MathMorphs: An Environment for Learning and Doing Math

Luciano Notarfrancesco and Leandro Caniglia

University of Buenos Aires

Facultad de Ciencias Exactas y Naturales

Mathematics Department

Introduction

Mathematics comes to life when mathematicians think about a theorem, read a definition, or express their points of view to others. This happens in classrooms and lectures; it's exciting, profound and beautiful. For centuries mathematicians have been improving the way they keep their notions and ideas on paper. Style, notations and terminology have been carefully developed to the highest possible degree of clarity and concision. The result of such enormous effort is a broad and impressive literate repository, which provides help to the new generations that have to absorb this forever-increasing body of knowledge.

Still, writing and reading Mathematics is difficult. It demands a great amount of time and effort to translate a clear idea into formal terms. In the writing process, the author must squeeze an alive, ideal world on the inert face of the paper. It is hard, because the author must tell the reader about that world instead of showing it. On the other hand, the reader is supposed to recreate the tale into her own intellect. This two way process consists of first coding thoughts into a strict notation, and later read that austere formalism trying to rebuild the original ideas the formalism is based on.

The systematic storing and retrieval of cognition is not a simple task. In the meantime, the mathematical objects are reduced to frozen symbols that look the same as Mathematics, when in fact they are nothing but notation.

What if mathematicians had a place to keep all their living objects? Not a planar place, but a multidimensional one, with an unlimited capacity to hold things inside. A space with colors and movement. A site where definitions get expressed and instanced without suffering from any kind of hibernation. A space to materialize ideas letting them evolve into new suggestions to the observer; where the mathematical objects and their relations would coexist showing new relationships. A comfortable and well-equipped laboratory for mathematical exploration and experimentation. If such a place existed, then it would be a repository of Mathematics, and not a repository of *texts* about Mathematics.

The identification of mathematical objects with their textual (symbolic) description is so deeply rooted in our minds that the distinction we are trying to stress might look obscure. Just think about the difference between the music and the score. Since 1997, the MathMorphs project is the logical consequence of one simple fact: realizing that such a fertile world is possible in the universe of Squeak.

Paradoxically, the description of the MathMorphs work included in this chapter has been written on paper. The following sections attempt to put in

words and frozen illustrations the kind of motivation we have experienced in classroom.

All the projects were worked out and implemented by the students. They are the authors of all this work and we want to thank them for all the effort and enthusiasm spent in showing us how to get fun with MathMorphs.

Acknowledgments

MathMorphs is the result of the efforts made by several people. Gerardo Richarte has been very creative and is always eager to think about anything. He invented the "type on air" magic and took part in many projects. Daniel Vaisencher made smart contributions and suggestions about MorphicWrappers. Pablo Malavolta proposed visual effects and worked hard in Algebraic Geometry. Andres Valloud programmed the Function Plotters, and got involved with MathMorphs from the beginning. Ariel Pacetti devised useful improvements in the implementation of Algebraic Numbers. Alejandro Weil worked with Euclidean Geometry. Eric Maximiliano Guevara implemented Linear Recursive Sequences. Alejandro Tolomei worked on Random Variables and Statistics. Gabriel Stern taught us how to model Metric Spaces and continuous functions. Pablo Schmerkin introduced BiologyMorphs. Ariel Schwartzman worked on PhysicsMorphs integrating many MathMorphs tools. Francisco Garau shared his studies on TypeInference presenting interesting problems to the group and discussing about possible solutions. Valeria Murgia made lots of fruitful suggestions on how to model with Smalltalk. Mathematicians Jorge and Juan Guccione argued deeply about the theoretic aspects of the work. To all these people, our love and gratitude.

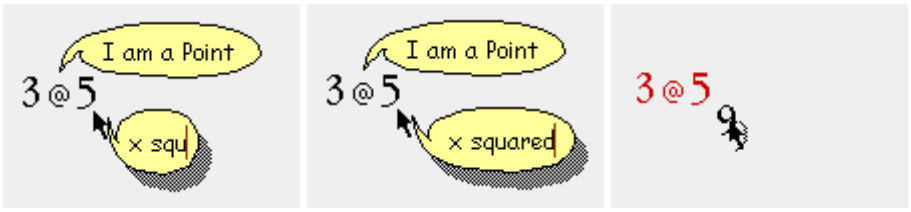
MorphicWrappers

Let us begin our MathMorphs tour describing our main tool to play with objects. A **MorphicWrapper** is a vehicle allowing any object to be included in a Morphic World. This is accomplished by usual techniques: a special kind of morph, the wrapper, acts as an invisible envelope surrounding the ordinary object with the behavior required by Morphic.

What is fun with MorphicWrappers is that they let you take advantage of the amazing potential that Morphic gives to all kind of users. The MorphicWrapper package also provides a set of tools that further extend the default user interface capabilities provided by Morphic.

CodeBalloons

Gerardo Richiarte first introduced the brilliant concept of “typing on air”. Soon, this feature became a distinctive aspect of the MorphicWrappers. A so-called **CodeBalloon** appear when you start typing "on air". You do not need workspaces any more. Simply type any Squeak expression on the fly, and a **CodeBalloon** will appear holding the expression as you type it in. When you hit the return key or when you click on the receiver, the expression is evaluated and the result is attached to your morphic hand.

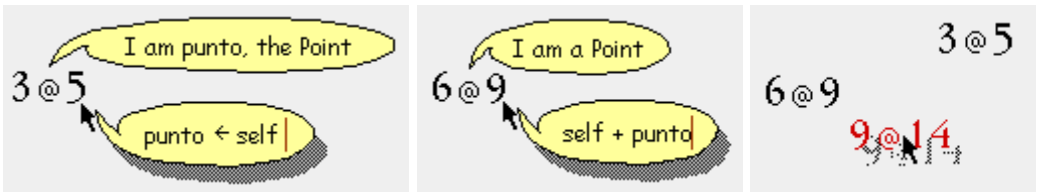


A CodeBalloon appears when you start typing

If the hand is focused on a **MorphicWrapper**, then the expression is evaluated in the context of the wrapped object. Thus, when writing the expression, you can use `self`, `super` and any of the instance or class variable names for that object.

Names

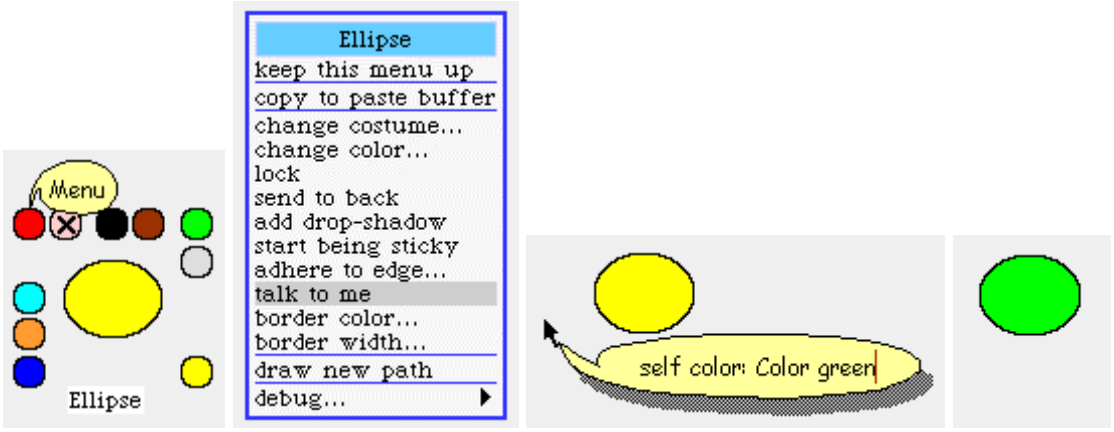
You can use **CodeBalloons** to name objects. These names are meaningful inside the world.



Naming objects with MorphicWrappers

Talking to normal morphs

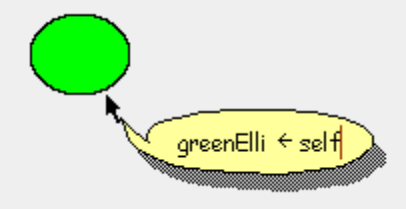
The MorphicWrappers let you talk to any morph using **CodeBalloons** and names. The red dot in the halo of any morph includes the 'talk to me' item.



Talking to ordinary morphs

You can use the 'talk to me' feature to name the morph in just the same way as with MorphicWrappers.

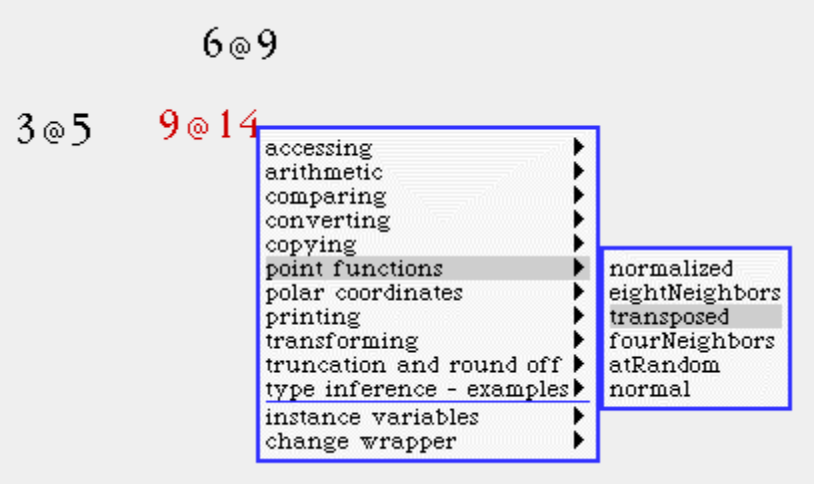
Of course, you can also type any expression when the morph is in focus. If the target morph does not eat keyboard events, then the expression will evaluate correctly using ‘self’ as the morph. You should use the ‘talk to me’ item, when the keystrokes are captured by the morph.



Naming an ellipse

Double click menus

Double clicking on a **MorphicWrapper** pops up a menu for the object being wrapped. This menu shows the object's protocol. The intended message is sent when you select it from the menu.



Unary protocol of Points

Typing history

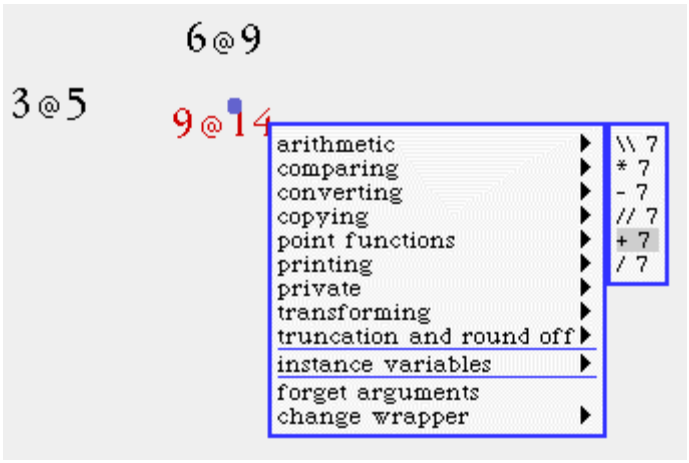
Each wrapper remembers the history of all expressions evaluated in its context. There is one expression history for each class of wrapped objects. The commands in the history can be accessed using the up and down arrows in the keyboard.

Arguments

The double click menu pops up the unary messages of the protocol when we have not included any argument. You can add arguments by dragging and dropping them on the receiver. Each argument appears as a **Satellite** flying around the object.

Dropped as argument, 7 is represented by a flying satellite (blue dot)

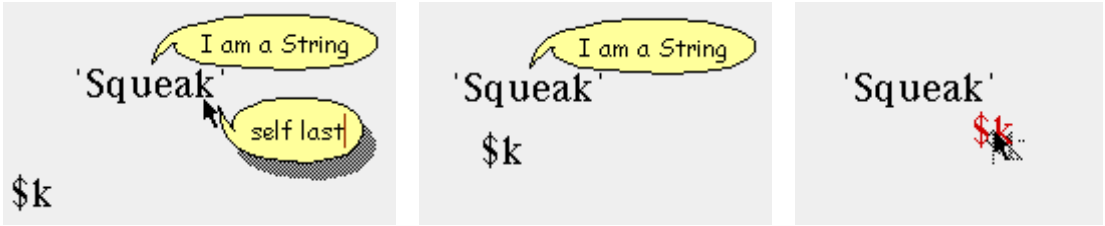
Now the double click menu shows all the messages with one argument. You can drop as many arguments as needed. If you want to get the arguments back, you can use the menu item called 'forget arguments'. Also, if you let them alone, the satellites will disappear after a few seconds.



