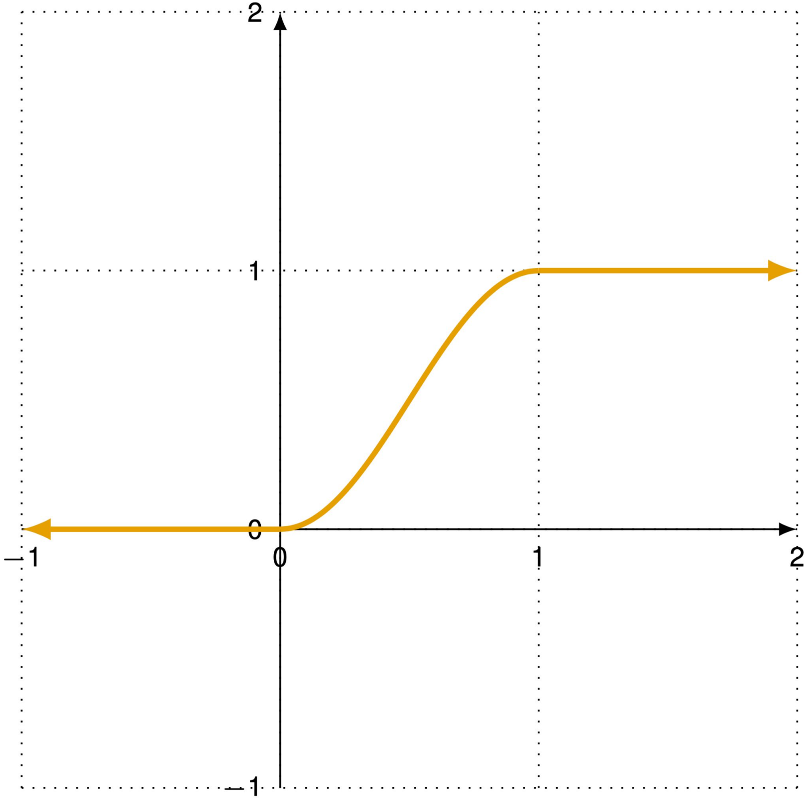
Conditional expressions

CS135 Lecture 03

L03.0 `cond` and `else`

Conditional expressions in mathematics.



Sometimes expressions should take one value under some conditions, and other values under other conditions.

A sin-squared window, used in signal processing, can be described by the following piecewise function:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 1 \\ \sin^2(x\pi/2) & \text{for } 0 \leq x < 1 \end{cases}$$

Conditional expressions in Racket

We can compute the sin-squared window function with a **`cond`** expression:

```
(define (f x)
   (cond [(< x 0) 0]
         [(>= x 1) 1]
         [(< x 1) (sqr (sin (* x pi 0.5)))]))
```

- Each argument to **`cond`** is a *question/answer pair*.
- The *question* is a Boolean expression
- The *answer* is a possible value of the conditional expression.
- By convention, square brackets are used around each question answer pair to improve readability.

Substitution steps (1 of 3)

```
(define (f x)
   (cond [(< x 0) 0]
         [(>= x 1) 1]
         [(< x 1) (sqr (sin (* x pi 0.5)))]))
(f 1/2)
```

⇒

```
 (cond [(< 1/2 0) 0]
       [(>= 1/2 1) 1]
       [(< 1/2 1) (sqr (sin (* 1/2 pi 0.5)))]))
```

Substitution steps (2 of 3)

⇒

```
 (cond [false 0]
       [(>= 1/2 1) 1]
       [(< 1/2 1) (sqr (sin (* 1/2 pi 0.5)))]))
```

⇒

```
 (cond [(>= 1/2 1) 1]
       [(< 1/2 1) (sqr (sin (* 1/2 pi 0.5)))]))
```

⇒

```
 (cond [false 1]
       [(< 1/2 1) (sqr (sin (* 1/2 pi 0.5)))]))
```

Substitution steps (3 of 3)

```
⇒ (cond [(< 1/2 1) (sqr (sin (* 1/2 pi 0.5)))]))

⇒ (cond [true (sqr (sin (* 1/2 pi 0.5)))]))

⇒ (sqr (sin (* 1/2 pi 0.5)))

⇒ (sqr (sin (* 1/2 #i3.141592653589793 1/2)))

⇒ (sqr (sin #i0.7853981633974483))

⇒ (sqr #i0.7071067811865475)

⇒ #i0.4999999999999999
```

`else`

Suppose our substitution steps reach:
… ⇒ `(cond [false …])`
The next step would be an error since there are no more question/answer pairs.

