Appendices

Daniel Marian Zurawski

May 1, 2013

I verify that I am the sole author of the programs contained in this folder, except where explicitly stated to the contrary. Daniel Zurawski, 17 April 2013.

1 Appendix A – Compilation trace of Lispish naive primality testing to JavaScript

```
Emit Lispish:  (defn is_prime [num] (let [prime_over_two (fn [num factor]
(if (> factor (Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))))] (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3))))
Emit-list head:  defn , tail:  (is_prime [num] (let [prime_over_two
(fn [num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))]
(cond (< num 2) false (= 2 num) true (= 0 (mod num 2)) false :else
(prime_over_two num 3))))
Emit-forms, head:  defn , full expression:  (defn is_prime [num]
(let [prime_over_two (fn [num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))]
(cond (< num 2) false (= 2 num) true (= 0 (mod num 2)) false :else
(prime_over_two num 3))))
Emit-defn, name:  , arg:  num , arg tail:  nil , rest:  ((let [prime_over_two
(fn [num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))]
(cond (< num 2) false (= 2 num) true (= 0 (mod num 2)) false :else
(prime_over_two num 3))))
Emit Lispish:  ((let [prime_over_two (fn [num factor] (if (> factor
(Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))))] (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3))))
Emit Lispish:  (let [prime_over_two (fn [num factor]
(if (> factor (Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))))] (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3)))
Emit-list head:  let , tail:  ([prime_over_two (fn [num factor]
(if (> factor (Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))))] (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3)))
Emit-forms, head:  let , full expression:  (let [prime_over_two
(fn [num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))]
(cond (< num 2) false (= 2 num) true (= 0 (mod num 2)) false
:else (prime_over_two num 3)))
type:  let , let:  let , x:  prime_over_two , y:  (fn [num factor]
(if (> factor (Math.sqrt num)) true (if (= 0 (mod num factor)) false
(recur num (+ 2 factor))))) , body:  (cond (< num 2) false
(= 2 num) true (= 0 (mod num 2)) false :else (prime_over_two num 3))
Emit Lispish:  prime_over_two
Emit Lispish:  (cond (< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3))
Emit-list head:  cond , tail:  ((< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3))
Emit-forms, head:  cond , full expression:  (cond (< num 2) false
(= 2 num) true (= 0 (mod num 2)) false :else (prime_over_two num 3))
Emit-cond, head:  cond , name:  cond , rest:  ((< num 2) false (= 2 num) true
(= 0 (mod num 2)) false :else (prime_over_two num 3)) , reverse after
```

```
partitioning:  ((:else (prime_over_two num 3)) ((= 0 (mod num 2)) false)
((= 2 num) true) ((< num 2) false))
Emit Lispish:  (prime_over_two num 3)
Emit-list head:  prime_over_two , tail:  (num 3)
Emit-forms, head:  prime_over_two ,
full expression:  (prime_over_two num 3)
Emit-call, name:  prime_over_two , args:  num , rest:  (3)
Emit Lispish:  num
Emit Lispish:  3
Emit Lispish:  (= 0 (mod num 2))
Emit-list head:  = , tail:  (0 (mod num 2))
Emit-op, head:  = , tail:  (0 (mod num 2))
Emit Lispish:  0
Emit Lispish:  (mod num 2)
Emit-list head:  mod , tail:  (num 2)
Emit-op, head:  mod , tail:  (num 2)
Emit Lispish:  num
Emit Lispish:  2
Emit Lispish:  false
