Exercises, Week 47 (17–22 Nov, 2025)

DM580: Functional Programming, SDU

Learning Objectives

After doing these exercises, you will be able to:

- Use `cabal` to create and manage a Haskell project and its packages.
- Use applicatives and monads in Haskell to structure ap type checker with concatenative error messages.
- Use Haskell and IO to structure a simple game loop.

Furthermore, if there are exercises from previous weeks that you did not finish, you are encouraged to revisit those.

1 A Language Project

In this exercise, you will setup and implement a language project, with a parser that you are given and a type checker that you will implement for a simple language.

We will set up our language project using `cabal`, Haskell's standard package management system.

Cabal is typically installed on your system along with GHC. You should be able to call it from your command line; e.g.:

```
$ cabal --version
cabal-install version 3.12.1.0
compiled using version 3.12.1.0 of the Cabal library
```

We recommend that you use cabal 3.10 or newer, but older versions are likely to work just fine for this assignment too.

1.1 Setting Up A Cabal Project

(These instructions mirror those from *Exercises, Week 45*, assignment 4.2.)

Assuming you have `cabal` installed, run the following in your terminal (not GHCi):

```
$ cabal init mylang
```

The `cabal init` shell command will guide you through setting up a default `cabal` file. The default settings are fine for the purpose of this assignment.

Once the `cabal` file has been set up, you should have a folder containing the following:

```
$ cd mylang
$ ls -l
total 24
drwxr-xr-x@ 3 cbp  staff    96 Oct 31 11:46 app
```

```
-rw-r--r--@ 1 cbp  staff   116 Oct 31 11:46 CHANGELOG.md
-rw-r--r--@ 1 cbp  staff  1515 Oct 31 11:46 LICENSE
-rw-r--r--@ 1 cbp  staff  2470 Oct 31 11:46 mylang.cabal
$ ls -l app
total 8
-rw-r--r--@ 1 cbp  staff  48 Nov  1 10:45 Main.hs
```

Open `mylang.cabal` and find the line that says:

```
    build-depends:    base ^≥4.21.0.0
```

Modify this to include the `megaparsec` package:

```
    build-depends:    base ^≥4.21.0.0, megaparsec
```

1.2 A Parser for a Simple Language

Appendix A contains a parser for our language. The parser is implemented using the `megaparsec` library, which is a *parser combinator* library that lets us program a parser using applicative functors and monads. It is not necessary for this exercise to understand parser combinators, but if you'd like to learn how the combinators are implemented, see Chapter 13 of the book.

The module in Appendix A.1 goes into a file `app/AST.hs`, and the module in Appendix A.2 goes into a file `app/Parser.hs`.

Now you can go to your `app/Main.hs` file and add these modules as imports; e.g.:

```haskell
module Main where

import AST
import Parser

main :: IO ()
main = putStrLn "Hello, Haskell!"
```

Test that your parser works by either loading your file in your editor, or by running this command in the command line from the root of your `mylang` project folder:

```
$ cabal repl
```

This should spin up a GHCi REPL from where you can test that the parser transforms input strings into abstract syntax trees:

```
ghci> runParse "1 + 2 * 3 + 4"
Right (Add <span>
        (Add <span> (Lit <span> 1) (Mul <span> (Lit <span> 2) (Lit <span> 3)))
        (Lit <span> 4)) 4))
```

Each abstract syntax tree node is associated with information about where in the source code the given abstract syntax node occurs; i.e., which line number, and which characters.

For the purpose of inspecting abstract syntax the `Show Span` instance in module `app/AST.hs` ensures that we simply print `<span>` instead of displaying this meta-information to language developers.

You can also validate that pretty-printing a parsed abstract syntax tree gives back a program that resembles the input:

```
ghci> prettyExpr <$> runParse "1 + 2 * 3 + 4"
Right "1 + 2 * 3 + 4"
```

Here `prettyExpr` is also defined `app/AST.hs`.

