Jatha
Common Lisp in Java

Ola Bini
JRuby Core Developer
ThoughtWorks Studios

ola.bini@gmail.com
http://olabini.com/blog
Common Lisp?
Common Lisp?

ANSI standard
Common Lisp?

ANSI standard

Powerful
Common Lisp?

ANSI standard

Powerful

Multiparadigm
Common Lisp?

ANSI standard

Powerful

Multiparadigm

Procedural
Common Lisp?

- ANSI standard
- Powerful
- Multiparadigm
  - Procedural
  - Functional
Common Lisp?

- ANSI standard
- Powerful
- Multiparadigm
  - Procedural
  - Functional
  - Object oriented
Common Lisp?

- ANSI standard
- Powerful
- Multiparadigm
  - Procedural
  - Functional
  - Object oriented
- Dynamic and lexical scope
Common Lisp?

- ANSI standard
- Powerful
- Multiparadigm
  - Procedural
  - Functional
  - Object oriented
- Dynamic and lexical scope
- CLOS: multimethods, method combinations
Common Lisp?

- ANSI standard
- Powerful
- Multiparadigm
  - Procedural
  - Functional
  - Object oriented
- Dynamic and lexical scope
- CLOS: multimethods, method combinations
- Macros and reader macros
Jatha history
Jatha history

1991: In C++
Jatha history

1991: In C++

1996: Need for GC => Java port
Jatha history

1991: In C++
1996: Need for GC => Java port
1997: Large subset of CL implemented
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
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
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
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
Jatha architecture
Jatha architecture

Simple, handwritten parser
Jatha architecture

Simple, handwritten parser

No reader macros
Jatha architecture

Simple, handwritten parser

No reader macros

Extensible compiler
Jatha architecture

Simple, handwritten parser
   No reader macros

Extensible compiler
   Every primitive is a class that executes itself
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
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
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
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
SECD

Functional programming language structure
SECD

Functional programming language structure

Peter Landin
SECD

Functional programming language structure

Peter Landin

Peter Henderson: “Functional Programming: Application and implementation”
SECD

Functional programming language structure

Peter Landin

Peter Henderson: “Functional Programming: Application and implementation”

Peter Kogge: “The Architecture of Symbolic Machines”
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
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
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
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
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
SECD instructions
SECD instructions

NIL: push a nil pointer on the stack
SECD instructions

NIL: push a nil pointer on the stack

LDC: load a constant argument on the stack
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
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
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
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
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
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
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
**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
SECD instructions cntd
SECD instructions cntd

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

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

DUM: Pushes a dummy value on environment
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
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
Jatha SECD extensions
Jatha SECD extensions

B register: for dynamic bindings
Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information
Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms
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
Jatha SECD extensions

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms

LDLOBAL op: load a global value, handling dynamic bindings

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

B register: for dynamic bindings

X register: like D, for dumping tag information

BLK op: handles non-local exit forms

LDGLOBAL 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
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
Jatha SECD extensions cntd
Jatha SECD extensions cntd

T op: pushes T on stack
Jatha SECD extensions cntd

T op: pushes T on stack

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

Lisp is easy to parse, right?
Jatha parser

Lisp is easy to parse, right?
Well, not that easy...
Jatha parser

Lisp is easy to parse, right?

Well, not that easy...

Case sensitivity
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)
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)
Compiler

Lisp expression => Lisp expression of op codes
Compiler

Lisp expression => Lisp expression of op codes

&REST
AND
DEFMACRO
DEFUN
IF
LAMBDA
LET
LETREC
MACRO
OR
PROGN
PRIMITIVE
QUOTE
SETO
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
Jatha SECD machine
Jatha SECD machine

No symbolic representation
Jatha SECD machine

No symbolic representation

Instances of SECDop pushed to registers
Jatha SECD machine

No symbolic representation

Instances of SECDop pushed to registers

Primitives are just opcodes
Jatha SECD machine

No symbolic representation

Instances of SECDop pushed to registers

Primitives are just opcodes

Since the “bytecode” is represented as Lisp
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
Typical primitives
Typical primitives

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

It’s very object oriented
Implementation tidbits

It's very object oriented

No singletons
Implementation tidbits

It’s very object oriented

No singletons

Have been this way since the beginning
Implementation tidbits

It's very object oriented

No singletons

Have been this way since the beginning

Basically everything is represented using Lisp types
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
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
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
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.
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.
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.
REPL
REPL

Print current package in prompt
REPL

Print current package in prompt

Read and parse input
REPL

Print current package in prompt

Read and parse input

Compile input
REPL

Print current package in prompt
Read and parse input
Compile input
Execute compiled code
REPL

Print current package in prompt

Read and parse input

Compile input

Execute compiled code

Set *, ** and ***
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
Major missing features
Major missing features

Arrays
Major missing features

Arrays

Complex numbers
Major missing features

Arrays

Complex numbers

&Optional and &Key arguments
Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)
Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

Good ERROR and CERROR implementations
Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

Good ERROR and CERROR implementations

CLOS
Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

Good ERROR and CERROR implementations

CLOS

Pathnames
Major missing features

Arrays
Complex numbers
&Optional and &Key arguments
Lambda functions (partial support)
Good ERROR and CERROR implementations
CLOS
Pathnames
Read macros
Major missing features

Arrays

Complex numbers

&Optional and &Key arguments

Lambda functions (partial support)

Good ERROR and CERROR implementations

CLOS

Pathnames

Read macros

Conditions
Future directions
Future directions

Java integration
Future directions

Java integration

Byte code compilation
Future directions

Java integration

Byte code compilation

More CL functions implemented
Future directions

Java integration

Byte code compilation

More CL functions implemented

LOOP macro
Future directions

Java integration
Byte code compilation
More CL functions implemented
LOOP macro
Full SETF
Future directions

Java integration

Byte code compilation

More CL functions implemented

LOOP macro

Full SETF

Look at something like UCW, let that drive implementation
Complications
Complications

Multiple values
Complications

Multiple values

Numerical classes
Complications

Multiple values

Numerical classes

SECD not well suited for compilation
Complications

- Multiple values
- Numerical classes
- SECD not well suited for compilation
- Extremely polymorphic call sites
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
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
Potential JVM solutions
Potential JVM solutions

Cheap tuples
Potential JVM solutions

Cheap tuples

Faster reflection
Potential JVM solutions

Cheap tuples
Faster reflection
Numerical tower (fast, maybe based on gnu.math)
Potential JVM solutions

Cheap tuples

Faster reflection

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

Interface invocation
JVM languages
JVM languages

Most aren’t full implementations
JVM languages

Most aren’t full implementations

Usually based on simple interpreters
JVM languages

Most aren’t full implementations

Usually based on simple interpreters

Or byte code (not JVM) based machines
JVM languages

Most aren’t full implementations

Usually based on simple interpreters
  Or byte code (not JVM) based machines
  Or pure compilation
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
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
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
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