a:  ((0==(num%2))?false:prime_over_two(num, 3)) , b:  ((= 2 num) true)
Emit Lispish:  (= 2 num)
Emit-list head:  = , tail:  (2 num)
Emit-op, head:  = , tail:  (2 num)
Emit Lispish:  2
Emit Lispish:  num
Emit Lispish:  true
a:  ((2==num)?true:((0==(num%2))?false:prime_over_two(num, 3))) , b:
((< num 2) false)
Emit Lispish:  (< num 2)
Emit-list head:  < , tail:  (num 2)
Emit-op, head:  < , tail:  (num 2)
Emit Lispish:  num
Emit Lispish:  2
Emit Lispish:  false
Emit Lispish:  (fn [num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))
Emit-list head:  fn , tail:  ([num factor] (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))
Emit-forms, head:  fn , full expression:  (fn [num factor] (if (> factor
(Math.sqrt num)) true (if (= 0 (mod num factor)) false (recur num
(+ 2 factor)))))
Emit-defn, name:  , arg:  num , arg tail:  (factor) , rest:  ((if (> factor
(Math.sqrt num)) true (if (= 0 (mod num factor)) false (recur num
(+ 2 factor)))))
Emit Lispish:  ((if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor)))))
Emit Lispish:  (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor))))
Emit-list head:  if , tail:  ((> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor))))
Emit-forms, head:  if , full expression:  (if (> factor (Math.sqrt num)) true
(if (= 0 (mod num factor)) false (recur num (+ 2 factor))))
Emit-if, condition:  (> factor (Math.sqrt num)) , true-form:  true ,
false-form:  ((if (= 0 (mod num factor)) false (recur num (+ 2 factor))))
Emit Lispish:  (> factor (Math.sqrt num))
Emit-list head:  > , tail:  (factor (Math.sqrt num))
Emit-op, head:  > , tail:  (factor (Math.sqrt num))
Emit Lispish:  factor
Emit Lispish:  (Math.sqrt num)
Emit-list head:  Math.sqrt , tail:  (num)
Emit-forms, head:  Math.sqrt , full expression:  (Math.sqrt num)
Emit-call, name:  Math.sqrt , args:  num , rest:  nil
Emit Lispish:  num
Emit Lispish:  true
Emit Lispish:  ((if (= 0 (mod num factor)) false (recur num (+ 2 factor))))
Emit Lispish:  (if (= 0 (mod num factor)) false (recur num (+ 2 factor)))
Emit-list head:  if , tail:  ((= 0 (mod num factor)) false (recur num
(+ 2 factor)))
Emit-forms, head:  if , full expression:  (if (= 0 (mod num factor)) false
(recur num (+ 2 factor)))
Emit-if, condition:  (= 0 (mod num factor)) , true-form:  false , false-form:
((recur num (+ 2 factor)))
Emit Lispish:  (= 0 (mod num factor))
```

```
Emit-list head:  = , tail:  (0 (mod num factor))
Emit-op, head:  = , tail:  (0 (mod num factor))
Emit Lispish:  0
Emit Lispish:  (mod num factor)
Emit-list head:  mod , tail:  (num factor)
Emit-op, head:  mod , tail:  (num factor)
Emit Lispish:  num
Emit Lispish:  factor
Emit Lispish:  false
Emit Lispish:  ((recur num (+ 2 factor)))
Emit Lispish:  (recur num (+ 2 factor))
Emit-list head:  recur , tail:  (num (+ 2 factor))
Emit-forms, head:  recur , full expression:  (recur num (+ 2 factor))
Emit recur, head:  recur , expression:  (recur num (+ 2 factor))
Emit-call, name:  arguments.callee , args:  num , rest:  ((+ 2 factor))
Emit Lispish:  num
Emit Lispish:  (+ 2 factor)
Emit-list head:  + , tail:  (2 factor)
Emit-op, head:  + , tail:  (2 factor)
Emit Lispish:  2
Emit Lispish:  factor

function is_prime(num) {return (function(prime_over_two) { return ((num<2)
?false:((2==num)?true:((0==(num%2))?false:prime_over_two(num, 3)))) })(
function (num, factor) {return ((factor>Math.sqrt((num))) ? (true):(((0==(
num%factor)) ? (false):(arguments.callee(num, (2+factor))))))})}
```

