Values

Saturday, September 12, 2020 20:26

So far our only values have been numbers, abbreviated Num in contracts
We've identified several subtypes: Int for integers and Nat for natural numbers

The literal representations of Booleans, Symbols and Strings:

- **Boolean**: true, false
- **Symbol**: a "word" that begins with a single quote such as 'earth or 'female
- **String**: characters (other than ") surrounded by double-quotes: "This is a string"

A **literal** is the way you write something down.
E.g. The literal for the number seven is 7.

Booleans

Saturday, September 12, 2020 20:37

Boolean-valued functions

- A function that tests whether two numbers x and y are equal has two possible **Boolean values**: true and false.
- An example application: (= x y)
- This is equivalent to determining whether the mathematical **proposition** "x=y" is true or false.
- Standard Racket uses #t and #true for true, and similarly, #f and #false for false.
- **Note: You should always use true and false**
- The sample application will give an error unless x and y have been defined.

```
(define x 3)
(define y 4)
(= x y)
```

Other types of comparisons

- (< x y)
- (> x y)
- (<= x y)
- (>= x y)
- **Comparisons** are functions which consume two numbers and produce a Boolean value.
- A sample contract:

```
;; = : Num Num → Bool
```

Complex relationships

- and, or and not are used to test complex relationships.
- Example: "3 <= x < 7" is represented as

```
(and (<= 3 x) (< x 7))
```

- and and or are actually **special forms**, like define. They look like functions but actually are not because their arguments are not evaluated before the "function" is applied

Computational differences

- The mathematical AND and OR connect two propositions
- In Racket, and and or may have more than two arguments
- and has value true exactly when **all** of its arguments have value true
- or has value true exactly when **at least one** of its arguments has value true

- not has value true exactly when its **one** argument has value false

Short-circuit evaluation

- DrRacket only evaluates as many arguments of and and or as is necessary to determine the value.
  Examples:

```
;; Eliminate easy cases first; might not need to do
;; the much slower computation of prime?
(and (odd? x) (> x 2) (prime? x))

;; Is the line considered "steep"?
(define (steep? delta-x delta-y)
  (or (= delta-x 0)     ; Avoid dividing by zero
      (>= (/ delta-y delta-x) 1)))

(define (not-steep? delta-x delta-y)
  (and (not (= delta-x 0))   ; Avoid dividing by zero
       (< (/ delta-y delta-x) 1)))
```

- In the second example above, once the (= delta-x 0) is evaluated to be true, the function will output true and stop since or only needs one true value
- In the third example above, once the (= delta-x 0) is evaluated to be true, the function will output false and stop since and stops evaluating once there is a false value
- This practice is sometimes called **short circuiting**

Predicates

- A **predicate** is a function that produces a Boolean result.
- Some built-in predicates in Racket: even?, negative? and zero?
- Examples of user-defined functions:

```
(define (between? low high numb)
  (and (< low numb) (< numb high)))

(define (can-vote? age)
  (>= age 18))
```

- Predicate names ending with a question mark is a convention

Conditionals

Saturday, September 12, 2020 22:14

Conditional expressions

- In Racket, we can computer |x| with the conditional expression:

```
(cond [(< x 0)   (- x)]
      [(>= x 0)      x])
```

- **Conditional expressions** use the special form cond.
- Each argument is a question/answer pair.
- The **question** is a Boolean expression.
- The **answer** is a possible value of the conditional expression.
- Square brackets are used by convention, for readability.
- abs is a built-in function in Racket.
- Properly nested brackets: [( )].
- Improperly nested brackets: [( ]] or [( ]).
- Here is a function my-abs because Racket won't let us redefine the built-in function.

```
(define (my-abs x)
  (cond [(< x 0) (- x)]
        [(>= x 0) x]))
```

General form

- The general form of a conditional expression is:

```
(cond [question1 answer1]
      [question2 answer2]
      ...
      [questionk answerk])
```

- The questions are evaluated in top-to-bottom order
- As soon as one question is evaluated to true, no further questions are evaluated
- Only one answer is ever evaluated
- An error is produced if no question evaluates to true

Example

$$f(x) = \begin{cases} 0 & \text{when } x = 0 \\ x \sin(1/x) & \text{when } x \neq 0 \end{cases}$$

```
(define (f x)
  (cond [(= x 0) 0]
        [else (* x (sin (/ 1 x)))]))
```

Simplifying conditional expressions

- Sometimes a question can be simplified by knowing that if it is asked, all previous questions have evaluated to false
- Example:

  Here are the common recommendations on which course to take after CS 135, based on the CS135 mark earned.

  - 0% $\leq$ mark $<$ 40%: CS 115 is recommended
  - 40% $\leq$ mark $<$ 50%: CS 135 is recommended
  - 50% $\leq$ mark $<$ 60%: CS 116 is recommended
  - 60% $\leq$ mark: CS 136 is recommended

  We might write the tests for the four intervals this way:

```
(define (course-after-cs135 grade)
  (cond [(< grade 40) 'CS115]
        [(and (>= grade 40) (< grade 50)) 'CS135]
        [(and (>= grade 50) (< grade 60)) 'CS116]
        [(>= grade 60) 'CS136]))
```

- The code shown is a straight-forward and correct implementation of the conditions.
- But when (>= grade 40) is evaluated in the second question/answer pair, we already know that grade is at least 40.
- Because of Racket's top-to-bottom evaluation of a cond, the function will output 'CS115 and stop if the grade is less than 40.
- Similarly, we cannot reach the third question/answer pair unless the grade is at least 50.
- Simplifying three of the tests:

```
(define (course-after-cs135 grade)
  (cond [(< grade 40) 'CS115]
        [(< grade 50) 'CS135]
        [(< grade 60) 'CS116]
        [else 'CS136]))
```

Tests

Monday, September 14, 2020 10:47

Tests for conditional expressions

- Write at least one test for each possible answer in the expression
- Should be simple and direct, aimed at testing that answer
- When the problem contains **boundary conditions** (like the cut-off between passing and failing), they should be tested explicitly
- Example:

```
(define (course-after-cs135 grade)
  (cond [(< grade 40) 'CS115]
        [(< grade 50) 'CS135]
        [(< grade 60) 'CS116]
        [else 'CS136]))
```

  There are four intervals and three boundary points, so seven tests are required
  For instance, 30, 40, 45, 50, 55, 60, 70

Testing and and or

- Consider

```
;; Is the line considered "steep"?
(define (steep? dx dy)
  (or (= dx 0)
      (>= (/ dy dx) 1)))
```

- We need:
  - one test case where dx is zero (first argument is true; second is not evaluated)
  - one test case where dx is nonzero and dy/dx >= 1 (first false; second true)
  - one test case where dx is nonzero and y/x < 1 (both false)
- More generally,
  - or: enough tests to make the expression true for each clause and one that makes the entire expression false
  - and: enough tests to make the expression false for each clause and one that makes the entire expression true

Closed-box vs. Open-box testing

- **Closed-box tests** are some of the tests, including the examples, that have been defined before the body of the function was written
- **Open-box tests** may depend on the code, for example, to check specific answers in conditional expressions
- Both types of tests are important

Note: use check-expect for all tests

Example

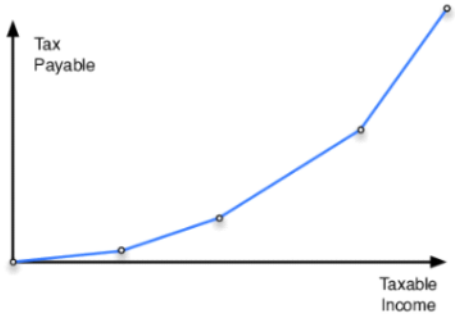
Monday, September 14, 2020 17:08

Purpose: Compute the Canadian tax payable on a specified income.

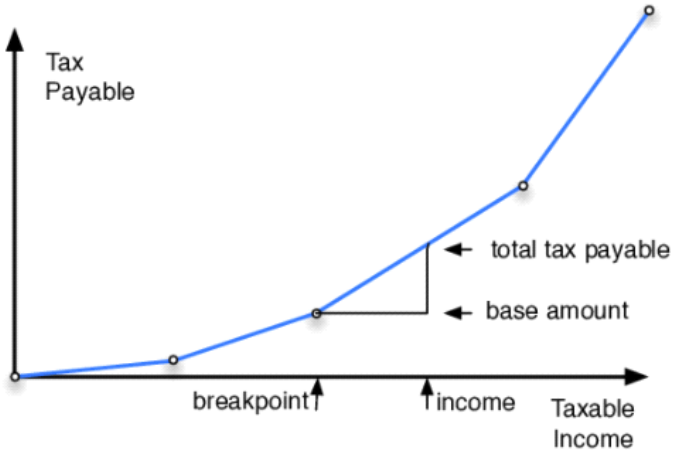
Background:

- 15% on the amount in [$0, $45,000]
- 20% on the amount in ($45,000, $90,000]
- 25% on the amount in ($90,000, $150,000]
- 30% on the amount in ($150,000, $200,000]
- 35% on the amount over $200,000

Note: These amounts are rounded from the actual amounts to make discussing them easier.



The "piecewise linear" nature of the graph complicates the computation of tax payable. One way to do it uses the **breakpoints** (x-value or salary when the rate changes) and **base amounts** (y-value or tax payable at breakpoints).



Examples:

Income	Tax Calculation
$40,000	$0.15 * 40000 = 6000$
$60,000	$0.15 * 45000 + 0.20 * (60000 - 45000) = 6750 + 3000 = 9750$
$100,000	$0.15 * 45000 + 0.20 * (90000 - 45000) + 0.25 * (100000 - 90000)$ $= 6750 + 9000 + 2500 = 18250$

```
(check-expect (tax-payable 40000) (* 0.15 40000))
(check-expect (tax-payable 60000) 9750)
(check-expect (tax-payable 100000) 18250)
```

Definition header & contract

```
;; tax-payable: Num → Num
;;    requires: income >= 0
```

Finalize purpose

```
;; (tax-payable income) computes the Canadian tax payable on income.
```

Some constants will be useful. Put these before the purpose and other design recipe elements.

```
;; Rates                    ;; Breakpoints in increasing order by income
(define rate1 0.15)         (define bp1 45000)
(define rate2 0.20)         (define bp2 90000)
(define rate3 0.25)         (define bp3 150000)
(define rate4 0.30)         (define bp4 200000)
(define rate5 0.35)
```

Instead of putting the base amounts into the programs as numbers, we can compute them from the breakpoints and rates

```
;; Base Amounts
;; basei is the base amount for interval [bpi,bp(i+1)]
;; that is, tax payable at income bpi
(define base1 (* (- bp1 0) rate1))
(define base2 (+ base1 (* rate2 (- bp2 bp1))))
(define base3 (+ base2 (* rate3 (- bp3 bp2))))
(define base4 (+ base3 (* rate4 (- bp4 bp3))))
```

Developing tax-payable

```
;; tax-payable: Num → Num
;; Requires:  income >= 0
(define (tax-payable income)
  (cond [(< income bp1) (* rate1 income)]
        [(< income bp2) (+ base1 (* rate2 (- income bp1)))]
        [(< income bp3) (+ base2 (* rate3 (- income bp2)))]
        [(< income bp4) (+ base3 (* rate4 (- income bp3)))]
        [else (+ base4 (* rate5 (- income bp4)))]))
```

At least 9 tests are needed for the preceding code (5 intervals and 4 boundaries)

Helper functions

- There are many similar calculations in the tax program, leading to the definition of the following **helper function**:

```
;; (cum-tax base rate low high) calculates the cumulative tax owed
;;   where base is the tax owed on income up to low and rate is
;;   the tax rate on income between low and high.

;; cum-tax: Num Num Num Num → Num
;; Requires base >= 0, rate >=0, 0 <= low <= high
(define (cum-tax base rate low high)
  (+ base (* rate (- high low))))
```

- tax-payable with a helper function

```
;; Base Amounts
(define base1 (cum-tax 0 rate1 0 bp1))
(define base2 (cum-tax base1 rate2 bp1 bp2))
(define base3 (cum-tax base2 rate3 bp2 bp3))
(define base4 (cum-tax base3 rate4 bp3 bp4))


(define (tax-payable income)
  (cond [(< income bp1) (cum-tax 0 rate1 0 income)]
        [(< income bp2) (cum-tax base1 rate2 bp1 income)]
        [(< income bp3) (cum-tax base2 rate3 bp2 income)]
        [(< income bp4) (cum-tax base3 rate4 bp3 income)]
        [else (cum-tax base4 rate5 bp4 income)]))
```

Helper functions are used for three purposes

- Reduce repeated code, sometimes referred to as "DRY", "Don't Repeat Yourself". Being DRY reduces the amount of code you need to write, debug and maintain
- Factor out complex calculations, giving a separate function that is usually easier to test
- Give names to operations. Having meaningful names in the code also helps with chunking things.

Style guidelines:

- Improve clarity with short definitions using well-chosen names
- Name all functions (including helpers) meaningfully; not "helper"
- Purpose, contract and one example are required

Other data

Wednesday, September 16, 2020 9:39

Symbolic data

- Racket allows one to define and use **symbols** with meaning to us
- A **symbol** is defined using an apostrophe: 'CS115
- 'CS115 is a value just like 0 or 115, but is more limited computationally
- Symbols allow a programmer to avoid using constants to represent names
- Symbols can be compared using the predicate symbol=?

```
(define home 'Earth)

(symbol=? home 'Mars) ⇒ false
```

- The contract for symbol=? is Sym Sym -> Bool. It consumes two symbols and produces a Boolean result
- Unlike numbers, symbols are **self-documenting**. You don't need to define constants for them

Characters

- A **character** is most commonly a printed letter, digit, or punctuation symbol
- a, G, ., + and 8 are all characters
- Other characters represent less visible things like a tab or a newline in text
- They are the simplest component of a string

Strings

- Strings are sequences of characters between double quotes
- Differences between strings and symbols
  - Strings are compound data (a sequence of characters)
  - Symbols cannot have certain characters in them (such as spaces)
  - More efficient to compare two symbols than two strings
  - More built-in functions for strings
- A few functions which operate on strings:

```
(string=? "alpha" "bet") ⇒ false
(string<? "alpha" "bet") ⇒ true
(string-append "alpha" "bet") ⇒ "alphabet"
(string-length "perpetual") ⇒ 9
(string-upcase "Hello") ⇒ "HELLO"
```

Symbols vs. strings

- Use symbols when a small, fixed number of labels are needed (e.g. planets) and comparing labels for equality is all that is needed.
- Use strings when the set of values is more indeterminate (e.g. names of students) or when more computation is needed (e.g. comparison in alphabetical order)
- When these types appear in contracts, they should be capitalized and abbreviated: Sym and Str
- Symbols are like multiple choice. Strings are like short answer

General equality testing

- Every type seen so far has an **equality predicate** (= for numbers, symbol=? for symbols, string=? for strings)
- The predicate equal? has a contract of Any Any -> Bool. It can be used to test the equality of two values which may or may not be of the same type.

```
(= 10 11) ⇒ false
(string=? "10" "10") ⇒ true
(= 10 "10") ⇒ Error
(equal? 10 "10") ⇒ false
```

- equal? works for all types of data we have seen so far (except inexact numbers)
- (symbol=? 'blue 100) breaks the contract so it will produce an error
- (equal? 'blue 100) will produce false
- Do not overuse equal? Use = for numbers, symbols=? for symbols and strings=? for strings.

Modelling

Wednesday, September 16, 2020 10:49

Modelling programming languages

- A program has a precise meaning and effect.
- If you run a program multiple times with exactly the same inputs, it better do the same thing every time
- A **model** of a programming language provides a way of describing the meaning of a program.
- It provides a way of understanding something, in this case a program.
- It is not necessarily the way DrRacket will execute the program, but it provides a way to predict the result that is easier for humans to understand than the internals of DrRacket

Spelling

Wednesday, September 16, 2020 11:23

Spelling rules for Beginning Student

- **Identifiers** are the names of constants, parameters and user-defined functions.
- They are made up of
  - letters
  - hyphens
  - underscores
  - a few other punctuation marks
  - at least one non-number
- They can't contain spaces or any of these: ( ) [ ] { } , ; ' ' " "
- Symbols start with a single quote ' followed by something obeying the rules for identifiers.

Semantics intro

Wednesday, September 16, 2020 11:41

Syntax, semantics and ambiguity:

There are three problems we need to address:

1. **Syntax**: The way we're allowed to say things
2. **Semantics**: the meaning of what we say
3. **Ambiguity**: valid sentences have exactly on meaning

In Racket, we need rules that always avoid these problems

Grammars

- We can use **grammars** to enforce syntax and avoid ambiguity
- For example, an English sentence can be made up of a subject, verb and object, in that order.
- We might express this as follows

  ⟨sentence⟩ = ⟨subject⟩ ⟨verb⟩ ⟨object⟩
- The textbook describes function definitions like this

  ⟨def⟩ = (**define** (⟨var⟩ ⟨var⟩ ... ⟨var⟩) ⟨exp⟩)
- The Help Desk presents the same idea as

```
definition = (define (id id id ...) expr)
```

Semantic model

Wednesday, September 16, 2020 12:14

Racket's semantic model

- A **semantic model** of a programming language provides a method of predicting the result of running any program
- Our model will repeatedly simplify the program via **substitution**
- A **substitution step** finds the leftmost subexpression eligible for rewriting
- **Every substitution step yields a valid program in full Racket**, until all that remains is a sequence of definitions and values

Application of built-in functions

- We reuse the rules for the arithmetic expressions we are familiar with to substitute the appropriate value for expressions like (+ 3 5) and (expt 2 10)

  `(+ 3 5)` $\Rightarrow$ `8`

  `(expt 2 10)` $\Rightarrow$ `1024`
- Formally, the substitution rule is

  `(f v1 ... vn)` $\Rightarrow$ `v` where `f` is a built-in function and `v` is the value of $f(v_1, \ldots, v_n)$.
- Essentially, the rule says that we just "know" what the built-in functions do from early grade school or by reading the DrRacket documentation

Ellipses

For built-in functions `f` with one parameter, the rule is:
`(f v1)` $\Rightarrow$ `v` where `v` is the value of $f(v_1)$

For built-in functions `f` with two parameters, the rule is:
`(f v1 v2)` $\Rightarrow$ `v` where `v` is the value of $f(v_1, v_2)$

For built-in functions `f` with three parameters, the rule is:
`(f v1 v2 v3)` $\Rightarrow$ `v` where `v` is the value of $f(v_1, v_2, v_3)$

We can't just keep writing down rules forever, so we use ellipses to show a *pattern*:
`(f v1 ... vn)` $\Rightarrow$ `v` where `v` is the value of $f(v_1, \ldots, v_n)$.

Application of user-defined functions

- Any argument which is not a value must first be simplified to a value using the rules for expressions
- We cannot claim to just "know" what a user-defined function does, so we need a new rule
- The general substitution rule is:

  `(f v1 ... vn)` $\Rightarrow$ `exp'`
  where `(define (f x1 ... xn) exp)` occurs to the left, and `exp'` is obtained by substituting into the expression `exp`, with all occurrences of the formal parameter `xi` replaced by the value `vi` (for `i` from 1 to n).

- **Note**: we are using a pattern ellipsis in the rules for both built-in and user-defined functions

to indicate several arguments

Example:

- A Racket program is read top-to-bottom, left-to-right. That is,

```
(define (foo x y) (+ x x y))
(foo 1 2)
```

  is read as

```
(define (foo x y) (+ x x y))  (foo 1 2)
```

- Evaluating (foo 1 2) means substituting the first argument (1) wherever the first parameter (x) occurs in the body expression and doing similarly for the second argument/parameter.
- The expression we get (+ 1 1 2) will be substituted back in place of (foo 1 2)

```
(define (foo x y) (+ x x y))  (foo 1 2) =>
(define (foo x y) (+ x x y))  (+ 1 1 2)
```

- Recall that => means "yields" and separates one substitution step from another

Another Example:

```
(define (term x y) (* x (sqr y)))

(term (- 3 1) (+ 1 2))
⇒ (term 2 (+ 1 2))
⇒ (term 2 3)
⇒ (* 2 (sqr 3))
⇒ (* 2 9)
⇒ 18
```

Constant definition

- A **constant definition** binds a name (the constant) to a value (the value of the expression)
- We add the substitution rule:

  `id ⇒ val`
  where `(define id val)` occurs to the left.

- Function definitions are always in simplest form and not further reduced, not always the case with constant definitions
- If the expression starts as (define p (* 3 3)) the (* 3 3) must be simplified to 9 first

Example

```
(define x 3)
(define y (+ x 1))
y ⇒
(define x 3)
(define y (+ 3 1))
y ⇒
(define x 3)
(define y 4)
y ⇒
(define x 3)
(define y 4)
4
```

To avoid a lot of repetition, we adopt the convention that we stop repeating a definition once its expression has been reduced to a value (since it cannot change after that).

```
(define x 3)
(define y (+ x 1))
y ⇒
(define y (+ 3 1))
y ⇒
(define y 4)
y ⇒
4
```

Substitution in cond expressions

- There are three rules: when the first expression is false, when it is true and when it is else

```
(cond [false exp] ...) ⇒ (cond ...)
```

```
(cond [true exp] ...) ⇒ exp
```

```
(cond [else exp]) ⇒ exp
```

- These suffice to simplify any cond expression
- Here the ellipses are serving a different role. They are showing an **omission**. The first rule just says "Whatever else appeared after the [false exp], you copy it over"
- Rule #2 - choose the leftmost subexpression to simplify
- It means that the question part of a cond's question/answer pair will always to be reduced to either true or false before we apply one of the three rules for cond
- cond is a special form. The arguments are not necessarily evaluated. In this case the question argument is evaluated but the answer argument is not

Example:

```
(define n 5) (define x 6) (define y 7)

(cond [(even? n) x][(odd? n) y])
⇒ (cond [(even? 5) x] [(odd? n) y])
⇒ (cond [false x][(odd? n) y])
⇒ (cond [(odd? n) y])
⇒ (cond [(odd? 5) y])
⇒ (cond [true y])
⇒ y
⇒ 7
```

If y is not defined:

```
(define n 5) (define x 6)

(cond [(even? n) x][(odd? n) y])
⇒ (cond [(even? 5) x] [(odd? n) y])
⇒ (cond [false x][(odd? n) y])
⇒ (cond [(odd? n) y])
⇒ (cond [(odd? 5) y])
⇒ (cond [true y])
⇒ y
⇒ y: this variable is not defined
```

DrRacket's rules differ. It scans the entire cond expression before it starts, notes that y is not defined, and shows an error.

Errors

- A **syntax error** occurs when a sentence cannot be interpreted using the grammar. It occurs in the Read phase of **REPL**. Some examples are:
  - misspellings
  - missing parentheses
  - invalid function applications

  Example: (10 + 1)
- A **run-time error** occurs when an expression cannot be reduced to a value by application of our evaluation rules. It occurs during the Evaluation phase of **REPL**.
  Some examples are:
  - division by zero
  - a cond with no question that returns true
  - running out of memory

  Example:

```
(cond [(> 3 4) x])
⇒ (cond [false x])
⇒ (cond )
⇒ cond: all question results were false
```

Substitution rules for and and or

- The simplification rules we use for Boolean expressions involving and and or are different from the ones the Stepper in DrRacket uses
- The end result is the same, but the intermediate steps are different
- The rules:

```
(and false ...) ⇒ false
```

```
(and true ...) ⇒ (and ...)
```

```
(and) ⇒ true
```

```
(or true ...) ⇒ true
```

```
(or false ...) ⇒ (or ...)
```

```
(or) ⇒ false
```

As in the rewriting rules for **cond**, we are using an omission ellipsis.

Substitution rules (so far)

1. Apply functions only when all arguments are values.
2. When given a choice, evaluate the leftmost expression first.
3. `(f v1...vn)` ⇒ `v` when `f` is built-in...
4. `(f v1...vn)` ⇒ `exp'` when `(define (f x1...xn) exp)` occurs to the left...
5. `id` ⇒ `val` when `(define id val)` occurs to the left.
6. `(cond [false exp] ...)` ⇒ `(cond ...)`
7. `(cond [true exp] ...)` ⇒ `exp`
8. `(cond [else exp])` ⇒ `exp`
9. `(and false ...)` ⇒ `false`
10. `(and true ...)` ⇒ `(and ...)`
11. `(and)` ⇒ `true`
12. `(or true ...)` ⇒ `true`
13. `(or false ...)` ⇒ `(or ...)`
14. `(or)` ⇒ `false`

Summary

- Doing a step-by-step reduction according to these rules is called **tracing** a program
- It is an important skill in any programming language or computational system

Lists Introduction

Friday, September 18, 2020 19:54

Introducing lists

- A **list** is a recursive structure - it is defined in terms of a smaller list.
- Consider a list of concerts:
  - A list of 4 concerts is a concert followed by a list of 3 concerts
  - A list of 3 concerts is a concert followed by a list of 2 concerts
  - A list of 2 concerts is a concert followed by a list of 1 concerts
  - A list of 1 concerts is a concert followed by a list of 0 concerts
- A list of zero concerts is special. We call it the **empty list**. It is represented in Racket by empty.
- In general, a list of n items is an item followed by a list of n-1 items.

Examples:

A sad state of affairs – no upcoming concerts to attend:

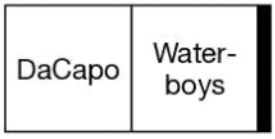
```
(define concerts0 empty)
```

A list with one concert to attend:

```
(define concerts1 (cons "Waterboys"
                        empty))
```

A new list just like `concerts1` but with a new concert at the beginning:

```
(define concerts2 (cons "DaCapo"
                        concerts1))
```

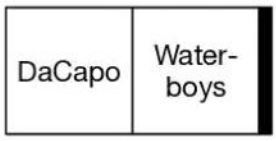


- Lists can be built up, one item at a time, with cons. The simplest list is the empty list, signified by the value empty. Every list we build adds additional items to an empty list.
- Lists can be bound to constants. For example, concerts0 and concerts1
- The empty list is shown as a solid black bar. It appears at the end of every list. Elements of a list are shown as boxes
- Lists are typically drawn with empty on the right. The first item to be added will be right beside it. The most recently added item will be on the left. This also matches the order in which the Racket code is written
- Lists may look like arrays but they are not. The computer can access the ith element of arrays but cannot with lists. Lists are more restrictive but have other advantages.

Another way to write `concerts2`:

Another way to write `concerts2`:

```
(define concerts2alt (cons "DaCapo"
                           (cons "Waterboys"
                                 empty)))
```



A list with one U2 and two DaCapo concerts:

```
(define concerts3 (cons "U2"
                        (cons "DaCapo"
                              (cons "DaCapo"
                                    empty))))
```

Basic lists constructs

- empty: a value representing an empty list
- (cons v lst): consumes a value and a list; produces a new, longer list
- (first lst): consumes a non-empty list; produces the first value
- (rest lst): consumes a non-empty list; produces the same list without the first value
- (empty? v): consumes a value; produces true if it is empty and false otherwise
- (cons? v): consumes a value; produces true if it is a cons value and false otherwise
- (lists? v): equivalent to (or (cons? v) (empty? v))

Extracting values from a list

```
(define clst (cons "U2"
                   (cons "DaCapo" (cons "Waterboys" empty))))
```

First concert:

`(first clst)` ⇒ `"U2"`

Concerts after the first:

`(rest clst)` ⇒ `(cons "DaCapo" (cons "Waterboys" empty))`

Second concert:

`(first (rest clst))` ⇒ `"DaCapo"`

The primary tools for extracting values from a list are first and rest. They may be used in combination to extract any value in the list.

Simple functions on lists

Using these built-in functions, we can write our own simple functions on lists.

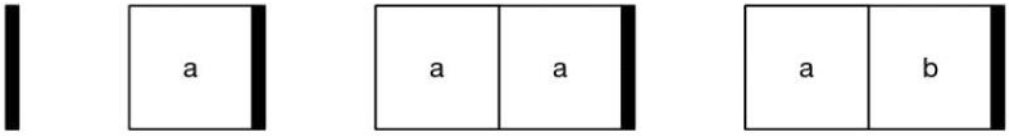
```
;; (next-concert loc) produces the next concert to attend or
;;     the empty string if loc is empty
;; Examples:
(check-expect (next-concert (cons "a" (cons "b" empty))) "a")
(check-expect (next-concert empty) "")

;; next-concert:  (listof Str) → Str
(define (next-concert loc)
  (cond [(empty? loc) ""]
        [else (first loc)]))
```

- This function should handle any list of concerts, which includes the empty list
- Applying first to an empty list produces an error, so we need to test for the empty list and handle it specially
- Note that the examples make use of the “concerts” “a” and “b” to keep them more concise

```
;; (same-consec? loc) determines if next two concerts are the same
;; Examples:
(check-expect (same-consec? empty) false)
(check-expect (same-consec? (cons "a" empty)) false)
(check-expect (same-consec? (cons "a" (cons "a" empty))) true)
(check-expect (same-consec? (cons "a" (cons "b" empty))) false)

;; same-consec?: (listof Str) → Bool
(define (same-consec? loc)
  (and (not (empty? loc))
       (not (empty? (rest loc)))
       (string=? (first loc) (first (rest loc)))))
```



- If a function produces a Boolean value it may be more natural to write the body as a Boolean expression
- A first attempt is to check if the first element of the list (first loc) is the same as the second element, (first (rest loc))
- If there are either zero or one concerts it fails when either first or rest is applied to an empty list
- The function makes use of short-circuit evaluation to avoid those errors. The order of the arguments to and matters. The check for whether loc is empty must come first

- The four images at the bottom of the slide illustrate the four examples

Contracts involving lists

- What is the contract for (next-concert loc)?
- We could use "List" for loc
- However, we almost always need to answer the question "List of what? Numbers? Strings? Any type?"

Notation in contracts

- We will use (listof X) in contracts, where X may be replaced with any type.
- Examples:
  - (listof Str)
  - (listof Num)
  - (listof Bool)
  - (listof Any)
- **Replace X with the most appropriate type available**
- (listof X) always includes the empty list, empty
- The left side of the contract should be as general as possible. If your function can correctly process a list of numbers, use (listof Num) rather than (listof Int) or (listof Nat)
- The right side of the contract as specific as possible. If your function always produces a natural number, use Nat in the contract rather than Int or Num

Formalities

Tuesday, September 22, 2020 13:53

Values

- List values are either
  - empty
  - (cons v l) where v is any Racket value (including list values) and l is a list value (which includes empty)
- Note that values and expressions look very similar

  Value: `(cons 1 (cons 2 (cons 3 empty)))`

  Expression: `(cons 1 (cons (+ 1 1) (cons 3 empty)))`
- Racket list values are traditionally given using **constructor notation** - the same notation we would use to construct value

Expressions

- The following are valid expressions
  - (cons e1 e2) where e1 and e2 are expressions
  - (first e1)
  - (rest e1)
  - (empty? e1)
  - (cons? e1)
  - (list? e1)
- The slide "Basic list constructs" specified that the function cons consumes any value and a list value
- Or contracts:
  - cons: Any (listof Any) -> (listof Any)
  - first: (listof Any) -> Any (requires the list to be non-empty)
  - ...
- cons can also be provided by expressions. For example:

```
(define lst (cons 1 (cons 2 (cons 3 empty))))
(cons (first (rest lst)) (rest lst))
```

  This is a valid expression because (first (rest lst)) and (rest lst) are both valid expressions

Substitution rules

The substitution rules are:

- `(first (cons a b))` ⇒ `a`, where `a` and `b` are values.
- `(rest (cons a b))` ⇒ `b`, where `a` and `b` are values.
- `(empty? empty)` ⇒ `true`.
- `(empty? a)` ⇒ `false`, where `a` is any Racket value other than `empty`.
- `(cons? (cons a b))` ⇒ `true`, where `a` and `b` are values.
- `(cons? a)` ⇒ `false`, where `a` is any Racket value not created using `cons`.

