(Provisional) Lecture 17: Trees, OCaml

10:00 AM, Oct 16, 2017

Contents

Objectives

By the end of this lecture, you will be able to:

- understand the applications and limitations of \texttt{lambda}
- understand the purpose, terminology and Racket representation of the \texttt{Tree} data structure
- analyze procedures involving trees
- learn about the \texttt{OCaml} language, and its differences from Racket
- learn about atomic types, \texttt{if} expressions, compound types and bindings in \texttt{OCaml}

1 Introduction

We finish our discussion of lambda expressions, describe the final bit of CS17 Racket, \texttt{let}, and introduce a combinatorial problem.

2 Review

So far you’ve learned a few things in CS17: how to write programs in (a subset of) Racket, how to use the design recipe to help you write recursive programs, how to \textit{analyze} recursive programs. And you’ve learned to use recursive diagrams to get things done.

Along the way, there have been some interesting ideas snuck in under the radar. One is that recursion is powerful. Another is that helper procedures can help you clarify your programs or to save computation time. Writing good tests can save you development time, too.

Perhaps the sneakiest of the new ideas is that making a problem more general (which we sometimes call “making it harder”) may actually make it \textit{easier} to solve recursively.

When you think about recursion, and recursive diagrams, there’s a point when you have a recursive result and need to think about how to get an overall result from it. If the recursive result isn’t quite what you need, getting the overall result can be a pain in the neck. (Think about our slow reverse procedure, or the attempts we made at fancy-censoring a string list!) But if your procedure has an extra argument (“What symbol should fancy-censoring start with?”), you can often use this extra argument to customize the recursive result to be one that’s more useful to you in generating the overall result. So while you’re solving a harder problem, you’re also working with more powerful tools (the ability to solve the harder problem on subproblems).
Another big idea so far is the having simple representations of structure (like lists) makes it easy to write programs that work with these representations: we can write a main “cond” that has only two or four cases, etc.

3 Limitations and applications of lambda

We’ve now learned about lambda, and it lets us produce procedures “on the fly,” which is great. But throughout the course, we’ve been writing recursive procedures. Can we write these with lambda? No! Because we’d need, inside the procedure, to invoke the current procedure, which is anonymous, i.e., does not have a name! Could let help us? If we write

```
(let
  ((len (lambda (alod)
    (cond
      [(empty? alod) 0]
      [(cons? alod) (+ 1 (len (rest alod)))])))))
(len (list 1 2 3)))
```

we get an error that len is not defined, alas: the rules of evaluation show that attempting to use len inside the lambda doesn’t work, because the local environment associated to our procedure doesn’t actually end up containing the name len.

Racket solves this by defining one more bit of syntax: letrec, which allows us to refer to the procedure we are defining within that procedure. This lets us write

```
(letrec
  ((len (lambda (alod)
    (cond
      [(empty? alod) 0]
      [(cons? alod) (+ 1 (len (rest alod)))])))))
(len (list 1 2 3)))
```

which produces the value 3 as you might expect. I’m not going to go into any of the details of this, because I want to leave racket as soon as possible.

But I have one more application of lambda that I want to discuss, because folks having trouble with this week’s homework have made it clear that there’s still some lambda-ness that they haven’t understood completely.

3.1 map from map2

Suppose I give you the procedure map2, and tell you that I want you, using it, to write map. Well, map2 consumes two lists and a two-argument procedure, but you want to write something that consumes one list and a single-argument procedure. Generating two lists isn’t too tough: you can just use your single list twice. So you might try writing

```
(define (my-map p alod)
  (map2 p alod alod))
```
which doesn’t work: the map2 procedure takes the first item of each list, and applies the procedure to those two as arguments. But \( p \) is a single-arg procedure. What to do? We have a single-arg procedure, and we need a two-arg procedure.

Well, the first thing to realize is that we need to somehow produce a procedure, and there are only two ways to do that: through “define” and “lambda”. And since you’re not allowed to have definitions inside expressions, we’re really stuck with lambda. Once we realize that, the lambda-expression is easy to write:

\[
\text{(define (my-map p alod)}
\text{  (map2 (lambda (x y) (p x)) alod alod))}
\]

...and we’re done!

## 4 Trees

Trees come up in Computer Science all the time. They are a generalization of lists that help us model various data:

- Parent-child relations: like family trees or file/folder structures

![Tree Diagram](image)

- Expressions: here, the main operation is multiplication of two terms. The first is the addition of 3 and 4, and the second is the difference of 2 and 1/6

![Expression Diagram](image)
• Hierarchy: (President is the boss of provost and Vice Pres for research; provost is the boss of ...)

