By support from ALU, Franz and Lispworks

Lisp for the Twenty First Century

Mark Tarver

dr.mtarver@ukonline.ac.uk

Lambda Associates

The Structure of this Presentation

- Three parts
  - Social challenges to Lisp - why not more popular?
  - My own work with Qi as a partial solution
  - Remaining challenges

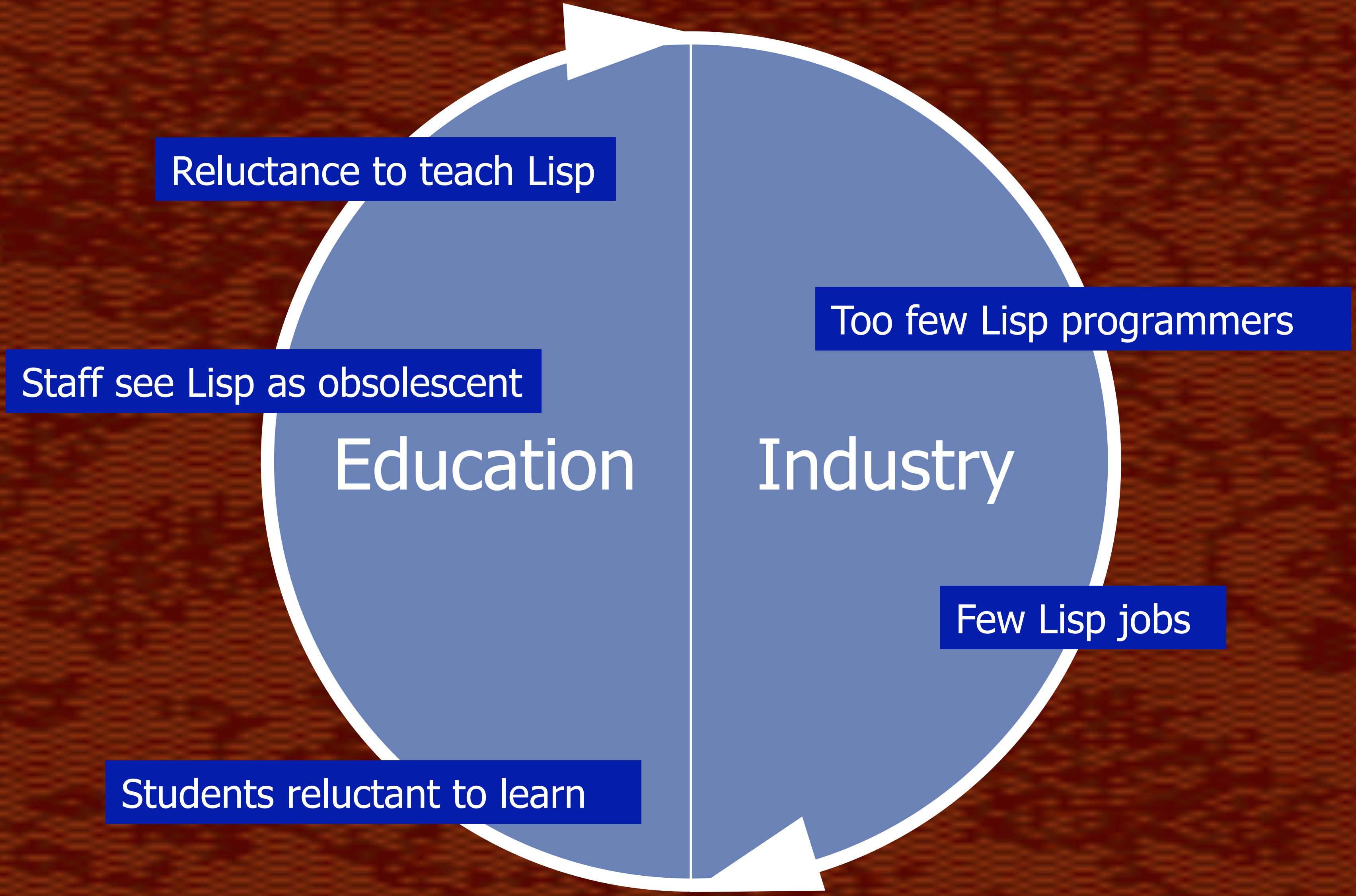
Lisp in the Job Market

- The Expertelligence experience
  - Wanted to employ Lisp & Prolog people
  - Ended with C++ because easier to find
- Students don't want to study FPLs – no job

Lisp in Education

- In 1990 CS dept. voted for FP as core course for year 1.
- Common Lisp was never considered.
  - lack of static typing
  - not λ calculus consistent (no currying, partial apps)
  - missing pattern matching
  - too procedural
- ML chosen – but that failed!

