Xeus-Haskell: Docs and Literate Sources

Masaya Taniguchi and xeus-haskell contributors

Contents

1 Welcome to Xeus-Haskell

Xeus-Haskell is a Jupyter kernel for Haskell based on the native implementation of the Jupyter protocol, xeus.

1.1 Preliminaries

To start using or developing Xeus-Haskell, please note the following:

- **MicroHs Backend:** Xeus-Haskell uses MicroHs, a minimal implementation of Haskell. Therefore, supported libraries and extensions are limited.
- **Native Protocol:** Based on xeus (C++), so the kernel must be compiled for the specific platform you are using.
- **WebAssembly Support:** You can compile this kernel to WebAssembly to run it directly in browsers!

1.2 Quickstart

Since this project is not yet available on Conda-forge or Emscripten-forge, you need to build it from source. We have organized the workflows using pixi.

1.2.1 For Native JupyterLab

```
git clone https://github.com/jupyter-xeus/xeus-haskell
cd xeus-haskell
pixi run -e default prebuild
pixi run -e default build
pixi run -e default install
pixi run -e default serve # JupyterLab is ready!
```

1.2.2 For WebAssembly JupyterLite

```
git clone https://github.com/jupyter-xeus/xeus-haskell
cd xeus-haskell
pixi install -e prebuild
pixi run -e wasm-build prebuild
pixi run -e wasm-build build
pixi run -e wasm-build install
# pixi run -e wasm-build fix-emscripten-links # Run this if you encounter linking errors
pixi run -e wasm-build serve # JupyterLite is ready!
```

1.3 A Deep Dive into Xeus-Haskell

- Contributing Guide
- Communication Protocol
- REPL Mechanism

- Roadmap & TODO
- AI Agent Guide

2 Communication and System Architecture

This document describes the internal communication flow of `xeus-haskell`, from the Jupyter frontend down to the MicroHs runtime.

2.1 System Stack

The communication stack is organized as follows: Jupyter Frontend $\leftrightarrow$ Xeus C++ Library $\leftrightarrow$ Xeus-Haskell Kernel $\leftrightarrow$ `mhs_repl.cpp` Bridge $\leftrightarrow$ Repl.lhs Compiler API $\leftrightarrow$ MicroHs Runtime.

2.2 Layer Details

2.2.1 Jupyter and Xeus Protocol Layer

This layer handles the ZeroMQ-based Jupyter protocol. It manages the heartbeat, shell, and control sockets.

2.2.2 Xeus to Kernel Interpreter Layer

The `xinterpreter` implementation hooks into the Xeus kernel lifecycle. It receives execution requests and delegates them to the Haskell backend.

2.2.3 C++ Bridge to Haskell Runtime Layer

This is the bridge layer between C++ and the Haskell-compiled Compiler API wrapper (`Repl.lhs`). The `mhs_repl.cpp` bridge performs three primary tasks:

1. **Initialize MicroHs Runtime**: The `MicroHsRepl` constructor locates runtime files (standard libraries) using `microhs_runtime_dir`. After initialization, it evaluates a trivial expression (`"0"`) to warm up the compiler and cache library compilation.
2. **Capture Standard Output**: To receive output from the Haskell world, we use POSIX pipes to redirect `stdout`.
   - C++ sets up a redirection from `stdout` to a temporary buffer.
   - After Haskell evaluation completes, the buffer is read.
   - For implementation details, see `capture_stdout` in the source code.
3. **Parse Captured Content**: The output is parsed to distinguish plain text from rich content (HTML, LaTeX, etc.) using the internal protocol described below.

2.2.4 4. REPL Engine and MicroHs Compiler Layer

This layer uses the MicroHs Compiler API to evaluate Haskell code strings in the runtime environment.

2.3 Rich Content Protocol

`xeus-haskell` supports rendering rich media (HTML, images, LaTeX) by embedding control sequences in the standard output stream.

2.3.1 Protocol Structure

Rich content is strictly delimited by **ASCII control characters**:

```
<STX><MIME_TYPE><US><CONTENT><ETX>
```

Component	ASCII (Hex)	Description
STX	02	**S**tart of **Tex**t
MIME_TYPE	-	The target MIME type (e.g., `text/html`)
US	1F	**U**nit **S**eparator
CONTENT	-	The raw data/string to be rendered
ETX	03	**E**nd of **Tex**t

2.3.2 Supported Content Types

- `text/plain`: Standard console output.
- `text/html`: Rendered as HTML in Jupyter.
- `text/latex`: Mathematical equations rendered via MathJax.
- `text/markdown`: Formatted text and tables.

All content is expected to be **UTF-8** encoded.

2.4 Haskell Display API

The protocol is abstracted in Haskell through the `Display` typeclass.

2.4.1 Display Typeclass Interface

To enable rich output for a custom type, implement the `Display` typeclass:

```haskell
import XHaskell.Display

-- | Data structure for the protocol
data DisplayData = DisplayData {
    mimeType :: String,
    content  :: String
}
```

2.4.2 Implementation Examples

```haskell
newtype HTMLString = HTMLString String
instance Display HTMLString where
    display (HTMLString s) = DisplayData "text/html" s

newtype LaTeXString = LaTeXString String
instance Display LaTeXString where
    display (LaTeXString s) = DisplayData "text/latex" s

```

```haskell
-- Default instance for plain strings
instance Display String where
    display s = DisplayData "text/plain" s
```

2.4.3 Protocol Generation

The `DisplayData` record implements a `Show` instance that automatically formats the string into the `<STX>...<ETX>` protocol.

```haskell
putStr $ show (display (HTMLString "<p style='color: red'>Hello Haskell!</p>"))
```

```
02 text/html 1F <p style='color: red'>Hello Haskell!</p> 03
```

3 Contributing to xeus-haskell

Thank you for your interest in contributing to `xeus-haskell`! Xeus and xeus-haskell are subprojects of Project Jupyter and are subject to the Jupyter governance and Code of conduct.

3.1 General Guidelines

For general documentation about contributing to Jupyter projects, see the Project Jupyter Contributor Documentation.

3.2 Community

The Xeus team organizes public video meetings. The schedule for future meetings and minutes of past meetings can be found on our Team Compass.

3.3 Getting Started

3.3.1 Prerequisites

The project uses Pixi for dependency management and workflow automation. If you haven't installed it yet:

```
curl -fsSL https://pixi.sh/install.sh | sh
```

3.3.2 Setting Up Your Environment

1. Fork the repository on GitHub.
2. Clone your fork locally:

```
git clone https://github.com/<your-username>/xeus-haskell.git
cd xeus-haskell
```

1. Initialize the development environment:

```
pixi run -e default prebuild
```

The `default` environment includes all necessary tools (CMake, C++ compilers, Python for testing, etc.).

