Lazy Functional Programming and Domain Specific Languages

Lazy programming

Lazy Evaluation	Eager Evaluation
Haskell	Java
.NET `System.Lazy<T>`	Python
Miranda	ML

From FutureLearn's Programming in Haskell, from University of Glasgow

Lazy Programming

- Haskell is *lazy*. So unless specifically told otherwise, Haskell won't execute functions until it needs to show you a result.
- This is made possible by referential transparency. If you know that the result of a function depends only on the parameters that function is given, it doesn't matter when you actually calculate the result of the function.
- Haskell, being a lazy language, takes advantage of this fact and defers actually computing results for as long as possible.

Lazy Programming

- Once you want your results to be displayed, Haskell will do just the bare minimum computation required to display them.

- Laziness also allows you to make seemingly infinite data structures, because only the parts of the data structures that you choose to display will actually be computed.

Lazy Evaluation

- Say you have a infinite list of numbers,

  xs = [1..] -- infinite list starting at 1

  fxs = [5..10] -- finite list

  and the **take** function

  (take n xs takes the first n elements)

  You can have

  >take 4 [1..]

  [1,2,3,4]

  ➢ fxs `zip` [1..]

  [(5,1), (6,2), (7,3), (8,4) (9,5), (10,6)]

Lazy Evaluation – other handy functions

- Generating infinite lists

```
repeat::a ->  [a]
repeat x = x: repeat x

cycle::[a] ->  [a]
cycle xs =xs ++cycle xs
```

Lazy Evaluation

Do not evaluate an expression unless its value is needed

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

iterate (*2) 1 => [1, 2, 4, 8, 16, …]
```

```
powertables :: [[Int]]
powertables = [ iterate (*n) 1 | n <- [2..] ]
  => [ [1, 2, 4,  8,…],   [1, 3, 9, 27,…],
          [1, 4,16, 64,…],    1, 5, 25,125,…],  … ]
```

Sieve of Eratosthenes Algorithm

1. Generate list 2, 3, 4, …
2. Mark first element p of list as prime
3. Delete all multiples of p from list
4. Return to step 2

```
primes :: [Int]
primes = map head (iterate sieve [2..])

sieve (p:xs) = [x | x <- xs, x `mod` p /= 0 ]


