Logic and Computation – CS 2800 Fall 2019

Lecture 4

The ACL2s universe

Expressions, syntax and semantics

Stavros Tripakis

Northeastern University
Khoury College of
Computer Sciences

Outline

- The ACL2s universe: atoms and conses
- Basic data types: Booleans, integers, rationals, …
- Expressions and values
- Syntax and semantics

- Quiz: today we'll start taking graded quizzes using poll everywhere

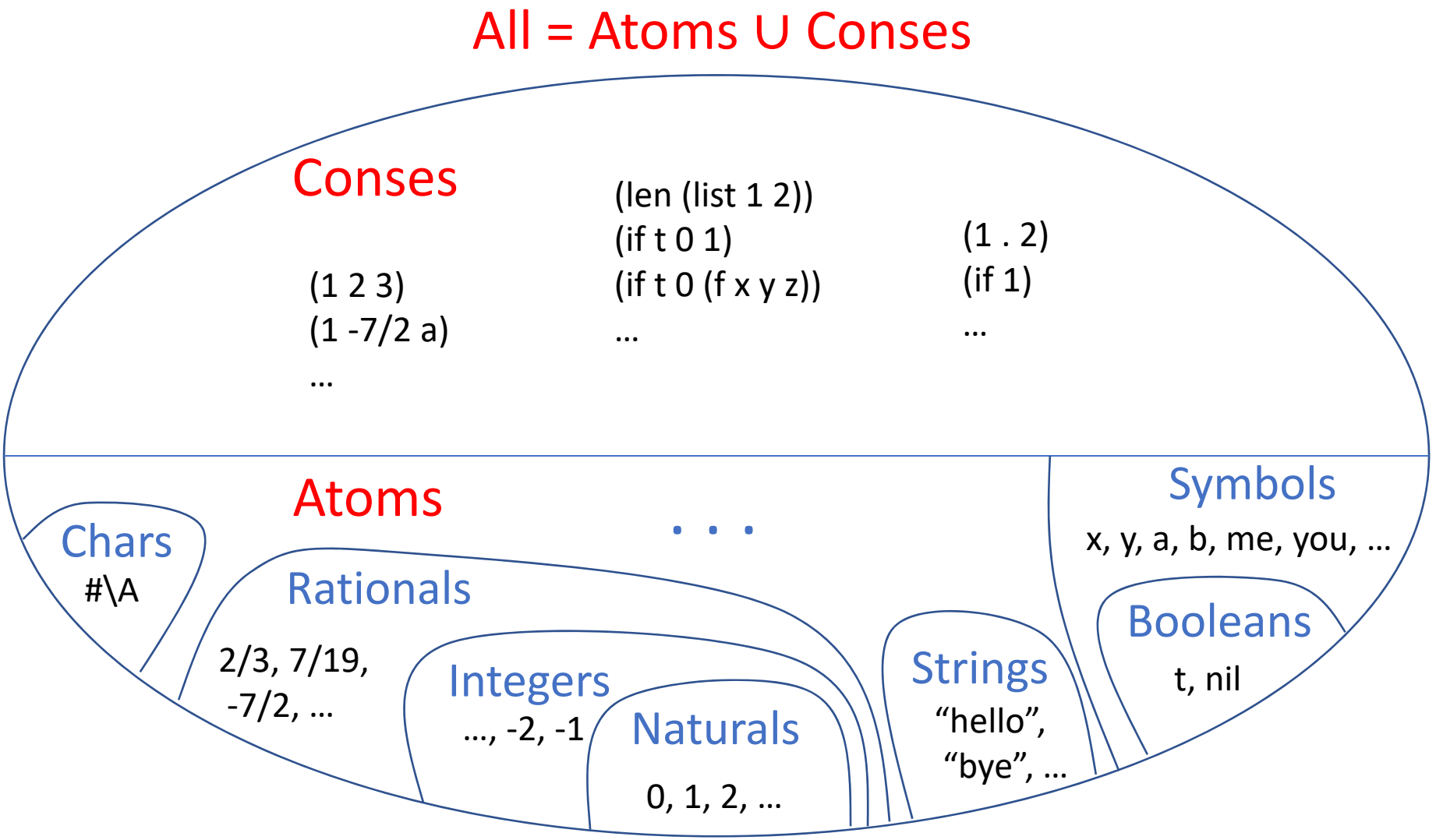
The ACL2s universe

The ACL2s universe

- ACL2s programs manipulate objects from the "ACL2s universe"

- What's in that universe?

