Accurate Programming
Third Halmstad Summer School on Testing
June 3-5, 2013

Testing and Verification in ACL2

Rex Page, University of Oklahoma

14:00 – 15:30

Support provided by HALMSTAD UNIVERSITY, FULBRIGHT, National Science Foundation

Goal

software that meets expectations

Specifying expectations

input output

- ✓ suite of test cases
  input, expected output (operation x^2): $2^2 = 4$, $3^2 = 9$
- ✓ corner cases
  carefully chosen input, expected output: $(-1)^2 = 1$
- ✓ operator relationships
  Boolean formulas: $x^2 \geq 0$, $(x + y)^2 = x^2 + 2xy + y^2$, …

Tests are theorems

- ✓ test cases: theorems with one-element domains
- ✓ operator relationships: more general theorems
- ✓ type checking: theorems proved by compilers

Theorems are derived from axioms

axioms

$x + 0 = x$	{+ identity}
$(-x) + x = 0$	{+ complement}
$x \times 1 = x$	{× identity}
$x \times 0 = 0$	{× null}
$x + y = y + x$	{+ commutative}
$x + (y + z) = (x + y) + z$	{+ associative}
$x \times (y + z) = (x \times y) + (x \times z)$	{distributive law}

x, y, z stand for any formula

Software verification

- ✓ axioms = programs
- ✓ theorems = properties of programs

Software framework

- ✓ programs = systems of equations
- ✓ constrains programs, but not computations

theorem: $(-1)\times(-1) = 1$

derivation

$(-1) \times (-1)$	
$= ((-1) \times (-1)) + 0$	{+ id}
$= ((-1) \times (-1)) + ((-1) + 1)$	{+ comp}
$= (((-1)\times(-1)) + (-1)) + 1$	{+ assoc}
$= (((-1)\times(-1)) + (-1)\times 1) + 1$	{× id}
$= ((-1)\times((-1) + 1)) + 1$	{dist law}
$= ((-1)\times 0) + 1$	{+ comp}
$= 0 + 1$	{× null}
$= 1 + 0$	{+ comm}
$= 1$	{+ id}

Some operations on lists (informal specs)

(cons x [x_1 x_2 … x_n]) = [x x_1 x_2 … x_n]
(first [x_1 x_2 … x_{n+1}]) = x_1
(rest [x_1 x_2 … x_{n+1}]) = [x_2 … x_{n+1}]
(append [x_1 x_2 … x_n] [y_1 y_2 … y_m]) = [x_1 x_2 … x_n y_1 y_2 … y_m]
(len [x_1 x_2 … x_n]) = n

Axioms (equations of computation … "programs")

(first (cons x xs)) = x	{first}
(rest (cons x xs)) = xs	{rest}
(append nil ys) = ys	{app0}
(append (cons x xs) ys) = (cons x (append xs ys))	{app1}
(len nil) = 0	{len0}
(len (cons x xs)) = 1 + (len xs)	{len1}

Theorems (properties derivable from axioms)

(append xs (append ys zs)) = (append (append xs ys) zs)	{app-assoc}
(len (append xs ys)) = (len xs) + (len ys)	{app-len}

Tests are all you need

Axioms, programs, properties … all can be formulated as tests

algebra of software

Properties (tests) that act as definitions

Two properties of concatenation

(append nil ys) = ys	{app0}
(append (cons x xs) ys) = (cons x (append xs ys))	{app1}

These equations have the following attributes

- ✓ comprehensive — all cases covered by left-hand-side formulas
- ✓ consistent — no two equations specify conflicting results
- ✓ computational

Properties (tests) that act as definitions

Two properties of concatenation

(append nil ys) = ys	{app0}
(append (cons x xs) ys) = (cons x (append xs ys))	{app1}

These equations have the following attributes

- ✓ comprehensive — all cases covered
- ✓ consistent — no two equations specify conflicting results
- ✓ computational
  circular reference on right-hand-side of equation closer to non-circular case than reference on left-hand-side

