STUDIOS
ThoughtWorks®

Jatha

Common Lisp in Java

Ola Bini
JRuby Core Developer
ThoughtWorks Studios

ola.bini@gmail.com
http://olabini.com/blog

STUDIOS

ThoughtWorks®

Common Lisp?

STUDIOS

ThoughtWorks®

Common Lisp?

ANSI standard

STUDIOS
ThoughtWorks®

Common Lisp?

ANSI standard

Powerful

STUDIOS

ThoughtWorks®

Common Lisp?

ANSI standard

Powerful

Multiparadigm

STUDIOS

ThoughtWorks®

Common Lisp?

ANSI standard

Powerful

Multiparadigm

- Procedural

STUDIOS

ThoughtWorks®

Common Lisp?

ANSI standard

Powerful

Multiparadigm

- Procedural
- Functional

STUDIOS

ThoughtWorks®

Common Lisp?

ANSI standard

Powerful

Multiparadigm

- Procedural
- Functional
- Object oriented

STUDIOS

ThoughtWorks®

Common Lisp?

ANSI standard

Powerful

Multiparadigm

- Procedural
- Functional
- Object oriented

Dynamic and lexical scope

STUDIOS

ThoughtWorks®

Common Lisp?

ANSI standard

Powerful

Multiparadigm

- Procedural
- Functional
- Object oriented

Dynamic and lexical scope

CLOS: multimethods, method combinations

STUDIOS ThoughtWorks®

Common Lisp?

ANSI standard

Powerful

Multiparadigm

- Procedural
- Functional
- Object oriented

Dynamic and lexical scope

CLOS: multimethods, method combinations

Macros and reader macros

STUDIOS

ThoughtWorks®

Jatha history

STUDIOS
ThoughtWorks®

Jatha history

1991: In C++

STUDIOS
ThoughtWorks®

Jatha history

1991: In C++

1996: Need for GC => Java port

STUDIOS
ThoughtWorks®

Jatha history

1991: In C++

1996: Need for GC => Java port

1997: Large subset of CL implemented

STUDIOS
ThoughtWorks®

Jatha history

1991: In C++

1996: Need for GC => Java port

1997: Large subset of CL implemented

2002: Was used to port the Algernon rules engine

STUDIOS
ThoughtWorks®

Jatha history

1991: In C++

1996: Need for GC => Java port

1997: Large subset of CL implemented

2002: Was used to port the Algernon rules engine

2003: Released as open source

STUDIOS
ThoughtWorks®

Jatha history

1991: In C++

1996: Need for GC => Java port

1997: Large subset of CL implemented

2002: Was used to port the Algernon rules engine

2003: Released as open source

2005: Support for macros

STUDIOS
ThoughtWorks®

Jatha history

1991: In C++

1996: Need for GC => Java port

1997: Large subset of CL implemented

2002: Was used to port the Algernon rules engine

2003: Released as open source

2005: Support for macros

STUDIOS

ThoughtWorks®

Jatha architecture

STUDIOS
ThoughtWorks®

Jatha architecture

Simple, handwritten parser

STUDIOS
ThoughtWorks®

Jatha architecture

Simple, handwritten parser

No reader macros

STUDIOS
ThoughtWorks®

Jatha architecture

Simple, handwritten parser

No reader macros

Extensible compiler

STUDIOS

ThoughtWorks®

Jatha architecture

Simple, handwritten parser

- No reader macros

Extensible compiler

- Every primitive is a class that executes itself

STUDIOS

ThoughtWorks®

Jatha architecture

- Simple, handwritten parser
  - No reader macros
- Extensible compiler
  - Every primitive is a class that executes itself
  - Compiler outputs Lisp values in SECD format

STUDIOS ThoughtWorks®

Jatha architecture

- Simple, handwritten parser
  - No reader macros
- Extensible compiler
  - Every primitive is a class that executes itself
  - Compiler outputs Lisp values in SECD format
- SECD Machine

STUDIOS
ThoughtWorks®

Jatha architecture

- Simple, handwritten parser
  - No reader macros
- Extensible compiler
  - Every primitive is a class that executes itself
  - Compiler outputs Lisp values in SECD format
- SECD Machine
  - With some extensions to handle CL weirdness

STUDIOS ThoughtWorks®

Jatha architecture

Simple, handwritten parser

- No reader macros

Extensible compiler

- Every primitive is a class that executes itself
- Compiler outputs Lisp values in SECD format

