Functional Programming and Theorem Proving for Undergraduates

A Progress Report

Carl Eastlund and Matthias Felleisen
Northeastern University

Rex Page
University of Oklahoma

History

- Before 2003
  - Traditional SE at OU (2-course sequence, 4th yr)
  - Process Design Testing/Validation
  - 60% 20% 20%
- 2003-2005
  - SE course using ACL2 (FDPE 2005 report)
  - Process Design Testing/Validation
  - 30% 35% 35%
  - Successful despite crude programming env
- 2006 - present
  - SE course with Dracula/ACL2 environment
  - 1st year course at NU using Dracula/ACL2

Mantra

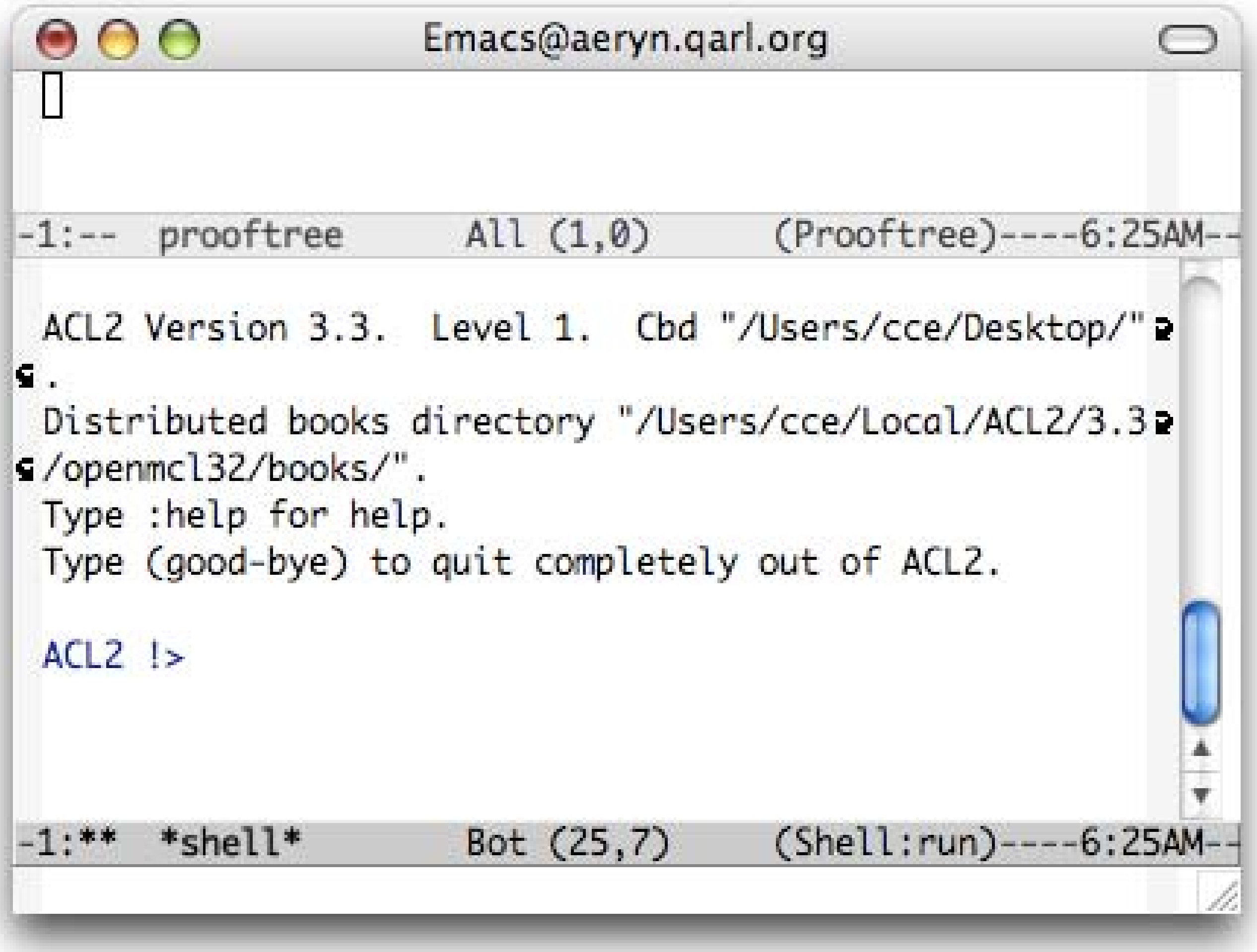
- Before 2003
  - Traditional SE at OU ... r)
  - Process ... ation
  - 60% ...
- ...
  - ...ng ACL2 (FDPE 2005 report)
  - ...ess Design Testing/Validation
  - 30% 35% 35%
  - Successful despite crude programming env
- 2006 - present
  - SE course with Dracula/ACL2 environment
  - 1st year course at NU using Dracula/ACL2

Engineering is the application of principles of science and mathematics to the design of useful things

ACL2

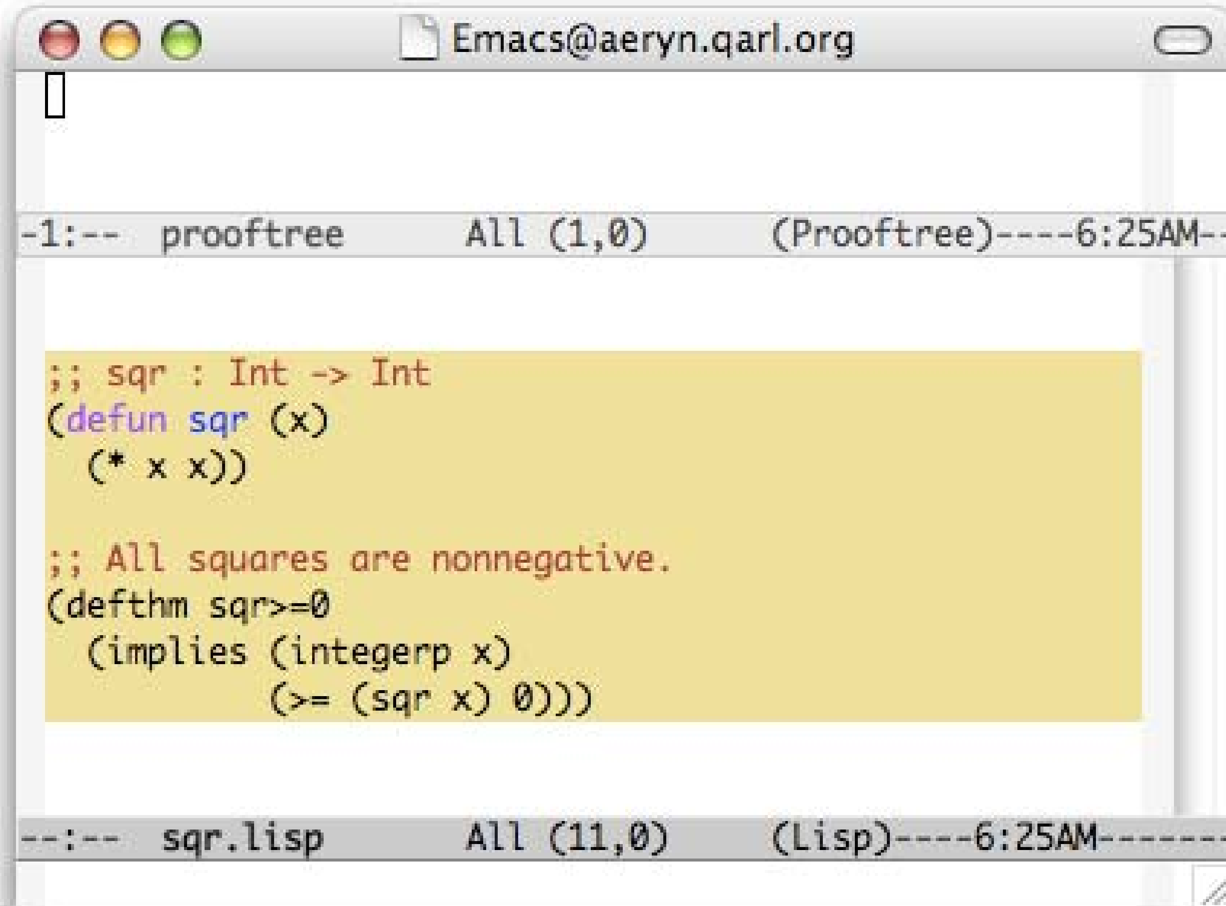
```
;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))
```

ACL2

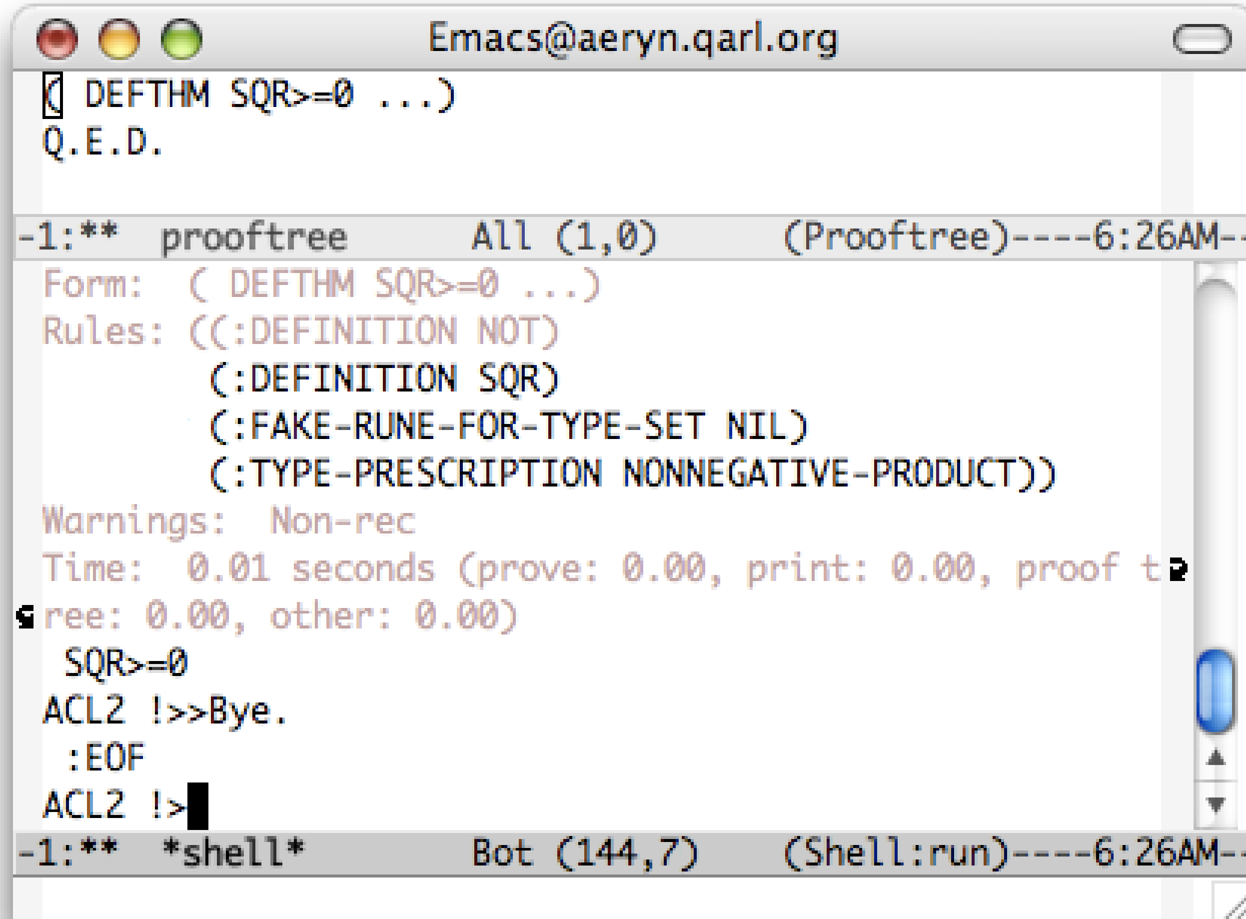


Emacs@aeryn.qarl.org
-1:-- prooftree All (1,0) (Prooftree)----6:25AM--
ACL2 Version 3.3. Level 1. Cbd "/Users/cce/Desktop/"
.
Distributed books directory "/Users/cce/Local/ACL2/3.3
/openmcl32/books/".
Type :help for help.
Type (good-bye) to quit completely out of ACL2.
ACL2 !>
-1:** *shell* Bot (25,7) (Shell:run)----6:25AM--

