

# DrACuLa: ACL2 in DrScheme

Dale Vaillancourt   Rex Page   Matthias Felleisen



# Undergraduate Software Engineering

Page teaches two SE courses at Oklahoma U.

- Covers usual topics - specification, documentation, time management, teamwork, code reviews, defect control, etc.
- Special emphasis on specification and defect control using logic.

# Undergraduate Software Engineering

Two typical course projects:

- Read in an image file, rotate it, flip it write a new file.
  - Prove `(equal (rotate image 360) image)`
  - Prove `(equal (flip (flip image)) image)`
- Read in stock market data, compute statistics. Prove arithmetic properties.

# Undergraduate Software Engineering

Overall, good results

- Student course evaluations are (mostly) positive
- Industry observers like what they see in code reviews.

## Three Problems Recur

- ACL2 environment is unfriendly & overwhelming.
- Projects need to engage students.
  - Text based exercises are too small and boring.
  - Current projects are not representative of what students will work on in the future.
- Students are new to (functional) program design.
  - Corollary: Students therefore have trouble reasoning and proving theorems about their programs.

# Our Proposal

- Build a student friendly ACL2 environment: Dracula.
- Environment should support development of graphical interactive software.
- Integrate program design into curriculum (longer term goal).

# **Dracula: A Student Friendly ACL2 Environment**

## A Friendly Reminder

- DrScheme & associated pedagogy represent over a decade of research and experience on friendly programming environments.
- Dracula is built on top of DrScheme.
- See section 2 of paper for more background on DrScheme.



# Being Friendly

- Smaller language => improved error messages.
- Graphically connect errors to the program source.
- Use a simple GUI to control the environment.

