Hackett

a metaprogrammable Haskell

Alexis King
Northwestern & PLT

Haskell

Haskell

Racket

Haskell

Racket

Haskell

Racket

Hackett!

Haskell deserves a **macro system**.

Macros can benefit
from **Haskell's type system**.

A Short Peek at Hackett

Hackett is a **Haskell**.

Hackett

```
(data Point (Point Integer Integer)
  #:deriving [Eq Show])
```

Haskell

```
data Point = Point Integer Integer
  deriving (Eq, Show)
```

Hackett

```
(case (string-split "," str)
  [(List a b) {Point <$> (from-param a)
                     <*> (from-param b)}]
  [_ (Left {"bad point: " ++ (show str)})])
```

Haskell

```
case stringSplit "," str of
  [a, b] → Point <$> fromParam a
                 <*> fromParam b
  _ → Left ("bad point: " ++ show str)
```

Hackett

```
(main (do (println "Server running on port 8080.")
          (run-server 8080)))
```

Haskell

```
main = do putStrLn "Server running on port 8080."
          runServer 8080
```

Hackett

```
(instance (From-Param Point)
  [from-param (λ [str] ....)])
```

Haskell

```
instance FromParam Point where
  fromParam str = ....
```

Hackett is a **Haskell**.

But…

```
(defserver run-server
  [GET "hello" → String → String ⇒ greet]
  [GET "add" → Integer → "to" → Integer
         → Integer ⇒ +]
  [GET "distance-from" → Point → "to"
         → Point → Double ⇒ distance])
```

```
(defserver run-server
  [GET "hello" → String → String ⇒ greet]
  [GET "add" → Integer → "to" → Integer
        → Integer ⇒ +]
  [GET "distance-from" → Point → "to"
        → Point → Double ⇒ distance])
```

Macros!

```
(defserver run-server
  [GET "hello" → String → String ⇒ greet])
```

⇝

```
(defn run-server : {Integer → (IO Unit)}
  [[port]
   (listen-on-port
    port (λ [req]
           (case (request→path-segments req)
             [(List "hello" a)
              (either→response
               (do [tmp ← (from-param a)]
                   (pure (greet tmp))))]
             [_ (Response 404 "Not Found")]))))])
```

`defserver : AST → AST`

Defined at compile-time!

Macros are **syntactic abstractions**.

12

Cleanly handling pattern matching errors self.haskell

submitted 1 year ago by wildptr

Hi /r/Haskell, I've been working on a small project where the public API is written in terms of the MTL stack `ExceptT ... (StateT ...)`, so I tend to write code that looks a lot like the following.

```
case [expr] of
    [pattern] -> [code]
    _         -> throwError [error]
```

In other words, I pattern match on some expression, and if it's valid, then I perform some computation, or else I throw an error and dump the application state to the user.

Since this shows up frequently, I figure I could write and export a set of functions that implements the above functionality on a set of common patterns (most matches tend to happen on expressions of the type of the application state). But the thing is, you can't write a general function that takes `[expr], [pattern], [code]`, and `[error]` that expands to the above without using some metaprogramming facility like TH.

So, is it worth encoding this pattern into a set of functions or just leave the user to handle extraneous patterns themselves?

Thanks in advance!

8 comments share save hide report

```
(do [x ← (case a
              [(Foo x) x]
              [_ (throw E1)])]
    [y ← (case (f x)
              [(Bar y) y]
              [_ (throw E2)])]
    [z ← (case (g y)
              [(Qux z) z]
              [_ (throw E3)])]
    (pure (h z)))
```

```
(do [x ← (case a
              [(Foo x) x]
              [_ (throw E1)])]
    [y ← (case (f x)
              [(Bar y) y]
              [_ (throw E2)])]
    [z ← (case (g y)
              [(Qux z) z]
              [_ (throw E3)])]
    (pure (h z)))
```

```
(case/throw [(Foo x) ← a     #:or E1]
            [(Bar y) ← (f x) #:or E2]
            [(Qux z) ← (g y) #:or E3]
            (pure (h z)))
```

```
(define-syntax-parser case/throw
  #:datum-literals [←]
  [(_ e) #'e]
  [(_ [(pat x) ← val #:or err]
      more ...+)
   #'(do [x ← (case val
                 [(pat x) x]
                 [_ (throw err)])]
        (case/throw more ...))])
```

```
(defn analyze-expr : {Bool → Expr → Expr}
  [[flag (Var v)]
   (if flag (change/1 v) (change/2 v))]
  [[flag (App e1 e2)]
   (App (analyze-expr flag e1)
        (analyze-expr flag e2))]
  [[flag (Lam v e)]
   (Lam v (analyze-expr flag e))]
  [[flag (Lam v e)]
   (Lam v (analyze-expr flag e))]
  [[flag (Case scrut alts)]
   (Case (analyze-expr flag scrut)
         {(analyze-alt flag) <$> alts})])

(defn analyze-alt : {Bool → Alt → Alt}
  [[flag (Alt dc pats e)]
   (Alt dc pats (analyze-expr flag e))])
```

```
(defn analyze-expr : {Bool → Expr → Expr}
  [[flag (Var v)]
   (if flag (change/1 v) (change/2 v))]
  [[flag (App e1 e2)]
   (App (analyze-expr flag e1)
        (analyze-expr flag e2))]
  [[flag (Lam v e)]
   (Lam v (analyze-expr flag e))]
  [[flag (Lam v e)]
   (Lam v (analyze-expr flag e))]
  [[flag (Case scrut alts)]
   (Case (analyze-expr flag scrut)
         {(analyze-alt flag) <$> alts})])

(defn analyze-alt : {Bool → Alt → Alt}
  [[flag (Alt dc pats e)]
   (Alt dc pats (analyze-expr flag e))])
```

```
(defn analyze-expr : {Bool → Expr → Expr}
  [[flag (Var v)]
   (if flag (change/1 v) (change/2 v))]
  [[flag (App e1 e2)]
   (App (analyze-expr flag e1)
        (analyze-expr flag e2))]
  [[flag (Lam v e)]
   (Lam v (analyze-expr flag e))]
  [[flag (Lam v e)]
   (Lam v (analyze-expr flag e))]
  [[flag (Case scrut alts)]
   (Case (analyze-expr flag scrut)
         {(analyze-alt flag) <$> alts})])

(defn analyze-alt : {Bool → Alt → Alt}
  [[flag (Alt dc pats e)]
   (Alt dc pats (analyze-expr flag e))])
```

```
(defn analyze-expr : {Bool → Expr → Expr}
  [[flag (Var v)]
   (if flag (change/1 v) (change/2 v))]
  [[flag (App e1 e2)]
   (App (analyze-expr flag e1)
        (analyze-expr flag e2))]
  [[flag (Lam v e)]
   (Lam v (analyze-expr flag e))]
  [[flag (Lam v e)]
   (Lam v (analyze-expr flag e))]
  [[flag (Case scrut alts)]
   (Case (analyze-expr flag scrut)
         {(analyze-alt flag) <$> alts})])

(defn analyze-alt : {Bool → Alt → Alt}
  [[flag (Alt dc pats e)]
   (Alt dc pats (analyze-expr flag e))])
```

```
(section ([flag : Bool])
  (defn analyze-expr : {Expr → Expr}
    [[(Var v)]
     (if flag (change/1 v) (change/2 v))]
    [[(App e1 e2)]
     (App (analyze-expr e1)
          (analyze-expr e2))]
    [[(Lam v e)]
     (Lam v (analyze-expr e))]
    [[(Lam v e)]
     (Lam v (analyze-expr e))]
    [[(Case scrut alts)]
     (Case (analyze-expr scrut)
           {analyze-alt <$> alts})])

  (defn analyze-alt : {Alt → Alt}
    [[(Alt dc pats e)]
     (Alt dc pats (analyze-expr e))]))
```