1.3 A Functor

Running your type checker should either yield a list of errors, representing typing issues found during type checking, or, when type checking succeeds, some result. To this end, your type checker should use this result type:

```
data Result a
  = Err [String]
  | OK a
  deriving (Show , Functor)
```

In order to derive the `Functor` instance for this data type via the `deriving Functor` command above, you can make the following modification to your `mylang.cabal`.

Find this line in your cabal file again:

```
    build-depends:    base ^≥4.21.0.0, megaparsec
```

Add a `default-extensions: DeriveFunctor` clause below this line, as follows:

```
    build-depends:    base ^≥4.21.0.0, megaparsec

    default-extensions: DeriveFunctor
```

1.3.1 Implement `Applicative` and `Monad` Instances

Implement an `Applicative` and `Monad` instance for `Result`. Your applicative instance should concatenate type errors, such that all encountered type errors are reported.

1.4 A First Type Checker

We are now ready to implement our first type checker in `Main.hs`.

We will use two different functions:

- `check` for checking that an expression has an expected type; and
- `infer` for inferring what type an expression has.

Here are the first few cases of your type checking and type inference functions:

```
check :: Expr -> Ty -> Result ()
check (Lit  _ _)     INT  = pure ()
check (Add  _ e1 e2) INT  = check e1 INT *> check e2 INT
check (Unit _)       UNIT = pure ()
check e              t    = case infer e of
```

```
  OK t' -> Err [ "Error: the expression "
                 ++ prettyExpr e
                 ++ " at " ++ prettySpan (exprSpan e)
                 ++ " was expected to have type "
                 ++ prettyTy t ++ " but has type "
                 ++ prettyTy t' ]
  Err _ -> Err [ "Error: the expression "
                 ++ prettyExpr e
                 ++ " at " ++ prettySpan (exprSpan e)
                 ++ " was expected to have type "
                 ++ prettyTy t ++ " but its type could not be determined." ]

infer :: Expr -> Result Ty
infer (Lit _ _)     = pure INT
infer (Add _ e1 e2) = INT <$ check e1 INT <* check e2 INT
infer (Unit _)      = pure UNIT
infer e             =
  Err [ "Error: the type checker cannot type check expression "
        ++ prettyExpr e
        ++ " at " ++ prettySpan (exprSpan e) ]

runTC :: String -> IO ()
runTC s = case runParse s of
  Left e -> putStrLn e
  Right x -> case infer x [] of
    Err es -> do
      putStrLn "The program has the following typing issues:"
      putStrLn ("- " ++ intercalate "\n- " es)
    OK t -> putStrLn (prettyTy t)
```

Note that:

- The `intercalate` function comes from `Data.List`, so in order to call it you need to import that module.
- The functions above use <$ and <* which are specialized versions of <$> (an alias for `fmap`) and <*>. Use GHCi and Hoogle to work out what they do; e.g.:

  ```
  λ> :t (<$)
  (<$) :: Functor f => a -> f b -> f a
  λ> :t (<*)
  (<*) :: Applicative f => f a -> f b -> f a
  ```

If you have implemented your `Applicative` and `Monad` instances correctly, you should see similar results as these:

```
ghci> runTC "1 + 2 + 3 + 4"
int
ghci> ghci> runTC "() + 2 + 3 + 4"
The program has the following typing issues:
- Error: the expression () at <input>:1:1-1:4 was expected
```

```
    to have type int but has type unit
ghci> runTC "() + 2 + () + 4"
The program has the following typing issues:
- Error: the expression () at <input>:1:1-1:4 was expected
    to have type int but has type unit
- Error: the expression () at <input>:1:10-1:13 was expected
    to have type int but has type unit
```

1.5 Implement Case for `Mul`

Recall that the data type `Expr` is defined as follows:

```haskell
data Expr
  = Var  Span String
  | Lit  Span Integer
  | Lam  Span String Expr
  | App  Span Expr Expr
  | Let  Span String (Maybe Ty) Expr Expr
  | Add  Span Expr Expr
  | Mul  Span Expr Expr
  | Unit Span
  deriving (Eq, Show)
```

Implement the missing case in your type checker for `Mul`.