Lisp's Catch-22 Cycle



Building the Right Thing

- Cannot manufacture jobs. Break cycle at education end.
- Modernise Common Lisp by building the Right Thing.
- Extend the effective lifetime of Lisp by making the Right Thing future-proof for another generation.

Characteristics of the Right Thing

1. Lisp compatible.
2. FOSS
3. Compact
4. Simple to learn
5. Efficient.
6. Has the characteristics of a modern FPL.
7. But if possible in advance of ML and Haskell.
8. Computationally adequate.
9. Customisable.
10. Well documented, theoretically secure.

Programs should be shorter than CL
Clarity without obscurity

Qi: the Core Language

```
(0-) (define father
           "Winston Churchill" -> "Randolph Churchill")
father

(1-) (tc +)
true

(2+) (define member
        {A → [A] → boolean}
        _ [] -> false
        X [X | _] -> true
        X [_ | Y] -> (member X Y))
member : (A → ((list A) → boolean))

(2-) (map *)
#<CLOSURE :LAMBDA [X6] [map X5 X6]>
                              : ((list number) → (list number))

(3+) ((* 7) 8) : number
56
```

Simplicity ✓

Modern FPL ✓

Jon Harrop Challenge Problem

- Apply the following rewrite rules from the leaves up

```
rational n + rational m -> rational(n + m)
rational n * rational m -> rational(n * m)
symbol x -> symbol x
0+f -> f
f+0 -> f
0*f -> 0
f*0 -> 0
1*f -> f
f*1 -> f
a+(b+c) -> (a+b)+c
a*(b*c) -> (a*b)*c
```

10^7 iterations, 2.6GHz

simplify

[* x [+ [+ [* 12 0] [+ 23 8]] y]]

The Results: O’Caml, Lisp, Qi

🟩	SBCL 1.0
🟦	Qi 9.2 / SBCL 1.0
🟥	O’Caml 3.09.03

	Time	LOC
mark	3.6s	15
nathan	3.4s	39
andre	15s	23
pascal	8.2s	24
dan	5.1s	34
jon	2.0s	15

```
let rec ( +: ) f g = match f, g with
  | `Int n, `
  | `Int (Int
```

```
(defin
  [O
  A

(de
  +
  +
  +
  +
  *
  *
  *
  *
  *
  *
  C
```

Nathan's Cod

```
(defun simplify
    (if (atom xe
      xexpr
      (let ((op (f
            (z (sec
            (y (thi
        (let* ((f (
               (g (s
               (nf (
               (ng
          (tagbod
           START
             (if (ec
MULTIPLY))
           OPTIMI
             (wher
redundant-chec
           TEST-P
             (wher
checks g))
```

Qi Object Code

```
(DEFUN simplify (V148)
(BLOCK NIL (TAGBODY
  (IF (CONSP V148) (LET ((Cdr159 (CDR V148)))
    (IF (CONSP Cdr159) (LET ((Cdr158 (CDR Cdr159)))
       (IF (CONSP Cdr158) (IF (NULL (CDR Cdr158))
         (RETURN (s (CAR V148) (simplify (CAR Cdr159))
           (simplify (CAR Cdr158))))
            (GO tag154))
            (GO tag154)))
            (GO tag154))))
          tag154
(RETURN V148))))

(DEFUN s (V149 V150 V151)
    (BLOCK NIL
       (TAGBODY (IF (EQ '+ V149)
          (IF (AND (NUMBERP V150) (NUMBERP V151))
                (RETURN (THE NUMBER (+ V150 V151)))
            (IF (EQL 0 V150) (RETURN V151)
                (IF (EQL 0 V151) (RETURN V150)
                     (IF (CONSP V151)
                            (LET ((Cdr170 (CDR V151)))
```

Compact

Efficient

Typing

- Static strong typing not a free lunch
- Qi makes it optional.
- Not based on algebraic approaches of the second-generation FPLs
  - Allows natural Lisp style programming
  - More powerful and interesting type systems
  - Skinnable and extensible

Sequent Calculus

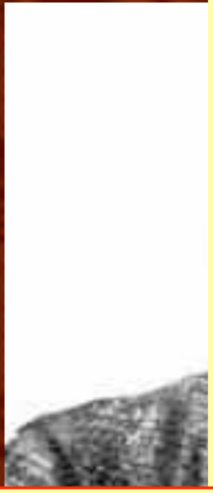
```
(1+) (datatype binary

          if (element? X [0 1])
          _______________
          X : zero-or-one;
```

ard Gentzen

notation

```
(2+) (define complement
          {binary --> binary}
          [0] -> [1]
```

