Chapter 2: Data Abstraction

2.2 An Abstraction for Inductive Data Types

- Recall inductive BNF definition of binary trees:

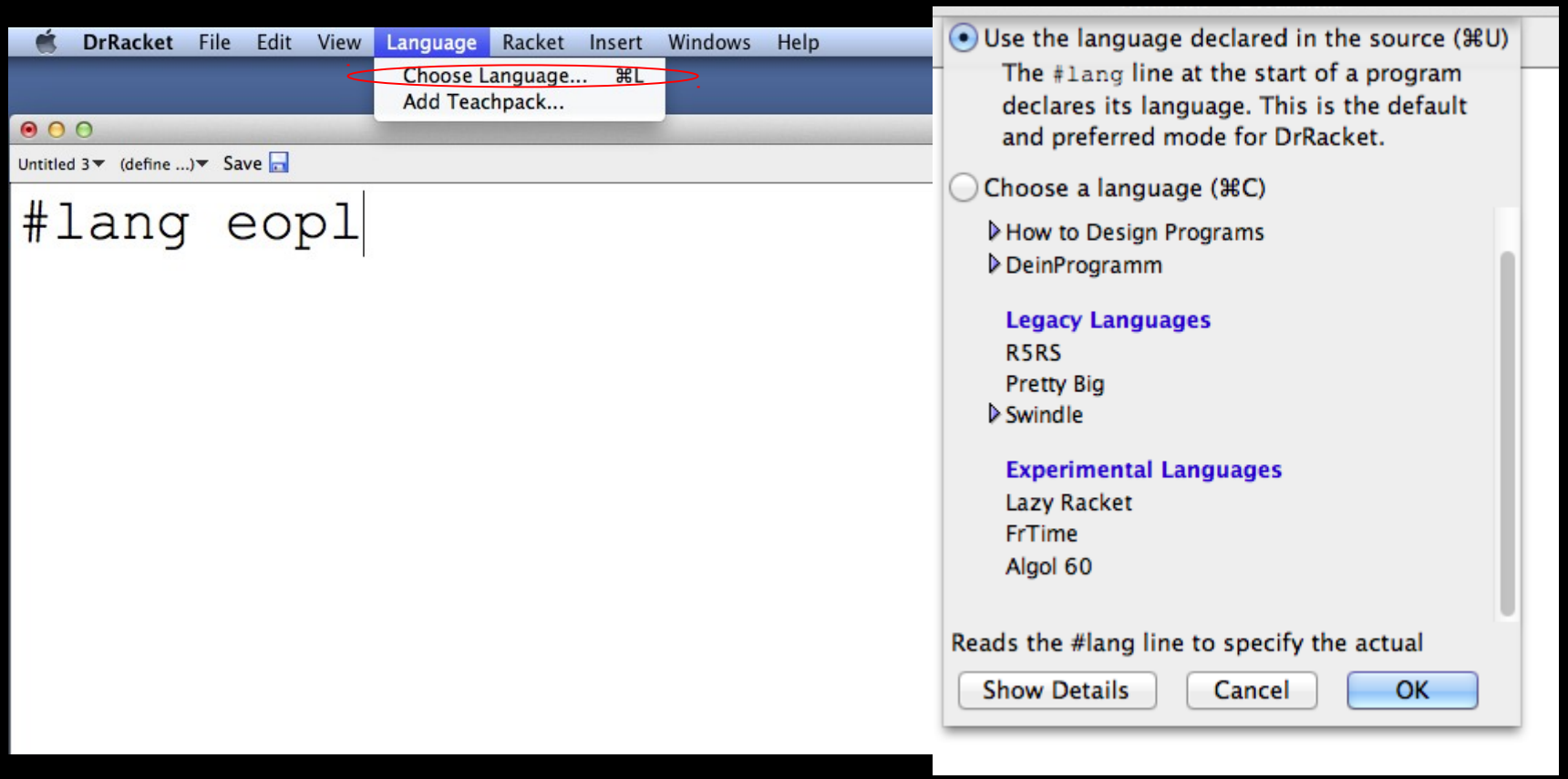
<bintree> ::= <number> | (<symbol> <bintree> <bintree>)

- To write programs using bintree, we'll need:
  - *constructors* for different kinds of bintrees (like Java)
  - a *predicate* for telling whether an object is a bintree

    (unlike Java, because Scheme types are dynamic)
  - a way of telling whether bintree is leaf or interior
  - a way of getting out the leaf or interior component

An Abstraction for Inductive Data Types

- Tell Dr. Racket to use the language extension from *Essentials of Programming Languages* (EOPL) textbook:



An Abstraction for Inductive Data Types

- Now we can use the **`define-datatype`** special form:

```
(define-datatype bintree bintree?
    (leaf-node
        (datum number?))
    (interior-node
        (key symbol?)
        (left bintree?)
        (right bintree?)))
```

- What is this doing?

- Defines a datatype **`bintree`**, with predicate **`bintree?`**

```
(define-datatype bintree bintree?
```

- A bintree is either

  - a leaf node consisting of a datum, which is a number

```
(leaf-node (datum number?))
```

  - an interior node consisting of a key that is a symbol, a left child that is a bintree, and a right child that is a bintree

```
(interior-node
        (key symbol?)
        (left bintree?)
        (right bintree?)))
```

- Creates a datatype with the following interface:

```
(define-datatype bintree bintree?
```

- A 1-argument procedure for constructing a leaf node, with a test to make sure that the argument is a number

```
  (leaf-node (datum number?))
```

- A 3-argument procedure for building an interior node, with a test for the first arg being a symbol and the second and third being a bintree

```
          (interior-node
                  (key symbol?)
                  (left bintree?)
                  (right bintree?)))
```

More on **`define-datatype`**

General format:

`(define-datatype` *type-name type-predicate-name*

{**`(`***variant*-*name* {**`(`***field-name predicate***`)`**}*}+**`)`**

```
(define-datatype bintree bintree?
    (leaf-node
        (datum number?))
```

Determining variants via **`cases`**

```
(define leaf-sum
  (lambda (tree)
    (cases bintree tree
      (leaf-node (datum) datum)
      (interior-node (key left right)
        (+ (leaf-sum left)
           (leaf-sum right))))))
```

(Essentially, a pattern-matcher)

Determining variants via **`cases`**

Supports type-checking:

```
> (define tree-a
          (interior-node 'a (leaf-node 2)
        (leaf-node 3)))
> (leaf-sum tree-a)
  5
> (leaf-sum 'foo)
  case: not a bintree: foo
```

Determining variants via **`cases`**

General format:

`(cases` *type-name expression*

{**`(`***variant-name* **`(`**{*field-name*}*__`)`__ *consequent*__`)`__}*

```
(cases bintree tree
  (interior-node (key left right)
    (+ (leaf-sum left)
       (leaf-sum right)))))
```