2 A Game with State

Implement a turn-based, stateful, interaction loop which lets a user control how a robot moves around on a board (given by an $n \times m$ matrix) with barricades.

The robot cannot move through blockades, and cannot move beyond the edges of the board.

In each turn, the user decides which cell on the board the robot moves to next. The user could, for example, be presented with the following options in each turn:

```
Press:
- 'w' then [ENTER] to move up;
- 'a' then [ENTER] to move left;
- 's' then [ENTER] to move down;
- 'd' then [ENTER] to move right.
```

Furthermore, each turn should display the state of the board. Below is a helper function for pretty-printing a board state.

Make sure that your game correctly handles invalid inputs (e.g., inputs other than ‘w’, ‘a’, ‘s’, or ‘d’) as well as invalid moves (e.g., moving up if the robot is already in the top row of the board).

Your game and game state should be modeled using the following types:

```haskell
-- The position of the robot
type Pos = (Int,Int)
```

```haskell
-- Entities that may inhabit the board
data Entity = Robot | Barricade deriving Show

-- A board, where each inner list represents a full row of the board.
-- Each row should have the same number of columns.
type Board = [[Maybe Entity]]

-- A record representing the game state
data State = State
  { width    :: Int     -- board width
  , height   :: Int     -- board height
  , robotPos :: Pos     -- position of robot
  , board    :: Board } -- board with barricades and a single robot.
```

Your game should allow players to decide the width and height of the board; e.g.:

```haskell
main :: IO ()
main = do
  putStrLn "Enter the width of the board:"
  wstr <- getLine
  let w = read wstr :: Int
  putStrLn "Enter the height of the board:"
  hstr <- getLine
  let h = read hstr :: Int
  {- Recommended next steps:

     1. populate the initial game board, then
     2. call game loop to render current game state,
     3. accept inputs from user on robot movement,
     4. update game state accordingly, and
     5. call game loop recursively with new game state. -}
  pure ()
```

The following helper functions 'render' (pretty-print) entities and boards:

```haskell
renderEntity :: Maybe Entity -> Char
renderEntity Nothing          = ' '
renderEntity (Just Robot)     = 'R'
renderEntity (Just Barricade) = '#'

-- Note: requires you to import `Data.List`
renderBoard :: Board -> String
renderBoard b
  = let w = length $ headDef [] b
    in intercalate ("\n" ++ take (w*2-1) (repeat '-') ++ "\n")
       $ map (intersperse '|' . map renderEntity) b

-- Auxiliary helper
headDef :: a -> [a] -> a
headDef def []  = def
```

```haskell
headDef _ (x:_) = x
```

A copy-paste friendly version of this assignment (and thus the helper functions above) is available on ItsLearning.

3 Challenge Assignments

These assignments build on the type checker assignment from earlier.

3.1 Challenge 1: Implement Cases for `Let` and `Var`

Names in programs must be well-bound.

That is, when we reference a name `x` in a program, that name must be bound in the program's context.

To ensure that names are well-bound, we use an auxiliary data structure `Ctx` which represents names and types of each name in the context:

```haskell
type Ctx = [(String, Ty)]
```

Variables `Var` are well-typed when they are bound in the context, and their type matches what we expect.

