Logic and Computation – CS 2800 Fall 2019

Lecture 6

ACL2s continued

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

rl: rotate list to the left

```
;; rl: TL x Nat -> TL
;; Given a list, l, and a natural number, n,
;; rl rotates the list to the left n times
```

```
(check= (rl (list 1 2 3) 1) (list 2 3 1))
(check= (rl (list 1 2 3) 2) (list 3 1 2))
(check= (rl (list 1 2 3) 3) (list 1 2 3))
```

rl: first attempt

```
;; rl: TL x Nat -> TL
;; Given a list, l, and a natural number, n,
;; rl rotates the list to the left n times
```

What if n=0?

```
(definec rl (l :tl n :nat) :tl
  (if (endp l)
      nil
      (rl (app (rest l) (list (first l))) (- n 1))))
```

```
(check= (rl (list 1 2 3) 1) (list 2 3 1))
(check= (rl (list 1 2 3) 2) (list 3 1 2))
(check= (rl (list 1 2 3) 3) (list 1 2 3))
```

rl: second attempt

```
;; rl: TL x Nat -> TL
;; Given a list, l, and a natural number, n,
;; rl rotates the list to the left n times
```

```
(definec rl (l :tl n :nat) :tl
  (if (equal n 0)
      l
      (rl (app (rest l) (list (first l))) (- n 1))))
```

All 3 tests pass. Seems OK…
But what if n>0 and l=nil?

```
(check= (rl (list 1 2 3) 1) (list 2 3 1))
(check= (rl (list 1 2 3) 2) (list 3 1 2))
(check= (rl (list 1 2 3) 3) (list 1 2 3))
```

rl: second attempt

```
;; rl: TL x Nat -> TL
;; Given a list, l, and a natural number, n,
;; rl rotates the list to the left n times
```

More tests!

```
(definec rl (l :tl n :nat) :tl
  (if (equal n 0)
      l
      (rl (app (rest l) (list (first l))) (- n 1))))
```

```
(check= (rl () 3) ())
(check= (rl (list 1 2 3) 0) (list 1 2 3))
(check= (rl (list 1 2 3) 1) (list 2 3 1))
(check= (rl (list 1 2 3) 2) (list 3 1 2))
(check= (rl (list 1 2 3) 3) (list 1 2 3))
```

← Test fails

rl: third attempt

```
;; rl: TL x Nat -> TL
;; Given a list, l, and a natural number, n,
;; rl rotates the list to the left n times

(definec rl (l :tl n :nat) :tl
  (cond
     ((equal n 0) l)
     ((endp l) nil)
     (t (rl (app (rest l) (list (first l))) (- n 1)))))


(check= (rl () 3) ())
(check= (rl (list 1 2 3) 0) (list 1 2 3))
(check= (rl (list 1 2 3) 1) (list 2 3 1))
(check= (rl (list 1 2 3) 2) (list 3 1 2))
(check= (rl (list 1 2 3) 3) (list 1 2 3))
```

ACL2s demo

- Run VM/eclipse
- Create new project
- Download homework 1 file from web site
- Copy file into new project; refresh; open file
- ACL2s mode
- Split window
- Evaluate simple expressions
- Define function rl
- See file 06-acl2s.lisp

Quizzes on `endp`

```
(defunc endp (l)
  :input-contract (listp l)
  :output-contract (booleanp (endp l))
  (not (consp l)))
```

- Quiz: is endp a recognizer?
  A. Yes
  B. No

Quizzes on `endp`

```
(defunc endp (l)
  :input-contract (listp l)
  :output-contract (booleanp (endp l))
  (not (consp l)))
```

- Quiz: can we change the input contract of endp into `t` (true)?
  A. Yes
  B. No

Quizzes on `endp`

```
(defunc endp (l)
  :input-contract (listp l)
  :output-contract (booleanp (endp l))
  (not (consp l)))
```

