Anatomy of Lisp

John Quigley
www.jquigley.com
jquigley@jquigley.com

Chicago BarCamp, 2007

The Considered life

"The unconsidered life is not worth living."

— Socrates

"That language is an instrument of human reason, and not merely a medium for the expression of thought, is a truth generally admitted."

– George Boole

What Is The Right Tool?

1. **Memory management**: garbage collection
2. **Object oriented**: modularity and encapsualtion
3. **Egalitarianism**: first class everything
4. **Libraries**: great stdlib, powerful third-party facilities
5. **Introspection**: program available as date

What Is The Right Tool?

1. **Memory management**: garbage collection
2. **Object oriented**: modularity and encapsualtion
3. **Egalitarianism**: first class everything
4. **Libraries**: great stdlib, powerful third-party facilities
5. **Introspection**: program available as date

What Is The Right Tool?

1. **Memory management**: garbage collection
2. **Object oriented**: modularity and encapsualtion
3. **Egalitarianism**: first class everything
4. **Libraries**: great stdlib, powerful third-party facilities
5. **Introspection**: program available as date

What Is The Right Tool?

1. **Memory management**: garbage collection
2. **Object oriented**: modularity and encapsualtion
3. **Egalitarianism**: first class everything
4. **Libraries**: great stdlib, powerful third-party facilities
5. **Introspection**: program available as date

What Is The Right Tool?

1. **Memory management**: garbage collection
2. **Object oriented**: modularity and encapsualtion
3. **Egalitarianism**: first class everything
4. **Libraries**: great stdlib, powerful third-party facilities
5. **Introspection**: program available as date

Lisp Is The Right Tool!

1. **Memory management**: bright gc, no pointers
2. **Object oriented**: *"With macros, closures, and run-time typing, Lisp transcends object-oriented programming. The generic function model is preferred to the message passing."*
3. **Egalitarianism**: packages, functions, closures, structures, arrays . . . everything is first-class!
4. **Libraries**: full library supplemented with asdf
5. **Introspection**: code as list as data

Lisp Is The Right Tool!

1. **Memory management**: bright gc, no pointers
2. **Object oriented**: *"With macros, closures, and run-time typing, Lisp transcends object-oriented programming. The generic function model is preferred to the message passing."*
3. **Egalitarianism**: packages, functions, closures, structures, arrays . . . everything is first-class!
4. **Libraries**: full library supplemented with asdf
5. **Introspection**: code as list as data

Lisp Is The Right Tool!

1. **Memory management**: bright gc, no pointers
2. **Object oriented**: *"With macros, closures, and run-time typing, Lisp transcends object-oriented programming. The generic function model is preferred to the message passing."*
3. **Egalitarianism**: packages, functions, closures, structures, arrays . . . everything is first-class!
4. **Libraries**: full library supplemented with asdf
5. **Introspection**: code as list as data

Lisp Is The Right Tool!

1. **Memory management**: bright gc, no pointers
2. **Object oriented**: *"With macros, closures, and run-time typing, Lisp transcends object-oriented programming. The generic function model is preferred to the message passing."*
3. **Egalitarianism**: packages, functions, closures, structures, arrays . . . everything is first-class!
4. **Libraries**: full library supplemented with asdf
5. **Introspection**: code as list as data

Lisp Is The Right Tool!

1. **Memory management**: bright gc, no pointers
2. **Object oriented**: *"With macros, closures, and run-time typing, Lisp transcends object-oriented programming. The generic function model is preferred to the message passing."*
3. **Egalitarianism**: packages, functions, closures, structures, arrays . . . everything is first-class!
4. **Libraries**: full library supplemented with asdf
5. **Introspection**: code as list as data

The Bottom Line

1. Lisp is a dynamic language as it grows to meet your needs
2. Lisp is a programmable proogramming langage
3. Lisp is for doing what you've been told is impossible.

The Bottom Line

1. Lisp is a dynamic language as it grows to meet your needs
2. Lisp is a programmable proogramming langage
3. Lisp is for doing what you’ve been told is impossible.

The Bottom Line

1. Lisp is a dynamic language as it grows to meet your needs
2. Lisp is a programmable proogramming langage
3. Lisp is for doing what you've been told is impossible.

The Greatest

> *"Lisp is the greatest single programming language ever designed."*
>
> — Alan Kay

Standard Syntax

Question: What is XML?

XML is standardized syntax used to express arbitrary hierarchical data.

To-do lists, web pages, medical records an dconfig files are all examples of XML use.

Standard Syntax

Question: What is XML?

XML is standardized syntax used to express arbitrary hierarchical data.

To-do lists, web pages, medical records an dconfig files are all examples of XML use.

Standard Syntax

Question: What is XML?

XML is standardized syntax used to express arbitrary hierarchical data.

To-do lists, web pages, medical records an dconfig files are all examples of XML use.