2 Appendix B – Source Code Listings

2.1 project.clj – Dependencies and project definition

```
(defproject ClojureToJavaScript "1.0.0-SNAPSHOT"
  :description "FIXME: write description"
  :dependencies [[org.clojure/clojure "1.3.0"]
                 [org.clojure/tools.trace "0.7.3"]
                 [org.clojure/tools.cli "0.2.1"]]
  :main lispish.core)
```

2.2 core.clj – Lispish Compilator

```
(ns #^{:author "Daniel Zurawski"
       :doc "A simple Lisp to JavaScript transcompiler written in Clojure."}
  lispish.core
  [:require
   [clojure.string :as str]]
  [:use
   [clojure.walk]
   [clojure.tools.trace]
   [clojure.tools.cli :only (cli)]]
  (:gen-class :main true))


(def op (set [’mod ’+ ’- ’* ’/ ’> ’< ’=]))
(def bop (set [’or ’and ’not]))
(def forms (set [’recur ’let ’if ’fn ’defn ’cond]))

;; Clojure is a single pass compiler, thus we have to use forward declaration
;; if we need to use a function before it’s declared
(declare emit-list)

(defn emit [expressions]
  "Take an s-expression and emit its corresponding JavaScript form"
  (do
    (println "Top level - Emit Lispish: " expressions)
    (cond
      (nil? expressions) "null"
      (symbol? expressions) (str expressions)
      (seq? expressions) (emit-list expressions)
      (integer? expressions) (str expressions)
      (float? expressions) (str expressions)
      (string? expressions) (str \" expressions \")
      :else (str expressions))))

;; Abstract Structural Binding - + falls in type, + in op and 2 2 in tail
(defn emit-op [type [op & tail]]
  "Emit s-expression with single operators and two arguments"
  (do (println "Emit-op, head: " op ", tail: " tail)
      ;; Interlace the arguments with the operator
      (if (= op ’not)
        (str "(!" (emit tail) ")")
        (str "(" (clojure.string/join
                (str (cond (= op ’=) "=="
                           (= op ’mod) "%"
                           (= op ’or) "||"
                           (= op ’and) "&&"
                           :else op))
                (map emit tail))
           ")"))))

(defn emit-let [type [let [x y] body]]
  (println "Emit-let, x: " x ", y: " y ", body: " body)
        (str "(function(" (emit x) ") { return "  (emit body)  " })(" (emit y) ")" ))

(defn emit-if [type [if condition true-form & false-form]]
  (println "Emit-if, condition: " condition ", true-form: " true-form ", false-form: " false-form)
  (str "("
       (emit condition)
       " ? ("
       (emit true-form)
       "):("
       (emit false-form)
       "))"))

(defn emit-defn [type [defn name [arg & more] & rest]]
  (do
    (println "Emit-defn, name: " ", arg: " arg ", arg tail: " more ", rest: " rest))
  (str (str "function " (if (= "˜" name) "" name) "("
            (if (nil? more) arg (str arg ", " (clojure.string/join ", " more))) ") {return "
       (emit rest)
       "}")))

(defn emit-fn [head expression]
  (emit-defn head (concat (take 1 expression) ’("˜") (drop 1 expression))))

(defn emit-call [head [name args & rest]]
  (println "Emit-call, name: " name ", args: " args ", rest: " rest)
  (str name
       "("
       (if (nil? rest)
         (str "(" (emit args) ")")
         (str (str (emit args)) ", " (clojure.string/join ", " (map emit rest))))
       ")"))

(defn emit-recur [head expression]
  (println "Emit recur, head: " head ", expression: " expression)
  (emit-call head (concat ’("arguments.callee") (drop 1 expression))))

(defn emit-cond [head [name & rest]]
  (let [rev (reverse (partition 2 rest))]
    (println "Emit-cond, head: " head ", name: " name ", rest: " rest ", reverse after partitioning: " rev)
    (reduce
          (fn [a b] (do (println "a: " a ", b: " b ) (str "(" (emit (first b)) "?" (emit (second b)) ":" a ")")) )
          (str (emit (second (first rev))))
          (drop 1 rev))))

(defn emit-forms [head expression]
  (do (println "Emit-forms, head: " head ", full expression: " expression)
      (cond (= head ’let) (emit-let head expression)
            (= head ’if) (emit-if head expression)
            (= head ’fn) (emit-fn head expression)
            (= head ’defn) (emit-defn head expression)
            (= head ’cond) (emit-cond head expression)
            (= head ’recur) (emit-recur head expression)
            :else (emit-call head expression) )))
```

```
(defn emit-list [expressions]
  (do
      (if (symbol? (first expressions))

        (let [head (symbol (first expressions))
              expressions (conj
                            (rest expressions) head)]

          (println "Emit-list head: " head
                   ", tail: " (rest expressions))
          (cond
            (or (contains? op head) (contains? bop head)) (emit-op head expressions)
            (contains? forms head) (emit-forms head expressions)

            :else (emit-forms head expressions)
            ))
        ;; Not safe, may run into stack overflow if this will be a list or not-recognized
        (emit (first expressions)))))

(defn lisp-to-js [forms]
  (let [code (read-string forms)]
    (println code)
    (emit code)))

(defn read-file-emit [st file-out]
    (let [form (read st false "")]
      (if (not (= form ""))
        (do
          (spit file-out (str (emit form) "\n")  :append true)
          (read-file-emit st file-out)))))

(defn read-file [file-in file-out]
  (with-open [r (java.io.PushbackReader.
                 (clojure.java.io/reader file-in))]
    (binding [*read-eval* false]
      (spit file-out "" :append false)
      (read-file-emit r file-out))))

(defn run
  "Print out the options and the arguments"
  [opts args]
  (cond
    (:input opts) (if (:output opts)
                    (read-file (:input opts) (:output opts))
                    (println "Please provide --output or -out, path where the output JavaScript file will be generated."))
    (seq args) (println (lisp-to-js (first args)))
    :else (println "No path to input source code specified and no code given as argument.")))

(defn -main [& args]
  (let [[opts args banner]
        (cli args
             ["-h" "--help" "Show help" :flag true :default false]
             ["-in" "--input" "REQUIRED: Path to Lispish source code."]
             ["-out" "--output" "OPTIONAL: Path to JavaScript output file."]
             )]
    (when (:help opts)
      (println banner)
      (System/exit 0))
    (if (or (:input opts) (seq args))
      (run opts args)
      (println banner))))
```