• Classification: life forms divides into Kingdoms, each divided into Phylums, each divided into Orders, etc.

• Recursive procedure invocation: Nodes contain expressions, and each recursive evaluation is placed in a child node.

4.1 Tree Terminology

We use various terms in defining a tree and describing relationships between nodes:

• Root: top of the tree
• Node: a tree with children
• Leaf: a tree without children
• Parent: the node connected above you
• Child: the node connected below you
• Sibling: leaves/nodes having the have same parent
• Ancestor: anyone between you and the root, including the root
• Descendant: a leaf/node is your descendant if you are its ancestor
• Depth: for a tree consisting of a single leaf, Depth = 0. For any other, Depth = 1 + max(depth(children)). For this example, Depth = 2.

• Binary tree: each internal node has 2 children

We can have two different models of trees, one where the leaves contain data, and another where they don’t.

Leaves that contain data:
Leaves that do not contain data:

In CS17, we will be using the second model as it is a nice analogy to the \texttt{cons} / \texttt{empty} structure of lists, and makes recursion more natural!

### 4.2 Representing trees in Racket, version 1

In our representation of trees, we will briefly break a CS17 rule and use mixed data in lists. We will be modeling binary trees, which have exactly two subtrees.

We will represent a tree with a node containing 7, and left and right subtrees both being leaves using Racket lists, as

\[(\text{list } 7 \text{ empty empty})\]
More generally, a leaf is represented by \texttt{empty}.
Nodes are written as

$$(\text{list node-val left-subtree right-subtree})$$

To abstract this representation, we write a data definition:

$$\begin{align*}
\text{;; an 'a tree is either} \\
\text{;; a leaf or} \\
\text{;; a node, consisting of} \\
\text{;; a value, of type 'a, and} \\
\text{;; two 'subtrees', each of type 'a tree, the "left" and the "right"} \\
\text{;; nothing else is an 'a tree.} \\
\text{;; We represent a leaf by the empty list, and} \\
\text{;; the node with value v, left-subtree t1, and right-subtree t2, by} \\
\text{;; (list v t1 t2)}
\end{align*}$$

We now need procedures to extract data from our tree. The procedures \texttt{left-subtree} and \texttt{right-subtree} will return the left and right subtrees of the input node:

$$\begin{align*}
\text{;; left-subtree: 'a node -> 'a tree} \\
\text{(define (left-subtree n) (first (rest n)))} \\
\text{;; right-subtree: 'a node -> 'a tree} \\
\text{(define (right-subtree n) (first (rest (rest n))))}
\end{align*}$$

To get the value of the node, we write \texttt{node-val}:

$$\begin{align*}
\text{;; node-val: 'a node -> 'a} \\
\text{(define (node-val n) (first n))}
\end{align*}$$

Now we need procedures to construct trees, for which we have \texttt{make-node} which makes a node given the value of the node and its left and right subtrees. \texttt{make-leaf} will just return empty:

$$\begin{align*}
\text{;; make-node: 'a * 'a tree* 'a tree -> 'a node} \\
\text{(define (make-node val left right) (list val left right))} \\
\text{;; make-leaf: . -> 'a tree} \\
\text{(define (make-leaf ) empty)}
\end{align*}$$

We also need procedures to check whether a tree is a node or a leaf, similar to checking \texttt{empty?} and \texttt{cons?} for lists:

$$\begin{align*}
\text{;; leaf?: 'a tree -> bool} \\
\text{(define leaf? empty?)} \\
\text{;; node?: 'a tree -> bool} \\
\text{(define node? cons?)}
\end{align*}$$
4.3 Depth of a tree

We define the depth of a tree to be the length of the longest path from the root to a leaf. This is the same as the number of vertical “levels” you can see in the tree. How would we go about writing a procedure to find this depth? First, let us write the design recipe!

;;; depth: 'a tree -> int
;;; input: t, an 'a tree
;;; output: the depth of t, where the depth of a leaf is defined to be 0.