There is no substitution rule for cons because of its role in list values.
If we included it, it would be (cons a b) => (cons a b) where a is a value and b is a list

Data definitions and templates

- The structure of a function often mirrors the structure of the data it consumes. As we encounter more complex data types, we will find it useful to be precise about their structures.
- We will do this by developing **data definitions**
- We can even develop function **templates** based on the data definitions of the values it consumes

List data definition

- Informally: a list of strings is either empty, or consists of a first string followed by a list of strings (the rest of the list)

```
;; A (listof Str) is one of:
;; * empty
;; * (cons Str (listof Str))
```

- This is a **recursive** data definition because the definition refers to itself in at least one case.
- A **base case** does not refer to itself
- We can generalize lists of strings to other types by using an X:

```
;; A (listof X) is one of:
;; * empty
;; * (cons X (listof X))
```

Templates and data-directed design

- The structure of a program often mirrors the structure of the data
- A **template** is a general framework within which we fill in specifics. It is derived from a data definition

Template for processing a (listof X)

```
1  ;; A (listof X) is one of:
2  ;; * empty
3  ;; * (cons X (listof X))
4
5  ;; (listof-X-template lox) PURPOSE
6  ;; Examples:
7  (check-expect (listof-X-template empty) ANSWER)
8  (check-expect (listof-X-template (cons X empty)) ANSWER)
9
10 ;; listof-X-template: (listof X) -> Any
11 (define (listof-X-template lox) ...
12   (cond [(empty? lox) ...]
13         [(cons? lox) ...
14          ... (first lox)...
15          ... (listof-X-template (rest lox)) ...]))
16
17 ;; Tests
```

We start with the data definition for a (listof X)

A function consuming a (listof X) will need to distinguish between these two cases

The ... represents a place to fill in code specific to the problem

Because (rest lox) is of type (listof X), we apply the same computation to it, that is, we apply listof-X-template

- This is the template for a function consuming a (listof X)
- Its form parallels the data definition

Processing lists

Wednesday, September 23, 2020 14:28

Processing lists: how many concerts?

Problem: Write a function to count the number of concerts in a list of concerts.

We begin with writing the purpose, examples, contract, and then copying the template and renaming the function and parameters.

```
;; (count-concerts loc) counts the number of concerts in loc
;; Examples:
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)

;; count-concerts: (listof Str) → Nat
(define (count-concerts loc)
  (cond [(empty? loc) ...]
        [else (... (first loc) ...
                   ... (count-concerts (rest loc)) ...)]))
```

- Three crucial questions to think about functions consuming a list
  - What does the function produce in the base case?
  - What does the function do to the first element in a non-empty list?
  - How does the function combine the value produced from the first element with the value obtained by applying the function to the rest of the list?

> Thinking about list functions

```
;; (count-concerts loc) counts the number of concerts in loc
;; Examples:
(check-expect (count-concerts empty) 0)
(check-expect (count-concerts (cons "a" (cons "b" empty))) 2)

;; count-concerts: (listof Str) → Nat
(define (count-concerts loc)
  (cond [(empty? loc) 0]
        [else (+ 1 (count-concerts (rest loc)))]))
```

- This is a **recursive** function. It uses recursion
- A function is recursive when the body of the function involves an application of the same function.

> Tracing count-concerts

```
(count-concerts (cons "a" (cons "b" empty)))
⇒ (cond [(empty? (cons "a" (cons "b" empty)) 0]
        [else (+ 1 (count-concerts
                    (rest (cons "a" (cons "b" empty))))])
⇒ (cond [false 0]
        [else (+ 1 (count-concerts
                    (rest (cons "a" (cons "b" empty))))])
⇒ (cond [else (+ 1 (count-concerts
                    (rest (cons "a" (cons "b" empty))))])
⇒ (+ 1 (count-concerts (rest (cons "a" (cons "b" empty)))))
⇒ (+ 1 (count-concerts (cons "b" empty)))
⇒ (+ 1 (cond [(empty? (cons "b" empty)) 0]
             [else (+ 1 (count-concerts (rest (cons "b" empty))))]))

⇒ (+ 1 (cond [false 0]
             [else (+ 1 (count-concerts (rest (cons "b" empty))))]))
⇒ (+ 1 (cond [else (+ 1 (count-concerts (rest (cons "b" empty))))]))
⇒ (+ 1 (+ 1 (count-concerts (rest (cons "b" empty)))))
⇒ (+ 1 (+ 1 (count-concerts empty)))
⇒ (+ 1 (+ 1 (cond [(empty? empty) 0]
                  [else (+ 1 (count-concerts (rest empty)))])))
⇒ (+ 1 (+ 1 (cond [true 0][else (+ 1 (count-concerts (rest empty)))])))
⇒ (+ 1 (+ 1 0))
⇒ (+ 1 1)
⇒ 2
```

Condensed traces

- The full trace contains too much detail, so we instead use a **condensed trace** of the recursive function
- It shows the important steps and skips over the trivial details

Termination

- It is important that our functions always **terminate** (stop running and produce an answer)
- Why does count-concerts always terminate?
- There are two conditions
  - **Base case** - produces 0 and immediately terminates
  - **Recursive case** - applies count-concerts to a shorter list. Each recursive application is to a shorter list, which must eventually become empty and terminate

Thinking recursively

- The similarity of recursion to induction suggests a way to develop recursive functions
  - Get the base case right
  - Assume that your function correctly solves a problem of size n (e.g. a list with n items)
  - Figure out how to use that solution to solve a problem of size n+1

Templates

Wednesday, September 23, 2020 21:09

Refining the (listof X) template

- Sometimes, each X in a (listof X) may require further process
- Indicate with a template for X as a helper function
- We assume this generic data definition and template from now on

```
;; listof-X-template: (listof X) → Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [else (... (X-template (first lox)) ...
                   ... (listof-X-template (rest lox)) ...)]))
```

Templates as generalizations

- A template provides the basic shape of the code as suggested by the data definition

Patterns

Wednesday, September 23, 2020 21:15

Patterns of recursion

- The list template has the property that the form of the code matches the form of the data definition
- This is called **simple recursion**

Simple recursion

- In simple recursion, every argument in a recursive function application is either
  - unchanged
  - one step closer to a base case according to the data definition

```
(define (func lst) ... (func (rest lst)) ...) ;; Simple
(define (func lst x) ... (func (rest lst) x) ...) ;; Simple
(define (func lst x) ... (func (process lst) x) ...) ;; NOT Simple
(define (func lst x)
 ... (func (rest lst) (math-function x)) ...) ;; NOT Simple
```

- In the first example, the only argument is one step closer to the base (lst with the first element removed).
  - A list that is one item shorter is not "one step closer to the base case" unless it is shorter because it is missing the first element.
  - Remove the second element does not count.
- The second example is like the first except there is an additional parameter that is passed along unchanged. The count-string is an example
- In the third example, process is meant to capture doing something to lst other than removing the first element.
  In the fourth example, math-function applied to x indicates that x is changed
  In both cases, the function is no longer simple recursion

Lists from lists

Wednesday, September 23, 2020 21:41

Some built-in functions in Racket

- In addition to the ones we've already covered (cons, first, rest, empty?, cons?, list?), the useful ones are append, length, member?, remove, and reverse.
- Don't use append, remove, and reverse until we introduce them.
- Functions that begin with 'c', end with 'r' and have a mixture of 'a's and 'd's in between are historical relics from Lisp. An example is caddr.
- The updated versions in Racket are first, second, third, etc.
- first is used A LOT. second … eighth are used only very occasionally. However, at this point in the course you are only allowed to use first

Producing lists from lists

- Consider negate-list, which consumes a list of numbers and produces the same list with each number negated

```
;; (negate-list lon) produces a list with every number in lon negated
;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
              (cons -2 (cons 12 empty)))

;; negate-list: (listof Num) → (listof Num)
(define (negate-list lon)
  (cond [(empty? lon) ...]
        [else (... (first lon) ... (negate-list (rest lon)) ... )]))
```

1. What should the function produce in the base case?
   This is answered by the first example.
2. What does the function do to the first element in a non-empty list?
   In this case, negate it. We already have (first lon) to get the first element. All we need to do is add (- ...) around it.
3. How does the function combine the value produced from the first element with the value obtained by applying the function to the rest of the list?
   In count-concerts we combined 1 and the recursive application using +. To combine a value with a list, we use cons

```
;; (negate-list lon) produces a list with every number in lon negated
;; Examples:
(check-expect (negate-list empty) empty)
(check-expect (negate-list (cons 2 (cons -12 empty)))
              (cons -2 (cons 12 empty)))

;; negate-list: (listof Num) → (listof Num)
(define (negate-list lon)
  (cond [(empty? lon) empty]
        [else (cons (- (first lon)) (negate-list (rest lon)))]))
```

Non-empty lists

- Some computations make sense only with a **non-empty list** (ne-listof X). For example, finding the maximum of a list of numbers
- A non-empty list of X (ne-listof X) is either
  - (cons X empty)
  - (cons X (ne-listof X))

Design Recipe Refine

Wednesday, September 23, 2020 22:28

Design recipe refinements

- When we introduce new types, we need to include it in the design recipe
- For each new type, place the following someplace between the top of the program and the first place the new type is used
  - The data definition
  - The template derived from that data definition
- Assignments do not need to include the data definition or template for (listof X)
- (ne-listof X) and other types should be included in assignments, unless the assignment states otherwise

Summary: Data definition and template

- Every data definition will have a name (e.g. (listof X)) that can be used in contracts
- In a self-referential data definition
  - at least one clause will use the definition's name to show how to build a larger version of the data (**recursive case**)
  - at least one clause must not use the definition's name (**base case**)

```
;; A (listof X) is one of:
;; * empty
;; * (cons X (listof X))
```

- The template follows directly from the data definition
- The overall shape of a self-referential template will be a cond expression with one clause for each clause in the data definition
- Self-referential data definition clauses lead to recursive expressions in the template
- Base case clauses will not lead to recursion

Strings

Thursday, September 24, 2020 19:07

Strings and list of characters

- Text is usually represented in a computer by strings
- In Racket, a string is a sequence of characters in disguise
- string->list is a built in function to convert a string to an explicit list of characters
- list->string does the reverse. It converts a list of characters into a string
- Racket's notation for the character 'a' is #\a
- The result of evaluating (string->list "test") is the list

```
(cons #\t (cons #\e (cons #\s (cons #\t empty))))
```

Counting characters in a string

> Counting characters in a string

Write a function to count the number of occurrences of a specified character in a string. Start by counting the occurrences in a list of characters.

```
;; (count-char/list ch loc) counts the number of occurrences
;;     of ch in loc.
;; Examples:
(check-expect (count-char/list #\e (string→list "")) 0)
(check-expect (count-char/list #\e (string→list "beekeeper")) 5)
(check-expect (count-char/list
        #\o (cons #\f (cons #\o (cons #\o (cons #\d empty))))) 2)

;; count-char/list: Char (listof Char) → Nat
(define (count-char/list ch loc) ... )
```

This function consumes a character (abbreviated Char) and a (listof Char)

The characters in the list do not have any specific meaning, so we named the paracter based on its structure, loc.

```
;; (count-char/list ch loc) counts the number of occurrences
;;     of ch in loc.
;; Examples:
(check-expect (count-char/list #\e (string→list "")) 0)
(check-expect (count-char/list #\e (string→list "beekeeper")) 5)

;; count-char/list: Char (listof Char) → Nat
(define (count-char/list ch loc)
  (cond [(empty? loc) 0]
        [else (+ (cond [(char=? ch (first loc)) 1]
                       [else 0])
                 (count-char/list ch (rest loc)))]))
```

If the first element mateches the character we are counting, count 1. Else count 0. Whichever it is, add with the result of processing the rest of the list.

```
;; count-char/list: Char (listof Char) -> Nat
```

Another way to structure

```
;; count-char/list: Char (listof Char) -> Nat
(define (count-char/list ch loc)
  (cond
    [(empty? loc)            0]
    [(char=? ch (first loc)) (+ 1 (count-char/list ch (rest loc)))]
    [else                    (count-char/list ch (rest loc))]))
```

Another way to structure this function.

```
;; count-char/list: Char (listof Char) -> Nat
(define (count-char/list ch loc)
  (cond
    [(empty? loc)                0]
    [else
       (cond [(char=? ch (first loc))
              (+ 1 (count-char/list ch (rest loc)))]
             [else
              (count-char/list ch (rest loc))])]))
```

A thrid version. This works but is considered poor style because it can be trivially transformed in to the simpler and more readable code shown above.

Wrapper functions

- A **wrapper function** is a simple function that "wraps" the main function and takes care of details like converting the string to a list
- count-char is a wrapper function for count-char/list

```
;; (count-char ch s) counts the number of occurrences
;;     of ch in s.
;; Examples:
(check-expect (count-char #\e "") 0)
(check-expect (count-char #\e "beekeeper") 5)

;; count-char: Char Str → Nat
(define (count-char ch s)
  (count-char/list ch (string→list s)))
```

count-char basically does the same thing as count-char/list but its parameter is a string.

It calls count-char/list by passing in the list version of the string

- Wrapper functions
  - are short and simple
  - always call another function that does much more
  - set up the appropriate conditions for calling the other function, usually by transforming one or more of its parameters or providing a starting value for one of its arguments

Review

Monday, September 28, 2020 21:04

Review

- Recall the **data definition** for a list:

```
;; A (listof X) is one of:
;; ⋆ empty
;; ⋆ (cons X (listof X))
```

- Given (cons 5 (cons 3 (cons 29 empty))) we can use the data definition to work backwards, proving that it is a (listof Int).
- In fact, with the built-in predicate integer? we can write a function that determines whether its argument is a list of integers

```
;; (listof-int? x) produces true if x is (listof Int) and
;;     false otherwise.
;; Examples:
(check-expect (listof-int? empty) true)
(check-expect (listof-int? (cons 1 (cons 2 empty))) true)
(check-expect (listof-int? (cons "a" empty)) false)
(check-expect (listof-int? "A string") false)

;; listof-int?:  Any -> Bool
(define (listof-int? x)
  (cond [(empty? x)  true]
        [(cons? x)   (and (integer? (first x))
                          (listof-int? (rest x)))]
        [else        false]))
```

- Recall the template for a list:

```
;; listof-X-template: (listof X) → Any
(define (listof-X-template lst)
  (cond [(empty? lst) ...]
        [else (... (first lst) ...
                   (listof-X-template (rest lst)) ...)]))
```

- We can repeat this reasoning on a recursive definition of natural numbers to obtain a template.

Data definition and template

Monday, September 28, 2020 21:41

Natural numbers:

- Data definition:

```
;; A Nat is one of:
;; * 0
;; * (add1 Nat)
```

- add1 is the built-in function that adds 1 to its argument
- The natural numbers start at 0 in computer science
- Template:

```
;; nat-template: Num → Any
(define (nat-template n)
  (cond [(zero? n) ...]
        [else (... n ...
               ... (nat-template (sub1 n)) ...)]))
```

- Suppose we have a natural number n.
- The test (zero? n) tells us which case applies
- If (zero? n) is false, then n has the value (add1 k) for some k
- To compute k, we subtract 1 from n, using the built-in sub1 function
- Because the result (sub1 n) is a natural number, we recursively apply the function

Example: a decreasing list

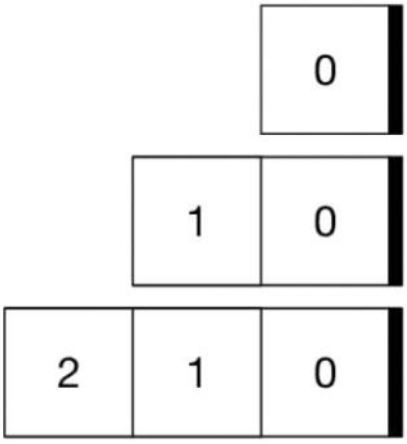
- Goal: countdown, which consumes a natural number n and produces a decreasing list of all natural numbers less than or equal to n

Goal: `countdown`, which consumes a natural number *n* and produces a decreasing list of all natural numbers less than or equal to *n*.

```
(countdown 0) ⇒ (cons 0 empty)

(countdown 1) ⇒ (cons 1 (cons 0 empty))

(countdown 2) ⇒ (cons 2 (cons 1 (cons 0 empty)))
```



With these examples, we proceed by filling in the template.

- We have some crucial questions to answer:
  - What do we produce in the base case?

(cons 0 empty))
    - In the recursive case, what do we do to transform n?
      In this case, we does nothing to n
    - What is the result of processing (f (sub1 n)) recursively?
      (countdown (sub1 n)) should be the list of natural numbers from n-1 down to n
    - How do we combine steps 2 and 3 to obtain the result for (f n)?
      We cons the result of step 2 (n) onto the result of step 3 (countdown (sub1 n))
- Function Implementation:

```
;; (countdown n) produces a decreasing list of Nats from n to 0
;; Example:
(check-expect (countdown 0) (cons 0 empty))
(check-expect (countdown 2) (cons 2 (cons 1 (cons 0 empty))))

;; countdown: Nat → (listof Nat)
(define (countdown n)
  (cond [(zero? n) (cons 0 empty)]
        [else (cons n (countdown (sub1 n)))]))
```

Intervals

Monday, September 28, 2020 22:38

Intervals of natural numbers

- The symbol Z is often used to denote the integers
- We can add subscripts to define **subsets** of the integers (aka **intervals**)

  For example, $\mathbb{Z}_{\geq 0}$ defines the non-negative integers, also known as the natural numbers.

  Other examples: $\mathbb{Z}_{>4}$, $\mathbb{Z}_{<-8}$, $\mathbb{Z}_{\leq 1}$.

Example: $Z_{>=7}$

- If we change the base case test from (zero? n) to (= n 7), we can stop the countdown at 7
- This corresponds to the following definition

```
;; An integer in Z≥7 is one of:
;; ⋆ 7
;; ⋆ (add1 Z≥7)
```

- We use this data definition as a guide when writing functions, but in practice we use a requires section in the contract
- countdown-to-7

```
;; (countdown-to-7 n) produces a decreasing list from n to 7
;; Example:
(check-expect (countdown-to-7 9) (cons 9 (cons 8 (cons 7 empty))))

;; countdown-to-7: Nat → (listof Nat)
;;   requires: n ≥ 7
(define (countdown-to-7 n)
  (cond [(= n 7) (cons 7 empty)]
        [else (cons n (countdown-to-7 (sub1 n)))]))
```

Generalizing countdown

- We can generalize countdown by providing the base value (e.g. 0 or 7) as a second parameter b (the base)
- The parameter b has to be passed unchanged in the recursion

```
;; (countdown-to n base) produces a decreasing list from n to base
;; Example:
(check-expect (countdown-to 4 2) (cons 4 (cons 3 (cons 2 empty))))

;; countdown-to: Int Int → (listof Int)
;;   requires: n >= base
(define (countdown-to n base)
  (cond [(= n base) (cons base empty)]
        [else (cons n (countdown-to (sub1 n) base))]))
```

- This function also works if the inputs are negative numbers, as long as n>=base

```
(countdown-to 1 -2)
⇒ (cons 1 (cons 0 (cons -1 (cons -2 empty))))
```

Counting up

Monday, September 28, 2020 23:03

Counting up

- What if we want an increasing count?
- Consider the non-positive integers Z<=0

```
;; A integer in ℤ≤0 is one of:
;; ⋆ 0
;; ⋆ (sub1 ℤ≤0)
```

- Examples: -1 is (sub1 0), -2 is (sub 1 (sub1 0))
- If an integer i is of the form (sub1 k), then k is equal to (add1 i)

Template

- Notice the additional requires section

```
;; nonpos-template: Int → Any
;; requires: n ≤ 0
(define (nonpos-template n)
  (cond [(zero? n) ...]
        [else (... n ...
               ... (nonpos-template (add1 n)) ...)]))
```

- countup:

```
;; (countup n) produces an increasing list from n to 0
;; Example:
(check-expect (countup -2) (cons -2 (cons -1 (cons 0 empty))))

;; countup: Int → (listof Int)
;;   requires: n <= 0
(define (countup n)
  (cond [(zero? n) (cons 0 empty)]
        [else (cons n (countup (add1 n)))]))
```

- As before, we can generalize this to counting up to b, by introducing b as a second parameter in a template

```
;; (countup-to n base) produces an increasing list from n to base
;; Example:
(check-expect (countup-to 6 8) (cons 6 (cons 7 (cons 8 empty))))

;; countup-to: Int Int → (listof Int)
;;   requires: n <= base
(define (countup-to n base)
  (cond [(= n base) (cons base empty)]
        [else (cons n (countup-to (add1 n) base))]))
```

- **You may not use reverse on assignments unless stated otherwise**

Sorting

Wednesday, September 30, 2020 19:24

Filling in the list template

```
;; (sort lon) sorts the elements of lon in non-decreasing order
(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) ...)

;; sort: (listof Num) → (listof Num)
(define (sort lon)
  (cond [(empty? lon) ...]
        [else (... (first lon)
                   (sort (rest lon)) ...)]))
```

If the list lon is empty, so is the result.
Otherwise, the template suggests doing something with the first element of the list, and the sorted version of the rest

```
;; (sort lon) sorts the elements of lon in non-decreasing order
(check-expect (sort (cons 2 (cons 0 (cons 1 empty)))) ...)

;; sort: (listof Num) → (listof Num)
(define (sort lon)
  (cond [(empty? lon) empty]
        [else (insert (first lon)
                      (sort (rest lon)))]))
```

insert is a recursive helper function that consumes a number and a sorted list, and inserts the number into the sorted list

The helper function insert

- We again use the list template for insert

```
;; (insert n slon) inserts the number n into the sorted list slon...
;; Examples:
(define test-result (cons 1 (cons 2 (cons 3 empty))))
(check-expect (insert 1 empty) (cons 1 empty))
(check-expect (insert 1 (cons 2 (cons 3 empty))) test-result)
(check-expect (insert 2 (cons 1 (cons 3 empty))) test-result)

;; insert: Num (listof Num) → (listof Num)
;;     requires: slon is sorted in non-decreasing order
(define (insert n slon)
  (cond [(empty? slon) ...]
        [else (... (first slon) ...
                   (insert n (rest slon)) ...)]))
```

- If slon is empty, the result is the list containing just n
- If slon is not empty, another conditional expression is needed

- n is the first number in the result if it is less than or equal to the first number in slon (<= n (first slon))
- Otherwise, the first number in the result is the first number in slon, and the rest of the result is what we get when we insert n into (rest slon) (insert n (rest slon))

```
(define (insert n slon)
  (cond [(empty? slon) (cons n empty)]
        [(<= n (first slon)) (cons n slon)]
        [else (cons (first slon) (insert n (rest slon)))]))
```

```
(insert 4 (cons 1 (cons 2 (cons 5 empty))))
⇒ (cons 1 (insert 4 (cons 2 (cons 5 empty))))
⇒ (cons 1 (cons 2 (insert 4 (cons 5 empty))))
⇒ (cons 1 (cons 2 (cons 4 (cons 5 empty))))|
```

- This algorithm is known as **insertion sort**

List abbreviations

Wednesday, September 30, 2020 20:54

List abbreviations

- The expression

  ```
  (cons exp1 (cons exp2 (... (cons expn empty)...)))
  ```

  can be abbreviated as

  ```
  (list exp1 exp2 ... expn)
  ```

- The result of (sort (cons 4 (cons 2 (cons 3 (cons 5 empty))))) can be expressed as (list 1 2 4 5). The application itself can be expressed as (sort (list 4 2 1 5))
- Difference between cons and list:
  - A list built with cons will explicitly show the empty at the end of the list
  - A list built with list will not show the empty
- We use list to construct a list of fixed size. And we use cons to construct a list from one new element (first) and a list of arbitrary size
- (second my-list) is an abbreviation for (first (rest my-list)). third, fourth and so on up to eighth are also defined. **Use these sparingly to improve readability.**

Exercise 2

- You want to add one more element to the list `lst`. Do you use `(cons elem lst)` or `(list elem lst)`? What's the difference between them?
- Why is `(list 1 2)` legal but `(cons 1 2)` is not?
- What's the difference between `(cons 1 empty)` and `(list 1 empty)`?

1. (cons elem lst) because lst is a list of arbitrary size
2. cons takes any value and a list value
3. (cons 1 empty) has one element, (list 1 empty) has two elements

Exercise 3

What is the length of:

```
(list (list "hat" "boots") "coat"
      (list 32.3 (list "mitts")) empty "scarf")
```

Determine the answer by hand, then use the `length` function to check your answer.

length = 5

Lists of lists

Friday, October 9, 2020 23:22

Lists containing lists

Here are two different two-element lists.

We now know two different ways to construct these lists:

```
(cons 1 (cons 2 empty))
(cons 3 (cons 4 empty))
```

OR

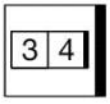
```
(list 1 2)
(list 3 4)
```

Lists can contain anything, including other lists, at which point these abbreviations can improve readability.

- Remember that in (cons v lst), v is a value. Any value.
- (cons 1 (cons 2 empty)) is a value. As a value, it can be put into a list just like any other value

> Lists containing lists

Here is a one-element list whose single element is one of the two-element lists we saw above.



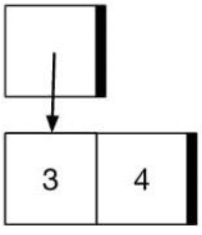
As before, we now know two different ways we could construct this.

```
(cons (cons 3 (cons 4 empty))
      empty)
```

OR

```
(list (list 3 4))
```

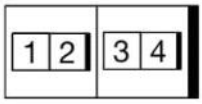
When the thing a list contains is complicated, we may draw an arrow to it, as shown on the right. This visualization represents the same list as the one above.



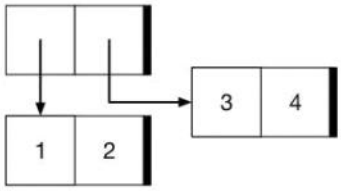
- Lists that contain lists are hard to draw when the whole value (a list) goes in a box, so we're introducing a new diagramming convention (bottom diagram)

> Lists containing lists

We can create a two-element list, each of whose elements is itself a two-element list. These are two different visualizations of **the same list**.



OR



Such a list can be created in code two different ways:

```
(cons (cons 1 (cons 2 empty))
      (cons (cons 3 (cons 4 empty))
            empty))
```

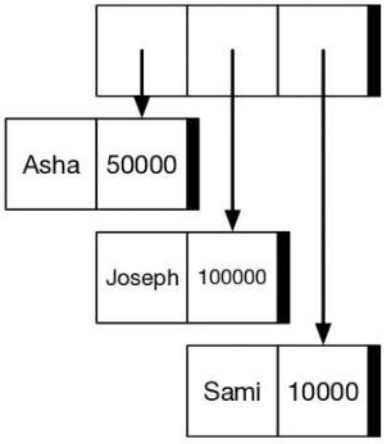
OR

```
(list (list 1 2)
      (list 3 4))
```

Clearly, the abbreviations are more expressive.

> Example: processing a payroll

A company needs to process its payroll – a list of employee names and their salaries. It produces a list of each employee name and the tax owed. The tax owed is computed with `tax-payable` from Module 04.

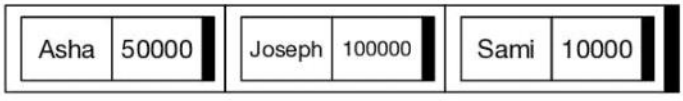


Payroll:

```
(list (list "Asha" 50000)
      (list "Joseph" 100000)
      (list "Sami" 7000))
```

TaxRoll:

```
(list (list "Asha" 7750)
      (list "Joseph" 18250)
      (list "Sami" 1050))
```



» Data definitions

```
;; A Payroll is one of:
;; * empty
;; * (cons (list Str Num) Payroll)


;; A TaxRoll is one of:
;; * empty
;; * (cons (list Str Num) TaxRoll)
```

Note the use of `(list Str Num)` rather than `(listof X)`.

- A payroll is the same as a (listof X) where X is a two-element list that we represent with (list Str Num). The first element is a string and the second is a number. Because the data definition has been made more specific, we give it a specific name: Payroll.
- Intuitively, an empty list is a Payroll. If you want a longer Payroll, cons the name of an employee and their salary onto a Payroll.
- Recap of (list …) vs (listof …)
  - Use (list …) for a fixed-length list.
  - (list …) will have one type parameter for each element in the list
  - Use (listof ...) when the length of the list is unknown
  - (listof …) will have exactly one argument: the type that every element in the list will have
  - They are **not** interchangeable

» Template

```
;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) ... (first pr) ...
                    ... (payroll-template (rest pr)) ...]))
```

A payroll is just a list, so this looks exactly like the `(listof X)` template – **so far...**

- The list's first item is known to be of the form (list Str Num)
- Reflect that fact in the template
  - reminds us of all the data available to us
  - allow us to access the parts of the sublist

» Template

```
;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) (... (first (first pr)) ...
                    ... (first (rest (first pr))) ...
                    ... (payroll-template (rest pr)) ...)]))
```

Some short helper functions will make our code more readable.

» Template

```
;; (name lst) produces the first item from lst -- the name.
;; name: (list Str Num) → Str
(define (name lst) (first lst))
;; (amount lst) produces the second item from lst -- the amount.
;; amount: (list Str Num) → Num
(define (amount lst) (first (rest lst)))

;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) (... (name (first pr)) ...
                         ... (amount (first pr)) ...
                         ... (payroll-template (rest pr)) ...)]))
```

extract elements from the sublists

» Start design recipe; fill in template

```
;; (compute-taxes payroll) calculates the tax owed for each
;; employee/salary pair in the payroll.
(check-expect (compute-taxes test-payroll) test-taxes)

;; compute-taxes: Payroll → TaxRoll
(define (compute-taxes payroll)
  (cond [(empty? payroll) ...]
        [(cons? payroll) (... (name (first payroll)) ...
                             ... (amount (first payroll)) ...
                             ... (compute-taxes (rest payroll)) ...)]))
```


Steps in the design recipe:

- Write the purpose
- Write some examples
- Write the header (rename payroll-template to something more problem-specific) and contract (note the use of Payroll and Taxroll)
- Refine the purpose

» Finish `compute-taxes`

```
;; (compute-taxes payroll) calculates the tax owed for each
;; employee/salary pair in the payroll.
(check-expect (compute-taxes test-payroll) test-taxes)

;; compute-taxes: Payroll → TaxRoll
(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [(cons? payroll)
         (cons (list (name (first payroll))
                     (tax-payable (amount (first payroll))))
               (compute-taxes (rest payroll)))]))
```


1. What do you do in the base case?
   There are no taxes owed if no one earned a salary. The contract says to produce a (listof X). empty fits both observations.
2. How do you transform the first element on the list?
   Compute the taxes on the salary. Put that number together with the name into a two-element list
3. What value is produced by the recursive processing of the rest of the list?
   A list of taxes owed by the employees on the rest of the list
4. How to combine the value produced in 2 with the value produced in 3?
   cons the newly calculated taxes owed to the beginning of the list

» Alternate solution

```
(define (compute-taxes-alt payroll)
  (cond [(empty? payroll) empty]
        [(cons? payroll) (cons (sr→tr (first payroll))
                               (compute-taxes-alt (rest payroll)))]))

;; (sr→tr salary-rec) consumes a salary record and produces the
;; corresponding tax record
;; sr→tr: (list Str Num) → (list Str Num)
(define (sr→tr salary-rec)
  (list (name salary-rec) (tax-payable (amount salary-rec))))
```


- A useful strategy is to break the problem into two parts
  - the part that handles the list
  - the part that handles one item from the list
- The overall structure of the code is simply the (listof-X-template) and allows one to focus on the distinct part of how one item is handled

» Alternate templates leading to the second solution

```
;; (payroll-template pr)
;; payroll-template: Payroll → Any
(define (payroll-template pr)
  (cond [(empty? pr) ...]
        [(cons? pr) (... (salary-rec-template (first pr)) ...
                         ... (payroll-template (rest pr)) ...)]))

(define (salary-rec-template sr) (... (name sr) ... (amount sr) ...))
```

Different kinds of lists

When we introduced lists in module 06, the items they contained were not lists. These were **flat lists**.

We have just seen **lists of lists**. A `Payroll` is a list containing a two-element flat list.

In later lecture modules, we will use lists containing unbounded flat lists.

We will also see **nested lists**, in which lists may contain lists that contain lists, and so on to an arbitrary depth.

Dictionaries

Friday, October 9, 2020 23:22

> Example dictionaries

More generally, a **dictionary** contains a number of unique **keys**, each with an associated **value**.

Examples:

- A dictionary: keys are words; values are definitions.
- Your contacts list: keys are names; values are telephone numbers.
- Course marks: keys are userids; values are marks.
- Stocks: keys are symbols; values are prices.

Many two-column tables can be viewed as dictionaries. The previous examples can all be viewed as two-column tables.

- An important aspect of the definition is that keys are unique.
- Given a key, we can look it up in a dictionary and get at most one value (it's possible that the key is not here)
- Values, on the other hand, may be duplicated

> Dictionary operations

What *operations* might we wish to perform on dictionaries?

- **lookup**: given a key, produce the corresponding value — sometimes called find or search
- **add**: add a (key,value) pair to the dictionary — sometimes called insert
- **remove**: given a key, remove it and its associated value

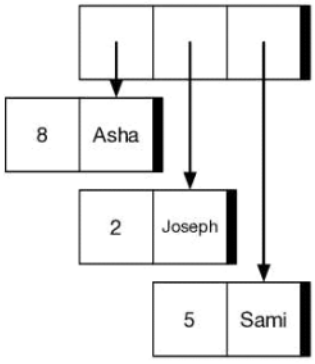
> Association lists

One simple solution uses an **association list**, which is just a list of (key, value) pairs.
We store the pair as a two-element list. For simplicity, we will use natural numbers as keys and strings as values.

```
;; An association list (AL) is one of:
;; * empty
;; * (cons (list Nat Str) AL)
;;        Requires: each key (Nat) is unique
```

Example:

```
(list (list 8 "Asha")
      (list 2 "Joseph")
      (list 5 "Sami"))
```



- The advantage of an association list is that it is simple to implement.
- The disadvantage is being increasingly slow as the dictionary gets larger (has more entries)

> Association lists

We can create association lists based on other types for keys and values. We use `Nat` and `Str` here just to provide a concrete example.

