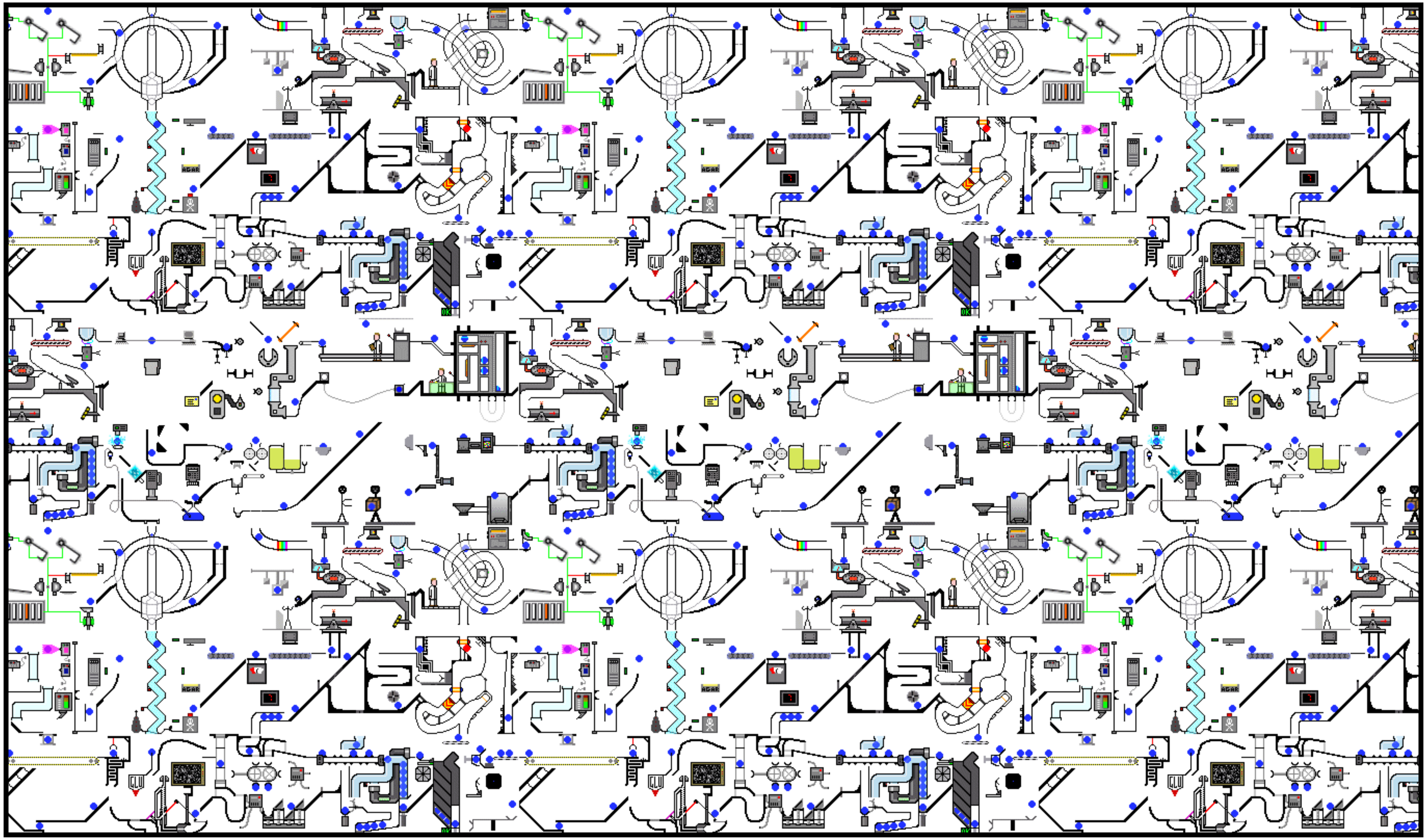
Graph Reduction



How to interpret Haskell

1. **Translate Haskell into a small core language**
    - lambda calculus + literals + recursive let + case + ...
2. **Represent core expressions as DAGs**
    - references are edges in the graph
    - supports sharing during evaluation
3. **Evaluate by "graph reduction"**
    - set of graph transformation rules
    - implements lazy evaluation