The last answer in a `cond` usually covers the case that none of the other question is `true`. For this last question we conventionally use the special construct `else`.

```
(define (f x)
   (cond [(< x 0) 0]
         [(>= x 1) 1]
         [else (sqr (sin (* x pi 0.5)))]))
```

Substitution rule #4: `cond` expressions

```
(cond [false e] ...) ⇒ (cond ...)

(cond [true e] ...) ⇒ e

(cond [else e]) ⇒ e
```

As you can see, we don’t really need **`else`**. We could just use **`true`** as the last question in a **`cond`**, but **`else`** makes it clearer that this last answer covers all remaining cases.

L03.1 Symbols

What to wear?

In Waterloo, temperatures can range from -34C to +38C.

- If it’s cold, I’ll wear a jacket.
- If it’s cool, I’ll wear sweater.
- Otherwise, I’ll just wear a shirt.

Write a function that consumes a temperature (`t`) and produce what to wear?

```
(define (cold? t) (< t 8))
(define (cool? t) (and (< t 16) (not (cold? t))))
(define (what-to-wear t) (???))
```

What to wear? Attempt #1: Define constants

```
(define (cold? t) (< t 8))
(define (cool? t) (and (< t 16) (not (cold? t))))

(define jacket 0)
(define sweater 1)
(define shirt 2)

(define (what-to-wear t)
  (cond [(cold? t) jacket]
        [(cool? t) sweater]
        [else shirt]))
```

What to wear? Attempt #2: Use symbols

```
(define (cold? t) (< t 8))
(define (cool? t) (and (< t 16) (not (cold? t))))

(define (what-to-wear t)
  (cond [(cold? t) 'jacket]
        [(cool? t) 'sweater]
        [else 'shirt]))
```

Symbols

Racket allows one to define and use *symbols* (or `Sym`) which have meaning to us (but not to Racket).

A symbol is defined using a leading apostrophe or ‘quote’: `'CS135`.

`'CS135` is a value just like `0` or `135`, but it is more limited computationally.

Symbols allow us to avoid using constants to represent names of clothes, courses, colours, planets, or types of music.

Unlike numbers, symbols are self-documenting – you don’t need to define constants for them. This is the primary reason we use them.

Hidden away in Racket, a symbol is still represented by a number.

Symbols

Symbols can be compared using the predicate **`symbol=?`**.

```
(define home 'Earth)
(symbol=? home 'Mars) ⇒ false
```

The predicate **`symbol?`** returns **`true`** if and only if its argument is a symbol.

```
(define mysymbol 'blue)
(symbol=? mysymbol 'blue) ⇒ true
(symbol=? mysymbol 'red) ⇒ false
(symbol=? mysymbol 42) ⇒ error
(symbol? mysymbol) ⇒ true
(symbol? 42) ⇒ false
```

L03.02 Testing

Testing

To write correct functions we first need to **understand the problem** we are trying to solve. To help understand a problem, we can start by writing some **test cases**.

- If it’s cold, I’ll wear a jacket.
- If it’s cool, I’ll wear sweater.
- Otherwise, I’ll just wear a shirt.

Suppose you don’t know what a person considers “cool” and “cold”, what would you expect the following to produce?

```
(what-to-wear -34)
(what-to-wear 38)
(what-to-wear 12)
```

Testing

- If it’s cold, I’ll wear a jacket.
- If it’s cool, I’ll wear sweater.
- Otherwise, I’ll just wear a shirt.

Suppose we know a person considers less than 8C to be “cold” and less than 16C to be “cool”, what would you expect the following to produce?

```
(what-to-wear 7)
(what-to-wear 8)
(what-to-wear 15)
(what-to-wear 16)
```