[–] **rampion** **9 points** 10 months ago

You can still do this trick for multiple functions at the same time:

```
(analyseExpr, analyseAlt) = distribute section where
  section :: Flag -> (Expr -> Expr, Alt -> Alt)
  section flag = (analyseExpr, analyseAlt) where

    analyseExpr :: Expr -> Expr
    analyseExpr (Var v) = if flag then change1 v else change2 v
    analyseExpr (App e1 e2) =
      App (analyseExpr e1) (analyseExpr e2)
    analyseExpr (Lam v e) = Lam v (analyseExpr e)
    analyseExpr (Case scrut alts) =
      Case (analyseExpr scrut) (analyseAlt <$> alts)

    analyseAlt :: Alt -> Alt
    analyseAlt (dc, pats, e) = (dc, pats, analyseExpr e)
```

where you can define `distribute` simply as:

```
distribute :: Functor f => f (a,b) -> (f a, f b)
distribute f = (fst <$> f, snd <$> f)
```

[–] nomeata [S] 4 points 10 months ago

Nice implementation of this. If we now only had syntax to get rid of the boiler plate code in your first three lines… :-)

permalink embed save parent give gold

```
(define-syntax-parser section
  #:literals [: defn]
  [(_ ([x:id : t] ...)
      {~and d (defn d-id:id _ ...)} ...)
   #'(def (Tuple d-id ...)
       (let ([f (λ [x ...]
                   (local [d ...]
                     (Tuple d-id ...)))])
         (Tuple
          (λ [x ...]
            (case (f x ...)
              [(Tuple d-id ...) d-id]))
          ...)))])
```

Macros are **syntactic abstractions**.

Macros give us **better DSLs**.

Macros help **eliminate boilerplate**.

Haskell programmers **want macros**.

Hackett gives macros to Haskell.

Clearly, Haskell can benefit from macros.

How can macros benefit from Haskell?

Haskell is already
fantastic at metaprogramming!

```haskell
boldP :: Parser String
boldP = do
  count 2 (char '*')
  txt ← some (alphaNumChar <|> char ' ')
  count 2 (char '*')
  return $ concat
    [ "<strong>", txt, "</strong>" ]
```

parsing — megaparsec

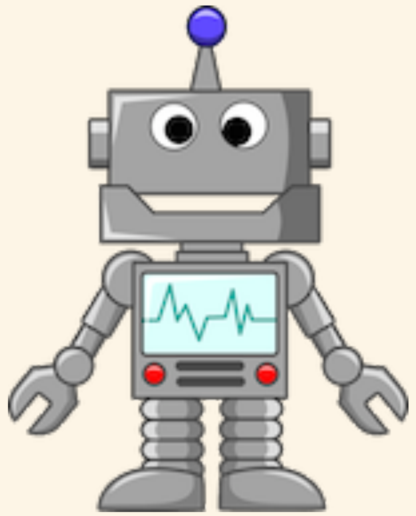
```haskell
fileInput :: Parser Input
fileInput = FileInput <$> strOption
  (  long "file"
  <> short 'f'
  <> metavar "FILENAME"
  <> help "Input file" )

stdInput :: Parser Input
stdInput = flag' StdInput
  (  long "stdin"
  <> help "Read from stdin" )
```

CLIs — `optparse-applicative`

```haskell
table_ [rows_ "2"]
       (tr_ (do td_ [ class_ "top"
                    , colspan_ "2"
                    , style_ "color:red"]
                    (p_ "Hello, attributes!")
                td_ "yay!"))
```

HTML — `lucid`

servant



```haskell
type API =
  "users" :> Get '[JSON] [(UserId, User)]
  :<|> "users" :> Capture "id" UserId
               :> Get '[JSON] User
```

```haskell
type API =
  "users" :> Get '[JSON] [(UserId, User)]
  :<|> "users" :> Capture "id" UserId
                :> Get '[JSON] User

getUsers :: IO [(UserId, User)]
getUsers = ...

getUser :: UserId → IO User
getUser = ...
```

```haskell
type API =
  "users" :> Get '[JSON] [(UserId, User)]
  :<|> "users" :> Capture "id" UserId
               :> Get '[JSON] User

getUsers :: IO [(UserId, User)]
getUsers = ...

getUser :: UserId → IO User
getUser = ...

main :: IO ()
main = run 8080 $
  serve @API (getUsers :<|> getUser)
```

What's the secret sauce?

