Beágyazott nyelvek

Bevezetés

Domain-specific language (DSL), „szakterület-specifikus nyelv”:

„A computer programming language of limited expressiveness focused on a particular domain.” – Martin Fowler, *Domain-Specific Languages*, 2011

Például: 4GL nyelvek, SQL, UNIX-os segédprogramok: shellek, reguláris kifejezések, *lex*(1), *yacc*(1).

- Az adott (szak)területhez tartozó fogalmak könnyebben megfoghatóak, közvetlenebb és egységesebb formában fejezhetőek ki.
- Ellenben minden más a (szak)területen kívül nehezen írható le.
- Nem feltétlenül Turing-teljes.
- Haskell = DSL for compilers :-)

DSL-ek megvalósítása

DSL-eket kétféleképpen szoktak megvalósítani:

- *Önálló (standalone)*: Saját szintaxissal rendelkezik, a végrehajtáshoz a hagyományos fordítási technikák segítségével fordítjuk a DSL programokat általános célú programokra.
  - A szintaxis a célközönség számára alakítható.
  - Külön elemző, fordító és fejlesztői környezet kell.
  - Gyakran redundáns és fáradságos fejlesztés.
- *Beágyazott (embedded)*: Egy (általános célú) befogadó nyelven (host language) keresztül adjuk meg programkönyvtárként, makrócsomagként.
  - embedded domain-specific language, EDSL

DSL-ek megvalósítása

DSL-eket kétféleképpen szoktak megvalósítani:

- *Önálló (standalone)*: Saját szintaxissal rendelkezik, a végrehajtáshoz a hagyományos fordítási technikák segítségével fordítjuk a DSL programokat általános célú programokra.
- *Beágyazott (embedded)*: Egy (általános célú) befogadó nyelven (host language) keresztül adjuk meg programkönyvtárként, makrócsomagként.
  - Az így rendelkezésre álló eszközök felhasználhatóak, beleértve a szintaktikai konvenciókat is.
  - Ha a befogadó nyelv merev, akkor a szintaxis kényelmetlen tud lenni.
  - Nehéz meghúzni az EDSL és befogadó nyelv közti határt.
  - A hibaüzenetek olykor semmitmondóak.

DSL-ek beágyazása

A funkcionális nyelvek elég jól használhatóak beágyazásra:

- Magasabb rendű függvények, sémák
- Lusta kiértékelés, végtelen adatszerkezetek
- Gazdag (programozható!) típusrendszer, típuskikövetkeztetés

Nyelvbeágyazási stílusok:

- *Mély beágyazás (deep embedding)*: A DSL egyes szerkezeteit megvalósító függvények egy absztrakt szintaxisfát (AST) építenek fel, amelyet transzformálhatunk, futtatunk (algebra).
- *Sekély beágyazás (shallow embedding)*: A DSL egyes szerkezeteit közvetlenül a neki megfelelő függvényekkel fejezzük ki, nincs AST, sem bejárás.
- Az előbbi kettő valamilyen kombinációja.

Példa: BASIC Haskellben

```
-- http://hackage.haskell.org/package/BASIC
import Language.BASIC

main :: IO ()
main = runBASIC $ do
  10   GOSUB 1000
  20   PRINT "* Welcome to HiLo *"
  30   GOSUB 1000
  100  LET I := INT(100 * RND(X))
  200  PRINT "Guess my number:"
  210  INPUT X
  220  LET S := SGN(I - X)
  230  IF S <> 0 THEN 300
  240  FOR X := 1 TO 5
  250  PRINT X * X; " You won!"
  260  NEXT X
  270  STOP
  300  IF S <> 1 THEN 400
  310  PRINT "Your guess"; X; " is too low."
  320  GOSUB 200
  400  PRINT "Your guess"; X; " is too high."
  410  GOSUB 200
  1000 PRINT "* * * * * * * * * * * * * * * * * * * *"
  1010 RETURN
  9999 END
```

További példák EDSL-ekre

```
-- http://hackage.haskell.org/package/haskelldb
import Database.HaskellDB

query = do
  cust ← table customers
  restrict (cust ! city .== .''Budapest'')
  project (cust ! customerID)


-- http://hackage.haskell.org/package/uu-parsinglib
import Text.ParserCombinators.UU

expr = Let <$ keyword ''let'' <*> decl <*
                keyword ''in'' <*> expr
       <|> operatorExpr


-- https://hackage.haskell.org/package/accelerate
import Data.Array.Accelerate

dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

Egy nagyon egyszerű DSL

Készítsünk egy DSL-t, amely egész számok véges halmazait írja le!

$$\textbf{type}\ \mathit{IntegerSet}$$

$$\begin{array}{lll} \mathit{empty} & :: \mathit{IntegerSet} & \\ \mathit{insert} & :: \mathit{Integer} \rightarrow \mathit{IntegerSet} & \rightarrow \mathit{IntegerSet} \\ \mathit{delete} & :: \mathit{Integer} \rightarrow \mathit{IntegerSet} & \rightarrow \mathit{IntegerSet} \\ \mathit{member} & :: \mathit{Integer} \rightarrow \mathit{IntegerSet} & \rightarrow \mathit{Bool} \end{array}$$

Példa az alkalmazásra:

$$x = \mathit{member}\ 3\ (\mathit{insert}\ 1\ (\mathit{delete}\ 3\ (\mathit{insert}\ 2\ (\mathit{insert}\ 3\ \mathit{empty}))))$$

Egy nagyon egyszerű DSL – mint könyvtár

Az egész számok halmaza tulajdonképpen könyvtárként is megvalósítható, a kapcsolódó típusok, függvények gyűjteményeként.