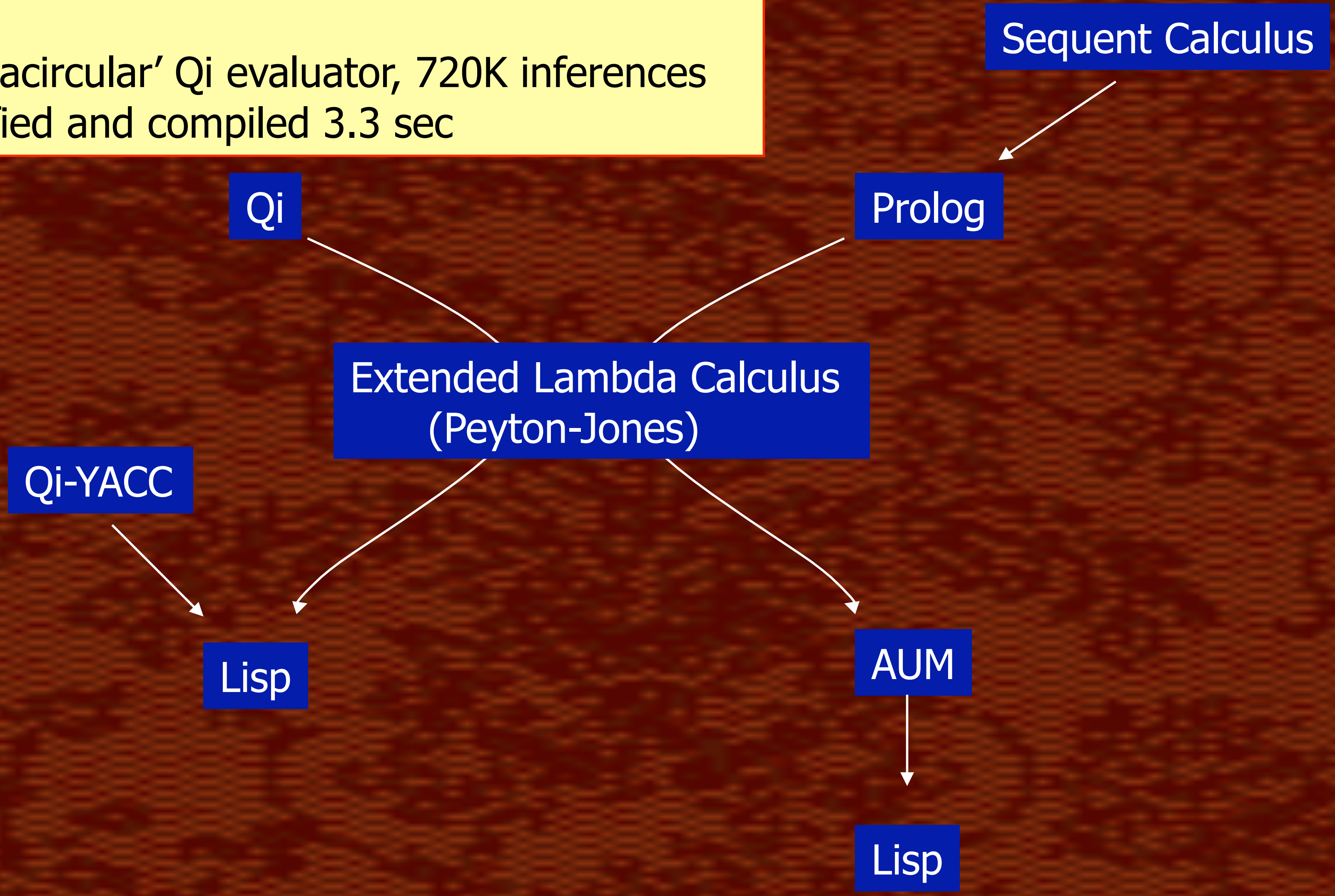
```
datatype zero_or_one = zero | one;
datatype empty = e;
datatype binary = place of zero_or_one * empty
                | bld of zero_or_one * binary;

fun complement (place(one, e)) = place(zero, e)
| complement (place(zero, e)) = place(one, e)
| complement (bld (one, Binary)) = bld(zero, complement Binary)
| complement (bld (zero, Binary)) = bld(one, complement Binary);
```



The Architecture of Qi

300 KLIPS, SBCL 1.0, 2,4 Ghz

‘metacircular’ Qi evaluator, 720K inferences verified and compiled 3.3 sec



Beyond ML/Haskell

Qi allows the construction of new generation of verified programs beyond ML/Haskell

How to Build an Algebra Program in Qi

www.lambdassociates.org/studies/study07.htm

Define The Syntax

```
(datatype expr

        ____________________________
        (number? X) : verified >> X : number;

        X : number;
         __________
         X : expr;

        if (not (element? X [- * / +]))
        X : symbol;
        __________
        X : expr;

        Op : (number --> number --> number);
        X : expr;
        Y : expr;
        ===========
        [X Op Y] : expr;)
```

Define the Operations

```
(define arith
  {expr --> expr}
   [X Op Y] -> (Op X Y)       where (and (number? X) (number? Y))
   X -> X)
```

Test:

```
(6+) (arith [9 - 8])
1 : expr
```

Syntactic and Semantic Validity

- Syntactic validity
  - f is syntactically valid if it has the type expr → expr
- Semantic validity
  - f is semantically valid if (f x) = x is math'lly provable for all x.
- Guarantee the integrity of our algebra tutor – all operations must syntactically and semantically valid.