Typeclasses

Typeclasses

```
> fmap (+ 1) [1, 2, 3]
[2, 3, 4]
```

```
> fmap (+ 1) (Just 2)
Just 3
```

```
> fmap (+ 1) Nothing
Nothing
```

Typeclasses

```
> empty :: [Integer]
[]

> empty :: Maybe Integer
Nothing
```

$$\texttt{empty} \xrightarrow{\textit{eval}} ???$$

$$\texttt{empty :: Maybe ()} \xrightarrow{eval} \texttt{Nothing}$$

$\texttt{empty :: Maybe ()} \xrightarrow{\textit{eval}} \texttt{Nothing}$

Evaluation of programs *depends on types!*

Type Erasure

Type Erasure

```
type Success = { success: true, value: boolean };
type Failed  = { success: false, error: string };

type Response = Success | Failed;

function handleResponse(response: Response) {
  if (response.success) {
    var value: boolean = response.value;
  } else {
    var error: string = response.error;
  }
}
```

Type Erasure

```
type Success = { success: true, value: boolean };
type Failed  = { success: false, error: string };

type Response = Success | Failed;

function handleResponse(response: Response) {
  if (response.success) {
    var value: boolean = response.value;
  } else {
    var error: string = response.error;
  }
}
```

`empty :: Maybe ()` $\xrightarrow{eval}$ `Nothing`

$\texttt{empty :: Maybe ()} \xrightarrow{\textit{compile}} \texttt{empty}_{\texttt{Maybe}} \xrightarrow{\textit{eval}} \texttt{Nothing}$

$$\texttt{empty :: Maybe ()} \xrightarrow{\textit{compile}} \texttt{empty}_{\texttt{Maybe}} \xrightarrow{\textit{eval}} \texttt{Nothing}$$

Typeclasses are (compile-time) functions
from types to expressions.

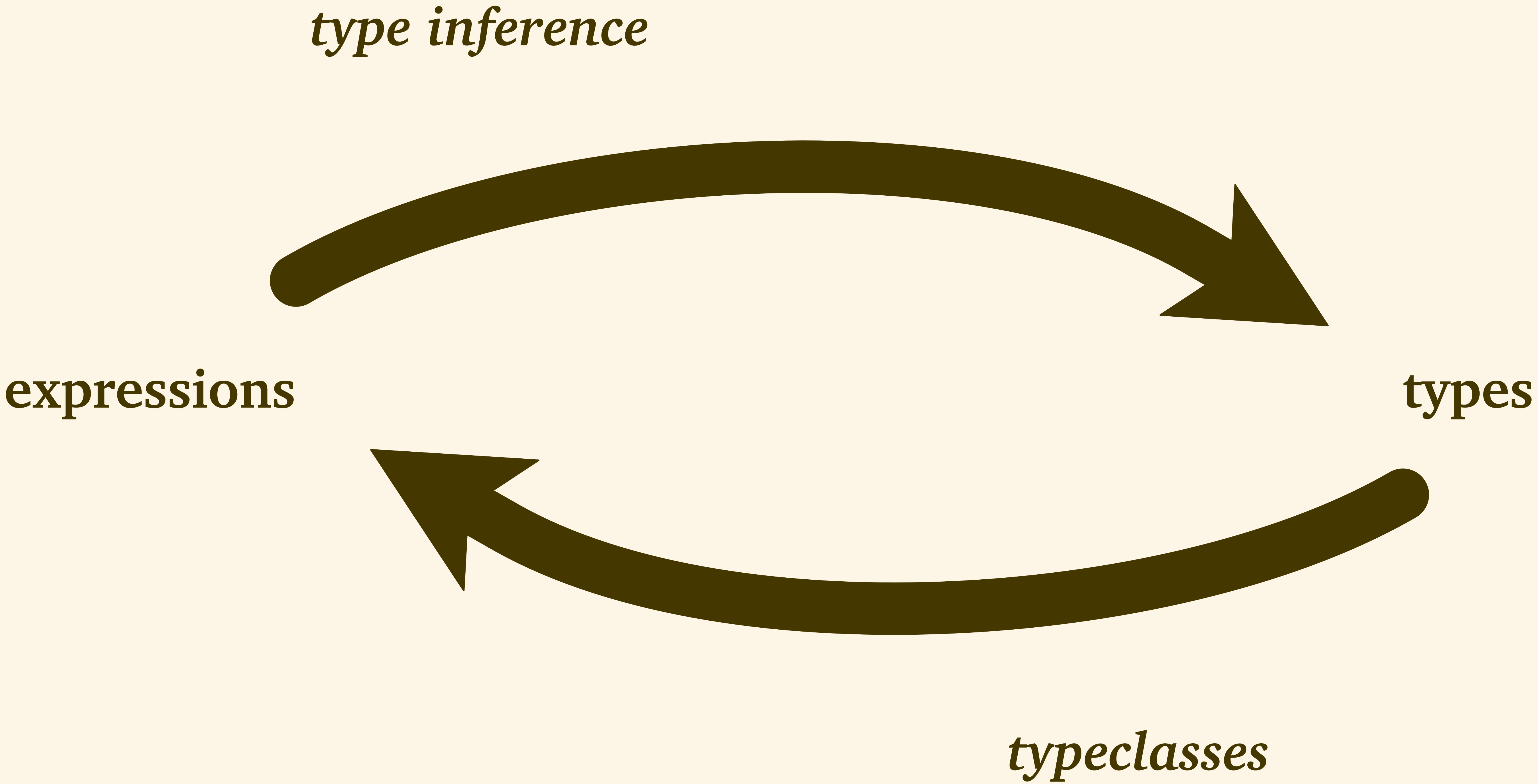
```
> empty :: [Integer]
[]

> empty ++ [1, 2, 3]
[1, 2, 3]
```