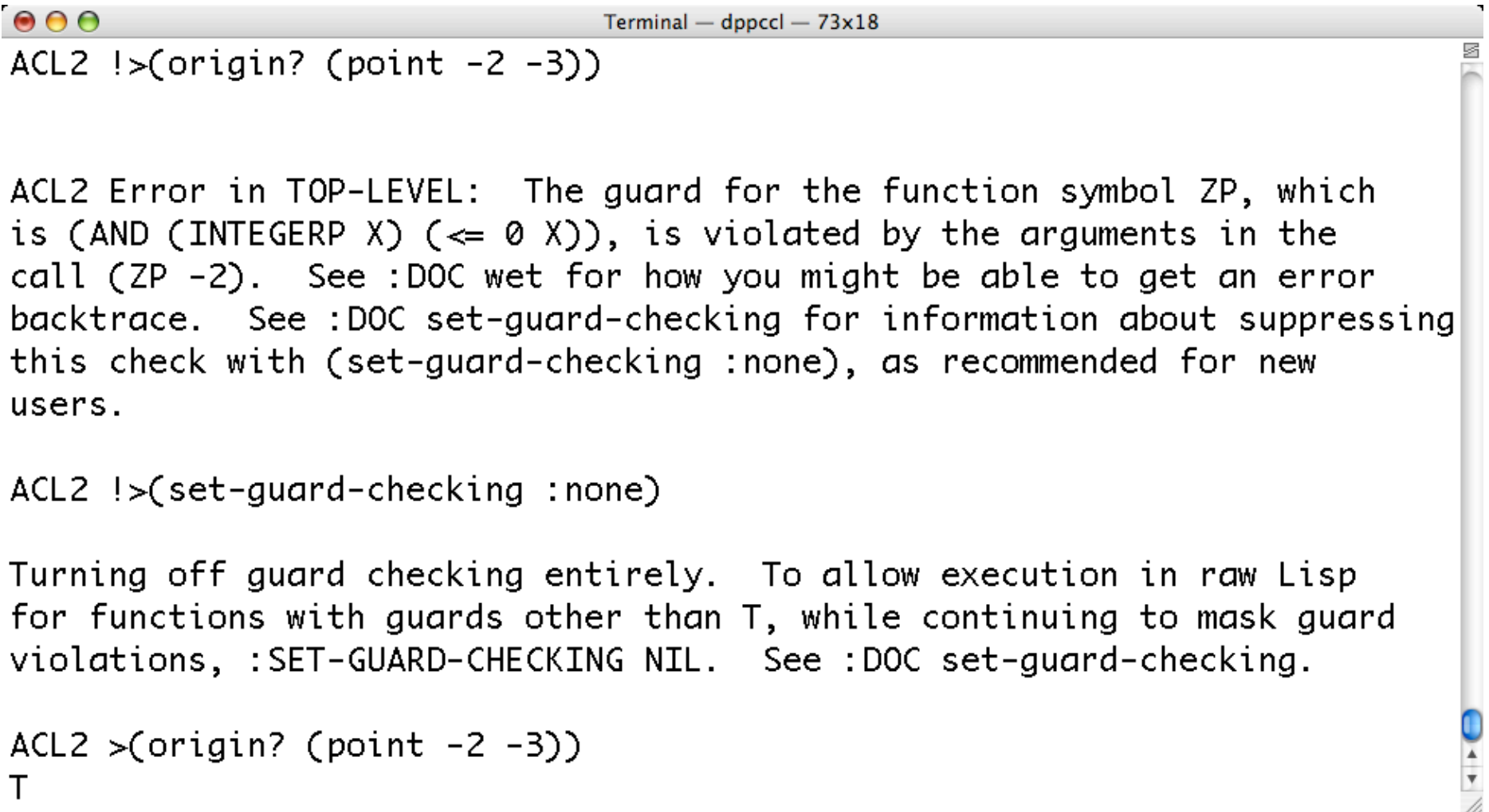
# Dracula's ACL2 Language

Provided forms:

- `defun`, `defconst`, `defthm`
- `cond`, `and`, `or`, `if`
- `defstructure`, `deflist`
- `let`, `quote`

Provides all documented primitive procedures.

# Guard Errors



```
Terminal — dppccl — 73x18
ACL2 !>(origin? (point -2 -3))

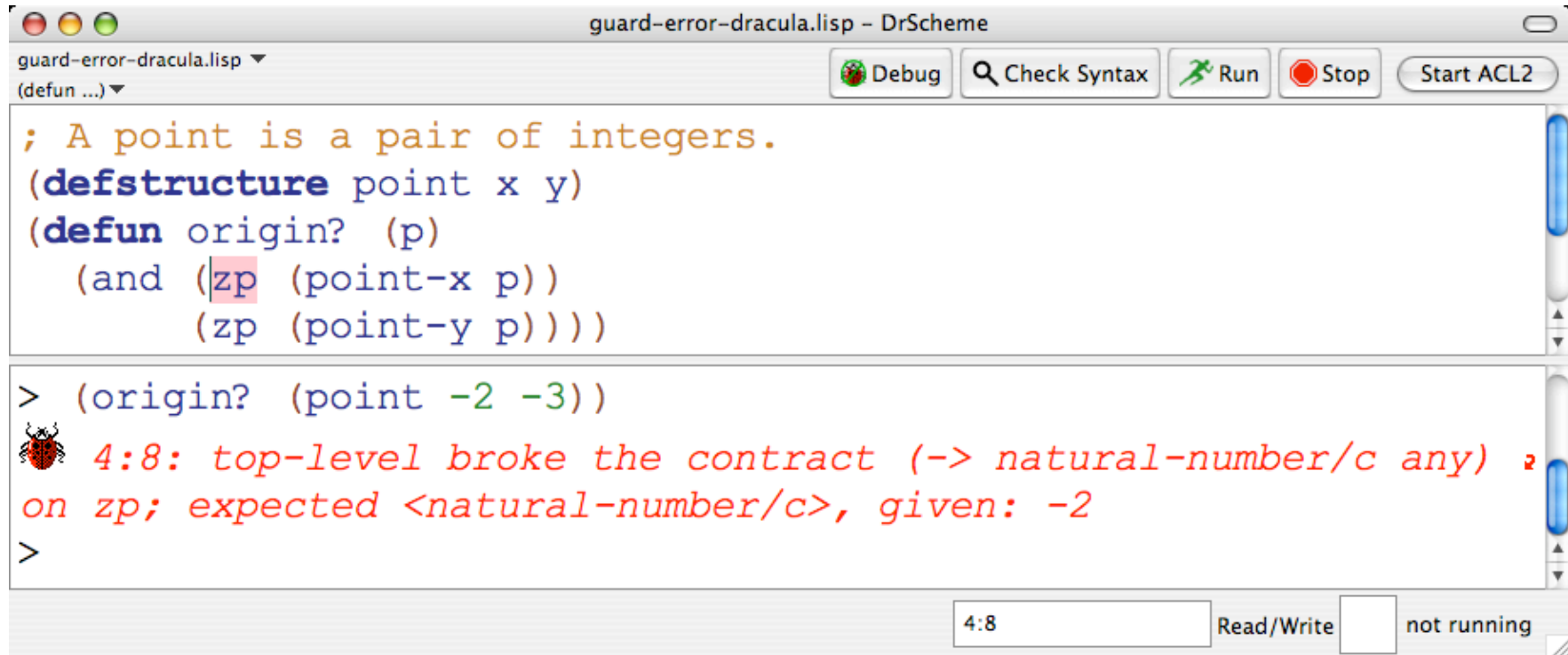
ACL2 Error in TOP-LEVEL: The guard for the function symbol ZP, which
is (AND (INTEGERP X) (<= 0 X)), is violated by the arguments in the
call (ZP -2). See :DOC wet for how you might be able to get an error
backtrace. See :DOC set-guard-checking for information about suppressing
this check with (set-guard-checking :none), as recommended for new
users.

ACL2 !>(set-guard-checking :none)

Turning off guard checking entirely. To allow execution in raw Lisp
for functions with guards other than T, while continuing to mask guard
violations, :SET-GUARD-CHECKING NIL. See :DOC set-guard-checking.

ACL2 >(origin? (point -2 -3))
T
```