Testing with `check-expect`

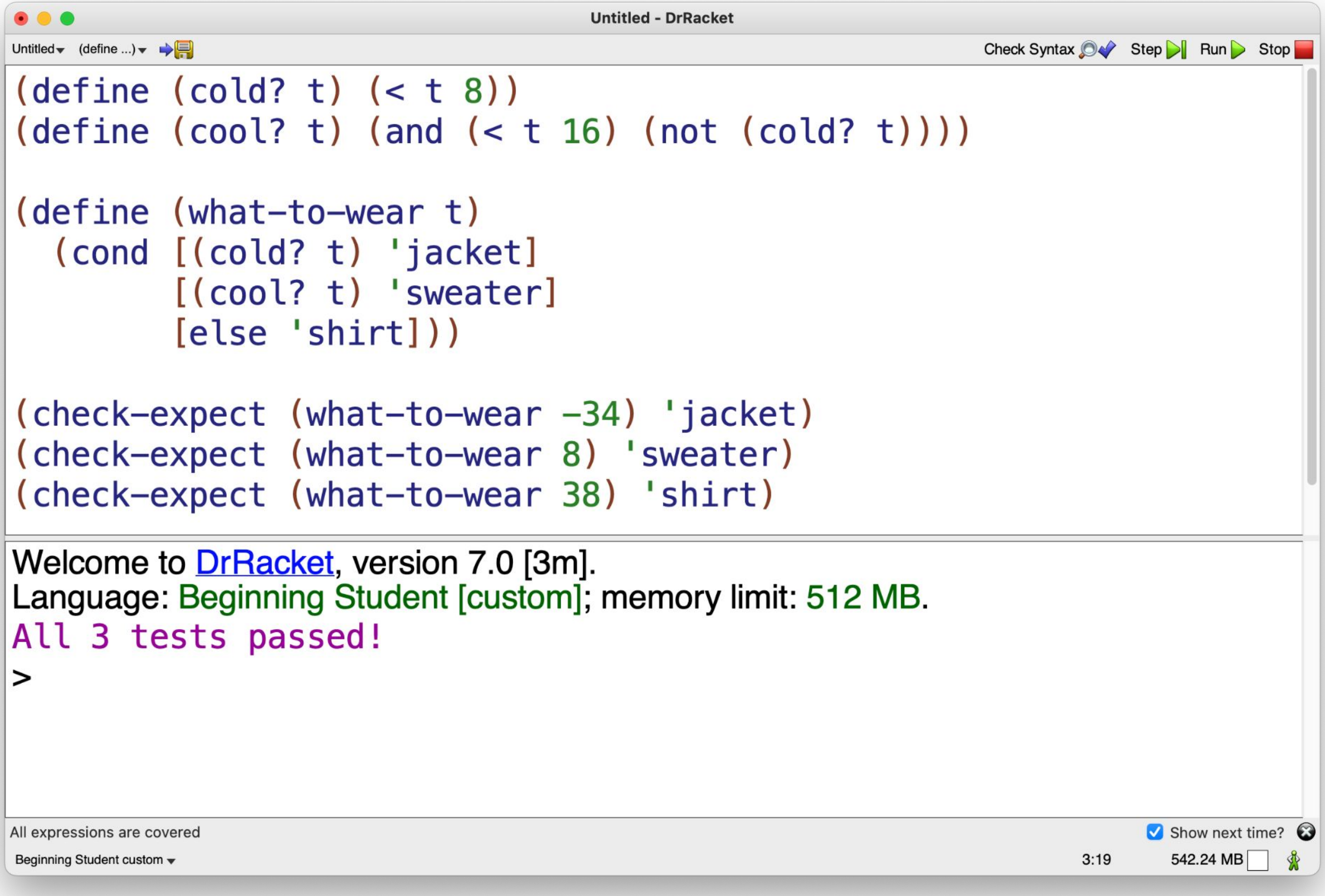
`check-expect` is a special Racket language feature we use for testing.

`(check-expect` *expr-test* *expr-expected*`)` consumes two expressions:

- *expr-test* is the expression (usually a function application) we are testing.
- *expr-expected* is the expected result; the “correct answer”.

```
(check-expect (what-to-wear -34) 'jacket)
(check-expect (what-to-wear 8) 'sweater)
(check-expect (what-to-wear 38) 'shirt)
```

This helps us understand the function, and demonstrates that it works correctly.



```
(define (cold? t) (< t 8))
(define (cool? t) (and (< t 16) (not (cold? t))))

(define (what-to-wear t)
  (cond [(cold? t) 'jacket]
        [(cool? t) 'sweater]
        [else 'shirt]))

(check-expect (what-to-wear -34) 'jacket)
(check-expect (what-to-wear 8) 'sweater)
(check-expect (what-to-wear 38) 'shirt)
```

```
Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.
All 3 tests passed!
>
```

Testing

To write correct functions we first need to **understand the problem**.

Writing test cases:
- …before we write a function,
  - helps us to understand the problem
- …after we write a function,
  - helps us to demonstrate that the function is correct

In CS135 we often call test cases written before the function “examples”. For complex problems, it can help to start by writing some examples by hand.

We will write lots of examples in CS135.