3.4 Development Workflow

3.4.1 Building the Kernel

To build and install the kernel in your local environment:

```
pixi run -e default build
pixi run -e default install
```

3.4.2 Running Tests

We use `pytest` and `ctest` for verifying the kernel:

```
# Run C++ tests
pixi run -e default ctest

# Run Python-based Jupyter kernel tests
pixi run -e default pytest
```

3.4.3 Working with WebAssembly (JupyterLite)

If you are contributing to the WebAssembly build, you need the `wasm-build` and `wasm-host` environments:

```
# Prepare the WASM host environment
pixi install -e wasm-build
pixi install -e wasm-host

# Build for WebAssembly
pixi run -e wasm-build prebuild
pixi run -e wasm-build build
pixi run -e wasm-build install

# Run the JupyterLite server locally
pixi run -e wasm-build serve
```

3.5 Project Structure

- `src/`: C++ source code for the kernel and REPL bridge.
- `include/`: C++ header files.
- `share/`: Haskell side of the implementation, including `Repl.lhs` and MicroHs libraries.

- `test/`: Python tests for kernel functionality.
- `xhaskell.tex`: Integrated LaTeX manual source that includes all architecture chapters and literate Haskell modules.

3.6 Style Guidelines

- **C++**: Follow modern C++ practices. We use Xeus as the base library.
- **Haskell**: Since we use MicroHs, avoid dependencies or language features not supported by the MicroHs compiler.
- **Documentation**: Write documentation in LaTeX (`.tex`) and keep examples runnable with the current Pixi workflows.

3.6.1 Building Documentation

Generate the project PDF documentation with:

```
pixi run -e docs generate
```

3.7 Submitting Changes

1. Create a new branch for your feature or bugfix.
2. Ensure all tests pass.
3. Submit a Pull Request with a clear description of the changes.

3.8 Community

Join our public video meetings! The schedule and minutes are available on our Team Compass.

4 TODO: Unimplemented Features & Roadmap

This document tracks planned improvements and currently missing features for `xeus-haskell`.
Status symbols: ✓ completed, ✗ pending, ℙ planned.

4.1 Core Kernel Features

✓ **`is_complete` implementation**: Map parsability of mixed cells to Jupyter's completeness status.

✗ **Incomplete Status**: Refine `is_complete` to distinguish between `invalid` and `incomplete` code (e.g., unclosed delimiters).

✓ **`inspect_request`**: Support for "Introspection" (Shift+Tab in Jupyter) to show documentation or type signatures for identifiers.

✗ **`history_request`**: Implementation of the Jupyter history protocol to allow searching and retrieving previous cell inputs.

✗ **Advanced Completion**: Improve `completion_request` to support qualified names (e.g., `Prelude.putStrLn`) and type-aware suggestions.

✗ **Kernel Interrupt**: Add support for interrupting long-running Haskell executions (SIGINT handling).

4.2 REPL Improvements

✗ **Selective Re-compilation**: Instead of concatenating all previous definitions for every new definition, implement a dependency-tracking system to only re-compile what is necessary.

✗ **Better Error Reporting**: Map MicroHs compile errors back to the specific line numbers in the Jupyter cell.

✗ **WASM Optimization**: Reduce the size of the WebAssembly bundle and optimize the initial "warmup" time in the browser.

✓ **Language Extensions**: Work with the MicroHs upstream to support more modern Haskell extensions (e.g., `GADTs`, `TypeFamilies`).

4.3 Rich Output Display System

ℙ **Standard Library Instances**: Add `Display` instances for more types in the MicroHs standard library.

ℙ **Data Visualization**: Create bridges for common Haskell charting libraries to output Jupyter-compatible JSON/Vega-Lite data.

ℙ **Widgets**: Initial investigation into supporting Jupyter Widgets (ipywidgets protocol).

4.4 Infrastructure & Testing

ℙ **Enhanced CI**: Add automated UI tests using `playwright` or `selenium` to verify the kernel works in a real JupyterLab/JupyterLite instance.

ℙ **Benchmarking**: Create a suite of performance benchmarks to track compilation and execution speed regressions.

ℙ **Conda-forge/Emscripten-forge**: Package the kernel for easier installation without needing to build from source.

5 Literate Haskell Source

The MicroHs REPL layer provides the continuity required by notebooks: each cell is evaluated against a persistent semantic context rather than in isolation. The C++ Xeus interpreter forwards requests through a small FFI surface, and this module translates those requests into explicit, auditable transitions over `ReplCtx`.

A practical constraint of MicroHs is that startup work and incremental compilation cost must be managed carefully. The implementation therefore keeps the FFI contract narrow and deterministic so cache reuse and error normalization remain predictable across repeated notebook interactions.

5.1 Foreign-Function Interface Module

This module is the semantic membrane between the C++ kernel process and the Haskell REPL engine. Its central task is to convert foreign calls into controlled state transitions over a long-lived `ReplCtx` while preserving a stable, C-friendly contract for status codes and error buffers.

Formally, each exported endpoint implements a relation of the form $\Delta_{\text{ffi}} : (h, x) \mapsto (h', s, p)$, where h is a stable pointer to mutable session state, x is decoded source input, h' is the post-state handle, $s \in \{c\_OK, c\_ERR\}$ is a machine-level status code, and p is optional textual payload. Query-style calls preserve state, while define/run/execute calls may advance cache and symbol tables.

The design objective is pragmatic: keep the foreign boundary narrow, deterministic, and explicit about ownership. This is why allocation and deallocation of returned C strings are encoded in the API itself, and why exception handling is normalized before crossing the language boundary.

```haskell
{-# LANGUAGE ForeignFunctionInterface #-}
module Repl (
  mhsReplNew,
  mhsReplFree,
  mhsReplDefine,
  mhsReplRun,
  mhsReplFreeCString,
  mhsReplCanParseDefinition,
  mhsReplCanParseExpression,
  mhsReplCompletionCandidates,
  mhsReplInspect,
  mhsReplIsComplete,
  mhsReplExecute
) where
import qualified Prelude ()
import MHSPrelude
import Control.Exception (try, SomeException, displayException)
import Data.IORef
import Data.List (nub)
import System.IO (putStrLn)
import Foreign.C.String (CString, peekCString, peekCStringLen, newCString)
import Foreign.C.Types (CInt, CSize)
import Foreign.Marshal.Alloc (free)
import Foreign.Ptr (Ptr, nullPtr)
import Foreign.StablePtr (StablePtr, newStablePtr, freeStablePtr, deRefStablePtr)
import Foreign.Storable (poke)
import Repl.Context
import Repl.Error
import Repl.Analysis
import Repl.Executor
```