Properties (tests) that act as definitions

Two properties of concatenation

(append nil ys) = ys {app0}
(append (cons x xs) ys) = (cons x (append xs ys)) {app1}

These equations have the following attributes

- ✓ comprehensive — all cases covered
- ✓ consistent — no two equations specify conflicting results
- ✓ computational
  circular reference on right-hand-side of equation closer to non-circular case than reference on left-hand-side

the 3 c's

Properties (tests) that act as definitions

Two properties of concatenation

(append nil ys) = ys {app0}
(append (cons x xs) ys) = (cons x (append xs ys)) {app1}

These equations have the following attributes

- ✓ comprehensive — all cases covered
- ✓ consistent — no two equations specify conflicting results
- ✓ computational
  circular reference on right-hand-side of equation closer to non-circular case than reference on left-hand-side

the 3 c's

Equations satisfying "the 3 c's" define an operator

- ✓ all properties of the operation derive from the equations
- ✓ including computational properties

Formal definition of concatenation (ACL2 syntax)

```
(defun append (xs ys)
  (if (consp xs) ; cons predicate: Is xs a cons?
      (cons (first xs) (append (rest xs) ys))  ; {app1}
      ys                                       ; {app0}
```


What is ACL2 ?

ACL2 — A Computational Logic for Applicative Common Lisp

- ✓ programming language and mechanized logic
  dialect of Common Lisp
  conservative logic for effective mechanization/automation
  1st-order logic, terminating functions, no mutable variables
- ✓ Boyer/Moore theorem-prover, latest edition
  40 years in the making
  Many commerical applications (AMD, Rockwell-Collins, NSA …)

Proof Pad — IDE for ACL2 (Eggensperger)

- ✓ programming + theorem proving = full ACL2
- ✓ + property-based testing
  … but no narrowing, no user-defined data generators
- ✓ other ACL2 IDEs
  Emacs (comes with ACL2 installation: Moore, Kaufmann)
  DrACuLA (DrRacket plug-in: PLT, Felleisen, Eastlund)
  ACL2 Sedan (Eclipse plug-in: Manolios, Dillinger, Vroon …)

Formal specifications of properties (Proof Pad)

Some expected properties (informal specs)

(append xs (append ys zs)) = (append (append xs ys) zs) {app-assoc}
(len (append xs ys)) = (len xs) + (len ys) {app-len}

Formal specifications of properties (Proof Pad)

(append xs (append ys zs)) = (append (append xs ys) zs) {app-assoc}
(len (append xs ys)) = (len xs) + (len ys) {app-len}

informal

formal property

```
(defproperty app-assoc-tst ; {app-assoc}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer))
   zs :value (random-list-of (random-integer)))
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))
```


Formal specifications of properties (Proof Pad)

(append xs (append ys zs)) = (append (append xs ys) zs) {app-assoc}
(len (append xs ys)) = (len xs) + (len ys) {app-len}

informal

formal properties (Proof Pad)

```
(defproperty app-assoc-tst ; {app-assoc}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer))
   zs :value (random-list-of (random-integer)))
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))

(defproperty append-preserves-len-tst; {app-len}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (= (len (append xs ys)) (+ (len xs) (len ys))))
```

Formal specifications of properties (Proof Pad)

(append xs (append ys zs)) = (append (append xs ys) zs) {app-assoc}
(len (append xs ys)) = (len xs) + (len ys) {app-len}

informal

formal properties (Proof Pad)

```
(defproperty app-assoc-tst ; {app-assoc}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer))
   zs :value (random-list-of (random-integer)))
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))

(defproperty append-preserves-len-tst; {app-len}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (= (len (append xs ys)) (+ (len xs) (len ys))))
```

demo 1

Formal specifications of theorems (ACL2)