SECD Machine

- With some extensions to handle CL weirdness

Lisp values all implement gigantic interface with available methods

STUDIOS

ThoughtWorks®

SECD

STUDIOS

ThoughtWorks®

SECD

Functional programming language structure

STUDIOS
ThoughtWorks®

SECD

Functional programming language structure

Peter Landin

STUDIOS

ThoughtWorks®

SECD

Functional programming language structure

Peter Landin

Peter Henderson: "Functional Programming: Application and implementation"

STUDIOS

ThoughtWorks®

SECD

Functional programming language structure

Peter Landin

Peter Henderson: "Functional Programming: Application and implementation"

Peter Kogge: "The Architecture of Symbolic Machines"

STUDIOS

ThoughtWorks®

SECD

Functional programming language structure

Peter Landin

Peter Henderson: "Functional Programming: Application and implementation"

Peter Kogge: "The Architecture of Symbolic Machines"

Stack, Environment, Code, Dump

STUDIOS
ThoughtWorks®

SECD

Functional programming language structure

Peter Landin

Peter Henderson: "Functional Programming: Application and implementation"

Peter Kogge: "The Architecture of Symbolic Machines"

Stack, Environment, Code, Dump

S is the stack, not used for instruction parameters

STUDIOS
ThoughtWorks®

SECD

Functional programming language structure

Peter Landin

Peter Henderson: "Functional Programming: Application and implementation"

Peter Kogge: "The Architecture of Symbolic Machines"

Stack, Environment, Code, Dump

S is the stack, not used for instruction parameters

C is the instruction pointer

STUDIOS
ThoughtWorks®

SECD

Functional programming language structure

Peter Landin

Peter Henderson: "Functional Programming: Application and implementation"

Peter Kogge: "The Architecture of Symbolic Machines"

Stack, Environment, Code, Dump

S is the stack, not used for instruction parameters

C is the instruction pointer

E is a list of lists of the environment

STUDIOS
ThoughtWorks®

SECD

Functional programming language structure

Peter Landin

Peter Henderson: "Functional Programming: Application and implementation"

Peter Kogge: "The Architecture of Symbolic Machines"