```
type IntegerSet = [Integer]

empty :: IntegerSet
empty = []

insert :: Integer → IntegerSet → IntegerSet
insert = (:)

delete :: Integer → IntegerSet → IntegerSet
delete x = filter (≢ x)

member :: Integer → IntegerSet → Bool
member x = any (≡ x)

-- Alkalmazás:
x = 3 `member` (1 `insert` (3 `delete` (2 `insert` (3 `insert` empty))))
```

A reprezentáció nem rejtett.

Egy nagyon egyszerű DSL – mint modul

```
module SetModule (Set, runSet, insert, delete, member) where
import Control.Monad.State

type IntegerSet = [Integer]

newtype Set α = S (State IntegerSet α)
  deriving (Functor, Applicative, Monad)

runSet :: Set α → α
runSet (S s) = evalState s []

insert :: Integer → Set ()
insert x = S $ modify (x :)

delete :: Integer → Set ()
delete x = S $ modify (filter (≢ x))

member :: Integer → Set Bool
member x = S $ get >>= return ∘ any (≡ x)

-- Alkalmazás:
x = runSet $ do { insert 3; insert 2; delete 3; insert 1; member 3 }
```

Egy nagyon egyszerű DSL – mint absztrakt adattípus

```
module SetADT (IntegerSet, empty, insert, delete, member) where

newtype IntegerSet = IS { unIS :: [Integer] }

empty :: IntegerSet
empty = IS []

insert :: Integer → IntegerSet → IntegerSet
insert x = IS ∘ (x :) ∘ unIS

delete :: Integer → IntegerSet → IntegerSet
delete x = IS ∘ filter (≢ x) ∘ unIS

member :: Integer → IntegerSet → Bool
member x = any (≡ x) ∘ unIS

-- Alkalmazás:
x = 3 `member` (1 `insert` (3 `delete` (2 `insert` (3 `insert` empty))))
```

Teljesen funkcionális stílus: nincs monád, mindig új halmazokat szerkesztünk.

Egy nagyon egyszerű DSL – mélyen beágyazva

```
module SetDeep (IntegerSet (Empty, Insert, Delete), member) where

data IntegerSet
  = Empty
  | Insert Integer IntegerSet
  | Delete Integer IntegerSet

member :: Integer → IntegerSet → Bool
_ `member` Empty            = False
x `member` (y `Insert` ys)  = (x ≡ y) ∨ x `member` ys
x `member` (y `Delete` ys)  = (x ≢ y) ∧ x `member` ys

-- Alkalmazás:
x = 3 `member` (1 `Insert` (3 `Delete` (2 `Insert` (3 `Insert` Empty))))
```

Egy nagyon egyszerű DSL – sekélyen beágyazva

```
module SetShallow (IntegerSet, empty, insert, delete, member) where

newtype IntegerSet = IS (Integer → Bool)

empty :: IntegerSet
empty = IS $ λ y . False

insert :: Integer → IntegerSet → IntegerSet
x `insert` (IS f) = IS $ λ y . (x ≡ y) ∨ f y

delete :: Integer → IntegerSet → IntegerSet
x `delete` (IS f) = IS $ λ y . (x ≢ y) ∧ f y