The ACL2s universe



Quiz

http://PollEv.com/stavrostrypa519

- nil is a:
  - A. atom
  - B. cons
  - C. A & B (i.e., both an atom and a cons)

Always pick the best answer.
For example, if A and B all both true, then pick answer C.

Quiz

http://PollEv.com/stavrostrypa519

- t is a:
  - A. symbol
  - B. atom
  - C. Boolean
  - D. A & B
  - E. B & C
  - F. A & B & C

Expressions and evaluation (syntax and semantics)

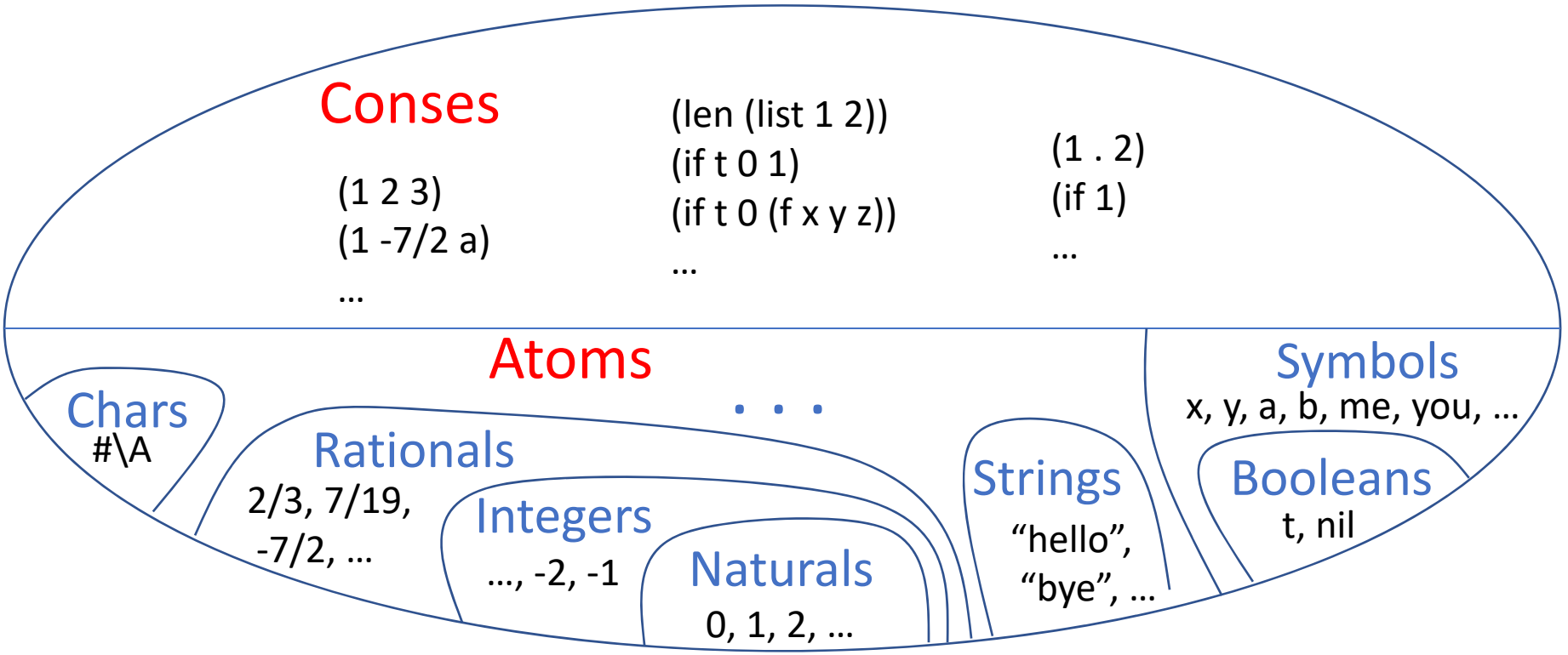
Expressions

Expressions are objects in the ACL2s universe that **can be evaluated**.