ACL2

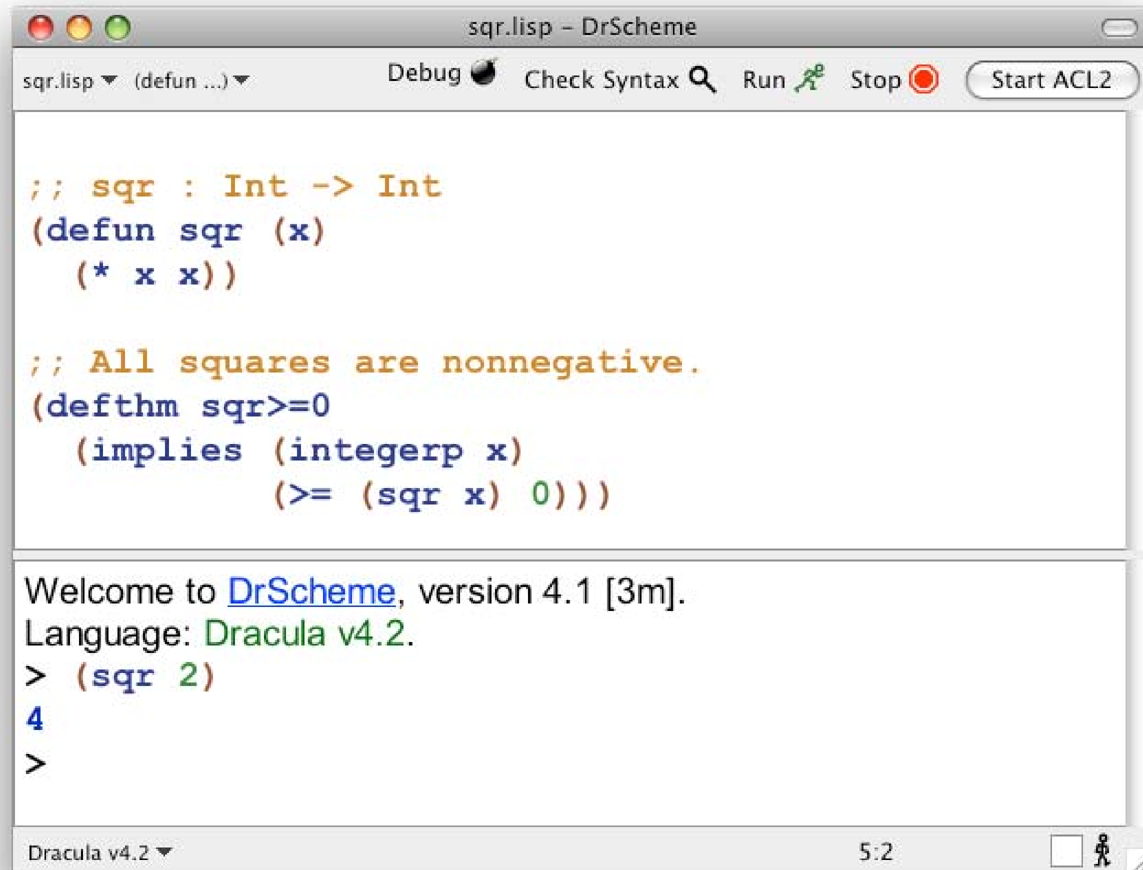

Emacs@aeryn.qarl.org
-1:-- prooftree All (1,0) (Prooftree)----6:25AM--
;; sqr : Int -> Int
(defun sqr (x)
  (* x x))
;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))
--:-- sqr.lisp All (11,0) (Lisp)----6:25AM------

ACL2

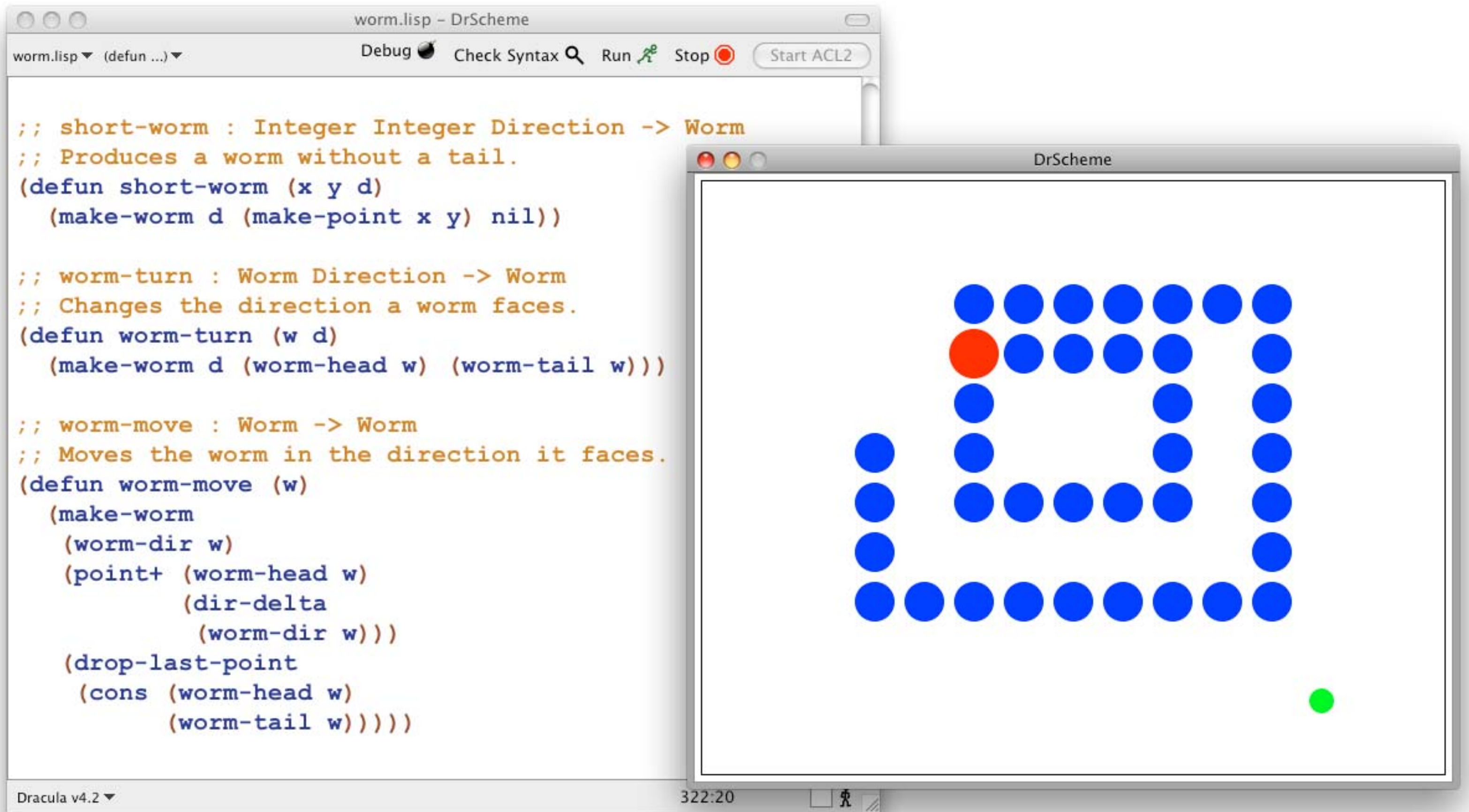


Emacs@aeryn.qarl.org
(DEFTHM SQR>=0 ...)
Q.E.D.
-1:** prooftree All (1,0) (Prooftree)----6:26AM--
Form: (DEFTHM SQR>=0 ...)
Rules: ((:DEFINITION NOT)
(:DEFINITION SQR)
(:FAKE-RUNE-FOR-TYPE-SET NIL)
(:TYPE-PRESCRIPTION NONNEGATIVE-PRODUCT))
Warnings: Non-rec
Time: 0.01 seconds (prove: 0.00, print: 0.00, proof t
ree: 0.00, other: 0.00)
SQR>=0
ACL2 !>>Bye.
:EOF
ACL2 !>
-1:** *shell* Bot (144,7) (Shell:run)----6:26AM--

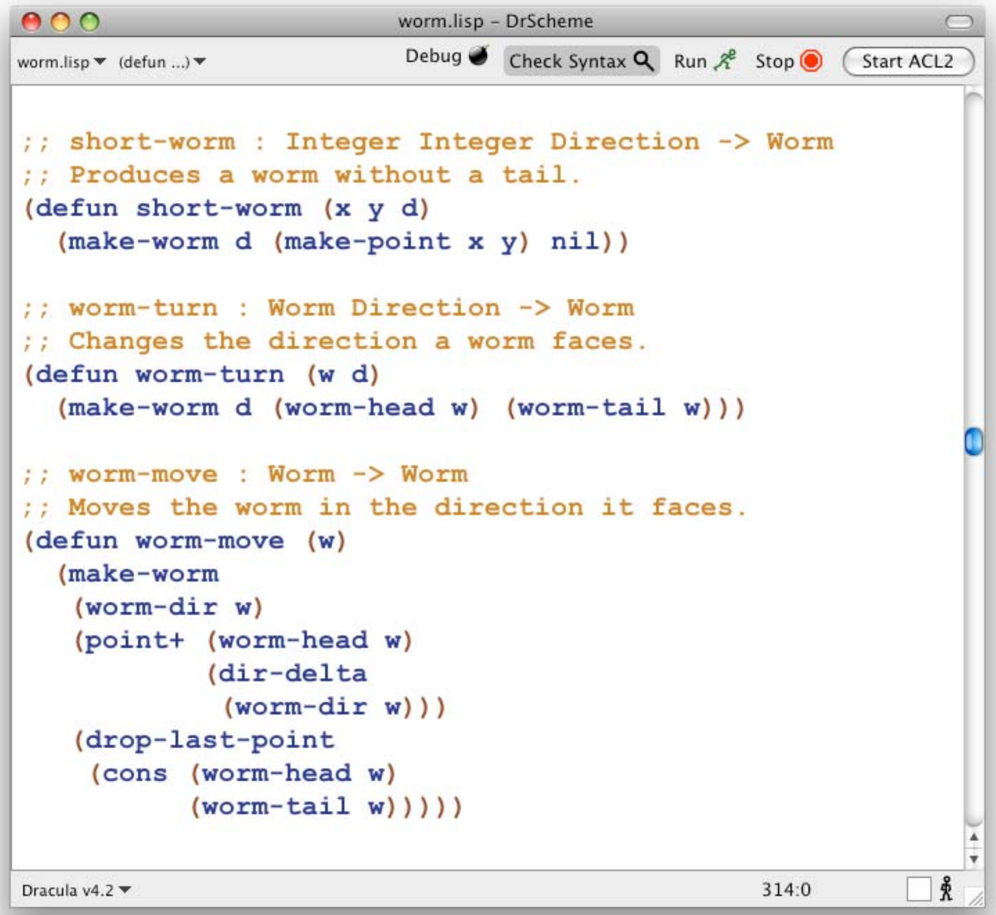
Dracula


sqr.lisp - DrScheme
sqr.lisp (defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
;; sqr : Int -> Int
(defun sqr (x)
  (* x x))
;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))
Welcome to DrScheme, version 4.1 [3m].
Language: Dracula v4.2.
> (sqr 2)
4
>
Dracula v4.2
5:2