Double click menu with one-argument messages

Identity

The MorphicWrappers support object identity. This means that no two wrappers will be present for the same object. As an example, suppose you have the string **'Squeak'** and the character **$k**. Then you ask **'Squeak'** for its last element. Since there is only one **$k** in the system, the answer will be the same **$k** you already have in the world. When you hit return (or click on **'Squeak'**), the **$k** character runs to your hand:



The identity principle is illustrated when the $k **character runs to assume its role as the answer of the message**

This behavior is a very good example of how Morphic makes many properties inherent to objects more apparent. You do not need to tell people about the identity principle; you let the objects play their roles in the show.

A **CarryingMorph** accomplishes the running effect. This invisible morph takes another morph as its **passenger** and **steps** to the target destination (in this case, the hand), carefully carrying its shipment. By changing the **drawOn:** method of the **CarryingMorph**, one could easily add visual effects that would take place during these short trips.

Another visual trick we use is to let the answer of a message *shake* when it is attached to the hand. Here we use a **ShakingMorph** that moves its target morph with decreasing strength each time it steps.

RequestBoxes

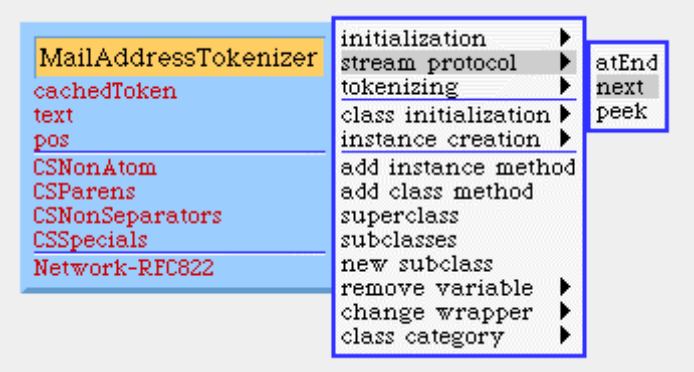
The use of **FillInTheBlank** becomes superfluous with the MorphicWrappers. Inside Morphic, we do not need to prompt for strings because we can prompt for any object. A **RequestBox** requests an object. When the user has the answer at hand, she drops it into the box. The **RequestBox** sends the answer back to the requesting object, and the process is done. You can cancel the request by throwing it to the trash.

RequestBoxes accept any object as input

Usually one responds to a **RequestBox** typing on air an expression that evaluates to the intended answer.

Programming with the MorphicWrappers

The MorphicWrapper package is a framework were more specialized wrappers can be added. With **ClassMorphicWrappers**, classes and metaclasses behave as morphs. As with any other object, to include an existing class in the world you simply write its name on air. The class pops up attached to the hand.



The double click menu exposes all methods

When viewed as morphs, classes show their structure: instance variable names, class variable names, pool dictionaries and category. The double click menu exposes the instance and class protocols. Other useful tools are also accessible from this menu.

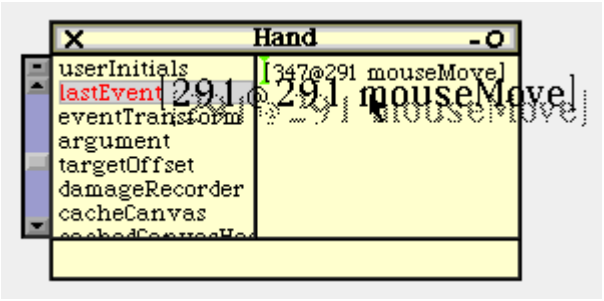
To edit a method, select it from the double click menu. A window for the method pops up.

Methods can be edited individually

Many operations are done by dragging and dropping. For example, instance variables, class variables, pool dictionaries and class categories are included in the class when a symbol (or string) is dropped on it. Lower case symbols are interpreted as instance variable names, while global dictionaries are interpreted as pools. If the symbol contains a dash and begins with an uppercase character, it is taken as the new class category. Other uppercase symbols are included as class variable names. The double click menu provides an optional way to change the class category; it is useful for category names not containing the dash character.

Dragging and dropping objects as morphs

When an object is dragged out of an **Inspector**, it is attached as a morph to the hand.



Dragged objects become MorphicWrappers

If the object being dragged is a morph, then the morph is attached to the hand. Otherwise, a **MorphicWrapper** on the object appears. The same conversion occurs when dragging classes or methods form a browser or a message list.

How to... A quick tour

task	how to	double click menu	pup-up menu	special keys
Evaluate an expression	Write the expression on air			Use Enter or Cr to accept
Evaluate an expression without attaching the answer to the hand	Write the expression on air. Be sure to end the expression with a period			Use Enter or Cr to accept
Name an object (works with most morphs)	Focus on the object and evaluate the expression: **name ← self**			
Name a morph (works with every morph)	Find the talk to me item and proceed as with normal objects		Use the red dot in the morph's halo	
Recall some expression	Focus the object and hit the up arrow on the keyboard			Up and down arrows
Send a message (I)	Use the name of the object and evaluate the expression associated to the message			
Send a message (II)	Focus the object and evaluate the expression associated to the message using **self** for the receiver. You may or may not include a leading **'self'**			
Send a message (III)		Find the message selector as an item in		

		the menu (see also how to add arguments)		
Send a message (IV), opening a debugger		Click on the message selector, an item in the menu, while holding down the Shift key		Shift key
Add an argument	Drop the argument on the wrapper			
Forget arguments	Wait, or use the double click menu, or...	... select the 'forget arguments' item		
Answer a **RequestBox**	Drop the answer on the box			
Cancel a **RequestBox**	Drop the box on the trash			
Delete a wrapper	Drop the wrapper on the trash, or hit backspace when the wrapper has the focus			Backspace
Get a class wrapper	Evaluate the expression consisting of the class name			
Edit a method	Evaluate the expression **ClassName | #selector**, or double click on the class wrapper	Find the selector under the category submenu		Save with Alt-s (Cmd-s in the Mac)
Create a new class	Get the superclass for the new class and...	... use the new subclass item		
Add an instance variable to a class	Drop the name of the variable on the class' wrapper			
Remove an instance variable from a class		Use the remove variable item		
Add a class variable to a class	Drop the name of the variable on the class' wrapper			
Add a pool dictionary to a class	Drop the name of the pool (or the pool itself) on the class' wrapper			
Change the class category	Drop the name of the new category, or...	... select the new category item		
Add a new		Select the		

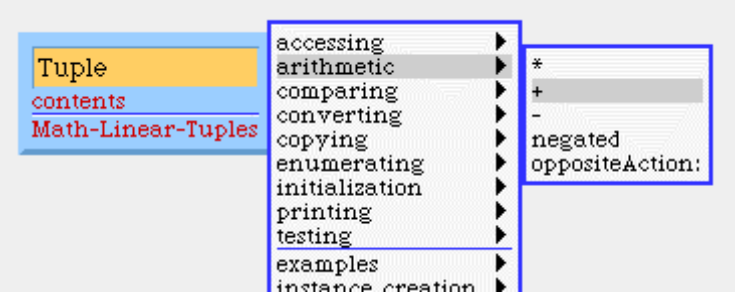
instance or class method (I)		action from the menu		
Add a new instance or class method (II)	Edit any other instance or class method, and change the selector before saving it			Save with Alt-s
Copy a method from one class to another	Drag and drop the method from the source class on the destination class			Shift while dropping. Save with Alt-s
Move a method from one class to another (removing it from the former)	Drag and drop the method from the source class on the destination class			Ctrl + Shift while dropping
Change the superclass of a class	Drop the class on the new superclass			Shift while dropping
Remove a method, change its category, run **prettyPrint** on it, see versions, senders, implementors, etc.	Edit the method and...		... select the proper item in the method's pop-up menu	
Include an object as a morph in the world	Drag the object from an Inspector, Browser or message list			

Linear Algebra

Expressiveness is a key aspect in MathMorphs' approach. A rich hierarchy of objects allows us to represent the involved entities in Squeak as they are in the mathematical world. The need for spurious conventions on the interpretation of data structures is eliminated. Linear algebra appears everywhere, and so it is a good point of departure to show how our ideas have been accomplished. Let's begin our journey through MathMorphs.

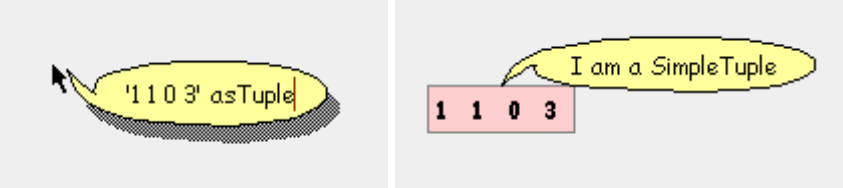
Tuples

The first linear objects we will consider are tuples. The protocol for these objects includes messages for accessing individual coordinates, for arithmetic operations as addition and scalar action, for testing whether the tuple is null, and some enumerative messages on the coordinates of the tuple.



Menu fragment showing the instance protocol of tuples

There is also a rich instance creation class protocol. For example, it is handy to convert a **String** into a **Tuple**. It is also convenient to supply a block and a range of indexes, so that the values of the block at the given indexes are collected to build the new tuple.

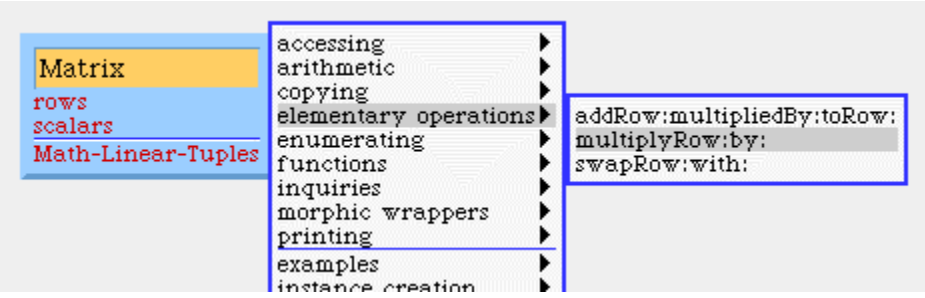


Using MorphicWrappers to work with tuples inside Morphic

A subclass of **Tuple** represents sparse, freely indexed tuples. These are appropriate for special purposes in which the indexes are not integers from 1 to n.

Matrixes

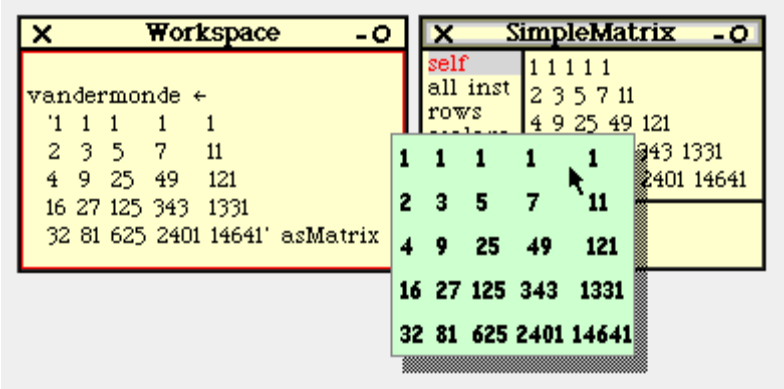
The next step in the process of building a linear algebra package is to represent matrixes. The instance protocol includes messages for accessing individual entries, doing arithmetic operations such as addition and matrix product as well as product by scalars and tuples. To access the entry at row **i** and column **j**, the **Point i@j** is used as an index. Elementary operations on rows are implemented as messages.



A fragment of the instance protocol of matrixes

Basic operations as transposing or the computation of the trace are also implemented in the instance protocol. On the contrary, more elaborated functions are delegated to external objects. There are **MatrixReducers** (described below), whose responsibility is to compute ranks, determinants,

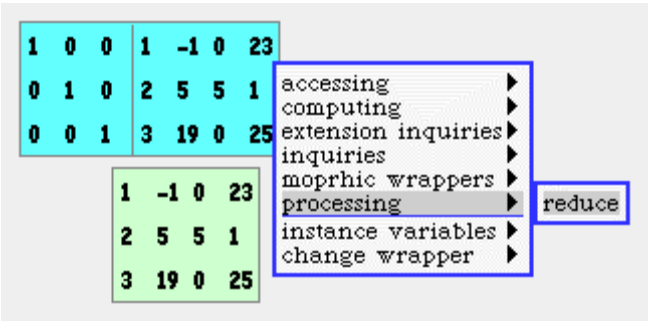
inverses, etc. These objects implement triangulation algorithms. Again, the class protocol includes many useful messages for instance creation.