Core language

```
data Literal = ...
```

data constructors, primitive functions, string and numeric literals, …

```
data Expr
  = Lit Literal
  | Ref Var
  | App Expr Expr
  | Lam Var Expr
  | Let Var Expr Expr
  | Case Expr [(Pat,Expr)]
```

lambda calculus (Ref, App, Lam)

```
data Pat
  = Default
  | Alt Literal [Var]
```

Example translation

```haskell
data Literal = ...

data Expr
  = Lit Literal
  | Ref Var
  | App Expr Expr
  | Lam Var Expr
  | Let Var Expr Expr
  | Case Expr [(Pat,Expr)]

data Pat
  = Default
  | Alt Literal [Var]
```

Recall: can translate type classes to dictionaries!

Haskell:

```haskell
map f []     = []
map f (x:xs) = f x : map f xs
```

Core (concrete):

```
let map = λf.λl.
  case l of
    []     -> []
    (x:xs) -> f x : map f xs
in ...
```

Core (abstract):

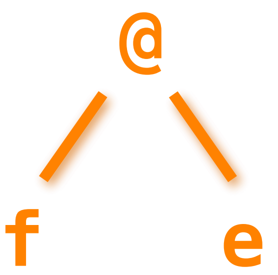
```
Let "map" (Abs "f" (Abs "l"
  (Case (Ref "l")
    [(Alt "[]" [], Lit "[]")
    ,(Alt ":" ["x","xs"],
       App (Lit ":")
        (App (Ref "f") (Ref "x"))
          (App (App (Ref "map") (Ref "f"))
            (Ref "xs"))]
  ...
```

Encoding core expressions as graphs

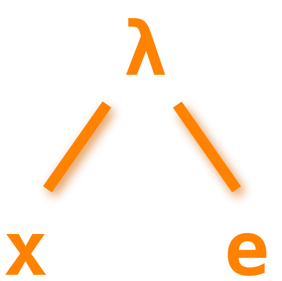
literals & primitives	leaves
function application	apply node: @
abstraction	lambda node: λ
let-expression	lambda + apply
references	back/cross edges

`2   +   :`

`f e →`

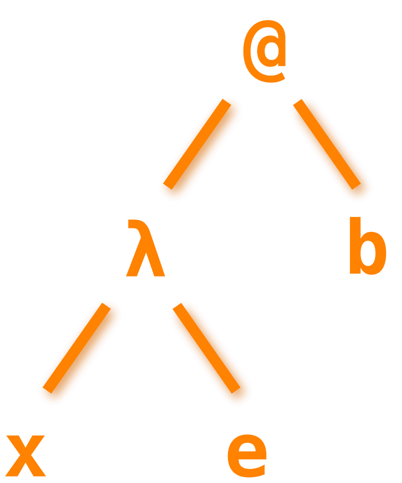


`λx.e →`




```
let x = b in e
      ≡
  (λx.e) b
```

→



Lazy evaluation

Goal: evaluate as few *application nodes* as possible

an unevaluated application node is called a ***thunk***

How do we know when we're done?

An expression **e** is in **weak head normal form** (WHNF) if it is:

- a *literal* or a *variable*
- an *abstraction*
- a partially applied *primitive function* or *constructor*

