# X-expressions in XMLisp:
# S-expressions and Extensible Markup Language Unite

Alexander Repenning & Andri Ioannidou
University of Colorado, University of Lugano, AgentSheets Inc.
Boulder, Colorado, 80309-430
ralex@cs.colorado.edu, andri@agentsheets.com

## Abstract

XMLisp unites S-expressions with XML into *X-expressions* that unify the notions of data sharing with computation. Using a combination of the Meta Object Protocol (MOP), readers and printers, X-expressions uniquely integrate XML at a language, not API level, into Lisp in a way that could not be done with other programming languages. Integration at a language level has significant advantages by making XML tangible to the programmer throughout existing Lisp development tools including editors, debuggers, inspectors, listeners and compilers. This integration with Lisp tools enables XML development in the incremental development style Lisp programmers have become accustomed to. This article describes XMLisp in the context of the AgentCubes simulation and game-authoring tool. AgentCubes is the 3D version of AgentSheets system, which is the world's most distributed Lisp-based educational simulation and game-authoring tool.

***Categories and Subject Descriptors*** D.1.3 [Programming Techniques]: Concurrent Programming.

***General Terms*** Algorithms, Performance, Design, Human Factors, Languages, Theory.

***Keywords*** Object-Oriented Programming, XML, Meta object protocol, 3D tools.

## 1. Introduction

The Extensible Markup Language (XML) has established itself as a widely used representation format to share and process information. A fundamental question is how programming languages should integrate support for XML to simplify the development of XML reading, processing, and writing services. Programming languages differentiate themselves with respect to their support for introspection, serialization and object-orientation in order to work as agile XML programming environments. In this article we claim that Common Lisp has a unique set of features that allow it to move even further along this dimension. Common Lisp is capable to completely integrate XML at a language instead of just an API level.

*X-expressions* are the unification of S-expressions [1, 2] with XML. Conceptually speaking, X-expressions unify notions of computation with data sharing. Because of this type of integration, XML becomes much more tangible to developers, enabling the incremental development style Lisp programmers have become accustomed to. XML expressions can be evaluated in listeners, complete or sub-elements of XML can be evaluated in regular Lisp editors such as EMACS, and XML can even be compiled using the Common Lisp compiler. Existing tools, such as inspectors, will print XML expressions and allow users to interactively explore complex XML structures. Most CLOS objects can be directly be serialized into XML. All this is possible because of the combination of the Common Lisp Meta Object Protocol (MOP) for introspection with extensions to the print-object and macro dispatch character function for serialization. This combination makes Common Lisp an exceedingly effective platform to read, process and write XML expressions.

Most existing XML libraries in Common Lisp and other programming languages are based on either document models or event-driven models. The widely used Document Object Model (DOM) [3] is an approach that is platform and language agnostic. DOM captures the structure and content of XML files as tree structures. Various Common Lisp implementations for DOM exist, including the Allegro Common Lisp DOM Library [4]. Event-driven model approaches use callbacks when parsing XML documents. The Simple API for XML (SAX) [5] is the most commonly used event-driven model parser for XML. Common Lisp implementations for SAX include [6, 7]. Discussion of document model versus event-driven model XML processing can be found elsewhere [8-10]. However, our more specific concern discussed in this paper is the effective integration of a complete roundtrip of XML processing including reading, processing and writing. DOM-based APIs are well balanced between reading and writing, but provide limited options with respect to processing. Once a document tree structure is read, it can easily be written back into a file. A big source of conceptual as well as computational overhead is that the node objects part of the DOM tree structure may have little in common with the actual application objects. Consequently developers have to write transformation code converting application objects into DOM objects and back. The event handling callback functions of SAX-based XML parsers can be more directly set up to avoid the need for in-between objects. Instead, these callbacks may create application objects directly. The trade off, unfortunately, is that SAX-based APIs offer little support to write back application objects into XML. Developers may have to manually write application object printing code or keep a redundant DOM-like structure.

Our fundamental goal in developing XMLisp was to find an easy to use, highly customizable approach to establish a complete *two-way mapping* between application objects and XML representations. Faced with the DOM versus SAX tradeoff, we felt that there had to be a third option. There is nothing wrong with these approaches and indeed in many situations, for instance when XML representations and application objects are fundamentally different, these approaches may be the best ones to use. However, in cases where there is a structural similarity between application objects and XML representation, it should be possible to use Common Lisp's introspection, serialization and object-orientation features more directly. More specifically, it should be possible to – completely transparently – read an XML expression, turn it into a CLOS object, process that CLOS object and render it back to XML without needing to keep a DOM tree in memory. We will use Scalable Vector Graphics (SVG) as a simple example of X-expressions. SVG [11] is an XML-based vector format that can be rendered by many web browsers. Evaluating an XML expression representing an SVG circle shape in XMLisp will simply create a CLOS instance of a CIRCLE class with its slots set to the XML attribute values. The CLOS instance of the circle in the result of the evaluation below is printed in its serialized form, namely its XML representation:

```
? (eval <circle cx="62" cy="135" r="20"/>)
  <circle cx="62" cy="135" r="20"/>
```

XMLisp could also process XML-related markup languages such as HTML. More complex examples illustrating the two way mapping can be found throughout the paper, but at this point it is important to note some of the things the developer did not have to do:

• no need to explicitly invoke any XML API function

• no need to build and parse a DOM tree

• no need to setup callbacks

• no need to create custom XML print functions

• no need to interface with external resource descriptions

XMLisp is a portable implementation of X-expressions for Common Lisp implementations requiring minimal MOP support. It was initially created to build cross-platform 3D authoring tools, such as AgentCubes presented in section 4. In this context our more general requirements included the following:

***Make CLOS objects serializable as XML***: Serialization is useful for saving objects into, and restoring them back from, files; or for sending then through network protocols such as SOAP. This read-process-write cycle should be possible without the need to create other intermediate representations, such as a DOM tree node. Instead, the overhead of XML should be so minimal that in most cases the definition of a CLOS class alone is sufficient for reading in, processing, and writing out XML. To the degree that any CLOS object is serializable, it can be done into XML expressions; and in turn, any object created from an XML expression can be serialized out and read back in again as an equivalent CLOS object. Thus serialization into XML will in general be possible for a variety of individual CLOS instances as well as for other intricately linked networks of instances, where the slot values involved contain only objects that are serializable by normal means (which includes most 'vanilla' Lisp data types).