To type check variables, our type checking and inference functions must therefore be extended with an additional parameter for typing contexts:

```haskell
check :: Expr -> Ty -> Ctx -> Result ()
infer :: Expr -> Ctx -> Result Ty
```

Implement the missing case for `Var`. If you have implemented it correctly, you should see similar results as:

```
ghci> runTC "x"
The program has the following typing issues:
- Error: the identifier x at <input>:1:1-1:2 is not bound.
ghci> runTC "x + 1"
The program has the following typing issues:
- Error: the identifier x at <input>:1:1-1:3
    was expected to have type int but it is not bound.
ghci> runTC "x + y"
The program has the following typing issues:
- Error: the identifier x at <input>:1:1-1:3
    was expected to have type int but it is not bound.
- Error: the identifier y at <input>:1:5-1:6
    was expected to have type int but it is not bound.
```

Next, implement the missing case for `Let`. For the abstract syntax node `Let sp x Nothing e1 e2`, the case should first infer the type of `e1`, then bind `x` to that inferred type in the context passed down to check or infer `e2`.

If you have implemented your case correctly, you should see similar results as:

```
ghci> runTC "let x = 1 in x"
int
ghci> runTC "let x = 1 in let y = () in x"
int
ghci> runTC "let x = 1 in let y = () in y"
unit
ghci> runTC "let x = 1 in let y = () in x + y"
The program has the following typing issues:
- Error: the identifier y at <input>:1:32-1:33
    was expected to have type int but has type unit
ghci> runTC "let x = 1 in let y = () in y + y"
The program has the following typing issues:
- Error: the identifier y at <input>:1:28-1:30
    was expected to have type int but has type unit
- Error: the identifier y at <input>:1:32-1:33
    was expected to have type int but has type unit
ghci> runTC "let x = 1 in let x = () in x"
unit
```

Note that let bindings may have a type annotation. For abstract syntax nodes with type annotations `Let sp x (Just t) e1 e2`, we should first check that `e1` has type `t`, then bind `x` to `t` in the context passed down to check or infer `e2`.

If you have implemented your case correctly, you should see similar results as:

```
ghci> runTC "let (x : int) = 1 in let (y : unit) = () in x"
int
ghci> runTC "let (x : int) = 1 in let (y : unit) = () in y"
unit
ghci> runTC "let (x : int) = 1 in let (y : unit) = () in y + x"
The program has the following typing issues:
- Error: the identifier y at <input>:1:45-1:47
    was expected to have type int but has type unit
ghci> runTC "let (x : int) = 1 in let (y : unit) = () in y + y"
The program has the following typing issues:
- Error: the identifier y at <input>:1:45-1:47
    was expected to have type int but has type unit
- Error: the identifier y at <input>:1:49-1:50
    was expected to have type int but has type unit
ghci> runTC "let (x : int) = 1 in let (x : unit) = () in x"
unit
```

3.2 Challenge 2: Implement Cases for `Lam` and `App`

Implement the missing cases for `Lam` and `App`, where `Lam sp x e` represents a lambda; e.g., `\x -> e`.

Hint: we can only check the type of a lambda when we know its parameter type; and we only know the parameter type when we are checking, never when we are inferring the function. Make sure that your functions use `check` when possible.

If you have implemented the cases correctly, you should see similar results as these:

```
ghci> runTC "(\\x -> x)"
The program has the following typing issues:
- Error: the type checker cannot type check expression \x -> x at <input>:1:2-1:9
ghci> runTC "(\\x -> x) 2 + 3"
int
ghci> runTC "let (f : int -> int) = \\x -> x in f"
int -> int
ghci> runTC "let (f : (int -> int) -> int) = \\g -> g 42
             in let (h : int -> int) = \\x -> x + 1 in f h"
int
ghci> runTC "(\\f -> f) ((\\x -> x) 1 + 2) + 1"
int
ghci> runTC "(\\f -> f) ((\\x -> x) 1 + 2) + 1"
int
ghci> runTC "(\\x -> x) 1"
The program has the following typing issues:
- Error: the type checker cannot type check expression \x -> x at <input>:1:2-1:9
```

A Appendix: Parser for a Simple Language

The parser was generated by Copilot. The following prompt was used:

> Can you build me a parser in Haskell for a small functional language with the following AST?:
>
> ```
> data Expr
>   = Var  Span String
>   | Lit  Span Integer
>   | Lam  Span String Expr
>   | App  Span Expr Expr
>   | Let  Span String Expr Expr
>   | Add  Span Expr Expr
>   | Mul  Span Expr Expr
>   | Unit Span
>   deriving (Eq, Show)
> ```
>
> Here, ‘Span‘ should contain location information (source positions) for each AST node.
>
> I’d like to use parser combinators for my parser.
>
> The parser should use similar precedence rules for parsing as, e.g., Haskell. The syntax should be Haskell inspired too.

The generated parser was subsequently debugged through a series of follow-up prompts, to fix compilation errors, and several manual adjustments were made to fix bugs. The parser is likely to contain more bugs than we have fixed already.

The pretty-printer in `AST.hs` was added using the following prompt:

> Can you also generate a pretty-printing function for ASTs?

A.1 AST.hs

```haskell
module AST
  ( Expr(..)
  , Span(..)
  , Ty (..)
  , exprSpan
  , prettyExpr
  , prettySpan
  , prettyTy
  ) where

import Text.Megaparsec.Pos ( SourcePos
                           , sourceName
                           , unPos
                           , sourceColumn
                           , sourceLine )

-- | Source span with a start and end 'SourcePos'
```

```haskell
data Span = Span
  { spanStart :: SourcePos
  , spanEnd   :: SourcePos
  }
  deriving (Eq)

instance Show Span where
  show _ = "<span>"

data Expr
  = Var  Span String
  | Lit  Span Integer
  | Lam  Span String Expr
  | App  Span Expr Expr
  | Let  Span String (Maybe Ty) Expr Expr
  | Add  Span Expr Expr
  | Mul  Span Expr Expr
  | Unit Span
  deriving (Eq, Show)

-- | Extract the span from any expression node
exprSpan :: Expr -> Span
exprSpan (Var s _)     = s
exprSpan (Lit s _)     = s
exprSpan (Lam s _ _)   = s
exprSpan (App s _ _)   = s
exprSpan (Let s _ _ _ _) = s
exprSpan (Add s _ _)   = s
exprSpan (Mul s _ _)   = s
exprSpan (Unit s)      = s

data Ty
  = UNIT
  | INT
  | ARR Ty Ty
  deriving (Eq, Show)

-- | Pretty-print an expression (ignores spans).
prettyExpr :: Expr -> String
prettyExpr = prettyExprPrec 0

-- precedence levels (higher binds tighter)
precLam, precLet, precAdd, precMul, precApp :: Int
precLam = 1
precLet = 1
precAdd = 4
precMul = 5
precApp = 6
```