Since we have a data definition, we could use `AL` for the type of an association list, as given in a contract.

Another alternative is to use `(listof (list Nat Str))`.

- We could use (listof (list Num Str)) every place we need the data type, principally in contracts
- However, the term AL or "association list" helps the reader understand the intent of the parameter is probably to look something up. The listof format does not convey that meaning
- This form also ignores the requirement that keys are unique

> Constructing the `al-template`

We can use the data definition to produce a template.

```
;; al-template: AL → Any
(define (al-template alst)
  (cond [(empty? alst) ...]
        [else (... (first (first alst)) ... ; first key
                   (second (first alst)) ... ; first value
                   (al-template (rest alst)))]))
```

A better implementation (except for the lack of documentation):

```
(define (key kv) (first kv))
(define (val kv) (second kv))
(define (al-template alst)
  (cond [(empty? alst) ...]
        [else (... (key (first alst)) ... (val (first alst)) ...
                   (al-template (rest alst)))]))
```

Helper functions make the code more readable in the improved version

> Lookup operation

Recall that `lookup` consumes a key and a dictionary (association list) and produces the corresponding value when it's found. But what should `lookup` produce if it fails?

When the key is not found in the association list, we can not produce a string. Every string, even `""`, is a valid value and might be the result of a successful lookup. The "not found" condition needs to be distinguishable, from successful searches. We'll use `false` to indicate failure.

```
(check-expect (lookup 2 (list (list 8 "Asha")
                              (list 2 "Joseph")
                              (list 5 "Sami")) "Joseph")
(check-expect (lookup 1 (list (list 8 "Asha")
                              (list 2 "Joseph")
                              (list 5 "Sami")) false)
```

> `lookup` implementation

```
(define (key kv) (first kv))
(define (val kv) (second kv))

;; (lookup-al k alst) produces the value corresponding to key k,
;;     or false if k not present

;; lookup-al:
(define (lookup-al k alst)
  (cond [(empty? alst) false]
        [(= k (key (first alst))) (val (first alst))]
        [else (lookup-al k (rest alst))]))
```

What is the contract for `lookup-al`?

We need a way to indicate that it can produce either a string or `false`.

- There are two base cases just like the insert function.
- For insert, the two base conditions were
  - when we got to the end of the list
  - when the item we were inserting belonged at the beginning of the list
- For lookup the two base cases are
  - when we get to the end of the list
  - when we find what we are looking for

> `(anyof ...)` notation in contracts

Use `(anyof X Y ...)` to mean any of the listed types or values.

Examples:

- `(anyof Num Str)` number or string
- `(anyof Str Num Bool)` string, number or bool
- `(anyof 1 2 3)` 1, 2 or 3
- `(listof (anyof Str false))`

```
;; foo: Num → (anyof Str Bool Num)
(define (foo x)
  (cond [(< x 0) "negative"]
        [(= x 0) false]
        [(= x 1) true]
        [else x]))
```

Dictionaries: summary

We will leave the `add` and `remove` functions as exercises.

The association list solution is simple enough that it is often used for small dictionaries.

For a large dictionary, association lists are inefficient. For example, consider the case where the key is not present and the whole list must be searched.

In a future module, we will impose structure to improve this situation.

Exercise 4

- Implement `add`.
- Implement `remove`.
- How could you improve the performance of an association list if the dictionary is large (using techniques from this lecture module)? Is there a way to avoid search the *whole* list most of the time?

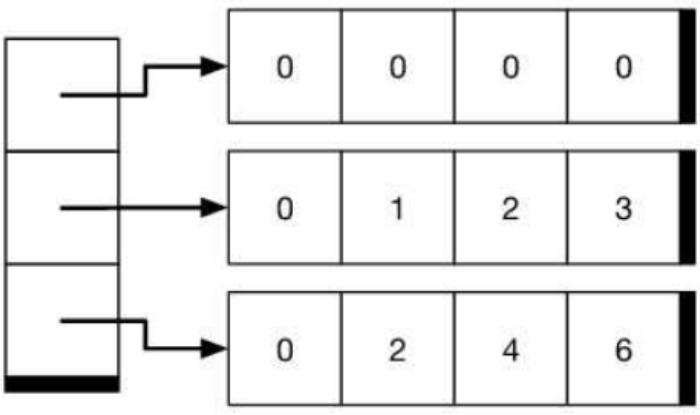
2D data

Saturday, October 10, 2020 10:40

Two-dimensional data

- Another use of lists of lists is to represent a two-dimensional table.
- For example, a multiplication table:

```
(mult-table 3 4) ⇒
(list (list 0 0 0 0)
      (list 0 1 2 3)
      (list 0 2 4 6))
```



- The c^{th} entry of the r^{th} row (numbering from 0) is r x c

Make one row

- Make one row of the table but counting the columns from 0 up to nc, doing the required multiplication for each one

```
;; (cols-to c r nc) produces entries c...(nc-1) of rth row of mult. table
;; Example:
(check-expect (cols-to 0 3 5) (list 0 3 6 9 12))
(check-expect (cols-to 0 4 5) (list 0 4 8 12 16))

;; cols-to: Nat Nat Nat → (listof Nat)
(define (cols-to c r nc)
  (cond [(>= c nc) empty]
        [else (cons (* r c) (cols-to (add1 c) r nc))]))
```

- cols-to constructs one row for the multiplication table by calculating what value each column should have

Put multiple rows together

```
;; (mult-table nr nc) produces multiplication table
;;     with nr rows and nc columns
;; Example:


(define (mult-table nr nc)
  (rows-to 0 nr nc))

;; (rows-to r nr nc) produces mult. table, rows r...(nr-1)
;; rows-to: Nat Nat Nat → (listof (listof Nat))
(define (rows-to r nr nc)
  (cond [(>= r nr) empty]
        [else (cons (cols-to 0 r nc) (rows-to (add1 r) nr nc))]))
```

- In rows-to, the counter is called r. We make a whole row with cols-to and cons that row onto the list. The result is a list of lists.
- mult-table is a wrapper function that calls rows-to with the correct values.

Processing two lists

Saturday, October 10, 2020 12:45

Processing two lists simultaneously

- We now look at a more complicated recursion, namely writing functions which consume two lists (or two data types, each of which has a recursive definition)
- We will distinguish three different cases, and look at them in order of complexity.
- The simplest case is when one of the lists does not require recursive processing

Processing two lists simultaneously (annotation)

- Two parameters, each of which is a list

```
;; twolist-template: (listof X) (listof X) -> Any
(define (twolist-template lst1 lst2) ... )
```

- Each of those two lists might be empty or not. Therefore, we might have something like this:

```
(define (twolist-template lst1 lst2)
  (cond [(and (empty? lst1) (empty? lst2)) ...]
        [(and (empty? lst1) (cons? lst2)) ...]
        [(and (cons? lst1) (empty? lst2)) ...]
        [(and (cons? lst1) (cons? lst2)) ...]))
```

- We can actually break it down into three cases
  - Process one list; the other goes along for the ride
  - Both lists are the same length; one element from each is processed at each step
  - The general case: unequal lengths, process one or the other or both

Case 1: processing just one list

- In the first case of processing two lists, we only actually process one. The second one just goes along for the ride.
- That means that of the four tests identified above, we only need to consider two: whether the first list is empty or not
- As an example, consider the function my-append

> Case 1: processing just one list

As an example, consider the function `my-append`.

```
;; (my-append lst1 lst2) appends lst2 to the end of lst1
;; Examples:
(check-expect (my-append empty (list 'a 'b 'c)) (list 'a 'b 'c))
(check-expect (my-append (list 3 4) (list 1 2 5)) (list 3 4 1 2 5))

;; my-append: (listof Any) (listof Any) → (listof Any)
(define (my-append lst1 lst2)
  ...)
```


» my-append

```
(define (my-append lst1 lst2)
  (cond [(empty? lst1) lst2]
        [else (cons (first lst1)
                    (my-append (rest lst1) lst2))]))
```

The code only does simple recursion on `lst1`.

The parameter `lst2` is “along for the ride”.

`append` is a built-in function in Racket.

» A condensed trace

```
(my-append (list 1 2 3) (list 4 5 6))
⇒ (cons 1 (my-append (list 2 3) (list 4 5 6)))
⇒ (cons 1 (cons 2 (my-append (list 3) (list 4 5 6))))
⇒ (cons 1 (cons 2 (cons 3 (my-append (list ) (list 4 5 6)))))
⇒ (cons 1 (cons 2 (cons 3 (list 4 5 6)))
```

The last line is the same as
`(cons 1 (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty))))))`.
That's the same as `(list 1 2 3 4 5 6)`.

Case 2: processing in lockstep

- To process two lists lst1 and lst2 in lockstep, they must be the same length and be consumed at the same rate
- lst1 is either empty or a cons, and the same is true of lst2. This means that there are four possibilities in total.
- However, because the two lists must have the same length, (empty? lst1) is true if and only if (empty? lst2) is true
- This means that only two of the possibilities are valid
- Example: dot product - multiplying entries in corresponding positions (first with first, second with second and so on) and sum the results

» dot-product

```
;; (dot-product lon1 lon2) computes the dot product
;;       of vectors lon1 and lon2
(check-expect (dot-product empty empty) 0)
(check-expect (dot-product (list 2) (list 3)) 6)
(check-expect (dot-product (list 2 3 4 5) (list 6 7 8 9))
              (+ 12 21 32 45))

;; dot-product: (listof Num) (listof Num) → Num
;;    requires: lon1 and lon2 are the same length
(define (dot-product lon1 lon2)
  (cond
    [(empty? lon1) 0]
    [else (+ (* (first lon1) (first lon2))
             (dot-product (rest lon1) (rest lon2)))]))
```


» A condensed trace

```
(dot-product (list 2 3 4)
             (list 5 6 7))
⇒ (+ 10 (dot-product (list 3 4)
                     (list 6 7)))
⇒ (+ 10 (+ 18 (dot-product (list 4)
                           (list 7))))
⇒ (+ 10 (+ 18 (+ 28 (dot-product (list )
                                 (list )))))
⇒ (+ 10 (+ 18 (+ 28 0)))
⇒ (+ 10 (+ 18 28))
⇒ (+ 10 46)
⇒ 56
```


Case 3: processing at different rates

- The third case is the most general where the lists may be of different lengths and may be processed at different rates
- As a result, either one or both could be empty
- If the two lists being consumed are of different lengths, four possibilities are possible

```
(and (empty? lon1) (empty? lon2))
(and (empty? lon1) (cons? lon2))
(and (cons? lon1) (empty? lon2))
(and (cons? lon1) (cons? lon2))
```

- The first possibility is a base case; the second and the third may or may not be

» Refining the template

```
(define (twolist-template lon1 lon2)
  (cond
    [(and (empty? lon1) (empty? lon2)) ...]
    [(and (empty? lon1) (cons? lon2))
     (... (first lon2) ... (rest lon2) ...)]
    [(and (cons? lon1) (empty? lon2))
     (... (first lon1) ... (rest lon1) ...)]
    [(and (cons? lon1) (cons? lon2))
     ??? ]))
```

The second and third possibilities may or may not require recursion.

The fourth possibility definitely requires recursion, but its form is unclear.

- In the second and the third possibilities where one list is empty, we extract the first element and the rest of the list for each non-empty list
- When one list is empty, it could be that all we need is the non-empty list without change. In this case no recursion is needed
- But it could be that the non-empty list needs further processing, leading to recursion

» Further refinements

There are many different possible natural recursions for the last **cond** answer ???:

```
(... (first lon1)
     (twolist-template (rest lon1) lon2))

(... (first lon2)
     (twolist-template lon1 (rest lon2)))

(... (first lon1) ... (first lon2)
     (twolist-template (rest lon1) (rest lon2)))
```

Which of these is appropriate depends on the specific problem we're trying to solve and will require further reasoning.

Example: merging two sorted lists

- Design a function merge that consumes two lists
- Each list is sorted in ascending order (no duplicate values)
- merge will produce one list containing all elements, also in ascending order
- As an example:

  ```
  (merge (list 1 8 10) (list 2 4 6 12)) ⇒ (list 1 2 4 6 8 10 12)
  ```

- We need more examples to see how to proceed

» Example: merging two sorted lists

```
;; Base cases
(check-expect (merge empty empty)            empty)

(check-expect (merge empty (list 2 6 9))     (list 2 6 9))

(check-expect (merge (list 1 3) empty)       (list 1 3))

;; Recursive cases
(check-expect (merge (list 1 4) (list 2))    (list 1 2 4))

(check-expect (merge (list 3 4) (list 2))    (list 2 3 4))
```


- If both lists are non-empty, the first element of the merged list would be the smaller of (first lon1) and (first lon2)
- If (first lon1) is smaller, then the rest of the answer is the result of merging (rest lon1) and lon2
- If (first lon2) is smaller, then the rest of the answer is the result of merging lon1 and (rest lon2)

» Merge code

```
(define (merge lon1 lon2)
  (cond [(and (empty? lon1) (empty? lon2)) empty]
        [(and (empty? lon1) (cons? lon2)) lon2]
        [(and (cons? lon1) (empty? lon2)) lon1]
        [(and (cons? lon1) (cons? lon2))
         (cond [(< (first lon1) (first lon2))
                (cons (first lon1) (merge (rest lon1) lon2))]
               [else (cons (first lon2) (merge lon1 (rest lon2)))])]))
```

» A condensed trace

```
(merge (list 3 4)
       (list 2 5 6))
⇒ (cons 2 (merge (list 3 4)
                 (list 5 6))))
⇒ (cons 2 (cons 3 (merge (list 4)
                         (list 5 6))))
⇒ (cons 2 (cons 3 (cons 4 (merge empty
                                 (list 5 6)))))
⇒ (cons 2 (cons 3 (cons 4 (cons 5 (cons 6 empty)))))
```


- Note that in the first question/answer pair, lon2 is empty, and the answer produced could be replaced with lon2
- The first two question/answer pairs are then

```
[(and (empty? lon1) (empty? lon2)) lon2]
[(and (empty? lon1) (cons? lon2)) lon2]
```

- Since both produce lon2 regardless of whether lon2 is empty or not, the two pairs can be replaced with just

```
[(empty? lon1) lon2]
```

- The last question/answer pair can be replaced with else. We then have [else (cond …)] which is considered bad style. The question/answer pairs of the inner cond can be promoted to the outer cond
- Together, these transformation result in the following:

```
(define (merge lon1 lon2)
  (cond [(empty? lon1) lon2] ; first two cases
        [(empty? lon2) lon1] ; third case
        [(< (first lon1) (first lon2))
         (cons (first lon1) (merge (rest lon1) lon2))]
        [else (cons (first lon2) (merge lon1 (rest lon2)))]))
```

Testing list equality

- Check whether two lists of numbers are equal (same elements, same order)

Testing list equality

```
;; (list=? lst1 lst2) determines if lst1 and lst2 are equal
;; list=?: (listof Num) (listof Num) → Bool
(define (list=? lst1 lst2)
  (cond
    [(and (empty? lst1) (empty? lst2)) ...]
    [(and (empty? lst1) (cons? lst2))
     (... (first lst2) ... (rest lst2) ...)]
    [(and (cons? lst1) (empty? lst2))
     (... (first lst1) ... (rest lst1) ...)]
    [(and (cons? lst1) (cons? lst2))
     (???)]))
```


- Two empty lists are equal (return true)
- If one list is empty and the other is not, they are not equal.
- If both of them are non-empty, then their first elements must equal

List equality code

```
(define (list=? lst1 lst2)
  (cond
    [(and (empty? lst1) (empty? lst2)) true]
    [(and (empty? lst1) (cons? lst2)) false]
    [(and (cons? lst1) (empty? lst2)) false]
    [(and (cons? lst1) (cons? lst2))
     (and (= (first lst1) (first lst2))
          (list=? (rest lst1) (rest lst2)))]))
```

Some further simplifications are possible.

Exercise 7

Like `merge`, the code for `list=?` can be transformed in various ways. Each problem stands alone, starting with the code on the previous slide. Like `merge`, whether the result is "better" or not depends on the metrics used.

1. Combine the second and third question/answer pairs.
2. Combine the first and second question/answer pairs; simplify the third.
3. Use **`else`**.
4. Combine 1 and 3.
5. Combine 2 and 3.
6. Get rid of the **`cond`** completely.

Processing a list and a number

Monday, October 12, 2020 10:10

Example: Does an item appear at least n times in this list?

> Examples for `at-least?`

```
;; (at-least? n elem lst) determines if elem appears
;;       at least n times in lst.
(check-expect (at-least? 0 'red (list 1 2 3)) true)
(check-expect (at-least? 3 "hi" empty) false)
(check-expect (at-least? 2 'red (list 'red 'blue 'red 'green)) true)
(check-expect (at-least? 3 'red (list 'red 'blue 'red 'green)) false)
(check-expect (at-least? 1 7 (list 5 4 0 5 3)) false)

;; at-least?: Nat Any (listof Any) → Bool
(define (at-least-template? n elem lst)
```


- The recursion involves the parameters n and lst, giving four possibilities

```
(cond [(and (zero? n) (empty? lst)) ...]
      [(and (zero? n) (cons? lst)) ...]
      [(and (> n 0) (empty? lst)) ...]
      [(and (> n 0) (cons? lst)) ...]))
```

- If n is zero, the function should produce true no matter whether the list is empty or not because every element always appears at least 0 times
- We now look at a more complicated recursion, namely writing functions which

> Improved at-least?

```
(define (at-least? n elem lst)
  (cond [(zero? n) true]
        [(empty? lst) false]
        ; list is nonempty, n ≥ 1
        [(equal? (first lst) elem) (at-least? (sub1 n) elem (rest lst))]
        [else (at-least? n elem (rest lst))]))
```

The first two cases, in which n is 0, both produce true, so they could be collapsed into a single case. [(zero? n) true]

We collapsed (and (> n 0) (empty? lst)) to just (empty? lst) because n is not zero

The nested cond was promoted to the top level cond.

> Two condensed traces

```
(at-least? 3 'green (list 'red 'green 'blue)) ⇒
(at-least? 3 'green (list 'green 'blue)) ⇒
(at-least? 2 'green (list 'blue)) ⇒
(at-least? 2 'green empty) ⇒ false

(at-least? 1 8 (list 4 8 15 16 23 42)) ⇒
(at-least? 1 8 (list 8 15 16 23 42)) ⇒
(at-least? 0 8 (list 15 16 23 42)) ⇒ true
```

Efficiency

Sunday, October 18, 2020 16:36

Simple recursion

- Recall from Module 06: In **simple recursion**, every argument in a recursive function application are either
  - unchanged, or
  - one step closer to a base case according to a data definition
- To identify simple recursion, look at the arguments to the recursive function application - places where the function applies itself recursively:

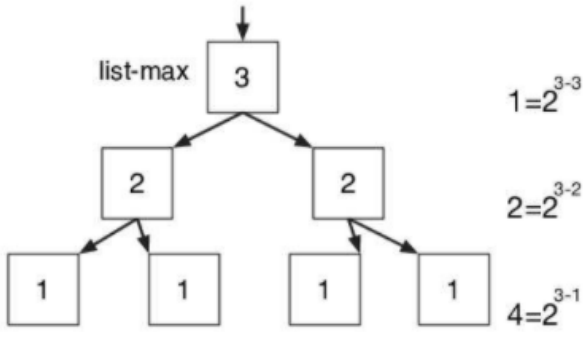
```
(define (func lst) ... (func (rest lst)) ...) ;; Simple
(define (func lst x) ... (func (rest lst) x) ...) ;; Simple
(define (func lst x) ... (func (process lst) x) ...) ;; NOT
Simple
(define (func lst x)
 ... (func (rest lst) (math-function x)) ...) ;; NOT Simple
```

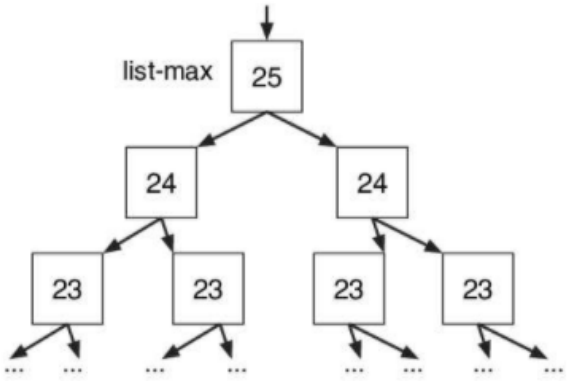
The limits of simple recursion

```
;; (max-list-v2 lon) produces the maximum element of lon
;; Examples:
(check-expect (max-list-v2 (list 6 2 3 7 1)) 7)

;; max-list-v2: (listof Num) → Num
;;   Requires: lon is nonempty
(define (max-list-v2 lon)
  (cond [(empty? (rest lon))  (first lon)]
        [(> (first lon) (max-list-v2 (rest lon))) (first lon)]
        [else  (max-list-v2 (rest lon))]))
```

There may be two recursive applications of max-list.
The code for max-list-v2 is correct.
But computing (max-list-v2 (countup-to 1 25)) is very slow.
This is because the initial application is on a list of length 25, and there are two recursive applications on the rest of this list, which is of length 24.
Each of those makes two recursive applications again.





max-list can make up to $2^n - 1$ recursive applications on a list of length n.

This is informally called **exponential blowup**.

Measuring efficiency

- We can take the number of recursive applications as a rough measure of a function's **efficiency**
- max-list-v2 can take up to $2^n - 1$ recursive applications on a list of length n
- length takes n recursive applications on a list of length n
- length's efficiency is proportional to n
  max-list-v2's efficiency is proportional to 2^n
- We express the former as O(n) and the later as $O(2^n)$
- "Families" of algorithms with similar efficiencies, from most efficient to least:

"Big-O"	Example
$O(1)$	no recursive calls; `tax-payable`
$O(\lg n)$	divide in half, work on one half; `binary-search` on a *balanced tree*
$O(n)$	one recursive application for each item; `length`
$O(n \lg n)$	divide in half, work on both halves; `quicksort`
$O(n^2)$	an $O(n)$ application for each item; `insertion-sort`
$O(2^n)$	two recursive applications for each item; `max-list`

- lg is $\log_2$
- Recognize when your function applies itself recursively twice and avoid that
- Note that the following code is not necessarily a problem:

```
(define (foo lst ...)
        ... (foo (rest lst) ...)   ;; application 1
        ... (foo (rest lst) ...)   ;; application 2
)
```

  It's ok if one application of (foo..) applies foo at either application 1 or application 2
  It's a problem if one application of foo does both of them.

Recap

Fast $O(n)$

```
(define (max-list-v1 lon)
  (cond [(empty? (rest lon))  (first lon)]
        [else (max (first lon) (max-list-v1 (rest lon)))]))
```

Slow $O(2^n)$

```
(define (max-list-v2 lon)
  (cond [(empty? (rest lon))  (first lon)]
        [(> (first lon) (max-list-v2 (rest lon))) (first lon)]
        [else  (max-list-v2 (rest lon))]))
```

- In many cases the slow version does the exact same computation twice (max-list-v2 (rest lon)). This leads to exponential blowup.
- The fast version does the computation (max-list-v1 (rest lon)) and passes that result to a helper function. The helper function can make use of that value as often as it needs to prevent calculating again, thus avoiding exponential blowup.

Accumulative recursion

Monday, October 19, 2020 20:20

A human approach

- Humans do not seem to use either of the two versions of max-list shown earlier.
- Instead, we tend to find the maximum of a list of numbers by scanning it, remembering the largest value seen so far.
- When we see a value that is larger than the largest seen so far, we remember the new value until we see another that is still larger.
- When we get to the end of the list, the largest value seen so far is the largest value in the list.

Accumulative recursion

- Computationally, we can pass down that largest value seen so far as a parameter called an **accumulator**
- This parameter accumulates the result of prior computation, and is used to compute the final answer that is produced in the base case.

```
;; (max-list/acc lon max-so-far) produces the largest
;;     of the maximum element of lon and max-so-far

;; max-list/acc: (listof Num) Num → Num
(define (max-list/acc lon max-so-far)
  (cond [(empty? lon) max-so-far]
        [(> (first lon) max-so-far)
         (max-list/acc (rest lon) (first lon))]
        [else (max-list/acc (rest lon) max-so-far)]))

(define (max-list-v3 lon)
  (max-list/acc (rest lon) (first lon)))
```

This is a wrapper. Wrappers are used with accumulative recursion due to the extra parameters which need to be initialized.

- max-list/acc is not simple recursion.
- In simple recursion, the arguments where max-list/acc is applied would be either one step closer to the base case or unchanged
- The first argument, (rest lon) is one step closer to the base case
- The second argument is sometimes max-so-far and sometimes (first lon)

Tracing max-list/acc

```
(max-list2 (list 1 2 3 9 5))
⇒ (max-list/acc (list 2 3 9 5) 1)
⇒ (max-list/acc (list 3 9 5) 2)
⇒ (max-list/acc (list 9 5) 3)
⇒ (max-list/acc (list 5) 9)
⇒ (max-list/acc (list ) 9)
⇒ 9
```

- This technique is known as **accumulative recursion**
- It is more difficult to develop and reason about such code, which is why simple recursion is preferable if it is appropriate
- Simple recursion will continue to be the main tool
- Sometimes accumulative recursion will be easier or yield a more efficient or elegant solution. In such cases, it is encouraged to use accumulative recursion

Indicators of the accumulative recursion pattern

- All arguments to recursive function applications are
  - unchanged, or
  - one step closer to a base case in the data definition, or
  - a partial answer (passed in an accumulator)
- An accumulative function requires at least one accumulator
- There may be more than one accumulator
- The value(s) in the accumulator(s) are used in one or more base cases
- The accumulatively recursive function usually has a wrapper function that sets the initial value of the accumulator(s)

Another accumulative example: reversing a list

- Using simple recursion

```
;; my-reverse: (listof X) → (listof X)
(define (my-reverse lst)
  (cond
    [(empty? lst) empty]
    [else (append (my-reverse (rest lst))
                  (list (first lst)))]))
```

- Intuitively, append does too much work in repeatedly moving over the produced list to add one element at the end
- This has the same worst-case behaviour as insertion sort, $O(n^2)$

Reversing a list with an accumulator

```
;; (my-reverse lst) reverses lst using accumulative recursion
;; Example:
(check-expect (my-reverse (list 1 2 3)) (list 3 2 1))

;; my-reverse: (listof X) → (listof X)
(define (my-reverse lst)      ; wrapper function
  (my-rev/acc lst empty))

(define (my-rev/acc lst acc)  ; helper function
  (cond [(empty? lst) acc]
        [else (my-rev/acc (rest lst)
                          (cons (first lst) acc))]))
```

- This is O(n)
- A condensed trace:

```
 (my-reverse (list 1 2 3 4 5))
⇒ (my-rev/acc (list 1 2 3 4 5) empty)
⇒ (my-rev/acc (list 2 3 4 5) (cons 1 empty))
⇒ (my-rev/acc (list 3 4 5) (cons 2 (list 1)))
⇒ (my-rev/acc (list 4 5) (cons 3 (list 2 1)))
⇒ (my-rev/acc (list 5) (cons 4 (list 3 2 1)))
⇒ (my-rev/acc (list ) (cons 5 (list 4 3 2 1)))
⇒ (list 5 4 3 2 1)
```

Generative recursion

Monday, October 19, 2020 21:40

Generative recursion: GCD

- The **Euclidean algorithm for Greatest Common Divisor (GCD)** can be derived from the following identity for m > 0

$$gcd(n, m) = gcd(m, n \bmod m)$$

We also have $gcd(n, 0) = n$.

- euclid-gcd

```
;; (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm

;; euclid-gcd: Nat Nat → Nat
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

- This function does not use simple or accumulative recursion.

Generative recursion

- The arguments in the recursive application were generated by doing a computation on m and n
- The function euclid-gcd uses generative recursion
- Functions using generative recursion are easier to get wrong, harder to debug and harder to reason about

Simple vs accumulative vs generative recursion

- In **simple recursion**, all arguments to the recursive function application are either unchanged, or one step closer to a base case in the data definition.
- In **accumulative recursion**, parameters are as above, plus parameters containing partial answers used In the base case
- In **generative recursion**, parameters are freely calculated at each step.

Compound data

Friday, October 2, 2020 20:24

Short, fixed-length, lists

- Payroll with names and salaries

```
(list (list "Asha" 50000)
      (list "Joseph" 100000)
      (list "Sami" 10000))
```

- Other kinds of data that always go together include:
  - Student (name, program, courses)
  - Point (x, y)
  - Book (author, title, number of pages)

Example: Student

- If we were to use a student list often, we might want to include:
  - Some helper functions to extract the name, program, and courses
  - A predicate to see if a given value represented a student
  - Error messages if we gave it another kind of list
- We cannot just check for a three-element list
- What we can do is to embed an extra value in the list that is highly unlikely to be in any other list
- If that value is present, then we will assume the list represents a student
- Otherwise, we will assume it does not

> Example: Student (1/3)

```
;; A Std (student) is a (make-std Str Str (listof Str))
```

The data definition allows us to use Std in contracts.