`c_OK` is the success status returned to C/C++ callers when an FFI action completes without error. It stays fixed at zero so callers can branch on a stable convention.

```
c_OK, c_ERR :: CInt
c_OK  = 0
```

`c_ERR` is the failure status returned to C/C++ callers when an FFI action fails. It stays fixed at `-1` and pairs with an optional allocated error message.

```
c_ERR = -1
```

`writeErrorCString` stores a newly allocated error message into an output pointer when one is provided. The function is intentionally no-op for null pointers, allowing callers to omit error buffers safely.

```
writeErrorCString :: Ptr CString -> String -> IO ()
writeErrorCString errPtr msg =
  if errPtr == nullPtr then pure ()
  else newCString msg >>= poke errPtr
```

`mhsReplNew` creates a fresh REPL handle backed by an `IORef ReplCtx`. It decodes the runtime path from C strings, initializes the context, and returns a stable pointer for cross-language ownership.

```
foreign export ccall "mhs_repl_new"         mhsReplNew          :: CString ->
↪  CSize -> IO ReplHandle
foreign export ccall "mhs_repl_free"        mhsReplFree         :: ReplHandle ->
↪  IO ()
foreign export ccall "mhs_repl_define"      mhsReplDefine       :: ReplHandle ->
↪  CString -> CSize -> Ptr CString -> IO CInt
foreign export ccall "mhs_repl_run"         mhsReplRun          :: ReplHandle ->
↪  CString -> CSize -> Ptr CString -> IO CInt
foreign export ccall "mhs_repl_execute"     mhsReplExecute      :: ReplHandle ->
↪  CString -> CSize -> Ptr CString -> IO CInt
foreign export ccall "mhs_repl_is_complete" mhsReplIsComplete :: ReplHandle ->
↪  CString -> CSize -> IO CString
foreign export ccall "mhs_repl_free_cstr"   mhsReplFreeCString  :: CString -> IO
↪  ()
foreign export ccall "mhs_repl_can_parse_definition" mhsReplCanParseDefinition ::
↪  CString -> CSize -> IO CInt
foreign export ccall "mhs_repl_can_parse_expression" mhsReplCanParseExpression ::
↪  CString -> CSize -> IO CInt
foreign export ccall "mhs_repl_completion_candidates" mhsReplCompletionCandidates
↪  :: ReplHandle -> IO ()
foreign export ccall "mhs_repl_inspect" mhsReplInspect :: ReplHandle -> CString ->
↪  CSize -> Ptr CString -> IO CInt
mhsReplNew :: CString -> CSize -> IO ReplHandle
mhsReplNew cstr csize = do
  str <- peekSource cstr csize
  ctx <- initialCtx str
  ref <- newIORef ctx
  newStablePtr ref
```

`mhsReplFree` releases the stable pointer created by `mhsReplNew`. It delegates directly to `freeStablePtr`, keeping ownership semantics explicit.

```
mhsReplFree :: ReplHandle -> IO ()
mhsReplFree = freeStablePtr
```

`mhsReplFreeCString` frees strings previously allocated by this module and returned through FFI out-pointers. It calls the matching allocator-side `free` so C++ can safely release messages.

```
mhsReplFreeCString :: CString -> IO ()
mhsReplFreeCString = free
```

`withHandleInput` is a shared adapter that decodes input and exposes both the context reference and decoded source string. This keeps repetitive pointer handling out of each endpoint implementation.

```
withHandleInput
  :: ReplHandle
  -> CString
  -> CSize
  -> (IORef ReplCtx -> String -> IO a)
  -> IO a
withHandleInput h srcPtr srcLen k = do
  ref <- deRefStablePtr h
  src <- peekSource srcPtr srcLen
  k ref src
```

`normalizeResult` folds both thrown exceptions and explicit `ReplError` values into a single `Either ReplError a` shape. That guarantees a uniform error path for all higher-level runners.

```
normalizeResult
  :: Either SomeException (Either ReplError a)
  -> Either ReplError a
normalizeResult result =
  case result of
    Left ex -> Left (ReplRuntimeError (displayException ex))
    Right val -> val
```

`runStatefulAction` executes an action that may mutate `ReplCtx` and writes the updated state back on success. It catches exceptions, normalizes failures, and maps outcomes to `c_OK`/`c_ERR` plus optional error strings.

```
runStatefulAction
  :: (ReplCtx -> String -> IO (Either ReplError ReplCtx))
  -> ReplHandle
  -> CString
  -> CSize
  -> Ptr CString
  -> IO CInt
runStatefulAction act h srcPtr srcLen errPtr =
  withHandleInput h srcPtr srcLen $ \ref src -> do
    ctx <- readIORef ref
    result <- try (act ctx src) :: IO (Either SomeException (Either ReplError
↪   ReplCtx))
```

```
    case normalizeResult result of
      Left err -> writeErrorCString errPtr (prettyReplError err) >> pure c_ERR
      Right ctx' -> writeIORef ref ctx' >> pure c_OK
```

`runQueryAction` executes a read-style action that returns textual output rather than a new context. It applies the same error normalization flow as stateful actions while writing result text into the supplied output pointer.

```
runQueryAction
  :: (ReplCtx -> String -> IO (Either ReplError String))
  -> ReplHandle
  -> CString
  -> CSize
  -> Ptr CString
  -> IO CInt
runQueryAction act h srcPtr srcLen outPtr =
  withHandleInput h srcPtr srcLen $ \ref src -> do
    ctx <- readIORef ref
    result <- try (act ctx src) :: IO (Either SomeException (Either ReplError
↪   String))
    case normalizeResult result of
      Left err -> writeErrorCString outPtr (prettyReplError err) >> pure c_ERR
      Right value -> newCString value >>= poke outPtr >> pure c_OK
```

`mhsReplDefine` handles a definition snippet and persists resulting context state. It is a thin alias to the shared stateful runner with `replDefine`.

```
mhsReplDefine :: ReplHandle -> CString -> CSize -> Ptr CString -> IO CInt
mhsReplDefine = runStatefulAction replDefine
```

`mhsReplRun` executes a runnable snippet within the current context. It reuses `runStatefulAction` so cache and symbols are updated consistently.

```
mhsReplRun :: ReplHandle -> CString -> CSize -> Ptr CString -> IO CInt
mhsReplRun = runStatefulAction replRun
```

`mhsReplExecute` is the unified entrypoint that supports split plans and meta commands. It delegates to `replExecute` through the shared stateful runner.

```
mhsReplExecute :: ReplHandle -> CString -> CSize -> Ptr CString -> IO CInt
mhsReplExecute = runStatefulAction replExecute
```