A To-do List

Let's use an example to-do list:

An example to-do list

```
<todo name="housework">
  <item priority="high">Clean the house.</item>
  <item priority="medium">Wash the dishes.</item>
  <item priority="medium">Buy more soap.</item>
</todo>
```

A To-do List

Let's use an example to-do list:

An example to-do list

```
<todo name="housework">
   <item priority="high">Clean the house.</item>
   <item priority="medium">Wash the dishes.</item>
   <item priority="medium">Buy more soap.</item>
</todo>
```

A To-do List

What happens if we submit this list to an XML parer? . . . Once the data is parsed, how is it represented in memory?

The most natural representation is as tree.

Anything that can be represented as a tree, can be represented in XML, and vice-versa.

A To-do List

What happens if we submit this list to an XML parer? . . . Once the data is parsed, how is it represented in memory?

The most natural representation is as tree.

Anything that can be represented as a tree, can be represented in XML, and vice-versa.

A To-do List

What happens if we submit this list to an XML parer? . . .
Once the data is parsed, how is it represented in memory?

The most natural representation is as tree.

Anything that can be represented as a tree, can be represented in XML, and vice-versa.

Code As A Tree

Question: What other type of data is often represented as a tree?

Any compiler inevitably parses the srouce code into an abstract syntax tree.

This shouldn´t surprise you: *source code is hierarchical.*

Code As A Tree

Question: What other type of data is often represented as a tree?

Any compiler inevitably parses the srouce code into an abstract syntax tree.

This shouldn´t surprise you: *source code is hierarchical.*

Code As A Tree

Question: What other type of data is often represented as a tree?

Any compiler inevitably parses the srouce code into an abstract syntax tree.

This shouldn´t surprise you: *source code is hierarchical.*

Code As XML

If all source code is a tree, and any tree can be represented as XML:

An example 'add' function

```
int add(int arg1, int arg2)
{
    return arg1 + arg2;
}
```

Code As XML

If all source code is a tree, and any tree can be represented as XML:

An example 'add' function

```
int add(int arg1, int arg2)
{
   return arg1 + arg2;
}
```

XML ’add’ function

Can you convert that function to an XML equivalent?

An example ’add’ function

```
<define-function return-type="int"name="add">
  <arguments>
    <argument type="int">arg1</argument>
    <argument type="int">arg2</argument>
  </arguments>
  <body>
    <return>
      <add value1="arg1"value2="arg2"/>
    </return>
  </body>
</define>
```

XML 'add' function

Can you convert that function to an XML equivalent?

An example 'add' function

```
<define-function return-type="int"name="add">
  <arguments>
    <argument type="int">arg1</argument>
    <argument type="int">arg2</argument>
  </arguments>
  <body>
    <return>
      <add value1="arg1"value2="arg2"/>
    </return>
  </body>
</define>
```

Code or Data

Classify our XML ’add’ function: is it *data*? *code*?

We could easily write a small interpreter for this XML code and we could execute it directly.

It’s data . . . and code.

Code or Data

Classify our XML ’add’ function: is it *data*? *code*?

We could easily write a small interpreter for this XML code and we could execute it directly.

It’s data . . . and code.

Code or Data

Classify our XML ‘add’ function: is it *data*? *code*?

We could easily write a small interpreter for this XML code and we could execute it directly.

It’s data . . . and code.

Code is Data

We’ve arrived at the following interesting point:

We now know that code is always data.

Code is Data

We’ve arrived at the following interesting point:

We now know that code is always data.

XML Is Flexible

Ant takes an XML file with specific build instructions and interprets them.

A simple XML instruction causes a Java class to be executed:

An Ant instruction

```
<copy todir="../new/dir">
  <fileset dir="src_dir"/>
</copy>
```

XML Is Flexible

Ant takes an XML file with specific build instructions and interprets them.

A simple XML instruction causes a Java class to be executed:

An Ant instruction

```
<copy todir="../new/dir">
  <fileset dir="src_dir"/>
</copy>
```

XML Is Flexible

Ant takes an XML file with specific build instructions and interprets them.

A simple XML instruction causes a Java class to be executed:

An Ant instruction

```
<copy todir="../new/dir">
  <fileset dir="src_dir"/>
</copy>
```

XML Is Powerful

That snippet copies a source directory to a destination directory.

Ant acts as a interpreter for a language that uses XML as its syntax.

Ant translates XML elements to appropriate Java instructions.

XML Is Powerful

That snippet copies a source directory to a destination directory.

Ant acts as a interpreter for a language that uses XML as its syntax.

Ant translates XML elements to appropriate Java instructions.

XML Is Powerful

That snippet copies a source directory to a destination directory.

Ant acts as a interpreter for a language that uses XML as its syntax.

Ant translates XML elements to appropriate Java instructions.

Why XML?

What is the **advantage** of using interpreted XML over simple Java code?

XML has the property of being flexible when introducing semantic constructs.

Why XML?

What is the **advantage** of using interpreted XML over simple Java code?

XML has the property of being flexible when introducing semantic constructs.

A Semantic Construct

Can we represent the 'copy' example above in Java?

An Ant instruction

```
CopyTask copy = new CopyTask();
Fileset fileset = new Fileset();

fileset.setDir("src_dir");
copy.setToDir("../new/dir");
copy.setFileset(fileset);

copy.execute();
```

A Semantic Construct

Can we represent the ‘copy’ example above in Java?