(append xs (append ys zs)) = (append (append xs ys) zs) {app-assoc}
(len (append xs ys)) = (len xs) + (len ys) {app-len}

informal

properties (Proof Pad)

```
(defproperty app-assoc-tst ; {app-assoc}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer))
   zs :value (random-list-of (random-integer)))
  (equal (append xs (append ys zs))
         (append (append xs ys) zs)))
(defthm app-assoc-thm ; {app-assoc}
   (equal (append xs (append ys zs))
          (append (append xs ys) zs)))
(defproperty append-preserves-len-tst; {app-len}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (= (len (append xs ys)) (+ (len xs) (len ys))))
(defthm append-preserves-len-thm; {app-len}
  (= (len (append xs ys)) (+ (len xs) (len ys))))
```

theorems (ACL2)

Additional properties of concatenation

```
(defproperty app-suffix-tst ; {app-sfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (nthcdr (len xs) (append xs ys))
         ys))
(defproperty app-prefix-tst ; {app-pfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (prefix (len xs) (append xs ys))
         xs))
```

Axioms defining prefix operator

(prefix 0 xs) = nil {*pfx0 a*}
(prefix n nil) = nil {*pfx0 b*}
(prefix (+ n 1) (cons x xs)) = (cons x (prefix n xs)) {*pfx1*}

Additional properties of concatenation

```
(defproperty app-suffix-tst ; {app-sfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (nthcdr (len xs) (append xs ys))
         ys))
(defproperty app-prefix-tst ; {app-pfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (prefix (len xs) (append xs ys))
         xs))
```

demo 2

Axioms defining prefix operator

(prefix 0 xs) = nil {*pfx0 a*}
(prefix n nil) = nil {*pfx0 b*}
(prefix (+ n 1) (cons x xs)) = (cons x (prefix n xs)) {*pfx1*}

Additional properties of concatenation

```
; import some theorems of algebra
(include-book "arithmetic-3/top" :dir :system)
(defproperty app-suffix-tst ; {app-sfx}
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (nthcdr (len xs) (append xs ys))
         ys))
(defproperty app-prefix-tst ; {app-pfx}
  (xs :value (random-list-of (random-integer))
      :where (true-listp xs) ; xs must be list
   ys :value (random-list-of (random-integer)))
  (equal (prefix (len xs) (append xs ys))
         xs))
(defthm append-prefix-thm ; {app-pfx}
  (implies (true-listp xs) ; xs must be a list
           (equal (prefix (len xs) (append xs ys))
                  xs)))
```

theorem becomes implication with :where-condition as hypothesis

Mechanization Is Necessary

without it, all is lost in the details

Even simple properties lead to big proofs

- ✓ millions of details in proofs about big programs
  - People can't keep track of millions of details
  - Besides, a proof at least is as likely to be wrong as a program
- ✓ people formulate properties … computers push details
  - proof organized into lemmas — similar to software components
    - rigorous, but not fully formal
    - like a paper-and-pencil proof, as done by mathematicians
  - some lemma architectures are better than others
    - like modular decomposition of software … design matters
  - formulation of properties is a big task
    - experience/judgment required … as in software development

Same strategy for testing

- ✓ formulate, then automate

Things that have been done with ACL2

AMD Athlon (K7) floating point processor (1999)

- ✓ AMD floating-point division circuit verified
  after 1994/97 Pentium floating-point division bugs
- ✓ Property verified

```
(implies (and (floating-point-numberp p 15 64)
              (floating-point-numberp d 15 64)
              (not (equal d 0))
              (rounding-modep mode))
         (equal  (divide p d mode)
                 (round (/ p d) mode)))...
```

Other commercial applications

- Motorola DSP microcode
- Rockwell Collins AAMP7 avionics software … ongoing
- CenTaur VLSI chip design … ongoing
- Numerous NSA applications … ongoing

High-assurance software and circuits

Test suites versus proofs

Testing

- ✓ AMD floating-point
  - AMD test suite of 80-million cases
  - gazillions of cases to consider ($2^{15+64} \times 2^{15+64} = 2^{158}$)
  - physically impossible to do that many tests

Proofs cover all cases

Extra investment for proof is substantial

- ✓ justified in some applications, but not all
- ✓ property-based testing justified in most applications