```
;; A large "random" value to check for legit student values
(define STD-TAG "std_391249569284455218")

;; (make-std name prog classes) makes a new student structure
;;    containing the name, program and classes for the student.
;; make-std: Str Str (listof Str) → Std
(define (make-std name prog classes)
  (list STD-TAG name prog classes))

;; A sample student for testing
(define Juan (make-std "Juan" "CS" (list "CS 135" "MATH 137")))
```

make-std makes a Std value. It takes the three attributes of a student and bundles them together with STD-TAG

> Example: Student (2/3)

```
;; (std? v) returns true if v is a Std and false otherwise.
(check-expect (std? Juan) true)
(check-expect (std? (list "Juan" "CS" (list "CS 135" "MATH 137"))) false)
(check-expect (std? "Juan") false)

;; std?:  Any → Bool
(define (std? s)
  (and (cons? s)
       (= (length s) 4)
       (string=? (first s) STD-TAG)))
```

It can consume anything, so we need to make sure it is a non-empty list (cons?)

A Std will have three pieces of data and the extra value (length 4)

STD-TAG needs to be the first value.

> Example: Student (3/3)

```
;; (std-name s) extracts the name field from student s; error
;;    if s is not a student
(check-expect (std-name Juan) "Juan")
(check-error (std-name (list "Juan"))
             "std-name: expects a std, given (list \"Juan\")")

;; std-name: Std → Str
(define (std-name s)
  (cond [(std? s) (second s)]
        [else (error "std-name: expects a std, given " s)]))
```

It's called an "selector function"

std-name checks to ensure that it consumed a Std value. If it didn't, it gives an error message. The built-in function error is used for that. check-error is used to test that an error was produced appropriately.

std-prog and std-classes are nearly identical to std-name.

Structures

- A Racket structure definition creates all of the above in only one line

```
(define-struct std (name prog classes))
;; A Std (student) is a (make-std Str Str (listof Str))
```

- define-struct is a special form that automatically creates functions identical to the functions on the

previous slides.

- The second line is the structure's data definition.
- Whenever you use define-struct, add a data definition to give the expected types.
- Given the data definition, Std may be used in contracts
- Functions automatically created:
  - make-std
  - std?
  - std-name
  - std-prog
  - std-classes
- The data definition for a structure always follows the form

```
;; A <type-name> (<English-name>) is a (make-<sname> <type> ...)
```

<sname> is the first argument to define-struct

- If define-struct is given n field names, it will define n+2 functions. Each function will include <sname> in its name.
  - The first one is make-<sname>. It's called a **constructor function** and is used to construct values
  - The second is a **predicate**
  - Then there is one **selector** for each of the n fields. Each is named <sname>-<fname> where <fname> is the name of a field.
  - The STD-TAG constant is added automatically behind the scenes by Racket. It is not our concern when we use structures and we do not have access to it

> Example: add-class

```
(define-struct std (name prog classes))
;; A Std (student) is a (make-std Str Str (listof Str))

;; (add-class s class) adds a new class to the student s.
(check-expect (add-class (make-std "Jo" "CS" (list "MATH 137")) "CS 135")
              (make-std "Jo" "CS" (list "CS 135" "MATH 137")))

;; add-class: Std Str → Std
(define (add-class s class)
  (make-std (std-name s)
            (std-prog s)
            (cons class (std-classes s))))
```

`(make-std n p c)` is considered a value (as long as `n`, `p`, and `c` are values) and will not be simplified.

(make-std "Jo" "CS" (list "Math 137") is a value

Formalities

Tuesday, October 20, 2020 17:26

Syntax and semantics

- The special form (define-struct sname (fname_1 … fname_n) defines the structure type sname with fields fname_1 to fname_n.
- It also automatically defines the following primitive functions:
  - **Constructor**: make-sname
  - **Selectors**: sname-fname_1 … sname-fname_n
  - **Predicate**: sname?
- Sname (note the capitalization) may be used in contracts.

Substitution rules

- (make-sname v_1 … v_n) is a value.
- The substitution rule for the ith selector is

  `(sname-fname_i (make-sname v_1 ... v_i ... v_n)) ⇒ v_i.`

- Finally, the substitution rules for the new predicate are:

  `(sname? (make-sname v_1 ... v_n)) ⇒ true`

  `(sname? V) ⇒ false` for `V` a value of any other type.

Structure templates

- The template function for a structure simply selects all its fields, in the same order as listed in the define-struct
- Example:

```
(define-struct std (name prog classes))
;; A Std (student) is a (make-std Str Str (listof Str))

;; std-template: Std → Any
(define (std-template s)
  (    ... (std-name s)
       ... (std-prog s)
       ... (std-classes s) ... ))
```

- The above (structure definition, data definition, and template function) are only required once per file

Example: Classlists

Tuesday, October 20, 2020 17:38

Example: Classlists

- Define a class list that contains students enrolled in a course.
- Develop functions that
  - Produce the names of the students in the class list
  - Add a new student to the classlist, preserving alphabetical order
  - Verify that all the students in a classlist have the class in their list of classes.

> Classlists (1/4)

```
(define-struct std (name prog classes))
;; A Std (student) is a (make-std Str Str (listof Str))

;; std-template: Std → Any
(define (std-template s)
  (    ... (std-name s) ... (std-prog s) ... (std-classes s)))

;; A Classlist is a (listof Std)

;; Sample students for testing
(define aj (make-std "AJ" "Math" (list "CS 135" "MATH 137")))
(define jo (make-std "Jo" "CS" (list "CS 135" "SPCOM 109")))
(define di (make-std "Di" "Math" (list "CS 135" "MATH 137")))
```

Structure definition and data definition for Std (student)

These definitions are only done **once**, no matter how many functions use them.

Data definition for Classlist can be omitted by writing (listof Std) in contracts, but it is easier and more understandable

> Classlists (2/4)

```
(define aj (make-std "AJ" "Math" (list "CS 135" "MATH 137")))
(define jo (make-std "Jo" "CS" (list "CS 135" "SPCOM 109")))
(define di (make-std "Di" "Math" (list "CS 135" "MATH 137")))

;; (class-names clst) produces a list of the student names in clst.
(check-expect (class-names (list aj jo di))
              (list "AJ" "Jo" "Di"))

;; class-names: Classlist → (listof Str)
(define (class-names clst)
  (cond [(empty? clst) empty]
        [(cons? clst) (cons (std-name (first clst))
                            (class-names (rest clst)))]))
```


- class-names consumes a Classlist which is a (listof Std). So the template to start with is the listof-X-template

```
;; listof-X-template: (listof X) -> Any
(define (listof-X-template lox)
  (cond [(empty? lox) ...]
        [(cons? lox) (... (first lox) ...
                          (listof-X-template (rest lox)) ...)]))
```

- (first lox) is a student. So to write out the full classlist-template we would apply the std-template

```
;; classlist-template: Classlist -> Any
(define (classlist-template clst)
  (cond [(empty? clst) ...]
        [(cons? clst)  (... (std-template (first clst)) ...
                            (classlist-template (rest clst)) ...)])
)
```

> Classlists (3/4)

```
(define aj (make-std "AJ" "Math" (list "CS 135" "MATH 137")))
(define jo (make-std "Jo" "CS" (list "CS 135" "SPCOM 109")))
(define di (make-std "Di" "Math" (list "CS 135" "MATH 137")))

;; (add-std s clst) produces a new classlist composed of student s
;;   all the students in clst.  Maintain alphabetical order.
(check-expect (add-std di (list aj jo))
              (list aj di jo))

;; add-std: Std Classlist → Classlist
;;   reqires: the classlist is in alphabetical order
(define (add-std s clst)
  (cond [(empty? clst) (list s)]
        [(string<? (std-name s) (std-name (first clst))) (cons s clst)]
        [else (cons (first clst) (add-std s (rest clst)))]))
```

No need to repeat the structure and data definitions.

> Classlists (4/4)

```
(define aj (make-std "AJ" "Math" (list "CS 135" "MATH 137")))
(define jo (make-std "Jo" "CS" (list "CS 135" "SPCOM 109")))
(define di (make-std "Di" "Math" (list "CS 135" "MATH 137")))

;; (all-enrolled? class clst) produces true iff each student in clst has
;;    class in their list of classes
(check-expect (all-enrolled? "CS 135" (list aj jo di)) true)
(check-expect (all-enrolled? "MATH 137" (list aj jo di)) false)

;; all-enrolled?:  Str Classlist → Bool
(define (all-enrolled? class clst)
  (cond [(empty? clst) true]
        [else (and (member? class (std-classes (first clst)))
                   (all-enrolled? class (rest clst)))]))
```

Mixed data

Tuesday, October 20, 2020 18:24

Mixed data

- Racket provides predicates such as number? and symbol? to identify data types
- define-struct also defines a predicate that tests whether its argument is that type of structure
- We can use these to check aspects of contracts and to write functions that consume mixed data - data of several (probably related) types
- Example: A university has undergraduate students as well as graduate students. Graduate students are like other students except they also have a supervisor

Data definitions

```
(define-struct ustd (name prog classes))
;; A UStd (undergraduate student) is a (make-ustd Str Str (listof Str))

(define-struct gstd (name prog supervisor classes))
;; A GStd (graduate student) is a (make-gstd Str Str Str (listof Str))

;; A Student is one of:
;; * a UStd
;; * a GStd

;; A Classlist is a (listof Student)
```

Types like Student and Classlist are for our benefit only. They help us understand the data that our functions consume and produce

- There is no structure definition for mixed data
- There is a data definition that describes the data and gives a name that can be used in contracts

Template function

- The template function for mixed data will determine the type of data and then include a template for that type

```
;; student-template:  Student → Any
(define (student-template s)
  (cond [(ustd? s) (... (ustd-name s)...
                        (ustd-prog s) ...
                        (ustd-classes s)...)]
        [(gstd? s) (... (gstd-name s)...
                        (gstd-prog s)...
                        (gstd-supervisor s)...
                        (gstd-classes s)...)]))
```

- An alternative for student-template would define and then use ustd-template and gstd-template:

```
(define (ustd-template s)
   (... (ustd-name s) ...
        (ustd-prog s) ...
        (ustd-classes s) ...))
(define (gstd-template s)
   (... (gstd-name s) ...
        (gstd-prog s) ...
        (gstd-supervisor s) ...
        (ustd-classes s) ...))

;; student-template: Student -> Any
(define (student-template s)
  (cond [(ustd? s) (... (ustd-template s) ...)]
        [(gstd? s) (... (gstd-template s) ...)]))
```

> Example: Update program

```
;; (update-prog std prog) updates the student's program ...
(check-expect (update-prog (make-ustd "Jo" "Math" empty) "CS")
              (make-ustd "Jo" "CS" empty))
(check-expect (update-prog (make-gstd "Di" "CS" "Ian" empty) "Arts")
              (make-gstd "Di" "Arts" "Ian" empty))

;; update-prog:  Student Str → Student
(define (update-prog std prog)
  (cond [(ustd? std) (make-ustd (ustd-name std)
                                prog
                                (ustd-classes std))]
        [(gstd? std) (make-gstd (gstd-name std)
                                prog
                                (gstd-supervisor std)
                                (gstd-classes std))]))
```

> Example: Filter by program

```
;; (filter-prog prog cl) produces a classlist consisting of only
;;    the students in cl who are in the program prog.
;; filter-prog: Str Classlist → Classlist
(define (filter-prog prog cl)
  (cond [(empty? cl) empty]
        [(in-prog? prog (first cl))
         (cons (first cl) (filter-prog prog (rest cl)))]
        [else (filter-prog prog (rest cl))]))

;; (in-prog? prog s) produces true iff student s is in program prog.
;; in-prog? String Student → Bool
(define (in-prog? prog s)
  (string=? prog (cond [(ustd? s) (ustd-prog s)]
                       [(gstd? s) (gstd-prog s)])))
```


anyof types

- Unlike UStd and GStd, the Student and Classlist types do not have a structure definition (i.e. define-struct)
- For contracts like these to make sense, we need to have the data definitions for Student and Classlist included as a comment in the program

```
;; update-prog:  Student Str → Student

;; filter-prog: Str Classlist → Classlist
```

- An alternative to Student would be to use

```
;; update-prog:  (anyof UStd GStd) Str → (anyof UStd GStd)
```

Checked functions

- Constructor functions do not check that their arguments have the correct type
- We can use type predicates to make a type-safe version

```
(define-struct ustd (name prog classes))
;; A UStd (undergraduate student) is a (make-ustd Str Str (listof Str))

(define (safe-make-ustd name prog classes)
  (cond [(and (string? name) (> (string-length name) 0)
              (string? prog) (> (string-length prog) 0)
              (list? classes)) (make-ustd name prog classes)]
        [else (error "Invalid argument types")]))

(check-error (safe-make-ustd "Jo" 123 empty) "Invalid argument types")
(check-error (safe-make-ustd "Jo" "CS" 'Sym) "Invalid argument types")
(check-expect (safe-make-ustd "J" "C" empty) (make-ustd "J" "C" empty))
```

Structures vs Lists

Wednesday, October 21, 2020 9:28

Structures vs. Lists

- We do not have to use structures. We could construct a class list with simple lists

```
(define cs135/s (list
                 (make-ustd "AJ" "CS" (list "CS 488" "CS 449"))
                 (make-gstd "Jo" "CS" "Ian" (list "CS 688" "CS 749"))
                 (make-ustd "Di" "Math" (list "CS 488" "PMATH 330"))))

(define cs135/l (list
                 (list "AJ" "CS" (list "CS 488" "CS 449"))
                 (list "Jo" "CS" "Ian" (list "CS 688" "CS 749"))
                 (list "Di" "Math" (list "CS 488" "PMATH 330"))))
```

Structures

- help avoid some programming error (e.g. extracting the wrong field)
- provide meaningful names that are easier to read and understand
- automatically generate functions

Lists

- make it possible to write "generic" functions that operate on several types of data
- can be expressed more compactly than structures

Quoting

Wednesday, October 21, 2020 9:53

Quoting

- cons notation emphasizes a fundamental characteristic of a list - it has a first element and the rest of the elements. Elements of the list can be computed as the list is constructed
- list notation makes our lists more compact but loses the remainder about the first element and the rest, Like cons, element of the list can be computed as the list is constructed.
- **Quote notation** is even more compact but loses the ability to compute elements during construction.

Examples:

- `'1 ⇒ 1, '"ABC" ⇒ "ABC", 'earth ⇒ 'earth`
- `'(1 2 3) ⇒ (list 1 2 3)`
- `'(a b c) ⇒ (list 'a 'b 'c)`
- `'(1 ("abc" earth) 2) ⇒ (list 1 (list "abc" 'earth) 2)`
- `'(1 (+ 2 3)) ⇒ (list 1 (list '+ 2 3))`
- `'() ⇒ empty`

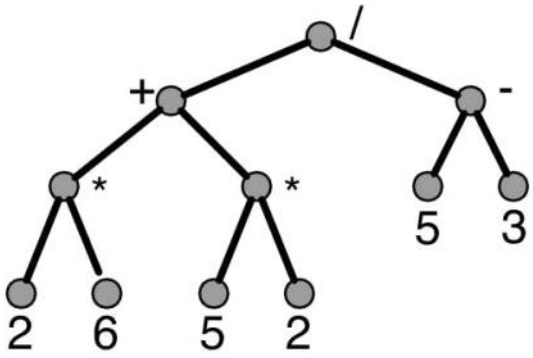
- Quoting applies to more than lists
- Quoted numbers, strings and characters remain unchanged
- Quoting (…) turns the … into a list
- Quoting a list of symbols "factors out" the quote to the front of the list
- Parentheses nested inside a quoted list are also turned into a list. They should not be quoted
- Because nested parentheses turn into sublists, we cannot easily include function applications. Racket turns the function into a symbol

Examples

Friday, October 23, 2020 12:34

> Example: binary expression trees

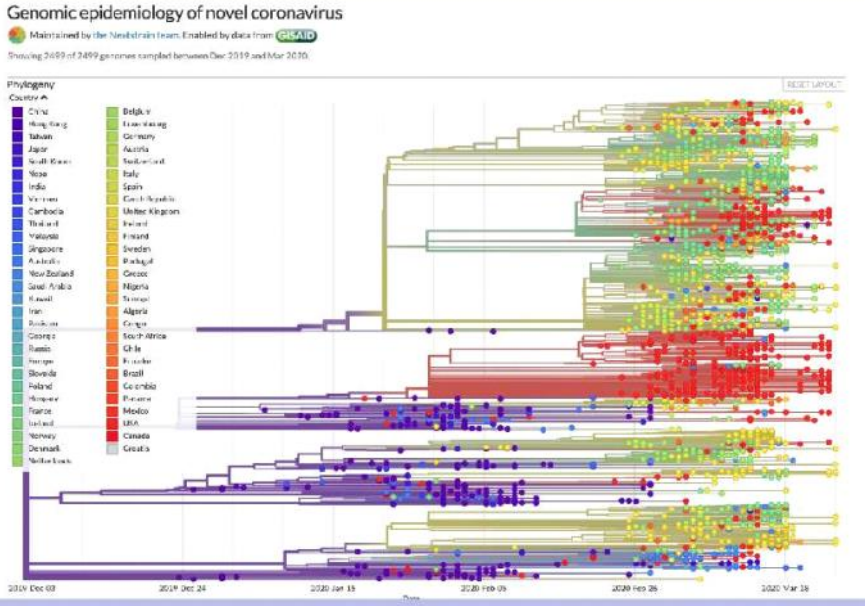
The expression $((2 * 6) + (5 * 2))/(5 - 3)$ can be represented as a **tree**:



> Example: Phylogentic trees

This phylogentic tree tracks the evolution of COVID-19 in the first four months of the recent pandemic.

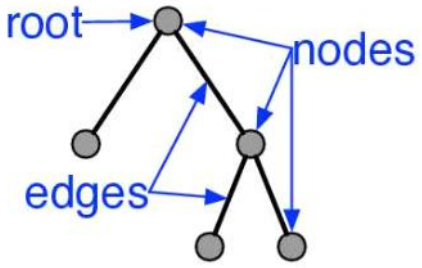
Image: nextstrain.org/ncov



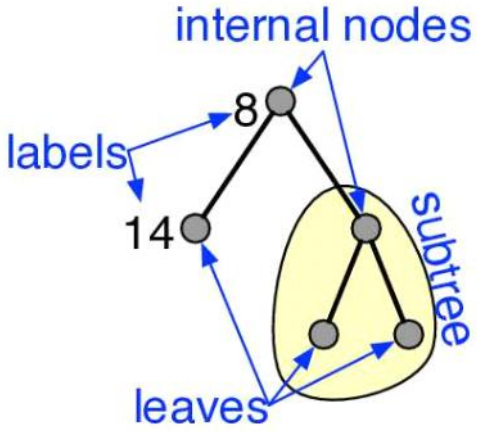
Tree terminology

- A tree is a set of **nodes** and **edges** where an edge connects two distinct nodes.
- A tree has three requirements
  - One node is identified as the **root**.
  - Every node c other than the root is connected by an edge to some other node p

p is called the **parent** and c is called the **child**
    - A tree is connected for every node n other than the root

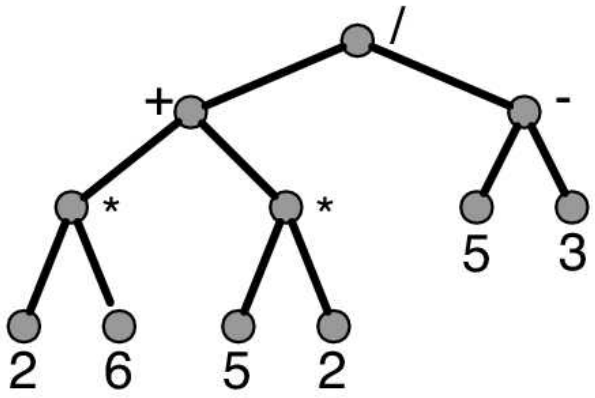


- Other useful terms
    - **leaves**: nodes with no children
    - **internal nodes**: nodes that have children
    - **labels**: data attached to a node
    - **ancestors of node n**: n itself, the parent of n, the parent of the parent, … up to the root
    - **descendants of n**: all the nodes that have n as an ancestor
    - **subtree rooted at n**: n and all of its descendants



Characteristics of trees

- Number of children of internal nodes:
    - exactly two
    - at most two
    - any number
- Labels:
    - on all nodes
    - just on leaves
- Order of children (matters or not)
- Tree structure (from data or for convenience)
- In some trees, each internal node will always have exactly two children while others may have an unlimited number.
- It is also useful to know whether the order of the children matters or not.
- Consider the binary expression tree
    - Each internal node has exactly two children. That's implied by the "binary" in the name.
    - It only deals with operators having a left and a right operand.
    - The order of the children in a binary expression tree. If we changed the order of the children of the division operator, we would get a different result when evaluating the expression



- For a binary search tree (BST), the structure will be very important to enable fast searching, but the structure will be decided by us rather than coming from the data.

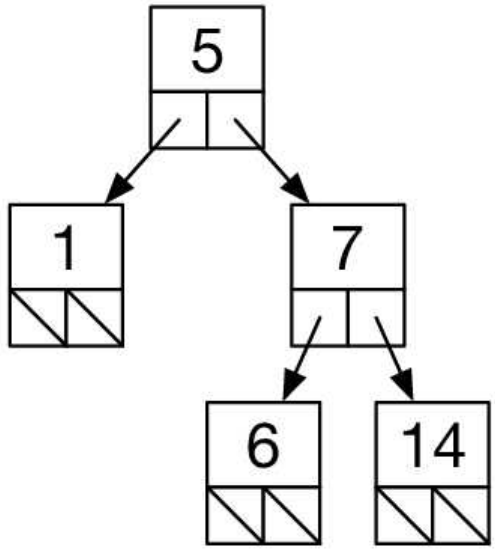
Binary Trees

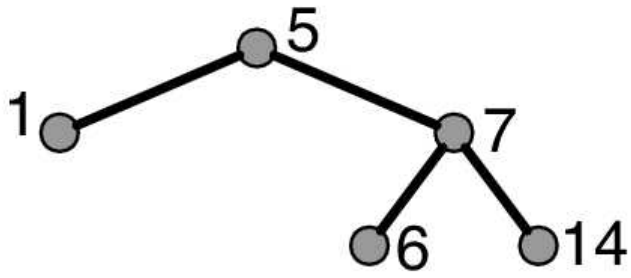
Friday, October 23, 2020 13:38

Binary trees

- A **binary tree** is a tree with at most two children for each node.
- Binary arithmetic expression trees and evolution trees are both examples of binary trees
- Characteristics:
  - Each internal node has at most two children
  - Our examples will have labels on all the nodes. It is however not a requirement of binary trees
  - Order of the children does not matter
  - Structure is for convenience

Drawing binary trees





Note: We will consistently use Nats in our binary trees, but we could use symbols, strings, structures, etc.

Binary tree data definition

```
(define-struct node (key left right))
;; A Node is a (make-node Nat BT BT)

;; A binary tree (BT) is one of:
;; * empty
;; * Node
```

- The node's label is called "key" in anticipation of using binary trees to implement dictionaries
- The BT data definition is an example of mixed data
  - One case is a particular kind of list, and empty list
  - The other case is a structure
- We could have used many other values instead of empty - any value that we can distinguish from a Node, such as 0, false or 'emptyTree
- We chose empty because it is an existing value that strongly suggests "empty tree".

Aside: Tips for building templates

- For each part of the data definition
  - If it is a defined data type, apply that type's template
  - If it says "one of" or is mixed data, include a cond to distinguish the cases
  - If it is **compound data** (a structure), extract each of the fields, in order
  - If it is a **list**, extract the first and rest of the list
- Add ellipses around each of the above
- Apply the above recursively

> Example: sum-keys

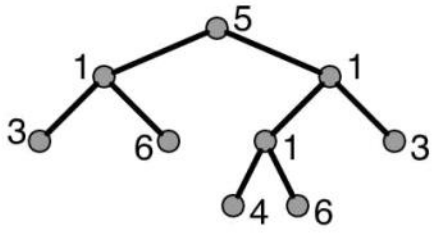
```
;; (sum-keys bt) sums the keys in the binary tree bt.
;; Examples
(check-expect (sum-keys empty) 0)
(check-expect (sum-keys (make-node 10 empty empty)) 10)
(check-expect (sum-keys (make-node 10
                                   (make-node 5 empty empty)
                                   empty)) 15)

;; sum-keys: BT → Nat
(define (sum-keys bt)
```

> Example: count-nodes

```
(define test-tree (make-node 5 (make-node 1 ...
                                (make-node 1 ...))))
;; (count-nodes tree k) counts the number of nodes in the tree that
;;    have a key equal to k.
(check-expect (count-nodes empty 5) 0)
(check-expect (count-nodes test-tree 1) 3)

;; count-nodes: BT Nat → Nat
(define (count-nodes tree k)
  (cond [(empty? tree) 0]
        [else (+ (cond [(= k (node-key tree)) 1]
                       [else 0])
                 (count-nodes (node-left tree) k)
                 (count-nodes (node-right tree) k))]))
```



> Example: Increment keys

```
;; (increment tree) adds 1 to each key in the tree.
;; increment: BT → BT
(define (increment tree)
  (cond
    [(empty? tree) empty]
    [else (make-node (add1 (node-key tree))
                     (increment (node-left tree))
                     (increment (node-right tree)))]))
```

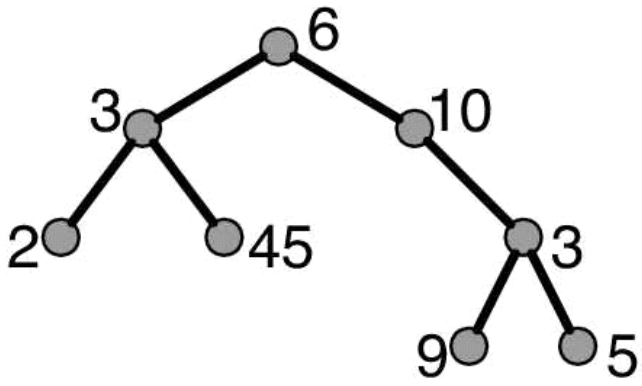

Searching binary trees

- Searching a binary tree for a given key - produce true if the key is in the tree and false otherwise
- The strategy
  - See if the root node contains the key we are looking for. If so, produce true.
  - Otherwise, recursively search in the left subtree and in the right subtree.
  - If either recursive search finds the key, produce true. Otherwise, produce false.

```
;; (search k tree) produces true if k is in tree; false otherwise.
;; search: Nat BT → Bool
(define (search-bt k tree)
  (cond [(empty? tree) false]
        [(= k (node-key tree)) true]
        [else (or (search-bt k (node-left tree))
                  (search-bt k (node-right tree)))]))
```

Find the path to a key

- Write a function, search-bt-path, that searches for an item in the tree. As before, it will return false if the item is not found. However, if it is found, the function will return a list of symbols 'left and 'right indicating the path from the root to the item
- If the tree contains a duplicate, produce the path to the left-most item



The path from 6 to 9 is

```
'(right right left).
```

> search-bt-path

```
;; search-bt-path-v1: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v1 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]
    [(list? (search-bt-path-v1 k (node-left tree)))
     (cons 'left (search-bt-path-v1 k (node-left tree)))]
    [(list? (search-bt-path-v1 k (node-right tree)))
     (cons 'right (search-bt-path-v1 k (node-right tree)))]
    [else false]))
```

'() means empty

Double calls to `search-bt-path`. Uggh!

- Strategy for solving this problem
  - A non-empty tree has three important parts: the root, the left subtree and the right subtree
  - If the root contains k, the value we are searching for, produce empty and we are done.
  - Otherwise, if only we could find a path to k in the left subtree, we could cons 'left onto that path
  - If we cannot find k in the left subtree, we could look in the right subtree. If we find a path there, all we need to do is cons the value 'right onto that path.
  - If k is not in the root, not in the left subtree and not in the right subtree, then it is not in the tree at all. we produce false.
  - How to check if k is in the left subtree? Apply search-bt-path. If it produces a list, we know k was found
  - However, after we have checked if k is in the left subtree, we still need to have the path to k so that we can have (cons 'left path-to-k). We need to call search-bt-path one more time
  - This strategy is correct, but leads to inefficient run-time
- The strategy above does not look that much like binary tree template:

```
;; bt-template: BT --> Any
(define (bt-template t)
  (cond [(empty? t) ...]
        [(node? t) (... (node-key t)
                        (bt-template (node-left t))
                        (bt-template (node-right t)))]))
```

- The next version of search-bt-path actually looks more like the template while also solving the efficiency issue

> Improved search-bt-path

```
;; search-bt-path-v2: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v2 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]     '() means empty
    [else (choose-path-v2 (search-bt-path-v2 k (node-left tree))
                          (search-bt-path-v2 k (node-right tree)))]))

(define (choose-path-v2 left-path right-path)
  (cond [(list? left-path) (cons 'left left-path)]
        [(list? right-path) (cons 'right right-path)]
        [else false]))
```


- The insight is that once a value is passed to a parameter it can be used multiple times without recalculating it
- So we search for k in both the left and right subtrees, passing the results to the helper function. choose-path can use those values multiple times without hurting the efficiency

Binary search trees

Tuesday, October 27, 2020 15:43

Data definition

```
;; A Binary Search Tree (BST) is one of:
;; * empty
;; * a Node

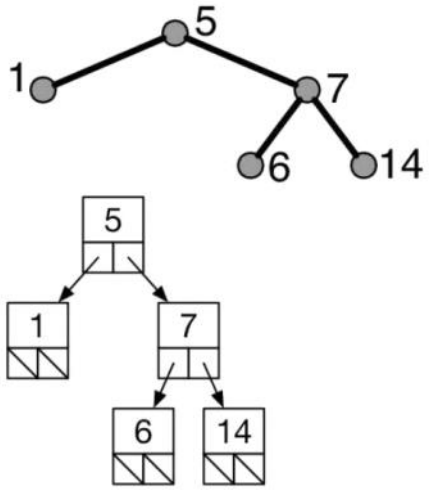
(define-struct node (key left right))
;; A Node is a (make-node Nat BST BST)
;; Requires: key > every key in left BST
;;           key < every key in right BST
```

The BST ordering property

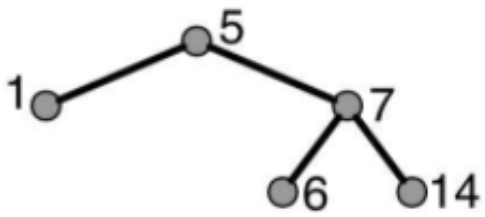
- key is greater than every key in left
- key is less than every key in right
- The ordering property holds in every subtree

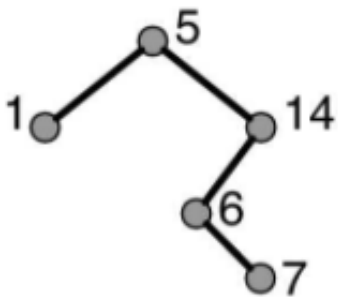
Example:

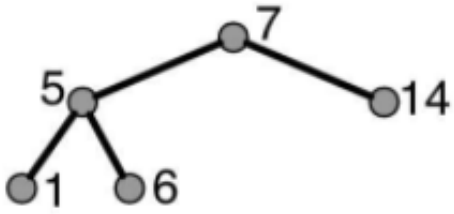
```
(make-node 5
  (make-node 1 empty empty)
  (make-node 7
             (make-node 6 empty empty)
             (make-node 14 empty empty)))
```



There can be several BSTs holding a particular set of keys.







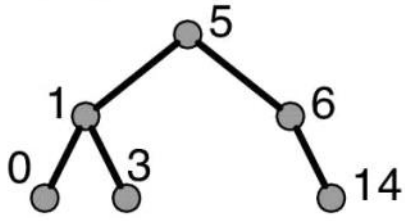
Searching in a BST (for n)

- If the BST is empty, then n is not in the BST.
- If the BST is a node (make-node k left right), and k equals n, then we have found it.
- Otherwise it might be in either the left or right subtree
  - If n < k, then n might be in the left subtree, and we only need to recursively search in left
  - If n > k, then n must be in the right subtree, and we only need to recursively search in right.
- Either way, we save one recursive function application

search-bst

```
;; (search-bst n t) produces true if n is in t; false otherwise.
;; search-bst: Nat BST → Bool
(define (search-bst n t)
  (cond[(empty? t) false]
       [(= n (node-key t)) true]
       [(< n (node-key t)) (search-bst n (node-left t))]
       [(> n (node-key t)) (search-bst n (node-right t))]))
```

The last clause, (> n (node-key t)), could be replaced with else



Adding to a BST

- If t is empty, then the result is a BST with only one node containing n

- If t is of the form (make-node k left right) and n = k, then the key is already in the tree and we can simply produce t
- Otherwise, n must go in either the left or right subtree
  - If n < k, then the new key must be added to left
  - If n > k, then the new key must be added to right
- Again, we only need to make one recursive function application
- The contract is bst-add: Nat BST --> BST
- When we consume a node that does not match the key, we need to produce a new node containing the old key, the transformed subtree, and the untransformed other subtree.

Creating a BST from a list

- If the list is empty, the BST is empty
- Is the list is of the form (cons k lst), we add the key k to the BST created from the lst. The first key in the list is inserted last.
- It is also possible to write a function that inserts key in the opposite orde

Augmenting trees

Tuesday, October 27, 2020 16:39

Augmenting trees

- So far nodes have been (define-struct node (key left right))
- We can **augment** the node with additional data (define-struct node (key val left right))
  - The name val is arbitrary - choose any name
  - The type of val is also arbitrary - number, string, structure, etc
  - Could augment with multiple values
  - The set of keys remains unique
  - The tree could have duplicate values

BST dictionaries

- An augmented BST can serve as a dictionary that can perform significantly better than an association list, which is a list of two-element lists
- We need to modify node to include the value associated with the key and search needs to return the associated value, if the key is found.

> search-bst-dict

```
(define-struct node (key val left right))
;; A binary search tree dictionary (BSTD) is either:
;; * empty
;; * (make-node Nat Str BSTD BSTD)

;; (search-bst-dict k t) produces the value associated with k
;;     if k is in t; false otherwise.
;; search-bst-dict: Nat BSTD → (anyof Str false)
(define (search-bst-dict k t)
  (cond[(empty? t) false]
       [(= k (node-key t)) (node-val t)]
       [(< k (node-key t)) (search-bst-dict k (node-left t))]
       [(> k (node-key t)) (search-bst-dict k (node-right t))]))
```

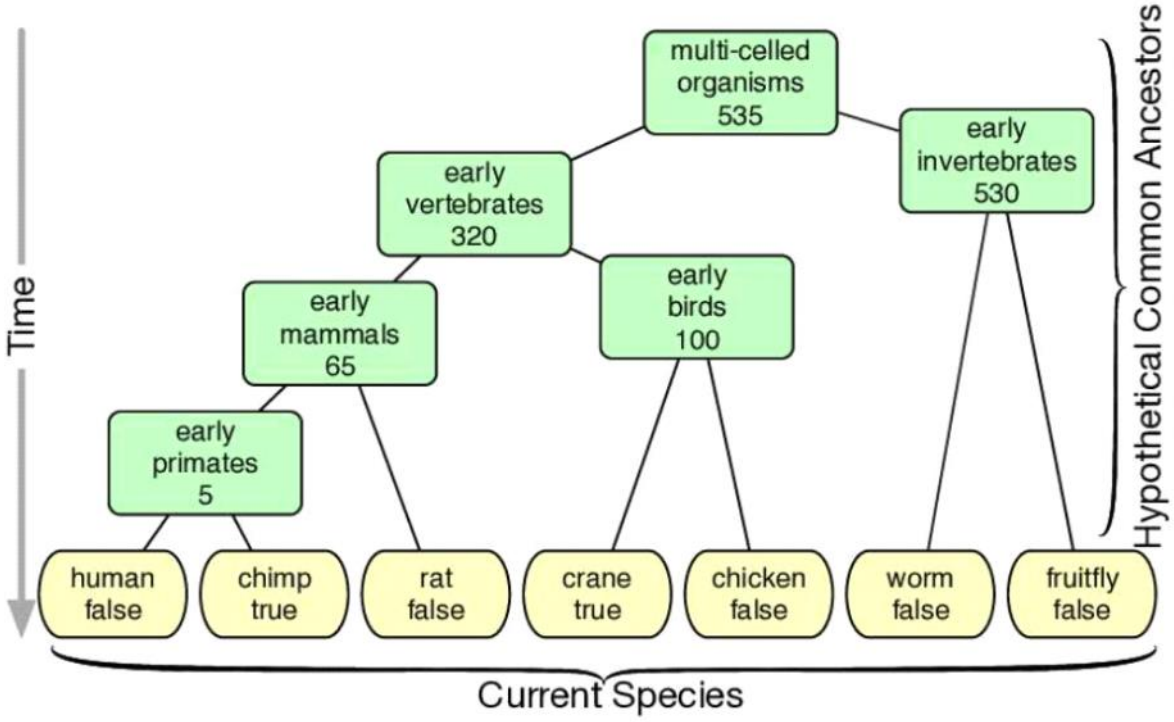
> search-bst-dict tests