member :: Integer → IntegerSet → Bool
x `member` (IS f) = f x

-- Alkalmazás:
x = 3 `member` (1 `insert` (3 `delete` (2 `insert` (3 `insert` empty))))
```

Egy nagyon egyszerű DSL – összefüggés

A sekély beágyazást a mély beágyazásból úgy kapjuk, ha felszámoljuk a köztes adatszerkezetet, vagyis az AST-t.

$$
\begin{aligned}
&\mathit{elements} :: \mathit{SetDeep.IntegerSet} \to \mathit{SetADT.IntegerSet} \\
&\mathit{elements}\ \mathit{Empty} = \mathit{SetADT.empty} \\
&\mathit{elements}\ (x\ \text{`}\mathit{Insert}\text{`}\ \mathit{xs}) = x\ \text{`}\mathit{SetADT.insert}\text{`}\ (\mathit{elements}\ \mathit{xs}) \\
&\mathit{elements}\ (x\ \text{`}\mathit{Delete}\text{`}\ \mathit{xs}) = x\ \text{`}\mathit{SetADT.delete}\text{`}\ (\mathit{elements}\ \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{membership} :: \mathit{SetADT.IntegerSet} \to \mathit{SetShallow.IntegerSet} \\
&\mathit{membership}\ \mathit{xs} = \mathit{SetShallow.IS}\ \$\ \lambda\ x\ .\ \mathit{any}\ (\equiv\ x)\ (\mathit{SetADT.unIS}\ \mathit{xs})
\end{aligned}
$$

Az implementációs technikák összehasonlítása

Mély beágyazás: a *legabsztraktabb* implementáció.

- Könnyű új értelmezőket (*observer*), nehéz viszont új műveleteket készíteni.
- Optimalizálás és konvertálás feldolgozás közben.
- Az AST vizualizálható és viselkedése nyomonkövethető.
- Nehezebb kihasználni a közös részkifejezések megosztását (*sharing*, ld. később bővebben).

Sekély beágyazás: a *legkonkrétabb* implementáció.

- Nehéz új értelmezőket, könnyű viszont új műveleteket készíteni.
- Közvetlenül denotációs szemantikát adunk meg.
- Viszonylag könnyen elérhetőek a Haskell lehetőségei (pl. rekurzió, sharing).
- Nehezen látható a viselkedése az egyetlen értelmező miatt.

Funkcionális előnyök: Típusok

- Az algebrai adattípusok kényelmesen használhatóak mélyen beágyazott nyelvek absztrakt szintaxisfáinak modellezésére:
  - Az értelmezők, esetenként a műveletek rekurzív függvényekként írhatóak meg, a szerkezet feletti indukcióval.
  - Az optimalizációk és transzformációk átrendezésekkel valósíthatóak meg.
  - Nagyon hasznos köztes forma integráció esetén, amikor szövegből hozzuk létre, és szövegre alakítjuk át.
- Különféle számítások írhatóak le ezen a módon (*IO* α), amelyek valamilyen konkrét Haskell-típushoz (α) kötődnek.
- Teljesen irányítható, hogy miként hozzuk létre, kombináljuk és számítjuk ki ezeket a számításokat.

Funkcionális előnyök: Típusok (példa)

Tekintsük a következő interfészt, nyelvet:

$$
\begin{array}{ll}
\mathit{mkInt} & :: \mathit{Int} \rightarrow \mathit{X}\ \mathit{Int} \\
\mathit{mkChar} & :: \mathit{Char} \rightarrow \mathit{X}\ \mathit{Char} \\
\mathit{combine} & :: \mathit{X}\ \alpha \rightarrow \mathit{X}\ \alpha \rightarrow \mathit{X}\ \alpha \\
\mathit{pair} & :: \mathit{X}\ \alpha \rightarrow \mathit{X}\ \beta \rightarrow \mathit{X}\ (\alpha,\ \beta) \\
\mathit{run} & :: \mathit{X}\ \alpha \rightarrow \alpha
\end{array}
$$

- *X Bool* sosem hozható létre.
- Ennek az EDSL-nek a típuskészlete kisebb.
- Az így beágyazott értékekhez a nekik megfelelő, optimális ábrázolást vagy számítási módot társíthatunk.

Funkcionális előnyök: Típusok (példa)

import *Control.Monad.ST*
import *Data.STRef*

$$
\begin{array}{ll}
\mathit{newSTRef} & :: \alpha \rightarrow \mathit{ST}\ \sigma\ (\mathit{STRef}\ \sigma\ \alpha) \\
\mathit{readSTRef} & :: \mathit{STRef}\ \sigma\ \alpha \rightarrow \mathit{ST}\ \sigma\ \alpha \\
\mathit{writeSTRef} & :: \mathit{STRef}\ \sigma\ \alpha \rightarrow \alpha \rightarrow \mathit{ST}\ \sigma\ () \\
\mathit{runST} & :: (\forall\ \sigma\ .\ \mathit{ST}\ \sigma\ \alpha) \rightarrow \alpha
\end{array}
$$

- A feladat:
  - Az erőforrás lefoglalása egy számításban
  - A legfoglalt erőforrás átadása egy másik számításnak
  - Az átadott erőforrás használata ott
- A megoldás (típusrendszerrel):
  - A számításokat és az erőforrásokat régiókkal paraméterezzük (mint egzisztenciális típus)
  - A számítások nem tudhatják, melyik régióban futnak
  - Az átadott hivatkozások által függés alakul ki

Funkcionális előnyök: Típusok (példa)

```
data Expr
  = ValI Integer
  | ValB Bool
  | Add Expr Expr
  | And Expr Expr
  | Eq Expr Expr
  | If Expr Expr Expr

eval :: Expr → Either Integer Bool
eval (ValI n) = Left n
eval (ValB b) = Right b
eval (x `Add` y) =
  case (eval x, eval y) of
    (Left m , Left n) → Left (m + n)
eval (x `And` y) =
  case (eval x, eval y) of
    (Right a, Right b) → Right (a ∧ b)
eval (x `Eq` y) =
  case (eval x, eval y) of
   (Left m , Left n)  → Right (m ≡ n)
   (Right m, Right n) → Right (m ≡ n)
eval (If x y z) =
  case (eval x) of
    Right b → if b then (eval y) else (eval z)
```

Funkcionális előnyök: Típusok (példa)

```
{-# LANGUAGE GADTs, KindSignatures #-}
data Expr :: ⋆ where
  ValI :: Integer → Expr
  ValB :: Bool → Expr
  Add :: Expr → Expr → Expr
  And :: Expr → Expr → Expr
  Eq :: Expr → Expr → Expr
  If :: Expr → Expr → Expr → Expr

eval :: Expr → Either Integer Bool
eval (ValI n) = Left n
eval (ValB b) = Right b
eval (x `Add` y) =
  case (eval x, eval y) of
    (Left m, Left n) → Left (m + n)
eval (x `And` y) =
  case (eval x, eval y) of
    (Right a, Right b) → Right (a ∧ b)
eval (x `Eq` y) =
  case (eval x, eval y) of
   (Left m, Left n)  → Right (m ≡ n)
   (Right m, Right n) → Right (m ≡ n)
eval (If x y z) =
  case (eval x) of
    Right b → if b then (eval y) else (eval z)
```

Funkcionális előnyök: Típusok (példa)