Higher Order Functions & Semantic Validity

```
______________
arith : valid;
```

These operations are the bricks of the algebra system.

```
(define compose
  {[(A --> A)] --> (A --> A)}
  [F] -> F
  [F | Fs] -> (/. X ((compose Fs) (F X))))
```

A quick test in *Qi*:

```
 (8+) ((compose [sqrt sqrt]) 16)
2 : number
```

The composition of valid functions should produce a valid function. We add this as a sequent rule.

```
     Fs : [valid];
    ________________________
    (compose Fs) : valid;
```

Higher Order Functions & Semantic Validity

Here's another very useful higher-order function.

```
(define recurse
 {(expr --> expr) --> expr --> expr}
 F [X Op Y] -> (F [(recurse F X) Op (recurse F Y)])
 F X -> (F X))
```

```
F : valid;
____________________
(recurse F) : valid;
```

fix is a Qi system function; it generates a fixpoint for inputs F and X such that (F X) = X by repeatedly applying F until (F X) = X is true.

The operation of applying a valid operation until you cannot reduce the input further is itself valid.

```
F : valid;
____________
(fix F) : valid;
```

OK Expressions

We want to incorporate this stuff about valid algebraic operations into the type checker so that when we build our algebra system, the typechecker can not only tell us "Yes, that function outputs syntactically legal algebra" but also it can tell us "And also the heuristics you are using are semantically valid".

So syntax AND semantics are secured.

To do this we define a class of ok exprs. An ok expr is an expression that results from the application of a valid algebraic operation to an expr.

$$\frac{\begin{array}{l}\mathbf{F : valid;}\\ \mathbf{X : expr;}\end{array}}{\mathbf{(F\ X) : ok\text{-}expr;}} \qquad\qquad \frac{\mathbf{X : ok\text{-}expr;}}{\mathbf{X : expr;}}$$

Also ok-exprs are exprs themselves.

X : ok-expr;

Bringing It Together

```
(define simp
 {valid --> expr --> ok-expr}
   F E -> (fix (recurse F) E))

(10+) (simp arith [x + [12 * [5 - 3]]])
[x + 24] : ok-expr
```

What benefits do you get from doing it this way vs Lisp?

1. Pattern directed programming is easier to read and debug.
2. You get built-in syntactic guarantees; you can never write a program that outputs anything other than syntactically legal algebra.
3. You get semantic guarantees; *Qi* will also verify that your algebra program performs the transformations correctly.

Ergo your algebra program is clearer and more reliable.

```
(0-) (define expt
          X Y -> (EXPT X Y))

ex
Wa

(1-
ex

(2-
tru

(3-
12.
```

```
(0-) (define cases
          [cases] -> [error "case failure~%"]

cases

(1-) (su
cases

(2-) (to
true

(3+)  ((/. X (cases (= X 1) X
                    (= X 2) X
                     true  0)) 3)
0 : number
```

```
(0-) (eval [* 2 3])
6

(1-) (eval [define my-tail
               [cons X Y] -> Y])
my-tail

(2-) (my-tail [1 2 3])
[2 3])
```

1. Lisp compatible. ✓
2. FOSS ✓
3. Simple to learn ✓
4. Compact ✓
5. Efficient. ✓
6. Characteristics of a modern FPL. ✓
7. In advance of ML and Haskell. ✓
8. Computationally adequate. ✗
9. Customisable. ✓
10. Documented, theoretically secure. ✓

CL is not Computationally Adequate

- CLTL is not computationally adequate.
  - foreign language interface?
  - graphics?
  - working over the web?
- Python vs Lisp; analogy with C19 UK industry
- The gap made up in two ways
  - by vendors offering their own solutions
  - by FOSS people

Lisp is Being Overtaken

- People addicted to free software
  - Bloodshed C++, TCL/tk, Python
- Weakness of FOSS model in Lisp
  - lack of standards, multiple solutions
  - poorly documented
  - under resourced
  - unmaintained
  - low profile
- Shrink-wrapped solution - Python

Bringing Lisp Home

- Lisp lacks a home and a direction.
  - Haskell – Glasgow U.
  - ML – Toyota Chicago, LFCS
  - Python – Guido at Google
  - Lisp – MIT? Not any more.
- Challenges need to be met by a cooperative effort from the Lisp community.
- Next stage in L21 > October 2008.

Thanks