Keep in mind that the structure of your program should follow the structure of your data. We have two possibilities for something being a tree - leaf and node. This gives:

;;; (define (depth t)
; (cond
; [ (leaf? t) ... ]
; [ (node? t) ... (node-val t) (left-subtree t) (right-subtree t) ... ]))

This is similar to finding the length of a list, except that there are two substructures. In the empty case, we return 0 because we have reached the end of the list. Similarly, the depth of a leaf is 0, since once we’ve reached a leaf we have reached the end of that branch in the tree. In the recursive case, we want to do something with the left and right subtrees.

;;; (define (depth t)
; (cond
; [ (leaf? t) 0]
; [ (node? t) ... (node-val t) (left-subtree t) (right-subtree t) ... ]))

We can find the depth of the left and right subtrees by recursively calling depth on them. How do we find the depth of our tree, given the depth of the left and right subtrees? We take the longer one of the two, and then add 1! (so that we’re counting our current code):

;;; (define (depth t)
; (cond
; [ (leaf? t) 0]
; [ (node? t) (+ 1 (max (depth (left-subtree t)) (depth (right-subtree t))))]))

4.4 tcontains17?

Using the depth procedure as a guide, let us try writing tcontains17?, a procedure that searches an int tree to say whether it contains the number 17. The design recipe will look like:

;;; tcontains17?: int tree -> bool
;;; input: t, an int tree
;;; output: true if the value at some node of t is 17; false otherwise
Let us think about when exactly a tree would contain 17:

If it is a leaf, it does not contain 17.
If it is a node, there are 3 possibilities:

- Either the value of the node is 17,
- or the left subtree contains 17,
- or the right subtree contains 17.

\[
\text{(define } (\text{tcontains17? } t) \\
(\text{cond} \\
[ (\text{leaf? } t) \text{ false} ] \\
[ (\text{node? } t) (\text{or} (= 17 (\text{node-val } t)) \\
(\text{tcontains17?} (\text{left-subtree } t)) \\
(\text{tcontains17?} (\text{right-subtree } t))))])
\]

Can we do this in a way that doesn’t abuse our rule about homogeneous lists? We can, using racket ”structs” (i.e., structures). We won’t actually use these in CS17, because OCaml provides something much nicer for doing this (i.e., doing this is more natural in OCaml). A general lesson is that different languages suit different tasks better.

4.5 \text{tcontains17? Runtime}

Let us now analyze our \text{tcontains17?} procedure!

Let \( T(d) \) be the number of elementary operations involved in evaluating \( \text{tcontains17? } t \) on any tree of depth \( d \).

The recursive call here is made twice, on the left and right subtree. We get the recurrence relation:

\[
T(0) = A \\
T(d) \leq B + 2T(d - 1)
\]

Using plug and chug to get a closed from conjecture, we get

\( T(d) \leq C \times 2^d \), for a constant \( C \) and \( d \geq 0 \)

Note that this requires a formal proof.

So we find the Big-O class to be \( T \in O(2^d) \).

We could have done this analysis for a tree containing \( n \) nodes instead of a tree of depth \( d \). Now, \( 2^d \), the Big-O class that we found, is approximately equal to the number of nodes in a binary tree. So we are going though each node in our tree approximately once in \text{tcontains17?}. However, if we do our analysis in terms of \( n \), the number of nodes, we get an exponential runtime, as opposed to linear, which is a very high overestimate. It seems like we would need a better method to accurately analyze our procedure, which we will see later.
5 Moving on: OCaml

In the spirit of learning concepts that transcend language, starting today, we will be leaving the world of Racket behind and moving on to a different language, called OCaml. OCaml is a direct descendant of a language called ML, for Meta Language, that’s been around since the late 1970s.

Because some programming languages are more appropriate for some tasks, chances are that you will learn many programming languages. Our plan is to get you started on that path right now. Also, we teach you several languages because we want you to learn first hand that programming, and more generally computer science, transcends syntax.

Is OCaml better than Racket? No. Neither is any more or less powerful than the other. Still, the languages do have their strengths and weaknesses. Today, we will introduce one of the primary strengths of OCaml: its type system. (The other, matching, is for next class!)