```
{-# LANGUAGE GADTs, KindSignatures #-}
data Expr :: ⋆ → ⋆ where
  ValI :: Integer → Expr Integer
  ValB :: Bool → Expr Bool
  Add  :: Expr Integer → Expr Integer → Expr Integer
  And  :: Expr Bool → Expr Bool → Expr Bool
  Eq   :: Eq α ⇒ Expr α → Expr α → Expr Bool
  If   :: Expr Bool → Expr α → Expr α → Expr α

eval :: Eq α ⇒ Expr α → α
eval (ValI n)       = n
eval (ValB b)       = b
eval (x `Add` y)    = (eval x) + (eval y)
eval (x `And` y)    = (eval x) ∧ (eval y)
eval (x `Eq` y)     = (eval x) ≡ (eval y)
eval (If x y z)     = if (eval x) then (eval y) else (eval z)
```

Probléma: Részkifejezések megosztása

Mély beágyazással elveszítjük a beágyazó nyelv bizonyos lehetőségeit, például a közös részkifejezések megosztását.

$$
\begin{aligned}
&\mathit{toString} :: \mathit{Expr} \rightarrow \mathit{String} \\
&\mathit{toString}\ (\mathit{Vall}\ n) \qquad\quad = \mathit{show}\ n \\
&\mathit{toString}\ (e_1\ \text{`}\mathit{Add}\text{`}\ e_2) = \\
&\quad \text{"("} \mathbin{++} \mathit{toString}\ e_1 \mathbin{++} \text{" + "} \mathbin{++} \mathit{toString}\ e_2 \mathbin{++} \text{")"} \\
&\ldots
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{tree} :: \mathit{Integer} \rightarrow \mathit{Expr} \\
&\mathit{tree}\ 0 = \mathit{Vall}\ 1 \\
&\mathit{tree}\ n = \\
&\quad \textbf{let}\ \mathit{shared} = \mathit{tree}\ (n - 1) \\
&\quad \textbf{in}\ (\mathit{shared}\ \text{`}\mathit{Add}\text{`}\ \mathit{shared})
\end{aligned}
$$

```
GHCi> toString (tree 3)
"(((1 + 1) + (1 + 1)) + ((1 + 1) + (1 + 1)))"
```

Hoppá! Megoldás: *Higher-Order Abstract Syntax (HOAS)*

HOAS: A típus bővítése (egyszerűsített)

$$
\begin{array}{l}
\textbf{data}\ \mathit{Expr}\ ::\ \star\ \rightarrow\ \star\ \textbf{where} \\
\quad \mathit{Val}\ \ ::\ \mathit{Integer}\ \rightarrow\ \mathit{Expr}\ \alpha \\
\quad \mathit{Add}\ ::\ \mathit{Expr}\ \alpha\ \rightarrow\ \mathit{Expr}\ \alpha\ \rightarrow\ \mathit{Expr}\ \alpha \\
\quad \mathit{Var}\ \ ::\ \alpha\ \rightarrow\ \mathit{Expr}\ \alpha \\
\quad \mathit{Let}\ \ ::\ \mathit{Expr}\ \alpha\ \rightarrow\ (\alpha\ \rightarrow\ \mathit{Expr}\ \alpha)\ \rightarrow\ \mathit{Expr}\ \alpha
\end{array}
$$

A változókkal kapcsolatban semmilyen feltételezésünk nem lehet:

$$
\begin{array}{l}
\mathit{tree}\ ::\ \mathit{Integer}\ \rightarrow\ \mathit{Expr}\ \alpha \\
\mathit{tree}\ 0\ = \mathit{Val}\ 1 \\
\mathit{tree}\ n\ = \\
\quad \mathit{Let}\ (\mathit{tree}\ (n\ -\ 1)) \\
\quad (\lambda\ \mathit{shared}\ .\ (\mathit{Var}\ \mathit{shared})\ `\mathit{Add}`\ (\mathit{Var}\ \mathit{shared}))
\end{array}
$$

HOAS: Az értelmező bővítése (egyszerűsített)

$$
\begin{array}{ll}
\mathit{eval} :: \mathit{Expr\ Integer} \rightarrow \mathit{Integer} & \\
\mathit{eval}\ (\mathit{Vall}\ n) & = n \\
\mathit{eval}\ (x\ `\mathit{Add}`\ y) & = (\mathit{eval}\ x) + (\mathit{eval}\ y) \\
\mathit{eval}\ (\mathit{Var}\ x) & = x \\
\mathit{eval}\ (\mathit{Let}\ e\ f) & = \mathit{eval}\ (f\ (\mathit{eval}\ e))
\end{array}
$$

A közös részkifejezések most már valóban láthatóak (*observable*), értelmezésük mikéntje is a mienk:

$$
\begin{array}{l}
\mathit{toString} :: \mathit{Expr\ String} \rightarrow \mathit{Int} \rightarrow \mathit{String} \\
\mathit{toString}\ (\mathit{Val}\ x)\ \_ = \mathit{show}\ x \\
\mathit{toString}\ (x\ `\mathit{Add}`\ y)\ c = \text{"("} \mathbin{+\!\!+} \mathit{toString}\ x\ c \mathbin{+\!\!+} \text{" + "} \mathbin{+\!\!+} \mathit{toString}\ y\ c \mathbin{+\!\!+} \text{")"} \\
\mathit{toString}\ (\mathit{Var}\ x)\ \_ = x \\
\mathit{toString}\ (\mathit{Let}\ e\ f)\ c = \\
\quad \textbf{let}\ v = \text{"x"} \mathbin{+\!\!+} \mathit{show}\ c \\
\quad \textbf{in}\ \text{"let"} \mathbin{+\!\!+} v \mathbin{+\!\!+} \text{" = "} \mathbin{+\!\!+} \mathit{toString}\ e\ (c + 1) \mathbin{+\!\!+} \\
\qquad \text{" in"} \mathbin{+\!\!+} \mathit{toString}\ (f\ v)\ (c + 1)
\end{array}
$$