Using MorphicWrappers to work with matrixes

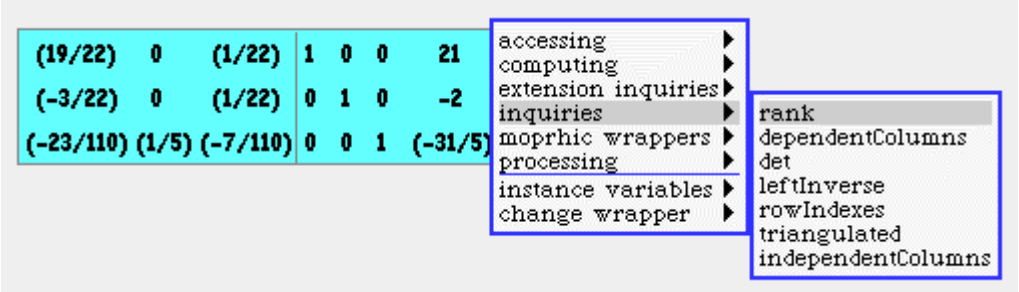
MatrixReducers

A design characteristic of MathMorphs is that algorithms are kept away from the input data. With this idea in mind, we are free to use the same algorithm with different species of inputs and to compare different algorithms when running on the same data. When an algorithm is instanced as an object by itself, one ends up with a machine holding a valuable knowledge about the theoretical aspects of the process and the special characteristics found in any particular run. We can ask the algorithm the time it has taken to run, the final state of any of its variables, etc. Implementations could include recording of special events, loop counters, history of execution, or playback options.



A MatrixReducer **before processing its input** Matrix

In this case, decoupling the algorithm from the objects it runs on has the beneficial consequence of keeping the implementation and the protocol of matrixes simple. Once the **MatrixReducer** has run on its input matrix, we can ask it about the rank, determinant, left inverse and any other information derived from its work.



The reducer, after processing its input, holds interesting information

Algebraic Ambients

Linear algebra theory does not require special hypothesis on the field of scalars. The theory is the same regardless of the nature of the coefficients. These may be rationals, reals, complexes, modular integers, algebraic numbers, etc. In MathMorphs we keep the same degree of generality, implementing all constructions independently from the field of scalars.

One could think that such degree of generality requires a complete implementation of rings and fields. While such kind of objects is interesting by itself, a high degree of dependence between these two frameworks would be contrary to our aims. Fortunately, very little information is needed to implement linear algebra definitions without losing generality. This information is just the zero and the unity of the field of scalars.

We use **AlgebraicAmbients** to keep independence from the field of scalars. An **AlgebraicAmbient** is an object that can answer the **unit** and **zero** elements of some prime field or ring. Since **zero** equals **unit - unit**, these objects only have to remember the unity as an instance variable.

As an example, let us consider the class **Tuple**. We want instance creation messages for the canonical vector e_i. When mathematicians say e_i, it is assumed that everybody knows the value of n. In other words, they are speaking about $e_{i,n}$. Also, and this is the interesting point, they are assuming that the field of scalars is implied or provided by the context. So, a complete instance creation method should look like this:

```
Tuple class | e: i dim: n scalars: scalars
```

where **scalars** is an **AlgebraicAmbient**. Since this object knows the unity and zero elements, the message can answer with the desired tuple.

The class **AlgebraicAmbient** also holds the collection of all available ambients (those that have been used so far). So, the message

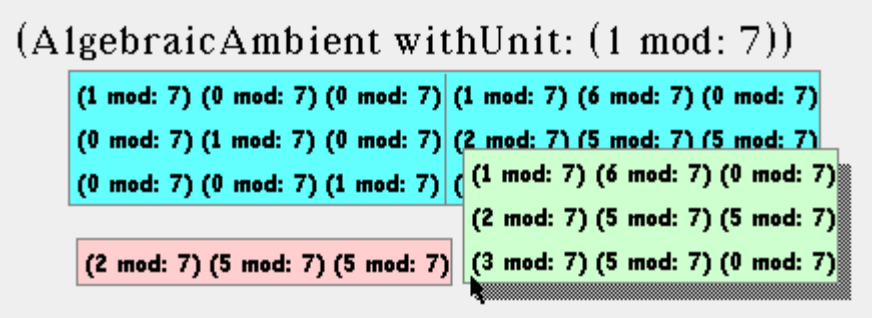
```
AlgebraicAmbient withUnit: unit
```

answers a previously created ambient with *unit* as the unity, or a newly created instance. Once we have the class, say, **IntegerMod5**, we can add a new ambient as follows:

```
AlgebraicAmbient withUnit: (1 mod: 5)
```

At initialization the class creates a "default" instance **withUnit: 1**, the **SmallInteger**. The class also keeps the last scalars used. When a specific scalar argument is not provided, the last scalars are used by default. The framework is unaffected when the scalars change. Moreover, different fields can coexist without having to take special care of them.

This design allows the same code to work regardless of the scalar system. Methods are written without any special considerations about the field of scalars, because they are based on mathematical definitions that do not impose specific restrictions. For instance, the matrix reduction algorithms are proven correct in *any* scalar field. These algorithms perform operations based on the properties of fields alone, without ever worrying about any field in particular.



The same code works with different scalar systems

As new numeric systems are added to the image, they can be immediately used as scalars in the package. Examples include real algebraic numbers, complex numbers, modular integers, finite fields, finite extensions of the rationals, etc.

LinearAmbients

We use **LinearAmbients** to define *abstract* linear spaces and perform functorial operations. These operations include basic ones such as tensor product, hom, dual and direct sum, etc; and any combination between them.

When we think about **Tuples**, **Matrixes**, **LinearForms**, **LinearEquations**, etc., we encounter subspaces and bases. A subspace has a basis, and we usually want to express a vector as a linear combination of the basis.

No matter what algorithm we employ, the fact is that all algorithms assume that we know how to write the coordinates of all involved vectors as tuples. In other words, each time we want to calculate the coordinates of a vector v in some basis B, we need to express the coordinates of the elements of B, and v, in terms of another basis.

The problem seems to be circular: to find the coordinates we must find some other coordinates first. However, it is not circular because all linear spaces are subspaces of a space with a canonical basis. Canonical bases are a clue to break down circularity.

Note that the notion of *canonical* is not mathematically defined. Nevertheless, a canonical basis has an intuitive property: it makes easy to find the coordinates of any vector. Examples of well-known canonical basis are:

1. Tuples: $e_1=(1, 0, 0, ...)$, $e_2=(0, 1, 0, ...)$, $e_3=(0, 0, 1, ...)$, ...
2. Matrixes: e_{11}, e_{12}, e_{13}, ..., e_{ij}, where e_{ij} is the matrix having **1** at **i@j** and zero everywhere else.
3. Single variable polynomials: 1, x, x^2, x^3, ...
4. Multivariate polynomials: 1, x, y, z, x^2, xy, xz, y^2, yz, z^2, ...

When we have a canonical basis we know how to compute the coordinates of any vector.

As we think about canonical bases we realize that they appear in connection with linear ambients. A "linear ambient" is a linear space that encloses all subspaces in the domain of a given problem. Examples of linear ambients are

Q^2, Q^3, Z_p^n, $k[x]$, $k[x]_n$, $k[x, y]$, k^{nxm}, ...

A linear ambient is not a subspace; it is *the* space. In Mathematics there are spaces and subspaces; the spaces are the ambients where all vectors live.

Each linear ambient has a canonical basis, where "canonical" means that the computation of the coordinates of any given vector is trivial. More precisely, we define the class

LinearAmbient ('basis')

where we have two instance methods:

LinearAmbient | coordinatesOf: vector

and

LinearAmbient | vectorWithCoordiantes: aTuple

Of course, the class **LinearAmbient** is abstract and the implementation of these methods is a subclass responsibility. Filling up the code for these two methods corresponds exactly to the fact that this task is supposedly easy when we have a canonical basis.

Thus, **LinearAmbient** is a framework that helps us to define concrete ambients. Here are some examples:

TupleAmbient "a number *n* is given"

Canonical basis: e_1, e_2, ..., e_n

TupleAmbient | coordinatesOf: vector

↑ vector

TupleAmbient | vectorWithCoordiantes: aTuple

↑ aTuple

MatrixAmbient "n and m are given"

Canonical basis: e_{ij}, (i running from 1 to n; j running from 1 to m)

```
MatrixAmbient | coordiantesOf: matrix
    ↑ Tuple
        dim: n * m
        fromBlock: [:k | matrix at: (k - 1 // m) + 1 @ (k - 1 \\ m) + 1]
MatrixAmbient | vectorWithCoordiantes: aTuple
    ↑ Matrix
        dim: n * m
        fromBlock: [:ij | aTuple at: (ij x - 1) * m + ij y]
```

DualAmbient "a subspace V of dim n with a basis is given"

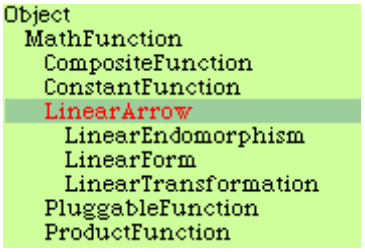
Canonical basis: the dual basis of the given basis of V

```
DualAmbient | coordinatesOf: form
    ↑ Tuple
        dim: n
        fromBlock: [:k | form at: (basis at: k)]
DualAmbient | vectorWithCoordiantes: aTuple
    | vector |
    vector ← (self canonic: 1) * (aTuple at: 1).
    2 to: n do: [:k | vector ← (self canonic: k) * (aTuple at: k) + vector].
    ↑ vector
```

Following these ideas it is easy to implement TensorAmbient, HomAmbient, ProductAmbient, PolynomialAmbient, etc. A few classes capture all relevant ambients and their functorial combinations. With the help of ambients, abstract subspaces become naturally integrated. **LinearAmbients** ensure that all abstract concepts can be modeled and that all algorithms will work.

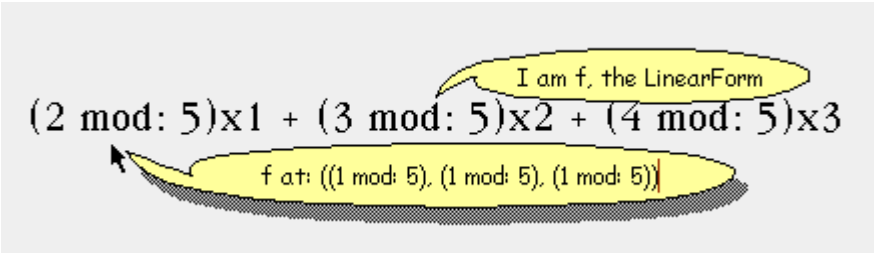
Arrows

Linear transformations are the arrows in the category of vector spaces. Since these transformations are functions, we have integrated them within the general hierarchy of mathematical functions. Our design consists of an abstract superclass named **LinearArrow** and three concrete subclasses for endomorphisms, linear forms and general linear transformations.



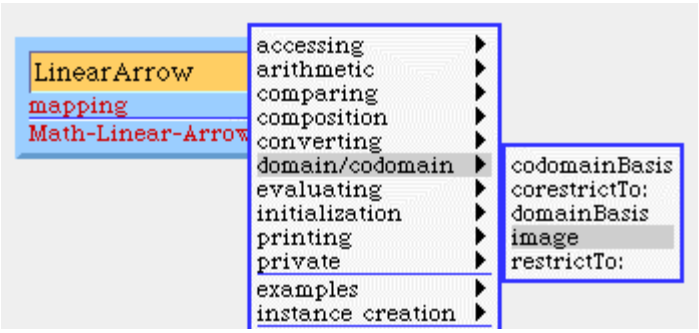
Class hierarchy of linear functions

As with the entire package, linear transformations may use any available algebraic system as the field of scalars. The domain and codomain can be also freely selected. This property gives the freedom to define transformations without worrying about indexes or coordinates.



A linear form in the field of integers mod 5 is been evaluated

A linear arrow can respond to arithmetic messages such as its sum with another arrow. They are also able to compute their images and kernels. The composition method implemented in **MathFunction** has been redefined taking into account that it must respond with a linear arrow.