# Guard Errors



The screenshot shows the DrScheme IDE window titled "guard-error-dracula.lisp - DrScheme". The editor contains the following code:

```
; A point is a pair of integers.  
(defstructure point x y)  
(defun origin? (p)  
  (and (zp (point-x p))  
        (zp (point-y p))))
```

The variable `zp` is highlighted in pink. Below the code, the REPL shows the command `> (origin? (point -2 -3))` and a red error message:

```
 4:8: top-level broke the contract (-> natural-number/c any) on zp; expected <natural-number/c>, given: -2
```

The status bar at the bottom indicates the error location as `4:8`, with `Read/Write` and `not running` buttons.

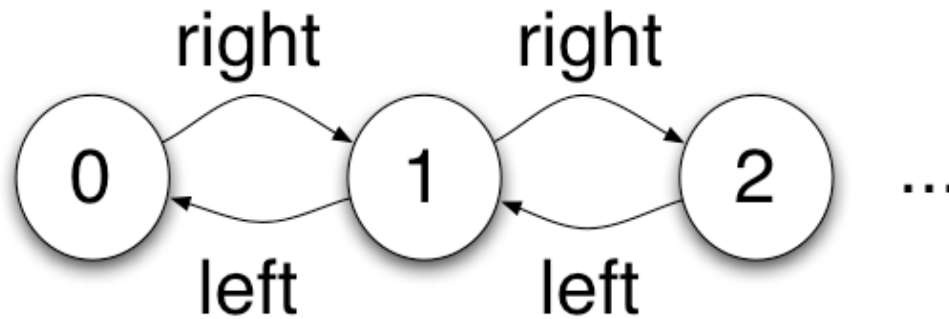
# Short Demonstration

# Developing Interactive, Graphical Programs

(Functionally!)

# Structure of an Interactive Program

Consider a simple counter program.



## Represent automaton as an ACL2 program

```
; An Action is either 'left or 'right

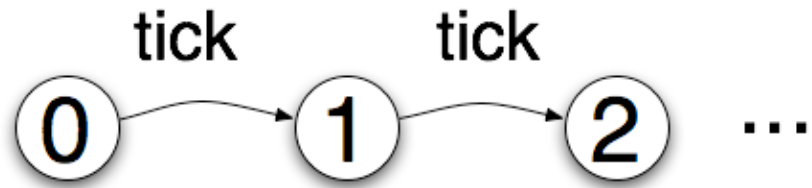
; A World is an integer

; transition : World Action → World
(defun transition (a-world action)
  (case action
    ((left) (1- a-world))
    ((right) (1+ a-world))))
```



## Tick Tock

Suppose we want the counter to increment automatically with a given clock period.