Firstly, OCaml is very much like Racket in some ways:

- Both are functional programming languages, i.e., functions are first-class entities in the language
- They make heavy use of lists
- They are very suited to recursion
- Many procedures begin with a cond-like structure

Next, let’s talk about some differences between the two languages:

- OCaml has an explicit notion of “types” and is very rigid about them. In Racket, we could ask - “is this number an integer?” In OCaml, we will explicitly say this argument to this procedure must be an integer. Types are a part of the code as opposed to just being in the comments.
- Arithmetic is ”infix”, i.e., we write 3 + 2 instead of (+ 3 2)
- In addition to lists, OCaml has tuples, which are collections of data which can be of different types. They have a fixed size, however.
- OCaml has “cond on steroids,” called pattern matching. It essentially extracts all parts (like first and rest) and names them.
- OCaml has struct-like types built in, in a very simple form.
- OCaml has variant types (like “a season is Fall or Winter or . . .”) built in.
- OCaml has recursive types (like lists) both built in and easy to create.
- lambda is called “fun”
- All functions take one argument, sort of. (we’ll learn more about this later!)

\(^1\)Exams and projects aside, of course! You didn’t think we’d just drop it all, did you?
\(^2\)Technically speaking, both languages are Turing complete, meaning they can both encode any “mechanical process” (see Lecture 01).

9
Now, let’s answer some common questions:

**Question:** What does the “O” in OCaml stand for?

**Answer:** “O” stands for objective. We will not use the object-oriented aspect of OCaml, however. Instead, you will learn object-oriented programming using Java and Scala in CS 18.

**Question:** What does the “Ca” in OCaml stand for?

**Answer:** According to Wikipedia, “CaML” stands for Categorial Abstract Machine Language.

**Question:** Will our Racket design recipe also be useful for OCaml?

**Answer:** Absolutely! The design recipe is not language-specific. It is something that should guide your programming always, in any language.

### 6 Do Types Matter?

Yes! As mentioned before, OCaml has a very rigid notion of types. You must indicate types when you’re coding your programs. Before running the program, OCaml does a type check to attempt to assign a type to every expression in your program. If any of the types are inconsistent or don’t make sense, it will give you an error. As a result, you cannot make polymorphic lists like (list 3 4 true) in OCaml, because its type-system allows only monotype lists. (Racket allowed mixed lists, but CS17 didn’t, in order to prepare you for OCaml!) Some error checks are also done for you, so you never need to write code that checks whether an argument to your program is a string for example, because OCaml will give you an error if it isn’t!

### 7 Atomic Types

Like Racket, OCaml has both atomic types and compound types. For our purposes, the atomic types are integers, floating point numbers, booleans, and strings. Compound types are composed of these atomic building blocks.

**Numbers**

Unlike Racket, OCaml does not support exact numbers. Instead, it has the following two number types:

- The first type of number is int. The primitives +, -, and * are infix operators on ints. The primitive / corresponds to Racket’s quotient, and mod corresponds to Racket’s remainder.

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Racket</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 + 18</td>
<td>(+ 17 18)</td>
</tr>
<tr>
<td>17 + -15</td>
<td>(+ 17 -15)</td>
</tr>
<tr>
<td>18 / 17</td>
<td>(quotient 18 17)</td>
</tr>
<tr>
<td>18 mod 17</td>
<td>(remainder 18 17)</td>
</tr>
</tbody>
</table>

  The int type can only represent integers in the range $[-2^{30}, 2^{30} - 1]$. If you were to add 1 to 1073741823, you would get -1073741824 as your answer. This is known as overflow.

For example:
• The second type of number is float, which is like an inexact number in Racket. If you add a “dot” at the end of your int, it becomes a float.

```
18. +. 17.
- : float = 35.
18. /. 17.
- : float = 1.05882352941176472
```

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Racket</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.17 +. 18.18</td>
<td>(+ 17.17 18.18)</td>
</tr>
<tr>
<td>17. +. -15.0</td>
<td>(+ 17. -15.0)</td>
</tr>
<tr>
<td>18. /. 17.</td>
<td>(/ 18 17)</td>
</tr>
<tr>
<td>18. ** 17.</td>
<td>(expt 18 17)</td>
</tr>
</tbody>
</table>

**Question:** What happens if you try to evaluate 15 +. 1.0 in OCaml?

**Answer:** You cannot. Nor can you evaluate 15 + 1.0.

**Question:** Can you convert from ints to floats and vice versa?

**Answer:** Yes, you can do the former using the built-in procedure float_of_int: e.g., float_of_int 17 => 17.0. To do the latter, you can use the procedure int_of_float, which truncates a float like this: int_of_float 17.77 => 17. But it might be safer to first take the floor or ceiling of a float, like this: int_of_float (ceil 17.77) => int_of_float 18.0 => 18.