Fragment of the LinearArrow instance protocol

Linear equations and linear systems

Once linear forms have been defined, one can consider linear equations by equating the form to a constant value. A collection of linear equations is a linear system. These systems are instances of a special class, namely, **LinearSystem**. The protocol of **LinearSystem** includes messages to gather information about the space of solutions. When the system is not homogeneous, the solution set is not a linear subspace but a linear variety instead; i.e., the displacement of a linear subspace by a particular solution.

Thus, **LinearSystems** are associated with **LinearVarieties**, a new class of linear objects.

To compute its solutions, a **LinearSystem** uses an external object having the purpose of solving its equations. Here we have followed the general idea of keeping algorithms away from inputs and outputs. Instances of **LinearSystemSolver** translate the equations into a matrix, and ask an appropriate **MatrixReducer** to perform the corresponding computations. Finally, they translate the matrix answered by the reducer back to the language of linear varieties and subspaces. The result of this design is a natural coordination of well-defined responsibilities. We have found that this kind of considerations make the work easier and without any loss of generality.

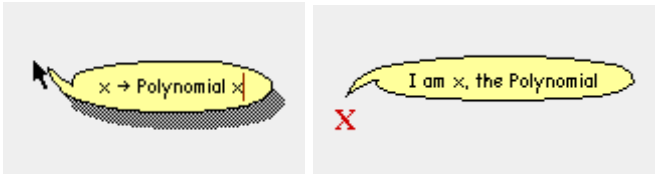
Polynomials

Numerical Analysis and Symbolic Computation show the field of Computational Mathematics from two different points of view. While Numerical Analysis has to do with floating point approximations, errors and iterative algorithms, Symbolic Algebra employs re-writing techniques and transformation rules to make the relationships among the entities at hand more apparent. Polynomials are a good example where both of these points of view are present. Numerical analysis is needed to compute the roots of a polynomial up to some error threshold; Symbolic Algebra provides the methods best suited to do arithmetic or calculate derivatives.

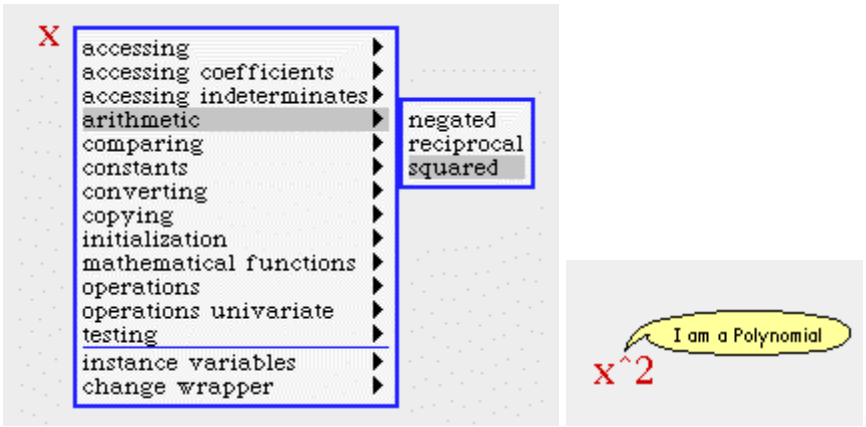
In MathMorphs we think that both points of view should be available at any time. Our ideal is an environment where the desire for any operation, numerical or symbolic, can be satisfied without any additional effort. As an example, consider the polynomial $x^2 - x - 1$. It has two roots: $\varphi = (1+5^{1/2})/2$ and $\varphi' = 1 - \varphi$. These roots are algebraic numbers, which means that we can use them in arithmetic expressions without losing precision, i.e., without introducing approximation errors. More clearly, we want to be sure about the accuracy of identities such as $\varphi\varphi' = -1$ and $\varphi + \varphi' = 1$, or any other expression of the same kind. On the other hand, the numbers φ and φ', or any other quantities obtained from them, can be approximated to "real" values going from symbols to floating point numbers. The later we move to real values, the greater precision we achieve. This crucial decision may not be taken beforehand, so we consider convenient the availability of both techniques at any time.

The anatomy of our implementation consists of two main parts: **Monomials** and **Polynomials**. **Polynomials** are built summing up **Monomials**. In turn, **Monomials** have a coefficient and a literal part, an instance of **MonomialLiteral**. For instance the polynomial $x^2 + 3y^2 + xz - 1$ has four monomials: x^2, $3y^2$, xz and -1. Here, $3y^2$ is a monomial with coefficient 3 and literal part y^2. Note that -1 is not a **SmallInteger** but a **Monomial** with an empty literal part; however we can subtract the **SmallInteger 1** from the polynomial $x^2 + 3y^2 + xz$. The illustrations below show how to create the constituent parts of this polynomial.

The instance creation messages **x**, **y**, **z** are implemented in **MonomialLiteral**, **Monomial** and **Polynomial**. In the examples, we have decided to work with **Polynomials**. Once we have all the monomials we need, we sum them up together in order to get the desired polynomial. Using MorphicWrappers this can be easily done using drop gestures and the double-click menu.

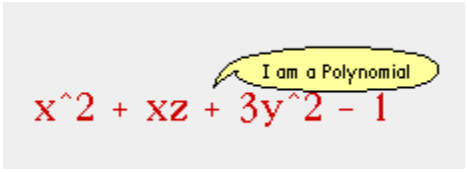


The polynomial x is created sending the #x message



Sending the #squared message to x and obtaining the result

Alternatively, we can write down the expression **x*x + (3*y*y) + (x*z) - 1** on air. This requires the variables **x**, **y** and **z** to be previously defined with the messages **x ← Polynomial x**, **y ← Polynomial y** and **z ← Polynomial z** respectively.



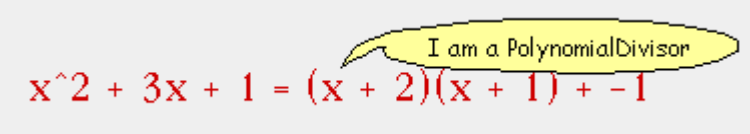
The resulting polynomial

Once we have the polynomial, we can perform all ordinary operations, such as the computation of derivatives, GCD, addition, subtraction, multiplication, division and pseudo-division, evaluation, etc. The class protocol also includes an instance creation message to obtain the interpolation polynomial corresponding to a sequence of values. The division and interpolation operations deserve special consideration.

Division and Pseudo-Division

Since the division of polynomials involves an algorithm with two inputs, dividend and divisor, and two outputs, quotient and remainder, it makes sense to employ objects specialized in the resolution of this problem. Instead of teaching the polynomials how to divide themselves, it is better to put this knowledge in a class by itself. Thus we have the classes **PolynomialDivisor**, **PolynomialPseudoDivisor** and **MultiPolynomialDivisor**. **PolynomialDivisor** implements the standard division algorithm, which works with univariated polynomials over a field. **PseudoPolynomialDivisor** performs essentially the same algorithm, specialized for polynomials over an Euclidean ring. Finally, **MultiPolynomialDivisor** implements the algorithm used with *Groebner* bases.

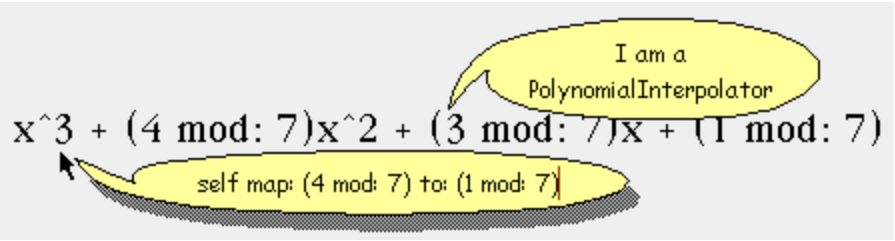
Let us say we need to divide $x^2 + 3x + 1$ by $x + 1$. We create a division algorithm for these polynomials sending the instance creation message **divide:by:** to **PolynomialDivisor**. Then, we can ask the division algorithm for the quotient and remainder. We proceed in a similar manner to divide multivariate polynomials or polynomials in an Euclidean ring.



A PolynomialDivisor **showing dividend, divisor, quotient and remainder**

Interpolation

PolynomialInterpolators are employed to find a polynomial f satisfying a sequence of conditions of the form $f(x_i) = y_i$. We add new conditions with the message **map:to:**. The interpolator corrects itself adding a new monomial so as to match the new condition while preserving those previously established. Again, our implementation does not depend on the field of coefficients. It works with any available algebraic ambient.



A PolynomialInterpolator using arithmetic modulo 7

Projective Spaces

Linear algebra's geometry is *affine* geometry. An affine space is always included in a *projective* space, but a notion of infinity is needed to describe

what affine spaces lack, the part of the projective space they do not include. With projective spaces, we can see affine spaces in a more global and unified way, and properties of affine mathematical objects are better understood when looking at their projective counterparts. For instance, in the affine plane we have parabolas, hyperbolas and ellipses, three pretty different kind of curves. But when we look at those curves in the projective plane, they are always ellipses that only differ in how the infinity line crosses them.

Projective Points

The points of a projective space are the lines in a vectorial space, i.e. linear subspaces of dimension 1. We added two methods to **LinearAmbient**, one to obtain the projective point associated with a vector, and the other to get a representative of a projective point. When we associate a projective point to a vector we are *homogenizing* it; reciprocally, a representative is a non-zero vector such that the point is its homogenization.

```
LinearAmbient | projectiveOf: vector

    ↑ LinearSubspace basis: (LinearBasis ambient: self; at: 1 put: vector))

LinearAmbient | vectorWithProjective: projectivePoint

    ↑ projectivePoint basis anyOne
```

Projective points are lines in a vectorial space, so we do not need any special class for them. In the default implementation, **LinearSubspaces** of dimension 1 are used as projective points. Subclasses may implement the methods above if necessary. For example, projective tuples and projective transformations require a more complex behavior than the one provided by **LinearSubspace**, so we introduced the classes **ProjectiveTuple** and **ProjectiveTransformation**, and redefined those methods in **TupleAmbient** and **HomAmbient**.

There are also **ProjectiveAmbients** and **ProjectiveSubspaces**, the projective counterparts of **LinearAmbients** and **LinearSubspaces**.

Projective geometry takes advantage of the computational tools available in linear algebra. This benefit is achieved by breaking down the entire space in affine regions named *charts*. The projective points laying outside a chart constitute the infinity hyperplane from the chart's point of view.

Algebraic Numbers

Real numbers are too difficult for the computer. The floating-point approach is illusory in many applications since it is imperfect in nature. Rational numbers and Integers offer a more interesting perspective for computer algebra. Also modular integer arithmetic is a source of examples and counter-examples to play with. Still, for the mathematician, these possibilities are too restrictive. Fortunately, a wider computational horizon is possible if we turn our attention to the field of algebraic numbers. Not only this field enlarges the range of possibilities; it also brings a rich group

of numerical systems such as the finite extensions of the rationals and the finite fields. What is nice about these quantities is that they require linear algebra and polynomials. Thus, algebraic numbers appear in the second layer of our project: the layer that combines basic mathematical objects already modeled.

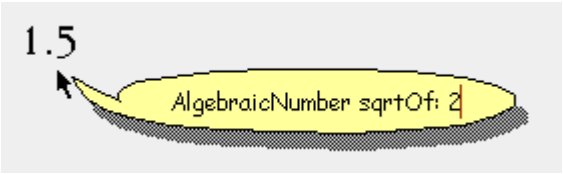
Algebraic Numbers allow us to deal with irrational numbers without incurring in approximations. As an example, let us consider the square root of *2*. Since this number is not rational, its floating-point representation is not accurate. As a consequence, we cannot take for sure that later calculations with such floating-point approximation will lead to the desired result. In order to be *completely sure* about the accuracy of any rational computation, we must adopt a different definition. The square root of *2* can be characterized as the second root of the polynomial $x^2 - 2$.

In general, an algebraic number is given as the i-th root of an integer polynomial (i.e., a polynomial with integer coefficients). Hence, two entities are required to define an algebraic number: a polynomial and an integer. Equivalently, the integer identifying the root number may be replaced with an interval such that the polynomial has only one root in it.