```
GHCi> eval (tree 2)
4
GHCi> toString (tree 2) 0
"let x0 = let x1 = 1 in (x1 + x1) in (x0 + x0)"
```

Közös részkifejezések: Observable sharing

De használható maga a Haskell *let*?

- Lehet olyan függvényt készíteni, amellyel a részkifejezések Haskellben megjelenő megosztását lehet felismerni. (*"observable sharing"*)
- Ez viszont mellékhatásos, az eredmény típusa az *IO* típusba csomagolható.
- Az így megfigyelt közös részkifejezésekkel aztán tudunk építeni egy olyan adattípust, ahol szintén explicit *Let* szerepel.
- Bővebben:

  `http://hackage.haskell.org/package/data-reify`

Vissza a típusokhoz: (Függő?) Típusozott *printf*()

```
{-# LANGUAGE GADTs, KindSignatures, TypeOperators #-}
data (:→) :: ⋆ → ⋆ → ⋆ where
  Lit   :: String → (α :→ α)
  Int   :: α :→ (Int → α)
  Chr   :: α :→ (Char → α)
  (:+)  :: (β :→ γ) → (α :→ β) → (α :→ γ)
infixl 5 :+

intp :: (α :→ β) → (String → α) → β
intp (Lit str) c = c str
intp Int       c = λ x . c (show x)
intp Chr       c = λ x . c [x]
intp (a :+ b) c = intp a (λ sa . intp b (λ sb . c (sa ++ sb)))
format :: (String :→ α) → α
format fmts = intp fmts id