2.3 core.clj – Test coverage of the naive Clojure recursive-descent-parser implementation

```
(ns lispish.test.core
  (:use [lispish.core])
  (:use [clojure.test]))

(deftest plus
  (is (= "(2+2)" (lisp-to-js "(+ 2 2)"))))
```

```
(deftest minus
  (is (= "(2-2)" (lisp-to-js "(- 2 2)"))))

(deftest multiply
  (is (= "(2*2)" (lisp-to-js "(* 2 2)"))))

(deftest divide
  (is (= "(2/2)" (lisp-to-js "(/ 2 2)"))))

(deftest logical-or
  (is (= "((5>10)||(10>5))" (lisp-to-js "(or (> 5 10) (> 10 5))"))))

(deftest logical-and
  (is (= "((5>10)&&(10>5))" (lisp-to-js "(and (> 5 10) (> 10 5))"))))

(deftest logical-and
  (is (= "(!(5>10))" (lisp-to-js "(not (> 5 10))"))))

(deftest if-form
  (is (= "((5>10) ? (true):(false))" (lisp-to-js "(if (> 5 10) true false)"))))

(deftest fn-form
  (is (= "function (x) {return (x*x)}" (lisp-to-js "(fn [x] (* x x))"))))

(deftest let-form
  (is (= "(function(x) { return (x*x) })(2)" (lisp-to-js "(let [x 2] (* x x))")
    )))

(deftest let-lambda-function
  (is (= "(function(times_five) { return times_five((5)) })(function (x)
    {return (x*5)})" (lisp-to-js "(let [times_five (fn [x] (* x 5))] (
    times_five 5))"))))

(deftest defn-form
  (is (= "function square(x) {return (x*x)}" (lisp-to-js "(defn square [x] (*
    x x))"))))

(deftest fibonacci-example
  (is (= "function fib(n) {return ((n<2) ? (1):((fib(((n-1)))+fib(((n-2))))))}"
         (lisp-to-js "(defn fib [n] (if (< n 2) 1 (+ (fib (- n 1)) (fib (- n 2)
          ))))" ))))

(deftest factorial-example
  (is (= "function factorial(n) {return ((n<2) ? (1):((n*factorial(((n-1))))))}
    "
         (lisp-to-js "(defn factorial [n] (if (< n 2) 1 (* n (factorial (- n 1)
          ))))"))))

(deftest ackermann-function
  (is (= "function ackermann(m, n) {return ((m==0)?(n+1):((n==0)?ackermann((m-1
    ), 1):ackermann((m-1), ackermann(m, (n-1)))))}"
         (lisp-to-js "(defn ackermann [m n]
                       (cond (= m 0) (+ n 1)
                             (= n 0) (ackermann (- m 1) 1)
                             :else (ackermann (- m 1) (ackermann m (- n 1)))))"))))

(deftest primality-checking-program
  (is (= "function is_prime(num) {return (function(prime_over_two) { return ((
    num<2)?false:((2==num)?true:((0==(num%2))?false:prime_over_two(num, 3)))) }
  )(function (num, factor) {return ((factor>Math.sqrt((num))) ? (true):(((0==(
    num%factor)) ? (false):(arguments.callee(num, (2+factor))))))})}"
         (lisp-to-js "(defn is_prime [num]
        (let [prime_over_two
                (fn [num factor]
                        (if (> factor (Math.sqrt num))
                                true
                                (if (= 0 (mod num factor))
                                        false
                                        (recur num (+ 2 factor)))))]
        (cond
                (< num 2) false
                (= 2 num) true
                (= 0 (mod num 2)) false
                :else (prime_over_two num 3))))")))) 
```