```haskell
prettyExprPrec :: Int -> Expr -> String
prettyExprPrec _ (Var _ name) = name
prettyExprPrec _ (Lit _ n) = show n
prettyExprPrec _ (Unit _) = "()"
prettyExprPrec ctx (Lam _ arg body) =
  parenIfNeeded ctx precLam $ "\\" ++ arg ++ " -> "
  ++ prettyExprPrec precLam body
prettyExprPrec ctx (Let _ name Nothing bound body) =
  parenIfNeeded ctx precLet $
    "let " ++ name ++ " = "
    ++ prettyExprPrec precLet bound ++ " in "
    ++ prettyExprPrec precLet body
prettyExprPrec ctx (Let _ name (Just t) bound body) =
  parenIfNeeded ctx precLet $
    "let (" ++ name ++ " : " ++ prettyTy t ++ ") = "
    ++ prettyExprPrec precLet bound ++ " in "
    ++ prettyExprPrec precLet body
prettyExprPrec ctx (App _ f x) =
  parenIfNeeded ctx precApp $ prettyExprPrec precApp f
  ++ " " ++ prettyExprPrec (precApp + 1) x
prettyExprPrec ctx (Add _ l r) =
  parenIfNeeded ctx precAdd $ prettyExprPrec precAdd l
  ++ " + " ++ prettyExprPrec (precAdd + 1) r
prettyExprPrec ctx (Mul _ l r) =
  parenIfNeeded ctx precMul $ prettyExprPrec precMul l
  ++ " * " ++ prettyExprPrec (precMul + 1) r

prettyTy :: Ty -> String
prettyTy = prettyTyPrec 0

prettyTyPrec :: Int -> Ty -> String
prettyTyPrec _   UNIT        = "unit"
prettyTyPrec _   INT         = "int"
prettyTyPrec ctx (ARR t1 t2) =
  parenIfNeeded ctx precARR $ prettyTyPrec (precARR+1) t1
                            ++ " -> " ++ prettyTyPrec precARR t2

precARR :: Int
precARR = 1

parenIfNeeded :: Int -> Int -> String -> String
parenIfNeeded ctx prec s =
  if prec < ctx then "(" ++ s ++ ")" else s


-- pretty-print a single SourcePos as "file:line:col"
prettySourcePos :: SourcePos -> String
```

```haskell
prettySourcePos p =
  let file = sourceName p
      line = unPos (sourceLine p)
      col  = unPos (sourceColumn p)
  in file ++ ":" ++ show line ++ ":" ++ show col

-- pretty-print a Span as either
--  - "file:line1:col1-line2:col2"  (if same file)
--  - "file1:line1:col1 - file2:line2:col2"
prettySpan :: Span -> String
prettySpan (Span s e)
  | sourceName s == sourceName e =
      let file = sourceName s
          l1 = unPos (sourceLine s)
          c1 = unPos (sourceColumn s)
          l2 = unPos (sourceLine e)
          c2 = unPos (sourceColumn e)
      in file ++ ":" ++ show l1 ++ ":" ++ show c1
         ++ "-" ++ show l2 ++ ":" ++ show c2
  | otherwise =
      prettySourcePos s ++ " - " ++ prettySourcePos e
```