## Counter Program v.2

```
; A World is a Natural Number  
  
; Transition function  
; tick : World -> World  
(defun tick (w) (1+ w))
```

## Making it run

We must also specify how to render each world:

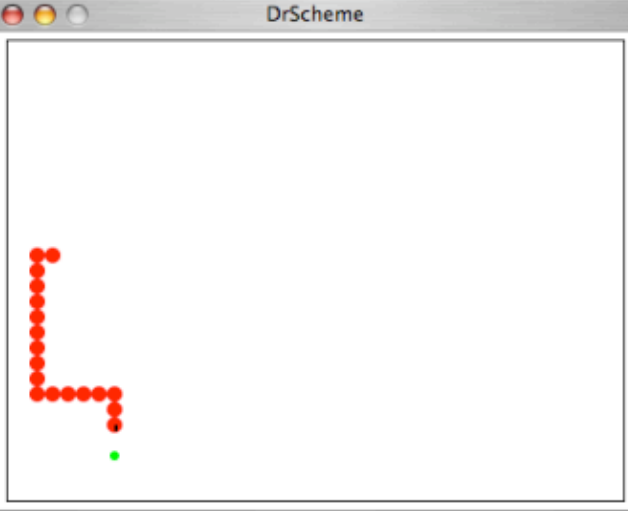
```
; render-world : World -> Image
; produce an image of the given world
(defun render-world (the-world)
  (text (int->string the-world) *size* *color*))
```

And we get the ball rolling with a few macro invocations:

```
(big-bang *width* *height*
          *clock-period* *initial-world*)
(on-key-event transition)
(on-tick-event tick)
(on-redraw render-world)
```

# A Little Fancier: Worm

```
worm.lisp - DrScheme
worm.lisp
(define ...)
When are two segments adjacent? We know
diameter *DIAMETER*. Further, given our
can only move vertically and horizontally;
of segments must have the same x-coordina
by *DIAMETER*. Or, their x-coordinates c
the same y-coordinates.
|#
;; adjacent? : Segment Segment -> Boolean
(defun adjacent? (seg-1 seg-2)
  (let ((x1 (posn-x seg-1))
        (y1 (posn-y seg-1))
        (x2 (posn-x seg-2))
        (y2 (posn-y seg-2)))
    (or (and (= x1 x2) (= (abs (- y1 y2))
                          *DIAMETER*))
        (and (= y1 y2) (= (abs (- x1 x2))
                          *DIAMETER*)))))
;; consecutive-segments-adjacent? : list-
;; Is every consecutive pair of segments
(defun consecutive-segments-adjacent? (los)
  (if (consp los)
      (or (null (cdr los))
          (and (adjacent? (car los) (cadr los))
                (consecutive-segments-adjacent? (cdr los))))
      (null los)))
```



## Representing the Worm

```
; A Segment is (segment natural natural)
(defstructure segment x y)
(defconst *segment-diameter* 10)

; Velocity is in '(up down left right)

; A Worm is (worm segment Velocity segment-list)
(defstructure worm head velocity tail)
```

But, not just any head and segment-list will do.

## Reasoning about GUI Programs

Use the transition functions

```
; worm-well-formed? : any → Boolean
; Is the given object a well-formed worm?
(defun worm-well-formed? (w)
  (consecutive-pairs-adjacent?
   (cons (worm-head w) (worm-tail w))))

(defthm initial-worm-well-formed
  (worm-well-formed? *initial-worm*))

(defthm worm-move-preserves-well-formedness
  (implies (worm-well-formed? w)
            (worm-well-formed? (worm-move w))))
```

Requires fewer than 10 lemmas.

# Classroom Experience

## A Subjective Survey

Last Fall - Page's SE I students used ACL2 via the console and their favorite editor.

Last Spring - same students used Dracula in SE II.

Page surveyed SE II class in April to get their thoughts on Dracula.



## Survey says...

- Students like Dracula's error reporting.
- Students like developing GUI software in Dracula.
- Students like Dracula's GUI to the theorem prover.
- Students identified a couple weaknesses in Dracula's implementation.
- Students want a debugger for Dracula.
- Instructor liked Dracula too. More details in paper.

# Looking Ahead

# Future Work

- Modules
  - for Dracula
  - for ACL2
- Formalize the World Teachpack framework
- Port the HtDP pedagogy to ACL2 program design.

# What is a module?

A **module** is a collection of definitions and expressions (not necessarily closed).

Desideratum: A module can be developed and reasoned about in the absence of implementation modules upon which it may depend.

