la_matrix( [[1,2],[3,4]] );`
	`m2 := << <<1 2>> <<3 4>> >>;`
	`→ m1 == m2 == << <<1.0 2.0>> <<3.0 4.0>> >>`
		→ only real valued matrices are supported

transforming vectors	`v := <<1 2 3>>;` `m1 := la_matrix(v);`	→ returns an $n \times 1$-matrix → the column-vector gets transformed into a one-row-matrix

addition / subtraction	`m := m1 + m2;`
	`m := m1 - m2;`
	`m += m1;`
	`m -= m2;`	

scalar *multiplication*	`m := << <<1 2>> <<3 4>> >> * (3);` `m *= (1/3);`	→ * is commutative
matrix multiplication	`a * b;` `a * v;`	→ only possible if a is a $m \times n$-matrix and b is a $n \times k$-matrix (returning a $n \times k$-matrix) → if v is a n-dimensional vector, it automatically is interpreted as an nx1-matrix and the result will be converted to an m-dimensional vector
exponentiation	`a ** 2;`	→ only possible for square matrices
inverse	`a ** -1;`	→ only possible for non-singular matrices
transposing	`a!;`	
Dimension m	`#a;`	
Dimension n	`#a[1];`	
Determinant	`la_det(a);`	→ the result might be a small non-zero value, even if the matrix is really singular (due to rounding errors)