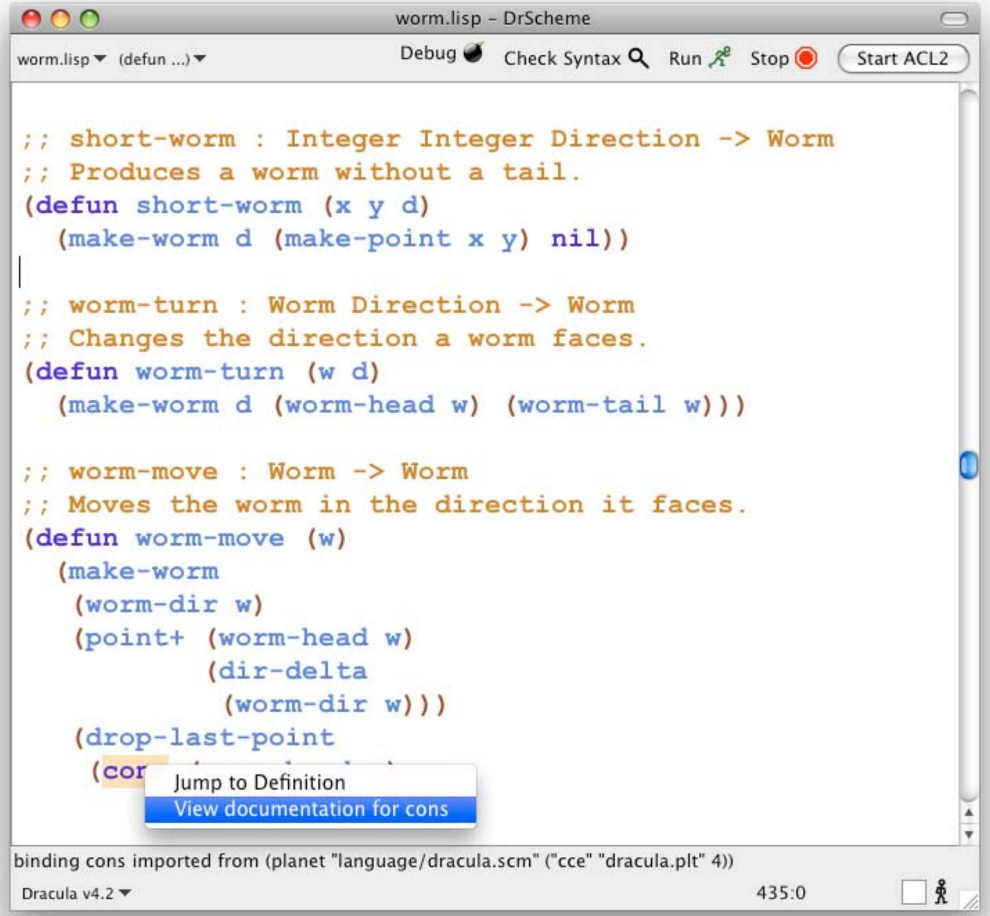
Dracula


worm.lisp - DrScheme
worm.lisp (defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
(make-worm d (make-point x y) nil))
;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
(make-worm d (worm-head w) (worm-tail w)))
;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (w)
(make-worm
(worm-dir w)
(point+ (worm-head w)
(dir-delta
(worm-dir w)))
(drop-last-point
(cons (worm-head w)
(worm-tail w)))))
Dracula v4.2
322:20
DrScheme

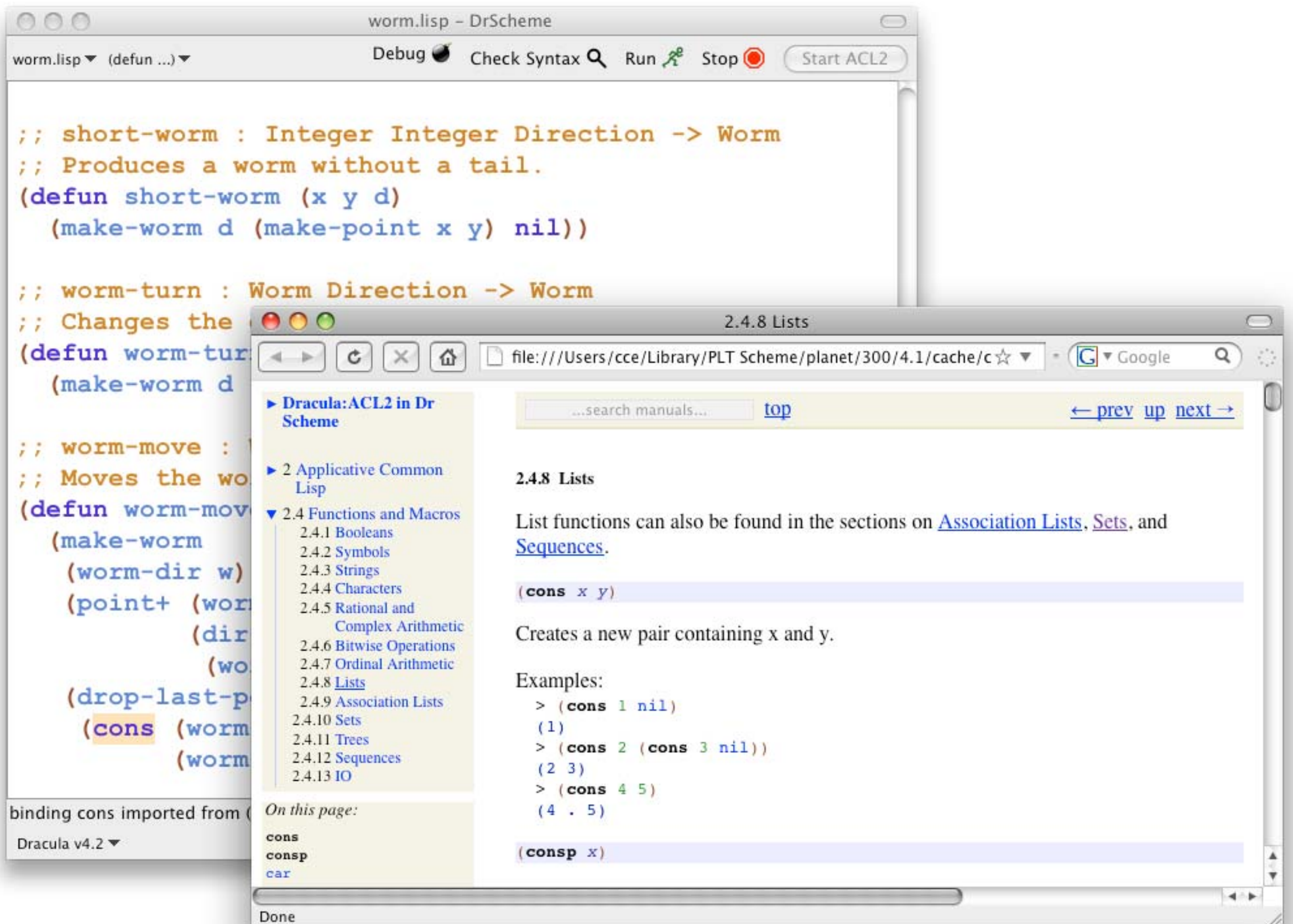
Dracula


worm.lisp - DrScheme
worm.lisp (defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
  (make-worm d (make-point x y) nil))
;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
  (make-worm d (worm-head w) (worm-tail w)))
;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (w)
  (make-worm
   (worm-dir w)
   (point+ (worm-head w)
           (dir-delta
            (worm-dir w)))
   (drop-last-point
    (cons (worm-head w)
          (worm-tail w)))))
Dracula v4.2
314:0

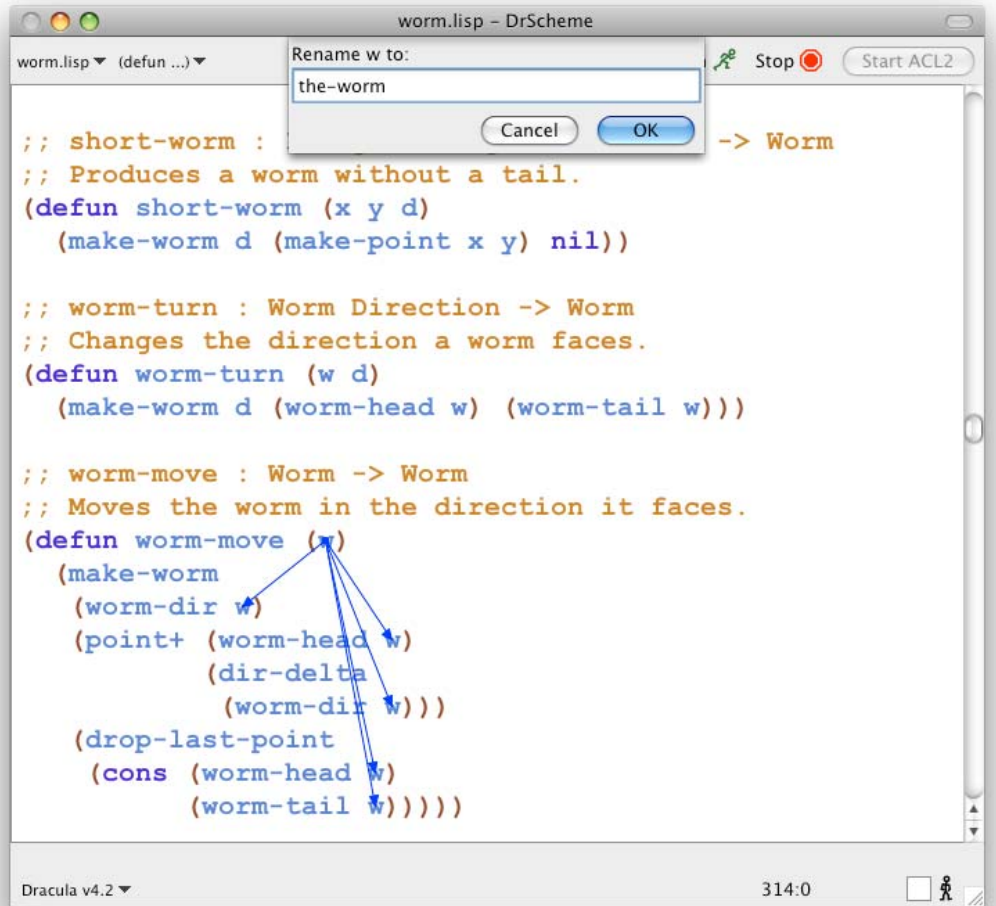
Dracula


worm.lisp - DrScheme
worm.lisp (defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
(make-worm d (make-point x y) nil))
;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
(make-worm d (worm-head w) (worm-tail w)))
;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (w)
(make-worm
(worm-dir w)
(point+ (worm-head w)
(dir-delta
(worm-dir w)))
(drop-last-point
Jump to Definition
View documentation for cons
binding cons imported from (planet "language/dracula.scm" ("cce" "dracula.plt" 4))
Dracula v4.2
435:0