***Allow cross-platform object sharing***: It should be possible to serialize platform-specific objects, e.g., a button, as a platform-independent XML expression, <button text="OK" action="exit-application"/> and to read this expression back into a platform-specific object, i.e., a Windows button on Windows and an Aqua button on a Mac OS X.

***Lisp defined semantics***: It should be possible to use Lisp to define the computational meaning of an XML expression. Does the expression just represent data? Is the expression <circle r="20"/> simply a readable serialization of a circle instance with a slot called "r" of value 20? If so, an X-expression representing a circle can be a constant by means of being a self-evaluating object. Alternatively, the meaning of an XML expression may include developer-defined computation. For instance, reading <window title="Agent"/> may create an actual window with all its additional information and computational side effects.

However, an XML expression does not have to evaluate to itself. For instance, if the expression <add value1="3" value2="4"/> represents a numerical operation it may evaluate into a regular Lisp object such as the number 7. Finally, the semantics of an XML expression may include notions of transformation returning a processed version of the original XML expression. For instance the expression

```
<simplify_sum>
  <add value1="x" value2="2"/>
  <subtract value1="x" value2="3"/>
</simplify_sum>
```

could return

```
<subtract value1="x" value2="1"/>
```

To achieve such semantic transformations in XMLisp, eval reads the X-expression, turns it into a CLOS object, and then applies the read-return-value method before it prints the result. The default behavior is to return the object just read in, but when necessary, the object is transformed according to the needs of the application (see transformer example in section 4.4).

***Object-Oriented Customization***: It should be fully possible to benefit from object-oriented programming such as mutimethods [12, 13] to customize reading, processing, and writing of XML. Consider this SVG example of a group shape containing a circle and a rectangle object:

```
<g id="world">
  <circle cx="5" cy="5" r="2"/>
  <rect x="15" y="15" width="100" height="50"/>
</g>
```

Because the element names are directly mapped to class names (g, circle and rect), the developer can define methods on these classes to customize reading, processing, and writing. For instance using the multimethod add-subobject ((group g) (shape shape)), the developer can aggregate shapes into groups as arrays instead of lists.

***Provide an incremental approach towards Lisp***: Being also educators we had our share of challenges to convey the beauty and the power of Lisp to computer science students. We have been thinking of X-expressions as stealth approach to get students burned by Lisp exposure in the context of some introduction to programming languages course re-introduced to Lisp. Initially they do not know that the X-expressions they are editing are directly evaluated in Lisp. Editing XML-based game levels, user interfaces and agent behaviors is considered "cool." Once the notion of X-expressions is revealed to the students then many are interested to give Lisp a second chance to explore computational extensions of their work.

This article does not focus on particular technical aspects of XMLisp. No claim is made here that XMLisp in its current implementation is faster or more complete than existing XML tools for Common Lisp. What is important is that, at a conceptual level, Common Lisp has a unique set of features allowing it to integrate XML in a way that most other languages could not. The following sections of this paper describe how X-expressions work and illustrate the requirements above in the context of the 3D authoring tool AgentCubes [14].

## 2. X-expressions: Syntax and Semantics

X-expressions are the extended set of expressions that unite S-Expressions with XML expressions. This section provides a conceptual overview of the fundamental mechanism of X-expressions at syntactic and semantic levels. Please note that we will not provide a low-level description of how XMLisp was implemented. XMLisp is available as an open source project [15] with existing ports to MCL, OpenMCL, Allegro Common Lisp.

### 2.1 Syntax

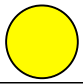
At the syntactic level, Common Lisp can be extended through readers and printers. If the goal is to be able to read and write XML expressions as part of Lisp, we can define custom readers to deal with characters such "<" and ">" in special ways. This needs to be done with some care to avoid conflicts with the already established meaning of these characters in regular S-expressions. XMLisp provides the developer with controls to turn the X-expression reader on or off in selected packages.

X-expressions are read into the current package (or a specified package, passed as a parameter to the load-object method), unless package prefixes are explicitly used. Namespaces are then accessed via package prefixes.

The Lisp reader turns XML expressions into instances of CLOS objects. It employs the MOP as source of meta-information to de-serialize an object. When reading <circle cx="62" cy="135" r="20"/>, the reader will look if there is a CIRCLE class and whether it has the necessary slots that match attribute names or subelements. It will use any available meta-information, such as the slot type, to decode XML strings into proper Lisp data types and values. The object created is a proper application object with all its user-defined accessors and methods. Therefore, the developer can invoke regular accessor functions as X-expressions to access slot values of these objects.

```
? (cx <circle cx="62" cy="135" r="20"/>)
62
```

The accessor returns the number object and not the string "62" because the class definition of circle includes number slot types. Slot types, if present, are meta information used to automatically dispatch CODECs (coders and decoders). XMLisp implements a large number of CODECs for basic data types such as numbers, strings, Booleans and composites such as lists, arrays, and hash tables. Developers can create custom CODECs and extend existing ones.

|  |  |
|---|---|
| XML | `<circle cx="62" cy="135" r="20"/>` |
| CLOS | (defclass CIRCLE (xml-serializer)<br>  ((cx :accessor cx :type number :documentation "center x")<br>   (cy :accessor cy :type number :documentation "center y")<br>   (r :accessor r :type number :documentation "radius"))<br>  (:documentation "SVG circle"))<br><br>(make-instance ‘circle :x 2) |
| Browser |  |

To be consistent with the read-eval-print loop (REPL) and more generally to be able to produce XML output, CLOS instances representing XML objects must be able to print themselves as valid XML expressions. The circle class in the example above is a subclass of xml-serializer, which adds the ability for the object to print itself in XML instead of being an unprintable CLOS object.

Extending reading and printing in Lisp is extremely powerful. Reading XML from the listener, files, or the network will create CLOS objects. This means developers can, very much in the explorative spirit of general Lisp programming, experiment with bits and pieces of XML expressions by typing them into the listener or evaluating X-expressions in a text editor. Extended printing means that CLOS instances representing XML objects will automatically show up as XML in all kinds of Lisp tools such as inspectors, tracers, graphers and stack traces. Files containing X-expressions can be compiled with the regular Lisp compiler. Compiled X-expression files are no longer sharable between platforms, but typically load significantly faster as they avoid the parsing overhead of XML.