`mhsReplIsComplete` reports whether input appears complete, incomplete, or invalid. It reads current context, computes status via `replIsComplete`, and returns a newly allocated CString.

```
mhsReplIsComplete :: ReplHandle -> CString -> CSize -> IO CString
mhsReplIsComplete h srcPtr srcLen =
  withHandleInput h srcPtr srcLen $ \ref src -> do
    ctx <- readIORef ref
    status <- replIsComplete ctx src
    newCString status
```

`mhsReplInspect` resolves a symbol and returns rendered type/value information. It uses `runQueryAction` with `replInspect` to keep output/error handling uniform.

```haskell
mhsReplInspect :: ReplHandle -> CString -> CSize -> Ptr CString -> IO CInt
mhsReplInspect = runQueryAction replInspect
```

`mhsReplCanParseDefinition` is a fast parse probe for definition snippets. It decodes source and returns 1/0 to match C-friendly boolean conventions.

```haskell
mhsReplCanParseDefinition :: CString -> CSize -> IO CInt
mhsReplCanParseDefinition ptr len = do
  code <- peekSource ptr len
  pure $ if canParseDefinition code then 1 else 0
```

`mhsReplCanParseExpression` is the expression counterpart to the definition parse probe. It uses the same C-int boolean return convention.

```haskell
mhsReplCanParseExpression :: CString -> CSize -> IO CInt
mhsReplCanParseExpression ptr len = do
  code <- peekSource ptr len
  pure $ if canParseExpression code then 1 else 0
```

`mhsReplCompletionCandidates` prints merged keyword and local identifier candidates. It reads current definitions, derives candidates, de-duplicates with `nub`, and emits one candidate per line to stdout.

```haskell
mhsReplCompletionCandidates :: ReplHandle -> IO ()
mhsReplCompletionCandidates h = do
  ref <- deRefStablePtr h
  ctx <- readIORef ref
  let source = currentDefsSource ctx
  let localIdents = completionCandidates source
  let allCandidates = nub (reservedIds ++ localIdents)
  mapM_ putStrLn allCandidates
```

`peekSource` turns C string input into a Haskell `String` while honoring optional length semantics. It supports null pointers, null-terminated strings, and explicit-length buffers.

```haskell
peekSource :: CString -> CSize -> IO String
peekSource ptr len
  | ptr == nullPtr = pure ""
  | len == 0       = peekCString ptr
  | otherwise      = peekCStringLen (ptr, fromIntegral len)
```

5.2 Snippet Analysis and Parsing Module

This module implements the syntactic decision procedure for interactive cells. In notebook use, one cell may mix declarations and executable expressions, so the runtime needs a principled way to infer intent without forcing users into a strict command grammar.

Operationally, the module blends parser-backed checks with lightweight lexical heuristics so that correctness and responsiveness coexist. Parser calls provide semantic authority for definition/expression validity, while token scanning supplies fast candidate extraction for completion.

The core abstraction is split search: for snippet s with line indices i, we test partitions (d_i, r_i) from largest prefix to smallest and accept the first valid plan. This greedy strategy approximates user intent in the common case where declarations precede executable tails.

```haskell
module Repl.Analysis (
  canParseDefinition,
  canParseExpression,
  extractDefinitionNames,
  SplitPlan(..),
  firstValidSplitPlan,
  trimWs,
  matchKeywordPrefix,
  completionCandidates,
  reservedIds,
  isIncomplete
) where
import qualified Prelude ()
import MHSPrelude
import Data.List (nub)
import Data.Maybe (listToMaybe, mapMaybe)
import MicroHs.Parse (parse, pExprTop, pTopModule)
import MicroHs.Expr (EModule(..), EDef(..), patVars)
import MicroHs.Ident (Ident, SLoc(..))
import MicroHs.Lex (Token(..), lex)
import Repl.Error
import Repl.Utils (ensureTrailingNewline, buildModule, indent, allwsLine, isws)
```

`SplitPlan` describes how a snippet can be interpreted by the executor. It captures either definition-only input or a split where definitions are followed by a runnable expression.

```haskell
data SplitPlan
  = SplitDefineOnly String
  | SplitDefineThenRun String String
```

`canParseDefinition` checks whether a snippet is a valid set of top-level definitions. It wraps the snippet into a synthetic module and delegates parsing to MicroHs.

```haskell
canParseDefinition :: String -> Bool
canParseDefinition snippet =
  case parse pTopModule "<xhaskell-define>" (buildModule (ensureTrailingNewline
↪  snippet)) of
    Right _ -> True
    Left _  -> False
```

`canParseExpression` checks whether a snippet is a valid expression. It parses with `pExprTop` and maps parser success to a boolean.

```haskell
canParseExpression :: String -> Bool
canParseExpression snippet =
  case parse pExprTop "<xhaskell-expr>" snippet of
```

```
    Right _ -> True
    Left _  -> False
```

`extractDefinitionNames` extracts bound identifiers from definition source. It parses as a module and delegates AST traversal to `definitionNamesFromModule`.

```
extractDefinitionNames :: String -> Either ReplError [Ident]
extractDefinitionNames snippet =
  case parse pTopModule "<xhaskell-define-names>" (buildModule snippet) of
    Left err -> Left (ReplParseError err)
    Right mdl -> Right (definitionNamesFromModule mdl)
```

`definitionNamesFromModule` collects identifiers introduced by each top-level AST node. It pattern matches `EDef` constructors and flattens all names into a single list.

```
definitionNamesFromModule :: EModule -> [Ident]
definitionNamesFromModule (EModule _ _ defs) = concatMap definitionNames defs
  where
    definitionNames def =
      case def of
        Data lhs _ _        -> [lhsIdent lhs]
        Newtype lhs _ _     -> [lhsIdent lhs]
        Type lhs _          -> [lhsIdent lhs]
        Fcn name _          -> [name]
        PatBind pat _       -> patVars pat
        Sign names _        -> names
        KindSign name _     -> [name]
        Pattern lhs _ _     -> [lhsIdent lhs]
        Class _ lhs _ _     -> [lhsIdent lhs]
        DfltSign name _     -> [name]
        ForImp _ _ name _   -> [name]
        Infix _ names       -> names
        _                   -> []
    lhsIdent (name, _) = name
```

`tokenize` produces lexical tokens from a source snippet. It uses MicroHs lexer with a default synthetic source location.

```
tokenize :: String -> [Token]
tokenize = lex (SLoc "" 1 1)
```

`extractIdents` filters tokens down to identifier names only. It uses list comprehension over `TIdent` tokens and discards everything else.

```
extractIdents :: [Token] -> [String]
extractIdents ts = [s | TIdent _ _ s <- ts]
```

`completionCandidates` derives potential completion strings from source text. It composes tokenization and identifier extraction.

```
completionCandidates :: String -> [String]
completionCandidates = extractIdents . tokenize
```