```
(define test-tree (make-node 5 "Susan"
                             (make-node 1 "Juan" empty empty)
                             (make-node 14 "David"
                                        (make-node 6 "Lucy" empty empty)
                                        empty)))

(check-expect (search-bst-dict 5 empty) false)
(check-expect (search-bst-dict 5 test-tree) "Susan")
(check-expect (search-bst-dict 6 test-tree) "Lucy")
(check-expect (search-bst-dict 2 test-tree) false)
```


Evolutionary trees

- **Evolutionary trees** are augmented binary trees that show the evolutionary relationships between species. Biologists believe that all life on Earth is part of a single evolutionary tree, indicating common ancestry
- Leaves represent a **current species**. They are augmented with a name and whether the species is endangered
- Internal nodes represent a **hypothesized common ancestor species** that split into two new species. Internal nodes are augmented with a name and an estimate of how long ago the split took place (in millions of years)
- Evolutionary trees are constructed by evolutionary biologists
  - Start with current species
  - Based on common attributes (including DNA sequences), hypothesize common ancestor species
  - Keep going with more and more common ancestor species
  - Back to a single common ancestor (the root)



Representing evolutionary trees

- Internal nodes each have exactly two children
- Each internal node has
  - the name of the common ancestor species
  - how long ago the common ancestor split into two new species
  - the two species that resulted from the split
- Leaves have
  - the name of the current species
  - the endangerment status (true if endangered; false otherwise)
- The order of children does not matter
- The structure of the tree is dictated by a hypothesis about evolution

Data definitions

```
;; An EvoTree (Evolutionary Tree) is one of:
;; * a Current (current species)
;; * an Ancestor (common ancestor species)

(define-struct current (name endangered))
;; A Current is a (make-current Str Bool)
(define-struct ancestor (name age left right))
;; An Ancestor is a (make-ancestor Str Num EvoTree EvoTree)
```

Note that the Ancestor data definition uses a pair of EvoTrees

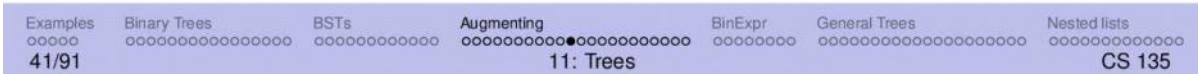
	Binary Tree	Evolutionary Tree
Data definition	A Node is a (make-node Nat BT BT). A binary tree (BT) is one of: • empty • Node	A Current is a (make-current Str Bool). An Ancestor is a (make-ancestor Str Num EvoTree EvoTree). An (EvoTree) is one of: • A Current • An Ancestor

Non-recursive case:	empty	Current structure
Recursive case	Node structure	Ancestor structure
Sample leaf node	(make-node 5 empty empty)	(make-current "human" false)
Smallest possible tree	empty	A single Current species
Subtree order	Sometimes matters	Does not matter
Number of children	Nodes may have 0, 1 or 2 children	Every internal node has exactly two children
Types of children	The left and right fields for a BT are BTs	The left and right fields for an Ancestor are not Ancestor nodes. They are EvoTrees. They might be Current nodes or they might be Ancestor nodes.

» Constructing the example evolutionary tree (1/2)

```
(define-struct current (name endangered))
;; A Current is a (make-current Str Bool)
(define-struct ancestor (name age left right))
;; An Ancestor is a (make-ancestor Str Num EvoTree EvoTree)

(define human (make-current "human" false))
(define chimp (make-current "chimp" true))
(define rat (make-current "rat" false))
(define crane (make-current "crane" true))
(define chicken (make-current "chicken" false))
(define worm (make-current "worm" false))
(define fruit-fly (make-current "fruit fly" false))
```



» Constructing the example evolutionary tree (2/2)

```
(define e-primates (make-ancestor "early primates" 5 human chimp))
(define e-mammals (make-ancestor "early mammals" 65 e-primates rat))
(define e-birds (make-ancestor "early birds" 100 crane chicken))
(define e-vertebrates
  (make-ancestor "early vertibrates" 320 e-mammals e-birds))
(define e-invertebrates
  (make-ancestor "early invertibrates" 530 worm fruit-fly))
(define mco
  (make-ancestor "multi-celled organisms"
                 535 e-vertebrates e-invertebrates))
```


> EvoTree Template (1/3)

```
;; An EvoTree (Evolutionary Tree) is one of:
;; * a Current (current species)
;; * an Ancestor (common ancestor species)

(define-struct current (name endangered))
;; A Current is a (make-current Str Bool)
(define-struct ancestor (name age left right))
;; An Ancestor is a (make-ancestor Str Num EvoTree EvoTree)


;; evotree-template:  EvoTree → Any
(define (evotree-template t)
  (cond [(current? t) (current-template t)]
        [(ancestor? t) (ancestor-template t)]))
```

» EvoTree Template (2/3)

```
;; current-template: Current → Any
(define (current-template cs)
  (... (current-name cs) ...
       (current-endangered cs) ...))

;; ancestor-template: Ancestor → Any
(define (ancestor-template as)
  (... (ancestor-name as) ...
       (ancestor-age as) ...
       (ancestor-left as) ...
       (ancestor-right as) ...))
```

This is a straightforward implementation based on the data definition.

It's also a good strategy to take a complicated problem (dealing with an EvoTree) and decompose it into simpler problems (dealing with a `Current` or an `Ancestor`).

- We know that (ancestor-left as) and (ancestor-right as) are EvoTrees, so we apply the EvoTree function to them.

```
;; ancestor-template:  Ancestor → Any
(define (ancestor-template as)
  (... (ancestor-name as) ...
       (ancestor-age as) ...
       (evotree-template (ancestor-left as)) ...
       (evotree-template (ancestor-right as)) ...))
```

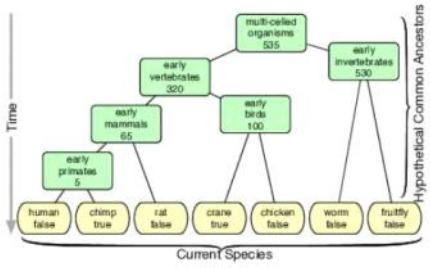
- ancestor-template uses evotree-template and evotree-template uses ancestor-template
- This is called **mutual recursion**. It is when a pair of functions call each other
- The base case in this example is [(current? t) (current-template t)] in evotree-template because current-template has no recursive applications

EvoTree Example 1

- Counts the number of current species within an evotree
- Strategy for solving this problem
  - If t is a current species, we have exactly one current species and no subtrees, so just return 1.
  - If t is a common ancestor, then count the number of current species in each of its two subtrees and add these numbers together

```
;; (count-current-species t): Counts the number of current species
;;     (leaves) in the EvoTree t.
(check-expect (count-current-species mco) 7)
(check-expect (count-current-species human) 1)

;; count-current-species: EvoTree → Nat
(define (count-current-species t)
  (cond [(current? t) (count-current t)]
        [(ancestor? t) (count-ancestor t)]))
```



```
;; count-current Current → Nat
(define (count-current t)
  1)

;; count-ancestor Ancestor → Nat
(define (count-ancestor t)
  (+ (count-current-species (ancestor-left t))
     (count-current-species (ancestor-right t))))
```

- In this case mutual recursion can be avoided by folding the two helper functions into count-current-species

```
(define (count-current-species t)
  (cond
    [(current? t) 1]
    [(ancestor? t) (+ (count-current-species (ancestor-left t))
                      (count-current-species (ancestor-right t)))]))
```

Traversing a tree

- A **tree traversal** refers to the process of visiting each node in a tree exactly once
- Visiting a node just means doing something with it. It might be checking to see if it has the value we are looking for, collecting some information from it, or transforming it in some way
- They key idea is that each node is visited at least once
- They are often classified by the order in which the nodes are visited
  - **pre-order**: visit the root, then each subtree
  - **in-order**: visit one subtree, the root, then the other subtree
  - **post-order**: visit both subtrees, then the root
- The order often affects the function's result
- The increment example from binary trees is one example of a pre-order traversal

List-names

» list-names

```
;; list-names: EvoTree → (listof Str)
(define (list-names t)
  (cond [(current? t) (list-cnames t)]
        [(ancestor? t) (list-anames t)]))

;; list-cnames: Current → (list of Str)
(define (list-cnames cs)
  (... (current-name cs) ...))

;; list-anames: Ancestor → (listof Str)
(define (list-anames as)
  (... (ancestor-name as) ...
       (list-names (ancestor-left as)) ...
       (list-names (ancestor-right as)) ...))
```

The contracts give important information that can guide the development.

What are they?

In list-cnames, the name is a single string. Just producing a string violates the contract. We change the body to (list (current-name cs))

In list-anames, there are recursive calls. We combine several lists into one list with append. For the ancestor's name, we do (list (ancestor-name as))

» list-names with an accumulator (1/2)

```
;; list-names: EvoTree → (listof Str)
(define (list-names t)
  (list-names/acc t '()))

;; list-names/acc: EvoTree (listof Str) → (listof Str)
(define (list-names/acc t names)
  (cond [(current? t) (list-cnames t names)]
        [(ancestor? t) (list-anames t names)]))
```

list-name is a wrapper function
The accumulator is set to empty initially

list-name/acc uses a parameter, names. which is the list of names seen so far in the traversal of the tree

We split the problem into two subproblems: listing the names if the tree is a current species and listing the names if it is an ancestor species

» list-names with an accumulator (2/2)

```
;; list-cnames: Current (listof Str) → (listof Str)
(define (list-cnames cs names)
  (cons (current-name cs) names))

;; list-ee-names: EvoEvent (listof Str) → (listof Str)
(define (list-anames as names)
  (cons (ancestor-name as)
        (list-names/acc (ancestor-left as)
                        (list-names/acc (ancestor-right as) names))))

;; Tests
(check-expect (list-names human) '("human"))
(check-expect (list-names e-mammals)
              '("early mammals" "early primates" "human" "chimp" "rat"))
```

list-cnames consumes the list of names seen so far and cons the name of the current species, thus producing a list of strings indicated by the contract

list-anames applies list-names/acc to each of the subtrees. It also needs to supply the names of the nodes visited so far in each of these applications


```
;; WRONG
(append (list (ancestor-name as))
        (list-names/acc (ancestor-left as) names)
        (list-names/acc (ancestor-right as) names))
```

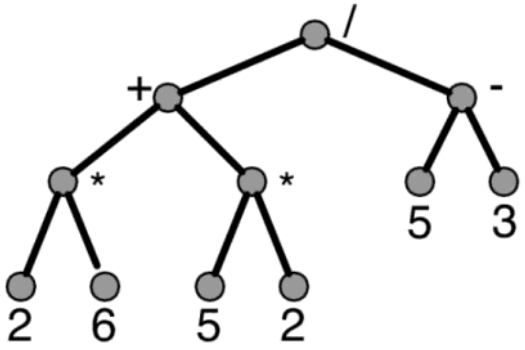
- The problem with this is that when list-names/acc is applied the second time the names argument is wrong
- Nodes that have now been visited (the first application) are not included in names

Binary arithmetic expression trees

Wednesday, October 28, 2020 22:22

Binary expression trees

- The expression ((2*6)+(5*2))/(5/3) can be represented as a **binary expression tree**
- A binary expression tree structures arithmetic expressions into a tree which makes it easy to calculate the value of the expression



Representing binary arithmetic expressions

- **Internal nodes** each have exactly two children
- **Leaves** have number labels
- **Internal nodes** have symbol labels
- Order of children matters
- The structure of the tree is dictated by the expression
- Data definition:

```
;; A binary arithmetic expression (BinExp) is one of:
;; * a Num
;; * a BINode

(define-struct binode (op left right))
;; A Binary arithmetic expression Internal Node (BINode)
;;     is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)
```

- Some examples of binary arithmetic expressions

```
5
```
the smallest possible binary arithmetic expression tree is a single number

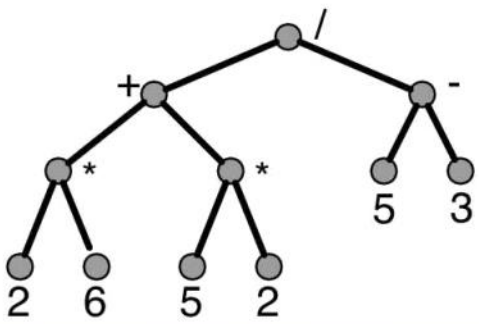
```
(make-binode '* 2 6)
```
the next smallest possible tree is two numbers and an operator

```
(make-binode '+ 2 (make-binode '- 5 3))
```

  - Leaf nodes will always be numbers
  - Internal nodes will always have an operator with left and right subtrees that are binary arithmetic expressions

» A more complex example

```
(make-binode '/
             (make-binode '+ (make-binode '* 2 6)
                             (make-binode '* 5 2))
             (make-binode '- 5 3))
```



This tree represents the expression (2 * 6 + 5 * 2) / (5 - 3)

To evaluate this expression, calculate the value of the left subtree, calculate the value of the right subtree, and then divide

> Templates for binary arithmetic expressions

```
;; binexp-template: BinExp → Any
(define (binexp-template ex)
  (cond [(number? ex) (... ex ...)]
        [(binode? ex) (binode-template ex)]))

;; binode-template: BINode → Any
(define (binode-template node)
  (... (binode-op node) ...
       (binexp-template (binode-left node)) ...
       (binexp-template (binode-right node)) ...))
```

binexp-template is mutually recursive because binexp-template calls binode-template and binode-template calls binexp-template

The base case is when the binary expression is just a number

The binode-template could have been absorbed into the binexp-template, turning the mutual recursion into ordinary recursion

> Evaluating expressions (1/2)

```
;; (eval ex) evaluates the expression ex and produces its value.
(check-expect (eval 5) 5)
(check-expect (eval (make-binode '+ 2 5)) 7)
(check-expect (eval (make-binode '/ (make-binode '- 10 2)
                                    (make-binode '+ 2 2))) 2)

;; eval: BinExp → Num
(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode ex)]))
```

If the binary expression is just a number, the value is that same number. Otherwise, it is an internal node with subtrees, so we apply eval-binode, a function based on the binode-template

> Evaluating expressions (2/2)

```
;; (eval-binode node) evaluates the expression represented by node.
;; eval-binode BINode → Num
(define (eval-binode node)
  (cond [(symbol=? '* (binode-op node))
         (* (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '/ (binode-op node))
         (/ (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '+ (binode-op node))
         (+ (eval (binode-left node)) (eval (binode-right node)))]
        [(symbol=? '- (binode-op node))
         (- (eval (binode-left node)) (eval (binode-right node)))]))
```

For each of the four arithmetic operators, we check which one it is, calculate the values of the left and right subtrees (using eval) and then do the right thing for that operator using the built-in function

» Eval, refactored

```
(define (eval ex)
  (cond [(number? ex) ex]
        [(binode? ex) (eval-binode (binode-op ex)
                                   (eval (binode-left ex))
                                   (eval (binode-right ex)))]))

(define (eval-binode op left-val right-val)
  (cond [(symbol=? op '*) (* left-val right-val)]
        [(symbol=? op '/) (/ left-val right-val)]
        [(symbol=? op '+) (+ left-val right-val)]
        [(symbol=? op '-) (- left-val right-val)]))
```

eval-binode now takes three parameters

General trees

Thursday, October 29, 2020 15:04

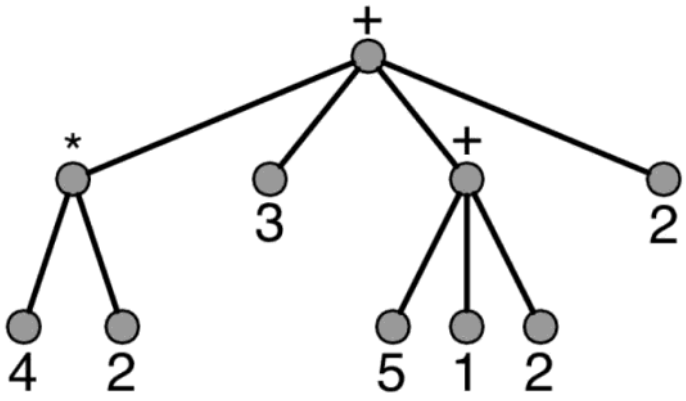
General trees

- Binary trees can be used for a large variety of application areas
- One limitation is the restriction on the number of children
- Trees with an arbitrary number of children (subtrees) in each node are called **general trees**

General arithmetic expressions

- Racket expressions using the functions + and * can have an unbounded number of arguments. For example,

```
(+ (* 4 2)
   3
   (+ 5 1 2)
   2)
```



Representing general arithmetic expression trees

- For a binary arithmetic expression, we defined a structure with three fields: the operator, the first argument and the second argument
- For a general arithmetic expression, we define a structure with two fields: the operator and a list of arguments (a list of arithmetic expressions)

```
;; An Arithmetic Expression (AExp) is one of:
;; * Num
;; * OpNode

(define-struct opnode (op args))
;; An OpNode (operator node) is a
;; (make-opnode (anyof '* '+) (listof AExp)).
```

Developing eval

> Developing `eval`

```
;; (eval exp) evaluates the arithmetic expression exp.
;; Examples:
(check-expect (eval 5) 5)
(check-expect (eval (make-opnode '+ (list 1 2 3 4))) 10)
(check-expect (eval (make-opnode '* (list 2 3 4))) 24)
(check-expect (eval (make-opnode '+ (list 1
                                          (make-opnode '* (list 2 3))
                                          3))) 10)

;; eval: AExp → Num
(define (eval exp)
...)
```


> Completed `eval` and `apply` (1/3)

```
;; (eval exp) evaluates the arithmetic expression exp.
;; Examples:
(check-expect (eval 5) 5)
(check-expect (eval (make-opnode '+ (list 1 2 3 4))) 10)
(check-expect (eval (make-opnode '* (list 2 3 4))) 24)
(check-expect (eval (make-opnode '+ (list 1
                                          (make-opnode '* (list 2 3))
                                          3))) 10)

;; eval: AExp → Num
(define (eval exp)
  (cond [(number? exp) exp]
        [(opnode? exp) (apply (opnode-op exp)
                              (opnode-args exp))]))
```

> Completed `eval` and `apply` (2/3)

```
;; (apply op args) applies the arithmetic operator op to args.
;; Examples:
(check-expect (apply '+ (list 1 2 3 4)) 10)
(check-expect (apply '* (list 2 3 4)) 24)
(check-expect (apply '+ (list 1 (make-opnode '* (list 2 3)))) 7)
(check-expect (apply '+ (list )) 0)
(check-expect (apply '* (list )) 1)
```


> Completed `eval` and `apply` (3/3)

```
1 ;; apply: (anyof '+ '*) (listof AExp) → Num
2 (define (apply op args)
3   (cond [(empty? args) (cond [(symbol=? op '+) 0]
4                              [(symbol=? op '*) 1])]
5         [(symbol=? op '+) (+ (eval (first args))
6                              (apply op (rest args)))]
7         [(symbol=? op '*) (* (eval (first args))
8                              (apply op (rest args)))]))
```

additive identity
multiplicative identity

Alternate data definition

- We can replace the structure opnode and the data definitions for AExp with a list:
- Data definition

```
;; An alternate arithmetic expression (AltAExp) is one of:
;; * a Num
;; * (cons (anyof '* '+) (listof AltAExp))
```

    - Each expression is a list consisting of a symbol (the operator) and a list of expressions

Structuring data using mutual recursion

- **Mutual recursion** arises when complex relationships among data result in cross references between data definitions
- The number of data definitions can be greater than two
- Structures and lists may also be used
- In each case
    - create templates from the data definitions, and
    - create one function for each template

Nested Lists

Friday, October 30, 2020 18:13

Nested lists

- We have flat lists (no nesting)

```
'(a 1 "hello" x)
```

- We also have lists of lists (one level of nesting)

```
'((1 "a") (2 "b") (3 "c"))
```

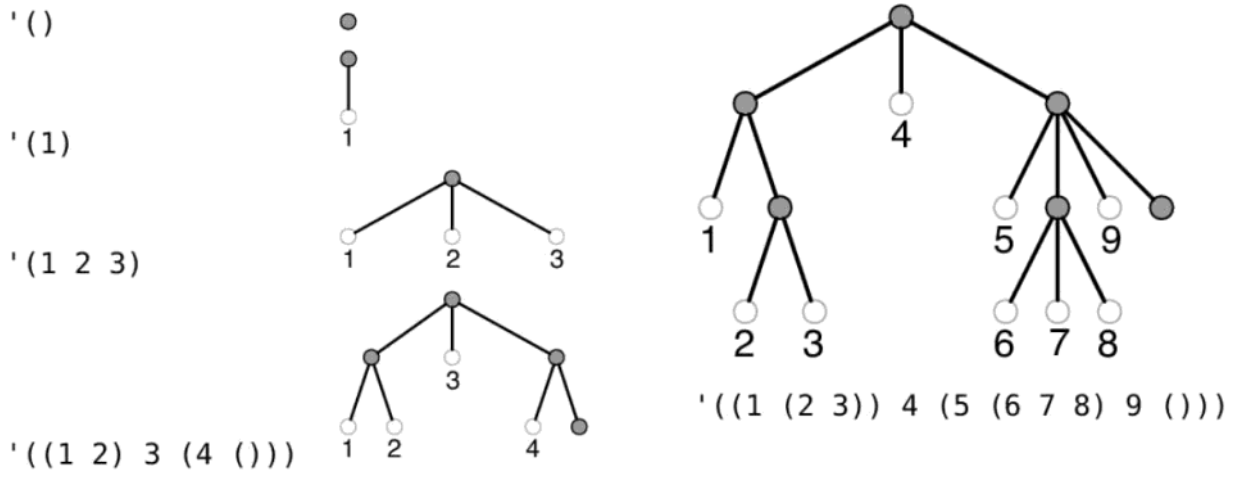
- We now consider **nested lists** (arbitrary nesting)

```
'((1 (2 3))
  4
  (5 (6 7 8) 9 ()))
```

- It is helpful to visualize these nested lists as trees

Visualizing nested lists

- It is often helpful to visualize a nested list as a tree, in which the leaves correspond to the elements of the list, and the internal nodes indicate the nesting



- This is an example of a leaf-labelled tree
  - Labels only appear on the leaves
  - Internal nodes are not labelled
- '(…) can be viewed as a "node" with the contents of the list as the children
- But unlike the nodes we have seen previously, these "nodes" can appear as a leaf when they are empty

Data definition for nested lists

- Sample nested lists

```
'()
'((1 2) 3 (4 ()))
'(1 2 3)
'(1 (2 3) 4)
```

- Observations:
  - A nested list might be empty
  - The first item of a non-empty nested list is either
    - a nested list
    - a single item (a number, not a list)
  - The rest of a non-empty nested list is a nested list

```
;; A nested list of numbers (Nest-List-Num) is one of:
;; * empty
;; * (cons Nest-List-Num Nest-List-Num)
;; * (cons Num Nest-List-Num)
```

- This can be generalized to generic types: (nested-listof X)

```
;; A nested list of X (nested-listof X) is one of:
;; * empty
;; * (cons (nested-listof X) (nested-listof X))
;; * (cons X (nested-listof X))
```

> Template for nested lists

The template follows from the data definition.

```
;; nest-lst-template: (nested-listof X) → Any
(define (nest-lst-template lst)
  (cond [(empty? lst) ...]
        [(list? (first lst)) (... (nest-lst-template (first lst)) ...
                                  (nest-lst-template (rest lst))  ...)]
        [else                (... (first lst) ...
                                  (nest-lst-template (rest lst)) ...)]))
```

The data definition has three clauses, which show up as the three question/answer pairs in the template

As with previous templates, when we see a specific data type (nested-listof X) in the data being consumed, we apply the appropriate template to it (nest-lst template)

> The function count-items

```
;; (count-items nl) counts the number of items in nl.
;; Examples:
(check-expect (count-items '()) 0)
(check-expect (count-items '((10 20) 30)) 3)
(check-expect (count-items '((10 20) () "thirty")) 3)

;; count-items: (nested-listof X) → Nat
(define (count-items lst)
  (cond [(empty? lst) 0]
        [(list? (first lst)) (+ (count-items (first lst))
                                (count-items (rest lst)))]
        [else                (+ 1
                                (count-items (rest lst)))]))
```


» Condensed trace of count-items

```
(count-items '((10 20) 30))
⇒ (+ (count-items '(10 20)) (count-items '(30)))
⇒ (+ (+ 1 (count-items '(20))) (count-items '(30)))
⇒ (+ (+ 1 (+ 1 (count-items '()))) (count-items '(30)))
⇒ (+ (+ 1 (+ 1 0)) (count-items '(30)))
⇒ (+ (+ 1 1) (count-items '(30)))
⇒ (+ 2 (count-items '(30)))
⇒ (+ 2 (+ 1 (count-items '())))
⇒ (+ 2 (+ 1 0)) ⇒ (+ 2 1) ⇒ 3
```

> Flattening a nested list

`flatten` produces a flat list from a nested list.

```
;; (flatten lst) produces a flat list with all the elements of lst.
;; Examples:
(check-expect (flatten '(1 2 3)) '(1 2 3))
(check-expect (flatten '((1 2 3) (a b c))) '(1 2 3 a b c))
(check-expect (flatten '((1 2 3) () (a (b c)))) '(1 2 3 a b c))

;; flatten: (nested-listof X) → (listof X)
(define (flatten lst)
```

We make use of the built-in Racket function `append`.

```
(append '(1 2) '(3 4)) ⇒ '(1 2 3 4)
```

Note that flatten produces all of the items in the consumed nested list in the same order.
We rarely used append so far in the course. Consider using append

- when the first list may have a length greater than one, or
- when there are more than two lists

> Flattening a nested list

```
;; (flatten lst) produces a flat list with all the elements of lst.
;; Examples:
(check-expect (flatten '(1 2 3)) '(1 2 3))
(check-expect (flatten '((1 2 3) (a b c))) '(1 2 3 a b c))
(check-expect (flatten '((1 2 3) () (a (b c)))) '(1 2 3 a b c))

;; flatten: (nested-listof X) → (listof X)
(define (flatten lst)
  (cond [(empty? lst) empty]
        [(list? (first lst)) (append (flatten (first lst))
                                     (flatten (rest lst)))]
        [else                (cons   (first lst)
                                     (flatten (rest lst)))]))
```

append is used when there are two lists

cons is used to add a single item to the front of a list

» Condensed trace of flatten

```
(flatten '((10 20) 30))
⇒ (append (flatten '(10 20)) (flatten '(30)))
⇒ (append (cons 10 (flatten '(20))) (flatten '(30)))
⇒ (append (cons 10 (cons 20 (flatten '()))) (flatten '(30)))
⇒ (append (cons 10 (cons 20 empty)) (flatten '(30)))
⇒ (append (cons 10 (cons 20 empty)) (cons 30 (flatten '())))
⇒ (append (cons 10 (cons 20 empty)) (cons 30 empty))
⇒ (cons 10 (cons 20 (cons 30 empty)))
```

If the first is a single item, cons it on the result of applying flatten to the rest of the list

If the first is a nested list, flatten it and append it with the result of applying flatten tp the rest of list

Motivation

Wednesday, November 4, 2020 0:03

Local definitions

- The functions and special forms we have seen so far can be arbitrarily nested, except define and check-expect
- So far, definitions have to be made "at the top level", outside any expression
- The intermediate language provides the special form local, which contains a series of local definitions plus an expression using them

```
(local [(define x_1 exp_1) ... (define x_n exp_n)] bodyexp)
```

Motivating local definitions

- Consider Heron's formula for the area of a triangle with sides a, b, c:

$$\sqrt{s(s-a)(s-b)(s-c)}, \text{ where } s = (a+b+c)/2$$

- We will describe several possibilities, starting with a direct implementation

Motivation: direct translation

> Motivation: direct translation

```
(define (t-area-v0 a b c)
  (sqrt
   (* (/ (+ a b c) 2)
      (- (/ (+ a b c) 2) a)
      (- (/ (+ a b c) 2) b)
      (- (/ (+ a b c) 2) c))))
```

The repeated computation of $s = (a+b+c)/2$ is awkward.

- Professional programmers try to follow the "**DRY Principle**" - "Don't Repeat Yourself"
- Repeating yourself
  - Allows bugs to be fixed at some places but not others
  - Is often less efficient for the computer
  - Involves more typing
  - Means the code must be understood again each time it occurs

> Motivation: rewrite expressions

We could notice that $s - a = (-a + b + c)/2$, and make similar substitutions.

```
(define (t-area-v1 a b c)
  (sqrt
    (* (/ (+ a b c) 2)
       (/ (+ (- a) b c) 2)
       (/ (+ a (- b) c) 2)
       (/ (+ a b (- c)) 2))))
```

This is slightly shorter, but its relationship to Heron's formula is unclear from just reading the code, and the technique does not generalize.

> Motivation: use a helper function (v1)

We could instead use a helper function.

```
(define (t-area-v2 a b c)
  (sqrt
   (* (s a b c)
      (- (s a b c) a)
      (- (s a b c) b)
      (- (s a b c) c))))

(define (s a b c)
  (/ (+ a b c) 2))
```

This generalizes well to formulas that define several intermediate quantities.

But the helper functions need parameters, which again makes the relationship to Heron's formula hard to see. And there's still repeated code and repeated computations.

- The helper function calculates the value of s in the original formula
- However, it is called four times and also has a terrible name

> Motivation: use a helper function (v2)

We could instead move the computation using *s* into a helper function, and provide the value of *s* as a parameter.

```
(define (t-area-v3 a b c)
  (t-area/s a b c (/ (+ a b c) 2)))

(define (t-area/s a b c s)
  (sqrt (* s (- s a) (- s b) (- s c))))
```

t-area-v3 is actually a wrapper function

This is more readable, and shorter, but it is still awkward because the value of *s* is defined in one function and used in another.

- This version also solves the DRY problem. The code to compute s is not repeated. It does not have to be typed multiple times and does not have to be executed multiple times

Motivation: use local

- The local special form we introduced provides a natural way to bring the definition and use together

```
(define (t-area-v4 a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

- Since local is another special form (like cond) that results in double parentheses, we will use square brackets to improve readability. This is another convention

Semantics

Wednesday, November 4, 2020 19:22

Semantics of local

- Local definitions permits reuse of names
- This is not new to us

```
(define n 10)
(define (myfn n) (+ 2 n))
(myfn 6)
```

This outputs 8, not 12

- The following produces (+ 4 3), not (+ 4 5)

```
(define x 5)
(define (fun a)
  (local [(define x 3)]
    (+ a x)))

(fun 4)
```

Reusing names

- (define n v) binds a value v to a name n
- The name of a formal parameter to a function may reuse a name such as n within the body of that function
- Similarly, a define within a local expression may reuse a name which has already been bound to another value or expression

Informal subsitution rule for local

- The substitution rule works by replacing every name defined in the local with a **fresh name** (**fresh identifier**), a new, unique name that has not been used anywhere else in the program
- Each old name within the local is replaced by the corresponding new name
- Because the new name has not been used elsewhere in the program, the local definitions (with the new name) can now be "promoted" to the top level of the program without affecting anything outside of the local
- We can now use our existing rules to evaluate the program

> Example: evaluating t-area4

We'll need a fresh identifier to replace `s`. We'll use `s_1`, which we just made up.

```
(t-area4 3 4 5) ⇒
(local [(define s (/ (+ 3 4 5) 2))]
  (sqrt (* s (- s 3) (- s 4) (- s 5)))) ⇒
(define s_1 (/ (+ 3 4 5) 2))
(sqrt (* s_1 (- s_1 3) (- s_1 4) (- s_1 5))) ⇒
(define s_1 (/ 12 2))
(sqrt (* s_1 (- s_1 3) (- s_1 4) (- s_1 5))) ⇒
(define s_1 6)
(sqrt (* s_1 (- s_1 3) (- s_1 4) (- s_1 5))) ⇒ ... 6
```

> Substitution rule for `local`

In general, an expression of the form
`(local [(define x_1 exp_1) ... (define x_n exp_n)] bodyexp)`
is handled as follows:

- `x_i` is replaced with a fresh identifier (call it `x_i_new`) everywhere in the `local` expression, for $1 \leq i \leq n$.
- The definitions `(define x_1_new exp_1) ... (define x_n_new exp_n)` are then lifted out (all at once) to the top level of the program, preserving their ordering.
- What remains looks like `(local [] bodyexp')`, where `bodyexp'` is the rewritten version of `bodyexp`. Replace the `local` expression with `bodyexp'`.