Two benefits:

- Namespace management - module authors opt to hide certain names (definitions).
- Separate reasoning - programmers can reason about their modules without looking at their colleagues' code (just need a spec).

## Module example

```
; Specification for insert:
(defthm insert-spec
  (implies (and (integerp x)
                (ordered-integer-listp lst))
           (ordered-integer-listp
            (insert x lst))))

(module sort
  (require insert)
  (defun sort (lst)
    (if (endp lst)
        '()
        (insert (first lst)
                 (sort (rest lst)))))
  (defthm sort-theorem
    (ordered-integer-listp (sort lst))))
```

## ACL2 lacks modules

Puzzle: Provide `g` but hide `helper`.

```
(defun helper (x) (+ x 2))  
(defun g (x) (helper (helper x)))
```

- `include-book`, `encapsulate`, and `local` will not do the trick.
- Packages do not suffice. Clients gets to decide which names to "hide" when building packages.

## Dracula's Modules

```
(module insert
  (provide/spec insert
    [integerp ordered-integer-listp
     -> ordered-integer-listp])
  (define insert (x lst) ...))
```

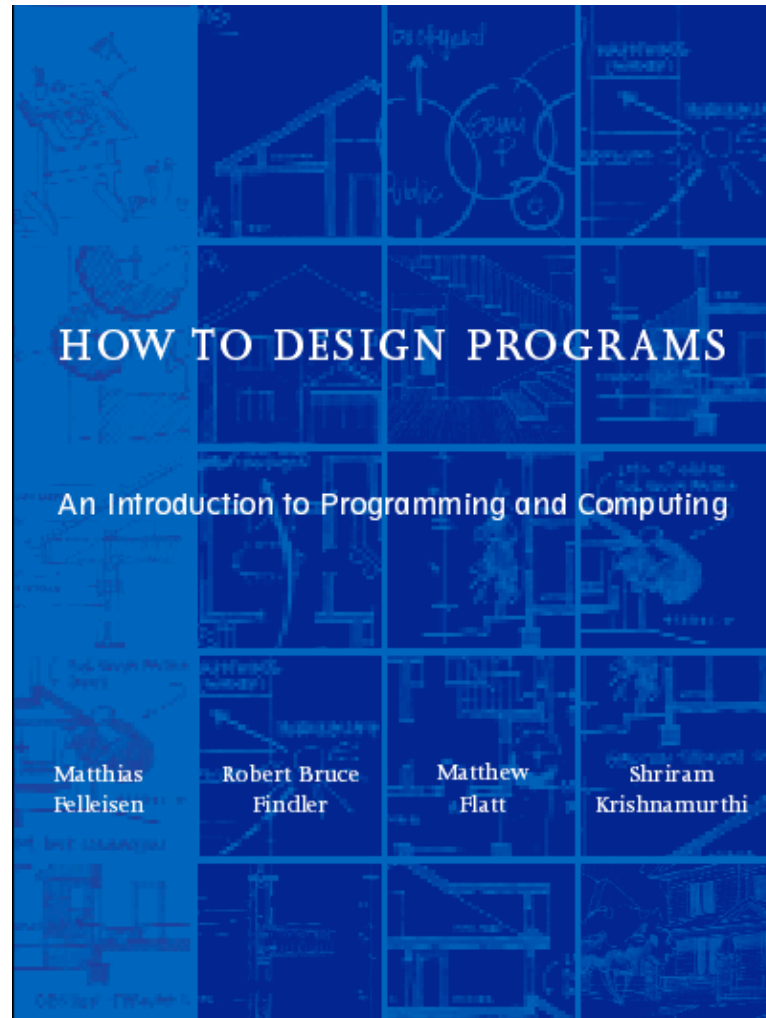
```
(module sort
  (require insert)
  (define sort (lst)
    ... (insert (first lst)
               (sort (rest lst)))
    ...))
```

# Formalizing the World Teachpack

There is still a logical disconnect between World-based programs and the actual behavior of the program.



# HtDP



## Data Definition

Suppose we want to design function to sort integer lists. Structure of code follows structure of data.

An `integer-list` is either

- `nil`, or
- `(cons integer integer-list)`

# Design Recipe for Sorting

Problem: Design a function that sorts a given list of integers.