The result is a **value** (an object in the universe).

Some objects are not valid expressions, e.g., `(if nil 0)` is not (why?).

If you are unsure whether something can be evaluated or not, try it in ACL2s!



Evaluation rules (semantics)

- $eval(expr)$ denotes the semantics of an expression $expr$
  - i.e., what $expr$ evaluates to at the REPL
  - Sometimes also denoted $[\![expr]\!]$

- Constants evaluate to themselves:
  - $eval(\mathrm{t}) = \mathrm{t}$
  - $eval(\mathrm{nil}) = \mathrm{nil}$
  - $eval(0) = 0$
  - $eval(7/2) = 7/2$
  - $eval(-2) = -2$
  - …

Semantics of `if`

- `if` is a function with the following signature:
  - `if : All x All x All -> All`
- Evaluation rules (semantics) of `if`:
  - $[\![(\text{if } test\ expr_1\ expr_2)]\!] = \begin{cases} [\![expr_1]\!] & \text{when } [\![test]\!] \neq \text{nil} \\ [\![expr_2]\!] & \text{when } [\![test]\!] = \text{nil} \end{cases}$
- E.g.:
  - $[\![(\text{if t nil t})]\!] =$
  - $[\![(\text{if (if t nil t) 1 2})]\!] =$

Semantics of `if`

- `if` is a function with the following signature:
  - `if : All x All x All -> All`
- Evaluation rules (semantics) of `if`:
  - $[\![(\text{if } test\ expr_1\ expr_2)]\!] = \begin{cases} [\![expr_1]\!] & \text{when } [\![test]\!] \neq \text{nil} \\ [\![expr_2]\!] & \text{when } [\![test]\!] = \text{nil} \end{cases}$
- E.g.:
  - $[\![(\text{if t nil t})]\!] = \text{nil}$
  - $[\![(\text{if } (\text{if t nil t})\ 1\ 2)]\!] = [\![(\text{if nil } 1\ 2)]\!] = 2$

Semantics of `if`

- `if` is a function with the following signature:
  - `if : All x All x All -> All`
- Evaluation rules (semantics) of `if`:
  - $[\![(\text{if } test\ expr_1\ expr_2)]\!] = \begin{cases} [\![expr_1]\!] & \text{when } [\![test]\!] \neq \text{nil} \\ [\![expr_2]\!] & \text{when } [\![test]\!] = \text{nil} \end{cases}$

- Note that we can only evaluate valid expressions! Otherwise, semantics is undefined. E.g., the arity of if must be satisfied:
  - $[\![(\text{if t nil})]\!] = ?$
  - $[\![(\text{if } 1\ 2\ 3)]\!] = ?$

Semantics of `if`

- `if` is a function with the following signature:
  - `if : All x All x All -> All`
- Evaluation rules (semantics) of `if`:
  - $[\![(\text{if } test\ expr_1\ expr_2)]\!] = \begin{cases} [\![expr_1]\!] & \text{when } [\![test]\!] \neq \text{nil} \\ [\![expr_2]\!] & \text{when } [\![test]\!] = \text{nil} \end{cases}$

- Note that we can only evaluate valid expressions! Otherwise, semantics is undefined. E.g., the arity of if must be satisfied:
  - $[\![(\text{if t nil})]\!] =$ undefined!
  - $[\![(\text{if } 1\ 2\ 3)]\!] = 2$ because $[\![1]\!] = 1 \neq \text{nil}$