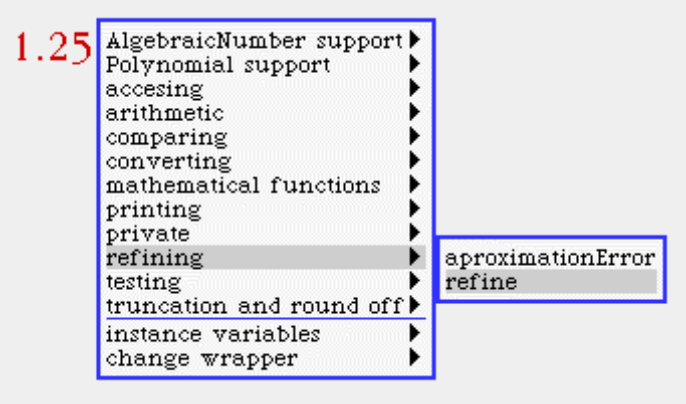
The price one pays for this infinite precision is finite. It is necessary to develop more complicated algorithms for the arithmetic operations. For instance, given two algebraic numbers, one must compute the interval and polynomial corresponding to its sum.

The mathematical theory behind algebraic numbers is simple and well known. In order to compute some arithmetic operation, one computes a certain polynomial matrix known as the resultant. Essentially, given the polynomials of two algebraic numbers, one must compute the resultant of some other polynomials derived from them. The determinant of this polynomial matrix is the desired polynomial for the given operation. The way those "other polynomials" are obtained from the original ones depends on the arithmetic operation that must be performed. There is also a rudimentary "interval algebra" that must be defined to isolate the result.

When algebraic numbers are created, they contain a polynomial and an interval. This interval contains only one root of the polynomial, which corresponds to the value of the algebraic number. This interval may be large, but any number of refinements can be applied to get closer and closer to the algebraic number.

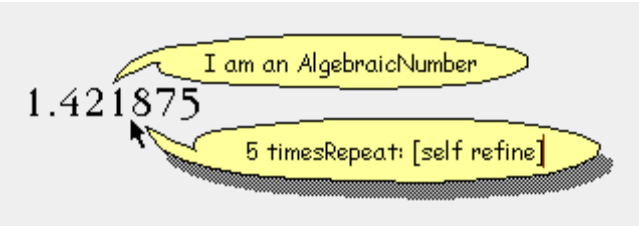


At first glance, the approximation may look poor, but...



... sending refine one obtains closer results.

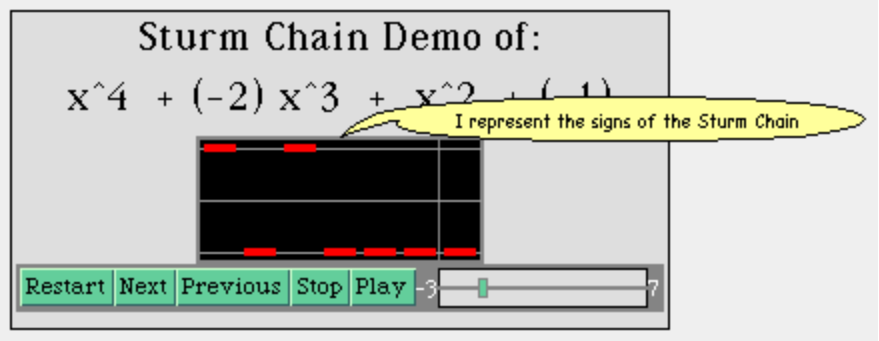
The protocol of algebraic numbers includes, of course, all arithmetic operations such as addition, multiplication, division, etc. What is more interesting, it includes the computation of the square root and in fact, of any other root. As new algebraic numbers appear, their defining polynomial and isolating interval are always available. The approximation error corresponds to the size of that interval, so it can be arbitrarily reduced with further refinements. Furthermore, the length of the interval may be shortened automatically by the number when it is employed as an operand in some operations.



Five refinements are applied to the square root of 2

The Sturm's Theorem

The key point of the implementation of Algebraic Numbers is the Sturm's Theorem (1834). This brilliant result gives a surprisingly simple method to find out the isolation interval. The theorem was the subject of study of another MathMorph's project by Eric Rodriguez Guevara.



A machine showing the sign changes in the Sturm's chain

The idea behind the study of the Sturm's theorem was to illustrate a mathematical proof with an animated morph. The theorem states that the number of roots in a given interval, *[-3, 7]* in this example, can be obtained counting the number of sign changes of any *Sturm chain* at both limits of the interval. One way to compute a valid chain is to start with the polynomial and its derivative, and perform successive divisions. The negative remainder of each division replaces the divisor, and the divisor replaces the dividend in the next iteration. The procedure parallels Euclid's algorithm to compute the *gcd* except that here the remainders are negated. There are other possible chains and all of them give the same result.

The demonstration of the figure shows a chain containing seven polynomials. As the interval's right limit moves from *-3* to *7*, the red dashes show the signs of the polynomials in the chain when evaluated at the moving end. At any given place, a sign change occurs when the sign goes from *+1* to *-1* or from *-1* to *+1* (zeros do not count).

The Sturm's theorem says that the number of real roots may be obtained as follows: (1) evaluate the polynomials in the chain in the left limit of the interval and count the number of sign changes occurring there; (2) proceed in the same way with the right limit, and (3) subtract the value computed in (2) from the value computed in (1). This is the number of roots falling in the interval. The only requirement is that the *gcd* between the polynomial and its derivative should be a constant; but this requirement is easily met dividing, if necessary, the polynomial by the *gcd* with its derivative.

The object responsible for computing all this information is an instance of the class **RootFinder**, illustrated below.

A RootFinder

Using the Sturm's theorem it is easy to verify if a given root is isolated in a given interval. It also makes easy to isolate all the real roots. This is the role that this theorem plays in the implementation of Algebraic Numbers.

Finite Groups

How should we model algebraic structures? Our central interest in MathMorphs is to design environments where mathematical objects are alive. The efforts in computational mathematics seem to be concentrated in the implementation of algorithms. We believe that before the implementation of an algorithm comes the definition of the objects that the algorithm acts on. This way of thinking leads us to model not only the elements but also the structures such as vectorial spaces, groups, rings, fields, etc.

While Squeak provides, say, the class **Fraction** to model individual elements, the field of rationals has not been necessary for ordinary applications. But in Mathematics one usually needs the algebraic structures in addition to the elements. This need has been clearly shown in the Linear Algebra package, where we introduced linear spaces and ambients. Finite groups are the simplest example of algebraic structures, so we want to use them as a way to gain more experience on how to model algebraic structures in general.

Subgroups

Finite groups (subgroups) are implemented as the collection of its elements. Methods for enumerative messages have been provided. This makes possible to define operations on groups just in the same way as they are defined in mathematics. The protocol for operations on groups includes the computation of the *center*, the *commutator*, the *order* of the group and the *exponent*. Here is an example:

```
Subgroup | centralizerOf: aCollection

    ↑ self select: [:each |

        (aCollection detect: [:one |

            each * one ~= (one * each)] ifNone: []) isNil]
```

The instance protocol also includes messages testing whether a group is *abelian*, *normal* or *cyclic*. It also includes methods for computing the translations (cosets), intersections, direct products, and factor groups. The

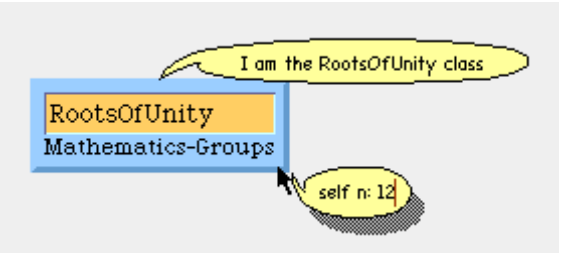
factor group *G/N* is composed of all the cosets of the form *Nx* for all *x* in *G*:

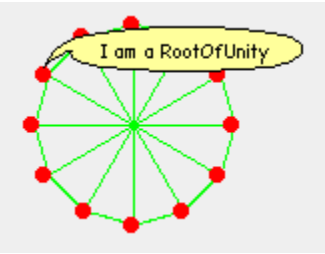
```
Subgroup | / normalSubgroup

    "Answer the factor group of the receiver by the argument."

    ↑ self collect: [:each | TranslatedSubgroup

                        subgroup: normalSubgroup translation: each]
```

Some particular examples of groups are modular integers, quaternions, homomorphisms, direct and semi-direct products, permutations and roots of unity.

A homomorphism is a function *f* from a group *G* to a group *H* such that $f(xy) = f(x)f(y)$ and $f^{-1}(x) = f(x^{-1})$. Homomorphisms are the *arrows* in the theory of groups. In our implementation, they know the domain *G*, codomain *H* and a dictionary that maps *x* to *f(x)*. The instance protocol includes messages for evaluating, for testing if they are automorphisms, endomorphisms, epimorphisms or isomorphisms; and for the computation of the inverse, the kernel, and the fiber at a given element.

We can create special classes to represent particular families of groups. For example, we created the **RootsOfUnity** subclass of **Subgroup**. An instance of order *n* in this family represents the group G_n. We customized this subclass with special behavior for instance creation and for the computation of factor groups. In order to create, say G_{12}, we only need to provide the number *12*; thus, our instance creation message is **RootsOfUnity n: 12**.



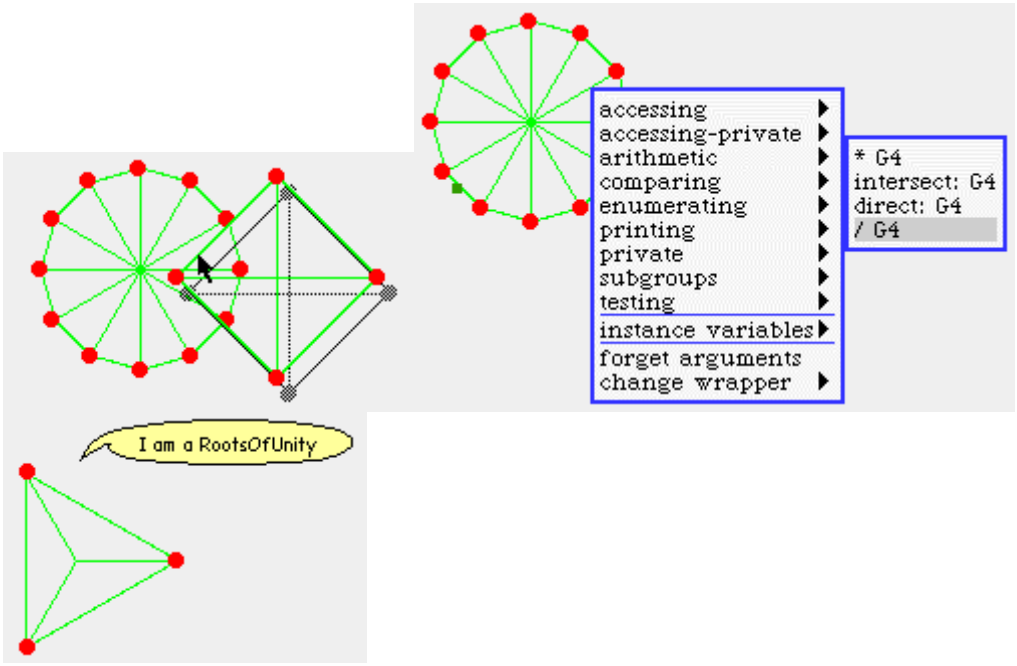


The group of 12th roots of unity and its individual roots (the red dots).

Since the factor group G_n/G_m is $G_{n/m}$, we redefine the corresponding method in the following way

RootsOfUnity | / aRootsOfUnity

↑ self class n: self order / aRootsOfUnity order



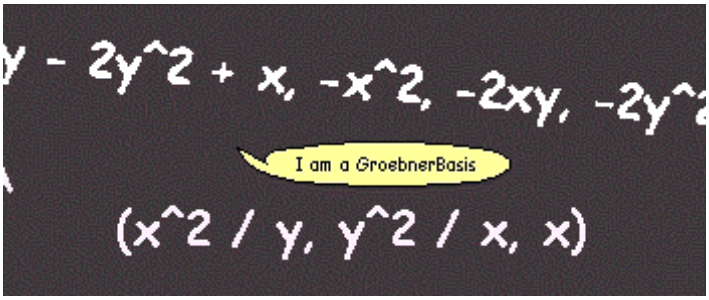
The result of dividing G_{12} by G_4 is shown to be G_3

Algebraic Geometry

In its seminal form, Algebraic Geometry deals with the set of zeros that are common to a finite collection of polynomials in several variables. These sets consisting of the solutions of polynomial equations are the so-called algebraic varieties. Here with "solutions" we mean the points in an algebraic closed field, such as the field of complex numbers.