} *may contain thunks*

In other words, **e** *has no top-level redex!*

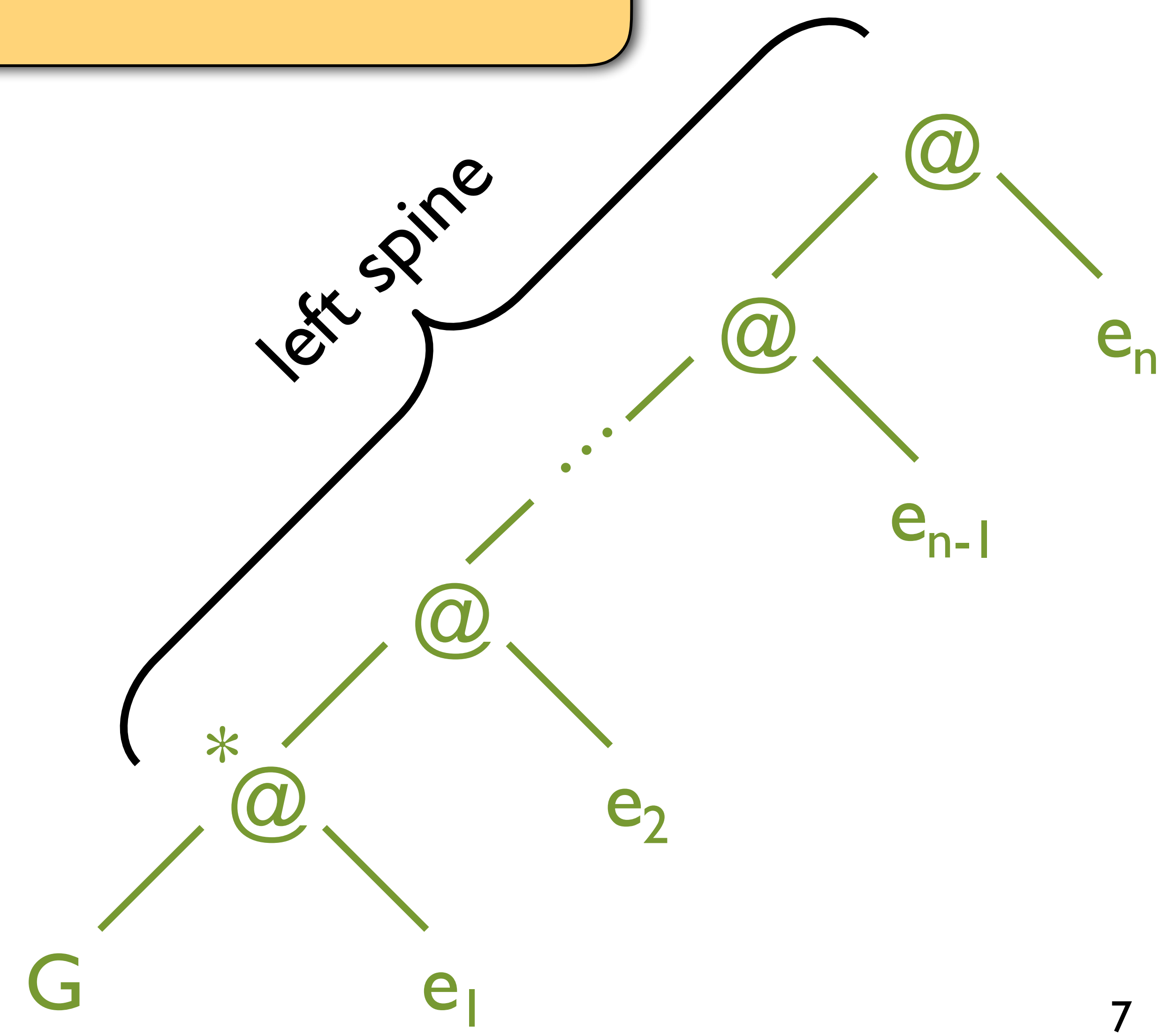
= nothing left to reduce in call-by-need (lazy) evaluation

Graph reduction

Repeat until graph is in WHNF:

- start from root, *find redex*
- if LHS is primitive function, reduce arguments
- perform reduction