## Design Recipe for Sorting

Problem: Design a function that sorts a given list of integers.

```
; sort : int-list → int-list  
; sort the given list  
(defun sort (alon)  
  ...)
```

## Design Recipe for Sorting

Problem: Design a function that sorts a given list of integers.

```
; sort : int-list → int-list  
; sort the given list  
(defun sort (alon)  
  ...)
```

```
(equal (sort nil) nil)  
(equal (sort '(3 2 1)) '(1 2 3))
```

## Design Recipe for Sorting

Problem: Design a function that sorts a given list of integers.

```
; sort : int-list → int-list
; sort the given list
(defun sort (alon)
  (cond ((endp alon) ...)
        ((cons? alon)
         ... (first alon) ...
         ... (rest alon) ...
         ... (sort (rest alon)) ...)))
```

```
(equal (sort nil) nil)
(equal (sort '(3 2 1)) '(1 2 3))
```

## Design Recipe for Sorting

Problem: Design a function that sorts a given list of integers.

```
; sort : int-list → int-list
; sort the given list
(defun sort (alon)
  (cond ((endp alon) nil)
        ((cons? alon)
         ... (first alon) ...
         ... (rest alon) ...
         ... (sort (rest alon)) ...)))

(equal (sort nil) nil)
(equal (sort '(3 2 1)) '(1 2 3))
```

## Design Recipe for Sorting

Problem: Design a function that sorts a given list of integers.

```
; sort : int-list → int-list
; sort the given list
(defun sort (alon)
  (cond ((endp alon) nil)
        ((cons? alon)
         (insert (first alon)
                 (sort (rest alon))))))
```

```
(equal (sort nil) nil)
(equal (sort '(3 2 1)) '(1 2 3))
```



# Proving a Theorem

; Prove half the correctness for sort

## Proving a Theorem

```
; Prove half the correctness for sort  
(defthm sort-produces-an-ordered-list  
  (ordered? (sort a-list)))
```

## Proving a Theorem

```
; Prove half the correctness for sort  
(defthm sort-produces-an-ordered-list  
  (ordered? (sort a-list)))
```

```
(ordered? (sort nil))  
(ordered? (sort '(3 2 1)))
```

## Proving a Theorem

```
; Prove half the correctness for sort
(defthm sort-produces-an-ordered-list
  (ordered? (sort a-list)))
```

```
(ordered? (sort nil))
(ordered? (sort '(3 2 1)))
```

Induction on `a-list` leads to:

```
(defthm insert-preserves-ordered?
  (implies (and (number? num)
                (ordered? a-list))
            (ordered? (insert num a-list))))
```

## Proving a Theorem

```
; Prove half the correctness for sort
(defthm sort-produces-an-ordered-list
  (ordered? (sort a-list)))
```

```
(ordered? (sort nil))
(ordered? (sort '(3 2 1)))
```

Induction on `a-list` leads to:

```
(defthm insert-preserves-ordered?
  (implies (and (number? num)
                (ordered? a-list))
           (ordered? (insert num a-list))))
```

# Design Recipe & The Method

## Design Recipe:

- Data Definition
- Contract & Purpose
- Examples
- Template
- Implementation
- Tests

## The Method:

- Purpose
- Conjecture
- Test
- Attempt Proof
- Discover Lemmas
- Complete Proof

Not the same, but similar in spirit!

## Wrapping up

## Related Work

Dracula's GUI is inspired by that found in systems such as:

- CoqIDE
- ProofGeneral

ACL2s - Eclipse-based environment by Dillinger, Manolios, & Vroon



# Implementation

Dracula is a plug-in for DrScheme.

- Translates Lisp to Scheme using PLT Scheme macros.
- Inherits lots of DrScheme infrastructure automatically.

First prototype took two weeks of my time.

- Documentation, testing, etc. took another few weeks.
- Schedule is realistic for you: I started off knowing only a little bit about PLT Scheme macros and DrScheme's extension API.

Thank You

<http://www.ccs.neu.edu/~dalev/ac12/>



## Formalizing Images

An ACL2 book specifies a new datatype of images.

Graphical functions are constrained to consume / produce images.

```
(defthm image?-predicate
  (booleanp (image? X)))

(defthm image?-distinct
  (implies (image? X)
    (and (not (symbolp X))
         (not (consp X))
         ...)))

(defthm circle-is-an-image
  (image? (circle radius color fill)))
...
```

Specification is weak, but no matter.