`reservedIds` is the baseline keyword set offered for completion. It is a static list merged with user-defined identifiers.

```
reservedIds :: [String]
reservedIds =
  [ "case", "class", "data", "default", "deriving", "do", "else"
  , "foreign", "if", "import", "in", "infix", "infixl", "infixr"
  , "instance", "let", "module", "newtype", "of", "then", "type"
  , "where", "_"
  ]
```

`isIncomplete` heuristically checks whether a snippet is unfinished. It tracks bracket nesting and quote states in one pass.

```
isIncomplete :: String -> Bool
isIncomplete s = go [] s
  where
    go st [] = not (null st)
    go st (c:cs)
      | c `elem` "([{" = go (c:st) cs
      | c `elem` ")]}" = case st of
          [] -> False
          (x:xs) -> if match x c then go xs cs else False
      | c == '"' = goString st cs
      | c == '\'' = goChar st cs
      | otherwise = go st cs
    goString st [] = True
    goString st ('\\':'"':cs) = goString st cs
    goString st ('"':cs) = go st cs
    goString st (_:cs) = goString st cs
    goChar st [] = True
    goChar st ('\\':'\'':cs) = goChar st cs
    goChar st ('\'':cs) = go st cs
    goChar st (_:cs) = goChar st cs
    match '(' ')' = True
    match '[' ']' = True
    match '{' '}' = True
    match _ _ = False
```

`firstValidSplitPlan` returns the first acceptable split strategy for a snippet. It tries split points from longest prefix to shortest and picks the first successful classification.

```
firstValidSplitPlan :: String -> String -> Maybe SplitPlan
firstValidSplitPlan currentDefs snippet =
  let snippetLines = lines (ensureTrailingNewline snippet)
  in listToMaybe (mapMaybe (classifySplit currentDefs snippetLines) [length
↪  snippetLines, length snippetLines - 1 .. 0])
```

`classifySplit` evaluates one split point and classifies it as define-only, define-then-run, or invalid.

It validates definition prefixes and then checks candidate run segments as expression or `do`-wrapped expression.

```
classifySplit :: String -> [String] -> Int -> Maybe SplitPlan
classifySplit currentDefs snippetLines splitIndex
  | not (canParseDefinition candidateDefs) = Nothing
  | all allwsLine runLines = Just (SplitDefineOnly defPart)
  | canParseExpression runPart = Just (SplitDefineThenRun defPart runPart)
  | canParseExpression doRunPart = Just (SplitDefineThenRun defPart doRunPart)
  | otherwise = Nothing
  where
    (defLines, runLines) = splitAt splitIndex snippetLines
    defPart = unlines defLines
    runPart = unlines (dropWhileEndLocal allwsLine runLines)
    doRunPart = "do\n" ++ indent runPart
    candidateDefs = currentDefs ++ defPart
```

`dropWhileEndLocal` removes a suffix that matches a predicate. It uses reverse-drop-reverse to avoid depending on library availability differences.

```
dropWhileEndLocal :: (a -> Bool) -> [a] -> [a]
dropWhileEndLocal f = reverse . dropWhile f . reverse
```

`trimWs` trims leading and trailing whitespace. It reuses `isws` and reverse-based stripping for both ends.

```
trimWs :: String -> String
trimWs = dropWhile isws . reverse . dropWhile isws . reverse
```

`matchKeywordPrefix` recognizes command keywords like `:type` only when followed by a word boundary. It prevents accidental partial matches such as `:typed`.

```
matchKeywordPrefix :: String -> String -> Maybe String
matchKeywordPrefix keyword snippet
  | startsWith keyword snippet && hasBoundary keyword snippet =
      Just (drop (length keyword) snippet)
  | otherwise = Nothing
  where
    hasBoundary key s =
      let rest = drop (length key) s
      in null rest || isws (head rest)
```

`startsWith` checks whether text begins with a specific prefix. It uses length-based `take` equality for a small, explicit helper.

```
startsWith :: String -> String -> Bool
startsWith prefix s = take (length prefix) s == prefix
```

5.3 Compilation and Execution Translation Module

This module is the compilation core of the REPL pipeline. It turns synthesized source text into executable internal form and then resolves selected bindings for evaluation against cached module context.

Conceptually, it defines two transformations: a compile step $\Gamma : (C, src) \mapsto (cmdl, C', \Sigma')$ and an execution step $\rho : (C', cmdl, ident) \mapsto IO()$. The first advances cache and symbols; the second realizes runtime effect by translating the chosen identifier through the accumulated binding map.

By isolating these operations, the surrounding executor can compose behaviors (define, run, probe) without duplicating compiler plumbing, and performance work can focus on cache evolution rather than call-site rewrites.

```
module Repl.Compiler (
  compileModule,
  runAction,
  runResultName,
  runResultIdent,
  runBlock
) where
import qualified Prelude ()
import MHSPrelude
import Data.List (foldl')
import Data.Maybe (isJust)
import MicroHs.Compile (Cache, compileModuleP, compileToCombinators)
import MicroHs.CompileCache (cachedModules)
import MicroHs.SymTab (SymTab, Entry(..), stLookup, stEmpty)
import MicroHs.Desugar (LDef)
import MicroHs.Exp (Exp(Var))
import MicroHs.Expr (EModule(..), EDef(..), ImpType(..))
import MicroHs.Ident (Ident, mkIdent, qualIdent, unQualIdent)
import qualified MicroHs.IdentMap as IMap
import MicroHs.Parse (parse, pTopModule)
import MicroHs.StateIO (runStateIO)
import MicroHs.TypeCheck (TModule(..), tBindingsOf, Symbols)
import MicroHs.Translate (TranslateMap, translateMap, translateWithMap)
import MicroHs.Builtin (builtinMdl)
import Unsafe.Coerce (unsafeCoerce)
import Repl.Context
import Repl.Error
import Repl.Utils
```

`runResultName` is the stable binding name used for generated run wrappers. Using a fixed symbol keeps downstream execution logic simple and deterministic.

```
runResultName :: String
runResultName = "runResult"
```

`runResultIdent` is the identifier form of `runResultName`. It is precomputed to avoid repeated conversions when invoking compiled actions.

```
runResultIdent :: Ident
runResultIdent = mkIdent runResultName
```

`runBlock` wraps a statement in a synthetic `IO ()` binding named `runResult`. The wrapper ensures execution uses `_printOrRun` consistently.

```haskell
runBlock :: String -> String
runBlock stmt = unlines
  [ runResultName ++ " :: IO ()"
  , runResultName ++ " = _printOrRun ("
  , indent stmt
  , "  )"
  ]
```