tp1 = format (Lit "Hello world")
tp2 = format (Lit "Hello " :+ Lit "world" :+ Chr) '!'
tp3 = format (Lit "The value of " :+ Chr :+ Lit " is " :+ Int) 'x' 3
tp4 = format (Lit "abc" :+ Int :+ Lit "cde") 5
```

Folytatása következik...

Funkcionális előnyök: Magasabb rendű függvények

A sekély beágyazás alapja a magasabb rendű függvények alkalmazása.

$$
\begin{aligned}
&\textbf{type}\ \mathit{Identifier} = \mathit{String}\\
&\textbf{type}\ \mathit{Env} = [(\mathit{Identifier},\ \mathit{Integer})]\\
&\textbf{type}\ \mathit{Expr} = \mathit{Env} \rightarrow \mathit{Integer}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{val} :: \mathit{Integer} \rightarrow \mathit{Expr}\\
&\mathit{var} :: \mathit{Identifier} \rightarrow \mathit{Expr}\\
&\mathit{bnd} :: (\mathit{Identifier},\ \mathit{Expr}) \rightarrow \mathit{Expr} \rightarrow \mathit{Expr}\\
&\mathit{add} :: \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr}\\
&\mathit{eval} :: \mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Integer}
\end{aligned}
$$

$$
\begin{aligned}
x_0 &= (\mathit{bnd}\ (\text{"}x\text{"},\ \mathit{val}\ 3)\ (\mathit{var}\ \text{"}x\text{"}))\ []\\
x_1 &= (\mathit{bnd}\ (\text{"}x\text{"},\ \mathit{val}\ 3)\ (\mathit{add}\ (\mathit{var}\ \text{"}x\text{"})\ (\mathit{var}\ \text{"}x\text{"})))\ []
\end{aligned}
$$

Előnyök: Magasabb rendű függvények (példa)

$$
\begin{aligned}
&\textbf{type}\ \mathit{Identifier} = \mathit{String}\\
&\textbf{type}\ \mathit{Env} = [(\mathit{Identifier},\ \mathit{Integer})]\\
&\textbf{type}\ \mathit{Expr} = \mathit{Env} \rightarrow \mathit{Integer}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{val} :: \mathit{Integer} \rightarrow \mathit{Expr}\\
&\mathit{val} = \mathit{const}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{var} :: \mathit{Identifier} \rightarrow \mathit{Expr}\\
&\mathit{var}\ n = \mathit{fromJust} \circ \mathit{lookup}\ n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{bnd} :: (\mathit{Identifier},\ \mathit{Expr}) \rightarrow \mathit{Expr} \rightarrow \mathit{Expr}\\
&\mathit{bnd}\ (n, x_1)\ x_2 = \lambda\, e\, .\, x_2\ ((n, x_1\ e) : e)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{add} :: \mathit{Expr} \rightarrow \mathit{Expr} \rightarrow \mathit{Expr}\\
&\mathit{add}\ x_1\ x_2 = \lambda\, e\, .\, (x_1\ e) + (x_2\ e)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr} \rightarrow \mathit{Env} \rightarrow \mathit{Integer}\\
&\mathit{eval} = \mathit{id}
\end{aligned}
$$

$$
\begin{aligned}
&x_0 = (\mathit{bnd}\ (\text{"}x\text{"},\ \mathit{val}\ 3)\ (\mathit{var}\ \text{"}x\text{"}))\ []\\
&x_1 = (\mathit{bnd}\ (\text{"}x\text{"},\ \mathit{val}\ 3)\ (\mathit{add}\ (\mathit{var}\ \text{"}x\text{"})\ (\mathit{var}\ \text{"}x\text{"})))\ []
\end{aligned}
$$

Funkcionális előnyök: Lusta kiértékelés

Lusta kiértékelés:

- A függvények paramétereit csak akkor értékeljük, amikor ténylegesen szükség van rájuk („igény szerint”) – vagy egyáltalán nem, pl. a sekély beágyazás különböző értelmezései.
- Mikor kiértékelünk valamit, megtartjuk az értékét és újra felhasználjuk.
- Véges és végtelen adatszerkezetek alkalmazása is lehetséges az értelmezés során.
- Egyszerűbb programokat tudunk így készíteni.
- A beágyazott nyelv kiértékelésének miként akár teljesen független is lehet (vagyis például mohó nyelvet írhatunk le így).

Funkcionális előnyök: Lusta kiértékelés

A ”Do not Repeat Yourself” (DRY) elv, modularitás hatékonyabb támogatása.

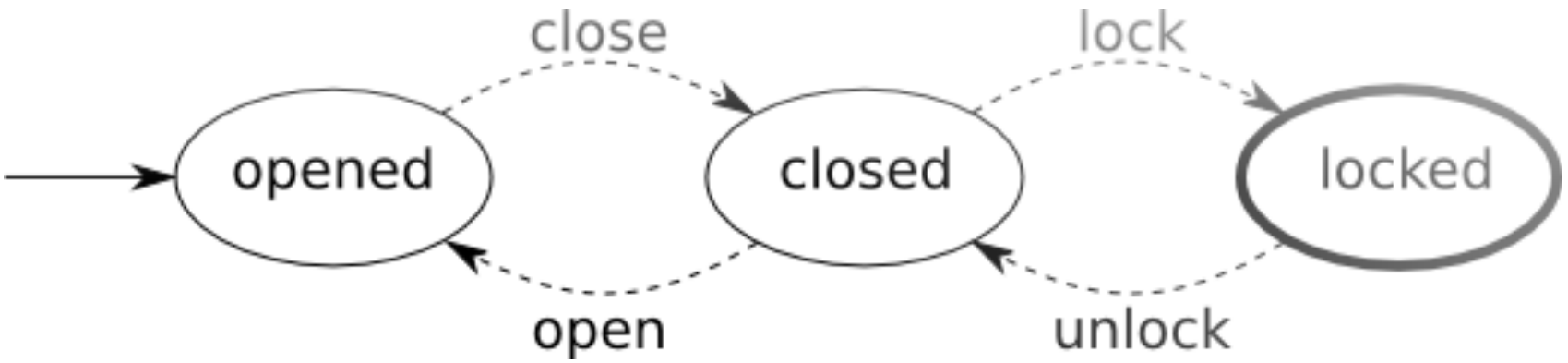
- A gyakori sémák könnyen kiemelhetőek, újrahasznosíthatóak, alkalmasak a vezérlési szerkezetek leírására.
- A mohó nyelvekben a sémák alkalmazása nem minden esetben éri meg, mivel a kiértékelés miatt azok *nem olvadnak össze*.

$$
\begin{aligned}
&\mathit{any} :: (\alpha \to \mathit{Bool}) \to [\alpha] \to \mathit{Bool} \\
&\mathit{any}\ p = \mathit{or} \circ \mathit{map}\ p
\end{aligned}
\qquad
\begin{aligned}
&\mathit{or} :: [\mathit{Bool}] \to \mathit{Bool} \\
&\mathit{or} = \mathit{foldr}\ \mathit{False}\ (\vee)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{any} :: (\alpha \to \mathit{Bool}) \to [\alpha] \to \mathit{Bool} \\
&\mathit{any}\ p = \mathit{foldr}\ \mathit{False}\ (\lambda\, x\ r\ .\ p\ x \vee r)
\end{aligned}
\qquad
\begin{aligned}
&\mathit{any} :: (\alpha \to \mathit{Bool}) \to [\alpha] \to \mathit{Bool} \\
&\mathit{any}\ p\ [\,] = \mathit{False} \\
&\mathit{any}\ p\ (y : \mathit{ys}) = p\ y \vee \mathit{any}\ p\ \mathit{ys}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{foldr} :: (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\
&\mathit{foldr}\ f\ z\ [\,] = z \\
&\mathit{foldr}\ f\ z\ (x : \mathit{xs}) = f\ x\ (\mathit{foldr}\ f\ z\ \mathit{xs})
\end{aligned}
$$

Funkcionális előnyök: Lusta kiértékelés (példa)



type *Door*

$$
\begin{array}{lll}
\mathit{initial} & :: & \mathit{Door} \\
\mathit{react} & :: & \mathit{Door} \rightarrow \mathit{Event} \rightarrow \mathit{Door} \\
\mathit{isLocked} & :: & \mathit{Door} \rightarrow \mathit{Bool}
\end{array}
$$

Funkcionális előnyök: Lusta kiértékelés (példa)