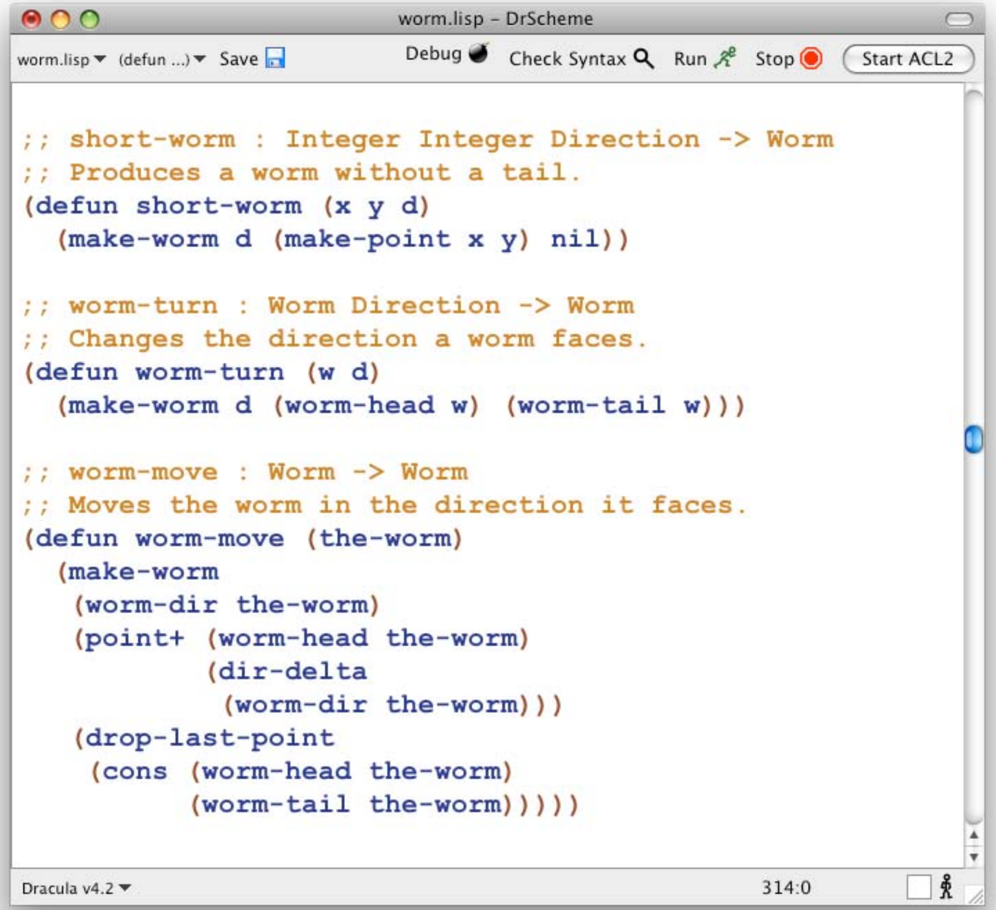
Dracula


worm.lisp - DrScheme
worm.lisp (defun ...)
Debug Check Syntax Run Stop Start ACL2
;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
(make-worm d (make-point x y) nil))
;; worm-turn : Worm Direction -> Worm
binding cons imported from
Dracula v4.2
2.4.8 Lists
file:///Users/cce/Library/PLT Scheme/planet/300/4.1/cache/c
Dracula: ACL2 in Dr Scheme
2 Applicative Common Lisp
2.4 Functions and Macros
2.4.1 Booleans
2.4.2 Symbols
2.4.3 Strings
2.4.4 Characters
2.4.5 Rational and Complex Arithmetic
2.4.6 Bitwise Operations
2.4.7 Ordinal Arithmetic
2.4.8 Lists
2.4.9 Association Lists
2.4.10 Sets
2.4.11 Trees
2.4.12 Sequences
2.4.13 IO
On this page:
cons
consp
car
...search manuals...
top
← prev up next →
2.4.8 Lists
List functions can also be found in the sections on Association Lists, Sets, and Sequences.
(cons x y)
Creates a new pair containing x and y.
Examples:
> (cons 1 nil)
(1)
> (cons 2 (cons 3 nil))
(2 3)
> (cons 4 5)
(4 . 5)
(consp x)
Done

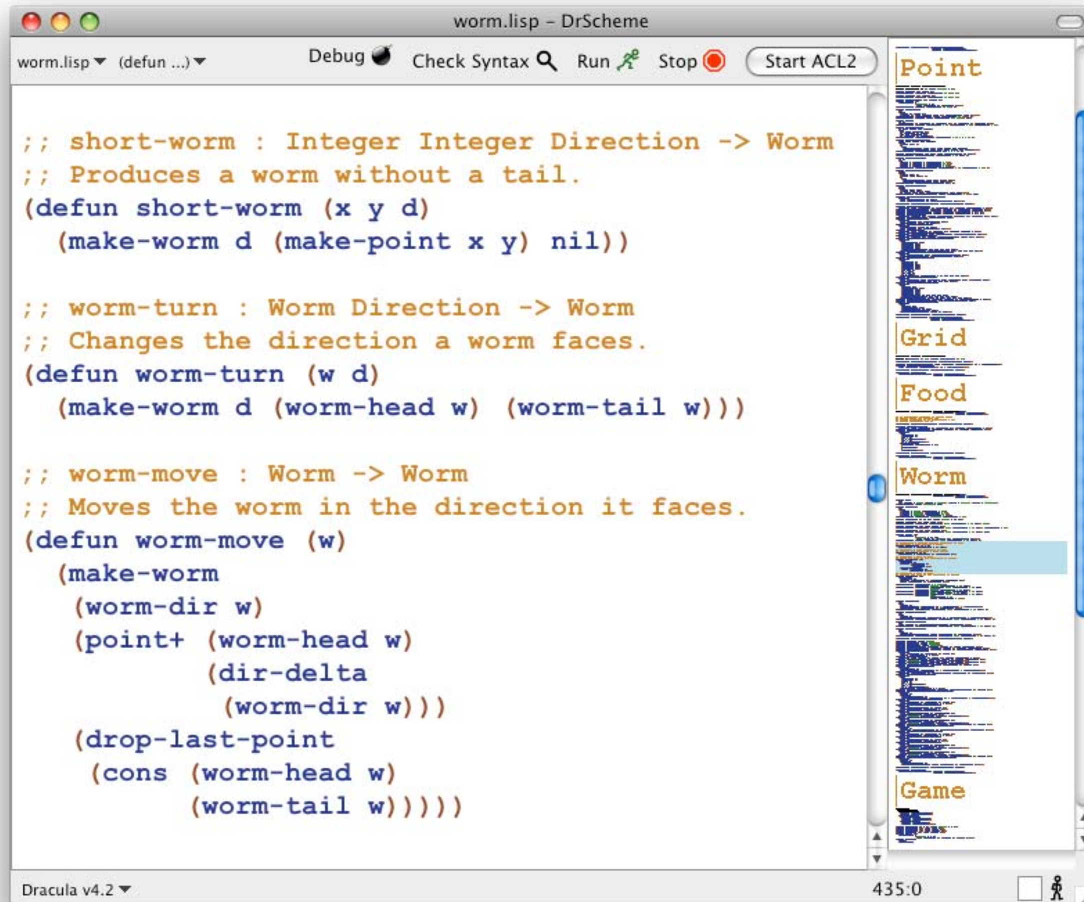
Dracula


worm.lisp - DrScheme
worm.lisp
(defun ...)
Rename w to:
the-worm
Cancel
OK
Stop
Start ACL2
;; short-worm :
-> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
(make-worm d (make-point x y) nil))
;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
(make-worm d (worm-head w) (worm-tail w)))
;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (w)
(make-worm
(worm-dir w)
(point+ (worm-head w)
(dir-delta
(worm-dir w)))
(drop-last-point
(cons (worm-head w)
(worm-tail w)))))
Dracula v4.2
314:0

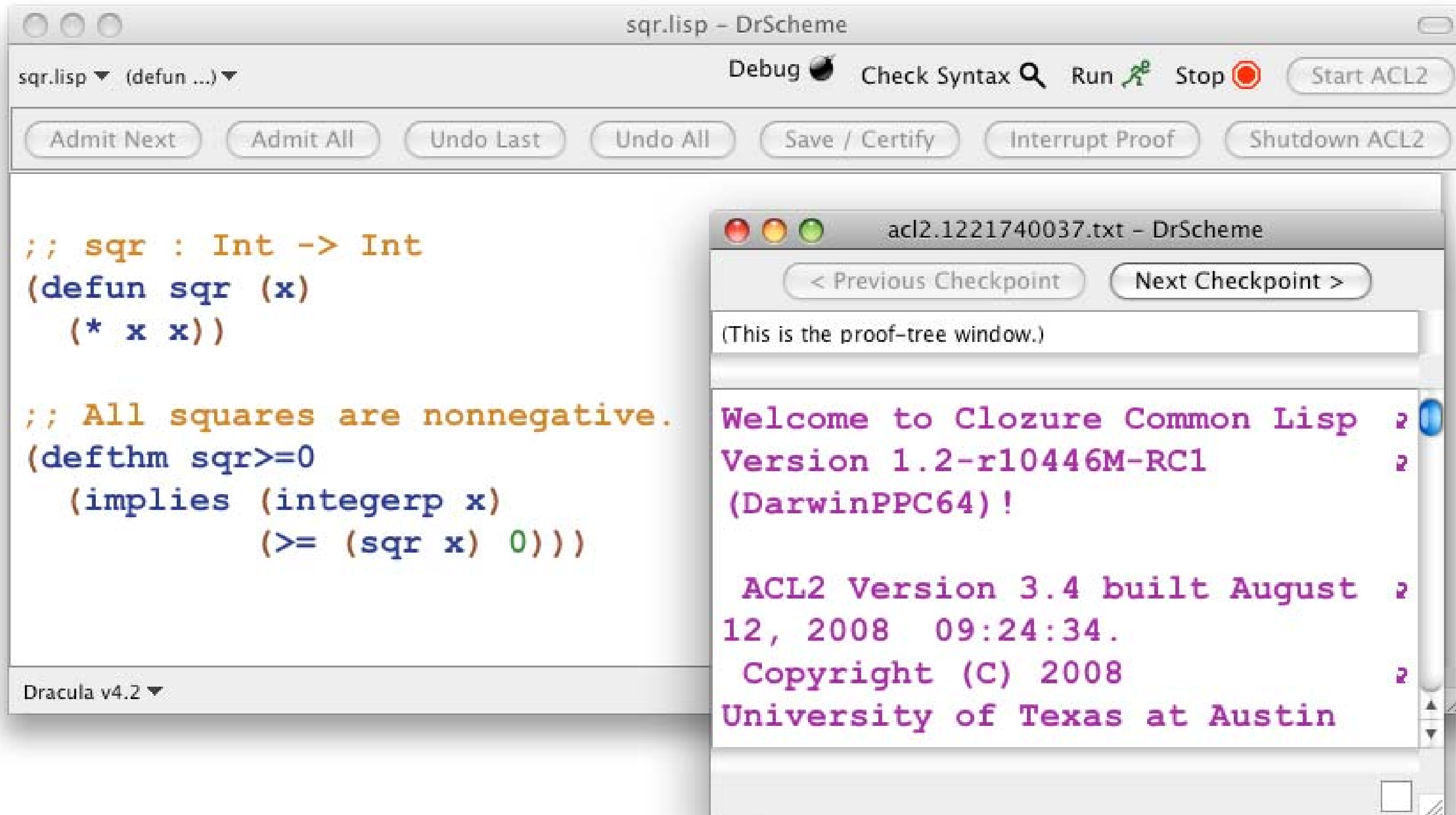
Dracula


worm.lisp - DrScheme
worm.lisp (defun ...) Save
Debug
Check Syntax
Run
Stop
Start ACL2
;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
  (make-worm d (make-point x y) nil))
;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
  (make-worm d (worm-head w) (worm-tail w)))
;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (the-worm)
  (make-worm
   (worm-dir the-worm)
   (point+ (worm-head the-worm)
           (dir-delta
            (worm-dir the-worm)))
   (drop-last-point
    (cons (worm-head the-worm)
          (worm-tail the-worm)))))
Dracula v4.2
314:0