`compileModule` parses and compiles source in the current REPL context. It runs the MicroHs compile pipeline with existing cache and returns compiled module, updated cache, and symbols.

```haskell
compileModule :: ReplCtx -> String -> IO (Either ReplError (TModule [LDef], Cache, Symbols))
compileModule ctx src =
  case parse pTopModule "<repl>" src of
    Left perr -> pure (Left (ReplParseError perr))
    Right mdl -> do
      r <- runStateIO (compileModuleP (rcFlags ctx) ImpNormal mdl) (rcCache ctx)
      let (((dmdl, syms, _, _, _), _), cache') = unsafeCoerce r
          cmdl = compileToCombinators dmdl
      pure (Right (cmdl, cache', syms))
```

`runAction` executes one compiled identifier in IO. It builds a translation map from cached modules, resolves the target variable, and runs it as an `IO ()` action.

```haskell
runAction :: Cache -> TModule [LDef] -> Ident -> IO (Either ReplError ())
runAction cache cmdl ident = do
  let defs      = tBindingsOf cmdl
      baseMap   = withBuiltinAliases $ translateMap $ concatMap tBindingsOf (cachedModules cache)
      actionAny = translateWithMap baseMap (defs, Var (qualIdent (tModuleName cmdl) ident))
      action    = unsafeCoerce actionAny :: IO ()
  action
  pure (Right ())
```

`withBuiltinAliases` augments a translation map with qualified builtin aliases. It inserts aliases only when they are missing to avoid overriding existing entries.

```haskell
withBuiltinAliases :: TranslateMap -> TranslateMap
withBuiltinAliases mp = foldl' addAlias mp (IMap.toList mp)
  where
    addAlias acc (ident, val) =
      let aliasIdent = qualIdent builtinMdl (unQualIdent ident)
      in if aliasIdent == ident || isJust (IMap.lookup aliasIdent acc)
           then acc
           else IMap.insert aliasIdent val acc
```

5.4 Persistent Context and Definition Storage Module

This module defines the persistent memory model of the notebook session. It is the place where a sequence of user cells is transformed into a coherent evolving context rather than a collection of unrelated

compile requests.

At a formal level, we treat context as $ReplCtx = (F, C, D, \Sigma)$, where F are compiler flags and paths, C is compile cache, D is ordered stored definitions, and Σ is symbol information for inspection and completion. All higher-level REPL operations are meaningful only through controlled updates to this tuple.

The key policy encoded here is redefinition shadowing. If a new snippet introduces names N, then old stored definitions that bind any element of N are removed before append, ensuring that the latest binding governs future execution while unrelated definitions remain stable.

```haskell
module Repl.Context (
  ReplCtx(..),
  StoredDef(..),
  initialCtx,
  defsSource,
  currentDefsSource,
  moduleSourceWith,
  moduleFromDefs,
  appendDefinition,
  ReplHandle
) where
import qualified Prelude ()
import MHSPrelude
import System.Environment (lookupEnv)
import Data.IORef
import Data.List (nub)
import Foreign.StablePtr (StablePtr)
import MicroHs.Flags (Flags, defaultFlags, paths)
import MicroHs.Compile (Cache, emptyCache)
import MicroHs.SymTab (SymTab, stEmpty)
import MicroHs.TypeCheck (Symbols)
import MicroHs.Ident (Ident)
import Repl.Error
import Repl.Utils
import Repl.Analysis (extractDefinitionNames)
```

`ReplHandle` is the FFI-stable pointer type used to hold mutable REPL state. It wraps `IORef ReplCtx` so updates can happen across calls while keeping one handle.

```haskell
type ReplHandle = StablePtr (IORef ReplCtx)
```

`StoredDef` stores one user definition snippet and the identifiers it binds. This pairing lets the system remove obsolete definitions when names are redefined.

```haskell
data StoredDef = StoredDef
  { sdCode :: String
  , sdNames :: [Ident]
  }
```

`ReplCtx` is the full mutable runtime state for the REPL session. It carries compile flags, incremental cache, stored definitions, and symbol tables.

```haskell
data ReplCtx = ReplCtx
  { rcFlags :: Flags
  , rcCache :: Cache
  , rcDefs  :: [StoredDef]
  , rcSyms  :: Symbols
  }
```

`initialCtx` builds the starting context from runtime directory and environment paths. It extends default MicroHs search paths with `MHS_LIBRARY_PATH` and initializes empty state containers.

```haskell
initialCtx :: String -> IO ReplCtx
initialCtx dir = do
  let flags = defaultFlags dir
  mpath <- lookupEnv "MHS_LIBRARY_PATH"
  let rpath = maybe "." id mpath
      extra = splitColon rpath
      rcFlags = flags { paths = paths flags ++ extra }
  pure ReplCtx { rcFlags = rcFlags, rcCache = emptyCache, rcDefs = [], rcSyms =
↪  (stEmpty, stEmpty) }
```

`defsSource` concatenates raw source for a list of stored definitions. Order is preserved so later recompiles mirror user input sequence.

```haskell
defsSource :: [StoredDef] -> String
defsSource = concatMap sdCode
```

`currentDefsSource` extracts concatenated definition source from the active context. It is a small projection helper used by parse and execution flows.

```haskell
currentDefsSource :: ReplCtx -> String
currentDefsSource = defsSource . rcDefs
```

`moduleSourceWith` builds a complete module by appending extra source to current defs. It relies on `buildModule` to inject required header/import scaffolding.

```haskell
moduleSourceWith :: ReplCtx -> String -> String
moduleSourceWith ctx extra = buildModule (currentDefsSource ctx ++ extra)
```

`moduleFromDefs` builds a complete module from an explicit definition list. It is used when compiling updated definition sets.

```haskell
moduleFromDefs :: [StoredDef] -> String
moduleFromDefs defs = buildModule (defsSource defs)
```

`stripRedefined` drops prior stored definitions that conflict with newly introduced names. A stored definition is kept only if all its bound names are absent from the replacement set.

```haskell
stripRedefined :: [StoredDef] -> [Ident] -> [StoredDef]
stripRedefined defs [] = defs
stripRedefined defs names = filter noOverlap defs
  where
    noOverlap def = all (`notElem` names) (sdNames def)
```

`appendDefinition` parses new snippet names, removes conflicting older defs, and appends new definition metadata. This keeps the latest definition for each name while preserving unrelated definitions.

```haskell
appendDefinition :: ReplCtx -> String -> Either ReplError [StoredDef]
appendDefinition ctx snippet =
    case extractDefinitionNames snippet of
        Left err -> Left err
        Right names ->
            let uniqueNames = nub names
                retainedDefs = stripRedefined (rcDefs ctx) uniqueNames
            in Right (retainedDefs ++ [StoredDef snippet uniqueNames])
```

5.5 Error Classification and Rendering Module

This module defines the error algebra shared by parser, compiler, runtime, and foreign interface boundaries. The objective is not only to report failure, but to preserve enough stage information so that downstream layers can respond coherently.