An additional benefit of integrating XML expressions into Lisp through custom readers and writers is that in-lining XML expressions into Lisp source code can be done without experiencing significant constraints from existing Lisp readers. Developers could not simply copy and paste valid XML expressions into Lisp source. Instead, they would have to edit these XML expressions to make them compatible with the Lisp reader. For instance, if a one would try to inline an XML expression as a string, e.g., (setq Link "<a href="http://www.agentsheets.com">") then it would be necessary to precede the double quote characters with a backslash escape character to allow the Lisp reader to recognize the entire "<a href="http://www.agentsheets.com">" expression as a single string. A custom reader does not inherit these constraints. It can read arbitrary content including Unicode characters.

### 2.2 Semantics

At the semantic level, structure and content of XML expressions needs to be mapped to CLOS instances. The MOP includes powerful introspection mechanisms to analyze the structure of CLOS objects, slot names, slot type any many other useful pieces of meta-data that can be employed to establish automatic mapping between CLOS instances and XML expressions.

For XMLisp to be able to deal with XML it uses the MOP to try to establish correspondence. When reading the element name "circle" XMLisp looks in the current package for a class of the same name. Then, reading the element attributes XMLisp looks up slot definitions of the circle class to find slots with matching names. It parses and decodes the value representations in the circle case simply numbers. In simple cases like this, the developers only need to create a class definition in order to create an XML interface.

Some XMLisp users have suggested achieving correspondence by automatically generating the class definition and necessary methods from the XML Document Type Definition (DTD). Given that the number of applications using DTDs is decreasing and even the designer of XML has publicly admitted that DTD was a mistake [16], we find it more suitable to use the Lisp class definition as a kind of document type specification. One could imagine building readers for automatically creating CLOS class definitions from schemas similar to XML Schema [17, 18], but XMLisp does not currently feature this.

The correspondence for document specification in XMLisp is based on 3 design principles:

1) ***Use Good Defaults***: Start with assumptions as simple as possible, such as that there should be a 1:1 correspondence between attribute names and slot names. Only require the definition of methods to deal with exceptions.

2) ***Incremental Refinement***: Allow developers to refine the correspondence incrementally by gradually providing more meta-information. For instance, the type of a slot may be deduced from an :initform value or provided explicitly by a developer through a :type slot keyword. For instance, by adding a :type boolean slot specification, XMLisp can print slot values as "true" and "false" instead of "nil" and "t." Developers can introduce their own types and define CODECs to print and read custom types accordingly.

3) ***Extensible Architecture***: All assumptions and mappings should be embodied as methods that can be extended and overwritten by the developer. To avoid name conflicts and to achieve higher flexibility it is essential that developers can change every aspect of how XMLisp interprets and processes XML, such as aggregating subelements in various forms, printing, and transforming. In order to define CLOS classes to be XML serializable, developers mix in the xml-serializer class. To extend and overwrite the default behavior developers have a large set of methods.

## 3. Related Work

The space of XML related work is vast. Numerous implementations of document model and event-driven model parsers and generators for all kinds of programming languages exist. In this section we limit our discussion to XML systems involving a combination of introspection and serialization. Lisp-based related systems include SWCLOS and CL-XML. Non-Lisp-based related systems include Ruby, JAXB and Water.

SWCLOS [16] is a Common Lisp based semantic web processor using the MOP to change the behavior of classes through meta-classes. SWCLOS reads and parses RDF tags even lazily creating CLOS class stubs for classes that are later defined in the file. SWCLOS does not integrate XML at a language level into Lisp via printers and readers, however. In the case were there is a RDF specification available SWCLOS and XMLisp could be effectively combined.

CL-XML [19] includes experimental MOP-based extensions to serializing arbitrary CLOS instances but does not include language integration via print and read extensions.

Ruby, with its bundled REXML library, is perhaps closest to XMLisp in that it also integrates XML at a language level using print/load marshalling mechanisms comparable to Common Lisp. Similar to XMLisp, Ruby allows developers to use interactive debuggers (corresponding to the Lisp listener) to read and print XML expressions. However, the Ruby load marshaling method is substantially more restrictive. To run the specialized XML parser, Ruby needs to first create a DOM-node like object and then run its load method to parse a string representing an XML expression. Consequently, Ruby does not directly create application objects. For instance, Ruby does not read an XML expression such as <circle r="20"/> and directly turn it into an instance of a circle class. Lisp, thanks to its macro reader function, can parse the expression up to the delimiter after the element name, i.e., "<circle ", create an instance of the circle object and set its attribute and element values. Ruby, in contrast, only creates a generic DOM-node object, which the programmer would have to turn into an application object. Ruby does include a powerful introspection mechanism that could be employed to map XML expressions directly to application objects instead of to DOM nodes. However, the lack of a macro character dispatch mechanism prevents Ruby from seamlessly integrating XML at a language level.

JAXB [20] generates Java classes from a schema. It is similar to XMLisp in that it creates structures that are more compact than DOM approaches. JAXB requires external Meta description to generate code that can de-serialize XML files into objects. Unlike X-expressions JAXB does not provide means to merge data sharing and computation into a single representation. That is, Java code and XML markup remain strictly separate in JAXB.

Water [21, 22] is a programming language with an XML-inspired syntax used for creating web services. Water has its conceptual roots in various dynamic programming languages including Lisp. Similar to X-expressions Water blurs the boundary between data sharing and computation. XMLisp could be used to make a Water-like programming language.

## 4. AgentCubes: a Game Authoring System using X-expressions

The function and value of X-expressions are best understood when explained in the context of the system originally inspiring them. AgentCubes is a highly extensible simulation and game-authoring architecture (Figure 1) capable of addressing a large range of application domains.