Dracula

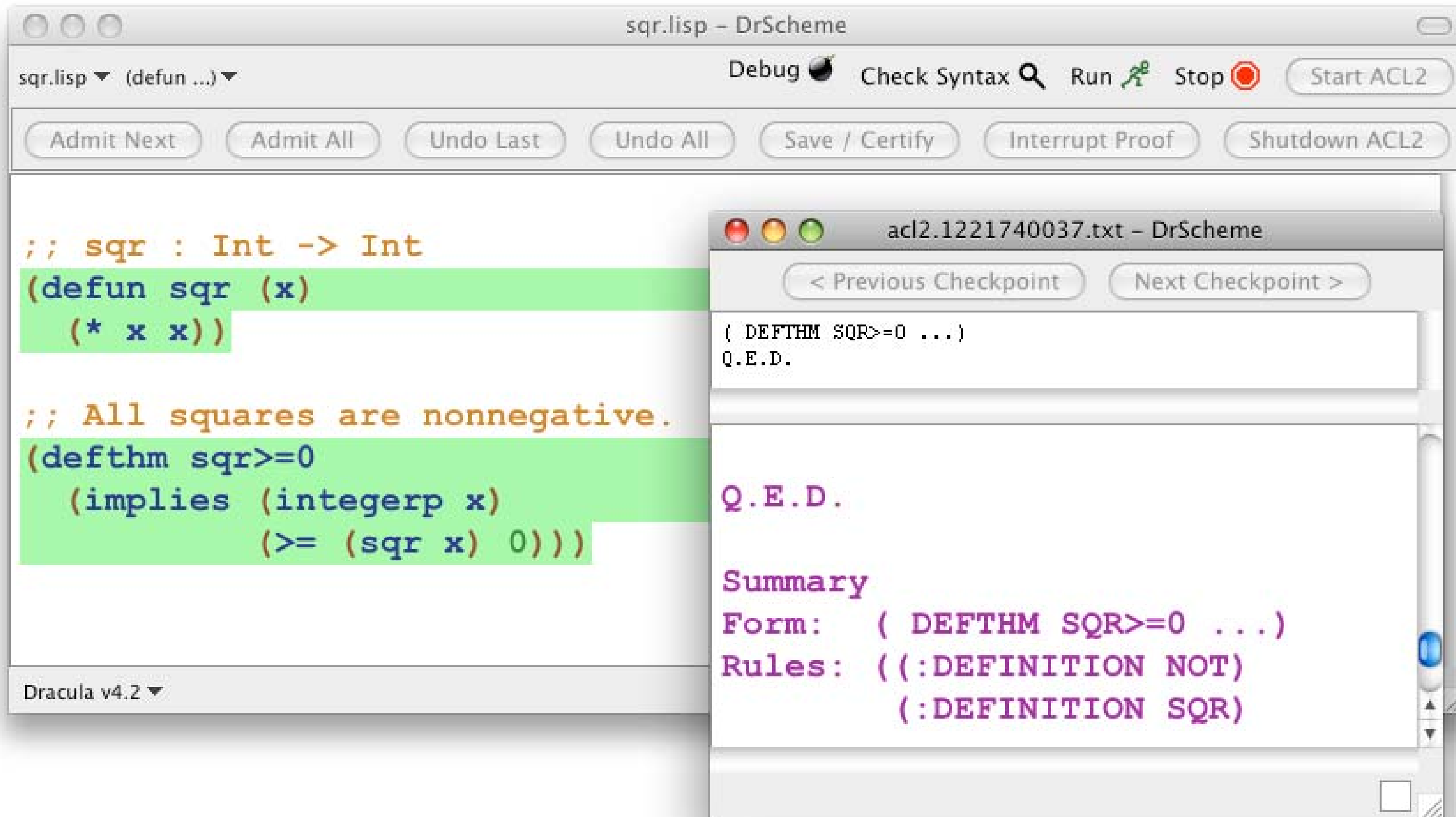

worm.lisp - DrScheme
worm.lisp
(defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
;; short-worm : Integer Integer Direction -> Worm
;; Produces a worm without a tail.
(defun short-worm (x y d)
(make-worm d (make-point x y) nil))
;; worm-turn : Worm Direction -> Worm
;; Changes the direction a worm faces.
(defun worm-turn (w d)
(make-worm d (worm-head w) (worm-tail w)))
;; worm-move : Worm -> Worm
;; Moves the worm in the direction it faces.
(defun worm-move (w)
(make-worm
(worm-dir w)
(point+ (worm-head w)
(dir-delta
(worm-dir w)))
(drop-last-point
(cons (worm-head w)
(worm-tail w)))))
Point
Grid
Food
Worm
Game
Dracula v4.2
435:0

Dracula



sqr.lisp – DrScheme
sqr.lisp (defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
Admit Next
Admit All
Undo Last
Undo All
Save / Certify
Interrupt Proof
Shutdown ACL2
;; sqr : Int -> Int
(defun sqr (x)
(* x x))
;; All squares are nonnegative.
(defthm sqr>=0
(implies (integerp x)
(>= (sqr x) 0)))
Dracula v4.2
acl2.1221740037.txt – DrScheme
< Previous Checkpoint
Next Checkpoint >
(This is the proof-tree window.)
Welcome to Clozure Common Lisp Version 1.2-r10446M-RC1 (DarwinPPC64)!
ACL2 Version 3.4 built August 12, 2008 09:24:34.
Copyright (C) 2008 University of Texas at Austin

Dracula


sqr.lisp – DrScheme
sqr.lisp (defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
Admit Next
Admit All
Undo Last
Undo All
Save / Certify
Interrupt Proof
Shutdown ACL2
;; sqr : Int -> Int
(defun sqr (x)
(* x x))
;; All squares are nonnegative.
(defthm sqr>=0
(implies (integerp x)
(>= (sqr x) 0)))
Dracula v4.2
acl2.1221740037.txt – DrScheme
< Previous Checkpoint
Next Checkpoint >
(DEFTHM SQR>=0 ...)
Q.E.D.
Q.E.D.
Summary
Form: (DEFTHM SQR>=0 ...)
Rules: ((:DEFINITION NOT)
(:DEFINITION SQR)

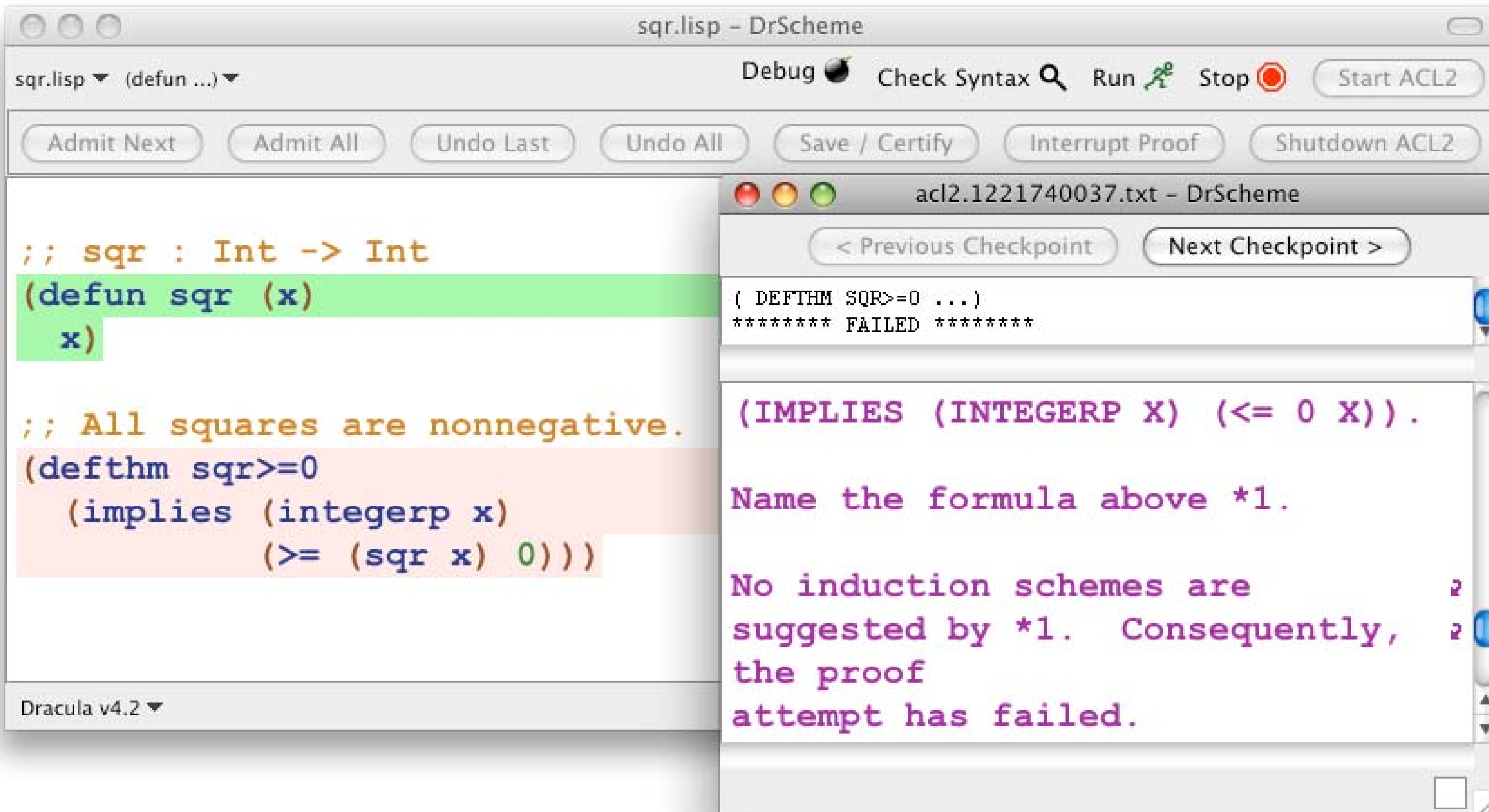
Dracula

```
;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))
```

Dracula

```
;; sqr : Int -> Int
(defun sqr (x)
  x)

;; All squares are nonnegative.
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))
```

Dracula



sqr.lisp – DrScheme
sqr.lisp
(defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
Admit Next
Admit All
Undo Last
Undo All
Save / Certify
Interrupt Proof
Shutdown ACL2
;; sqr : Int -> Int
(defun sqr (x)
x)
;; All squares are nonnegative.
(defthm sqr>=0
(implies (integerp x)
(>= (sqr x) 0)))
Dracula v4.2
acl2.1221740037.txt – DrScheme
< Previous Checkpoint
Next Checkpoint >
(DEFTHM SQR>=0 ...)
******** FAILED ********
(IMPLIES (INTEGERP X) (<= 0 X)).
Name the formula above *1.
No induction schemes are
suggested by *1. Consequently,
the proof
attempt has failed.

Program Design

- How to Design Programs code:

```
;; sqr : Int -> Int
(define (sqr x)
  (* x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
```

Program Design

- Dracula code:

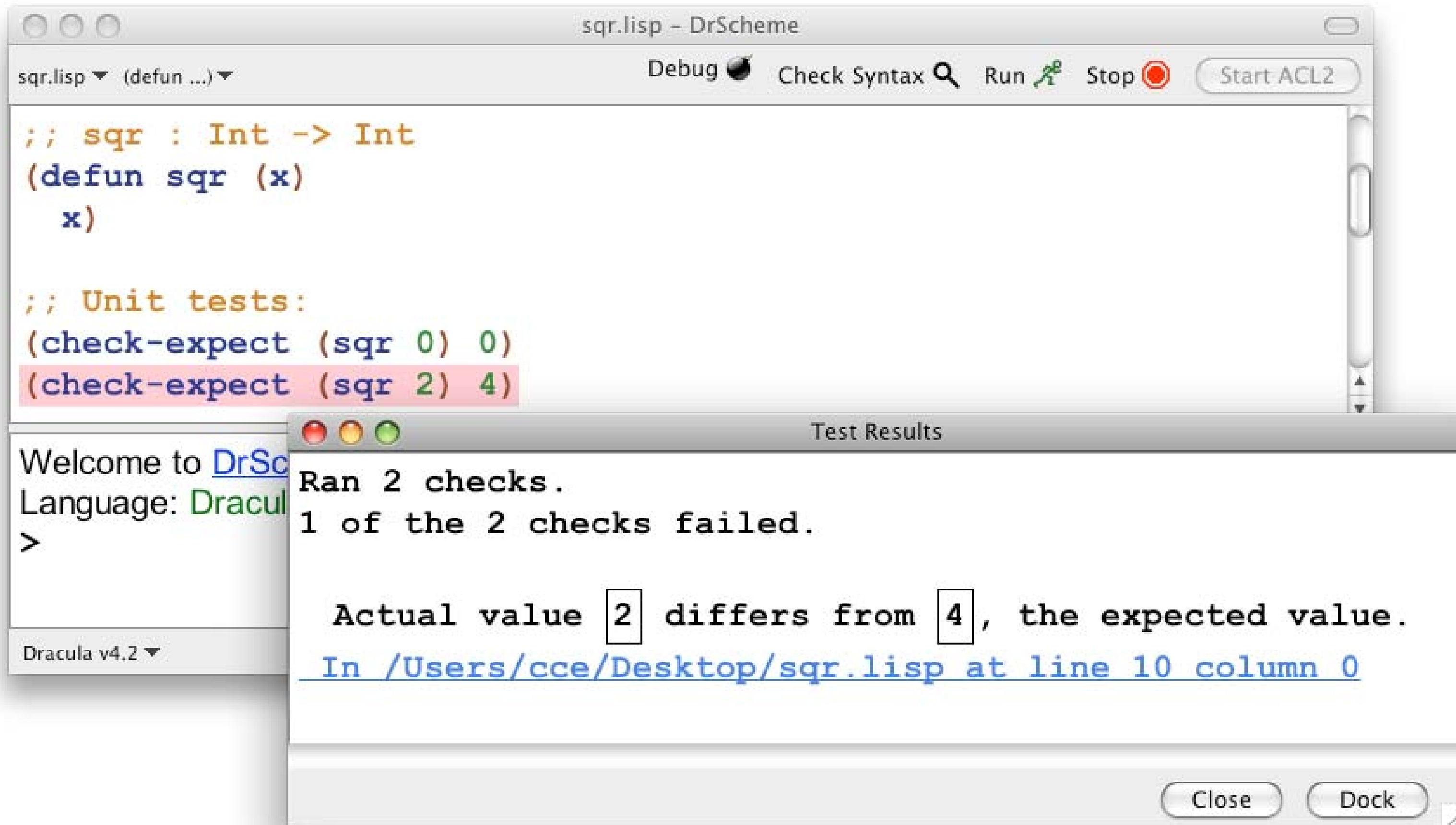
```
;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
```