myprimes = takewhile  (< 10000)  primes
```

Lazy IO

```
headFile f = do
        c <- readFile f
        let c' = unlines. take 5. lines $c
        putStrLn c'
```

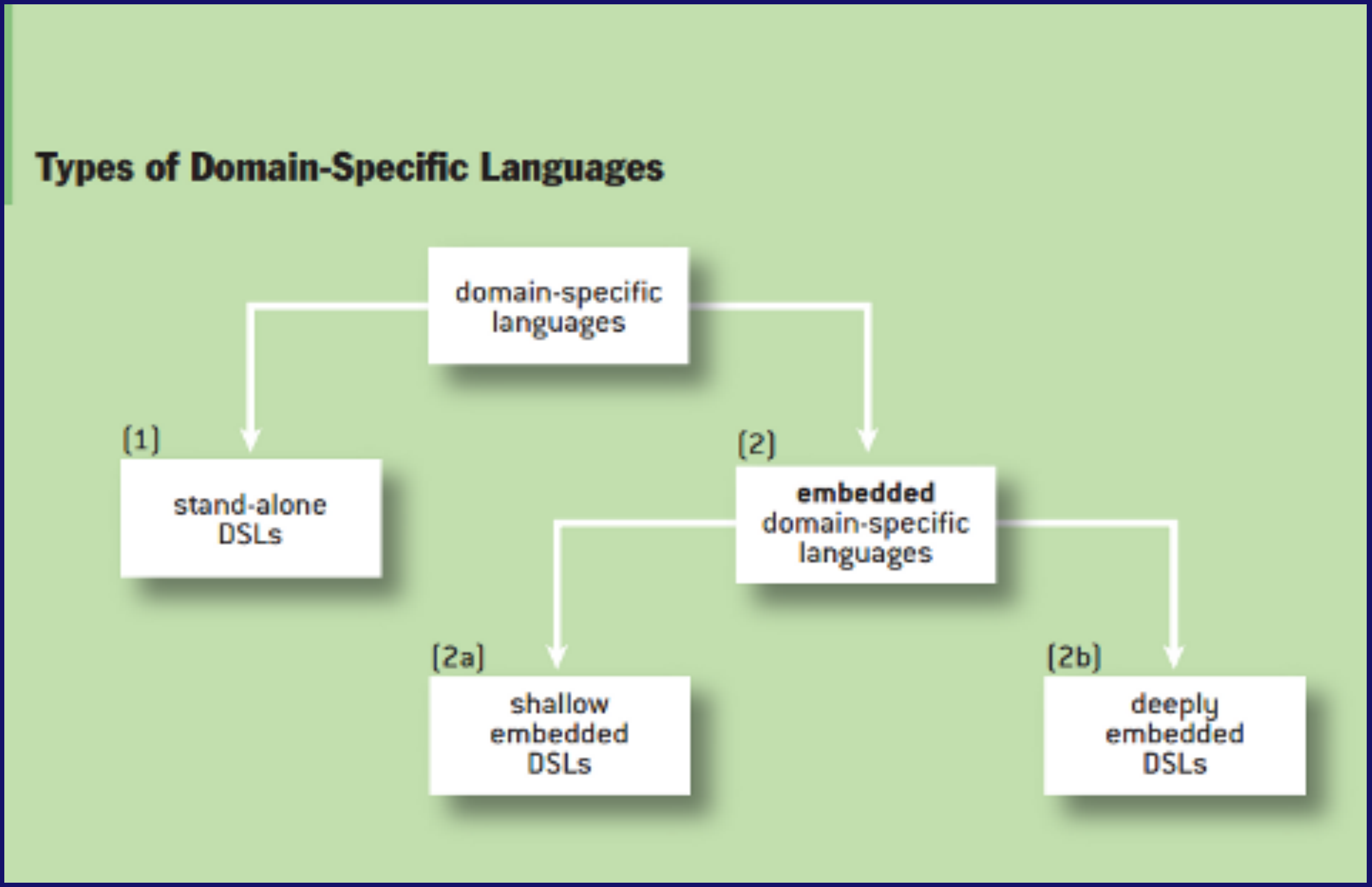
```
the
cat
sat
on
the
mat
seven
-- contents of “text.txt”
```

text.txt

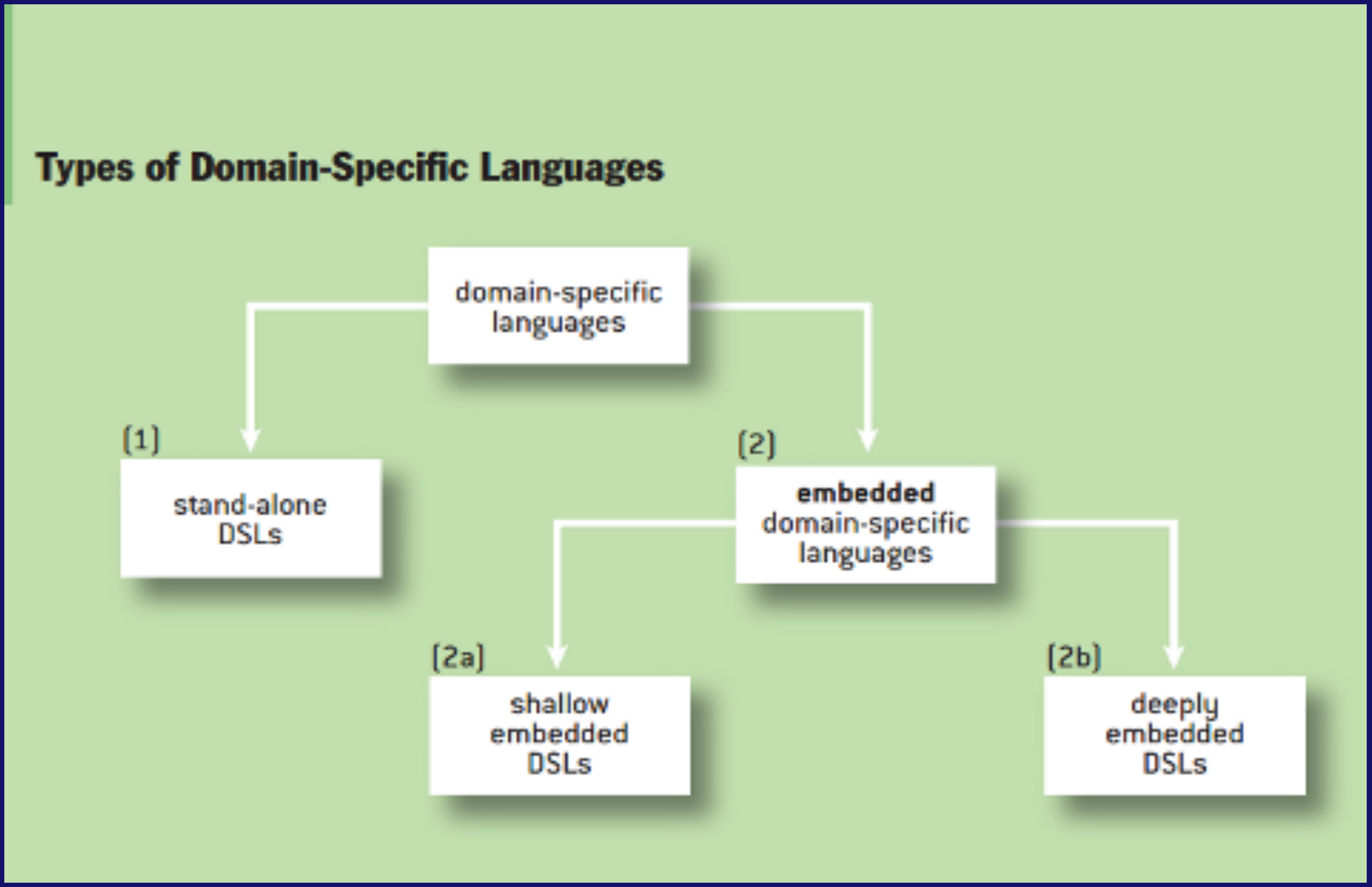
```
>headFile “text.txt”
the
cat
sat
on
the
```

Running headFile

Domain Speficic Languages (DSLs)



Domain Speficic Languages (DSLs)



Domain Specific Languages

A DSL is a special-purpose language,

- designed to encapsulate possible computations in a specific domain.
- In the examples of MATLAB, SQL, Verilog, and spreadsheets, the domains would be scientific modelling, database queries and updates, hardware circuits, and financial computations, respectively.

Domain Specific Languages

- Considering SQL specifically, there is nothing it does that could not be done in Java or C, or any other general-purpose programming language.
- SQL simply bundles the actions needed to interact with a database into a usable and productive package, and the language becomes the interface to communicate requests to the database engine.

Domain Specific Languages

- There are two fundamental types of DSLs.
- The first is a first-class language, with its own compiler or interpreter, and it is often used in its own ecosystem.
  - SQL,MA TLAB, Latex all fall into this category.
- The other class of DSL is a language embedded in a host language. Such languages can have the look and feel of being their own language, but they live in the host language.

Domain Specific Languages

- An EDSL (embedded DSL) is a language inside a language.
- Haskell,, is a great host for EDSLs because of
  - flexible overloading,
  - powerful type system, and
  - lazy semantics.

Domain Specific Languages

- An EDSL is a library in a host language that has the look, feel, and semantics of its own language, customized to a specific problem domain.
- By reusing the facilities and tools of the host language, an EDSL considerably lowers the cost of both developing and maintaining a DSL.

Example ESDL -QuickCheck

- Consider the challenge of writing test cases—or more specifically, writing the *properties* that test cases need to satisfy:

```
-- The reverse of a reversed list is itself

prop_reverse_twice (xs :: [Int]) =
        reverse (reverse xs) == xs
```

Example ESDL -QuickCheck

In this example,

prop_reverse_twice

is a regular Haskell function that takes a list of Int and returns a Boolean, based on the validity of what is being proposed—specifically, that two reverses cancel each other out.

prop_reverse_twice

is *also* a domain-specific statement and as such can be considered a sublanguage inside Haskell.

Example ESDL -QuickCheck

This style of using functions (in this case, functions with names prefixed with prop_, taking a number of typed arguments, and returning a conditional) is a ***small language***.

The property checker written in Haskell is also an EDSL for properties, called QuickCheck. This EDSL can be run using a function also called quickCheck:

```
Prelude Test.QuickCheck> quickCheck
prop_reverse_twice
+++ OK, passed 100 tests.
```

Example ESDL -QuickCheck

- By running ***quickCheck*** with this explicit and specific property, the EDSL executes inside Haskell.
- The ***quickCheck*** function generates 100 test cases for the property and executes them on the fly. If they all hold, then the system prints a message reflecting this.
- The test cases are generated using the type class system—***QuickCheck*** gives specific types the power of test-case generation—and the *quickCheck* function uses this to generate random tests.

Building a Deep EDSL

There is another class of EDSLs, however: specifically, those that use a deep embedding to build an abstract syntax tree.

The result of a computation inside a deeply embedded DSL (deep EDSL) is a structure, not a value, and this structure can be used to compute a value or be cross-compiled before being evaluated.

Building a Deep EDSL