The design and implementation AgentCubes was originally the main motivation to build XMLisp. At various levels of the architecture we needed a versatile framework capable of accessing and modifying extendible data and meta-data. Specifically, we needed to represent file structures, media meta-information, visual programming code, game engine states, scene graphs, 3D models, and reusable patterns as objects that could be efficiently serialized. At the same time we wanted to employ a cross-platform and cross-language serialization format suited for exchange with other tools. XML seemed ideally suited for this purpose. However, our previous experience with XML in our AgentSheets authoring tool [23-26] in Lisp and Java indicated that a lot of effort was required to create and maintain XML-based representations. Existing DOM and SAX implementations require substantial development overhead to create efficient read, process, write round-trip interfaces between objects and XML files. We felt that it should be possible to let the programming language do most of the labor by eliminating the need of using intermediate representation objects such as DOM nodes and writing custom XML print functions. Hence, XMLisp was developed.

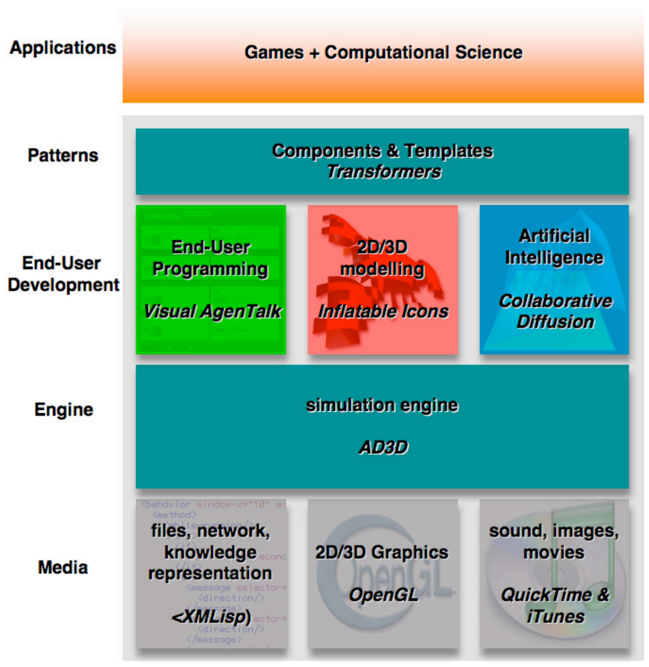


**Figure 1: The AgentCubes Architecture.**

The AgentCubes architecture (Figure 1) uses X-expressions at all levels:

- *Media*: XMLisp resource descriptions for sounds, textures, fonts.
- *Engine*: XMLisp definitions of agents and 3D scenes.
- *End-User Development*: XMLisp representations of simple and complex GUIs as well as the end-user visual programming language used to define agent behaviors.
- *Patterns*: XMLisp definitions of transformers used to instantiate components and behaviors from templates.
- *Applications*: XMLisp used for inter-application, cross-platform communication between CL-HTTP and Flash and for location-aware GPS and GIS based communication in Mobility Agents.

The following sections provide examples of how X-expressions are used at these different levels.

### 4.1 Media Level: accessing and describing resources

At the media level, XMLisp is used to capture resources, including links to media such as images, sounds, cursors, and texture files, as well as meta-information describing these resources.

A simple example showing how XMLisp deals with object-oriented customization is fonts. OpenGL has very limited text support pushing most of the burden of rendering text to the developer. A simple texture-based font can be captured as the combination of a texture map (Figure 2) with glyph information indicating the symbol name and boundary box for each character.

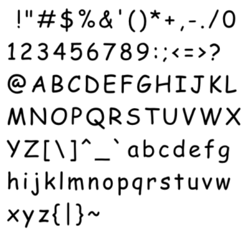


**Figure 2: Texture map of Comic Sans ASCII font**

A font class captures meta-information about the font including the name of the font. The glyphs slot is used to store sets of glyphs.

```
(defclass FONT (xml-serializer)
  ((font-name :accessor font-name :initform "helvetica")
   (font-size :accessor font-size :initform 16)
   (start :accessor start :initform 32 :type integer)
   (end :accessor end :initform 127 :type integer)
   (texture :accessor texture :initform nil)
   (texture-size :accessor texture-size :type integer)
   (glyphs :accessor glyphs :type array))
  (:documentation "An antialiased font based on a font specification"))
```

A glyph contains a name and the coordinates of the bounding box in the font texture map. Defining the coordinates to be of type float ensures parsing of string attribute values in XML into valid floats through a float CODEC.

```
(defclass GLYPH (xml-serializer)
  ((name :accessor name :initform #\Space :type
character)
   (x0 :accessor x0 :type float)
   (y0 :accessor y0 :type float)
   (x1 :accessor x1 :type float)
   (y1 :accessor y1 :type float))
  (:documentation "A glyph represents a character symbol in a font"))
```

A complete font description then becomes a simple X-expression, which can be saved as an XML file:

```
<font font-name="Comic Sans MS" font-size="49">
  <glyph x0="0.0" y0="0.8671875" x1="0.029296875"
y1="1.0"/>
  <glyph name="!" x0="0.041015625" y0="0.8671875"
x1="0.064453125" y1="1.0"/>
  <glyph name="&quot;" x0="0.076171875" y0="0.8671875"
x1="0.1171875" y1="1.0"/>
  <glyph name="#" x0="0.12890625" y0="0.8671875"
x1="0.208984375" y1="1.0"/>
  <glyph name="$" x0="0.220703125" y0="0.8671875"
x1="0.287109375" y1="1.0"/>
… etc, etc, …
</font>
```

The automatic aggregation of glyph elements is based on name mapping. When attempting to add a glyph subelement to the font element, XMLisp will look for a symbol name with the plural form of glyph (glyph -> glyphs). Of course, this may not be what the developer had intended. In these cases, the developer could overwrite the add-subelement (font, glyph) multimethod to aggregate glyphs into a different slot of font or use a completely different aggregation scheme altogether.

## 4.2 Engine Level: representing agents and scenes

At the game engine level of AgentCubes, agent instances and entire scenes containing agents need to be managed. Agents are contained in scenes, which in turn are contained in windows representing 3D worlds. Agents include methods to render themselves in 3D using OpenGL functions. Evaluating X-expressions based on agents creates editable 3D worlds.