After decades of development and generalization of the theory, some mathematicians realized that all the knowledge accumulated in this domain was not enough for studying algebraic varieties from a computational perspective. The lack of effective methods became apparent, and two new disciplines were born: Computational Algebraic Geometry and Computational Commutative Algebra. Old problems deserved new interest. Suddenly many mathematicians began to ask themselves how to compute the dimension of a variety, it's projective closure, it's degree, or, in the zero dimensional case, the concrete points that make up the geometrical set.

Once the first algorithms were rescued from history, it was understood as a matter of fact that the inherent complexity of these problems was too high to be efficiently solved with computers. New methods were needed, and a more careful analysis of the underlying hypothesis had to be formulated. Among the new tools, *Groebner* bases showed to be the most useful.

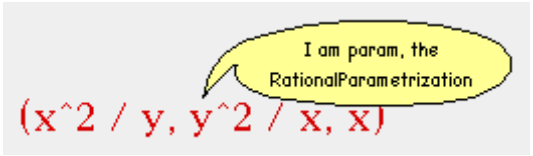


Squeak as a blackboard for mathematical thoughts

In this project we introduce some usual notions studied in Computational Algebraic Geometry and Commutative Algebra in the form of Squeak objects. The computational point of view and its practical limitations are illustrated facing the "Implicitation problem". This problem can be stated as follows: given an algebraic variety described in rational parametric form, find a system of polynomial equations defining it in implicit form.

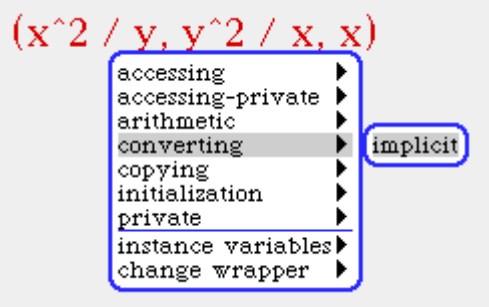
Let us see an example for the sake of clarity. Consider the set of all points with coordinates of the form $(x^2/y, y^2/x, x)$. We want to find a set of polynomial equations whose zeros are the topological closure of this set.

The parametric form allows us to deduce some properties of the variety. For example, since it has two parameters: *x* and *y*, the variety has a dimension of two. Still, the implicit form has some advantages. For instance, using the parametric form it is not easy to verify whether a given point belongs to the variety. Also, the parametric form is expressed in terms of rational functions. So, it cannot be evaluated for all possible values of *x* and *y*, meaning that some points at the boundary of this surface cannot be expressed in this way. In general, to be able to use the tools provided by Commutative Algebra, one needs a set of polynomials defining the variety.



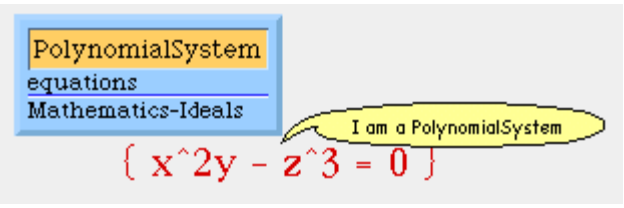
A named instance of RationalParametrization

In the current implementation, parametric representations of varieties are instances of **RationalParametrization**. Once such a parametrization has been created, it can be asked for a system of polynomial equations. The solution of this system is the closed variety given by the parametrization. The instance protocol of a **RationalParametrization** includes the message **implicit**. This message answers with a proper instance of **PolynomialSystem**.



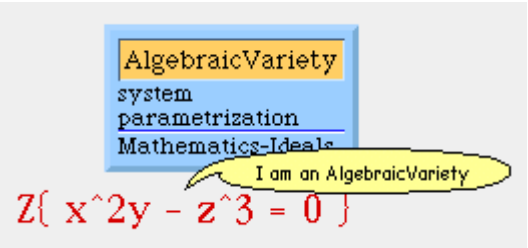
Sending the implicit message to ask for a polynomial system

Having classes to model each of the relevant notions allows us to avoid artificial conventions. Thus, we can go from a **RationalParametrization** to a **PolynomialSystem** and then to an **AlgebraicVariety**. Neither Arrays nor other meaningless data structures have any chance to remind us that all this magic takes place in a computer.



The resulting system and its class

The project also includes classes to represent Groebner bases, ideals, and of course, polynomials.



An algebraic variety as a set of zeroes

How it works

We cannot give a mathematical proof here, but we can explain the way we proceed to solve the Implicitation problem. The algorithm starts with the parametric form. It first replaces all variables with new ones. In our example the parametric form becomes $(u^2/v, v^2/u, u)$. Secondly, new variables are introduced using the standard names. In the example these are x, y, z. Next, the new variables are used to eliminate denominators from the equation $(x, y, z) = (u^2/v, v^2/u, u)$. In the example we obtain: $xv - u^2$, $yu - v^2$, $z - u$. Now, a last variable is added as to eliminate the product of all denominators; in our case: $1 - wuv$. Then a Groebner basis is computed

using the lexicographic order defined by $w > v > u > z > y > x$. This is the hard part because of the complexity of the problem. In the worst case this could be impracticable. Here we have chosen an example that finds the basis within a few seconds. The computation adds three new polynomials: $yz^2w - v$, $xy - zv$, $x^2y - z^3$. The last step of the algorithm consists of selecting those polynomials in the Groebner basis that contain the variables x, y and z only. Here, there is only one polynomial with this property: $x^2y - z^3$. Thus, the variety is implicitly defined with the equation $x^2y - z^3 = 0$.

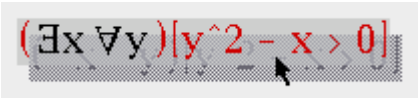
Tarski Geometry

Classic Algebraic Geometry works with complex numbers. The same theory is extended to any algebraically closed field. There is a real counterpart of this discipline. When real numbers are used instead of complex numbers, polynomial equations have to be replaced with polynomial inequalities. The real solutions of these new systems are called *semialgebraic sets*. Real Geometry (also known as Tarski Geometry) and Real Commutative Algebra develop these notions in depth. Again, the computational viewpoint opens new and interesting problems related with algorithmic and complexity. But this time, the geometry and algebraic concepts are enlightened with results and notions coming from Logic.

The underlying theory introduced by Tarski is based on the elimination of quantifiers from first order logical sentences. "First order" means that the variables under the scope of existential or universal quantifiers represent real numbers, not sets. The elimination theorem states that any such sentence is equivalent to another one that is free form quantifiers. The building blocks of these sentences are multivariate polynomial inequalities. Inequalities are combined with disjunctions, conjunctions and implications. The variables occurring in the polynomials are bounded under the scope of existential or universal quantifiers.

As an example consider the sentence $(\exists x\ \forall y)\ [y^2 - x > 0]$. Here we have one inequality and two quantifiers. This sentence turns out to be true, since x can be taken negative. Thus, $1 > 0$, as any other tautology, is a quantifier-free sentence equivalent to the former.

There are circumstances in which the truth of an expression is far from being apparent. But, on the other hand, all quantifier-free sentences containing only bounded variables happen to be trivial since they cannot involve any variable at all. These sentences only include constant inequalities such as $1 > 0$, $1 < 0$ and so on. So, the quantifier elimination can be used as an effective method to find out the truth-value of a sentence, provided that all its variables are bounded.



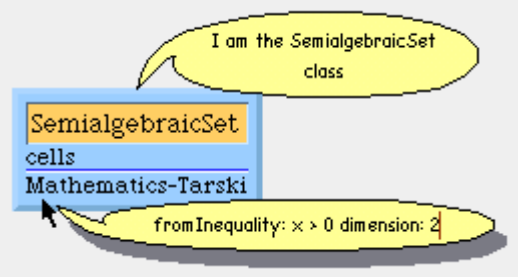
A instance of QuantifiedSentece

Effective is not the same as efficient. The inherent complexity of the elimination algorithm is exponential. This limitation parallels the one found in Algebraic Geometry. So, in practice, one cannot expect to solve all

possible expressions. However, there are interesting cases to play with. Following our example, let us consider now the semialgebraic set in R^2 consisting of all pairs (x, y) such that $x > 0$. We can create the polynomial inequality sending the message $x > 0$, where x names the polynomial x. The instance creation message is

SemialgebraicSet fromInequality: x > 0 dimension: 2**.**

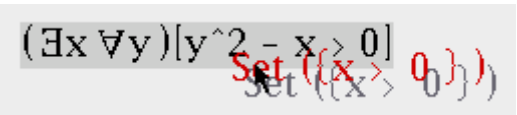
The dimension must be provided when it is not possible to deduce it from the number of variables occurring in the inequality.



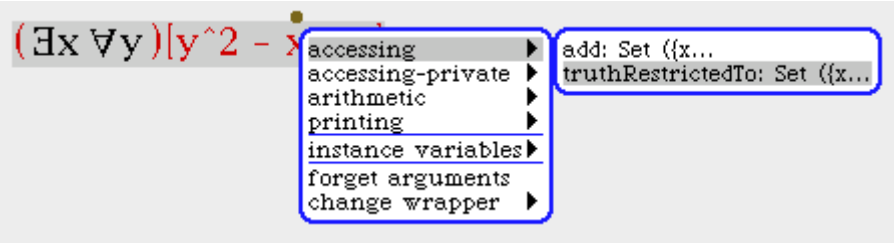
Creating the semialgebraic set {(x, y) | x > 0}

The class answers with the corresponding semialgebraic set (note from the illustration that when writing the message on air, the name 'self' can be omitted. Since the receiver is obviously the class **SemialgebraicSet**, the use of 'self' becomes superfluous).

Now we are going to use the new semialgebraic set to cause a variation in the logic of our tautological sentence. Instead of asking it for its truth-value, we are going to ask it for the truth restricted to the semialgebraic set. We can do this in two simple steps with the MorphicWrappers. We first drop the set on the sentence and next choose the **truthRestrictedTo:** method from the double click menu of the sentence.



Step 1: the semialgebraic set is dropped on the sentence



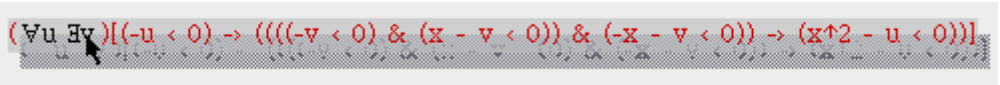
Step 2: the message is sent from the double click menu

Surprisingly, the tautology answers with false. At first glance, this could seem to be wrong, since tautologies should be always true. Why is this answer correct? Because there not exist a positive value of x less than any squared real value of y.

As a second example let us consider the metric definition of continuity. In order to keep it simple, let us think about the concrete case of the continuity of the function x^2 at the point $x = 0$. The formal definition reads as follows:

$$(\forall\varepsilon)[\varepsilon > 0]\ (\exists\delta)[(\delta > 0) \wedge ((-\delta < x) \wedge (x < \delta)) \Rightarrow (x^2 < \varepsilon)].$$

As we can see, this sentence only uses polynomial inequalities in the variables ε, δ and x. Then we can create it as an instance of **QuatifiedSentence**. In the figure, we have employed the letter u for epsilon, and the letter v for delta.



A QuantifiedSentence **expressing the continuity of x^2 at 0**

Double clicking on this sentence, we obtain **true**; i.e., our **QuantifiedSentence** is able to ensure us that our function is continuous at 0.

The Tarski Geometry project makes use of many of the tools developed in other projects. Here we employ **Polynomials**, **AlgebraicNumbers**, **Matrixes**, and so on. It also introduces new classes including **PolynomialEquation**, **PolynomialInequality**, **BooleanConnective** with subclasses for **Disjunction**, **Conjunction** and **Implication**; **Quantifier**, **QuantifiedSentence**, **SemialgebraicSet**, etc.

The hard part relies on the Cylindrical Algebraic Decomposition algorithm (CAD). As usual in the designs we have seen so far, the algorithm takes a class by itself. Objects of this class can decompose R^n into connected components. These components are the so-called *semialgebraic cells*. In such cells, all polynomials from a given set keep their sign invariant. Such a decomposition is called a *cell complex* of R^n.

In our implementation, the connected components are instances of **SemialgebraicSet**. When $n = 1$, we have the simplest case. A special subclass, **SemialgebraicLineCell**, takes care of this.