A.2 Parser.hs

```haskell
module Parser
  ( parseExpr
  , runParseIO
  , runParse
  ) where

import AST

import Control.Monad (void)
import Data.Void (Void)
import Text.Megaparsec
import Text.Megaparsec.Char
import qualified Text.Megaparsec.Char.Lexer as L

type Parser = Parsec Void String

-- | Space consumer: spaces, line comments starting with `--`, and block comments
sc :: Parser ()
sc = L.space space1 (L.skipLineComment "--") (L.skipBlockComment "{-" "-}")

lexeme :: Parser a -> Parser a
lexeme = L.lexeme sc

symbol :: String -> Parser String
```

```haskell
symbol = L.symbol sc

parens :: Parser a -> Parser a
parens = between (symbol "(") (symbol ")")

reserved :: String -> Parser ()
reserved w = void (string w) *> notFollowedBy alphaNumChar *> sc

-- | Record a span around a parser
spanAround :: Parser a -> Parser (Span, a)
spanAround p = do
  start <- getSourcePos
  x <- p
  end <- getSourcePos
  pure (Span start end, x)

identifier :: Parser String
identifier = (lexeme . try) (p >>= check)
  where
    -- allow identifiers to start with a normal letter or a small set of
    -- special characters (underscore/apostrophe), but NOT a digit
    p = (:) <$> (letterChar <|> char '_' <|> char '\'') <*> many (alphaNumChar <|> char
    check w
      | w `elem` ["let","in"] = fail ("reserved word " ++ w)
      | otherwise = pure w

unitP :: Parser Expr
unitP = do
  (sp, _) <- spanAround (symbol "()")
  pure (Unit sp)

varP :: Parser Expr
varP = do
  (sp, name) <- spanAround identifier
  pure (Var sp name)

-- | Parse a lambda: \x y -> body  (also accepts λ)
lamP :: Parser Expr
lamP = do
  start <- getSourcePos
  void (lexeme (char '\\' <|> char 'λ'))
  params <- some identifier
  void (symbol "->")
  body <- exprP
  end <- getSourcePos
  -- nest lambdas for multiple parameters; give each lam the span from start..body-end
  let mkLam name acc = Lam (Span start end) name acc
  pure (foldr mkLam body params)
```

```haskell
-- let can be annotated: either `let x = ...` or `let (x : ty) = ...`
letP :: Parser Expr
letP = do
  start <- getSourcePos
  reserved "let"
  -- either: (name : ty)  or plain identifier
  (name, mty) <-
    choice
      [ do
          (_sp, (n, ty)) <- spanAround (between (symbol "(") (symbol ")") $ do
            n <- identifier
            void (symbol ":")
            t <- tyP
            pure (n, t))
          pure (n, Just ty)
      , do n <- identifier; pure (n, Nothing)
      ]
  void (symbol "=")
  bound <- exprP
  reserved "in"
  body <- exprP
  end <- getSourcePos
  pure (Let (Span start end) name mty bound body)

atom :: Parser Expr
atom = choice [unitP, parens exprP, numberP, lamP, letP, varP]

-- | Simple numeric literal parser: keep as `Var` (string) to match the given
-- AST
numberP :: Parser Expr
numberP = do
  (sp, digits) <- spanAround (lexeme (some digitChar))
  let n = read digits :: Integer
  pure (Lit sp n)

-- | Build application nodes by left-folding adjacent atoms
appP :: Parser Expr
appP = do
  first <- atom
  rest <- many atom
  pure (foldl makeApp first rest)
  where
    makeApp l r = App (Span (spanStart (exprSpan l)) (spanEnd (exprSpan r))) l r

-- helper: binary operator parser producing a combining function
binOp :: String -> (Span -> Expr -> Expr -> Expr) -> Parser (Expr -> Expr -> Expr)
binOp sym ctor = do
```

```
  void (symbol sym)
  pure (\t1 t2 -> ctor (Span (spanStart (exprSpan t1)) (spanEnd (exprSpan t2))) t1 t2)

-- Simple left-associative chain helper
chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainl1 p op = do
  x <- p
  rest x
  where
    rest x = (do f <- op
                 y <- p
                 rest (f x y)) <|> pure x

mulP :: Parser Expr
mulP = chainl1 appP (binOp "*" Mul)

addP :: Parser Expr
addP = chainl1 mulP (binOp "+" Add)

-- | Top-level expression parser: let / lambda / operators
exprP :: Parser Expr
exprP = choice [letP, lamP, addP]

------------------------------------------------------------
-- Types parsing
------------------------------------------------------------

-- right-associative chain helper
chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser a
chainr1 p op = do
  x <- p
  (do f <- op
      y <- chainr1 p op
      pure (f x y)) <|> pure x

tyAtom :: Parser Ty
tyAtom =
  choice
    [ do _ <- reserved "unit"; pure UNIT
    , do _ <- reserved "int"; pure INT
    , do t <- between (symbol "(") (symbol ")") tyP; pure t
    ]

tyP :: Parser Ty
tyP = chainr1 tyAtom (do void (symbol "->"); pure (\a b -> ARR a b))

-- | Parse a full input string into an Expr
parseExpr :: String -> Either (ParseErrorBundle String Void) Expr
```

```haskell
parseExpr = runParser (sc *> exprP <* eof) "<input>"

-- | Small utility to run the parser on stdin or a file and print result
runParseIO :: IO ()
runParseIO = do
  input <- getContents
  case parseExpr input of
    Left err -> putStrLn (errorBundlePretty err)
    Right ast -> print ast

runParse :: String -> Either String Expr
runParse str = do
  case parseExpr str of
    Left err -> Left $ errorBundlePretty err
    Right ast -> pure ast
```