expressions

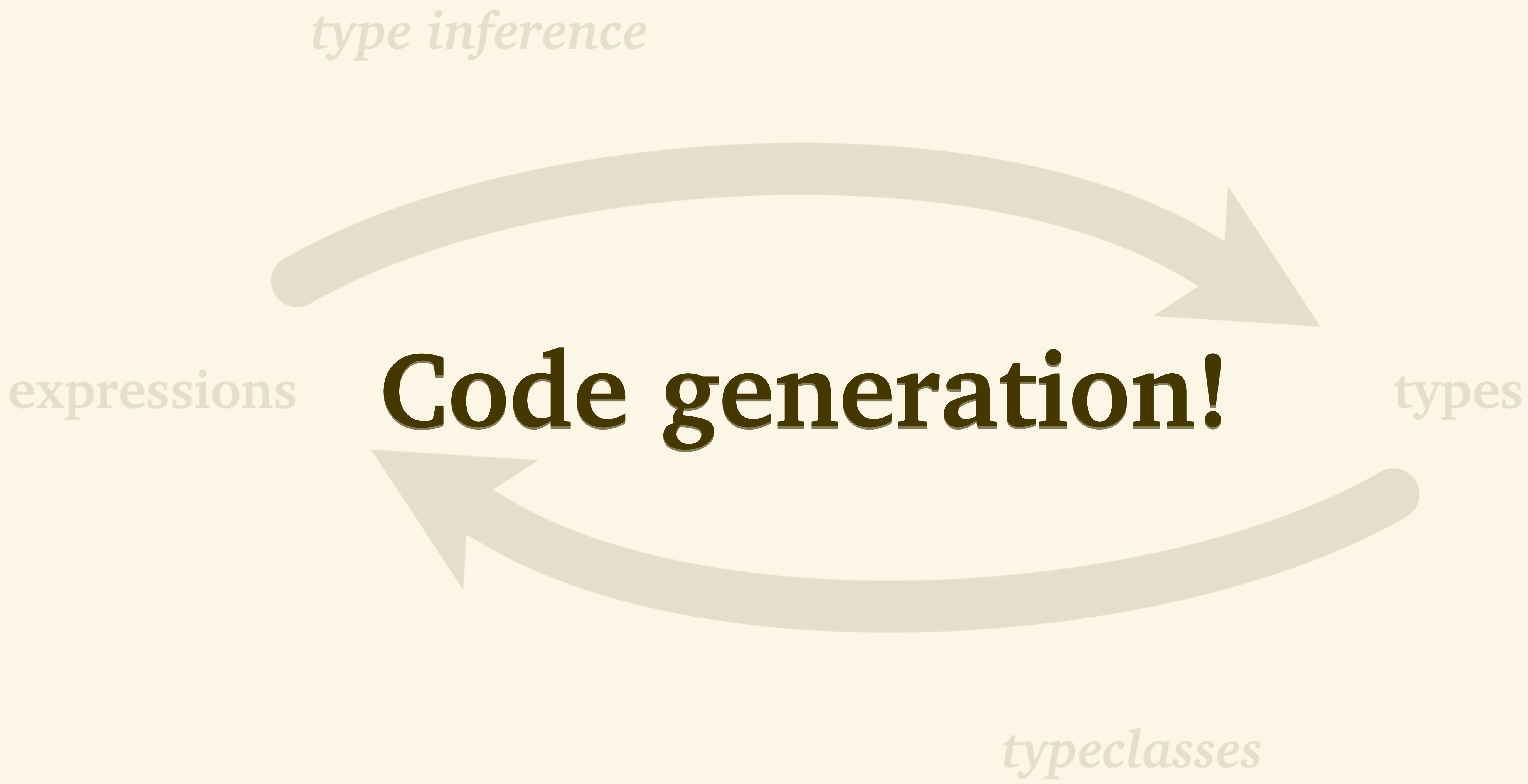
types

type inference

expressions **types**


type inference
expressions
types
typeclasses

Code generation!



```haskell
type API =
  "users" :> Get '[JSON] [(UserId, User)]
  :<|> "users" :> Capture "id" UserId
                 :> Get '[JSON] User

getUsers :: IO [(UserId, User)]
getUsers = ...

getUser :: UserId → IO User
getUser = ...

main :: IO ()
main = run 8080 $
  serve @API (getUsers :<|> getUser)
```

Macro Metaprogramming

vs.

Typeclass Metaprogramming

`(mac` *expr*`)`

```
(mac expr)
```

⇝

```
(let ([foo bar])
  (case (baz expr foo)
    [(List x _) (Just x)]
    [_ Nothing]))
```

```
(mac expr)
```

⇝

```
(let ([foo bar])
  (case (baz expr foo)
    [(List x _) (Just x)]
    [_ Nothing]))
```

```
(def x (List 1 2 3))
```

```
(show x)
```

⇝

```
(show(List Integer) x)
```

⇝

```
((λ [xs] {"(List "
          ++ (string-join
               " "
               (map show xs))
          ++ ")"})
 x)
```

```
{"(List "
 ++ (string-join " " (map show xs))
 ++ ")"}
```

```
{"(List "
 ++ (string-join " " (map show xs))
 ++ ")"}
```

```
{"(List "
 ++ (string-join " " (map show xs))
 ++ ")"}
```

⇝

```
{"(List "
 ++ (string-join " " (map show_Integer xs))
 ++ ")"}
```

```
{"(List "
 ++ (string-join " " (map show xs))
 ++ ")"}
```

⇝

```
{"(List "
 ++ (string-join " " (map show_Integer xs))
 ++ ")"}
```

⇝

```
{"(List "
 ++ (string-join " " (map integer→string xs))
 ++ ")"}
```

Macros excel at *local* code transformations.

Can provide custom syntax.

Typeclasses permit *global* code transformations.

Tethered to the syntax of the host language.

Can we get **both**?

We already have

$$\texttt{mac} : \texttt{AST} \rightarrow \texttt{AST}$$

Can we have

$$\texttt{mac} : \langle \texttt{AST}, \texttt{Type} \rangle \rightarrow \texttt{AST}$$

?

Yes!