All of this (the renaming, the lifting, and removing the `local` with an empty definitions list) is a **single step**.

Revising function substitution

- The previous statement about using our existing rules was not quite correct
- Consider the code below:

```
(define (foo x y)
  (+ (local [(define x y)
             (define z (+ x y))]
       (+ x z))
     x))

(foo 2 3)
```

When (foo 2 3) is evaluated, the x inside the local is not substituted with 2 because x is redefined in local

`(f v_1 ... v_n)` ⇒ `exp'` where `(define (f x_1 ... x_n) exp)` occurs to the left, and `exp'` is obtained by substituting into the expression `exp`, with all occurrences of the formal parameter `x_i` replaced by the value `v_i` (for `i` from 1 to n) *except where `x_i` has been redefined within `exp` (e.g. within a `local`).*

Reasons

Friday, November 6, 2020 19:56

Reasons to use local

- **Clarity**: naming subexpressions
- **Efficiency**: avoid recomputation
- **Encapsulation**: hiding stuff
- **Scope**: reusing parameters

Clarity: naming subexpressions

- A subexpression used twice within a function body always yields the same value
- Using local to give the reused subexpression a name improves the readability of the code
- Recall t-area. Naming the subexpression made the relationship to Heron's Formula clear

```
(define (t-area-v4 a b c)
  (local [(define s (/ (+ a b c) 2))]
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

Clarity: mnemonic names

- Sometimes we choose to use local in order to name subexpressions mnemonically to make the code more readable, even if they are not reused
- This may make the code longer

```
(define-struct coord (x y))
(define (distance p1 p2)
  (sqrt (+ (sqr (- (coord-x p1) (coord-x p2)))
           (sqr (- (coord-y p1) (coord-y p2))))))
```

```
(define (distance p1 p2)
  (local [(define delta-x (- (coord-x p1) (coord-x p2)))
          (define delta-y (- (coord-y p1) (coord-y p2)))]
    (sqrt (+ (sqr delta-x) (sqr delta-y)))))
```

Efficiency: avoid recomputation

- We can use local to avoid recomputation

» Efficiency: `max-list` without local

```
;; (max-list-v2 lon) produces the maximum element of lon
;; Examples:
(check-expect (max-list-v2 (list 6 2 3 7 1)) 7)

;; max-list-v2: (listof Num) → Num
;;   Requires: lon is nonempty
(define (max-list-v2 lon)
  (cond [(empty? (rest lon))  (first lon)]
        [(> (first lon) (max-list-v2 (rest lon))) (first lon)]
        [else  (max-list-v2 (rest lon))]))
```


» Efficiency: `max-list` with local

```
;; (max-list-v4 lon) produces the maximum element of lon
(check-expect (max-list-v4 (list 1 3 2 5 4)) 5)

;; max-list-v4: (listof Num) → Num
;;   requires: lon is nonempty
(define (max-list-v4 lon)
  (cond [(empty? (rest lon))  (first lon)]
        [else
         (local [(define max-rest (max-list-v4 (rest lon)))]
           (cond [(> (first lon) max-rest) (first lon)]
                 [else max-rest]))]))
```


This is the fourth approach we have seen to find the maximum value in a non-empty list:

1. max-list-v1 used the built-in helper function, max
2. max-list-v2 had two recursive applications that led to significant efficiency issues
3. max-list-v3 solved the problem with an accumulator
4. max-list-v4 solves the problem with local

» Efficiency: search-bt-path: original

```
;; search-bt-path-v1: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v1 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]
    [(list? (search-bt-path-v1 k (node-left tree)))
     (cons 'left (search-bt-path-v1 k (node-left tree)))]
    [(list? (search-bt-path-v1 k (node-right tree)))
     (cons 'right (search-bt-path-v1 k (node-right tree)))]
    [else false]))
```

This version of search-bt-path often applies itself twice to the same argument, resulting in an exponential growth in the number of applications as the tree becomes larger

» Efficiency: search-bt-path: helper function

```
;; search-bt-path-v2: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v2 k tree)
  (cond
    [(empty? tree) false]
    [(= k (node-key tree)) '()]
    [else (choose-path-v2 (search-bt-path-v2 k (node-left tree))
                          (search-bt-path-v2 k (node-right tree)))]))

(define (choose-path-v2 left-path right-path)
  (cond [(list? left-path) (cons 'left left-path)]
        [(list? right-path) (cons 'right right-path)]
        [else false]))
```

This version uses a helper method to avoid recomputing values

» Efficiency: search-bt-path: with local

```
;; search-bt-path-v3: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v3 k bt)
  (cond
    [(empty? bt) false]
    [(= k (node-key bt)) '()]
    [else
     (local [(define left-path (search-bt-path-v3 k (node-left bt)))
             (define right-path (search-bt-path-v3 k (node-right bt)))]
       (cond [(list? left-path) (cons 'left left-path)]
             [(list? right-path) (cons 'right right-path)]
             [else false]))]))
```

This version uses local to avoid recomputing values

- This new version of search-bt-path avoids making the same recursive function application twice, and does not require a helper function
- But it still suffers from an inefficiency: we always traverse the entire tree, even if the correct solution is found immediately in the left subtree
- We can avoid the extra search of the right subtree using nested locals

» Efficiency: search-bt-path: with nested local

```
;; search-bt-path-v4: Nat BT → (anyof false (listof Sym))
(define (search-bt-path-v4 k bt)
  (cond
    [(empty? bt) false]
    [(= k (node-key bt)) '()]
    [else
     (local [(define left-path (search-bt-path-v4 k (node-left bt)))]
       (cond [(list? left-path) (cons 'left left-path)]
             [else (local [(define right-path (search-bt-path-v4
                                               k (node-right bt)))]
                     (cond [(list? right-path) (cons 'right right-path)]
                           [else false]))]))]))
```

The left-path is only computed if the outer else clause is reached. The path is checked and, if only required, is a second local evaluated and the right-path is computed

Encapsulation

- **Encapsulation** is the process of grouping things together in a "capsule"
- We have already seen data encapsulation in the use of structures
- There is also an aspect of **information hiding** to encapsulation which we did not see with structures
- The local bindings are not visible (have no effect) outside the local expression. Thus, they can "hide" information from other parts of the programs

Behaviour encapsulation

- Local definitions can bind names to functions as well as values. Evaluating the local expression creates new, unique names for the functions just as for the values
- This is known as **behaviour encapsulation**
- It allows us to move helper functions within the function that uses them
  - They are invisible outside the function
  - They do not clutter the "namespace" at the top level
  - They cannot be used by mistake

Example: sum-list

```
(define (sum-list lon)
  (local [(define (sum-list/acc lst sofar)
            (cond [(empty? lst) sofar]
                  [else (sum-list/acc (rest lst)
                                      (+ (first lst) sofar))]))]
    (sum-list/acc lon 0)))
```

- Advantages of making the accumulatively recursive helper function local:
  - It makes clear that the helper has no use outside of sum-list
  - It facilitates reasoning about the program
- local is often used with wrapper functions
- The helper function - the one that does most of the work - is defined within the local

» Example: Insertion sort

```
(define (isort lon)
  (local [(define (insert n slon)
            (cond [(empty? slon) (cons n empty)]
                  [(<= n (first slon)) (cons n slon)]
                  [else (cons (first slon) (insert n (rest slon)))]))]
    (cond [(empty? lon) empty]
          [else (insert (first lon) (isort (rest lon)))])))
```


Encapsulation and the design recipe

- A function can enclose the cooperating helper functions inside a local, as long as these are not needed by other functions
- When this happens, the enclosing function and all the helpers act as a cohesive unit
- However, the local helper functions require contracts and purposes, but not examples or tests
- The helper functions can be tested by writing suitable tests for the enclosing function
- Make sure that the local helper functions are still tested completely

» Design recipe example

```
;; Full Design Recipe for isort goes here...
(define (isort lon)
  (local [;; (insert n slon) inserts n into slon, preserving the order
          ;; insert: Num (listof Num) → (listof Num)
          ;; requires: slon is sorted in nondecreasing order
          (define (insert n slon)
            (cond [(empty? slon) (cons n empty)]
                  [(<= n (first slon)) (cons n slon)]
                  [else (cons (first slon) (insert n (rest slon)))]))]
    (cond [(empty? lon) empty]
          [else (insert (first lon) (isort (rest lon)))])))
```

Note that insert only needs a purpose and a contract

Examples and tests are not required

Mutual Recursion

- local can also handle mutually recursive functions

```
(local
  [(define (foo z)
     (cond [(= z 0) 0]
           [else (+ 1 (bar (- z 1)))]))
   (define (bar z)
     (cond [(= 0 z) 0]
           [else (+ 1 (foo (- z 1)))]))]
  (foo 5))
```

- foo and bar are each given fresh names and are lifted to the program's top level
- They then act just like other pairs of mutually recursive functions

Scope: reusing parameters

- Making helper functions local can reduce the need to have parameters go along for the ride

```
;; (countup-to n) produces a list of the numbers from 0 to n
;; Example:
(check-expect (countup-to 5) (list 0 1 2 3 4 5))

;; countup-to: Nat → (listof Nat)
(define (countup-to n)
  (countup-from-to 0 n))

;; countup-from-to: Nat Nat → (listof Nat)
(define (countup-from-to from to)
  (cond [(> from to) empty]
        [else (cons from (countup-from-to (add1 from) to))]))

(define (countup-v2 n)
  (local [(define (countup-from from)
            (cond [(> from n) empty]
                  [else (cons from (countup-from (add1 from)))]))]
    (countup-from 0)))
```

The ending point, to, never changes
Hence we describe it as "going along for the ride"

- n no longer needs to be a parameter to countup-from, because it is in scope
- If we evaluate (countup-v2 10), a renamed version of countup-from with n replaced by 10 is lifted to the top level
- If we evaluate (countup-v2 20), a renamed version of countup-from with n replaced by 20 is lifted to the top level

Example: mult-table

- Recall that

```
(check-expect (mult-table 3 4)
              (list (list 0 0 0 0)
                    (list 0 1 2 3)
                    (list 0 2 4 6)))
```

» mult-table: original

```
;; mult-table: Nat Nat → (listof (listof Nat))
(define (mult-table nr nc)
  (rows-to 0 nr nc))

;; (rows-to r nr nc) produces mult. table, rows r...(nr-1)
;; rows-to: Nat Nat Nat → (listof (listof Nat))
(define (rows-to r nr nc)
  (cond [(>= r nr) empty]
        [else (cons (cols-to 0 r nc) (rows-to (add1 r) nr nc))]))

;; (cols-to c r nc) produces entries c...(nc-1) of rth row of mult. table
;; cols-to: Nat Nat Nat → (listof Nat)
(define (cols-to c r nc)
  (cond [(>= c nc) empty]
        [else (cons (* r c) (cols-to (add1 c) r nc))]))
```


» mult-table: with local

```
(define (mult-table2 nr nc)
  (local [;; (rows-to r) produces mult. table, rows r...(nr-1)
          ;; rows-to: Nat → (listof (listof Nat))
          (define (rows-to r)
            (cond [(>= r nr) empty]
                  [else (cons (cols-to 0 r) (rows-to (add1 r)))]))

          ;; (cols-to c r) produces entries c...(nc-1) of rth row
          ;; cols-to: Nat Nat → (listof Nat)
          (define (cols-to c r)
            (cond [(>= c nc) empty]
                  [else (cons (* r c) (cols-to (add1 c) r))]))
          ]
    (rows-to 0)))
```

We will revisit this code again in M14.

- When the functions rows-to and cols-to are lifted to the top level they will have the values for nr and nc "embedded" in the body of the function
- Advantages to this version of mult-table include
  - Making clear that rows-to and cols-to belong to mult-table and are not expected to have other uses
  - Not needing to spend effort to determine that nc and nr do not change as the helper

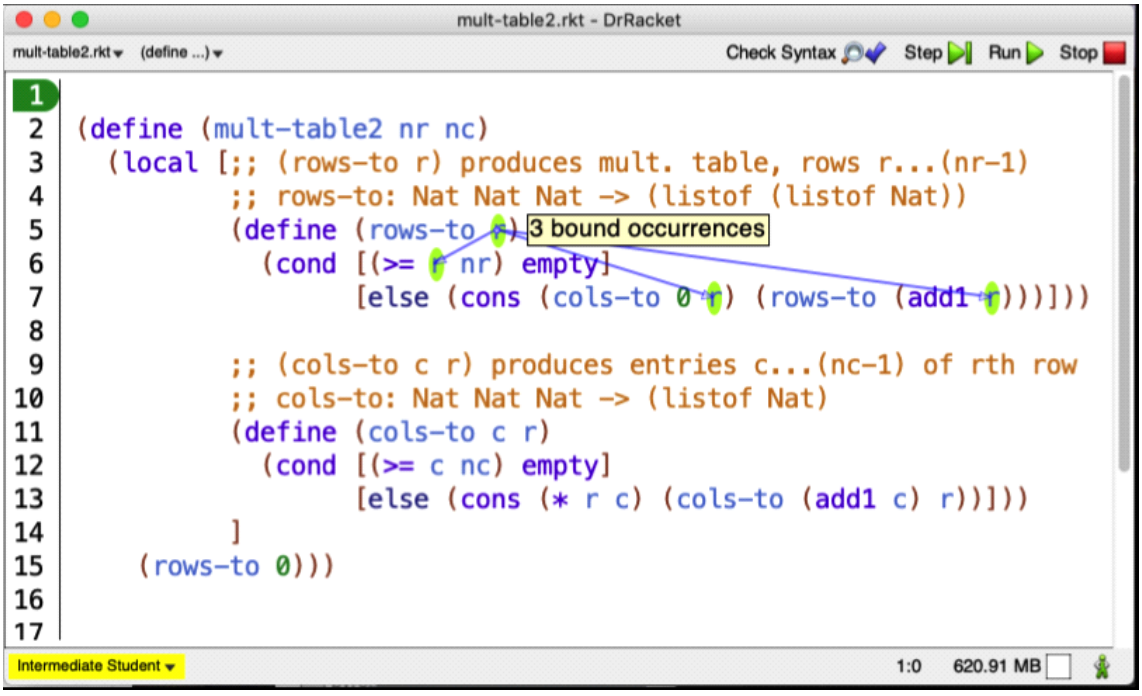
functions execute
- Simplifying the parameters for the helper functions and thus reducing the chances they are mixed up

Terminology

Friday, November 6, 2020 21:55

Terminology associated with local

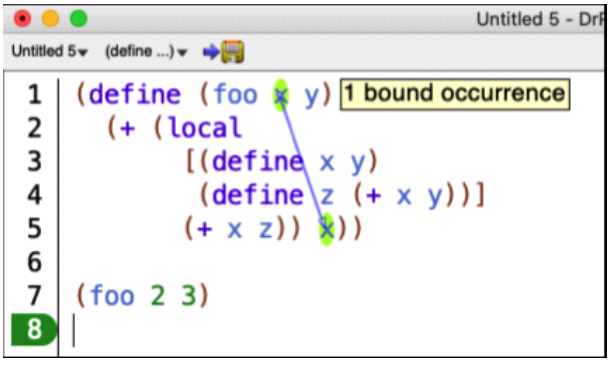
- The **binding occurrence** of a name is its use in a definition, or formal parameter to a function
- The **associated bound occurrences** are the uses of that name that correspond to that binding
- The **lexical scope** of a binding occurrence is all places where that binding has effect, taking note of holes caused by reuse of names
- **Global scope** is the scope of top-level definitions
- If you click "Check Syntax" and then hover your mouse over a binding occurrence, it will draw arrows to the bound occurrences




```
(define (mult-table2 nr nc)
  (local [;; (rows-to r) produces mult. table, rows r...(nr-1)
          ;; rows-to: Nat Nat Nat -> (listof (listof Nat))
          (define (rows-to r)
            (cond [(>= r nr) empty]
                  [else (cons (cols-to 0 r) (rows-to (add1 r)))]))

          ;; (cols-to c r) produces entries c...(nc-1) of rth row
          ;; cols-to: Nat Nat Nat -> (listof Nat)
          (define (cols-to c r)
            (cond [(>= c nc) empty]
                  [else (cons (* r c) (cols-to (add1 c) r))]))
          ]
    (rows-to 0)))
```

- Hovering over a bound occurrence will draw an arrow from the binding occurrence and highlight the other bound occurrences
- The "holes" are the occurrences of x that are not highlighted due to the re-definition of x in local




```
(define (foo x y)
  (+ (local
       [(define x y)
        (define z (+ x y))]
       (+ x z)) x))

(foo 2 3)
```

Intro

Saturday, November 7, 2020 21:14

First class values

- Racket is a functional programming language, primarily because Racket's functions are **first class values**
- Functions have the same status as the other values we have seen. They can be:
  - consumed as function arguments
  - produced as function results
  - bound to identifiers
  - stored in lists and structures
- Functions are first class values in the Intermediate Student (and above) versions of Racket

Consume

Saturday, November 7, 2020 21:26

Consuming functions

- In Intermediate Student a function can consume another function as an argument

```
(define (foo f x y) (f x y))

(foo + 2 3) ⇒ (+ 2 3) ⇒ 5

(foo * 2 3) ⇒ (* 2 3) ⇒ 6

(foo append '(a b c) '(1 2 3))
⇒ (append '(a b c) '(1 2 3))
⇒ '(a b c 1 2 3)
```

Example

- Consider two similar functions, eat-apples and keep-odds

```
(define (eat-apples lst)
  (cond [(empty? lst) empty]
        [(not (symbol=? (first lst) 'apple))
         (cons (first lst) (eat-apples (rest lst)))]
        [else (eat-apples (rest lst))]))

(define (keep-odds  lst)
  (cond [(empty? lst) empty]
        [(odd? (first lst))
         (cons (first lst) (keep-odds  (rest lst)))]
        [else (keep-odds  (rest lst))]))
```

- What these two functions have in common is their general structure
- Where they differ is in the specific predicate used to decide whether an item is removed from the answer or not
- Because functions are first class values, we can write one function to do both these tasks because we can supply the predicate as an argument to that function

> Abstracting `keep-odds` to `my-filter`

```
(define (keep-odds lst)
  (cond [(empty? lst) empty]
        [(odd? (first lst))
         (cons (first lst) (keep-odds (rest lst)))]
        [else (keep-odds (rest lst))]))

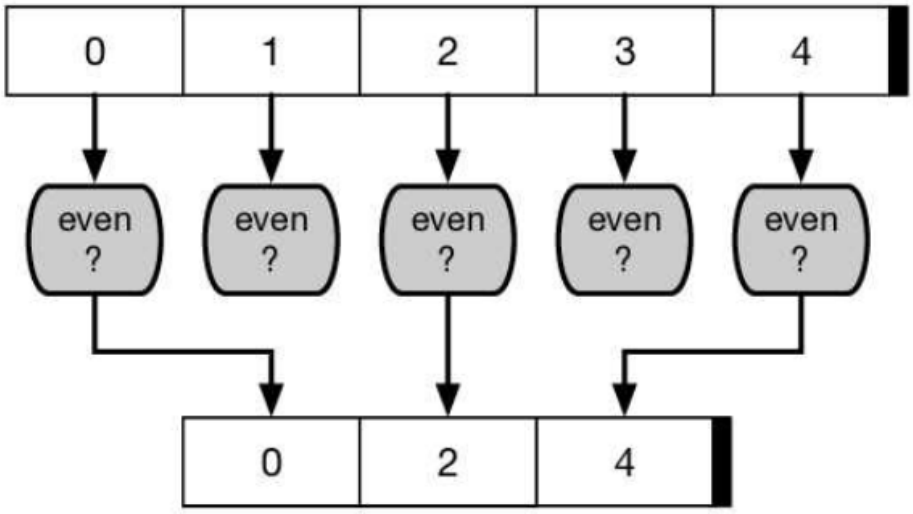
(define (my-filter pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst))
         (cons (first lst) (my-filter pred? (rest lst)))]
        [else (my-filter pred? (rest lst))]))
```


» Tracing `my-filter`

```
(define (my-filter pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst))
         (cons (first lst) (my-filter pred? (rest lst)))]
        [else (my-filter pred? (rest lst))]))

(my-filter even? (list 0 1 2 3 4))
⇒ (cond [(empty? (list 0 1 2 3 4)) empty]
        [(even? (first (list 0 1 2 3 4)))
         (cons (first (list 0 1 2 3 4))
               (my-filter even? (rest (list 0 1 2 3 4))))]
        [else (my-filter even? (rest (list 0 1 2 3 4)))]))
⇒ (cons 0 (my-filter even? (list 1 2 3 4)))
⇒ (cons 0 (my-filter even? (list 2 3 4)))
⇒*(cons 0 (cons 2 (cons 4 empty)))
```


- my-filter performs the same actions as the built-in function filter
- filter handles the general operation of selectively keeping items on a list
- Functions such as filter that consume a (listof X) and a function to generalize it are called **abstract list functions** (abbreviated ALFs) or higher order functions



Using my-filter

```
(define (keep-odds lst) (my-filter odd? lst))

(define (not-symbol-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples lst) (my-filter not-symbol-apple? lst))
```

- The function filter. which behaves identically to my-filter, is built into Intermediate Student and full Racket
- filter and other abstract list functions provided in Racket are used to apply common patterns of simple recursion
- Functions like eat-apples that require a custom predicate are often ideal candidates for local
- For example, instead of the code shown on the slide, consider

```
(define (eat-apples lst)
  (local [(define (not-symbol-apple? item) (not (symbol=? item 'apple)))]
    (filter not-symbol-apple? lst)))
```

Advantages of functional abstraction

- **Functional abstraction** is the process of creating abstract functions such as filter
- Advantages include
  - reducing code size
  - avoiding cut-and-paste
  - fixing bugs in one place instead of many
  - improving one functional abstraction improves many applications

Produce

Sunday, November 8, 2020 14:58

Producing functions

- We have seen that local could be used to create functions during a computation, to be used in evaluating the body of the local
- Because functions are values, the body of the local can produce such a function as a value

Example: make-adder

```
(define (make-adder n)
  (local
    [(define (f m) (+ n m))]
    f))
```

Trace of (make-adder 3)

```
 (make-adder 3)
⇒ (local [(define (f m) (+ 3 m))] f)
⇒ (define (f_1 m) (+ 3 m)) f_1
```

- (make-adder 3) is the renamed function f_1, which is a function that adds 3 to its argument
- We can do the following things:
  - apply this function immediately
  - use it in another expression
  - put it in a data structure

Example: make-adder applied immediately

```
 ((make-adder 3) 4)
⇒ ((local [(define (f m) (+ 3 m))] f) 4)
⇒ (define (f_1 m) (+ 3 m)) (f_1 4)
⇒ (+ 3 4) ⇒ 7
```

- Before:
  - First position in an application must be a built-in or user-defined function
  - A function name has to follow an open parenthesis
- Now:
  - First position can be an expression (computing the function to be applied). Evaluate it along with the other arguments
  - A function application can have two or more open parentheses in a row: ((make-adder 3) 4)

A note on scope

```
(define (add3 m)
  (+ 3 m))
```

```
(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))
```

- In add3, the parameter m is of no consequence after add3 is applied. Once add3 produces its value, m can be safely forgotten
- In make-adder, after the parameter n is applied it does have a consequence.

It is embedded into the result, f, where it is "remembered" and used again, potentially many times

Producing and consuming functions

- Using local to produce a function gives us a way to create semi-custom functions "on the spot" to use in expressions.
- This is particularly useful with Abstract List Functions (ALFs) such as filter

Bind

Sunday, November 8, 2020 15:54

Binding functions to identifiers

- The result of make-adder can be bound to an identifier and then used repeatedly

```
(define add2 (make-adder 2))
(define add3 (make-adder 3))

(add2 3) ⇒ 5
(add3 10) ⇒ 13
(add3 13) ⇒ 16
```

- We can bind a value like 3 or "Hello" to an identifier to make a constant
- Because functions are values, we can do that with functions, too
- (make-adder 2) produces a function. (define add2 (make-adder 2)) gives that function a name so it can be used over and over

Tracing a bound identifier

```
 (define add2 (make-adder 2))
⇒ (define add2 (local [(define (f m) (+ 2 m))] f))
⇒ (define (f_1 m) (+ 2 m)) ; rename and lift out f
   (define add2 f_1)

 (add2 3)
⇒ (f_1 3)
⇒ (+ 2 3)
⇒ 5
```

Store

Sunday, November 8, 2020 16:16

Storing functions in lists and structures

- Recall our code in module 11 for evaluating arithmetic expressions

```
(define-struct opnode (op args))
;; An OpNode is a (make-opnode (anyof '* '+) (listof AExp)).
;; An AExp is (anyof Num OpNode)

;; (eval exp) evaluates the arithmetic expression exp.
;; Examples:
(check-expect (eval 5) 5)
(check-expect (eval (make-opnode '+ (list 1 2 3 4))) 10)
(check-expect (eval (make-opnode '* (list ))) 1)

;; eval: AExp → Num
```

> Example: `eval` and `apply` from M11

```
;; eval: AExp → Num
(define (eval exp)
  (cond [(number? exp) exp]
        [(opnode? exp) (my-apply (opnode-op exp) (opnode-args exp))]))

;; (my-apply op args) applies the arithmetic operator op to args.
;; my-apply: (anyof '+ '*) (listof AExp) → Num
(define (my-apply op args)
  (cond [(empty? args) (cond [(symbol=? op '+) 0]
                             [(symbol=? op '*) 1])]
        [(symbol=? op '+) (+ (eval (first args))
                             (my-apply op (rest args)))]
        [(symbol=? op '*) (* (eval (first args))
                             (my-apply op (rest args)))]))
```

Code that is underlined means it repeats at least once in my-apply

> Example: Evaluating expressions with functions

In `opnode` we can replace the symbol representing a function with the function itself:

```
(define-struct opnode (op args))
;; An opnode is a (make-opnode (anyof ... (listof AExp)))
;; An AExp is (anyof Num opnode)

(check-expect (eval 3) 3)
(check-expect (eval (make-opnode + '(2 3 4))) 9)
(check-expect (eval (make-opnode + '())) 0)
```

The key insight is that we will store the function itself (a first class value) in the opnode structure
The … in opnode's data definition is because we do not have a way to express a function's type

Some observations about Intermediate Student that will be handy:

```
(+ 1 2) ⇒ 3
(+ 1) ⇒ 1
(+ ) ⇒ 0
```

(+) produces the additive identity, 0

```
(* 2 3) ⇒ 6
(* 2) ⇒ 2
(* ) ⇒ 1
```

(*) produces the multiplicative identity, 1

> Example: Evaluating expressions with functions

`eval` does not change. Here are the changes to `my-apply`:

Old:

```
(define (my-apply op args)
  (cond [(empty? args) (cond [(symbol=? op '+) 0]
                             [(symbol=? op '*) 1])]
        [(symbol=? op '+) (+ (eval (first args))
                             (my-apply op (rest args)))]
        [(symbol=? op '*) (* (eval (first args))
                             (my-apply op (rest args)))]))
```

New:

```
(define (my-apply op args)
  (cond [(empty? args) (op )]
        [else  (op (eval (first args))
                   (my-apply op (rest args)))]))
```


- my-apply is now consuming a function rather than a symbol
- In the base case, (op) produces the identity appropriate to the operator
- In the recursive case. The op is applied to the first expression on the list of arguments and the result of applying the operator to the rest of the arguments
- This works for any binary function that is also defined for zero arguments

> Example: Functions in a table (1/2)

```
(define trans-table (list (list '+ +)
                          (list '* *)))
```

Key idea: make a dictionary (implemeted as an association list) mapping symbols to the functions they represent
lookup-al consumes a symbol and produces the corresponding function

```
(define trans-table (list (list '+ +)
                          (list '* *)))
```

association list) mapping symbols to the functions they represent
lookup-al consumes a symbol and produces the corresponding function

```
;; (lookup-al key al) finds the value in al corresponding to key
;; lookup-al: Sym AL → ???
(define (lookup-al key al)
  (cond [(empty? al) false]
        [(symbol=? key (first (first al))) (second (first al))]
        [else (lookup-al key (rest al))]))
```

Now `(lookup-al '+ trans-table)` produces the function +.

```
((lookup-al '+ trans-table) 3 4 5) ⇒ 12
```

The double parenthesis here means we are evaluating look-al to obtain the function to use

> Example: Functions in a table (2/2)

```
;; (eval ex) evaluates the arithmetic expression ex.
;; eval: AExp → Num
(define (eval ex)
  (cond [(number? ex) ex]
        [(cons? ex) (my-apply (lookup-al (first ex) trans-table)
                              (rest ex))]))

;; (my-apply op exlist) applies op to the list of arguments.
;; my-apply: ??? (listof AExp) → Num
(define (my-apply op args)
  (cond [(empty? args) (op )]
        [else  (op (eval (first args))
                   (my-apply op (rest args)))]))
```

We use lookup-al to translate a symbol into a function it represents. Then it becomes the op parameter to my-apply
We are using a list rather than the opnode structure. Therefore we get the operator with (first ex) and the arguments with (rest ex)

my-apply consumes a function and a list of arguments, which is identical to the previous version

Summary: Functions in lists and structures

- We have stored functions in both a structure and a list
- Using a function instead of a symbol get rid of repetitive code in my-apply
- Using quote notation makes our expressions more succinct, but forced us to deal again with symbols to represent functions
- Putting symbols and functions in an association list provided a clean solution
  Adding a new binary function (also defined for zero arguments) only requires a new line in trans-table

Functions as first class values (summary)

- As a first class value, we can do anything with a function that we can do with other values
  - consume: my-apply consumes the operator
  - produce: lookup-al looks up a symbol, producing the corresponding function
  - bind: results of lookup-al to op (a parameter)
  - store: stored in trans-table

Contracts and types

Monday, November 9, 2020 16:37

Contracts and types

- **Contracts** describe the type of data consumed and produced by a function
- What is the type of a function consumed or produced by another function?
- We can use the contract for a function as its type
- For example, the type of > is (Num Num -> Bool), the contract of that function

> Contracts as types: Example

```
(define trans-table (list (list '+ +)
                          (list '* *)))

;; (lookup-al key al) finds the value in al corresponding to key
;; lookup-al: Sym (listof (list Sym (Num Num → Num))) →
;;       (anyof false (Num Num → Num))
(define (lookup-al key al)
  (cond [(empty? al) false]
        [(symbol=? key (first (first al))) (second (first al))]
        [else (lookup-al key (rest al))]))
```


Contracts for abstract list functions

- filter consumes a function and a list, and produces a list
- We might be tempted to conclude that its contract is
- `(Any → Bool) (listof Any) → (listof Any)`
- The application (filter odd? (list 1 2 3)) does not obey the contract (the contract for odd? is Int -> Bool) but still works as desired
- There is a relationship among the two arguments to filter and the result of filter

Parametric types

- An application of (filter pred? lst) can work on any type of list, but the predicate provided should consume elements of that type of list
- In other works, we have a dependency between the type of the predicate and the type of the list
- To express this, we use a **type variable,** such as X, and use it in different places to indicate where the same type is needed
- It is a symbol that stands for some specific but currently unknown type

The contract for filter

- filter consumes a list of type (listof X)
- This implies that the predicate must consume an X. The predicate must also produce a Bool. It thus has a contract (and type) of (X -> Bool)
- filter produces a list of the same type it eonsumes
- Therefore the contract for filter is:

```
;; filter: (X → Bool) (listof X) → (listof X)
```

- Here X stands for the unknown data type of the list
- filter is **polymorphic** or **generic**. It works on many different types of data

Using contracts to understand

- Many of the difficulties one encounters in using abstract list functions can be overcome by careful attention to contracts
- For example, the contract for the function provided as an argument to filter says that it consumes one argument and produces a Boolean value.
- This means we must take care to never use filter with an argument that is a function consuming two variables, or producing a number

Example

Monday, November 9, 2020 17:21

Example: Simulating structures

- We can use the ideas of producing and binding functions to simulate structures
- Consider a structure representing a point:

```
(define-struct point (x y))
;; A Point is a (make-point Num Num)
```

- This can be simulated with a function:

```
;; (mk-point x y) produces a "structure" representing (x,y).
;; mk-point: Num Num → _____
(define (mk-point x y)
  (local [(define (symbol-to-value s)
            (cond [(symbol=? s 'x) x]
                  [(symbol=? s 'y) y]))]
    symbol-to-value))
```

Tracing mk-point

```
(define p1 (mk-point 3 4))
⇒ (define p1 (local [(define (symbol-to-value s)
                       (cond [(symbol=? s 'x) 3]
                             [(symbol=? s 'y) 4]))]
               symbol-to-value))
```

- Notice how the parameters have been substituted into the local definition
- We now rename symbol-to-value and lift it out

```
⇒ (define (symbol-to-value_1 s)
    (cond [(symbol=? s 'x) 3]
          [(symbol=? s 'y) 4]))
  (define p1 symbol-to-value_1)
```

- p1 is now a function with the x and y values we supplied to mk-point coded in
- To get out the x value, we can use (p1 'x)

```
(p1 'x) ⇒ (symbol-to-value_1 'x) ⇒ ... ⇒ 3
```

- We can define a few convenience functions to simulate the structure accessor functions point-x and point-y

```
(define (point-x p) (p 'x))
(define (point-y p) (p 'y))
```

- If we apply mk-point again with different values, it will produce a differnet rewritten and lifted version of symbol-to-value, say symbol-to-value_2

Simulating structures summary

- The result of a particular application, say (mk-point 3 4) is a "copy" of symbol-to-value with 3 and 4 subsituted for x and y, repectively
- That "copy" can be used much later, to retrieve the value of x or y that was supplied to mk-point
- This is possible because the "copy" of symbol-to-value, even though it was defined in a local definition, survives after the evaluation of the local is finished

Anonymous functions

Thursday, November 12, 2020 16:13

Abstraction

- **Abstraction** is the process of finding similarities or common aspects, and forgetting unimportant differences
- Example: writing a function
  - The differences in parameter values are forgotten, and the similarity is captured in the function body
  - Similarities between functions are captured in function templates

Anonymous functions

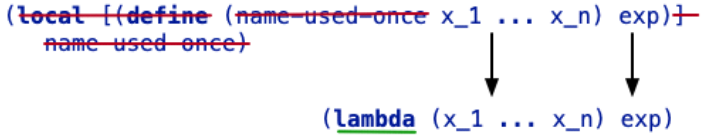
```
(define (not-symbol-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples lst) (filter not-symbol-apple? lst))
```

- not-symbol-apple? is unlikely to be needed elsewhere
- We can avoid cluttering the top level with such definitions by putting them in local expressions

```
(define (eat-apples lst)
  (local [(define (not-symbol-apple? item)
            (not (symbol=? item 'apple)))]
    (filter not-symbol-apple? lst)))
```

Introducing lambda

- ```
  (local [(define (name-used-once x_1 ... x_n) exp)]
    name-used-once)
  ```
  can also be written as
  ```
  (lambda (x_1 ... x_n) exp)
  ```
- lambda can be thought of as "make-function"
- It can be used to create a function which we can then use as a value, for example, as the value of the first argument of filter
- When a function produced in a local is only used once, lambda gets rid of the code that is not actually needed



We need lambda, a list of parameter names and the expression using them
  ```

> Example: define `eat-apples` with `lambda`

We can use **`lambda`** to replace

```
(define (eat-apples lst)
  (filter (local [(define (not-symbol-apple? item)
                    (not (symbol=? item 'apple)))]
            not-symbol-apple?)
          lst))
```

with the following:

```
(define (eat-apples lst)
  (filter (lambda (item) (not (symbol=? item 'apple))) lst))
```

But how does this work? As usual, we'll approach it with a trace.