Stack, Environment, Code, Dump

S is the stack, not used for instruction parameters

C is the instruction pointer

E is a list of lists of the environment

D is temporary storage for other registers, return stack

STUDIOS

ThoughtWorks®

SECD instructions

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack

LD: load a variable value on the stack

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack

LD: load a variable value on the stack

parameter to LD looks like this: (2 . 3), 2 is depth, 3 is ordinal

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack

LD: load a variable value on the stack

parameter to LD looks like this: (2 . 3), 2 is depth, 3 is ordinal

SEL: takes two list arguments, sets C to one of them based on S

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack

LD: load a variable value on the stack

parameter to LD looks like this: (2 . 3), 2 is depth, 3 is ordinal

SEL: takes two list arguments, sets C to one of them based on S

the next C is put in D until finished

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack

LD: load a variable value on the stack

parameter to LD looks like this: (2 . 3), 2 is depth, 3 is ordinal

SEL: takes two list arguments, sets C to one of them based on S

the next C is put in D until finished

JOIN: ends a SEL, returning a value from D to C

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack

LD: load a variable value on the stack

parameter to LD looks like this: (2 . 3), 2 is depth, 3 is ordinal

SEL: takes two list arguments, sets C to one of them based on S

the next C is put in D until finished

JOIN: ends a SEL, returning a value from D to C

LDF: takes a function argument and constructs a closure

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack

LD: load a variable value on the stack

parameter to LD looks like this: (2 . 3), 2 is depth, 3 is ordinal

SEL: takes two list arguments, sets C to one of them based on S

the next C is put in D until finished

JOIN: ends a SEL, returning a value from D to C

LDF: takes a function argument and constructs a closure

AP: pops a closure and parameters and applies it

STUDIOS
ThoughtWorks®

SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack

LD: load a variable value on the stack

parameter to LD looks like this: (2 . 3), 2 is depth, 3 is ordinal

SEL: takes two list arguments, sets C to one of them based on S

the next C is put in D until finished

JOIN: ends a SEL, returning a value from D to C

LDF: takes a function argument and constructs a closure

AP: pops a closure and parameters and applies it

will save S E C on D and replace them to run the code

STUDIOS

ThoughtWorks®

SECD instructions cntd

STUDIOS
ThoughtWorks®

SECD instructions cntd

RET: Pops a return value, restores S E C and push return value

STUDIOS
ThoughtWorks®

SECD instructions cntd

RET: Pops a return value, restores S E C and push return value

DUM: Pushes a dummy value on environment

STUDIOS
ThoughtWorks®

SECD instructions cntd

RET: Pops a return value, restores S E C and push return value

DUM: Pushes a dummy value on environment

RAP: Recursive apply, like AP

STUDIOS
ThoughtWorks®

SECD instructions cntd

RET: Pops a return value, restores S E C and push return value

DUM: Pushes a dummy value on environment

RAP: Recursive apply, like AP

Uses a dummy value to replace environment, for recursive call

STUDIOS

ThoughtWorks®

Jatha SECD extensions

STUDIOS
ThoughtWorks®

Jatha SECD extensions

B register: for dynamic bindings

STUDIOS
ThoughtWorks®

Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

STUDIOS
ThoughtWorks®

Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms

STUDIOS

ThoughtWorks®

Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms

LD_GLOBAL op: load a global value, handling dynamic bindings

STUDIOS
ThoughtWorks®

Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms

LD_GLOBAL op: load a global value, handling dynamic bindings

LDCF op: load function from a symbol slot instead of C

STUDIOS

ThoughtWorks®

Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms

LD_GLOBAL op: load a global value, handling dynamic bindings

LDCF op: load function from a symbol slot instead of C

LDR op: load &REST arguments

STUDIOS

ThoughtWorks®

Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms

LD_GLOBAL op: load a global value, handling dynamic bindings

LDCF op: load function from a symbol slot instead of C

LDR op: load &REST arguments

LIS op: handle optional arguments

STUDIOS
ThoughtWorks®

Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms

LD_GLOBAL op: load a global value, handling dynamic bindings

LDCF op: load function from a symbol slot instead of C

LDR op: load &REST arguments

LIS op: handle optional arguments

RTN_IF, RTN_IT ops: conditional return, used for AND, OR

STUDIOS
ThoughtWorks®

Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms

LD_GLOBAL op: load a global value, handling dynamic bindings

LDCF op: load function from a symbol slot instead of C

LDR op: load &REST arguments

LIS op: handle optional arguments

RTN_IF, RTN_IT ops: conditional return, used for AND, OR

SP_BIND, SP_UNBIND ops: binds/unbinds special variables

STUDIOS

ThoughtWorks®

Jatha SECD extensions cntd

STUDIOS
ThoughtWorks®

Jatha SECD extensions cntd

T op: pushes T on stack

STUDIOS
ThoughtWorks®

Jatha SECD extensions cntd

T op: pushes T on stack

TAG_B op: used to implement TAGBODY, pushes tag info to X

STUDIOS
ThoughtWorks®

Jatha SECD extensions cntd

T op: pushes T on stack

TAG_B op: used to implement TAGBODY, pushes tag info to X

TAG_E op: end of TAGBODY, pops X

STUDIOS
ThoughtWorks®

Jatha SECD extensions cntd

T op: pushes T on stack

TAG_B op: used to implement TAGBODY, pushes tag info to X

TAG_E op: end of TAGBODY, pops X

STUDIOS

ThoughtWorks®

Jatha parser

STUDIOS
ThoughtWorks®

Jatha parser

Lisp is easy to parse, right?

STUDIOS
ThoughtWorks®

Jatha parser

Lisp is easy to parse, right?

Well, not that easy...

STUDIOS
ThoughtWorks®

Jatha parser

Lisp is easy to parse, right?

Well, not that easy...

Case sensitivity

STUDIOS

ThoughtWorks®

Jatha parser

Lisp is easy to parse, right?

Well, not that easy...

Case sensitivity

Lists (and dotted pairs)
Strings (including escapes)
Characters
Symbols
Mixed case symbols (including escapes)
Quotes (single, back, splice)
Numbers
Packages
Keywords
Reader macros (not fully implemented, though)

STUDIOS

ThoughtWorks®

Jatha parser

Lisp is easy to parse, right?

Well, not that easy...

Case sensitivity

Lists (and dotted pairs)
Strings (including escapes)
Characters
Symbols
Mixed case symbols (including escapes)
Quotes (single, back, splice)
Numbers
Packages
Keywords
Reader macros (not fully implemented, though)