**Question:** What is the range of numbers floating point numbers can represent?

**Answer:** You will learn all about how computers represent numbers in CS33. For now, you can find the answer to your specific question here (OCaml abides by the IEEE standard): [http://en.wikipedia.org/wiki/Double-precision_floating-point_format](http://en.wikipedia.org/wiki/Double-precision_floating-point_format)

**Booleans** Just as in Racket, OCaml has two boolean values, **true** and **false**. They are of type **bool**. The primitive procedures **and** and **or** are spelled && and || in OCaml. These procedures take as input two expressions that evaluate to bools and output a bool.

**Question:** Is this expression legal?

```
(15 < 16) && (17. < 18.)
```

**Answer:** Absolutely. Both subexpressions are of type **bool**.

To compare values in OCaml, we use the binary infix operators <, >, <=, >=, ==, and <>. The last of these tests for inequality.

As in Racket, these operators are polymorphic: i.e., they can be used to compare any two values of the same type. But in OCaml, you can’t mix and match types. So integers and floats are incompatible: 1 > 2 is valid and 1.0 > 2.0 is valid, but 1 > 2.0 is not. Similarly, 1 = 2 and 1.0 = 2.0 are syntactically valid, but 1 = 2.0 is not.
Unlike the arithmetic operators, which are sometimes suffixed by dots, the built-in comparators apply to values of any type, as long as the two values are of the same type, so there is no need for any dots, as in `<.` or `>`.

OCaml does not have a construct like `cond`. Instead of `cond`, OCaml has a much more powerful system, called pattern matching, which you will learn about very soon.

**Strings**  A string in OCaml is just like a string in Racket. The caret, `^`, is a primitive binary infix operator which produces the concatenation of any two strings, like the `string-append` procedure in Racket.

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Racket</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;seven&quot; ^ &quot;teen&quot;</td>
<td>(string-append &quot;seven&quot; &quot;teen&quot;)</td>
</tr>
</tbody>
</table>

### 8 If-Expressions

Similar to the `if` expressions in Racket, OCaml has if expressions.

The shape of an `if` expression in OCaml is: `if ⟨question⟩ then ⟨answer-t⟩ else ⟨answer-f⟩`. For example,

```ocaml
if 17 < 18 then "hello"
else "goodbye"
```

Here is an example of nested `ifs`:

```ocaml
if 17 < 18 then
  if 18 < 19 then "hello"
  else "goodbye"
else "goodbye, again"
```

Every OCaml expression is typed at compile time, so the following is invalid OCaml code:

```ocaml
if 17 < 18 then "fifteen"
else 16
```

This is not a statement which goes if X, do Y, else do Z, but rather an expression, as in Racket. An `if` expression has a value. Therefore, the `if` expression needs to have a type. If the `true` and `false` cases produce different types, there is no way to define a type for the entire `if` expression. So `true` and `false` cases must have the same type.

### 9 Compound Types

You learned about two compound types in Racket: structures (structs) and lists. In OCaml, you will work with analogous compound types. Tuples (and sometimes records) take the place of structures, and lists are themselves lists.
**Tuples**  A tuple is a container for a fixed number of values. OCaml tuples serve the same purpose as (small) Racket structures. Both facilitate the manipulation of compound data that can constitute smaller data of varying types.

A tuple is shaped like this:

\[(<expr>, <expr>, ..., <expr>)\]

Technically, this syntax is a built-in tuple constructor, meaning it is a mechanism by which to construct compound data.

Here are some examples of tuples and their corresponding types:

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&quot;hello&quot;, &quot;world&quot;)</td>
<td>string * string</td>
</tr>
<tr>
<td>(&quot;it's&quot;, true)</td>
<td>string * bool</td>
</tr>
<tr>
<td>(true, 17)</td>
<td>bool * int</td>
</tr>
<tr>
<td>((true, 3.14), &quot;hello&quot;)</td>
<td>(bool * float)* string</td>
</tr>
</tbody>
</table>

**Question:** Is \((<expr>)\) a tuple?

**Answer:** No, it is not. A tuple must have at least two components.

**Question:** If tuples are like structures in Racket, then what in OCaml are the analogs of constructors, selectors, and type predicates?

**Answer:** In OCaml, tuples are constructed using the special syntax presented above, and below you’ll see the special syntax for constructing lists. Pattern matching (coming soon, to a lecture near you!) replaces selectors and type predicates.