Lazy vs strict evaluation

- Evaluation rules (semantics) of if:
  - $[\![(\text{if } test\ expr_1\ expr_2)]\!] = \begin{cases} [\![expr_1]\!] & \text{when } [\![test]\!] \neq \text{nil} \\ [\![expr_2]\!] & \text{when } [\![test]\!] = \text{nil} \end{cases}$

- ACL2s if is **lazy**. This means that expressions are evaluated only when necessary:
  - $[\![test]\!]$ is always evaluated
  - $[\![expr_1]\!]$ is only evaluated if $[\![test]\!]$ turns out to be something other than nil.
  - $[\![expr_2]\!]$ is only evaluated if $[\![test]\!]$ turns out to be nil.

- All other ACL2s functions are **strict**:
  - First all arguments to the function are evaluated
  - Then function is applied to the argument values

Syntax vs semantics: recap

- In general:
  - Syntax: some text or graphics
  - Semantics: what this text/graphics means

- In our case:
  - Syntax: an object in the ACL2s universe
  - Semantics: what value the object evaluates to, provided that the object is a (valid) expression!

- Notes:
  - Some objects are not expressions, therefore do not evaluate to anything!
  - The values are themselves objects, which might be evaluated again, provided they are (valid) expressions!

Quiz

http://PollEv.com/stavrostrypa519

- How many times can -13/3 be evaluated?
  A. Zero times (not a valid expression)
  B. 1 time
  C. 2 times
  D. Any number of times

Semantics of `equal`

- `equal` is a function with the following signature:
  - `equal : All x All -> Boolean`
- Evaluation rules (semantics) of `equal`:
  - $[\![(\text{equal } expr_1\ expr_2)]\!] = \begin{cases} \text{nil} & \text{when } [\![expr_1]\!] \neq [\![expr_2]\!] \\ \text{t} & \text{when } [\![expr_1]\!] = [\![expr_2]\!] \end{cases}$
- E.g.:
  - $[\![(\text{equal } 3 \text{ nil})]\!] =$
  - $[\![(\text{equal } (\text{if } (\text{if t nil t}) \ 1 \ 2) \ 2)]\!] =$

Semantics of `equal`

- `equal` is a function with the following signature:
  - `equal : All x All -> Boolean`
- Evaluation rules (semantics) of `equal`:
  - $[\![(\text{equal } expr_1\ expr_2)]\!] = \begin{cases} \text{nil} & \text{when } [\![expr_1]\!] \neq [\![expr_2]\!] \\ \text{t} & \text{when } [\![expr_1]\!] = [\![expr_2]\!] \end{cases}$
- E.g.:
  - $[\![(\text{equal } 3\ \text{nil})]\!] = \text{nil}$
  - $[\![(\text{equal } (\text{if } (\text{if t nil t})\ 1\ 2)\ 2)]\!] = \text{t}$

booleanp

- `booleanp` is a recognizer: it takes anything as input and returns a Boolean (the "p" is for "predicate")
  - `booleanp : All -> Boolean`

```
(definec booleanp (x :all) :boolean
  (if (equal x t)
    t
    (equal x nil)))
```

booleanp

- Let's evaluate this expression:

$$[\![(\text{booleanp } 3)]\!] =$$

```
(definec booleanp (x :all) :boolean
  (if (equal x t)
    t
    (equal x nil)))
```

booleanp

- Let's evaluate this expression:

$$[\![(\text{booleanp } 3)]\!]$$
$$= \{ \text{ definition of booleanp } \}$$
$$[\![(\text{if } (\text{equal } 3\ \text{t})\ \ \text{t}\ \ (\text{equal } 3\ \text{nil}))]\!]$$
$$= \{ \text{ lazy if, semantics of equal } \}$$
$$[\![(\text{if nil}\ \ \text{t}\ \ (\text{equal } 3\ \text{nil}))]\!]$$
$$= \{ \text{ semantics of if } \}$$
$$[\![(\text{equal } 3\ \text{nil})]\!]$$
$$= \{ \text{ semantics of equal } \}$$
$$\text{nil}$$

```
(definec booleanp (x :all) :boolean
  (if (equal x t)
    t
    (equal x nil)))
```

`and`, version 1

- How would you define logical `and` (conjunction)?

```
(definec and (a :bool  b :bool) :bool
  (if a b nil))
```

Not the way and is really defined!
We'll see why later.