Other examples of using lambda with filter

```
(define lst '(3 5 9 5 5 4))
(filter (lambda (x) (= 5 x)) lst)                    ==> '(5 5 5)
(filter (lambda (x) (and (<= 3 x) (<= x 5))) lst)    ==> '(3 5 5 5 4)
(filter (lambda (s) (char>? s #\Z)) '(#\B #\a #\y)) ==> '(#\a #\y)
```

» Tracing `eat-apples`

```
(define (eat-apples lst)
  (my-filter (lambda (item) (not (symbol=? item 'apple))) lst))

(eat-apples '(pear apple))
⇒ (my-filter (lambda (item) (not (symbol=? item 'apple)))) '(pear apple))
⇒ (cond [(empty? '(pear apple)) empty]
        [((lambda (item) (not (symbol=?  item 'apple))))
         (first '(pear apple))
         (cons (first '(pear apple))
               (my-filter (lambda (item) (not (symbol=? item 'apple))))
                  (rest '(pear apple))))]
        [else (my-filter (lambda (item) (not (symbol=? item 'apple))))
                         (rest '(pear apple)))]))
```

What does the underlined expression mean?


```
((lambda (item) (not (symbol=? item 'apple))))  (first '(pear apple)))
```

- The double parentheses indicates that we need to compute the function (the inner expression) to apply to the arguments (the outer expression)
- In this case, compute means creating the function using lambda
- Lambda expressions are already in the simplest form, so the next step in the trace is to reduce the arguments to values:

  ⇒ `((lambda (item) (not (symbol=? item 'apple))))  'pear)`
- Finally, each argument is matched with the corresponding parameter and then substituted into the function's body expression each place that parameter appears. The entire expression is replaced with the rewritten body expression

  ⇒ `(not (symbol=? 'pear 'apple))`

> Using `lambda`

We can use `lambda` to simplify `make-adder`. Instead of

```
(define (make-adder n)
  (local [(define (f m) (+ n m))]
    f))
```

we can write:

```
(define (make-adder n)
  (lambda (m) (+ n m)))
```

```
(define (make-adder n)
  (lambda (m) (+ n m)))
```

Syntax

Thursday, November 12, 2020 17:12

Syntax and semantics

- When we first encountered ((make-adder 3) 4), we noted the differences in function application:
  - Before Module 13
    - First position in an application must be a built-in or user-defined function
    - A function name has to follow an open parenthesis
  - Module 13 and later
    - First position can be an expression (computing the function to be applied). Evaluate it along with the other arguments
    - A function application can have two or more open parentheses in a row:

```
((make-adder 3) 4) or

((lambda (x y) (+ x y x)) 1 2)
```

- These observations are also true of using lambda

> Substitution rule

We need a rule for evaluating applications where the function being applied is anonymous (a `lambda` expression).

`((lambda (x_1 ... x_n) exp) v_1 ... v_n)` ⇒ `exp'`

where `exp'` is `exp` with all occurrences of `x_1` replaced by `v_1`, all occurrences of `x_2` replaced by `v_2`, and so on.

As an example:

```
((lambda (x y) (* (+ y 4) x)) 5 6)
⇒ (* (+ 6 4) 5)
⇒ ... ⇒ 50
```

> Example: Tracing with lambda (1)

```
(define foo (lambda (x) (+ 10 x)))

(foo 5)
⇒ ((lambda (x) (+ 10 x)) 5)
⇒ (+ 10 5)
⇒ 15
```

In this example, foo is defined as a constant.

Like any constant, its value needs to be substituted into the expression

> Example: Tracing with lambda (2)

Here's `make-adder` rewritten using **`lambda`**.

```
(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

What is `((make-adder 3) 4)`?

> Example: Tracing with lambda (2)

```
(define make-adder
  (lambda (x)
    (lambda (y)
      (+ x y))))
```

```
(define make-adder (lambda (x) (lambda (y) (+ x y))))
((make-adder 3) 4) ⇒    ;; substitute the lambda expression
(((lambda (x) (lambda (y) (+ x y))) 3) 4) ⇒
((lambda (y) (+ 3 y)) 4) ⇒
(+ 3 4) ⇒ 7
```

`make-adder` is defined as a constant using lambda. Like any other constant, `make-adder` is replaced by its value (the **`lambda`** expression).

lambda and function definitions

- lambda underlies the definition of functions
- Until now, we have had two different types of definitions

```
;; a definition of a numerical constant
(define interest-rate 3/100)
;; a definition of a function to compute interest
(define (interest-earned amount) (* interest-rate amount))
```

- But there is really only one kind of define, which binds a name to a value
- Internally,

```
(define (interest-earned amount) (* interest-rate amount))
```

is translated to

```
(define interest-earned (lambda (amount) (* interest-rate amount)))
```

which binds the name interest-earned to the function value

```
(lambda (amount) (* interest-rate amount))
```

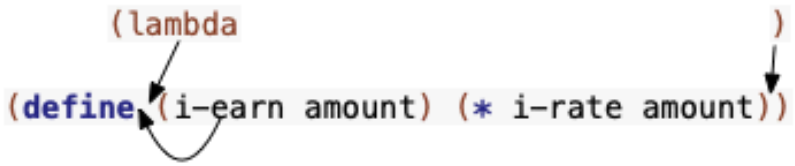
- Using short names to make the transformation

```
(define i-rate 0.03)
(define (i-earn amount) (* i-rate amount))
```

```
(define i-earn (local [(define (f amount) (* i-rate amount))] f))
```

```
(define i-earn (lambda (amount) (* i-rate amount)))
```

```
(define i-earn (lambda (amount) (* i-rate amount)))
```




```
(define i-earn (lambda (amount) (* i-rate amount)))
```

Example

Sunday, November 15, 2020 15:41

> Example: character transformation in strings

We'd like a function, `transform`, that transforms one string into another according to a set of rules that are specified when it is applied.

In one application, we might want to change every instance of 'a' to a 'b'. In another, we might transform lowercase characters to the equivalent uppercase character and digits to '*'.

```
(check-expect (transform "abracadabra" ...) "bbrbcbdbbrb")
(check-expect (transform "Testing 1-2-3" ...) "TESTING *-*-*")
```

We use ... to indicate that we still need to supply some arguments.

> Example: inspiration

We could imagine `transform` containing a **`cond`**:

```
(cond [(char=? ch #\a) #\b]
      [(char-lower-case? ch) (char-upcase ch)]
      [(char-numeric? ch) #\*]
      ...)
```

But this fails for a number of reasons:

- The rules are "hard-coded"; we want to supply them when `tranform` is applied.
- A lower case 'a' would always be transformed to 'b'; never to 'B'.

But the idea is inspiring...

> Example: core idea

Suppose we supplied `transform` with a list of question/answer pairs:

```
;; A TransformSpec is one of:
;; * empty
;; * (cons (list Question Answer) TransformSpec)
```

Like **cond**, we could work our way through the `TransformSpec` with each character. If the Question produces `true`, then apply the Answer to the character. If the Question produces `false`, go on to the next Question/Answer pair.

What are the types for `Question` and `Answer`?

Map

Sunday, November 15, 2020 16:15

Deriving `map`

Here are two early list functions we wrote.

```
(define (negate-list   lst)
  (cond [(empty? lst)     empty]
        [else (cons (- (first lst))
                    (negate-list (rest lst)))]))

(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [else (cons (sr->tr (first payroll))
                    (compute-taxes (rest payroll)))]))
```


> Abstracting another set of examples

We look for a difference that can't be explained by renaming (it being what is applied to the first item of a list) and make that a parameter.

```
(define (compute-taxes payroll)
  (cond [(empty? payroll) empty]
        [else (cons (sr->tr (first payroll))
                    (compute-taxes (rest payroll)))]))

(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                    (my-map f (rest lst)))]))
```

The underlined portions of my-map are the changes required to account for the differences between negate-list and compute-taxes

> Tracing `my-map`

```
(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                    (my-map f (rest lst)))]))
```

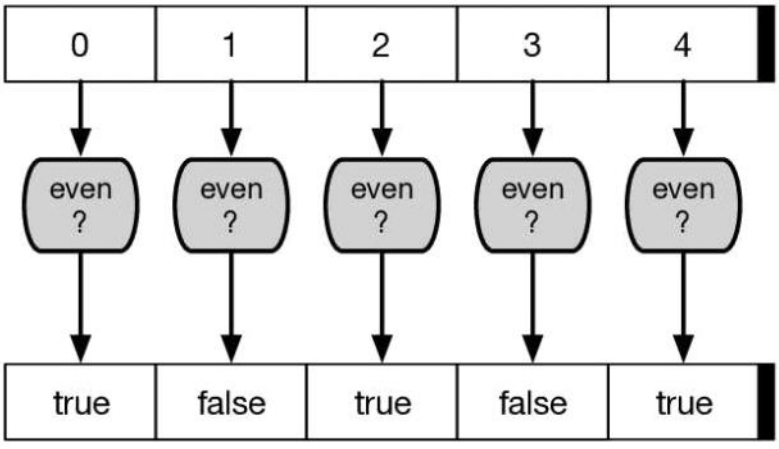
```
(my-map sqr (list 3 6 5))
⇒ (cons 9 (my-map sqr (list 6 5)))
⇒ (cons 9 (cons 36 (my-map sqr (list 5))))
⇒ (cons 9 (cons 36 (cons 25 (my-map sqr empty))))
⇒ (cons 9 (cons 36 (cons 25 empty)))
```

`my-map` performs the general operation of transforming a list element-by-element into another list of the same length.

Effects of my-map

- (my-map f (list x_1 x_2 … x_n)) has the same effect as evaluating (list (f x_1) (f x_2) … f(x_n))

```
(my-map even? '(0 1 2 3 4))
```



Using my-map

- We can use my-map to give short definitions of a number of functions we have written to consume lists

```
(define (negate-list lst)   (my-map - lst))
(define (compute-taxes lst) (my-map sr->tr lst))
```

The contract for my-map

- my-map consumes a function and a list, and produces a list

Foldr

Tuesday, November 17, 2020 17:38

Abstract list functions that produce values

- The functions we have worked with so far consume and produce lists
- What about abstracting from functions such as count-symbols and sum-of-numbers, which consume lists and produce simple values?

> Examples

```
(define (sum-of-numbers lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst)
                 (sum-of-numbers (rest lst)))]))

(define (prod-of-numbers lst)
  (cond [(empty? lst) 1]
        [else (* (first lst)
                 (prod-of-numbers (rest lst)))]))

(define (all-true? lst)
  (cond [(empty? lst) true]
        [else (and (first lst)
                   (all-true? (rest lst)))]))
```


Similarities and differences

- Each example has a base case which is a value to be returned when the list supplied is empty
- Each example is applying some function to combine (first lst) and the result of a recursive function application with argument (rest lst)

Comparison to the list template

```
(define (list-template lst)
  (cond [(empty? lst) ...]      base value
        [else (... (first lst) ...
                   combine  (list-template (rest lst)) ...)]))
```

- We replace the first ellipsis by a base value
- We replace the rest of the ellipses by some function which combines (first lst) and the result of a recursive function application on (rest lst)
- This suggests passing the base value and the combining function as parameters to an abstract list function

> The abstract list function `foldr`

```
(define (my-foldr combine base lst)
  (cond [(empty? lst) base]
        [else (combine (first lst)
                       (my-foldr combine base (rest lst)))]))
```

`foldr` is also a built-in function in Intermediate Student With Lambda.

combine: a function that combines two values, the first thing on the list and the result of processing the rest of the list

base: a value to provide when we hit the base case

> Tracing `my-foldr`

```
(define (my-foldr combine base lst)
  (cond [(empty? lst) base]
        [else (combine (first lst)
                       (my-foldr combine base (rest lst)))]))
```

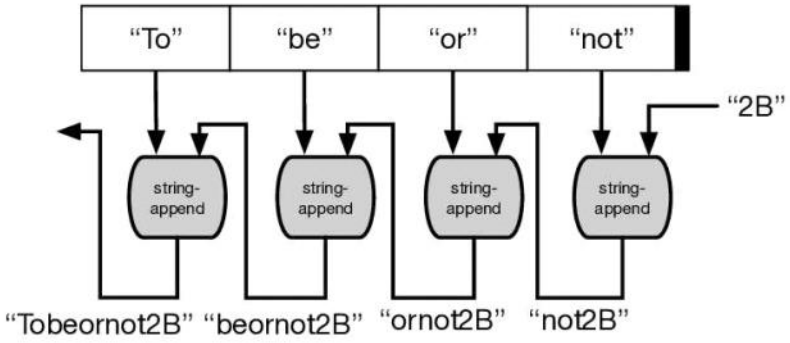
```
(my-foldr f 0 (list 3 6 5)) ⇒
(f 3 (my-foldr f 0 (list 6 5))) ⇒
(f 3 (f 6 (my-foldr f 0 (list 5))) ⇒
(f 3 (f 6 (f 5 (my-foldr f 0 empty))) ⇒
(f 3 (f 6 (f 5 0))) ⇒ ...
```

Intuitively, the effect of the application `(foldr f b (list x_1 x_2 ... x_n))` is to compute the value of the expression `(f x_1 (f x_2 (... (f x_n b))))`.

> Tracing `my-foldr`

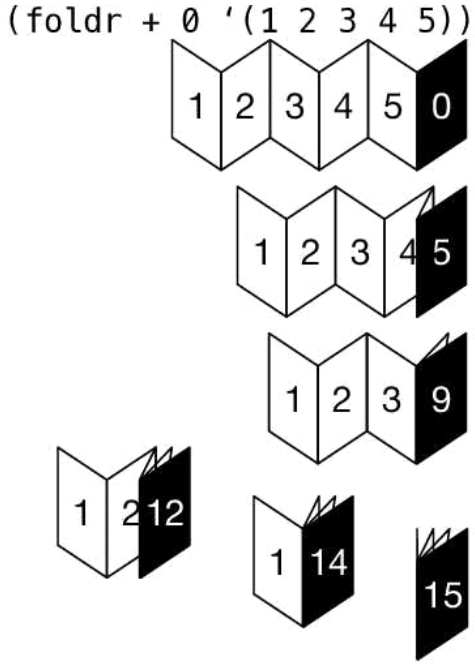
```
(foldr f b (list x_1 x_2 ... x_n)) ⇒ (f x_1 (f x_2 (... (f x_n b))))

(foldr string-append "2B" '("To" "be" "or" "not")) => "Tobeornot2B"
```



foldr

- foldr is short for "fold right"
- The reason for the name is that it can be viewed as "folding" a list using the provided combine function, starting from the right-hand end of the list



Contract for foldr

- (X Y -> Y) Y (listof X) -> Y

> Using `foldr`

```
(define (sum-of-numbers lst) (foldr + 0 lst))
```

```
(define (sum-of-numbers lst) (foldr + 0 lst))
```

If `lst` is `(list x_1 x_2 ... x_n)`, then by our intuitive explanation of `foldr`, the expression `(foldr + 0 lst)` reduces to

```
(+ x_1 (+ x_2 (+ ... (+ x_n 0))))
```

Thus `foldr` does all the work of the template for processing lists, in the case of `sum-of-numbers`.

Using foldr

- The function provided to foldr consumes two parameters - one is an element in the list which is an argument to foldr, and one is the result of reducing the rest of the list
- Sometimes one of those arguments should be ignored, as in the case of using foldr to compute count-symbols

```
(define (count-symbols lst)
  (cond [(empty? lst) 0]
        [else (+ 1
                 (count-symbols (rest lst)))]))
```

- The first argument to the function provided to foldr contributes 1 to the count. Its actual value is irrelevant
- In this case, the function provided to foldr can ignore the value of the first parameter and just add 1 to the result of recursing on the rest.

```
(define (count-symbols lst) (foldr (lambda (x rror) (add1 rror)) 0 lst))
```

- The function provided to foldr, namely

```
(lambda (x rror) (add1 rror))
```

ignores its first argument

- Its second argument is the result of recursing on the rest (rror) of the list. In this case it is the length of the rest of the list, to which 1 must be added

Using foldr to produce lists

- So far, the funcitons we have been providing to foldr have produced numerical results, but they can also produce cons expressioons
- foldr is an abstraction of simple recursion on lists, so we should be able to use it to implement negate-list, which takes the first element from the list, negates it, and cons it onto the result of the recursive function application
- We need to define a function (lambda (x rror) …) that combines x and rror where x is the first element of the list and rror is the result of the recursive function application on the rest of the list.

> negate-list using foldr

The function we need is

```
(lambda (x rror) (cons (- x) rror))
```

We'll start using the terminology non-recursive function to mean a function that doesn't use recursion at all or any recursion is done via an abstract list function such as foldr.

Thus we can give a **non-recursive** version of `negate-list` (that is, `foldr` does all the recursion).

```
(define (negate-list lst)
  (foldr (lambda (x rror) (cons (- x) rror)) empty lst))
```

Because we generalized `negate-list` to `map`, we should be able to use `foldr` to define `map`.

> my-map using foldr

Let's look at the code for `my-map`.

```
(define (my-map f lst)
  (cond [(empty? lst) empty]
        [else (cons (f (first lst))
                    (my-map f (rest lst)))]))
```

empty is the base case

Clearly `empty` is the base value, and the combining function provided to `foldr` is something involving `cons` and `f`.

> my-map using foldr

In particular, the function provided to `foldr` must apply `f` to its first argument, then `cons` the result onto its second argument (the reduced rest of the list).

```
(define (my-map f lst)
  (foldr (lambda (x rror) (cons (f x) rror)) empty lst))
```

We can also implement `my-filter` using `foldr`.

my-filter using foldr

- One approach is to make this code match the structure of foldr so we can identify the values to pass to foldr as the base and the combining function

```
(define (my-filter pred? lst)
  (cond [(empty? lst) empty]
        [(pred? (first lst))
         (cons (first lst) (my-filter pred? (rest lst)))]
        [else (my-filter pred? (rest lst))]))

(define (my-foldr combine base lst)
  (cond [(empty? lst) base]
        [else (combine (first lst)
                       (my-foldr combine base (rest lst)))]))
```

my-filter does not look much like the code for foldr yet

- We start by moving the last two question/answer pairs into an else clause

```
(define (my-filter pred? lst)
  (cond [(empty? lst) empty]
        [else (cond [(pred? (first lst))
                     (cons (first lst) (my-filter pred? (rest lst)))]
                    [else              (my-filter pred? (rest lst))])]))
```

- Next, make the contents of the else clause into a function

```
(define (my-filter pred? lst)
  (local [(define (maybe-cons x filtered)
            (cond [(pred? x) (cons x filtered)]
                  [else filtered]))]
    (cond [(empty? lst) empty]
          [else (maybe-cons (first lst)
                            (my-filter pred? (rest lst)))])))
```

- The last three lines match foldr. We can now use foldr by passing empty to base and the

function maybe-cons to the combining function

- We also see that maybe-cons is a function that is only used once and can therefore be replaced with a lambda expression

```
(define (my-filter pred? lst)
  (foldr (lambda (x filtered)
           (cond [(pred? x) (cons x filtered)]
                 [else filtered]))
         empty lst))
```

Summary: ALFs vs. the list template

- Anything that can be done with the list template can be done using foldr, without explicit recursion (unless it ends the recursion early, like insert)
- Experienced Racket programmers still use the list template, for readability and maintainability

Abstract list functions should be used judiciously, to replace relatively simple uses of recursion.

Foldl

Friday, November 20, 2020 22:27

Generalizing accumulative recursion

Let's look at several past functions that use recursion on a list with one accumulator.

```
;; code from lecture module 12

(define (sum-list lst0)
  (local [(define (sum-list/acc lst sum-so-far)
            (cond [(empty? lst) sum-so-far]
                  [else (sum-list/acc (rest lst)
                                      (+    (first lst) sum-so-far))]))]
    (sum-list/acc lst0 0)))

(check-expect (sum-list '(1 2 3 4)) 10)
```


> Generalizing accumulative recursion

Let's look at several past functions that use recursion on a list with one accumulator.

```
;; code from lecture module 9 rewritten to use local

(define (rev-list lst0)
  (local [(define (rev-list/acc lst lst-so-far)
            (cond [(empty? lst) lst-so-far]
                  [else (rev-list/acc (rest lst)
                                      (cons (first lst) lst-so-far))]))]
    (rev-list/acc lst0 empty)))

(check-expect (rev-list '(1 2 3 4 5)) '(5 4 3 2 1))
```


foldl

- The differences between these two functions are
  - the initial value of the accumulator

- the computation of the new value of the accumulator, given the old value of the accumulator and the first element of the list

> foldl

```
(define (my-foldl combine base lst0)
  (local [(define (foldl/acc lst acc)
            (cond [(empty? lst) acc]
                  [else (foldl/acc (rest lst)
                                   (combine (first lst) acc))]))]
    (foldl/acc lst0 base)))


(define (sum-list lon) (my-foldl + 0 lon))
(define (my-reverse lst) (my-foldl cons empty lst))
```

`foldl` is defined in the Intermediate Student language and above.

> foldl

We noted earlier that intuitively, the effect of the application
`(foldr f b (list x_1 x_2 ... x_n))`
is to compute the value of the expression
`(f x_1 (f x_2 (... (f x_n b) ...)))`

What is the intuitive effect of the following application of `foldl`?

```
(foldl f b (list x_1 ... x_n-1 x_n))
```


Tracing my-foldl

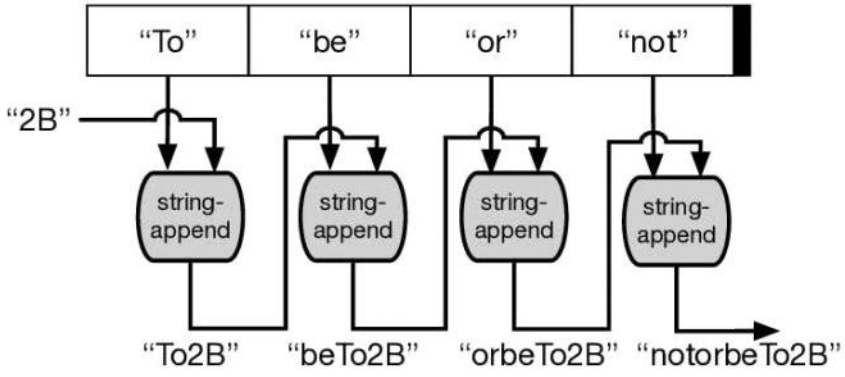
```
(my-foldl f 0 (list 3 6 5))
=> (foldl/acc (list 3 6 5) 0)
=> (foldl/acc (list 6 5) (f 3 0))
=> (foldl/acc (list 5) (f 6 (f 3 0)))
=> (foldl/acc (list ) (f 5 (f 6 (f 3 0))))
=> (f 5 (f 6 (f 3 0)))
```

- Intuitively, the effect of the application (foldl f b (list x_1 ... x_n-1 x_n)) is to compute the value of the expression (f x_n (f x_n-1 (... (f x_1 b))))

> Tracing `foldl`

```
(foldl f b (list x_1 x_2 ... x_n)) (f x_n (f x_n-1 (... (f x_1 b))))

(foldl string-append "2B" '("To" "be" "or" "not")) ⇒ "notorbeTo2B"
```



- foldr and foldl may give the same or different results.

```
(foldr + 0 '(1 2 3 4)) => 10
(foldl + 0 '(1 2 3 4)) => 10
```

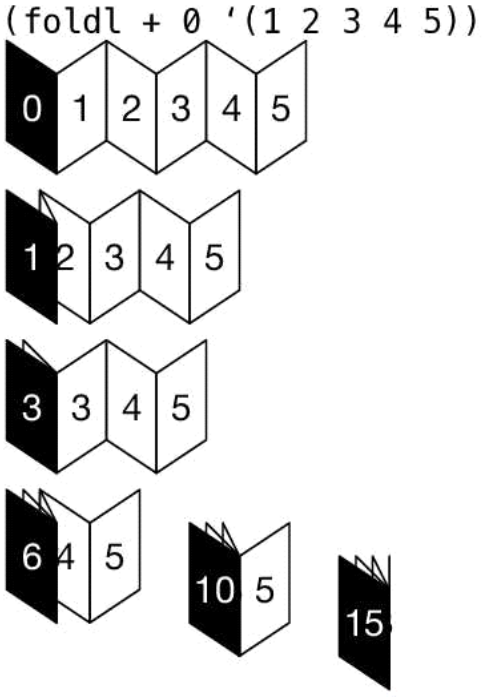
```
(foldr string-append "2B" '("To" "be" "or" "not"))
=> "Tobeornot2B"
(foldl string-append "2B" '("To" "be" "or" "not"))
=> "notorbeTo2B"
```

- In general, if the tcombining operation is commutative, foldr and foldl will generate the same result
- In these examples, + is commutative but string-append is not

foldl

- foldl is short for "fold left"

- The reason for the name is that it can be viewed as "folding" a list using the provided function, starting from the left-hand end of the list



- Contract: (X Y -> Y) Y (listof X) -> Y

Build-list

Friday, November 20, 2020 23:57

Deriving build-list

- Another useful built-in ALF is build-list.
- It consumes a natural number n and a function f, and produces the list
  `(list (f 0) (f 1) ... (f (sub1 n)))`
- Examples:

```
(build-list 4 (lambda (x) x)) ⇒ (list 0 1 2 3)

(build-list 4 (lambda (x) (* 2 x))) ⇒ (list 0 2 4 6)
```

- build-list abstracts the "count up" pattern, and it is easy to write our own version.

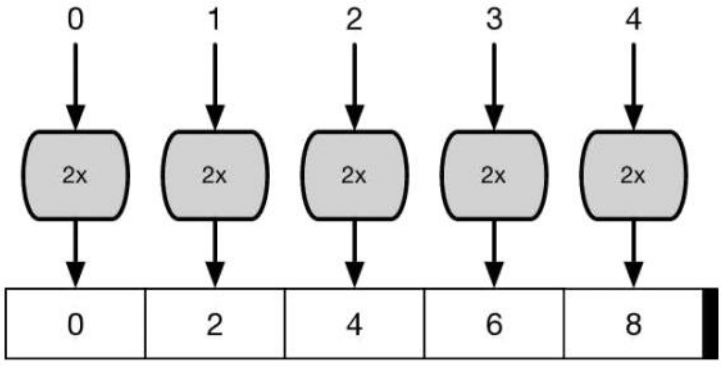
> my-build-list

```
(define (my-build-list n f)
  (local [(define (list-from i)
            (cond [(>= i n) empty]
                  [else (cons (f i) (list-from (add1 i)))]))]
    (list-from 0)))
```

Contract: Nat (Nat -> X) -> (listof X)

> Visualizing `build-list`

```
(build-list 5 (lambda (x) (* 2 x)))
```



> Build-list examples

$\sum_{i=0}^{n-1} f(i)$

```
(define (sum n f)
  (foldr + 0 (build-list n f)))

(sum 4 sqr)
⇒ (foldr + 0 (build-list 4 sqr))
⇒ (foldr + 0 '(0 1 4 9))
⇒ 14
```

foldr and build-list are both built-in, so we do not trace them

We split the summation into two parts, each handled by an ALF

The first part is a list of f applied to each of the numbers in 0, 1, …, n-1, handled by build-list

The second part is summing them up, handled by foldr.

> Simplify `mult-table`

We can now simplify `mult-table` even further.