```
data State = Opened | Closed | Locked deriving Eq
data Event = Open | Close | Lock | Unlock

type Door = State

initial :: Door
initial = Opened

react :: Door → Event → Door
react s e = case (s, e) of
  (Closed, Open)   → Opened
  (Opened, Close)  → Closed
  (Closed, Lock)   → Locked
  (Locked, Unlock) → Closed
  _                → s

isLocked :: Door → Bool
isLocked Locked = True
isLocked _      = False
```

Funkcionális előnyök: Lusta kiértékelés (példa)

```
data Tree  = Node State (Event → Door)
type Door  = Tree

initial :: Door
initial = fetch trees Opened where
  trees = [construct Opened, construct Closed, construct Locked]
  construct s = Node s (fetch trees ∘ f s)
  fetch t s = head [ x | x@(Node s1 _) ← t, s1 ≡ s ]

  f s e = case (s, e) of
    (Closed, Open)   → Opened
    (Opened, Close)  → Closed
    (Closed, Lock)   → Locked
    (Locked, Unlock) → Closed
    _                → s

react :: Door → Event → Door
react (Node _ f) e = f e

isLocked :: Door → Bool
isLocked (Node Locked _) = True
isLocked _ = False
```

Beágyazás másként: Template Haskell

Template Haskell: Bevezetés

```
{-# LANGUAGE TemplateHaskell #-}
module Fibs where

import Language.Haskell.TH

fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

-- [| ... |]: Oxford bracket, quasi quote, „template” (Q monád)
-- $x, $(...): splice, "evaluate at compile time"
fibQ :: Int → Q Exp
fibQ n = [| fibs !! n |]
```

```
GHCi> :set -XTemplateHaskell
GHCi> :load Fibs.hs
...
GHCi> $(fibQ 22)
17711
```

Template Haskell: A típusozott *printf*() visszatér

```
{-# LANGUAGE TemplateHaskell #-}
module Printf (printf) where

import Language.Haskell.TH
import Language.Haskell.TH.Syntax

data Format = D | S | L String

-- parse "%d is %s" →*β [D, L " is ", S]
parse :: String → [Format]

gen :: [Format] → Q Exp → Q Exp
gen [] x          = x
gen (D : xs) x    = [| λ n . $(gen xs [| $x ++ show n |]) |]
gen (S : xs) x    = [| λ s . $(gen xs [| $x ++ s |]) |]
gen (L s : xs) x  = gen xs [| $x ++ $(lift s) |]

printf :: String → Q Exp
printf s = gen (parse s) [| "" |]
```

```
GHCi> :load Printf.hs
GHCi> $(printf "Error: %s on line %d.") "unexpected" 42
"Error: unexpected on line 42."
```

Template Haskell: Mit lehet még csinálni?

- *Feltételes fordítás:* A különféle előfeldolgozási megoldások helyett egy megbízhatóbb, típusozott alternatíva.
- *Önreflexió:* A programnak lehetősége van a saját belső szerkezetének elérésére, elemzésére.
- *Procedurális programszerkesztés:* Olyan metaprogramok írásának támogatása, amelyek azt adják meg, miként kell egy nála egyszerűbb programot előállítani, pl. *printf*.
- *Továbbfejlesztett absztrakciók:* Magasabb rendű függvények leképezése elsőrendű függvényekre, vagy egész számokkal indexelt függvények ($zip1$, $zip2$, ..., $zipN$) előállítása típusosan.
- *Optimalizációk, kiterjesztések:* Kísérleti kiterjesztések, problémafüggő optimalizációk megvalósítása a fordítóprogram módosítása nélkül.

Template Haskell: Függvényépítő függvények (példa)

$$
\begin{aligned}
&\mathit{sel} :: \mathit{Int} \to \mathit{Int} \to \mathit{Q\ Exp} \\
&\mathit{sel}\ i\ n = [\!|\ \lambda\, x\ .\ \$(\mathit{caseE}\ [\!|\ x\ |\!]\ [\mathit{alt}])\ |\!]\ \textbf{where} \\
&\quad \mathit{alt} :: \mathit{Q\ Match} \\
&\quad \mathit{alt} = \mathit{match}\ \mathit{pat}\ (\mathit{normalB}\ \mathit{rhs})\ [] \\
&\quad \\
&\quad \mathit{pat} :: \mathit{Q\ Pat} \\
&\quad \mathit{pat} = \mathit{tupP}\ [\ \mathit{varP}\ a \mid a \leftarrow \mathit{as}\ ] \\
&\quad \\
&\quad \mathit{rhs} :: \mathit{Q\ Exp} \\
&\quad \mathit{rhs} = \mathit{varE}\ (\mathit{as}\ !!\ (i - 1)) \\
&\quad \\
&\quad \mathit{as} :: [\mathit{Name}] \\
&\quad \mathit{as} = [\ \mathit{mkName}\ (\text{"}a\text{"} \mathbin{+\!\!+} \mathit{show}\ i) \mid i \leftarrow [1..n]\ ]
\end{aligned}
$$

$$
\begin{aligned}
&\$(\mathit{sel}\ 2\ 3) \longrightarrow \\
&\quad \lambda\, x\ .\ \textbf{case}\ x\ \textbf{of} \\
&\qquad (a_1, a_2, a_3) \to a_2
\end{aligned}
\qquad\qquad
\begin{aligned}
&\$(\mathit{sel}\ 3\ 4) \longrightarrow \\
&\quad \lambda\, x\ .\ \textbf{case}\ x\ \textbf{of} \\
&\qquad (a_1, a_2, a_3, a_4) \to a_3
\end{aligned}
$$