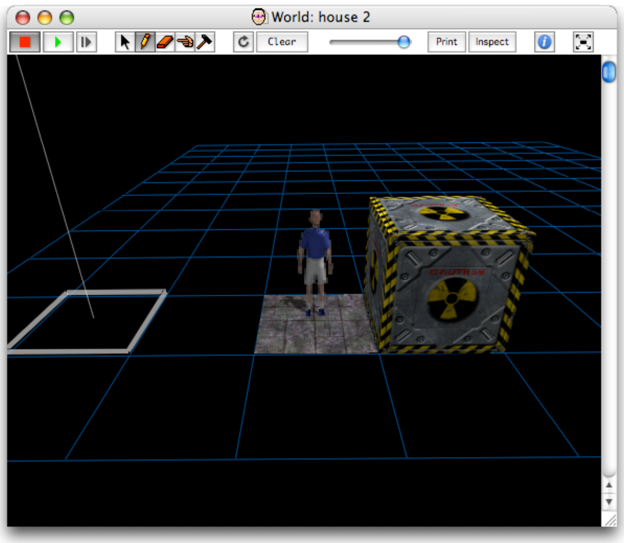
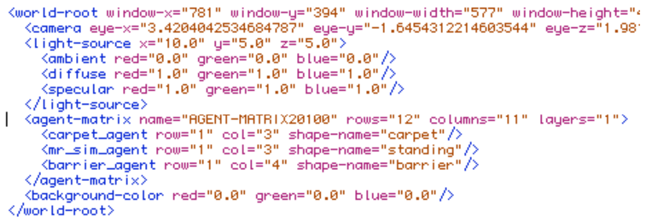


**Figure 3: Editable 3D AgentCubes World containing three agents (person, tile and box).**

The main object contained in the world is the agent-matrix, which is a three-dimensional cube containing stacks of agents. In addition to the visible objects in the world, additional scene objects such as cameras and light sources exist in the 3D scene. Both visible and invisible objects and world attributes are captured in the XML representation of the world:



```
<world-root window-x="781" window-y="394" window-width="577" window-height="
  <camera eye-x="3.420404253468478" eye-y="-1.645431221460354" eye-z="1.98
  <light-source x="10.0" y="5.0" z="5.0">
    <ambient red="0.0" green="0.0" blue="0.0"/>
    <diffuse red="1.0" green="1.0" blue="1.0"/>
    <specular red="1.0" green="1.0" blue="1.0"/>
  </light-source>
| <agent-matrix name="AGENT-MATRIX20100" rows="12" columns="11" layers="1">
    <carpet_agent row="1" col="3" shape-name="carpet"/>
    <mr_sim_agent row="1" col="3" shape-name="standing"/>
    <barrier_agent row="1" col="4" shape-name="barrier"/>
  </agent-matrix>
  <background-color red="0.0" green="0.0" blue="0.0"/>
</world-root>
```

Running simulations and games will result in different configurations of the world depending on the specified behavior of the objects in the world. Moreover, through tools available in AgentCubes, the user can add, remove, copy, and move agents in the world. At any point in time, the current state of the world can be printed again and stored in an XML file.

## 4.3 End-User Development Level

At the end-user development level, XMLisp is used to create user interfaces from simple to complex and capture elements of high-level end-user programming languages used to define agent behaviors in simulations and games.

### 4.3.1 Simple GUIs: Windows & Dialog Items

X-Expressions can be turned into complete graphical user interfaces. We have wrapped dialog items, such as buttons, menus, sliders, and pop up menus, found in OS X (MCL) and Windows (Allegro Common Lisp), into a portable set of GUI components. Using XMLisp methods, the platform-specific versions of the components serialize themselves into platform independent X-expressions. For instance,

```
<application-window>
  <row align="right" valign="bottom" padding="20">
    <button text="Maybe"/>
    <cancel-button/>
    <ok-button/>
  </row>
</application-window>
```

creates an APPLICATION-WINDOW instance which manifest itself as window on the screen in Mac OS X (Figure 4).

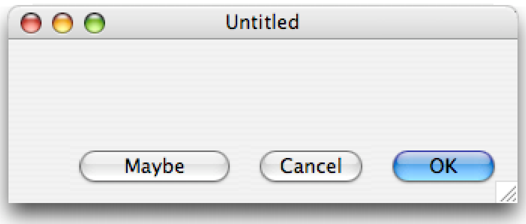


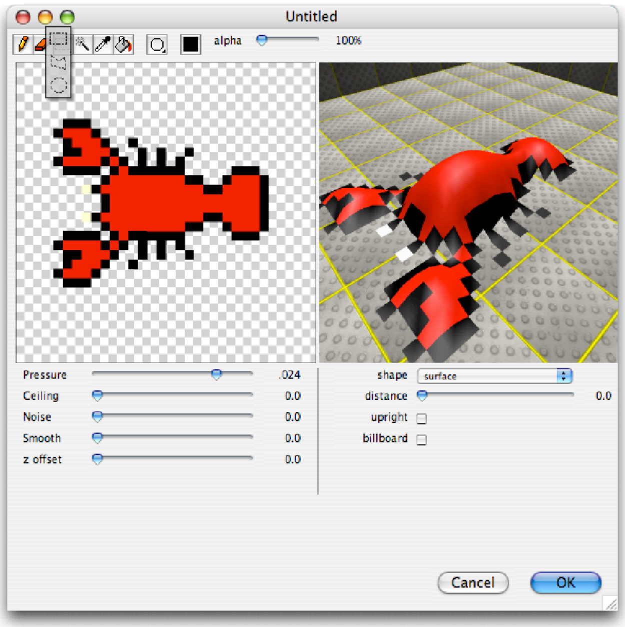
**Figure 4: A simple application window generated from an X-Expression in MCL for OS X**

Defining such dialogs in XML simplifies cross-platform compatibility for Lisp applications running on different platforms. It is the semantics for what the application-window element upon parsing that has the side effect of creating the window. However, creating the GUI element may need to be done differently on different platforms.

### 4.3.2 Complex GUIs: Inflatable Icons editor

A substantially more complex example of a user interface created with X-expressions is the Inflatable Icon editor. Inflatable Icons [27] is a new, patent-pending, technique that interactively extrudes 2D pixel-based images into polygon-based 3D models. Through the use of a diffusion-based inflation process with input from users (e.g. inflation pressure, symmetry, noise) suitable 2D artwork can serve as input for an interactive 2D to 3D transformation process.