**Lists**  Lists are as fundamental in OCaml as they are in Racket.

As usual, there are two kinds of lists:

- An atomic list called the empty list and spelled `[ ]`.
- A pair, \(d :: \text{alod}\), where \(d\) is a datum of some type and \(\text{alod}\) is a list of data of that type.

Recall that \texttt{cons} stands for “construct.” Racket’s prefix procedure \texttt{cons} corresponds to an infix constructor, namely ::, in OCaml.

**Note:** The pair constructor :: is right associative: i.e., it binds most tightly to the right.

<table>
<thead>
<tr>
<th>OCaml</th>
<th>Racket</th>
</tr>
</thead>
<tbody>
<tr>
<td>17 :: [ ]</td>
<td>(cons 17 empty)</td>
</tr>
<tr>
<td>&quot;hello&quot;:: &quot;world&quot;:: [ ]</td>
<td>(cons &quot;hello&quot; (cons &quot;world&quot; empty))</td>
</tr>
<tr>
<td>17+18 :: 31+32 :: 22+51 :: [ ]</td>
<td>(cons (+ 17 18) (cons (+ 31 32) (cons (+ 22 51) empty)))</td>
</tr>
</tbody>
</table>

The first and third of these are of type int list, while the second is of type string list.
In OCaml, there is syntactic sugar for both writing and displaying lists, which looks like this:

```
["expr"> "," <expr> "," ... "," <expr> "]
```

For example,

```
[17+1 ; 16-1]
- : int list = [18 ; 15]

["hello" ; "world"]
- : string list = ["hello" ; "world"]
```

Similarly,

```
17 :: [ ]
- : int list = [17]

17 :: 17 + 1 :: [ ]
- : int list = 17 :: 18 :: [ ]
- : int list = 17 :: [18]
- : int list = [17 ; 18]
```

```
(4 + 1 :: 4 + 2 :: [ ]) :: [10 ; 20 ; 30] :: [ ]
- : int list list = (4 + 1 :: 4 + 2 :: [ ]) :: [[10 ; 20 ; 30]]
- : int list list = (4 + 1 :: 6 :: [ ]) :: [[10 ; 20 ; 30]]
- : int list list = (4 + 1 :: [6]) :: [[10 ; 20 ; 30]]
- : int list list = [5 :: [6]] :: [[10 ; 20 ; 30]]
- : int list list = [[5 ; 6 ; 10 ; 20 ; 30]]
```

OCaml lists are **monomorphic**, which means that all the elements of a list must be of the same type. Hence, `["hello" ; 17]` is an error, for example.

Furthermore, we can use the `@` symbol to append lists as follows:

```
[1; 2] @ [3; 4]
- : int list = [1; 2; 3; 4]
```

### 10 Binding to Identifiers

If you want to bind an identifier to a value, you can do:

```
let x = 5;;
```

What if you want \( f(x) \) to be \( x + 1 \)\.? You can do:

```
let f x = x + 1;;
```

You can also put parentheses around the \( x \), but a lot of OCaml programmers don’t:
let f(x) = x + 1;;

The next line is the CS17 approved form of that, where I’ve put parentheses around the x and said x is an integer, and the result of the computation is an integer, and the way you compute it is adding 1 to x. What I did here is called type annotation— I’ve annotated every type.

let f(x: int):int = x + 1;;

What happens if I type this in OCaml?

let f(x) = x + 1;;
- : val f : int -> int = <fun>

As you can see, OCaml produces the name, the type, and the value of the function. You might be wondering— how did it know it was an int -> int function? If you look at the expression on the right, x+1, + is a function that takes two arguments, both of which are integers. 1 is an integer, and OCaml looks at this and knows x must be an integer. And, it knows if you add an integer to an integer, you get an integer. So, it infers that the procedure you’ve written is int -> int.

While you may be tempted to not do the type annotation, and let OCaml tell you what type signature you should use, it’s a really bad idea. Here’s what happens— you write your code, and at some point you make a typo. OCaml tries its best to make sense of what you’ve typed, and OCaml will make the wackiest inferences you will imagine. Far better to actually say what you meant as you go along so it can tell you right away that you screwed up.

11 Summary

Ideas

• Introduced the purpose and terminology of the Tree data structure
• Introduced the OCaml language and its handling of data types
• Learned about atomic types, if expressions, compound types and bindings in OCaml

Skills

• Understanding the applications and limitations of using lambda
• Implementing Trees in Racket
• Analyzing procedures involving trees

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs17/feedback