Unit Tests

- Dracula code:

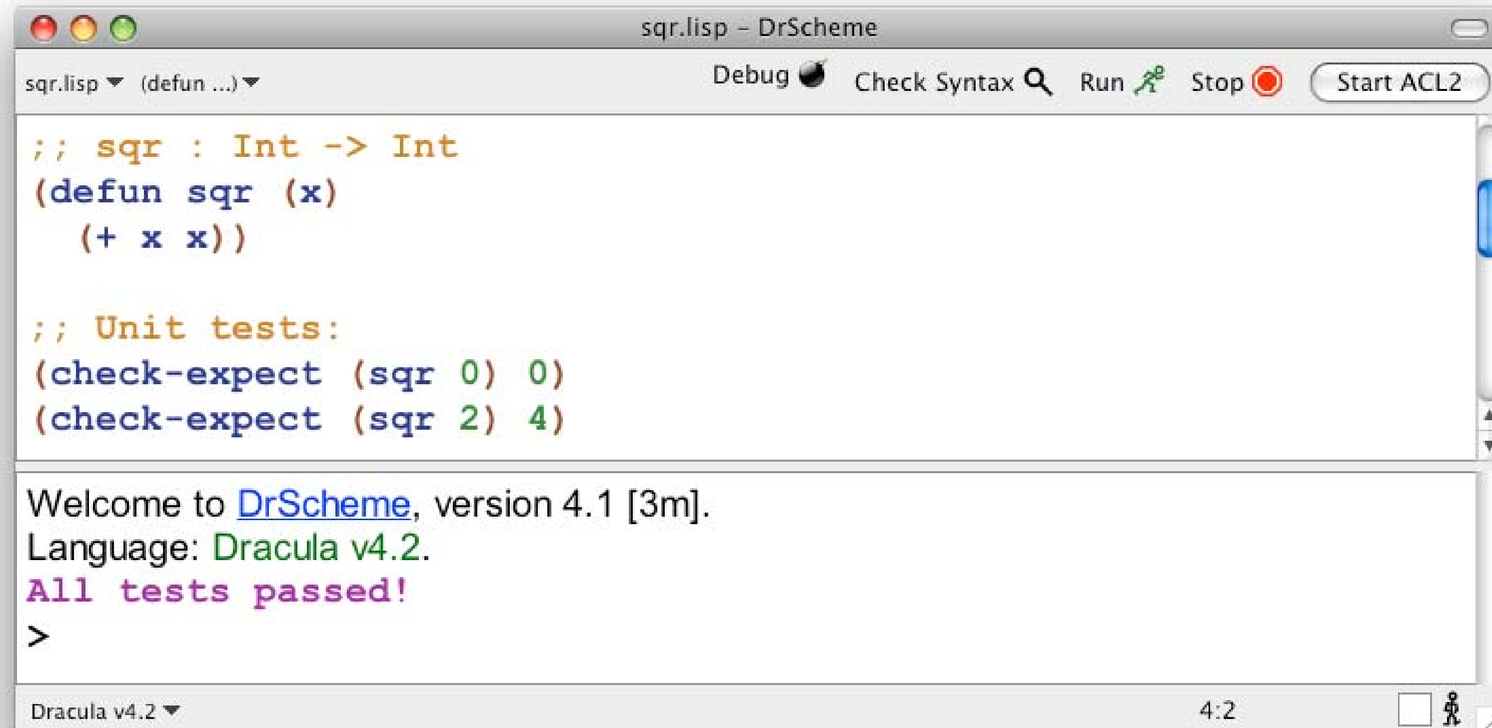
```
;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;; Unit tests:     (==> assert-event)
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
```

Unit Tests



sqr.lisp - DrScheme
sqr.lisp (defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
;; sqr : Int -> Int
(defun sqr (x)
x)
;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
Welcome to DrSc
Language: Dracul
>
Dracula v4.2
Test Results
Ran 2 checks.
1 of the 2 checks failed.
Actual value 2 differs from 4, the expected value.
In /Users/cce/Desktop/sqr.lisp at line 10 column 0
Close
Dock

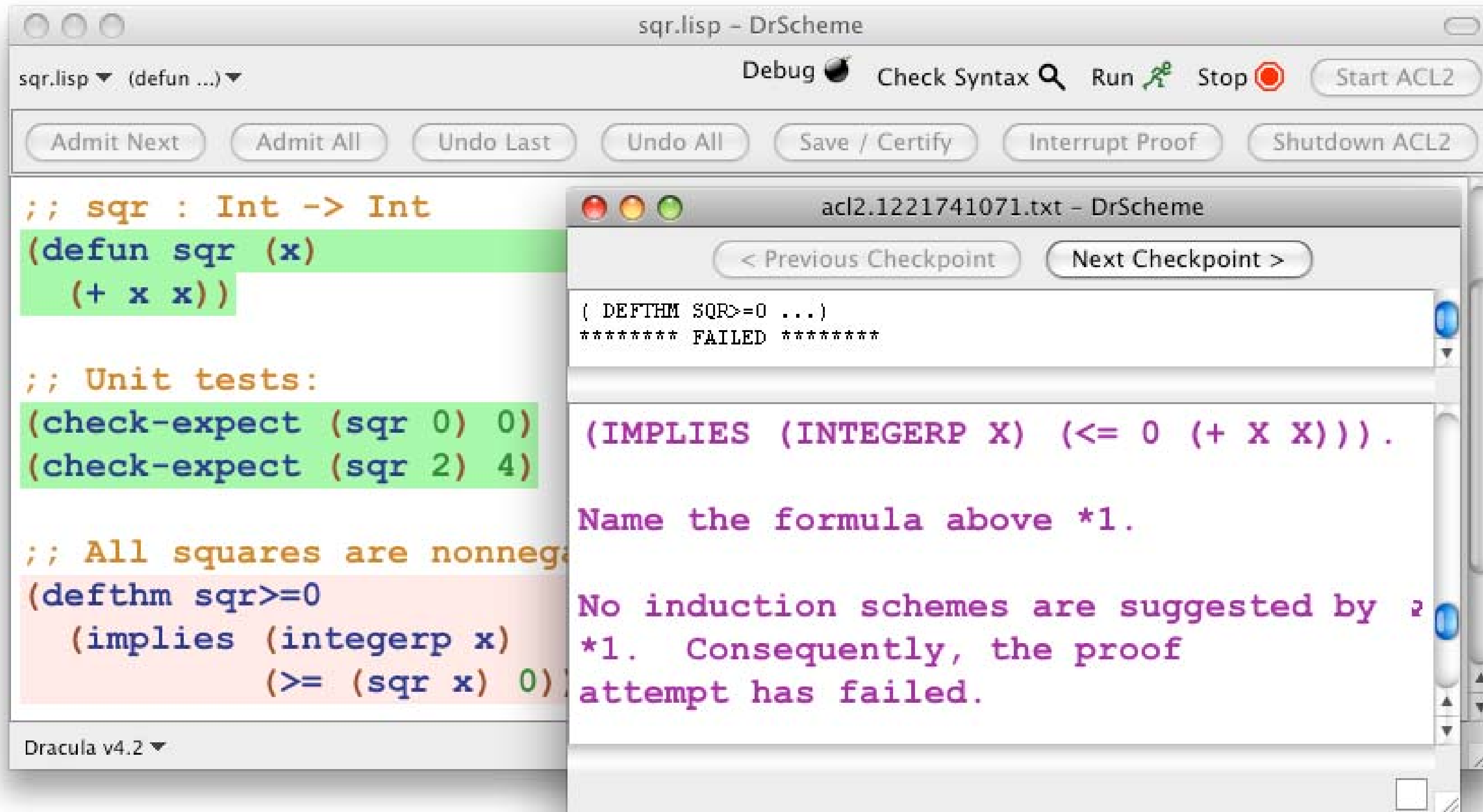
Unit Tests


sqr.lisp - DrScheme
sqr.lisp
(defun ...)
Debug
Check Syntax
Run
Stop
Start ACL2
;; sqr : Int -> Int
(defun sqr (x)
  (+ x x))
;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
Welcome to DrScheme, version 4.1 [3m].
Language: Dracula v4.2.
All tests passed!
>
Dracula v4.2
4:2

Beyond Unit Tests

```
;; sqr : Int -> Int
(defun sqr (x)
  (+ x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
```

Beyond Unit Tests



DoubleCheck

```
;; ACL2 theorem:
(defthm name
  (implies (and precondition ...)
                postcondition)))

;; DoubleCheck property:
(defproperty name
  (x [:where precondition]
     [:value distribution] ...)
  postcondition)
```

DoubleCheck

```
;; ACL2 theorem:
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))

;; DoubleCheck property:
(defproperty sqr>=0
  (x)
  (implies (integerp x)
           (>= (sqr x) 0)))
```

DoubleCheck

```
;; ACL2 theorem:
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))

;; DoubleCheck property:
(defproperty sqr>=0
  (x :where (integerp x))
  (>= (sqr x) 0))
```

DoubleCheck

```
;; ACL2 theorem:
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))

;; DoubleCheck property:
(defproperty sqr>=0
  (x :where (integerp x)
     :value (random-integer))
  (>= (sqr x) 0))
```

DoubleCheck

```
;; Simple distributions:
(random-string)
(random-integer)

;; Parameterized distributions:
(random-between low high)
(random-list-of dist [:size size])

;; Write new distributions:
(defrandom name (arg ...) expr)
```