```
(define (mult-table nr nc)
  (build-list nr
              (lambda (r)
                (build-list nc
                            (lambda (c)
                              (* r c))))))
```

Introduction

Sunday, November 22, 2020 20:13

Generative recursion

- **Simple** and **accumulative recursion** are ways to derive code whose form parallels a data definition
- In **generative recursion**, the recursive cases are generated based on the problem to be solved
- The non-recursive cases also do not follow the data definition
- It is much harder to come up with such solutions to problem. And it often requires deeper analysis and domain-specific knowledge

Example revisited: GCD

```
;; (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm
;; euclid-gcd: Nat Nat → Nat
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

- Why does this work?
- **Correctness**: Math 135 proof
- **Termination**: An application terminates if it can be reduced to a value in finite time

Termination

Sunday, November 22, 2020 21:08

Termination of recursive function

- Why did our functions using simple recursion terminate?
- A **simple recursive function** always makes recursive applications on smaller instances, whose size is bounded below by the **base case** (e.g. the empty list)
- We can thus bound the **depth of recursion**, the number of applications of the function before arriving at a base case
- As a result, the evaluation cannot go on forever

Depth of recursion example

```
(define (sum-list lst)
  (cond [(empty? lst) 0]
        [else (+ (first lst) (sum-list (rest lst)))]))

(sum-list (list 3 6 5 4))                          ;; 1
⇒ (+ 3 (sum-list (list 6 5 4)))                    ;; 2
⇒ (+ 3 (+ 6 (sum-list (list 5 4))))                ;; 3
⇒ (+ 3 (+ 6 (+ 5 (sum-list (list 4)))))            ;; 4
⇒ (+ 3 (+ 6 (+ 5 (+ 4 (sum-list (list ))))))       ;; arrived at base case
⇒ (+ 3 (+ 6 (+ 5 (+ 4 0)))) ⇒ ... ⇒ 18
```

- The depth of recursion of any application of sum-list is equal to the length of the list to which it is applied

Termination of euclid-gcd

- In the case of euclid-gcd, our measure of progress is the size of the second argument
- If the first argument is smaller than the argument, the first recursive application switches them, which makes the second argument smaller
- After that, the second argument always gets smaller in the recursive application, but it is bounded below by 0
- Thus, any application of euclid-gcd has a depth of a recursion bounded by the second argument

Quicksort

Sunday, November 22, 2020 21:09

Hoare's Quicksort

- The **quicksort algorithm** is an example of **divide and conquer**
  - divide a problem into smaller subproblems
  - recursively solve each one
  - combine the solutions to solve the original problem
- Quicksort sorts a list of numbers into non-decreasing order by first choosing a **pivot** element from the list
- The subproblems consist of the elements less than the pivot, and those greater than the pivot

Pivot and subproblems

- The easiest pivot to select from a list lon is (first lon)
- A function which tests whether another item is less than the pivot is (lambda (x) (< x (first lon)))
- The first subproblem is then (filter (lambda (x) (< x (first lon))) lon)
- A similar expression will find the second subproblem (numbers greater than the pivot)

```
> my-quicksort
```

```
;; (my-quicksort lon) sorts lon in non-decreasing order
(check-expect (my-quicksort '(5 3 9)) '(3 5 9))

;; my-quicksort: (listof Num) → (listof Num)
(define (my-quicksort lon)
  (cond [(empty? lon) empty]
        [else (local [(define pivot   (first lon))
                      (define less    (filter (lambda (x) (< x pivot))
                                              (rest lon)))
                      (define greater (filter (lambda (x) (>= x pivot))
                                              (rest lon)))]
                (append (my-quicksort less)
                        (list pivot)
                        (my-quicksort greater)))]))
```


Quicksort termination

- Termination of quicksort follows from the fact that both subproblems have fewer elements than the original list (since neither contains the pivot)
- Thus, the depth of reursion of an application of my-quicksort is bounded above by the number of elements in the argument list
- This would not have been true if we had mistakenly written

  ```
  (filter (lambda (x) (>= x pivot)) lon)
  ```

  instead of the correct

  ```
  (filter (lambda (x) (>= x pivot)) (rest lon))
  ```

Built-in quicksort

- The built-in function quicksort consumes two arguments, a list and a comparison function

```
(quicksort '(1 5 2 4 3) <) ⇒ '(1 2 3 4 5)
(quicksort '(1 5 2 4 3) >) ⇒ '(5 4 3 2 1)

(quicksort '("chili powder" "anise" "basil") string<?)
⇒ (list "anise" "basil" "chili powder")
```

Design recipe

Sunday, November 22, 2020 21:49

Modifying the design recipe

- The design recipe becomes much more vague when we move away from data-directed design
- The purpose statement remains unchanged, but additional documentation is often required to describe how the function works
- Examples are needed to illustrate the workings of the algorithm
- We cannot apply a template since there is no data definition
- For divide and conquer algorithms, there are typically tests for the easy cases that do not require recursion, followed by the formulation and recursive solution of subproblems, and then combination of the solutions

Example

Tuesday, November 24, 2020 14:31

Example: breaking strings into lines

- Computer character sets include not only alphanumeric characters and punctuation, but "control" character as well
- Example in Racket: #\newline
- \n appearing in a string constant is interpreted as a single newline character
- The string "ab\ncd" is is a five-character string with a newline as the third character. It would typically be printed as "ab" on one line and "cd" on the next line

Getting started

- Consider converting a string such as "one\ntwo\nthree" into a list of strings, (list "one" "two" "three"), one for each line
- The solution will start with an application of string->list
- This problem can be solved using simple recursion on the resulting list of characters, but it will be hard
- In this case the generative solution is easier

The generative idea

- Instead of thinking of the list of characters as a list of characters, think of it as a list of lines

o n e \n t w o \n t h r e e

o n e \n t w o \n t h r e e

- A list of lines is either empty or a line followed by a list of lines
- Start with helper functions that divide the list of characters into the first line and the rest of the lines

> Helper: first-line

```
;; (first-line loc) produces longest newline-free prefix of loc
;; Examples:
(check-expect (first-line empty) empty)
(check-expect (first-line '(#\a #\newline)) '(#\a))
(check-expect (first-line (string->list "abc\ndef")) '(#\a #\b #\c))

;; first-line: (listof Char) → (listof Char)
(define (first-line loc)
  (cond [(empty? loc) empty]
        [(char=? (first loc) #\newline) empty]
        [else (cons (first loc) (first-line (rest loc)))]))
```

> Helper: `rest-of-lines`

```
;; (rest-of-lines loc) produces loc with everything up to
;;    and including the first newline removed
;; Examples:
(check-expect (rest-of-lines empty) empty)
(check-expect (rest-of-lines '(#\a #\newline)) empty)
(check-expect (rest-of-lines '(#\a #\newline #\b)) '(#\b))

;; rest-of-lines: (listof Char) → (listof Char)
(define (rest-of-lines loc)
  (cond [(empty? loc) empty]
        [(char=? (first loc) #\newline) (rest loc)]
        [else (rest-of-lines (rest loc))]))
```


> List of lines template

We can create a "list of lines template" using these helpers.

```
(define (loc->lol loc)
  (local [(define fline (first-line loc))
          (define rlines (rest-of-lines loc))]
    (cond [(empty? loc) ...]
          [else (... fline ... (loc->lol rlines) ...)])))
```

Using local just to give names to the first line and the rest of the lines
The core of the template is the cond expression, which looks a lot like the listof-X template

> `loc->lol`

```
;; loc->lol: (listof Char) → (listof Str)
(define (loc->lol loc)
  (local [(define fline (first-line loc))
          (define rlines (rest-of-lines loc))]
    (cond [(empty? loc) empty]
```

base case: when loc is empty, produce an empty list of lines

```
(local [(define fline (first-line loc))
        (define rlines (rest-of-lines loc))]
  (cond [(empty? loc) empty]
        [else (cons (list->string fline)
                    (loc->lol rlines))]))

;; string->lines: Str -> (listof Str)
(define (string->lines s)  (loc->lol (string->list s)))
```

base case: when loc is empty, produce an empty list of lines

fline is a list of characters, so convert it to string with list->string. Then cons it on to (loc->rlines) (loc->lol rlines) produces a list of strings, each one corresponding to a line

`loc->lol` looks a lot like the template for functions consuming a `(listof X)`. That was simple recursion. Is this also simple recursion?

No, this is **not** simple recursion. Why not?

Because the argument rlines to loc->lol is many steps closer to the base case

Generative recursion

- Why is this generative recursion?
- loc->lol can be rewritten as

```
(define (loc->lol loc)
    (cond [(empty? loc) empty]
          [else (cons (list->string (first-line loc))
                      (loc->lol (rest-of-lines loc)))]))
```

- The recursive call to loc->lol is not using the data definition for a list of characters
- It often gets many steps closer to the base case in one recursive application
- It is using a data definition of a "list of lines", but there is a higher-level abstraction that we imposed on top of the (listof Char), the actual argument
- The key part of the generative recursion pattern is that the argument to loc->lol is being generated by rest-of-lines
- When we use generative recursion, we need to be careful about termination
- Why does string-> lines always terminate?
- Each recursive call is applied to (rest-of-lines loc) where loc is non-empty, but rest-of-lines produces either empty (which leads directly to termination in loc->lol) or a list of characters that is at least one character shorter
- Therefore, the length of the argument to loc->lol is always decreasing until it becomes empty and the function terminates

> Using simple recursion

```
;; list->lines: (listof Char) -> (listof (listof Char))
(define (list->lines loc)
  (cond
    [(empty? loc) (list empty)]
    [(and (empty? (rest loc)) (char=? #\newline (first loc)))
     (list empty)]
    [else
     (local [(define r (list->lines (rest loc)))]
       (cond
         [(char=? #\newline (first loc)) (cons empty r)]
         [else (cons (cons (first loc) (first r)) (rest r))]))]))

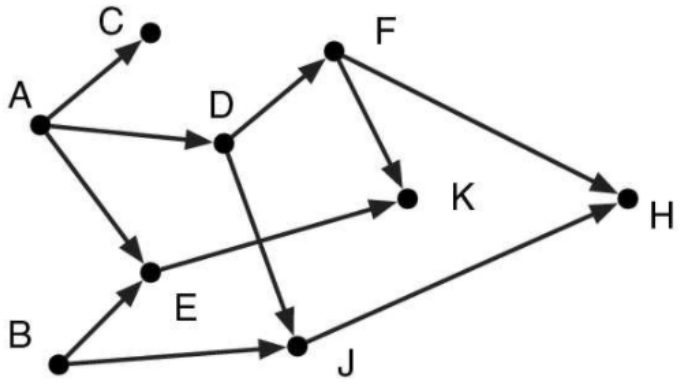

(define (string->lines str)
  (map list->string (list->lines (string->list str))))
```

Intro

Thursday, November 26, 2020 22:43

Directed graphs

- A **directed graph** consists of a collection of **nodes** (also called **vertices**) together with a collection of **edges**
- An edge is an ordered pair of nodes, which we can represent by an arrow form one node to another
- In a **directed graph**, each edge has a direction (indicated by the head of each arrow)
- In an **undirected graph**, edges do not have directions
- Evolution trees and expression trees were both directed graphs of a special type where an edge represented a parent-child relationship



Graph terminology

- Given an edge (v,w), we say that w is an **out-neighbour** of v, and v is an **in-neighbour** of w
- A sequence of nodes $v_1, v_2, \ldots, v_k$ is a **path** or **route** of length k-1 if $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$ are all edges
- If $v_1 = v_k$, this is called a **cycle**
- Directed graphs without cycles are called **DAG**s (**directed acyclic graphs**)

Representation

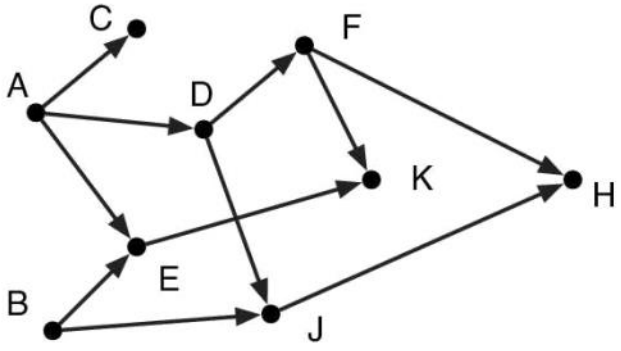
Thursday, November 26, 2020 23:00

Representing graphs

- We can represent a node by a symbol (its name), and associate with each node a list of its out-neighbours
- This is called the **adjacency list** representation
- More specifically, a graph is a list of pairs, each pair consisting of a symbol (the node's name) and a list of symbols (the names of the node's out-neighbours)
- This is very similar to a parent node with a list of children

> Our example as data

```
(define g
  '((A (C D E))
    (B (E J))
    (C ())
    (D (F J))
    (E (K))
    (F (K H))
    (H ())
    (J (H))
    (K ())
  )
```



> Data definitions

To make our contracts more descriptive, we will define a `Node` and a `Graph` as follows:

```
;; A Node is a Sym

;; A Graph is one of:
;; * empty
;; * (cons (list v (list w_1 ... w_n)) g)
;;   where g is a Graph
;;         v, w_1, ... w_n are Nodes
;;         v is the in-neighbour to w_1 ... w_n in the Graph
;;         v does not appear as an in-neighbour in g
```


- A Graph is a (listof X) where X is a two element list (a pair)
- Each pair contains a node name (a symbol), and a list of out-neighbours (a list of symbols)
- The order of the pairs does not matter. The order of the list of symbols does not matter.
- The order of the node and its associated out-neighbours in a pair does matter
- We often put the list of nodes in alphabetical order
- Make sure that each node, v, is only listed once in the list of nodes

> The template for graphs

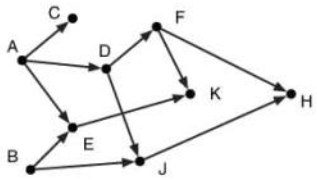
```
;; graph-template: Graph → Any
(define (graph-template g)
  (cond
    [(empty? g) ...]
    [(cons? g)
     (... (first (first g))              ; first node in graph list
      ... (listof-node-template
                (second (first g)))      ; list of adjacent nodes
      ... (graph-template (rest g)) ...)]))
```

> neighbours

We can use the graph template to write a function that produces the out-neighbours of a node. We'll need this function in just a moment.

```
;; (neighbours v g) produces list of neighbours of v in g
;; Examples:
(check-expect (neighbours 'D g) '(F J))
(check-error (neighbours 'Z g) "Node not found")

;; neighbours: Node Graph → (listof Node)
;;   requires:  v is a node in g
(define (neighbours v g)
  (cond
    [(empty? g) (error "Node not found")]
    [(symbol=? v (first (first g))) (second (first g))]
    [else (neighbours v (rest g))]))
```



The code assumes that the provided node is in the graph

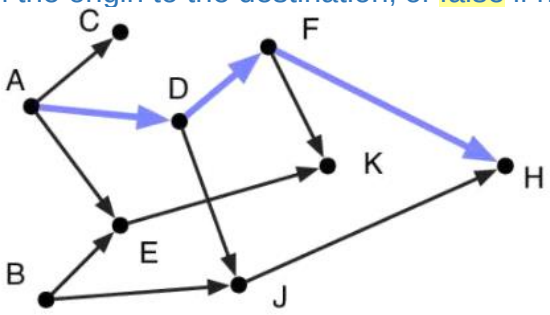
It produces an error if the requirement fails

Paths v1

Thursday, November 26, 2020 23:37

Finding paths

- A path in a graph can be represented by an ordered list of the nodes on the path
- We wish to design a function find-path that consumes a graph plus origin and destination nodes, and produces a path from the origin to the destination, or false if no such path exists




```
(find-path 'A 'H g) ⇒ '(A D F H) or '(A D J H)

(find-path 'D 'H g) ⇒ '(D F H) or '(D J H)

(find-path 'C 'H g) ⇒ false

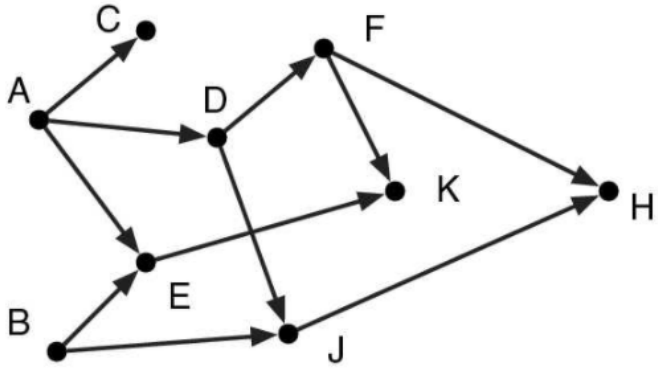
(find-path 'A 'A g) ⇒ '(A)
```

Cases for find-path

- Simple recursion does not work for find-path. We must use generative recursion
- If the origin equals the destination, the path consists of just this node
- Otherwise, if there is a path, the second node on that path must be an out-neighbour of the origin node
- Each out-neighbour defines a subproblem (finding a path from it to the destination)

Building a path from a solved sub-problem

- In our example, any path from A to H must pass through C, D or E
- If we knew a path form C to H, from D to H, or from E to H, we could create one from A to H

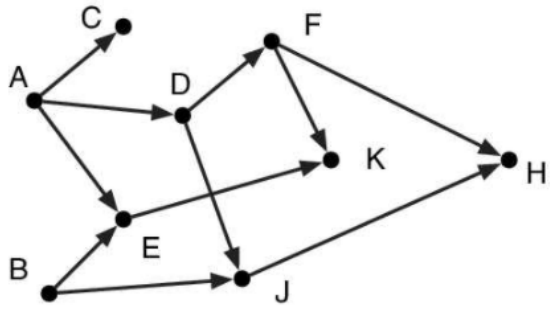


Backtracking algorithms

- Backtracking algorithms try to find a path from an origin to a destination
- If the initial attempt does not work, such an algorithm "backtracks" and tries another choice
- Eventually, either a path is found, or all possibilities are exhausted, meaning there is no path
- Recall: search-bt-path-v1 searches for a path in the left side of a binary tree. If it there is no path, it "backtracks" and searches the right side of the binary tree

Backtracking in this example

- In this example, we can see the "backtracking" since the search for a path from A to H can be seen as moving forward in the graph looking for H
- If this search fails (for example, at C), then the algorithm "backs up" to the previous node and tries the next neighbour
- If we find a path from D to H, we can just cons A to the beginning of this path



Exploring the list of out-neighbours

- We need to apply find-path on each of the out-neighbours of a given node
- The neighbours function gives us a list of all the out-neighbours associated with that node
- This suggests writing find-path/list which consumes a list of nodes and will apply find-path to each one until it either finds a path to the destination or exhausts the list

Mutual recursion

- This is the same recursive patter that we saw in the process of expression trees and evolutionary trees
- For expression trees, we had two mutually recursive functions, eval and apply
- Here, we have two mutually recursive functions, find-path and find-path/list

> find-path

```
;; (find-path orig dest g) finds path from orig to dest in g if it exists
;; find-path: Node Node Graph → (anyof (listof Node) false)
(define (find-path orig dest g)
  (cond [(symbol=? orig dest) (list dest)]
        [else (local [(define nbrs (neighbours orig g))
                      (define ?path (find-path/list nbrs dest g))]
                (cond [(false? ?path) false]
                      [else (cons orig ?path)]))]))
```

We're using `?path` to mean it might hold a path or it might not.

> find-path/list

```
;; (find-path/list nbrs dest g) produces a path from
;;     an element of nbrs to dest in g, if one exists
;; find-path/list: (listof Node) Node Graph → (anyof (listof Node) false)
(define (find-path/list nbrs dest g)
  (cond [(empty? nbrs)   false]
        [else (local [(define ?path (find-path (first nbrs) dest g))]
                (cond [(false? ?path)
                       (find-path/list (rest nbrs) dest g)]
                      [else ?path]))]))
```

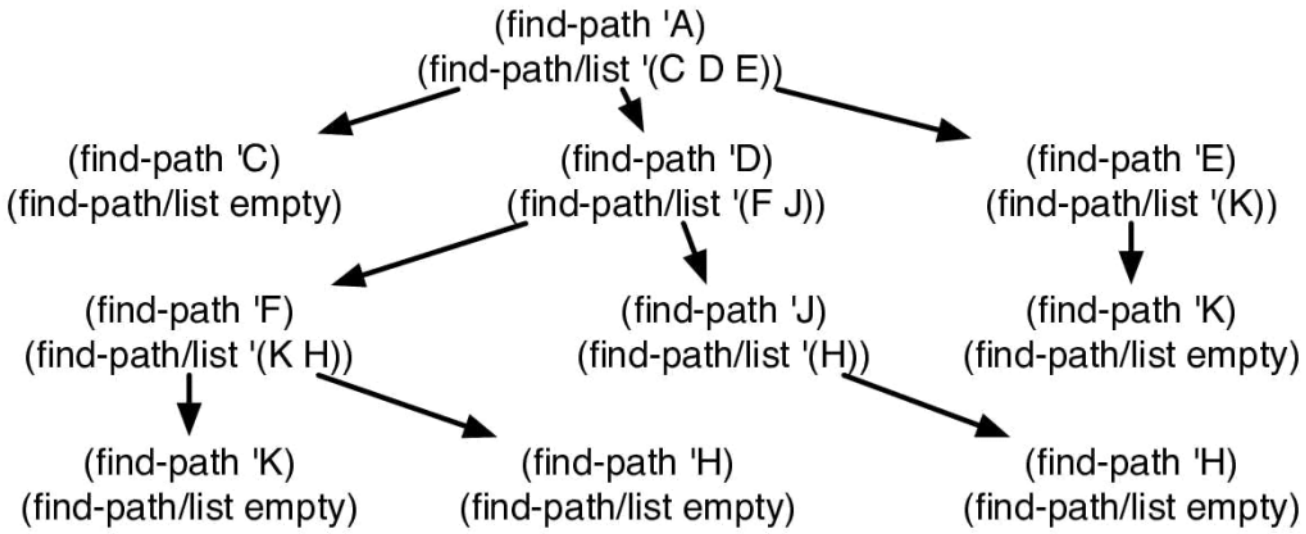

> Tracing (find-path 'A 'B g) (1/2)

If we wish to trace `find-path`, trying to do a linear trace would be very long, both in terms of steps and the size of each step. Our traces also are listed as a linear sequence of steps, but the computation in `find-path` is better visualized as a tree.

We will use an alternate visualization of the potential computation (which could be shortened if a path is found).

The next slide contains the trace tree. We have omitted the arguments `dest` and `g` which never change.

» Tracing `(find-path 'A 'B g)` (2/2)



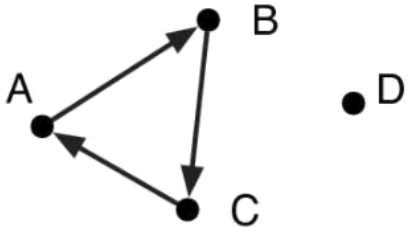
Termination of find-path (no cycles)

In a directed acyclic graph, any path with a given origin will recurse on its (finite number) of neighbours by way of find-path/list

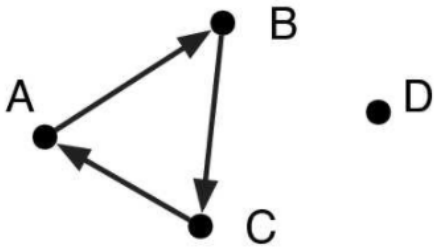
- The origin will never appear in this call or any subsequent calls to find-path. If it did, we would have a cycle in our DAG
- Thus, the origin will never be explored in any later call, and thus the subproblem is smaller
- Eventually, we will reach a subproblem of size 0 (when all reachable nodes are treated as the origin)
- Thus find-path always terminates for directed acyclic graphs

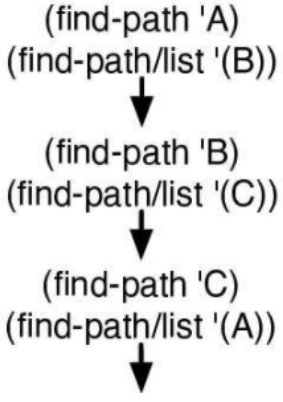
Non-termination of find-path (cycles)

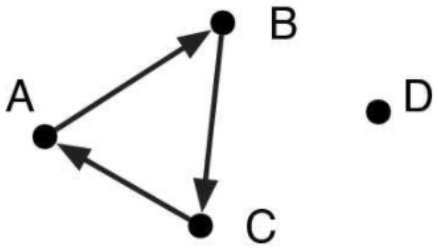
- It is possible that find-path may not terminate if there is a cycle in the graph

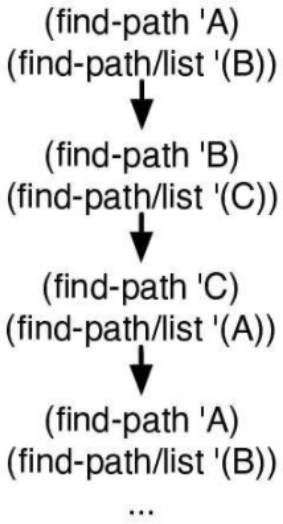



```
'((A (B))
  (B (C))
  (C (A))
  (D ()))
```






A
B
C
D


(find-path 'A)
(find-path/list '(B))
(find-path 'B)
(find-path/list '(C))
(find-path 'C)
(find-path/list '(A))
(find-path 'A)
(find-path/list '(B))
...

Paths v2

Monday, November 30, 2020 17:32

Paths v2: Handling cycles

- We can use accumulative recursion to solve the problem of find-path possibly not terminating if there are cycles in the graph
- To make backtracking work in the presence of cycles, we need a way of remembering what nodes have been visited
- Our accumulator will be a list of visited nodes
- The simplest way to do this is to add a check in find-path/list

> find-path/list

```
;; find-path/list: (listof Node) Node Graph (listof Node) →
;;                 (anyof (listof Node) false)
(define (find-path/list nbrs dest g visited)
  (cond [(empty? nbrs) false]
        [(member?  (first nbrs) visited)
         (find-path/list (rest nbrs) dest g visited)]
        [else (local [(define path (find-path/acc (first nbrs)
                                                  dest g visited))]
                (cond [(false? path)
                       (find-path/list (rest nbrs) dest g visited)]
                      [else path]))]))
```

The new parameter accumulates the nodes that have been visited so far

If find-path/list comes to a neighoburing node that has already been visited, we skip it and go on to the next neighbour on the list

find-path/list's accumulator

- The code for find-path/list does not add anything to the accumulator, but it uses the accumulator
- Adding to the accumulator is done in find-path/acc which applies find-path/list to the list of neighbours of some origin node
- That origin node must be added to the accumulator passed as an argument to find-path/list

> find-path/acc

```
;; find-path/acc: Node Node Graph (listof Node) →
;;                (anyof (listof Node) false)
(define (find-path/acc orig dest g visited)
  (cond [(symbol=? orig dest) (list dest)]
        [else (local [(define nbrs (neighbours orig g))
                      (define path (find-path/list nbrs dest g
                                                   (cons orig visited)))]
                (cond [(false? path) false]
                      [else (cons orig path)]))]))
```

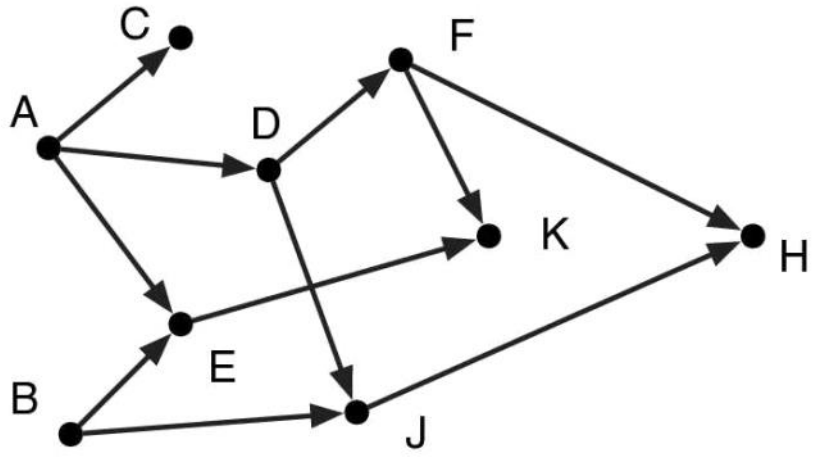
Added the visited parameter to accumulate the nodes visited so far

Updating the accumulator by adding the current origin and passing it to find-path/list

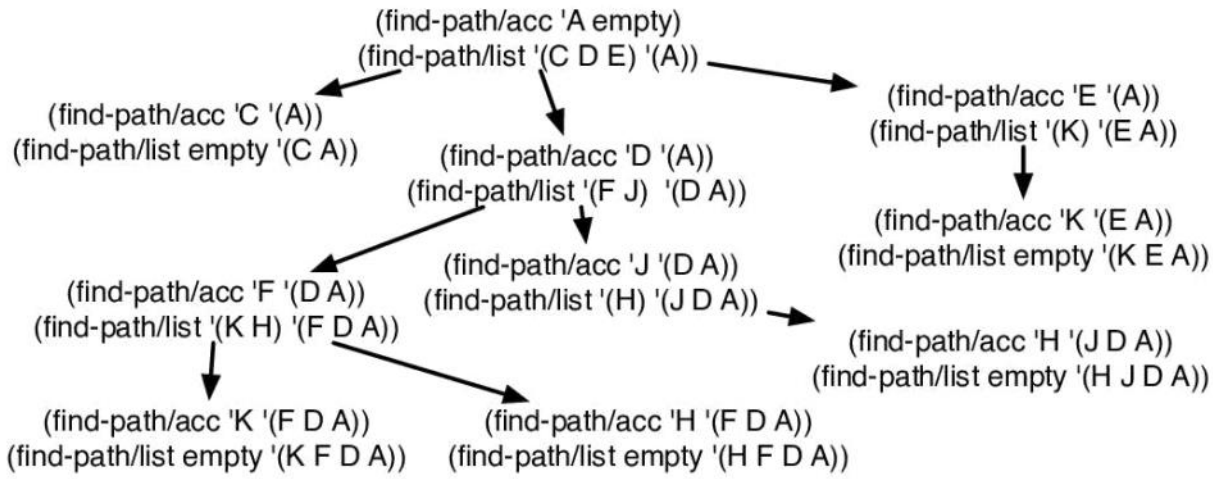
This is a generative recursion that happens to also use an accumulator

```
(define (find-path orig dest g)        ;; new wrapper function
  (find-path/acc orig dest g '()))
```


> Tracing our examples (1/4)



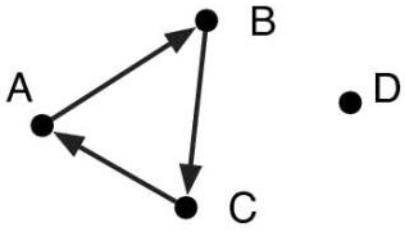
> Tracing our examples (2/4)



Note that the value of the accumulator in `find-path/list` is always the reverse of the path from A to the current origin (first argument).

- This example has no cycles, so the trace convinces us that we have not broken the function on acyclic graphs
- It also works on graphs with cycles
- The accumulator ensures that the depth of recursion is no greater than the number of nodes in the graph, so find-path terminates

> Tracing our examples (4/4)



(find-path/acc 'A empty)
(find-path-list '(B) '(A))

↓

(find-path/acc 'B '(A))
(find-path-list '(C) '(B A))

↓

(find-path/acc 'C '(B A))
(find-path-list '(A) '(C B A))

no further calls to find-path/acc

Cycle is solved, but…

- Backtracking now works on graphs with cycles, but it can be inefficient, even if the graph has no cycles
- If there is no path from the origin to the destination, then find-path will explore every path from the origin, and there

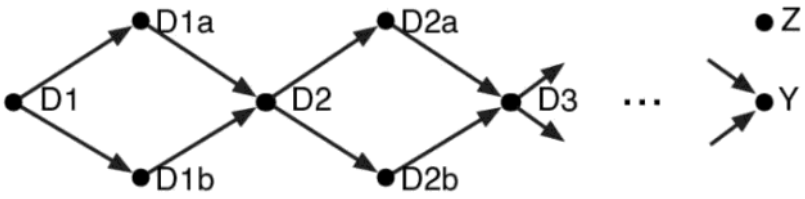
could be an exponential number of them

Paths v3

Tuesday, December 1, 2020 15:03

Paths v3: Efficiency

- If there is no path from the origin to the destination, then find-path will explore every path from the origin, and there could be an exponentail number of them



- If there are d diamonds, then tehre are 3d+2 nodes in the graph, but 2^d paths from D1 to Y, all of which will be explored

Understanding the problem (1/2)

- Applying find-path/acc to origin D1 results in find-lath/list being applied to '(D1a D1b), and then find-path/acc being applied to origin D1a
- There is no path from D1a to Z, so this will eventually produce false, but in the process, it will visit all the other nodes of the graph except D1b and Z
- find-path/lsit will then apply find-path/acc to D1b, which will visit all the same nodes again
- We added an acucmulator to keep track of all the nodes we have visted and not revisit any tahat are on the list. However, the accumulator is forgetful when find-path backtracks

Understanding the problem (2/2)

- When find-path/list is applied to the list of nodes nbrs, it first applies find-path/acc to (first nbrs) and then, if that fails, it applies itself to (rest nbrs)
- To avoid revisiting nodes, the failed computation should pass the list of nodes it has seen on to the next computation
- It will do this by returning the list of visited nodes instead of false when it fails to find a path
- However, we must be able to distinguish this list from a successfully found path, which is also a list of nodes

Remembering what the list of nodes represents

- We will encapsulate each kind of list in its own structure
- We can then easily use the structure predicates to check whether the list of nodes represents a path (success) or visited nodes (failure)

```
(define-struct success (path))
;; A Success is a (make-success (listof Node))

(define-struct failure (visited))
;; A Failure is a (make-failure (listof Node))

;; A Result is (anyof Success Failure)
```

> find-path/list

```
;; find-path/list: (listof Node) Node Graph (listof Node) → Result
(define (find-path/list nbrs dest g visited)
  (cond [(empty? nbrs) (make-failure visited)]
        [(member? (first nbrs) visited)
         (find-path/list (rest nbrs) dest g visited)]
        [else (local [(define result (find-path/acc (first nbrs)
                                                    dest g visited))]
                (cond [(failure? result)
                       (find-path/list (rest nbrs) dest g
                                       (failure-visited result))]
                      [(success?  result) result]))]))
```

?path is renamed result for clarity.

> find-path/acc

```
;; find-path/acc: Node Node Graph (listof Node) → Result
(define (find-path/acc orig dest g visited)
  (cond [(symbol=? orig dest) (make-success (list dest))]
        [else (local [(define nbrs (neighbours orig g))
                      (define result (find-path/list nbrs dest g
                                                     (cons orig visited)))]
                (cond [(failure?  result) result]
                      [(success?  result)
                          (make-success (cons orig
                                              (success-path result)))]))]))
```

?path is renamed result for clarity.

> find-path

```
;; find-path: Node Node Graph → (anyof (listof Node) false)
(define (find-path orig dest g)
  (local [(define result (find-path/acc orig dest g empty))]
    (cond [(success?  result) (success-path result)]
          [(failure?  result) false])))
```