```
GHCi> let sel_3_4 = $(sel 3 4)
GHCi> :type sel_3_4
sel_3_4 :: (t, t1, t2, t3) -> t2
GHCi> sel_3_4 ('a','b','c','d')
'c'
```

Template Haskell: Több függvény gyártása (példa)

```
genSels :: Int → Q [Dec]
genSels n = declare [ s j i | i ← [2..n], j ← [1..i] ] where
  declare [] = return []
  declare (d : ds) = do
    x  ← d
    xs ← declare ds
    return (x ++ xs)

  s x y = do
    let name = mkName $ "sel" ++ show x ++ "" ++ show y
    body ← sel x y
    return [FunD name [Clause [] (NormalB body) []]]
```

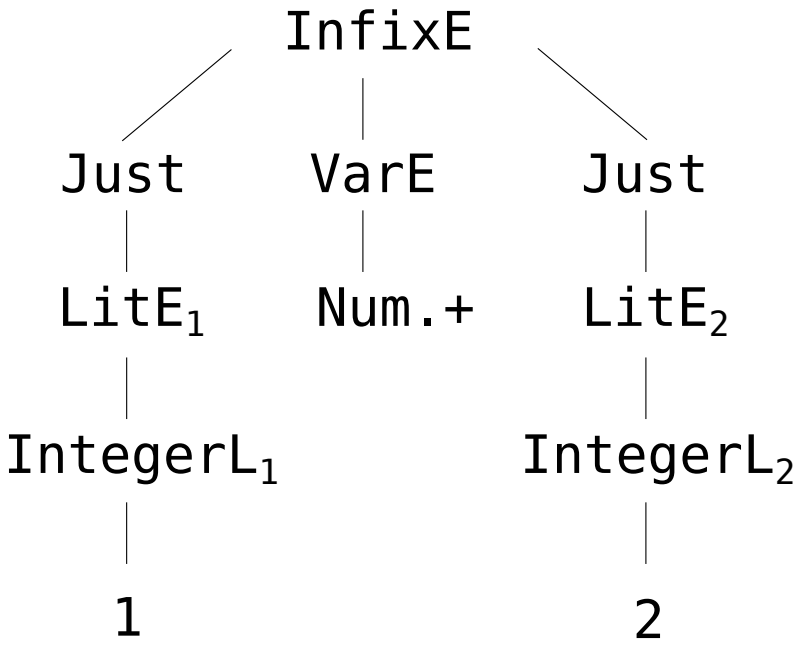
Sels.hs (külön modul):

```
import Language.Haskell.TH
import Sel

$(genSels 10)
```

A Quotation (Q) monádról

```
GHCi> import Language.Haskell.TH.Syntax
GHCi> import Language.Haskell.TH.Lib
GHCi> :type runQ
runQ :: Quasi m => Q a -> m a
GHCi> runQ [| 1 + 2 |]
InfixE
  (Just (LitE (IntegerL 1)))
  (VarE GHC.Num.+)
  (Just (LitE (IntegerL 2)))
GHCi> :{
  $(return (InfixE
    (Just (LitE (IntegerL 1)))
    (VarE (mkName "+"))
    (Just (LitE (IntegerL 2)))
  ))
:}
3
GHCi> :type [| 1 + 2|]
[| 1 + 2 |] :: Q Exp
GHCi> :type [d| x = 5 |]
[d| x = 5 |] :: Q [Dec]
GHCi> :type [t| Int |]
[t| Int |] :: Q Type
GHCi> :type [p| (x,y) |]
[p| (x,y) |] :: Q Pat
```



Quasi Quotes

```
{-# LANGUAGE QuasiQuotes #-}
data E = EInt Int | EAdd E E | EMetaVar String | ...

pExp = spaced pTerm `chainl1` spaced op where
  pTerm     = pNum <|> pMetaVar
  pNum      = do { x ← digit; return (EInt $ digitToInt x) }
  pMetaVar  = do { char '$'; v ← ident; return (EMetaVar v) }
  ident     = do { c ← small; cs ← many idchar; return (c : cs) }
  op        = do { char '+'; return EAdd }
  ...
expr = QuasiQuoter
  { quoteExp = quoteEExpr, quotePat = quoteEPat
  , quoteDec = undefined, quoteType = undefined } where
  quoteEExpr s = do
      pos ← getPosition
      exp ← parseE pos s
      dataToExpQ (const Nothing) exp
  ...
```

Külön modul:

```
simplify :: E → E
simplify [expr| 0 + $x |] = x

main = print $ simplify [expr| 0 + 2 |]
```

Template Haskell: Hibakeresés és önreflexió

```
GHCi> runQ [| 1 + 2 |] >>= putStrLn . pprint
1 GHC.Num.+ 2
GHCi> :set -ddump-splices
GHCi> $([| 1 + 2 |])
<interactive>:11:3-13: Splicing expression
    [| 1 + 2 |] ======> (1 + 2)
3
GHCi> :type id
id :: a -> a
GHCi> $(stringE . show =<< reify 'id)
...
"VarI GHC.Base.id
  (ForallT [PlainTV a_1627399870] []
    (AppT (AppT ArrowT (VarT a_1627399870)) (VarT a_1627399870)))
      Nothing
      (Fixity 9 InfixL)"
GHCi> $(stringE . show =<< reify ''Bool)
...
"TyConI (DataD [] GHC.Types.Bool []
  [NormalC GHC.Types.False [],NormalC GHC.Types.True []] [])"
```