Practice exercises *(20 minutes)*

Ex 1: insert element at the end of a list

Suppose snoc inserts an element at the end of a list

(snoc x [x_1 x_2 … x_n]) = [x_1 x_2 … x_n x]

a) Complete the following equations

(snoc x nil) = ?? {snoc0}

(snoc x [x_1 x_2 … x_{n+1}]) = ?? {snoc1} (use circular reference)

b) Define snoc formally (that is, in ACL2)

Ex 2: reverse a list

Use snoc to define a list reversal operator in ACL2

(rev [x_1 x_2 … x_{n-1} x_n]) = [x_n x_{n-1} … x_2 x_1]

Ex 3: properties of reverse

a) Define/test a property of the formula (rev (rev xs))
b) Define/test a property of (rev (append xs ys))
c) Use ACL2 to verify (prove) those properties

Download Proof Pad: http://proofpad.org

Multiplexor – informal specification

(mux [x_1 x_2 x_3 …] [y_1 y_2 y_3 …]) = [x_1 y_1 x_2 y_2 x_3 y_3 …]

Definitional properties of mux

(mux nil [y_1 … y_m]) = ??

(mux [x_1 … x_n] nil) = ??

(mux [x_1 … x_{n+1}] [y_1 … y_{m+1}]) = ??

Multiplexor – informal specification

(mux [x_1 x_2 x_3 …] [y_1 y_2 y_3 …]) = [x_1 y_1 x_2 y_2 x_3 y_3 …]

Definitional properties of mux

the 3 c's

(mux nil [y_1 … y_m]) = [y_1 … y_m]

(mux [x_1 … x_n] nil) = [x_1 … x_n])

(mux [x_1 … x_{n+1}] [y_1 … y_{m+1}]) = [x_1 y_1 … ?? …]

(mux [x_2 … x_{n+1}] [y_2 … y_{m+1}])

Formal definition (ACL2)

```
(defun mux (xs ys)
  (if (not (consp xs))
      ys                                        ; mux0
      (if (not (consp ys))
          xs                                    ; mux10
          (cons (first xs) (cons (first ys)
                (mux (rest xs) (rest ys))))))) ; mux11
```


Demultiplexor – informal specification

(dmx [x_1 x_2 x_3 …]) = [[x_1 x_3 x_5 …] [x_2 x_4 x_6 …]]

Some properties of dmx

(dmx nil) = [nil nil]

(dmx [x_2 … x_{n+1}]) = [[x_2 x_4 x_6 …] [x_3 x_5 x_7 …]]

= [odds evns]

the 3 c's

(dmx [x_1 x_2 … x_{n+1}]) = [(cons x_1 odds) evns]

Formal definition (ACL2)

```
(defun dmx (xys)
  (if (consp xys)
      (let* ((x (first xys))
             (ysxs (dmx (rest xys)))
             (ys (first ysxs))
             (xs (second ysxs)))
        (list (cons x xs) ys))          ; dmx1
      (list xys xys)))                  ; dmx0
```

eh?

Expectations – mux, dmx

(mux [x_1 x_2 x_3 …] [y_1 y_2 y_3 …]) = [x_1 y_1 x_2 y_2 x_3 y_3 …]

(dmx [x_1 x_2 x_3 …]) = [[x_1 x_3 x_5 …] [x_2 x_4 x_6 …]]

Length preservation

```
(defthm dmx-preserves-length-thm
  (= (len xys)
     (+ (len (first (dmx xys)))
        (len (second (dmx xys))))))

(defthm mux-preserves-length-thm
  (= (len (mux xs ys))
     (+ (len xs) (len ys))))
```


Expectations – mux, dmx

(mux [x_1 x_2 x_3 …] [y_1 y_2 y_3 …]) = [x_1 y_1 x_2 y_2 x_3 y_3 …]

(dmx [x_1 x_2 x_3 …]) = [[x_1 x_3 x_5 …] [x_2 x_4 x_6 …]]