- Quiz: is endp a recognizer?
  - A. Yes
  - B. No

Its input contract is not t

- Quiz: can we change the input contract of endp into `t` (true)?
  - A. Yes
  - B. No

```
(defunc atom (l)
  :input-contract t
  :output-contract (booleanp (atom l))
  (not (consp l)))
```

Same body, different contracts

```
(defunc endp (l)
  :input-contract (listp l)
  :output-contract
      (booleanp (endp l))
  (not (consp l)))
```

```
(defunc atom (l)
  :input-contract t
  :output-contract
      (booleanp (atom l))
  (not (consp l)))
```

- Why would we want two different functions with identical bodies and output contracts, but different input contracts?
  - Use endp for lists; if we make an error, ACL2s will find it for us.

Outline

- Quote
- Let

Quote

Semantics of quote: $[\![' x]\!] = ([\![(\text{quote } x)]\!] = x$

Examples:

- $[\![' 1]\!] = 1$
- $[\![' \text{t}]\!] = \text{t}$
- $[\![' \text{nil}]\!] = \text{nil}$
- $[\![' ()]\!] = ()$ = nil
- $[\!['' ()]\!] =$ '() = 'nil
- $[\![' (1\ 2)]\!] = (1\ 2)$
- $[\!['' (1\ 2)]\!] =$ '(1 2)
- $[\![' \text{x}]\!] = \text{x}$
- $[\!['' \text{x}]\!] =$ ‘x
- $[\![' (\text{if x y z})]\!] =$ (if x y z)

Quiz

- What is $[\![''(1\ 2\ 3)]\!]$?
  A. nil
  B. (1 2 3)
  C. '(1 2 3)
  D. ''(1 2 3)
  E. Undefined (not a valid expression)

Quiz

- What is $[\!['(\text{if t } 0)]\!]$?
  A. nil
  B. (if t 0)
  C. ‘(if t 0)
  D. Undefined (not a valid expression)

Quiz

- How many times can we evaluate (list 'list 1 2 3) ?
  A. 0 times (not a valid expression)
  B. 1 time
  C. 2 times
  D. 3 times
  E. An arbitrary number of times

let

```
(let ((x1 e1)
      (x2 e2)
      ...
      (xn en))
   body)
```

binds variables x1, x2, …, xn, in parallel, to the values of expressions e1, e2, …, en, respectively; and then evaluates the body using that binding.

For example:

```
(let ( (x '(a b c))
       (y '(c d)) )
       (app (app x y) (app x y)))
```

= (a b c c d a b c c d)

let

```
(let ( (x '(a b c))
       (y '(c d)) )
       (app (app x y) (app x y)))
```

= (a b c c d a b c c d)

What if we do this instead:

```
(let ( (x '(a b c))
       (y '(c d))
       (z (app x y)) )
       (app z z))
```

? = (a b c c d a b c c d)

let

```
(let ( (x '(a b c))
       (y '(c d)) )
       (app (app x y) (app x y)))
```

= (a b c c d a b c c d)

What if we do this instead:

```
(let ( (x '(a b c))
       (y '(c d))
       (z (app x y)) )
       (app z z))
```

let binds **in parallel**. Variables x and y are undeclared here. Doesn't work (error).

let and let*

```
(let ( (x '(a b c))
       (y '(c d)) )
       (app (app x y) (app x y)))
```

= (a b c c d a b c c d)

```
(let* ( (x '(a b c))
        (y '(c d))
        (z (app x y)) )
        (app z z))
```

= (a b c c d a b c c d)

let and let*

```
(let ((x1 e1)
      (x2 e2)
      ...
      (xn en))
   body)
```

binds variables x1, x2, …, xn, **in parallel**, to the values of expressions e1, e2, …, en, respectively; and then evaluates the body using that binding.

```
(let* ((x1 e1)
       (x2 e2)
       ...
       (xn en))
   body)
```

binds variables x1, x2, …, xn, **in sequence**, to the values of expressions e1, e2, …, en, respectively; and then evaluates the body using that binding.