The user interface for manipulating the inflation parameters (Figure 5) is generated using XMLisp. An X-expression defines the all the editor elements (tools, sliders, 2D image editor and inflated 3D viewer, checkboxes, buttons etc) and their relative positioning.



Pressure .024
Ceiling 0.0
Noise 0.0
Smooth 0.0
z offset 0.0
shape surface
distance 0.0
upright ☐
billboard ☐

Figure 5: Inflatable Icon Editor built with XMLisp GUI

For instance, the Selection Tool for selecting parts of the 2D image (shown extended in the upper left corner of Figure 5) has choices for selecting rectangular, polygonal, or circular areas. It is defined in XML as:

```
<choice-image-button tooltip="Selection Tool">
  <image-choice image="select-rect-button.png"
action="rect-tool-action"/>
  <image-choice image="select-polygon-button.png"
action="polygon-tool-action"/>
  <image-choice image="select-ellipse-button.png"
action="ellipse-tool-action"/>
</choice-image-button>
```

The 2D image editor and 3D inflated model viewer are defined as XMLisp X-expressions as:

```
<row-of-squares>
  <2d-icon-editor name="icon-editor" img="lobster.jpg"
img-height="32" img-width="32" action="change-icon-
action"/>
  <inflated-icon-editor name="model-editor"/>
</row-of-squares>
```

The slider for controlling the inflation's pressure and its associated text is defined as an X-expression:

```
<row align="stretch">
   <label width="65" text="Pressure"/>
   <slider action="adjust-pressure-action"/>
   <label name="pressuretext" align="right" width="35"
text="0.0"/>
</row>
```

#### 4.3.3 Visual Programming Languages and editors

As an end-user authoring tool, AgentCubes features an end-user visual programming language called Visual AgenTalk based on our previous work with AgentSheets [25]. The language elements consist of conditions and actions that can be combined into rules and in turn into methods. An agent's behavior consists of a collection of these methods. Code at any level is expressed in XML. For instance, the Next-to condition that can be used by an agent to check for the existence of any number of the specified agents around it is expressed as shown in Table 1.

As XMLisp reads the XML representation of the language it uses an "expand" macro to create the corresponding Lisp code (Table 1). Each language element produces its own Lisp expansion. Nested elements expand recursively to generate Lisp representations of the code that can then be executed by the engine in AgentCubes.

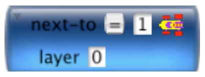
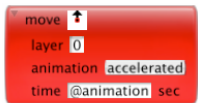
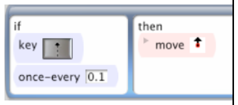
| Command | X-expression | Lisp |
|---|---|---|
|  | `<next-to operator="="`<br>`number="1" shape="car2"/>` | `(NEXT-TO SELF`<br>`'= 1 '\|car2\|`<br>`0)` |
|  move<br>layer<br>0<br>animation<br>accelerated<br>time<br>@animation<br>sec | `<move`<br>`duration="@animation">`<br>`   <direction drow="-1"/>`<br>`</move>` | `(MATRIX-MOVE`<br>`SELF 1 0 0`<br>`(GET-THE-`<br>`PROPERTY-`<br>`VALUE`<br>`'ANIMATION)`<br>`'ACCELERATED-`<br>`TRANSLATION-`<br>`ANIMATOR)` |
|  | `<rule>`<br>`  <if>`<br>`    <key label="up`<br>`arrow"/>`<br>`    <once-every`<br>`seconds="0.1"/>`<br>`  </if>`<br>`  <then>`<br>`    <move`<br>`duration="@animation">`<br>`      <direction`<br>`drow="1"/>`<br>`    </move>`<br>`  </then>`<br>`</rule>` | `(when (and`<br>`(KEY-WAS-`<br>`PRESSED SELF`<br>`126)`<br>` (TIMER-DUE-P`<br>`SELF 100))`<br>`  (progn`<br>`(MATRIX-MOVE`<br>`SELF 1 0 0`<br>` (GET-THE-`<br>`PROPERTY-`<br>`VALUE`<br>`'ANIMATION)`<br>`'ACCELERATED-`<br>`TRANSLATION-`<br>`ANIMATOR)))` |

Table 1: Equivalent VAT language pieces at the GUI, XML, and Lisp levels

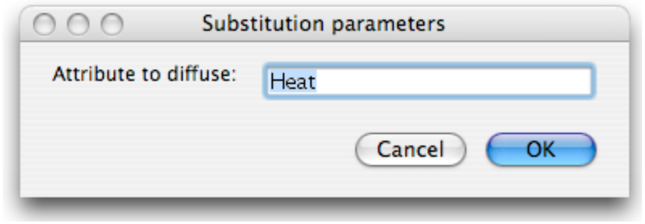
### 4.4 Pattern Level: transformers

AgentCubes features template-based programming [28] both for creating shapes and behaviors for agents. This is achieved through transformers that change general template XML into concrete XML based on user-defined parameters. For instance, a transformer template for diffusion calculation behavior is defined as follows:

```
<transformer>
  <substitution>
    <replace what="$diffusion-var$" how="substitute-with-
user-value">
      <user-parameter name="Attribute to diffuse"
type="editable-text" description="Enter the name of the
attribute you wish to diffuse." default="heat"/>
    </replace>
  </substitution>
  <list>
```

```
    <method comments="diffuse the attribute
&quot;$diffusion-var$&quot;">
      <on selector="Diffuse"/>
        <rule>
          <if/>
          <then>
            <set attribute="$diffusion-var$" value="0.25
* ($diffusion-var$[left] + $diffusion-var$[right] +
$diffusion-var$[up] + $diffusion-var$[down])"/>
          </then>
        </rule>
    </method>
  </list>
</transformer>
```

The transformer represents a substitution by defining what to replace and how. In this example, we are to replace the diffusion variable that will be given by the user. As XMLisp parses the transformer, the user is prompted to enter the attribute to diffuse as shown below.