Numbers

Numbers and number recognizers

- Built-in functions:
  - `integerp : All -> Boolean`
  - `rationalp : All -> Boolean`
- Semantics:
  - $[\![(\text{integerp } x)]\!]$ is t iff $[\![x]\!]$ is an integer
  - $[\![(\text{rationalp } x)]\!]$ is t iff $[\![x]\!]$ is a rational
  - In ACL2s we get "real" (mathematical) numbers, not approximations like in C, Java, …
  - All integers are rationals
  - 4/2 is the same number as 2, and 3/6 = 2/4 = 1/2 = …
    - **Different syntax, same semantics!**
    - Can the same syntax result in different semantics?

Rationals

- $[\![4/2]\!] = 2$
- $[\![2/4]\!] = 1/2$
- $[\![-132/765]\!] = -44/255$
- etc

- Built-in functions:
  - `numerator: Rational -> Integer`
  - `denominator: Rational -> Pos`

Integers > 0

More numeric functions

- Built-in functions:
  - `+, *, / : Rational x Rational -> Rational`
  - `< : Rational x Rational -> Boolean`
  - `unary-- : Rational -> Rational`
  - `unary-/ : Rational \ {0} -> Rational`

posp

- Let's try to define the recognizer posp of positive integers:
  - `posp : All -> Boolean`

```
(definec posp (x :all) :bool
  (and (integerp x) (< 0 x)))
```

Do you see anything wrong with the above definition?
Assume the function `and` is defined as previously:

```
(definec and (a :bool  b :bool) :bool
  (if a b nil))
```

posp

- Let's try to define the recognizer posp of positive integers:
  - `posp : All -> Boolean`

```
(definec posp (x :all) :bool
  (and (integerp x) (< 0 x)))
```

Contract violation! E.g., when x is a list.

This is because `and` is defined as a function => it is strict.

How can we fix this?

```
(definec and (a :bool  b :bool) :bool
  (if a b nil))
```

posp

- Let's try to define the recognizer posp of positive integers:
  - `posp : All -> Boolean`

```
(definec posp (x :all) :bool
  (if (integerp x) (< 0 x) nil))
```

One possible fix is to not use `and` at all.

posp

- Let's try to define the recognizer posp of positive integers:
  - `posp : All -> Boolean`

```
(definec posp (x :all) :bool
  (and (integerp x) (< 0 x)))
```

But maybe another solution is to make `and` lazy?

But how? (defined functions are all strict)

=> MACROS!

`and`, version 2 (macro)

```
(and)         -> t
(and x)       -> x
(and x y)     -> (if x y nil)
(and x y z) -> (if x (if y z nil) nil)
...
```

Macros are first expanded (rewritten) into their definitions.
Then evaluation happens as usual.

Both `and` and `or` are macros

```
(and)          -> t
(and x)        -> x
(and x y)      -> (if x y nil)
(and x y z)    -> (if x (if y z nil) nil)
...
```

```
(or)           -> nil
(or x)         -> x
(or x y)       -> (if x x y)
(or x y z)     -> (if x x (if y y z))
...
```

Do `and` and `or` expect Booleans?

Do they always return Booleans?

Boolean functions

(note: they return `bool`, but they accept `all`)

```
(definec implies (a :all b :all) :bool
  (if a (if b t nil) t))
```

```
(definec not (a :all) :bool
  (if a nil t))
```

```
(definec iff (a :all b :all) :bool
  (if a
      (if b t nil)
      (if b nil t)))
```

```
(definec xor (a :all b :all) :bool
  (if a
      (if b nil t)
      (if b t nil)))
```

Next time

- Introduction to ACL2s continued
- Conses, lists, true lists