Conservation of elements

```
(defthm dmx-conserves-of-elements-thm
  (iff (member-equal e xys)
       (or (member-equal e (first (dmx xys)))
           (member-equal e (second (dmx xys))))))

(defthm mux-conserves-of-elements-thm
          (iff (member-equal e (mux xs ys))
               (or (member-equal e xs)
                   (member-equal e ys))))
```


Round-trip properties – mux, dmx

(mux [x_1 x_2 x_3 …] [y_1 y_2 y_3 …]) = [x_1 y_1 x_2 y_2 x_3 y_3 …]

(dmx [x_1 x_2 x_3 …]) = [[x_1 x_3 x_5 …] [x_2 x_4 x_6 …]]

mux inverts dmx

```
(defproperty mux-inverts-dmx-tst
  (xys :value (random-list-of (random-integer)))
  (equal (mux (first  (dmx xys))
              (second (dmx xys)))
         xys))
```


Round-trip properties – mux, dmx

(mux [x_1 x_2 x_3 …] [y_1 y_2 y_3 …]) = [x_1 y_1 x_2 y_2 x_3 y_3 …]

(dmx [x_1 x_2 x_3 …]) = [[x_1 x_3 x_5 …] [x_2 x_4 x_6 …]]

mux inverts dmx

```
(defproperty mux-inverts-dmx-tst
  (xys :value (random-list-of (random-integer)))
  (equal (mux (first  (dmx xys))
              (second (dmx xys)))
         xys))
```

dmx inverts mux

```
(defproperty dmx-inverts-mux-tst
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (dmx (mux xs ys))
         (list xs ys)))
```

demo 3

Round-trip properties – mux, dmx

(mux [x_1 x_2 x_3 …] [y_1 y_2 y_3 …]) = [x_1 y_1 x_2 y_2 x_3 y_3 …]

(dmx [x_1 x_2 x_3 …]) = [[x_1 x_3 x_5 …] [x_2 x_4 x_6 …]]

mux inverts dmx

```
(defproperty mux-inverts-dmx-tst
  (xys :value (random-list-of (random-integer)))
  (equal (mux (first  (dmx xys))
              (second (dmx xys)))
         xys))
```

dmx inverts mux

```
(defproperty dmx-inverts-mux-tst
  (xs :value (random-list-of (random-integer))
   ys :value (random-list-of (random-integer)))
  (equal (dmx (mux xs ys))
         (list xs ys)))
```

demo 3

whoops !

Round-trip properties – mux, dmx

(mux [x_1 x_2 x_3 …] [y_1 y_2 y_3 …]) = [x_1 y_1 x_2 y_2 x_3 y_3 …]

(dmx [x_1 x_2 x_3 …]) = [[x_1 x_3 x_5 …] [x_2 x_4 x_6 …]]

mux inverts dmx

```
(defproperty mux-inverts-dmx-tst
  (xys :value (random-list-of (random-integer)))
  (equal (mux (first  (dmx xys))
              (second (dmx xys)))
         xys))
```

dmx inverts mux

```
(defproperty dmx-inverts-mux-tst
  (n  :value (random-natural)
   xs :value (random-integer-list-of-length n)
   ys :value (random-integer-list-of-length n))
  (implies (and (true-listp xs) (true-listp ys)
                (= (len xs) (len ys)))
           (equal (dmx (mux xs ys))
                  (list xs ys))))
```

demo 3B

Ordered merge

informal spec

(mrg [x_1 x_2… x_n] [y_1 y_2 … y_m]) = [z_1 z_2 … z_{n+m}]
where $z_1 \le z_2 \le \ldots \le z_{n+m}$ if
if $x_1 \le x_2 \le \ldots \le x_n$ and $y_1 \le y_2 \le \ldots \le y_m$

definitional properties of mrg

(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys))) {x≤y}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys)) {y<x}
(mrg xs nil) = xs {mg0}
(mrg nil ys) = ys {mg1}