(With some caveats.)

compiler

Parse → Compile

compiler + types

Parse → Typecheck → Compile

compiler + macros

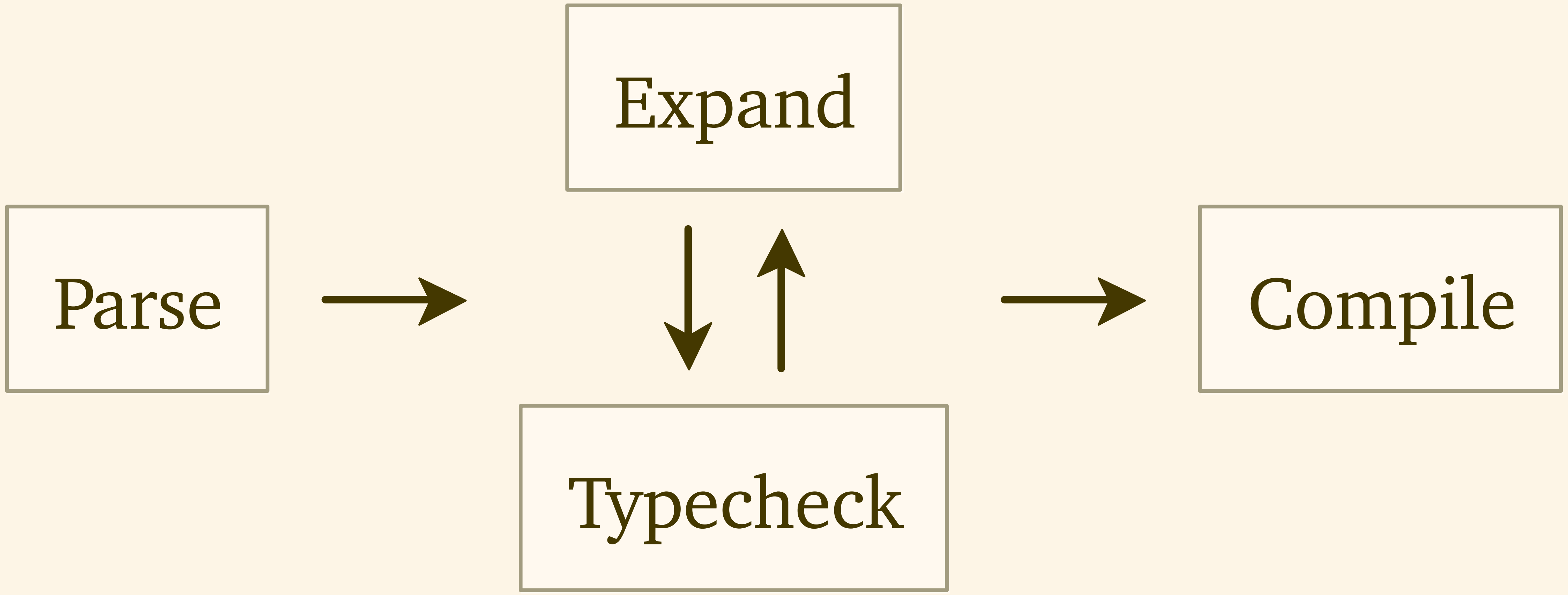
Parse → Expand → Compile

compiler + types + macros

Parse → Expand → Typecheck → Compile

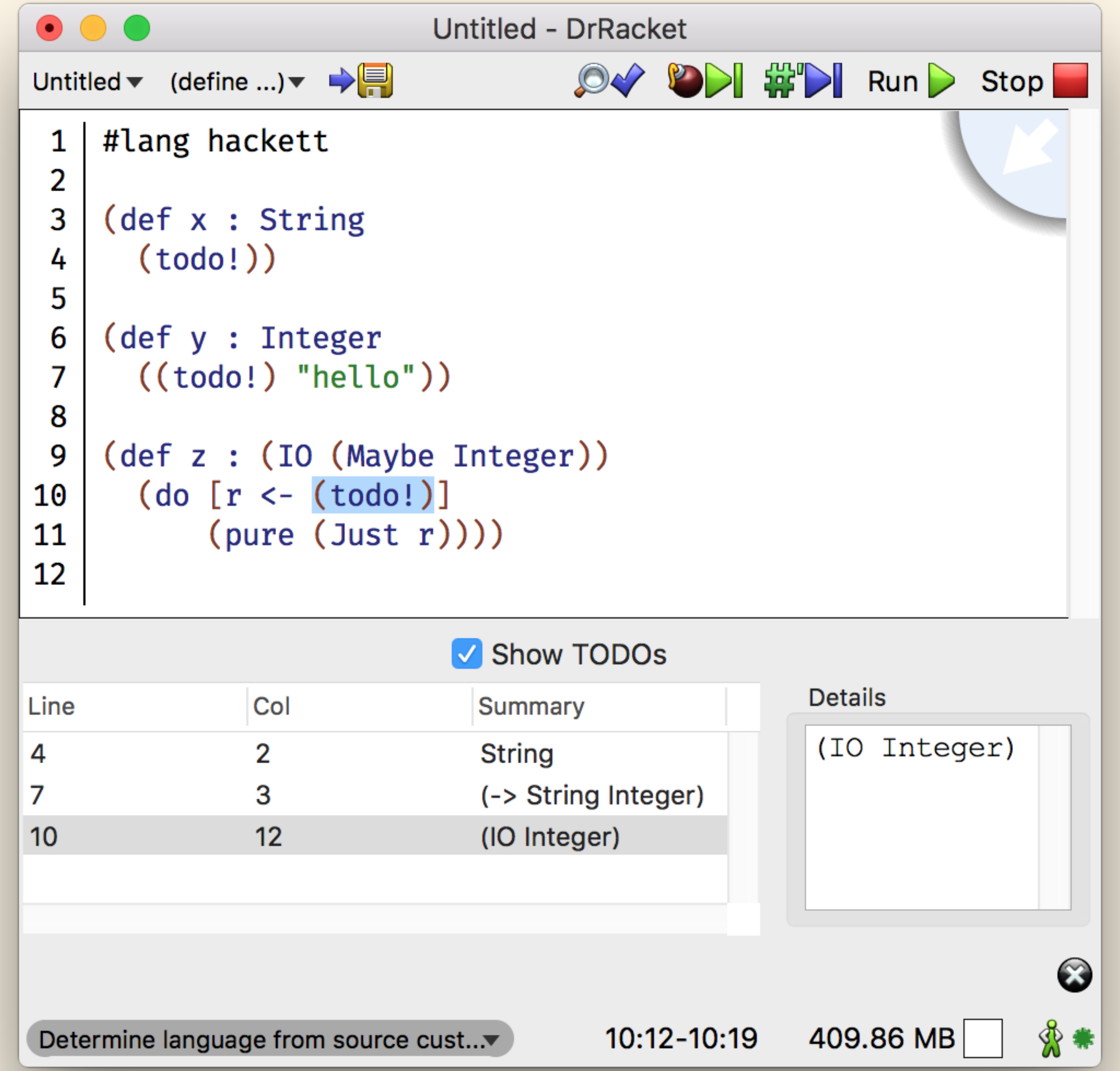
“Lisp-flavored Haskell”