STUDIOS

ThoughtWorks®

Compiler

STUDIOS

ThoughtWorks®

Compiler

Lisp expression => Lisp expression of op codes

STUDIOS
ThoughtWorks®

Compiler

Lisp expression => Lisp expression of op codes

&REST
AND
DEFMACRO
DEFUN
IF
LAMBDA
LET
LETREC
MACRO
OR
PROGN
PRIMITIVE
QUOTE
SETQ

STUDIOS
ThoughtWorks®

Compiler

Lisp expression => Lisp expression of op codes

&REST
AND
DEFMACRO
DEFUN
IF
LAMBDA
LET
LETREC
MACRO
OR
PROGN
PRIMITIVE
QUOTE
SETQ

Indexing of variables

STUDIOS
ThoughtWorks®

Jatha SECD machine

STUDIOS
ThoughtWorks®

Jatha SECD machine

No symbolic representation

STUDIOS
ThoughtWorks®

Jatha SECD machine

No symbolic representation

Instances of SECDop pushed to registers

STUDIOS
ThoughtWorks®

Jatha SECD machine

No symbolic representation

Instances of SECDop pushed to registers

Primitives are just opcodes

STUDIOS
ThoughtWorks®

Jatha SECD machine

No symbolic representation

Instances of SECDop pushed to registers

Primitives are just opcodes

Since the “bytecode” is represented as Lisp

STUDIOS
ThoughtWorks®

Jatha SECD machine

No symbolic representation

Instances of SECDop pushed to registers

Primitives are just opcodes

Since the “bytecode” is represented as Lisp

And since primitives are Lisp values, it can just be executed the same

STUDIOS

ThoughtWorks®

Typical primitives

STUDIOS
ThoughtWorks®

Typical primitives

+
APPEND
APPLY
APROPOS
AREF
CAR
CDR
ATOM
`
CONS
FUNCALL
GO
SETF
USE-PACKAGE

STUDIOS

ThoughtWorks®

Implementation tidbits

STUDIOS
ThoughtWorks®

Implementation tidbits

It's very object oriented

STUDIOS

ThoughtWorks®

Implementation tidbits

It's very object oriented

No singletons

STUDIOS
ThoughtWorks®

Implementation tidbits

It's very object oriented

No singletons

Have been this way since the beginning

STUDIOS
ThoughtWorks®

Implementation tidbits

It's very object oriented

No singletons

- Have been this way since the beginning

Basically everything is represented using Lisp types

STUDIOS
ThoughtWorks®

Implementation tidbits

It's very object oriented

No singletons

Have been this way since the beginning

Basically everything is represented using Lisp types

Bytecode is a Lisp list of Lisp symbols and other values

STUDIOS
ThoughtWorks®

Implementation tidbits

It's very object oriented

No singletons

- Have been this way since the beginning

Basically everything is represented using Lisp types

- Bytecode is a Lisp list of Lisp symbols and other values
- Registers are Lisp lists

STUDIOS
ThoughtWorks®

Implementation tidbits

It's very object oriented

No singletons

- Have been this way since the beginning

Basically everything is represented using Lisp types

- Bytecode is a Lisp list of Lisp symbols and other values
- Registers are Lisp lists
- Primitives are Lisp values

STUDIOS

ThoughtWorks®

Implementation tidbits

It's very object oriented

No singletons

- Have been this way since the beginning

Basically everything is represented using Lisp types

- Bytecode is a Lisp list of Lisp symbols and other values
- Registers are Lisp lists
- Primitives are Lisp values
- etc.

STUDIOS

ThoughtWorks®

Implementation tidbits

It's very object oriented

No singletons

- Have been this way since the beginning

Basically everything is represented using Lisp types

- Bytecode is a Lisp list of Lisp symbols and other values
- Registers are Lisp lists
- Primitives are Lisp values
- etc.

STUDIOS

ThoughtWorks®

Implementation tidbits

It's very object oriented

No singletons

- Have been this way since the beginning

Basically everything is represented using Lisp types

- Bytecode is a Lisp list of Lisp symbols and other values
- Registers are Lisp lists
- Primitives are Lisp values
- etc.