We can regard `ReplError` as a tagged union over pipeline phases: $E = E_{\text{parse}} + E_{\text{compile}} + E_{\text{runtime}}$. This decomposition keeps operational diagnostics interpretable while still allowing uniform transport through `Either ReplError a` and C-string-based FFI messages.

```haskell
module Repl.Error (
  ReplError(..),
  prettyReplError
) where
import qualified Prelude ()
import MHSPrelude
```

`ReplError` classifies failures by pipeline stage. Separating parse, compile, and runtime failures makes error handling and messaging clearer across module boundaries.

```haskell
data ReplError
  = ReplParseError String
  | ReplCompileError String
  | ReplRuntimeError String
  deriving (Eq, Show)
```

`prettyReplError` converts structured errors into plain, user-facing text. It prepends a stable stage prefix while preserving the original detail payload.

```haskell
prettyReplError :: ReplError -> String
prettyReplError e =
  case e of
```

```
    ReplParseError s   -> "Parse error: "   ++ s
    ReplCompileError s -> "Compile error: " ++ s
    ReplRuntimeError s -> "Runtime error: " ++ s
```

5.6 Runtime Command Execution Module

This module is the operational orchestrator of the REPL. It composes analysis, compilation, execution, and symbol lookup into user-visible commands while preserving a single evolving context across requests.

Its structure can be read as a transition system over context: $\delta : (ctx, input) \mapsto Either\ ReplError\ ctx'$. Specialized transitions (`replDefine`, `replRun`, `replExecute`, `replIsComplete`) share this shape so they can be chained uniformly and reasoned about as one control algebra.

This design keeps policy explicit: parsing strategy lives in `Repl.Analysis`, state storage in `Repl.Context`, and compilation mechanics in `Repl.Compiler`, while this module remains the place where end-user intent is interpreted and dispatched.

```
module Repl.Executor (
  replDefine,
  replRun,
  replExecute,
  replInspect,
  replIsComplete
) where
import qualified Prelude ()
import MHSPrelude
import Control.Applicative ((<|>))
import Data.Maybe (isJust)
import MicroHs.Ident (mkIdent)
import MicroHs.SymTab (stLookup, Entry(..))
import MicroHs.Expr (showExpr)
import MicroHs.TypeCheck (Symbols)
import System.IO (putStrLn)
import Repl.Context
import Repl.Error
import Repl.Utils
import Repl.Analysis
import Repl.Compiler
```

`MetaCommand` represents REPL command directives that are not plain code execution. It currently covers `:type` and `:kind` command payloads.

```
data MetaCommand
  = TypeOfExpr String
  | KindOfType String
```

`Completion` models the cell completeness state consumed by frontends. It distinguishes complete, incomplete, and invalid snippets.

```
data Completion
  = Complete
```

```
  | Incomplete
  | Invalid
```

`andThen` composes two `IO (Either ...)` steps in a left-biased manner. It forwards `Left` immediately and applies the next action only on `Right`.

```
andThen
  :: IO (Either ReplError a)
  -> (a -> IO (Either ReplError b))
  -> IO (Either ReplError b)
andThen act next = do
  result <- act
  case result of
    Left err -> pure (Left err)
    Right val -> next val
```

`right` lifts a pure value into the `IO (Either ReplError ...)` success shape. It is a tiny helper used to reduce repetition in success branches.

```
right :: a -> IO (Either ReplError a)
right = pure . Right
```

`replDefine` appends a definition snippet and recompiles the full definition module. On success it updates stored definitions, compile cache, and symbol tables.

```
replDefine :: ReplCtx -> String -> IO (Either ReplError ReplCtx)
replDefine ctx snippet =
  case appendDefinition ctx (ensureTrailingNewline snippet) of
    Left err -> pure (Left err)
    Right defsWithNew ->
      compileModule ctx (moduleFromDefs defsWithNew) `andThen`
        \(_, cache', syms') ->
          right ctx{ rcDefs = defsWithNew, rcCache = cache', rcSyms = syms' }
```

`replRun` executes a statement/expression in the current context without mutating stored definitions. It compiles a generated run block and executes `runResultIdent`.

```
replRun :: ReplCtx -> String -> IO (Either ReplError ReplCtx)
replRun ctx stmt =
  compileModule ctx (moduleSourceWith ctx (runBlock stmt)) `andThen`
    \(cmdl, cache', syms') ->
      runAction cache' cmdl runResultIdent `andThen`
        \() -> right ctx{ rcCache = cache', rcSyms = syms' }
```

`replIsComplete` classifies snippet completeness for notebook frontend behavior. It checks empty input, valid splitability, and structural incompleteness heuristics.

```
replIsComplete :: ReplCtx -> String -> IO String
replIsComplete ctx snippet = pure (toText completion)
  where
    completion
```

```
      | all isws snippet = Complete
      | hasValidSplit = Complete
      | isIncomplete snippet = Incomplete
      | otherwise = Invalid
    hasValidSplit = isJust (firstValidSplitPlan (currentDefsSource ctx) snippet)
    toText state =
      case state of
        Complete -> "complete"
        Incomplete -> "incomplete"
        Invalid -> "invalid"
```

`replInspect` resolves a symbol from value/type tables and renders its signature. It tries value scope first, then type scope, and returns a runtime error if absent.

```
replInspect :: ReplCtx -> String -> IO (Either ReplError String)
replInspect ctx name = do
  let ident = mkIdent name
      (typeTable, valueTable) = rcSyms ctx
      valueHit = lookupRendered "value" ident valueTable name
      typeHit = lookupRendered "type" ident typeTable name
      notFound = ReplRuntimeError ("Identifier not found: " ++ name)
  pure (maybe (Left notFound) Right (valueHit <|> typeHit))
```

`replExecute` is the main execution entrypoint for snippets. It routes meta commands first, then tries split-plan execution strategies.

```
replExecute :: ReplCtx -> String -> IO (Either ReplError ReplCtx)
replExecute ctx snippet
  | Just cmd <- parseMetaCommand snippet = runMetaCommand ctx cmd
replExecute ctx snippet =
  case firstValidSplitPlan (currentDefsSource ctx) snippet of
    Nothing -> pure (Left (ReplParseError "unable to parse snippet"))
    Just (SplitDefineOnly defPart) -> replDefine ctx defPart
    Just (SplitDefineThenRun defPart runPart) -> defineThenRun ctx defPart runPart
```