Ordered merge

informal spec

(mrg [x_1 x_2… x_n] [y_1 y_2 … y_m]) = [z_1 z_2 … z_{n+m}]
where $z_1 \le z_2 \le \ldots \le z_{n+m}$ if
if $x_1 \le x_2 \le \ldots \le x_n$ and $y_1 \le y_2 \le \ldots \le y_m$

definitional properties of mrg

(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys))) {x≤y}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys)) {y<x}
(mrg xs nil) = xs {mg0}
(mrg nil ys) = ys {mg1}

formal definition

```
(defun mrg (xs ys)
  (if (and (consp xs) (consp ys))
      (let* ((x (first xs)) (y (first ys)))
        (if (<= x y)
            (cons x (mrg (rest xs) ys))    ; {mgx}
            (cons y (mrg xs (rest ys))))) ; {mgy}
      (if (not (consp ys))
          xs          ; ys is empty        ; {mg0}
          ys)))       ; xs is empty        ; {mg1}
```


Ordered merge

informal spec

(mrg [x_1 x_2… x_n] [y_1 y_2 … y_m]) = [z_1 z_2 … z_{n+m}]
where $z_1 \le z_2 \le \ldots \le z_{n+m}$ if
if $x_1 \le x_2 \le \ldots \le x_n$ and $y_1 \le y_2 \le \ldots \le y_m$

definitional properties of mrg

(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys))) {x≤y}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys)) {y<x}
(mrg xs nil) = xs {mg0}
(mrg nil ys) = ys {mg1}

additional expectations

```
(defthm mrg-preserves-order-thm
   (implies (and (up xs) (up ys))
            (up (mrg xs ys))))
```

(up [x_1 x_2… x_n]) = $x_1 \le x_2 \le \ldots \le x_n$

Ordered merge

informal spec

(mrg [x_1 x_2… x_n] [y_1 y_2 … y_m]) = [z_1 z_2 … z_{n+m}]
where $z_1 \le z_2 \le \ldots \le z_{n+m}$ if
if $x_1 \le x_2 \le \ldots \le x_n$ and $y_1 \le y_2 \le \ldots \le y_m$

definitional properties of mrg

(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys))) {x≤y}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys)) {y<x}
(mrg xs nil) = xs {mg0}
(mrg nil ys) = ys {mg1}

additional expectations

```
(defthm mrg-preserves-order-thm
   (implies (and (up xs) (up ys))
            (up (mrg xs ys))))
(defthm mrg-conservation-of-values-thm
  (iff (member-equal e (mrg xs ys))
       (or (member-equal e xs) (member-equal e ys))))
```

(up [x_1 x_2… x_n]) = $x_1 \le x_2 \le \ldots \le x_n$
(member-equal x [x_1 x_2… x_n]) = ($x=x_1$) or ($x=x_1$) or … or ($x=x_n$)

Merge-sort

informal spec

(msort [x_1 x_2… x_n] = [z_1 z_2 … z_n]
where $z_1 \le z_2 \le \ldots \le z_n$
and [z_1 z_2 … z_n] is a permutation of = [z_1 z_2 … z_n]

definitional properties of merge-sort

(msort [ ]) = [ ] {ms0}
(msort [x]) = [x] {ms1}
(msort [x_1 x_2… x_n]) = (mrg (msort [x_1 x_2… $x_{\lceil n/2 \rceil}$] (msort [$x_{\lceil n/2 \rceil+1}$ … x_n])) {ms2}

additional expectations

```
(defthm msort-order-thm
   (up (msort xs))))
(defthm msort-conservation-of-values-thm
   (iff (member-equal e (msort xs))
        (member-equal e xs)))
```