The single variable case is easy. Suppose we have to find the CAD of R with respect to the polynomials $\{f_1, \ldots, f_n\}$. We find all the real roots of $f_1, \ldots, f_n$, say $a_1 < a_2 < \ldots < a_m$. Then, the CAD we were looking for is:

$$[-\infty, a_1),\ [a_1, a_1],\ (a_1, a_2),\ [a_2, a_2],\ (a_2, a_3),\ \ldots\ [a_m, a_m],\ (a_m, +\infty].$$

As we have seen before, when the polynomials have rational coefficients, the roots can be effectively computed using Sturm's theorem. In general, the roots a_i are Algebraic numbers and we can handle them with infinite precision. Each component can be asked for a sample point laying on it. Sample points are constructed as follows:

- the sample point of $(-\infty, a_1)$ is $a_1 - 1$;
- the sample point of $(a_m, +\infty)$ is $a_m + 1$;

- the sample point of $[a_i, a_i]$ is a_i,
- and the sample point of (a_i, a_{i+1}) is $(a_i + a_{i+1}) / 2$ (middle point).

Let us illustrate the algorithm with an example: decide whether the sentence $x^2 - 1 > 0$ is true or false. First we find the CAD of R with respect to $\{x^2 - 1\}$. The roots of $x^2 - 1$ are -1 and 1, so the CAD is

$(-\infty, -1)$, $[-1, -1]$, $(-1, 1)$, $[1, 1]$ and $(1, +\infty)$.

Sample points are, respectively,

$-2, -1, 0, 1$ and 2.

We know that the polynomial $x^2 - 1$ is sign-invariant over each of the components, so if it is positive in all components, it is positive all over R. We evaluate $x^2 - 1$ at each of the sample points checking the signs. From there, we can conclude whether the sentence is true or false. It turns out to be false, since the values we obtain are:

$(-2)^2 - 1$, $1^2 - 1$, $0^2 - 1$, $1^2 - 1$ and $2^2 - 1$

or

3, 0, -1, 0 and 3.

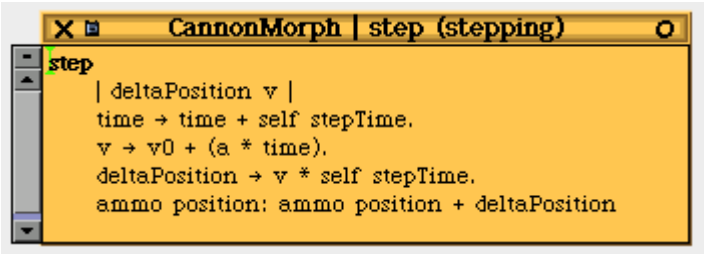
Note that while our sentence is not true on R, the CAD shows the cells where it does hold. From the evaluation above we deduce that the sentence $x^2 - 1 > 0$ defines the semialgebraic set $(-\infty, -1) \cup (1, \infty)$.

The construction of the CAD in the multivariate case works by induction on the dimension. The interested reader will find the mathematical details in [Algorithmic Algebra, B. Mishra, Springer-Verlag, 1993].

PhysicsMorphs and BiologyMorphs

The MathMorphs project has motivated some students to start experimenting with other sciences following the spirit initiated in MathMorphs. The first experiments about PhysicsMorphs and BiologyMorphs have given us lots of fun. Kinematics and Wave theory offer good sources of examples. Other projects include Optics and Kalman filters. Let us briefly describe some of them.

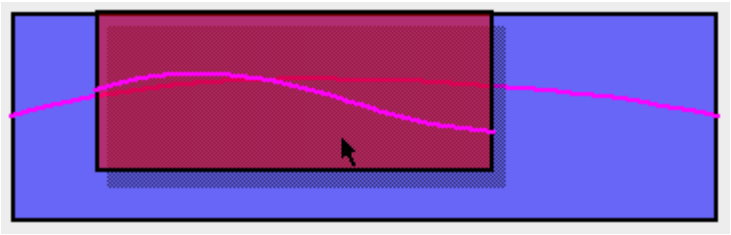
A very simple and still representative object to play with is the **CannonMorph**. The cannon obeys the law of gravity. Any other morph can be taken as ammunition. The cannon impels the ammunition with initial velocity proportional to its own extent. In each step the cannon updates the position of the ammo following Newton's laws. This simple example can be implemented in two different ways. Since the function describing the position is a polynomial in the indeterminate t (time), the computations can be made exact. On the other hand, numerical methods can be used to approximate the derivative of this function, i.e., the velocity, with the quotient of small increments in space and time.



Only one simple method gives the ammo its natural trajectory

Differential equations

A more elaborated example involving numerical methods is the solution of differential equations.



Two VibratingStringMorphs **moving as waves**

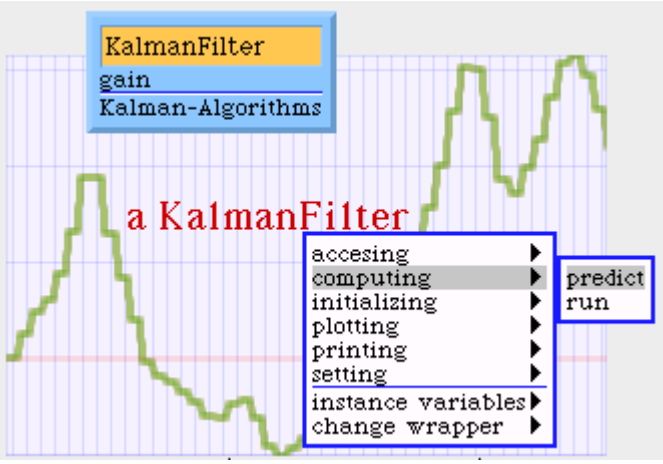
Consider a string under tension. When it is separated from its resting position and then released, as in the case of string musical instruments, a shaking movement is started along the string. The vibration consists of a wave phenomena that can be described with a function $y = f(x, t)$. Here x and y are the horizontal and vertical coordinates of a point in the string and t is the time. This function satisfies a differential equation known as the *wave* equation. The general solution of this equation represents all possible functions describing the position of each point in the string at any given instant t. The exact solution corresponds to the initial conditions defining the position and velocity of the string just before it is released.

In many books, such functions are drawn in order to illustrate different waveforms. But since these functions involve 3 variables, they must be represented using 3D graphs. However, one of these dimensions is temporal, not spatial. Thus, these graphs lack the main characteristic of the object they are describing: movement.

A better way to illustrate a solution is letting the two spatial dimensions change with time. The whole effect is the one found in "real" strings under similar conditions. To accomplish that, a **VibratingStringMorph** solves the wave equation in each instant t (i.e., in "real time"). Since the methods to solve this kind of differential equations are numerical, the positions of a sample of points are computed at regular intervals. In each step, the **VibratingStringMorph**, a subclass of **CurveMorph**, changes the points defining it. And, of course, the string oscillates as expected.

Kalman filters

High-Energy Physics studies the fundamental subatomic constituents of the matter. Particle accelerators provide subatomic particles for this study. When, say, protons-antiprotons at high-energy collide, subatomic particles appear. Physicians employ different detectors to determine the trajectory of these particles.



A KalmanFilter**, its class and its plot**

Think of a detector as a set of concentric cylinders. Each cylinder corresponds to a layer. A *hit* is the signal produced by a particle at some point in the detector. Hits occur on different layers. *Tracks* describe the trajectory of the charged particles along a magnetic field. Hence the hits, can be interpreted as the points of intersection of the tracks with each of the layers. The goals of track reconstruction are pattern recognition (track finding) and track fitting.

A Kalman filter is used to predict the position of the particle in layer $k + 1$ from its position in layer k. Relevant information comes from the magnetic field. There is also a stochastic process. When a particle goes from layer k to layer $k + 1$, the detector itself introduces a random dispersion.

The method employed to find the tracks is iterative. It uses the notion of *global* track. A global track takes into account the hits, the parameters of the fit and a bound for the error. The filter combines the predictions for the layer $k + 1$ with the hits detected so far. It takes the best fit and goes on propagating from layer to layer.

Kalman filters allow to find global tracks since they resolve track finding and track fitting at the same time. In each iteration, the filter finds the hits corresponding to a given track and the parameters of the fit.

Smooth changes are introduced in the parameters of the tracks. The changes, which are kept under certain threshold error, try to obtain a better fit around the vertexes. The approach is efficient since the matrixes processed by the filter have dimensions of $5x6$ or below.

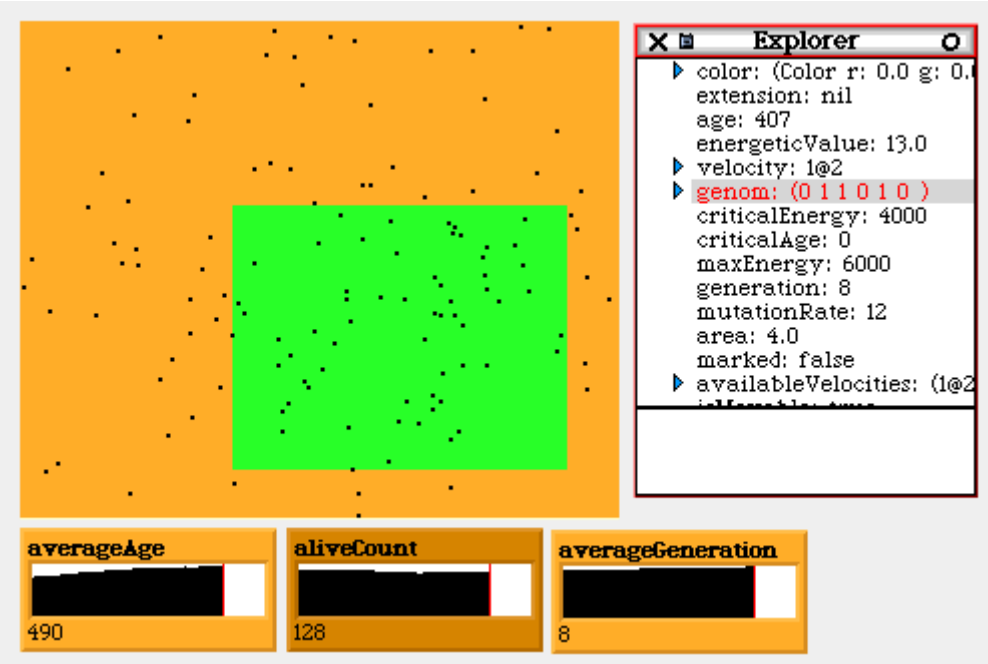
This project uses many of the tools developed in MathMorphs such as: Linear Algebra, Random Variables, Numerical Integration and Function

Plotting. It adds the Kalman Filters and models for **Tracks**, **Detector**, etc. We want to express our gratitude to Ariel Schwartzman who developed this work, playing long hours with MathMorphs and PhysicsMorphs.

BiologyMorphs

Pablo Shmerkin came with the idea of developing BiologyMorphs. His main interest is about artificial life. His project is made up from objects of classes such us **Genom**, **GeneticCode**, **Biomorph**, **Squeakobe**, **EcosystemMorph** and **BiosphereMorph**.

A biosphere can hold more than one ecosystem. Each ecosystem has its own set of parameters. These parameters can be altered by the user. If required, the biosphere can collect all genetic information and place it into a **BookMorph**.



In the illustration we see a biosphere with two ecosystems inside it, differentiated by their color. The visual counters show the evolution of the biosphere as natural selection operates on the biomass, and as the user introduces its own changes (artificial selection). The dots are moving microbes, each of them having a genom. The object explorer window shows the instance variables of a specific microbe. The green ecosystem is a kind of *Garden of eden*, so there are a lot of microbes there. As generations goes by, microbes in each ecosystem develop quite different behaviors. Those in *the Garden of eden* will remain as static as possible, while those in the orange ecosystem, a more hostile ecosystem, will move frenetically in search of food.

Related projects

MathMorphs, as a collaborative community, has motivated some of its members to initiate individual projects. These projects fall in the boundaries

of the group's main interests. They have had the virtue to face the group with new areas and possibilities. On the other hand, the group has acted as a qualified and enthusiastic audience giving valuable feedback to the authors. At the time of this writing we can name three major projects: Type Inference by Francisco Garau, Arithmetic Coding by Andres Valloud and Text-to-speech by Luciano Notarfrancesco.

Type inference

Type Inference is an ambitious project currently under construction. The main responsible for this project is Francisco Garau. He has studied the problem in detail and has implemented a type inference engine for Squeak. Garau has been submitted the work to the University of Buenos Aires in order to get his graduate degree.

Type inference is not a new area in Computer Science; it has been widely used in functional languages. But in the past years, there has been an increasing interest to take those ideas to the object oriented languages. Squeak is no exception.

The main interest behind type inference is to know in advance the types that arbitrary expressions will hold at run-time. There are different approaches to solve this problem. Many of them differ in the formalism and the notion of what a type is.