compiler + types + type-aware macros



Hackett is *more than* Lisp-flavored Haskell.

(Thank you, Chang, Knauth, and Greenman
for *Type Systems as Macros*!)


Untitled - DrRacket
Untitled
(define ...)
Run
Stop
#lang hackett
(def x : String
  (todo!))
(def y : Integer
  ((todo!) "hello"))
(def z : (IO (Maybe Integer))
  (do [r <- (todo!)]
      (pure (Just r))))
Show TODOs
Line Col Summary
4 2 String
7 3 (-> String Integer)
10 12 (IO Integer)
Details
(IO Integer)
Determine language from source cust...
10:12-10:19
409.86 MB

```
(define-syntax todo!
  (make-expected-type-transformer
   (syntax-parser
     [(_ e ...)
      (let* ([type-str (type→string
                         (get-expected this-syntax))]
             [message (string-append
                        (source-location→prefix this-syntax)
                        "todo! with type "
                        type-str)])
        (syntax-property
         (quasisyntax/loc this-syntax (error! #,message))
         'todo (todo-item type-str type-str)))])))
```

```
> (inst Nothing Integer)
: (Maybe Integer)
Nothing
```

```
(define-syntax-parser inst
  [(_ e:expr {~type inst-t:type} ...)
   #:do [(define t-count (length (attribute inst-t)))
         (define-values [e- tₑ] (τ⇒! #'e))]
   #:with {~#%type:forall* [x ...] t_mono} tₑ
   #:fail-when (< (length (attribute x)) (length (attribute inst-t)))
   (~a "given " t-count " type(s), but " (type→string tₑ)
       " only has " (length (attribute x)) " type variable(s) available for"
       " instantiation")
   #:do [(define-values [xs_inst xs_keep] (split-at (attribute x) t-count))]
   #:with t_inst (insts #`(?#%type:forall* #,xs_keep t_mono)
                        (map cons xs_inst (attribute inst-t)))
   #:with e+residual #`(let-values ([() inst-t.residual] ...) #,e-)
   (quasisyntax/loc this-syntax
     (: #,(attach-type #'e+residual tₑ) t_inst))])
```

```
> (cata 0 negate (Just 42))
-42

> (cata id show (Right True))
"True"
> (cata id show (Left "bang"))
"bang"

> (cata 0 + (List 1 2 3))
6
```

```
(do [maybe-user
      ← (sql-find
          (and
           {(string-downcase user.email) = email}
           {user.password = password}))]

    (case maybe-user
      [(just user) (set-session/redirect user)]
      [nothing     render-login-403]))
```

```
(do [maybe-user
      ← (sql-find
          (and
           {(string-downcase user.email) = email}
           {user.password = password}))]

    (case maybe-user
      [(just user) (set-session/redirect user)]
      [nothing     render-login-403]))
```

Hackett is a **Haskell** extended with **macros**.

This gives us **language extensibility** and **better DSLs**.

Macros **augment** existing Haskell metaprogramming.

Complementary, providing **local** *and* **global** transformations.

Type-directed macros are available.

Made possible by **interleaving** expansion and typechecking.

Hopefully, much more to come here in the future!

Thanks!

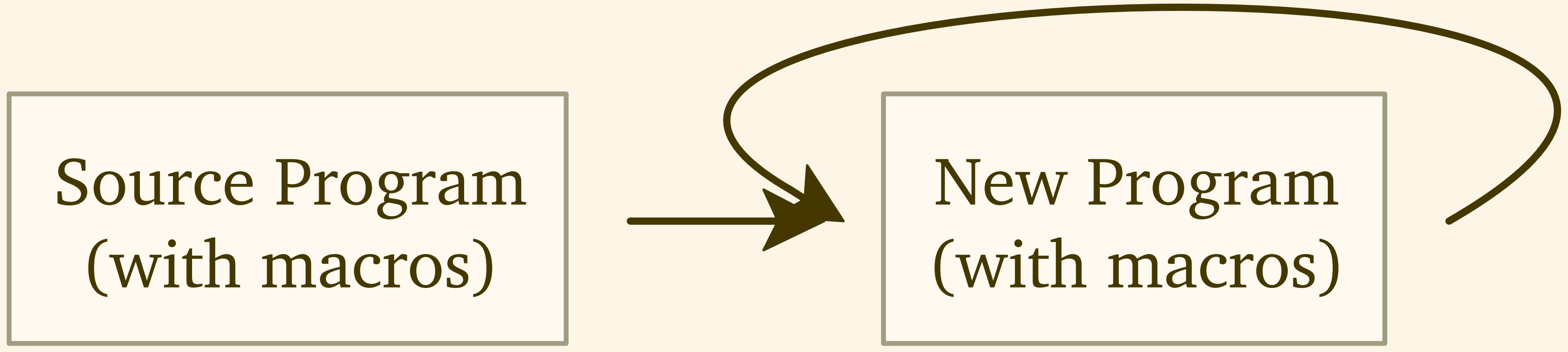
```
(macro-foo a b c)

(macro-bar x y z)
```