Self check

✓

Suggest some test cases for a function that computes:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 1 \\ \sin^2(x\pi/2) & \text{for } 0 \leq x < 1 \end{cases}$$

```
(check-expect (f -99) 0)
(check-expect (f 0) 0)
(check-expect (f 1) 1)
(check-expect (f 99) 1)
(check-expect (f 1/2) 1/2)???
```

Testing with **`check-within`**

`check-within` is a special Racket language feature like **`check-expect`**.

`(check-expect` *expr-test expr-expected delta***`)`** consumes three expressions:

- *expr-test* is the expression (usually a function application) we are testing.
- *expr-expected* is the expected result.
- *delta* is the allowed absolute difference between expected and actual results.

```
(check-within (sqrt 2) 1.414 0.001)
(check-within (f 1/2) 0.5 0.0001)
```

For the check to succeed, the actual value of *expr-test* must be within *delta* of *expr-expected.* In CS135, the value of *delta* is usually defined by the problem.

Tests in assignment submissions

Before you write your function, write some test cases by hand to make sure that you understand the problem (sometimes called “examples”). Try to think of “boundary cases”, places where the function may change its behavior, e.g. freezing (0C). Try to think about what the function should do with extreme examples, e.g., very cold , very hot, exactly 100C, etc.

After you write your function, write test cases that cover the boundary cases in `cond` expressions. For each question/answer pair (except `else`) there should be at least one test case where the question is `true` and at least one test case where the answer is **false**.

It’s hard to test too much. At a minimum, assignment submissions *must* test all boundary cases in `cond` expressions.

L03.03 Contracts

Contract

A contract formally defines what type of arguments a function consumes and what type of result it produces. A contract is written as a comment before the function definition.

```
;; g: Num Num -> Num
(define (g x y)
   (+ (sqr x) (* 6 x y) (sqr y) (* 9 x) (- (* 3 y)) -100))

;; cool?: Num -> Bool
(define (cool? t) (and (< t 16) (not (cold? t))))
```

Types can be: `Bool`, `Int`, `Nat`, `Num`, `Rat`, or `Sym` (with more to come)

Contracts with Sym

We usually write

```
;; what-to-wear: Num -> Sym
```

If we want to be very precise, we can also write

```
;; what-to-wear: Num -> (AnyOf 'jacket, 'sweater, 'shirt)
```

At this point in the course, the first version is acceptable unless we specifically request **`AnyOf`** in an assignment or exam.

Lecture 03 Summary

Assignment submissions (end of L03)

For each function we ask you to write, you should include:

1. Question number
2. Purpose (what does the function produce and consume)
3. Contract
4. Function definition
5. Test cases (including any examples you write)

Assignment submission example

```
;;
;; Question 6b)
;;
;; what-to-wear consumes a temperature and produces
;; a symbol indicating the clothing to wear
;;
;; what-to-wear: Num -> Sym

(define (what-to-wear temperature)
  (cond [(< temperature 8) 'jacket]
        [(< temperature 16) 'sweater]
        [else 'shirt]))
```

continued…

Assignment submission example (continued)

```
(check-expect (what-to-wear -34) 'jacket)
(check-expect (what-to-wear 12) 'sweater)
(check-expect (what-to-wear 38) 'shirt)
(check-expect (what-to-wear 7) 'jacket)
(check-expect (what-to-wear 8) 'sweater)
(check-expect (what-to-wear 15) 'sweater)
(check-expect (what-to-wear 16) 'shirt)
```

What happens next?

Over four lectures we will develop our model of computation:

1. Values and expressions
2. Functions
3. **Conditional expressions**
4. Recursion

After the final step, we will have built a complete “computer”, essentially from math.

We will then add “lists” to our model of computation to simplify data organization.

We will then explore a variety of basic algorithms and data structures using lists.

L03: You should know

- How to use conditional expressions
  - How to write question/answer pairs with `cond`, `else`
  - Substitution rules for conditional expressions.
  - How to step a conditional expression.
- How to use symbols
  - How to produce and consume symbols
  - `Sym`, `symbol?`, `symbol=?`
- How to test your functions
  - `check-expect, check-within`
  - How to identify boundary cases from a `cond.`
- How to define and understand contracts
- What to include in your assignment submissions (to L03).

L03: Allowed constructs

Newly allowed constructs:

`[ ]`(surrounding question/answer pairs)

`check-expect check-within cond else symbol? symbol=?`

`AnyOf Sym`

Previously allowed constructs:

`( ) + - * / = < > <= >= ;`

`abs acos and asin atan cos define e exp expt false inexact?`

`log max min not or pi quotient remainder sin sqr sqrt tan`

`true`

`Bool Int Nat Num Rat`