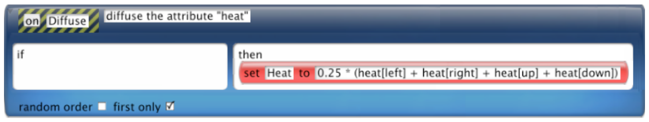
Substitution parameters

Attribute to diffuse: Heat

The user decides to create heat diffusion, thus the “Heat” attribute is entered. Upon successful user input, XMLisp calls the read-return-value multimethod specialized for the transformer to perform the substitution and yield new XML immediately usable in the agent’s behavior:

```
<method comments="diffuse the attribute
&quot;heat&quot;">
  <on selector="DIFFUSE"/>
    <rule>
      <if/>
      <then>
        <set attribute="Heat" value="0.25 * (heat[left] +
heat[right] + heat[up] + heat[down])"/>
      </then>
    </rule>
  </method>
```

This XML added to the agent’s behavior visually manifests itself in the behavior editor as a new method shown below.



on Diffuse diffuse the attribute "heat"

if

then

set Heat to 0.25 * (heat[left] + heat[right] + heat[up] + heat[down])

random order ☐ first only ☑

Transformers are similar to XML transformation in XSLT and other XML transformation languages. The main difference is that processing is integrated in the parsing and does not need to first build a source tree from input XML and then on the second pass process the XML like XSLT.

## 4.5 Application Level: cross-platform data sharing

At the application level, XMLisp is used to process network communications between clients and servers that are implemented in different languages and run on different platforms. For instance applications, such as Mobility Agents include a server running CL-HTTP interfacing with relational databases and Flash clients running on handheld computers.

### 4.5.1 Mobility Agents

Mobility Agents [29] is a location-aware technology that leverages existing GPS infrastructure and GIS information to compute highly personalized information and deliver it on PDAs and cell phones. It was developed to provide multimodal prompts on handheld devices to travelers with cognitive disabilities helping them to recognize, for instance, the right bus to reach a specified destination. At the same time, it communicates the trip status and location of the traveler in relation to known landmarks and street addresses to a caregiver.

The Mobility Agents system employs multiple mission status interface approaches on servers and clients, ranging from 3D real-time visualizations (Figure 6 right) and Flash traveler interfaces (Figure 6 left) to SMS and instant messaging-based text interfaces (Figure 7).

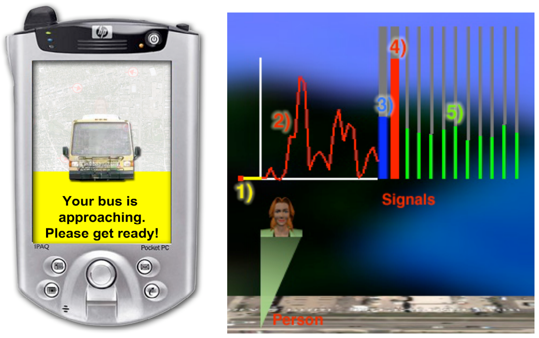


**Figure 6: Traveler (left) and caregiver interface detail showing signals associated with a traveler (right). Information includes 1) heading (direction the person is moving); 2) network lag time graph; 3) Phone signal level; 4) battery level; 5) GPS satellites and signal strength.**

Location and status information is sent from other web servers and the Flash clients to the Lisp-based Mobility Agent server via XML messages. Here again X-expressions are used for cross platform (desktop computer versus PDA) and cross language (Common Lisp versus Flash) development. For instance, bus GPS location updates sent by buses equipped with GPS via networks to web servers accessible to our Mobility Agents:

```
<bus-gps-event latitude="40.016415S0" longitude="-
105.26298S0" timestamp="77388.0"
heading="142.1999969482422" bus-id="9063225"/>
```

Traveler GPS location update, including information about the available GPS satellites picked up by the receiver:

```
<client-gps-event latitude="39.98525" longitude="-
105.249797" timestamp="191043.296875"
```

```
speed="9.337554931640625" heading="147.4600067138672"
altitude="1635.1">
    <satellite prn="2" snr="50"/>
    <satellite prn="7" snr="33"/>
    <satellite prn="6" snr="37"/>
    <satellite prn="5" snr="48"/>
    <satellite prn="17" snr="45"/>
    <satellite prn="24" snr="31"/>
    <satellite prn="121"/>
</client-gps-event>
```

Client status update, including phone signal and battery level:

```
<device-status>
  <phone-signal-level val="16383"/>
  <battery-life-percent main="86" backup="100"/>
</device-status>
```

The above XML messages result in updating the 3D visual interface displaying buses and travelers on a map as well as the traveler's specific display with the appropriate signals as shown in Figure 6, right.

The Mobility Agents system also includes an XML-based GIS database creation and maintenance mechanism. Using GIS information like the sample shown below for a few locations in Boulder, CO, mobility agents can display were the traveler is in relation to known locations or landmarks in a city and deliver the Context-Aware Instant Messages to the caregivers.

```
<geographic-information-system name="Boulder">
  <location sub-type="Indian" type="Restaurant"
street="619 S Broadway St" name="Tandoori Grill"
longitude="-105.2483" latitude="39.9841"/>
  <location sub-type="Government" type="city offices"
street="1739 Broadway St" name="Boulder Zoning
Inspection" longitude="-105.2796" latitude="40.0161"/>
  <location sub-type="K-12" type="School" street="805
Gillaspie Dr" name="New Vista Senior High School"
longitude="-105.252" latitude="39.9826"/>
<location type="movie theater" street="2985 Pearl St."
name="Mann Crossroads 6" longitude="-105.2541"
latitude="40.0233"/>
  <location type="park" state="CO" name="Carpenter Park"
longitude="-105.25416666666666"
latitude="40.01166666666666"/>
  <location type="hospital" state="CO" name="Boulder
Medical Center" longitude="-105.28333333333333"
latitude="40.02611111111111"/>
  <location type="Street Address" state="CO" zip="80303"
house-number="3302" street="Apache Rd" name="3302 Apache
Rd" longitude="-105.24903057957447"
latitude="39.995227855172416"/>
-- etc, etc… --
</geographic-information-system>
```

For instance, the traveler Melanie, which is represented by a buddy in an Instant Messaging application (Figure 7), updates her status as she is moving through town in the bus or on foot. The status message includes "Domino's Pizza", the name of a restaurant, and "1000 Euclid Avenue", a street address in Boulder. This real-time information is important to describe Melanie's location to her caregiver.