An Ant instruction

```
CopyTask copy = new CopyTask();
Fileset fileset = new Fileset();

fileset.setDir("src_dir");
copy.setToDir("../new/dir");
copy.setFileset(fileset);

copy.execute();
```

A Special Operator

That code was almost the same as the original XML.

Question: What's different?

Answer: the XML snippet introduces a special semantic construct for copying.

A Special Operator

That code was almost the same as the original XML.

Question: What's different?

Answer: the XML snippet introduces a special semantic construct for copying.

A Special Operator

That code was almost the same as the original XML.

Question: What's different?

Answer: the XML snippet introduces a special semantic construct for copying.

Hypothetical Java

If we could do it in Java, it would look like this:

This Java Isn't Feasible

```
copy("../new/dir")
{
    fileset("src_dir");
}
```

Hypothetical Java

If we could do it in Java, it would look like this:

This Java Isn’t Feasible

```
copy("../new/dir")
{
    fileset("src_dir");
}
```

About That Implementation

We could extend the Java language to introduce an operator for copying files.

We would do this by modifying the AST grammar that the Java compiler accepts.

We can't do this with standard Java ficilities, but we can do it in XML.

About That Implementation

We could extend the Java language to introduce an operator for copying files.

We would do this by modifying the AST grammar that the Java compiler accepts.

We can't do this with standard Java ficilities, but we can do it in XML.

About That Implementation

We could extend the Java language to introduce an operator for copying files.

We would do this by modifying the AST grammar that the Java compiler accepts.

We can't do this with standard Java ficilities, but we can do it in XML.

Extending Ant

Why not extend Ant, in Ant itself?

If Ant provided constructs to develop tasks in Ant itself we’d reach a higher level of abstraction.

Extending Ant

Why not extend Ant, in Ant itself?

If Ant provided constructs to develop tasks in Ant itself we'd reach a higher level of abstraction.

Ant Tasks

Consider the possibility:

This Java Isn´t Feasible

```
<task name="Test">
   <echo message="Hello World!"/>
</task>
<Test />
```

If we could write a “task” task in Java and make Ant able to extend itself using Ant-XML!

Ant Tasks

Consider the possibility:

This Java Isn′t Feasible

```
<task name="Test">
   <echo message="Hello World!"/>
</task>
<Test />
```

If we could write a “task” task in Java and make Ant able to extend itself using Ant-XML!

Ant Tasks

Consider the possibility:

This Java Isn't Feasible

```
<task name="Test">
  <echo message="Hello World!"/>
</task>
<Test />
```

If we could write a "task" task in Java and make Ant able to extend itself using Ant-XML!

Welcome To Lisp

Oh, by the way, **you’re looking at Lisp code**.

Why Not XML

Self-extending Ant wouldn't be useful.

The reason for this is XML's verbosity.

The solution to this problem involves using a less verbose alternative to XML.

Why Not XML

Self-extending Ant wouldn't be useful.

The reason for this is XML's verbosity.

The solution to this problem involves using a less verbose alternative to XML.

Why Not XML

Self-extending Ant wouldn’t be useful.

The reason for this is XML’s verbosity.

The solution to this problem involves using a less verbose alternative to XML.

Symbolic Expressions

We don’t have to use XML’s angle brackets to represented trees.

We could use other formats.

One such format is called a symbolic expression.

S-expressions accomplish the same goals as XML.

Symbolic Expressions

We don′t have to use XML′s angle brackets to represented trees.

We could use other formats.

One such format is called a symbolic expression.

S-expressions accomplish the same goals as XML.

Symbolic Expressions

We don’t have to use XML’s angle brackets to represented trees.

We could use other formats.

One such format is called a symbolic expression.

S-expressions accomplish the same goals as XML.

Symbolic Expressions

We don't have to use XML's angle brackets to represented trees.

We could use other formats.

One such format is called a symbolic expression.

S-expressions accomplish the same goals as XML.

Lisp ‘copy’

lisp implementation of ‘copy’

```
(copy
  (todir "../new/dir")
  (fileset (dir "src_dir")))
```

Lisp ‘copy’

lisp implementation of ‘copy’

```
(copy
   (todir "../new/dir")
   (fileset (dir "src_dir")))
```

Lisp Representation

What’s different with our Lisp representation?

- angled brackets seem to be replaced by parens
- dispense of unnecessary *‘(/element)’*

Lisp Representation

What’s different with our Lisp representation?

- angled brackets seem to be replaced by parens
- dispense of unnecessary *‘(/element)’*

A Lisp ‘task’

Let’s look at our ‘task’ code in something that looks like Lisp:

Lisp ‘task’

```
(task (name "Test")
    (echo (message "Hello World!")))

(Test)
```

A Lisp ‘task’

Let’s look at our ‘task’ code in something that looks like Lisp:

Lisp ‘task’

```
(task (name "Test")
   (echo (message "Hello World!")))

(Test)
```

Welcome To Lisp

S-expressions are called lists in Lisp lingo.

The Lisp code above is a tree, implemented via a Lisp list.

Welcome to Lisp, you’ll enjoy your stay.