Merge-sort

informal spec

(msort [x_1 x_2… x_n] = [z_1 z_2 … z_n]
where $z_1 \le z_2 \le \ldots \le z_n$
and [z_1 z_2 … z_n] is a permutation of = [z_1 z_2 … z_n]

definitional properties of merge-sort

(msort [ ]) = [ ] {ms0}
(msort [x]) = [x] {ms1}
(msort [x_1 x_2… x_n]) = (mrg (msort [x_1 x_2… $x_{\lceil n/2 \rceil}$] (msort [$x_{\lceil n/2 \rceil+1}$ … x_n])) {ms2}

additional expectations

```
(defthm msort-order-thm
   (up (msort xs))))
(defthm msort-conservation-of-values-thm
   (iff (member-equal e (msort xs))
        (member-equal e xs)))
```

demo 4

any split putting half of the elements in one list and half in the other is okay

Ordered merge

informal spec

(mrg [x_1 x_2… x_n] [y_1 y_2 … y_m]) = [z_1 z_2 … z_{n+m}]
where $z_1 \le z_2 \le \ldots \le z_{n+m}$ if
if $x_1 \le x_2 \le \ldots \le x_n$ and $y_1 \le y_2 \le \ldots \le y_m$

definitional properties of mrg

```
(mrg (cons x xs) (cons y ys)) = (cons x (mrg xs (cons y ys)))   {x≤y}
(mrg (cons x xs) (cons y ys)) = (cons y (mrg (cons x xs) ys))   {y<x}
(mrg xs nil) = xs                                               {mg0}
(mrg nil ys) = ys                                               {mg1}
```

formal definition

```
(defun mrg (xs ys)
  (declare (xargs :measure (+ (len xs) (len ys))))
  (if (and (consp xs) (consp ys))
      (let* ((x (first xs)) (y (first ys)))
        (if (<= x y)
            (cons x (mrg (rest xs) ys))          ; {mgx}
            (cons y (mrg xs (rest ys))))) ; {mgy}
      (if (not (consp ys))
          xs                  ; ys is empty      ; {mg0}
          ys)))               ; xs is empty      ; {mg1}
```

suggested induction scheme

demo 4B

Merge-sort

formal definition of merge-sort

```
(defun msort (xs)
  (if (consp (rest xs)) ; (len xs) > 1?
      (let* ((odds-evens (dmx xs)) ; xs = [x1 x2 ...]
             (odds       (first odds-evens))
             (evns       (second odds-evens)))
        (mrg (msort odds) (msort evns))) ;    {ms2}
      xs))                ; xs = [x1] or empty {ms1}
```


Merge-sort

formal definition of merge-sort

```
(defun msort (xs)
  (if (consp (rest xs)) ; (len xs) > 1?
      (let* ((odds-evens (dmx xs)) ; xs = [x1 x2 ...]
             (odds       (first odds-evens))
             (evns       (second odds-evens)))
        (mrg (msort odds) (msort evns))) ;    {ms2}
      xs))                ; xs = [x1] or empty {ms1}
```

maybe it doesn't know dmx reduces the list length

```
(defthm dmx-reduces-len-thm ; lemma, msort termination
   (implies (consp (rest xs))
            (and (< (len(first(dmx xs))) (len xs))
                 (< (len(second(dmx xs))) (len xs)))))
```


Merge-sort

formal definition of merge-sort

demo 4C

```
(defun msort (xs)
  (declare  ; suggest using lemma to prove termination
   (xargs :measure (len xs)
          :hints (("Goal"
            :use ((:instance dmx-reduces-len-thm))))))
  (if (consp (rest xs)) ; (len xs) > 1?
      (let* ((odds-evens (dmx xs)) ; xs = [x1 x2 ...]
             (odds       (first odds-evens))
             (evns       (second odds-evens)))
        (mrg (msort odds) (msort evns))) ;    {ms2}
      xs))                ; xs = [x1] or empty {ms1}
```

maybe it doesn't know dmx reduces the list length

```
(defthm dmx-reduces-len-thm ; lemma, msort termination
   (implies (consp (rest xs))
            (and (< (len(first(dmx xs))) (len xs))
                 (< (len(second(dmx xs))) (len xs)))))
```


Practice exercises (30 minutes)

Linear Encoding

Define operators to encode and decode lists of natural numbers within a given range, and verify that decode inverts encode

Encode

The encoding process adds adjacent numbers in the input list, modulo the given range

Decode

Inverse of encode

Project description

http://blog.accurate-programming.org/

The End

June 4, 2013
14:00-15:30 session