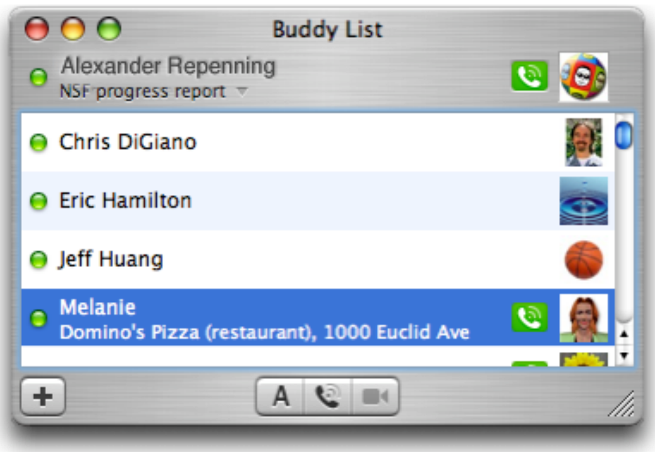


**Figure 7: A buddy list for an Instant Messaging application (Apple iChat in OS X)**

The XMLisp ability to integrate XML into Lisp in such a seamless way made the creation of applications such as the Mobility Agents much more viable.

## Conclusions

X-expressions unify S-expressions with XML. By making full use of Common Lisp introspection and serialization mechanisms, XMLisp uniquely integrates XML into a programming language. XML expressions can be evaluated in listeners; complete XML elements or sub-elements can be evaluated in regular Lisp editors such as EMACS; and XML can even be compiled using the Common Lisp compiler. Existing tools such as inspectors will print XML expressions and allow users to interactively explore complex XML structures. CLOS objects can be directly be serialized into XML. All this is possible because of the combination of the Common Lisp Meta Object Protocol for introspection with extensions to the print-object and macro dispatch character function for serialization. This combination makes Common Lisp an exceedingly effective platform to read, process and write XML expressions not only locally within an application, but also cross multiple platforms and via networks. This kind of integration could not be achieved with other languages. This work is still in an early stage but perhaps it can be perceived as small step towards Lisp becoming "the Lisp of the Internet".

## 7. Acknowledgments

This material is based in part upon work supported by the National Science Foundation and the National Institutes of Health under Grants Numbers 0349663 and 0205625. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the National Institutes of Health.

## 8. References

[1] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Communications of the ACM*, vol. 3, pp. 184 - 195, 1960.

[2] Wikipedia, "S-expression", http://en.wikipedia.org/wiki/S-expression.

[3] W3C, "Document Object Model (DOM)", http://www.w3.org/DOM/.

[4] Franz, "Document Object Model (DOM) in Allegro Common Lisp", http://www.franz.com/support/documentation/8.0/doc/dom.htm.

[5] "SAX", http://www.saxproject.org/.

[6] "SSAX", http://sourceforge.net/projects/ssax, http://ssax.sourceforge.net/.

[7] Franz, "A Sax XML Parser for Allegro Common Lisp", http://www.franz.com/support/documentation/8.0/doc/sax.htm.

[8] R. Muvva, "DOM Vs SAX What is best?" http://www.code101.com/Code101/DisplayArticle.aspx?cid=37.

[9] Wikipedia, "Simple API for XML", http://en.wikipedia.org/wiki/Simple_API_for_XML.

[10] Wikipedia, "Document Object Model", http://en.wikipedia.org/wiki/Document_Object_Model.

[11] "Scalable Vector Graphics (SVG)", http://www.w3.org/Graphics/SVG/.

[12] D. H. H. Ingalls, "A simple technique for handling multiple polymorphism," presented at Conference on Object Oriented Programming Systems Languages and Applications, Portland, Oregon, USA, 1986.

[13] Wikipedia, "Multiple dispatch", http://en.wikipedia.org/wiki/Multimethods.

[14] A. Repenning and A. Ioannidou, "AgentCubes: Raising the Ceiling of End-User Development in Education through Incremental 3D," presented at IEEE Symposium on Visual Languages and Human-Centric Computing, 2006, Brighton, United Kingdom, 2006.

[15] AgentSheets Inc., "<XMLisp)", http://www.agentsheets.com/lisp/XMLisp/.

[16] J. Gray, "A conversation with Tim Bray," *ACM Queue*, vol. 3, pp. 20 - 25, 2005.

[17] A. Møller and M. I. Schwartzbach, "Schema Languages," in *An Introduction to XML and Web Technologies*: Addison-Wesley, 2006.

[18] W3C, "XML Schema", http://www.w3.org/XML/Schema.

[19] J. Anderson, "CL-XML", http://pws.prserv.net/James.Anderson/XML/.

[20] "JAXB", http://java.sun.com/webservices/jaxb/.

[21] M. Plusch, Water: Simplified Web Services and XML Programming: Wiley, 2002.

[22] "Water Language", http://www.waterlanguage.org/.

[23] A. Ioannidou and A. Repenning, "End-User Programmable Simulations," *Dr. Dobb's*, pp. 40-48, 1999.

[24] A. Repenning and A. Ioannidou, "Agent-Based End-User Development," *Communications of the ACM*, vol. 47, pp. 43-46, 2004.

[25] A. Repenning and J. Ambach, "Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing," presented at Proceedings of the 1996 IEEE Symposium of Visual Languages, Boulder, CO, 1996.

[26] A. Repenning and A. Ioannidou, "Behavior Processors: Layers between End-Users and Java Virtual Machines," presented at Proceedings of the 1997 IEEE Symposium of Visual Languages, Capri, Italy, 1997.

[27] A. Repenning, "Inflatable Icons: Diffusion-based Interactive Extrusion of 2D Images into 3D Models.," *The Journal of Graphical Tools*, vol. 10, pp. 1-15, 2005.

[28] A. Ioannidou, "Programmorphosis: a Knowledge-Based Approach to End-User Programming," presented at Interact 2003: Bringing the Bits together, Ninth IFIP TC13 International Conference on Human-Computer Interaction, Zürich, Switzerland, 2003.

[29] A. Repenning and A. Ioannidou, "Mobility Agents: Guiding and Tracking Public Transportation Users," presented at Proceedings of AVI 2006 Advanced Visual Interfaces International Working Conference, Venice, Italy, 2006.