`runMetaCommand` dispatches parsed meta commands to their handlers. It keeps branching logic centralized for command extensions.

```
runMetaCommand :: ReplCtx -> MetaCommand -> IO (Either ReplError ReplCtx)
runMetaCommand ctx cmd =
  case cmd of
    TypeOfExpr expr -> replTypeOf ctx expr
    KindOfType ty -> replKindOf ctx ty
```

`replTypeOf` infers and prints the type of an expression via a probe definition. It generates a temporary binding, compiles it, and reads the inferred value symbol.

```
replTypeOf :: ReplCtx -> String -> IO (Either ReplError ReplCtx)
replTypeOf ctx expr =
  runTypedProbe
    ctx
    ":type expects an expression"
```

```
    "failed to infer expression type"
    trimmedExpr
    src
    inferType
  where
    trimmedExpr = trimWs expr
    probeName = "xhReplTypeProbe"
    probeIdent = mkIdent probeName
    probeDef = unlines
      [ probeName ++ " = ("
      , indent trimmedExpr
      , "  )"
      ]
    src = moduleSourceWith ctx probeDef
    inferType (_, valueTable) =
      case stLookup "value" probeIdent valueTable of
        Right (Entry _ sigma) -> Just (showExpr sigma)
        Left _ -> Nothing
```

`replKindOf` infers and prints the kind of a type expression. It compiles a temporary type alias probe and reads the inferred type symbol.

```
replKindOf :: ReplCtx -> String -> IO (Either ReplError ReplCtx)
replKindOf ctx ty =
  runTypedProbe
    ctx
    ":kind expects a type"
    "failed to infer type kind"
    trimmedType
    src
    inferKind
  where
    trimmedType = trimWs ty
    probeName = "XhReplKindProbe"
    probeIdent = mkIdent probeName
    probeDef = "type " ++ probeName ++ " = " ++ trimmedType ++ "\n"
    src = moduleSourceWith ctx probeDef
    inferKind (typeTable, _) =
      case stLookup "type" probeIdent typeTable of
        Right (Entry _ kind) -> Just (showExpr kind)
        Left _ -> Nothing
```

`parseMetaCommand` parses command prefixes and returns a structured command value. It trims input and applies boundary-aware keyword matching.

```
parseMetaCommand :: String -> Maybe MetaCommand
parseMetaCommand raw =
  parseWith ":type" TypeOfExpr snippet <|> parseWith ":kind" KindOfType snippet
  where
    snippet = trimWs raw
    parseWith keyword ctor s = ctor <$> matchKeywordPrefix keyword s
```

`defineThenRun` executes a split snippet by defining first and running second. It composes `replDefine` and `replRun` through `andThen`.

```haskell
defineThenRun :: ReplCtx -> String -> String -> IO (Either ReplError ReplCtx)
defineThenRun ctx defPart runPart = replDefine ctx defPart `andThen` \ctx' ->
↪  replRun ctx' runPart
```

`runTypedProbe` is the shared engine behind `:type` and `:kind` probing. It validates input, compiles probe source, extracts inferred text, prints it, and keeps context unchanged on success.

```haskell
runTypedProbe
  :: ReplCtx
  -> String
  -> String
  -> String
  -> String
  -> (Symbols -> Maybe String)
  -> IO (Either ReplError ReplCtx)
runTypedProbe ctx emptyInputError inferError shown src infer
  | null shown = pure (Left (ReplRuntimeError emptyInputError))
  | otherwise =
      compileModule ctx src `andThen` \(_, _, syms) ->
        case infer syms of
          Nothing -> pure (Left (ReplRuntimeError inferError))
          Just inferred -> do
            putStrLn ("> " ++ shown ++ " :: " ++ inferred)
            right ctx
```

`lookupRendered` formats one symbol-table hit as inspect text. It performs scoped lookup and renders `name :: signature` when successful.

```haskell
lookupRendered scope ident table shown =
  case stLookup scope ident table of
    Right (Entry _ sigOrKind) -> Just (shown ++ " :: " ++ showExpr sigOrKind)
    Left _ -> Nothing
```

5.7 Shared Utility Functions Module

This module contains the small, deterministic utilities that make the larger REPL pipeline stable. Although each function is simple, together they enforce textual normalization and module-shape invariants relied on by parsing, split planning, and execution code generation.

The guiding principle is to keep low-level string and list transformations explicit and dependency-light. In practice, these helpers define canonicalization laws such as newline completion, indentation shaping, and module-header injection, so higher layers can assume normalized source structure.

```haskell
module Repl.Utils (
  splitColon,
  indent,
  ensureTrailingNewline,
  isws,
  allwsLine,
  dropWhileEnd,
  buildModule,
  moduleHeader
```

```
) where
import qualified Prelude ()
import MHSPrelude
import Data.List (reverse, dropWhile)
```

`splitColon` splits a colon-separated path string into list elements. It uses recursive `break` so behavior remains simple and dependency-free.

```
splitColon :: String -> [String]
splitColon s = case break (== ':') s of
  (a, [])     -> [a]
  (a, _:rest) -> a : splitColon rest
```

`indent` prefixes each line with two spaces. This is used when embedding user statements into generated wrapper definitions.

```
indent :: String -> String
indent = unlines . map ("  " ++) . lines
```

`ensureTrailingNewline` guarantees source ends with a newline. It normalizes empty input to a single newline and appends one when missing.

```
ensureTrailingNewline :: String -> String
ensureTrailingNewline s
  | null s          = "\n"
  | last s == '\n' = s
  | otherwise       = s ++ "\n"
```

`isws` checks whether a character is recognized as whitespace in this module. The predicate is intentionally explicit and reused by completion/parse helpers.

```
isws :: Char -> Bool
isws c = c == ' ' || c == '\t' || c == '\n' || c == '\r'
```

`allwsLine` checks whether an entire line is whitespace. It applies `isws` to all characters in the line.

```
allwsLine :: String -> Bool
allwsLine = all isws
```

`dropWhileEnd` removes a trailing suffix matching a predicate. It uses reverse/dropWhile/reverse to avoid requiring extra library features.

```
dropWhileEnd :: (a -> Bool) -> [a] -> [a]
dropWhileEnd p = reverse . dropWhile p . reverse
```

`moduleHeader` is the fixed prelude section for synthetic REPL modules. It imports exactly what generated evaluation wrappers rely on.

```haskell
moduleHeader :: String
moduleHeader = unlines
  [ "module Inline where"
  , "import Prelude"
  , "import System.IO.PrintOrRun"
  , "import Data.Typeable"
  , "import Numeric"
  ]
```

`buildModule` combines `moduleHeader` with accumulated definitions. It creates the full source unit sent through the compile pipeline.

```haskell
buildModule :: String -> String
buildModule defs = moduleHeader ++ defs
```