In this project, the concept of type is heavily biased towards what might be useful to the compiler. These are called *concrete types*. They hold the most precise and detailed information that the compiler would need, i.e., the classes of all objects resulting from an expression in run-time.

In the inferring process, type information is always kept at the most detailed level. As an example, the type of **1@1** is **<Point x: <SmallInteger> y: <SmallInteger>>**. Note that the type of an object knows the types of all its instance variables.

Some very useful applications arise from a type inference engine:

- An image stripper, that throws away all the code you do not use (thus a 'hello world' application could be minimal).
- A static checker, that warns if a message-not-understood could happen at run-time.
- An optimizing compiler that, when there is no ambiguity, could replace a message send with a direct call to the method (or even inline it).

It should be clear that, in order to infer types, an initial expression must be given. Precisely, that expression, also called "program", is what makes feasible some of the above uses. (Independently, Lex Spoon is also interested in type inference. His "Lucid" system does not need an initial expression; Spoon’s main target is program understanding.)

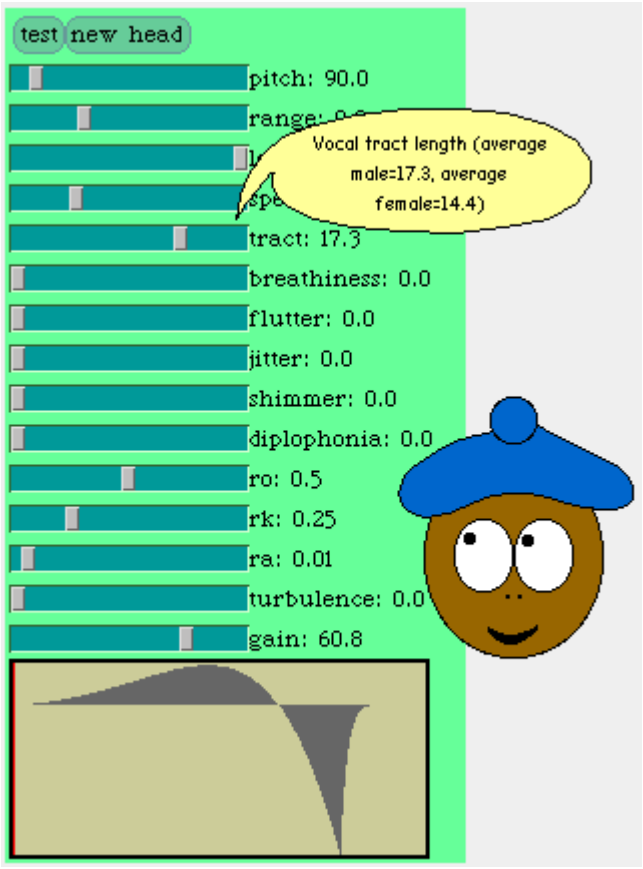
Our work is based on Ole Agesen's thesis, adapting his ideas from Self to Squeak, and also making some improvements on the treatment of blocks and instance variables. His ideas about the parallelism between run-time and type inference-time were taken a step further. Thus, you will find classes like **TiSystem**, **TiInterpreter**, **TiClass**, **TiCompiledMethod**, **TiCompiler**, **TiCompiledBlock**, **TiPrimitive**, and so on.

Text to speech

The Squeak Text-To-Speech (TTS) system includes classes for doing formant synthesis, phonetic transcription, and prosody generation.

The synthesizer itself, **KlattSynthesizer**, is a Klatt-style cascade/parallel formant synthesizer. This type of synthesizer was developed by Dennis Klatt for the MITalk (now DECTalk) text-to-speech system.

In the synthesizer, filters are organized in two *branches*: a parallel branch more useful for consonants, and a cascade branch best suited for vowels.



The synthesizer has some global settings such as sampling rate, milliseconds (or samples) per frame and number of formants in the cascade branch. There are also 52 time-varying parameters that are updated every 10 milliseconds or so, all at once, setting the current frame to a new **KlattFrame**.

These 52 **KlattFrame** parameters specify the formant frequencies, bandwidths and amplitudes, the amplitude of each excitation source (friction, aspiration or voicing), the voice quality, and the fundamental frequency or pitch.

Special care was put on the voice quality parameters, so as to make the synthesis of different voice personalities and even pathological voices

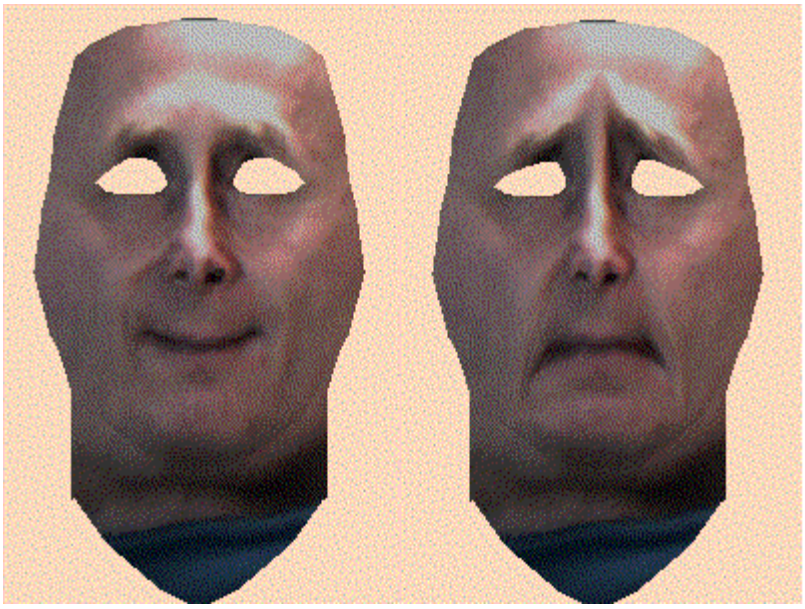
easier. At the time of writing, this is the most complete publicly available Klatt synthesizer.

In the TTS system, phonetic transcription is achieved by means of dictionary lookup and contextual text-to-phoneme rules. Each **PhoneticTranscriber** has a collection of **PhoneticRules**, and optionally a lexicon. When a transcriber is asked for the transcription of a word, it searches for the word in the lexicon first, and if the word is not found then the rules are used.

A list of **PhoneticEvents** is generated from the **Phonemes** after the phonetic transcription. Each **PhoneticEvent** includes a phoneme, duration, loudness, and a pitch contour. Some simple prosodic rules are employed to assign duration and intonation to the event list. After this, the events are played on a **Voice**.

There are two kind of voices currently implemented in the Squeak system. The first voice is the **KlattVoice**. This voice produces sound from **PhoneticEvents**. Each phoneme is mapped to one or more **KlattSegments**, then **KlattFrames** are generated from the **KlattSegments**, and finally the **KlattFrames** are played on a **KlattSynthesizer**.

The other voice available in Squeak is a **GesturalVoice**. It can play **GesturalEvents** (for lips, eyes, moods) and **PhoneticEvents** animating a face. Several voices can be combined into a **CompositeVoice**, so the TTS system is able to do synchronized face animation and speech synthesis on a composite voice made up of a **KlattVoice** and a **GesturalVoice**.



The author (LEN) has extended this system in several directions. For instance, much better and natural voices have been achieved doing *diphones concatenative* synthesis with LPC and Residual Pitch Synchronous LPC diphones. Also, the **GesturalVoice** is being extended to use a 3D face

with Waters' muscles model, which is specially well suited for the realization of emotions.

Coding and compression

One of the final exams for the course "Objetos Matemáticos en Smalltalk" was a compression project by Andrés Valloud. Its goals were to find a suitable design for modeling data compression, and to implement some compression schemes. The basic design followed the guidelines described in the book "Text Compression", by Timothy C. Bell, John G. Cleary, and Ian H. Witten. This book was published by Prentice Hall in 1990. The book emphasizes the separation of the *model* and the *coder* for any given compressor. The project has shown that in most cases it is good practice to separate the coder from the model, although this separation should not be too strong.

The project began with an experiment to see if the model proposed by the book "Text Compression" was acceptable in Smalltalk. The book is C oriented, which does not reflect the proposed separation of the model and the coder. This experiment consisted of building a lossless ADPCM compressor with exchangeable models and coders. It showed that there are very concrete benefits that arise from separating the model and the coder. Improvements are easy to obtain, both in terms of efficiency of execution and of compression performance. The ADPCM compressor is consistently better than any combination of Lempel Ziv and Huffman compressors such as Zip. It is also comparable to other proprietary lossless audio compressors such as Ultra Compressor 2 and Rar. It is also extremely efficient. Using only Smalltalk code, this compressor running on a modest machine can process at least 10,000 symbols per second. Modern computers greatly increase this figure. The lossless ADPCM compressor was also the subject of the paper "Lossless Audio Compression and its Implementation", written by the author of the project. It was submitted to and published in a student paper contest for the 28th JAIIO conference, from September 9 to September 13, 1999. This event was held at Buenos Aires University.

The second part of the project consisted of an arithmetic coder driven by finite state and finite context models. The independence of the model and the coder was essential for achieving reasonable execution efficiency. This compressor is now undergoing further changes in order to obtain independence of the arithmetic used (fractions, integers, floating point numbers).

A conclusion arising from this project is that although splitting the coder from the model is a good thing, it is not enough. In most cases, the model could use some information arising from the coder, especially the coding efficiency for each symbol. It has also been found that the order in which the probability intervals are ordered in a finite state model may affect the coding efficiency of the arithmetic coder.

Several side **developments** were necessary to support this project. They include the **BitStream**, **Tree**, **Probability**, **ProbabilityInterval**, and **BitChunk** classes with their associated objects, along with bit manipulation methods.

Conclusions

It has been said that Squeak is a vehicle. We are trying to make MathMorphs one of the points of departure from which get the vehicle running.

In 1997 we begun a course of pure mathematics based on Smalltalk at the University of Buenos Aires. Squeak quickly became the natural environment among the students, and that allowed us to discover new possibilities on how to teach and learn math. We think this is a good place to present a short report of our exciting experience.

One of the authors of this chapter has been working at the Mathematics Department of the University of Buenos Aires for the last twenty years. He considered himself an innovative teacher. By getting involved with the Squeak idiosyncrasy, he has realized how much the conventional approach was constraining him. The transmission of knowledge to new mathematicians frequently fails in two ways: a) brilliant theory dissertations contrast with poor provision of artificial examples, and b) no relevant work is devoted to make the students conceive problems of their own.

In the course "Objetos Matemáticos en Smalltalk", Squeak is used as a laboratory, and not merely as a programming tool. The great thing is that Squeak provides a suitable atmosphere where mathematical objects easily become alive.

The communion between Squeak and Mathematics is not an accident. The spirit of Squeak is similar to that found in high Mathematics, where the consequences of fundamental ideas are followed without any loss of generality.

Few and well established principles naturally correspond to axioms. Precise definitions play a crucial role, both in Squeak and in math. Smalltalk notation is a kind of algebraic notation. Inheritance and polymorphism have algebraic roots too. Squeak objects and messages resemble the *categorical* objects and arrows found in the underlying structure of mathematical theories. Squeak openness conforms to the mathematical conception of proof.

While object orientation is normally an abstraction task, where real things have to be represented in a virtual space, the same practice has the inverse result when mathematical notions are modeled. The model of a mathematical concept is more tangible than the concept itself. Instead of *abstracting*, one experiences the rather unusual feeling of *concreting*.

Along these few years we have also noticed many interesting facts regarding pedagogy. A few of them are:

- The students learn Squeak as a natural consequence of thinking about mathematical ideas.
- "Well known" mathematical notions suddenly show unsuspected properties.
- Some theorems are naturally generalized in uncommon ways. As a result, deeper than normal understanding is achieved.
- "Living examples" that naively begin as simple forms of code testing, quickly become rich sources of new questions and problems.

- The classical barriers between formal definitions and intuitive ideas are changed into useful and precise specifications on how to move from the paper or blackboard to the Squeak world in a straightforward way.
- Algorithmic thinking and geometry, usually absent in the conventional approach, get included in the whole subject of study.

Squeak has been essential to our project not only for practical reasons (Smalltalk, free, platform-independent, morphic interface, etc.) but also for the presence and spirit of the Squeak community.

We want to share our experience of using Squeak to teach mathematics. Paraphrasing John Maloney, we want to say that nothing makes us happier than enabling students of all ages gain a deeper, more personal understanding of powerful ideas, and Squeak is showing us how to do that.