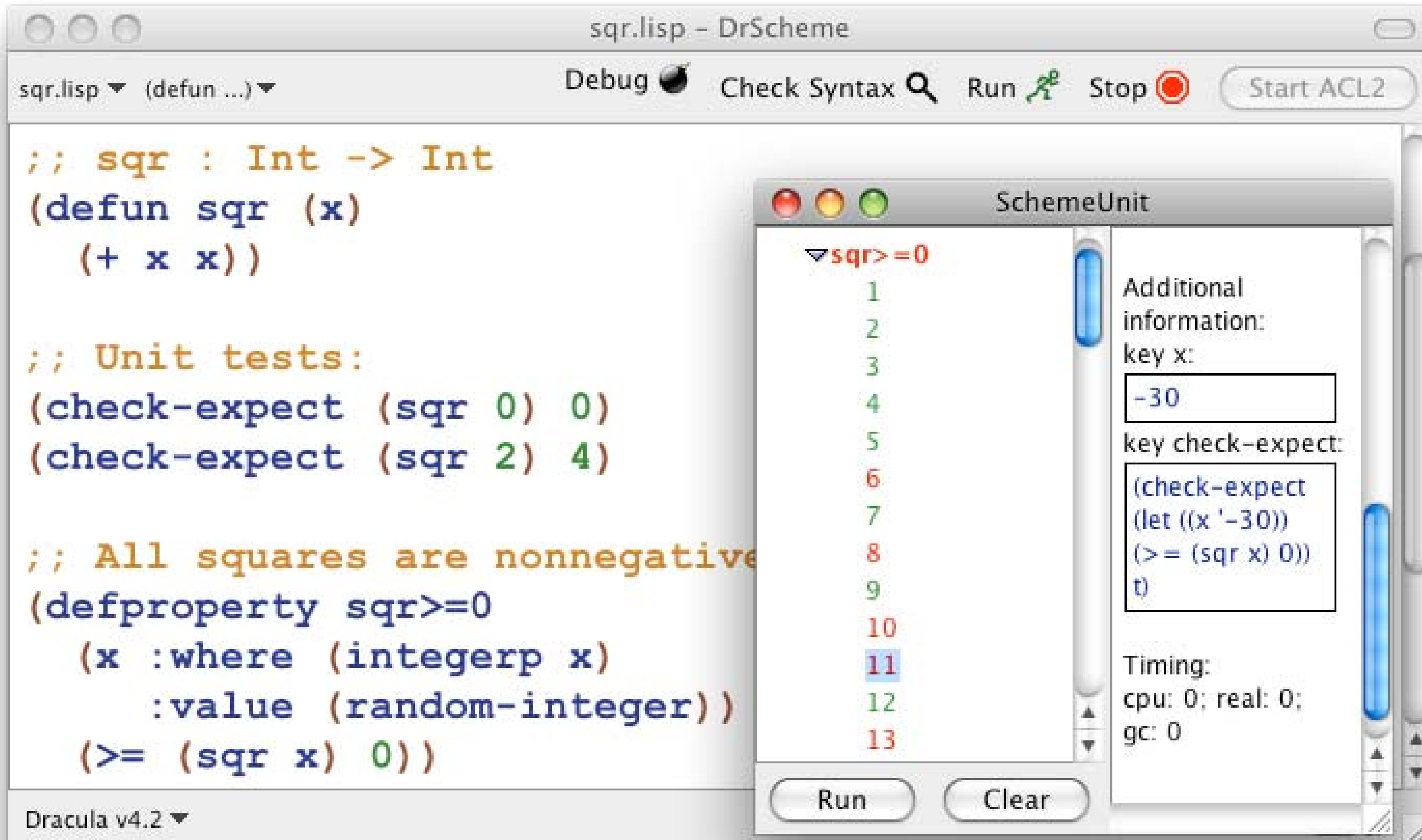
DoubleCheck

```
;; ACL2 theorem:
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))

;; DoubleCheck property:(==> defthm)
(defproperty sqr>=0
  (x :where (integerp x)
     :value (random-integer))
  (>= (sqr x) 0))
```

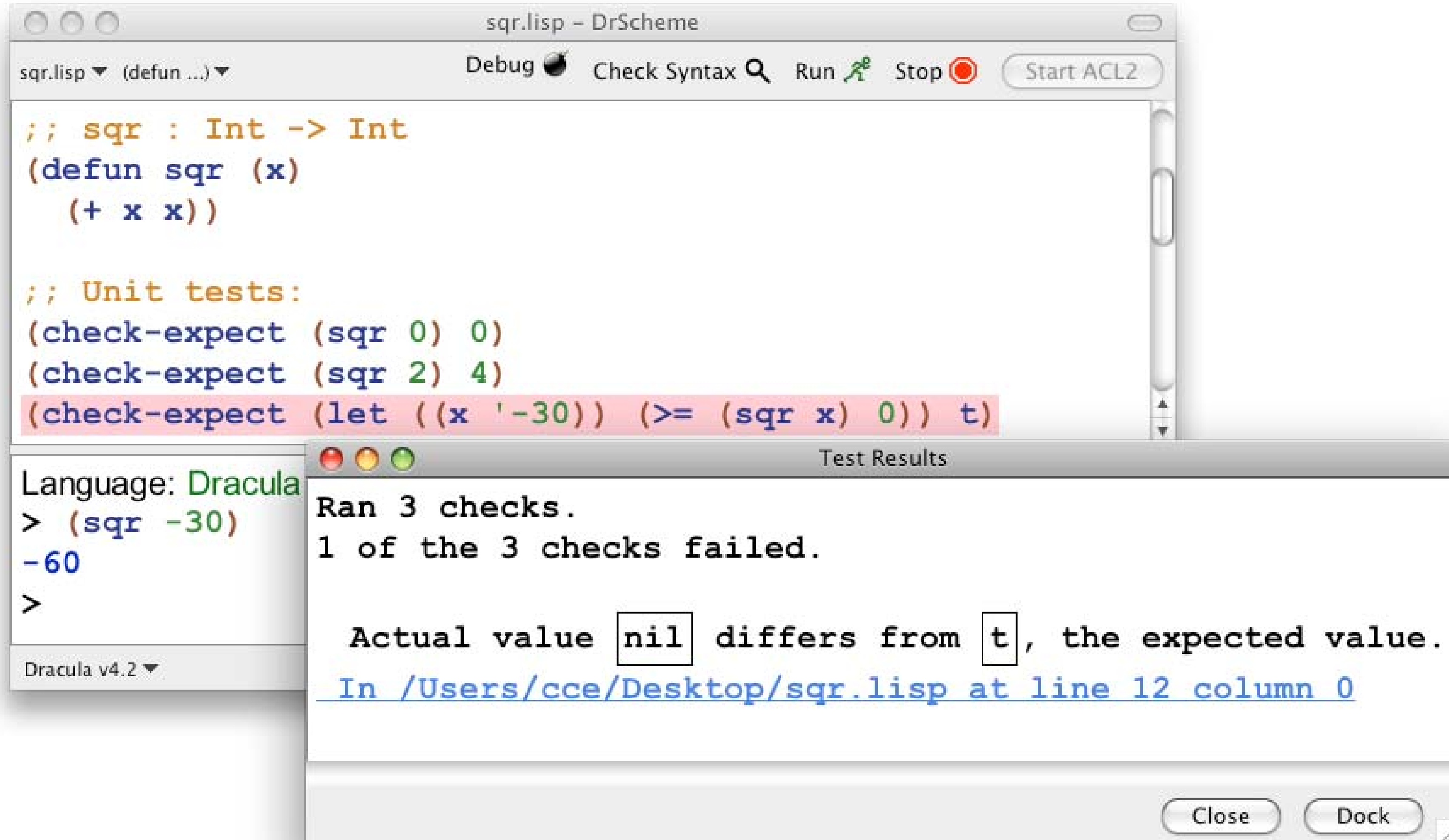
DoubleCheck

```
;; ACL2 theorem:
(defthm sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))

;; Ideal syntax (future work):
(defproperty sqr>=0
  (implies (integerp x)
           (>= (sqr x) 0)))
```