Finding a redex:
first @ on left spine whose
whose LHS is not an @



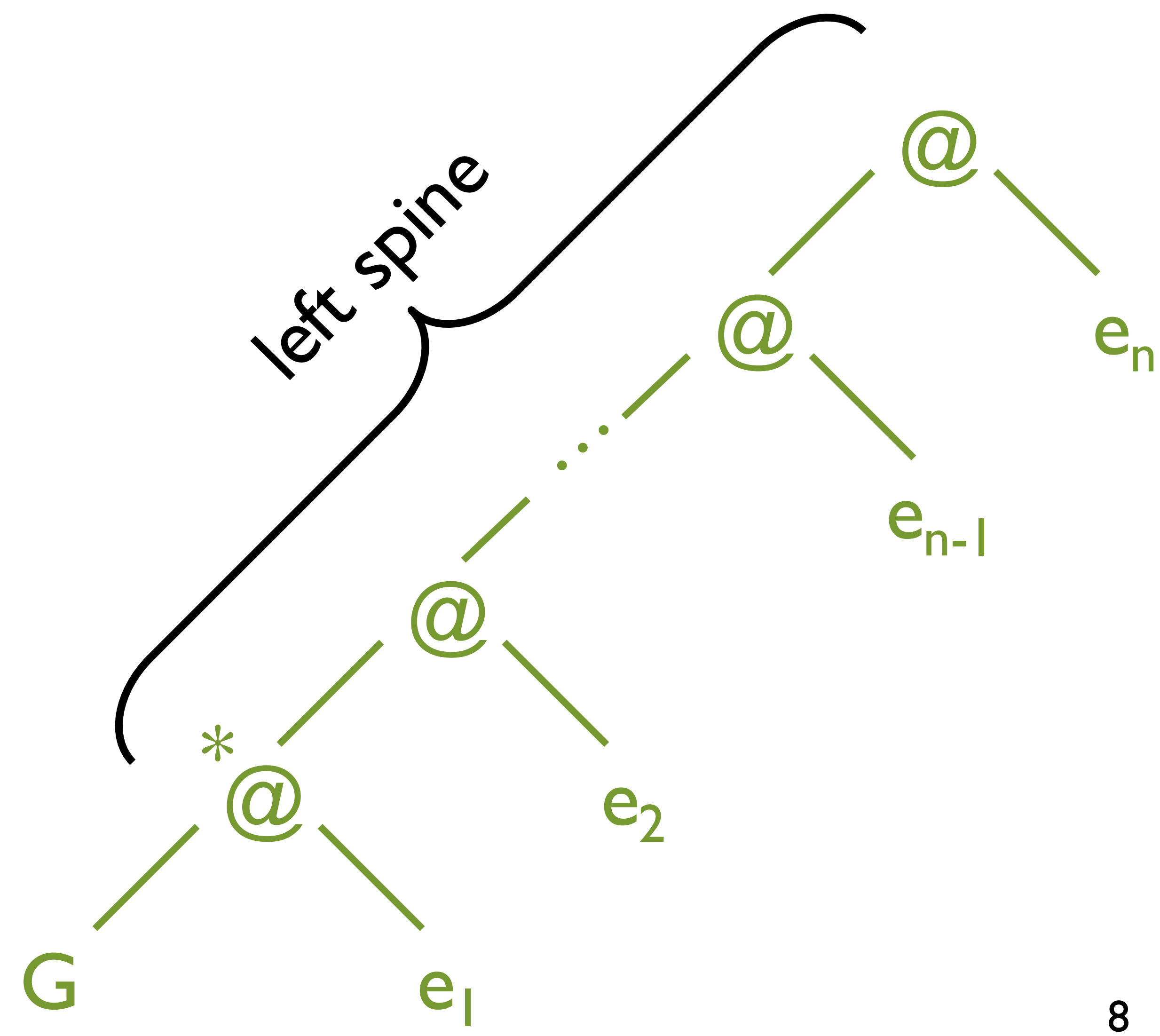
Constructor and primitive reduction

If G is constructor of arity k < n

1. substitute @ nodes w/ constructor node

If G is primitive of arity k < n

1. (reduce arguments)
2. apply function



β-reduction

If G is a **λ** *node*

1. copy lambda body
2. redirect references to argument
3. overwrite root