Source Program
(with macros)

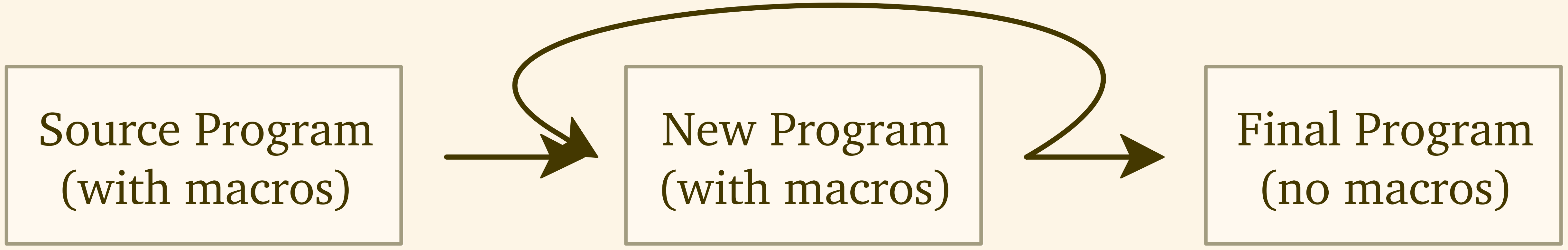
```
(macro-foo a b c)

(macro-bar x y z)
```

Source Program (with macros) → New Program (with macros)

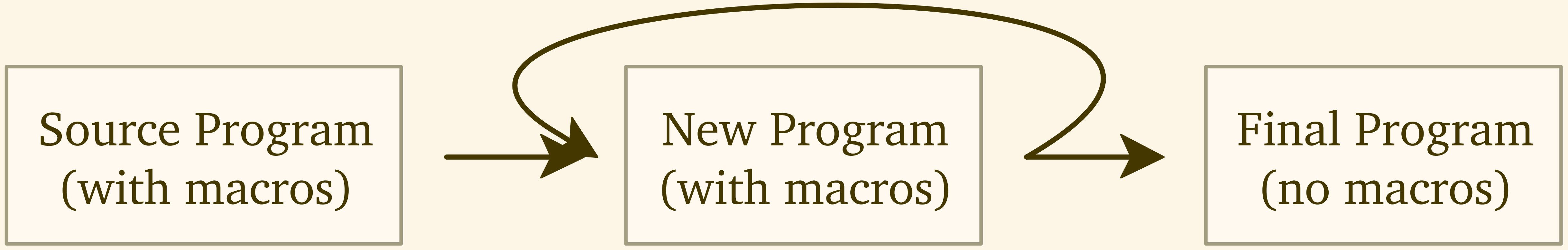
```
(let ([a b]) (macro-qux c))

(macro-bar x y z)
```




```
(let ([a b]) (λ (_) c))

(macro-bar x y z)
```

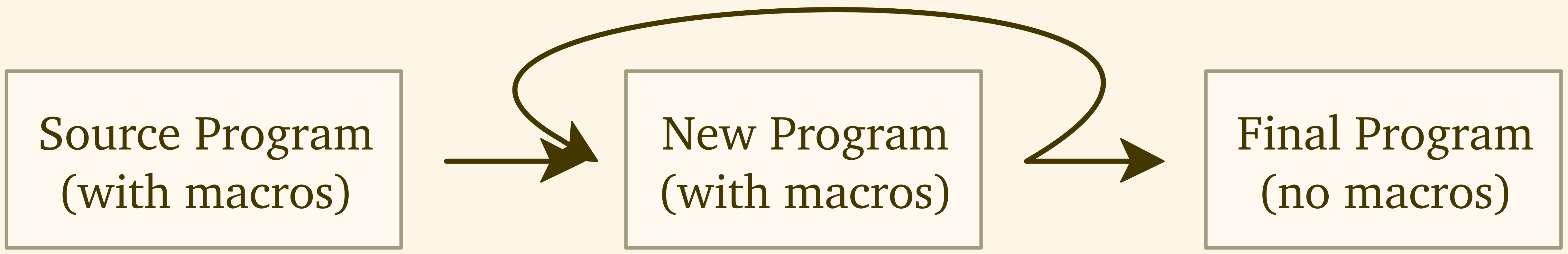



```
(let ([a b]) (λ (_) c))

(case x [y z])
```




```
(let ([a b]) (λ (_) c))

(case x [y z])
```

Runtime error!




```
(macro-foo a b c)

(macro-bar x y z)
```

Runtime error!