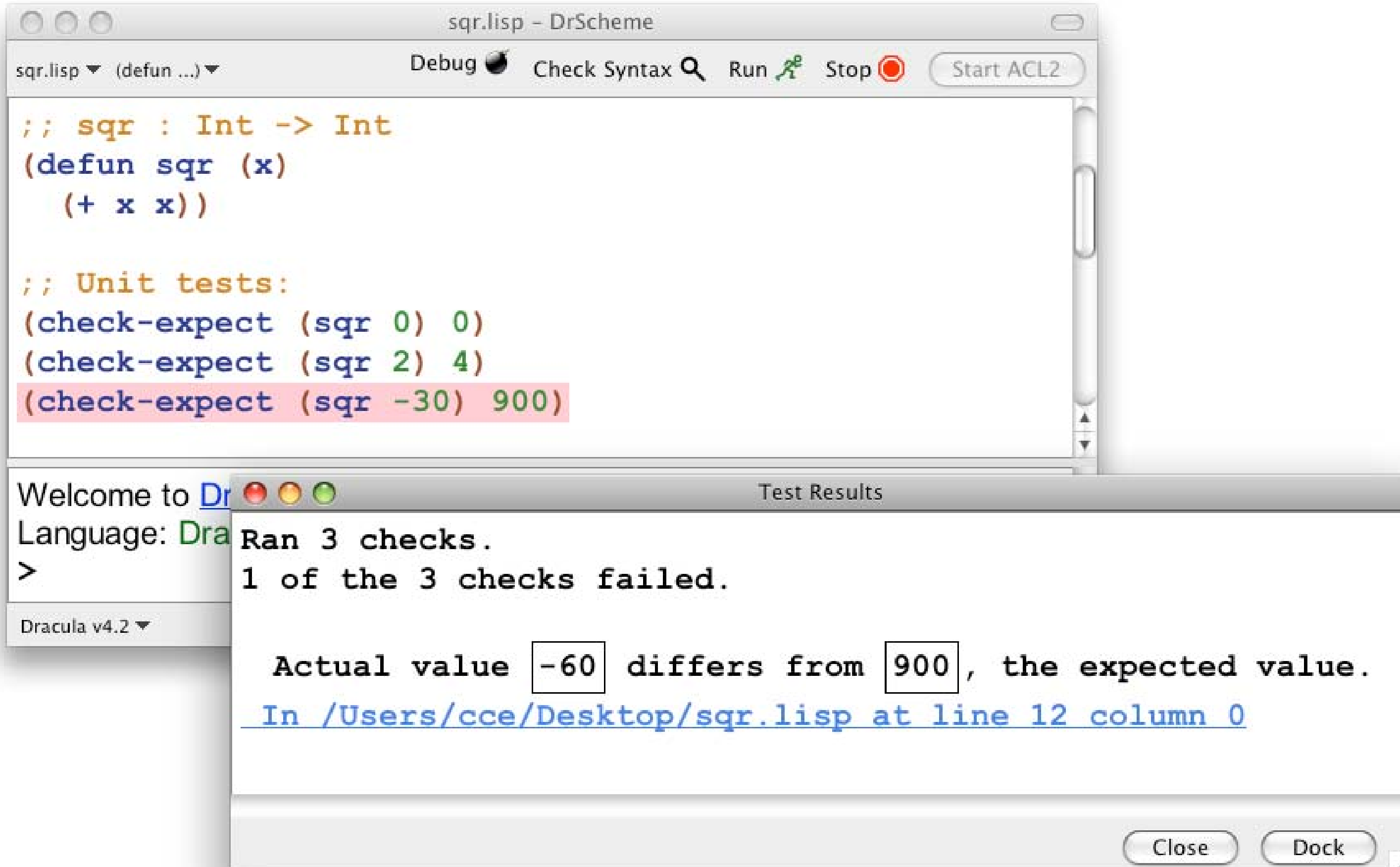
DoubleCheck



DoubleCheck



DoubleCheck



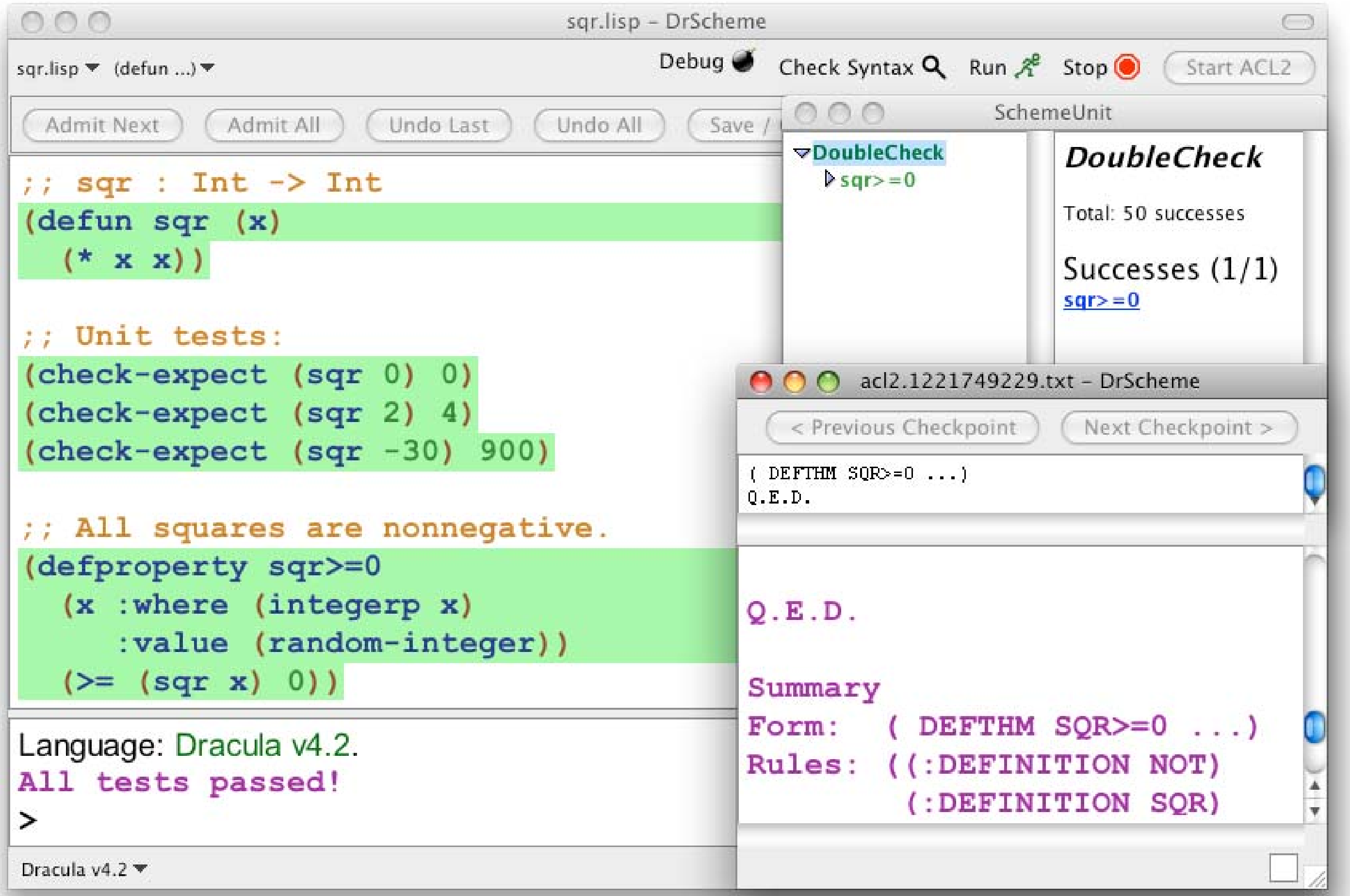
DoubleCheck

```
;; sqr : Int -> Int
(defun sqr (x)
  (+ x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
(check-expect (sqr -30) 900)
```

DoubleCheck

```
;; sqr : Int -> Int
(defun sqr (x)
  (* x x))

;; Unit tests:
(check-expect (sqr 0) 0)
(check-expect (sqr 2) 4)
(check-expect (sqr -30) 900)
```

DoubleCheck



Software Engineering Courses at OU

- SE-i
  - Process (30%) - Humphrey PSP
  - Design (35%) - FP in ACL2
  - Testing/Validation (35%)
    - Predicate-based, automated testing (DblChk)
    - Mechanized logic for full verification (ACL2)
  - Software development projects
    - 6 individual projects: Design/Code/PSP rpt
      - Early projects: small components
      - Later projects: applications using components
    - 2 team projects
      - Building on components and applications
      - Seven deliverables in all

Software Engineering Courses at OU

- SE-i
  - Process (30%) - Humphrey PSP
  - Design (35%) - FP in ACL2
  - Testing/Validation (35%)
    - Predicate-based, automated testing (DblChk)
    - Mechanized logic for full verification (ACL2)
  - Software development projects
    - 6 individual projects: Design/Code/PSP rpt
      - Early projects: small components
      - Later projects: applications using components
    - 2 team projects
      - Building on components and applications
      - Seven deliverables in all

60%

30%

10% other

Software Engineering Courses at OU

- SE-ii
  - Organized around one sfw devp project
  - Team project (4 - 6 students per team)
  - Project size
    - 3,000 - 5,000 lines of code, before ACL2
    - 2,000 - 3,000 lines of code, since intro of ACL2
  - 12 separate (team) deliverables
    - Engineering std, design/schedule, code, installation/usage doc, defect history, tests/theorems, meeting log, …
    - 3 presentations - last to Advisory Board
  - Individual journals — expanded PSP rpt

Background of SE Students

- Standard CS curriculum
  - ABET, math heavy
- No significant FP experience
  - Minor exposure in PL course
- Serious logic course (70% of students)
  - Reasoning about hdw/sfw properties
- So, SE is first serious exposure to FP
  - Almost all succeed in
    - Learning FP
    - Predicate-based testing
  - Success with ACL2 mechanized logic
    - Most acquire a reasonable level of comfort
    - 10% to 20% gain proficiency with ACL2 logic

Example SE-i Project

- Linear encode/decode
  - Message: $x_0\ x_1\ \ldots\ x_{n-2}\ x_{n-1}$, $0 \le x_k < m$
  - Encoding: … $(x_k + x_{k+1}) \bmod m$ …, where x_n=m-1
- Define encode, decode, and predicates
  - encode, decode, code-list?
- Define correctness properties
  - k^{th} element of encoded list is $(x_k + x_{k+1}) \bmod m$
  - decode inverts encode

Example SE-i Project

- Linear encode/decode
  - Message: $x_0\ x_1\ \ldots\ x_{n-2}\ x_{n-1}$, $0 \leq x_k < m$
  - Encoding: … $(x_k + x_{k+1}) \bmod m$ …, where x_n=m-1
- Define correctness properties
  - decode inverts encode
- Inversion property

```
(defproperty decode-inverts-encode
  (m  :value (random-between 2 100)
  (xs :value (random-list-of
               (random-between 0 (- m 1))))
  (equal (decode m (encode m xs)) xs)))
```

Example SE-i Project

- Linear encode/decode
  - Message: $x_0\ x_1\ \ldots\ x_{n-2}\ x_{n-1}$, $0 \leq x_k < m$
  - Encoding: … $(x_k + x_{k+1})$ mod m …, where x_n=m-1
- Define correctness properties
  - decode inverts encode
- **Inversion property as (untrue) theorem**

```
(defthm decode-inverts-encode-thm
   (equal (decode m (encode m xs)) xs)))
```

Example SE-i Project

- Linear encode/decode
  - Message: $x_0\ x_1 \ldots x_{n-2}\ x_{n-1}$, $0 \le x_k < m$
  - Encoding: … $(x_k + x_{k+1}) \bmod m$ …, where x_n=m-1
- Define correctness properties
  - decode inverts encode
- Inversion property with preconditions

```
(defproperty decode-inverts-encode
  (m  :where (and (integerp m) (>= m 2))
      :value (random-between 2 100)
   xs :where (code-list? m xs)
      :value (random-list-of
                (random-between 0 (- m 1))))
  (equal (decode m (encode m xs)) xs)))
```

Example SE-i Project

- Linear encode/decode
  - Message: $x_0\ x_1\ \ldots\ x_{n-2}\ x_{n-1}$, $0 \le x_k < m$
  - Encoding: … $(x_k + x_{k+1})$ mod m …, where x_n=m-1
- Define correctness properties
  - decode inverts encode
- Inversion property as theorem

```
(defthm decode-inverts-encode-thm
  (implies (and (integerp m)
                (>= m 2)
                (code-list? m xs))
           (equal (decode m (encode m xs))
                  xs)))
```

Example SE-i Project

- Linear encode/decode
  - Message: $x_0\ x_1\ \ldots\ x_{n-2}\ x_{n-1}$, $0 \leq x_k < m$
  - Encoding: … $(x_k + x_{k+1})$ mod m …, where x_n=m-1
- Define correctness properties
  - k^{th} element of encoded list is $(x_k + x_{k+1})$ mod m
- **Right-stuff property as (untrue) theorem**

```
(defthm encoded-elements-are-correct-thm
  (implies (and (integerp m) (>= m 2)
                (code-list? m xs)
                (integerp k))
           (= (nth k (encode m xs))
              (mod (+ (nth k xs)
                      (nth (+ k 1) xs))
                   m))))
```

Example SE-i Project

- Linear encode/decode
  - Message: $x_0\ x_1\ \ldots\ x_{n-2}\ x_{n-1}$, $0 \le x_k < m$
  - Encoding: … $(x_k + x_{k+1})$ mod m …, where x_n=m-1
- Define correctness properties
  - k^{th} element of encoded list is $(x_k + x_{k+1})$ mod m
- Right-stuff property as theorem

```
(defthm encoded-elements-are-correct-thm
  (implies (and (integerp m) (>= m 2)
                (code-list? m xs)
                (natp k)(< k (- (len xs) 1))
           (= (nth k (encode m xs))
              (mod (+ (nth k xs)
                      (nth (+ k 1) xs))
                   m))))
```

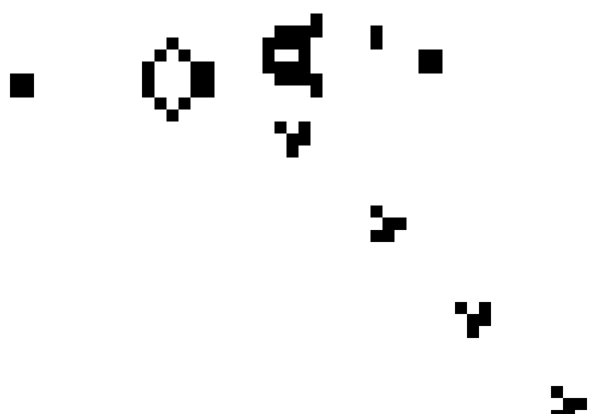
Example SE-i Project

- Linear encode/decode
  - Message: $x_0\ x_1\ \ldots\ x_{n-2}\ x_{n-1}$, $0 \le x_k < m$
  - Encoding: … $(x_k + x_{k+1}) \bmod m$ …, where $x_n = m-1$
- Define correctness properties
  - k^{th} element of encoded list is $(x_k + x_{k+1}) \bmod m$
- **Right-stuff property as vacuous theorem**

```
(defthm encoded-elements-are-correct-thm
  (implies (and (integerp m) (>= m 2)
                (code-list? m xs)
                (<= k 0)(> k (len xs)))
           (= (nth k (encode m xs))
              (mod (+ (nth k xs)
                      (nth (+ k 1) xs))
                   m))))
```

Team Project Example from SE-ii

- Conway game of life (cellular automaton)
  - Multiple topologies - sphere, cylinder, torus, Klein
  - Six solutions, 1200 - 7000 lines of code, avg: 3000
  - 7000-line implementation included
    - Three-dimensional rendering
    - Over 100 properties verified by ACL2 mechanized logic
    - Ten properties on 3D-rending (eg, no bit-plane errors)



Gosper glider gun

Reactions to SE Courses

- Students
  - PSP unpopular (time & defect logs, plans…)
  - Functional programming
    - Almost all get it, eventually
    - 10% complain
    - 10% - 20% really like it
    - The rest take it as an interesting challenge
  - Property-based testing
    - Just started this semester
    - Students seem to like it
    - Smoothes the way towards theorems
  - Theorems
    - Top quarter like it, bottom quarter gets lost
- Advisory board (from computing industry)
  - Positive comments nearly universal

Outreach

- Three-day workshop, May 2008
  - Participants: 13 CS instructors from 6 states
  - Lectures (35%) plus hands-on projects (65%)
  - Two leaders, plus two aids with ACL2 expertise
- Lessons learned
  - Theorems are easier than automated testing
    - Appropriate random distributions add complication
  - Specifying properties requires careful thought
    - Incorrect or vacuous theorems— common first attempts
    - Payoff— better understanding of software
  - Projects must be carefully constructed
    - Ensure reasonable solutions (solve them in advance)
- MEPLS semiannual meeting

Outreach

- Three-day workshop, May 2008
  - Participants: 13 CS instructors from 6 states
  - Lectures (35%) plus hands-on projects (65%)
  - Two leaders, plus two aids with ACL2 expertise
- Lessons learned
  - Theorems are easier than automated testing
    - Appropriate random distributions add complication
  - Specifying properties requires careful thought
    - Incorrect or vacuous theorems— common first attempts
    - Payoff— better understanding of software
  - Projects must be carefully constructed
    - Ensure reasonable solutions (solve them in advance)
- MEPLS semiannual meeting
- Google **Dracula DrScheme, Rex SEcollab, MEPLS**

Plans for Future

- Integrated testing / verification
- Dracula module facility
- Coordinated projects (on website)
  - Building from components to applications
  - Four tracks, 4 - 6 projects in each track
- Outreach workshops
  - SIGCSE tutorial
  - Three-day workshops

The End