STUDIOS

ThoughtWorks®

REPL

STUDIOS

ThoughtWorks®

REPL

Print current package in prompt

STUDIOS

ThoughtWorks®

REPL

Print current package in prompt

Read and parse input

STUDIOS

ThoughtWorks®

REPL

Print current package in prompt

Read and parse input

Compile input

STUDIOS

ThoughtWorks®

REPL

Print current package in prompt

Read and parse input

Compile input

Execute compiled code

STUDIOS

ThoughtWorks®

REPL

Print current package in prompt

Read and parse input

Compile input

Execute compiled code

Set *, ** and ***

STUDIOS

ThoughtWorks®

REPL

Print current package in prompt

Read and parse input

Compile input

Execute compiled code

Set *, ** and ***

Print result of execution

Demo
Jatha in action

STUDIOS

ThoughtWorks®

Major missing features

STUDIOS
ThoughtWorks®

Major missing features

Arrays

STUDIOS
ThoughtWorks®

Major missing features

Arrays

Complex numbers

STUDIOS
ThoughtWorks®

Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

STUDIOS
ThoughtWorks®

Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

STUDIOS
ThoughtWorks®

Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

Good ERROR and CERROR implementations

STUDIOS
ThoughtWorks®

Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

Good ERROR and CERROR implementations

CLOS

STUDIOS
ThoughtWorks®

Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

Good ERROR and CERROR implementations

CLOS

Pathnames

STUDIOS

ThoughtWorks®

Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

Good ERROR and CERROR implementations

CLOS

Pathnames

Read macros

STUDIOS
ThoughtWorks®

Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

Good ERROR and CERROR implementations

CLOS

Pathnames

Read macros

Conditions

STUDIOS
ThoughtWorks®

Future directions

STUDIOS
ThoughtWorks®

Future directions

Java integration

STUDIOS
ThoughtWorks®

Future directions

Java integration

Byte code compilation

STUDIOS
ThoughtWorks®

Future directions

Java integration

Byte code compilation

More CL functions implemented

STUDIOS
ThoughtWorks®

Future directions

Java integration

Byte code compilation

More CL functions implemented

LOOP macro

STUDIOS
ThoughtWorks®

Future directions

Java integration

Byte code compilation

More CL functions implemented

LOOP macro

Full SETF

STUDIOS
ThoughtWorks®

Future directions

Java integration

Byte code compilation

More CL functions implemented

LOOP macro

Full SETF

Look at something like UCW, let that drive implementation

STUDIOS

ThoughtWorks®

Complications

STUDIOS

ThoughtWorks®

Complications

Multiple values

STUDIOS
ThoughtWorks®

Complications

Multiple values

Numerical classes

STUDIOS
ThoughtWorks®

Complications

Multiple values

Numerical classes

SECD not well suited for compilation

STUDIOS
ThoughtWorks®

Complications

Multiple values

Numerical classes

SECD not well suited for compilation

Extremely polymorphic call sites

STUDIOS
ThoughtWorks®

Complications

Multiple values

Numerical classes

SECD not well suited for compilation

Extremely polymorphic call sites

Hard to implement new things since reflection isn't used

STUDIOS
ThoughtWorks®

Complications

Multiple values

Numerical classes

SECD not well suited for compilation

Extremely polymorphic call sites

Hard to implement new things since reflection isn't used

STUDIOS

ThoughtWorks®

Potential JVM solutions

STUDIOS
ThoughtWorks®

Potential JVM solutions

Cheap tuples

STUDIOS

ThoughtWorks®

Potential JVM solutions

Cheap tuples

Faster reflection

STUDIOS
ThoughtWorks®

Potential JVM solutions

Cheap tuples

Faster reflection

Numerical tower (fast, maybe based on gnu.math)

STUDIOS
ThoughtWorks®

Potential JVM solutions

Cheap tuples

Faster reflection

Numerical tower (fast, maybe based on gnu.math)

Interface invocation

STUDIOS

ThoughtWorks®

JVM languages

STUDIOS

ThoughtWorks®

JVM languages

Most aren't full implementations

STUDIOS

ThoughtWorks®

JVM languages

Most aren't full implementations

Usually based on simple interpreters

STUDIOS
ThoughtWorks®

JVM languages

Most aren't full implementations

Usually based on simple interpreters

Or byte code (not JVM) based machines

STUDIOS
ThoughtWorks®

JVM languages

Most aren't full implementations

Usually based on simple interpreters

- Or byte code (not JVM) based machines
- Or pure compilation

STUDIOS
ThoughtWorks®

JVM languages

Most aren't full implementations

Usually based on simple interpreters

- Or byte code (not JVM) based machines
- Or pure compilation

Many use reflection

STUDIOS
ThoughtWorks®

JVM languages

Most aren't full implementations

Usually based on simple interpreters

- Or byte code (not JVM) based machines
- Or pure compilation

Many use reflection

Many use loads of interfaces

STUDIOS
ThoughtWorks®

JVM languages

Most aren't full implementations

Usually based on simple interpreters

- Or byte code (not JVM) based machines
- Or pure compilation

Many use reflection

Many use loads of interfaces

Byte code based solution will not help most of these

STUDIOS
ThoughtWorks®

JVM languages

Most aren't full implementations

Usually based on simple interpreters

- Or byte code (not JVM) based machines
- Or pure compilation

Many use reflection

Many use loads of interfaces

Byte code based solution will not help most of these

Method handles have much larger impact

Q and A