When Macros Go Wrong

Errors in expanded code.

```
(let ([a b]) (λ (_) c))
```

Misused macros.

```
(let ([1 2]) 3)
```

Identifier name conflicts.

```
(define-syntax-rule (or x y)
  (let ([tmp x]) (if tmp tmp y)))
(let ([tmp 42]) (or #f (* 2 tmp)))
```

```haskell
data HsExpr p
  = HsVar       (Located (IdP p))
  | HsUnboundVar UnboundVar
  | HsConLikeOut ConLike
  | HsRecFld (AmbiguousFieldOcc p)
  | HsOverLabel (Maybe (IdP p)) FastString
  | HsIPVar   HsIPName
  | HsOverLit (HsOverLit p)
  | HsLit       (HsLit p)
  | HsLam       (MatchGroup p (LHsExpr p))
  | HsLamCase (MatchGroup p (LHsExpr p))
  | HsApp       (LHsExpr p) (LHsExpr p)
  | HsAppType (LHsExpr p) (LHsWcType p)
  | HsAppTypeOut (LHsExpr p) (LHsWcType GhcRn)
  | OpApp         (LHsExpr p)
```

```haskell
data HsExpr p
  = HsVar     (Located (IdP p))
  | HsUnboundVar UnboundVar
  | HsConLikeOut ConLike
  | HsRecFld (AmbiguousFieldOcc p)
  | HsOverLabel (Maybe (IdP p)) FastString
  | HsIPVar   HsIPName
  | HsOverLit (HsOverLit p)
  | HsLit     (HsLit p)
  | HsLam     (MatchGroup p (LHsExpr p))
  | HsLamCase (MatchGroup p (LHsExpr p))
  | HsApp     (LHsExpr p) (LHsExpr p)
  | HsAppType (LHsExpr p) (LHsWcType p)
  | HsAppTypeOut (LHsExpr p) (LHsWcType GhcRn)
  | OpApp       (LHsExpr p)
                (LHsExpr p)
                (PostRn p Fixity)
                (LHsExpr p)
  | NegApp      (LHsExpr p)
                (SyntaxExpr p)
  | HsPar       (LHsExpr p)
  | SectionL    (LHsExpr p)
                (LHsExpr p)
  | SectionR    (LHsExpr p)
                (LHsExpr p)
  | ExplicitTuple
        [LHsTupArg p]
        Boxity
  | ExplicitSum
          ConTag
          Arity
          (LHsExpr p)
          (PostTc p [Type])
  | HsCase      (LHsExpr p)
                (MatchGroup p (LHsExpr p))
  | HsIf        (Maybe (SyntaxExpr p))
                (LHsExpr p)
                (LHsExpr p)
                (LHsExpr p)
  | HsMultiIf   (PostTc p Type) [LGRHS p (LHsExpr p)]
  | HsLet       (LHsLocalBinds p)
                (LHsExpr  p)
  | HsDo        (HsStmtContext Name)
                (Located [ExprLStmt p])
                (PostTc p Type)
  | ExplicitList
                (PostTc p Type)
                (Maybe (SyntaxExpr p))
                [LHsExpr p]
  | ExplicitPArr
                (PostTc p Type)
                [LHsExpr p]
  | RecordCon
      { rcon_con_name :: Located (IdP p)
      , rcon_con_like :: PostTc p ConLike
      , rcon_con_expr :: PostTcExpr
      , rcon_flds     :: HsRecordBinds p }
  | RecordUpd
      { rupd_expr :: LHsExpr p
      , rupd_flds :: [LHsRecUpdField p]
      , rupd_cons :: PostTc p [ConLike]
      , rupd_in_tys  :: PostTc p [Type]
      , rupd_out_tys :: PostTc p [Type]
      , rupd_wrap :: PostTc p HsWrapper
      }
  | ExprWithTySig
                (LHsExpr p)
                (LHsSigWcType p)
  | ExprWithTySigOut
                (LHsExpr p)
                (LHsSigWcType GhcRn)
  | ArithSeq
                PostTcExpr
                (Maybe (SyntaxExpr p))
                (ArithSeqInfo p)
  | PArrSeq
                PostTcExpr
                (ArithSeqInfo p)
  | HsSCC       SourceText
                StringLiteral
                (LHsExpr p)
  | HsCoreAnn   SourceText
                StringLiteral
                (LHsExpr p)
  | HsBracket    (HsBracket p)
  | HsRnBracketOut
      (HsBracket GhcRn)
      [PendingRnSplice]
  | HsTcBracketOut
      (HsBracket GhcRn)
      [PendingTcSplice]
  | HsSpliceE  (HsSplice p)
  | HsProc      (LPat p)
                (LHsCmdTop p)
  | HsStatic (PostRn p NameSet)
             (LHsExpr p)
  | HsArrApp
        (LHsExpr p)
        (LHsExpr p)
        (PostTc p Type)
        HsArrAppType
        Bool
  | HsArrForm
        (LHsExpr p)
        (Maybe Fixity)
        [LHsCmdTop p]
  | HsTick
     (Tickish (IdP p))
     (LHsExpr p)
  | HsBinTick
     Int
     Int
     (LHsExpr p)
  | HsTickPragma
     SourceText
     (StringLiteral,(Int,Int),(Int,Int))
     ((SourceText,SourceText),(SourceText,SourceText))
     (LHsExpr p)
  | EWildPat
  | EAsPat      (Located (IdP p))
                (LHsExpr p)
  | EViewPat    (LHsExpr p)
                (LHsExpr p)
  | ELazyPat    (LHsExpr p)
  |  HsWrap     HsWrapper
                (HsExpr p)
```

126 lines!

```
(defserver run-server
  [GET "hello" → String → String ⇒ greet])
```

```
(defserver run-server
  [GET "hello" → String → String ⇒ greet])
```

( ) [ ]

lists

```
(defserver run-server
  [GET "hello" → String → String ⇒ greet])
```

( ) [ ]
lists

x
identifiers

```
(defserver run-server
  [GET "hello" → String → String ⇒ greet])
```

`() []`	`x`	`"foo"`
lists	identifiers	strings

```
(defserver run-server
  [GET "hello" → String → String ⇒ greet])
```

`() []`	`x`	`"foo"`
lists	identifiers	strings

Macros vs. Splices

macro : `AST` → `AST`

splice : `Value …` → `AST`

Macros

```
(defserver run-server
  [GET "hello" → String → String ⇒ greet])
```

Splices

```
$(mkServer "runServer"
           MethodGet
           [ PathSegment "hello"
           , ValSegment 'String
           , ValSegment 'String ]
           ''greet)
```

Splices in Template Haskell

splice : `Value` … → `AST`*

* *may not contain splices!*

GHC Stage Restriction

```haskell
x = 7
$(doSomethingAtCompileTime x)
```

```haskell
someLocalTemplateFunction :: String → Q [Dec]
someLocalTemplateFunction = ...

$(someLocalTemplateFunction "foo")
```

GHC Stage Restriction

```
(begin-for-syntax
  (def x 7))
(do-something-at-compile-time x)
```

```
(define-syntax (some-local-macro stx)
  ...)
(some-local-macro "foo")
```

Expansion Order

```
$(makeSomeDatatype "Foo" ''Bar)
$(makeSomeDatatype "Bar" ''Foo)
```

```
(def-some-datatype Foo Bar)
(def-some-datatype Bar Foo)
```

Template Haskell	Hackett
splices	macros
stage restriction	flexible phase system
declaration groups	mutually recursive definitions
pseudo-hygiene	full hygiene
procedural only	pattern-based macros
limited reflection	type-directed macros