Scratch-like Programming in Haskell

Using xeus-haskell MicroHS Console

Based on Scratch to Haskell Tutorial

February 21, 2026

Contents

1 Introduction

This document provides complete instructions for using Scratch-like programming concepts in the xeus-haskell MicroHS console. It integrates functional programming concepts with visual programming ideas familiar from Scratch.

Tip

Use the companion file `scratch_xeus_microhs.hs` which contains all the code examples from this guide, organized by sections for easy copy-pasting into the console.

1.1 What is xeus-haskell MicroHS Console?

The xeus-haskell MicroHS console is a minimal Haskell REPL (Read-Eval-Print Loop) environment available within Jupyter notebooks. It provides:

- Interactive Haskell evaluation
- Pure functional programming
- Limited but sufficient feature set for learning
- Integration with Jupyter notebooks

1.2 Getting Started

To begin using the xeus-haskell MicroHS console:

1. Visit `https://hubtizard.github.io/xeus-haskell/`. This will load JupyterLite and work-in-progress notebooks into your browser.
2. In the file browser, select `ScratchGuide/scratch_xeus_microhs.hs` and download it (right click on file name) and open with a text editor.
3. In the launcher window, click on **Console**.
4. In the console window, select the three vertical dots in the top right hand side and choose **Prompt to Left**. It may take a minute or two for the kernel to start (the circle next to the dots turns opaque)

You are ready to start copying and pasting code!

Important Warning

The MicroHS console is **NOT** the same as:

- Full GHCi (Glasgow Haskell Compiler interactive)
- Standalone MicroHS command-line tool
- Regular xeus-haskell notebook cells

2 Console Commands and Limitations

2.1 What Works

```
-- Basic arithmetic
2 + 3

-- Function definition
square x = x * x

-- Type synonyms
type Apple = Int

-- Lambda expressions
(\x -> x + 1) 5

-- List operations
map (*2) [1,2,3]
```

2.2 What Doesn't Work

Important Warning

The following commands are **NOT SUPPORTED**:

- `:load filename.hs` – Cannot load files
- `:info Name` – Information queries
- `import` statements – Limited library support
- `IORef` – No mutable state
- `System.Random` – No external libraries

2.3 What Works: Type Queries

The `:type` command **is available** in xeus-haskell MicroHS console for checking types:

```
x = 5::Apple

:type x
-- Output x  :: Int

f = (2+) :: Int -> Int -- an explicit function type signature

:type f
--Output Int -> Int

:type map
-- Output: map :: (a -> b) -> [a] -> [b] -- maybe written (a -> b) ->
   ([a] -> [b])) as -> is right associative the last set of brackets
   can be removed
```

Tip

Use `:type` to explore types as you learn! This helps you understand:

- What type a value has
- What type a function expects and returns
- How polymorphic functions work

2.4 Using Type Queries with :t

Since the `:type` command is available, you can directly check types:

```
-- Check value types
:type 5
:type 3.14
:type "Alice"

-- Check function types
:type (2+)
:type moveRight
:type map

-- Check complex expressions
:type moveRight initialState
:type repeatN 5 moveRight
```

2.5 Alternative: Using Explicit Annotations

You can also use explicit type annotations for clarity or documentation:

```
-- Annotate values
x = 5 :: Int
y = 3.14 :: Double
name = "Alice" :: String

-- Annotate functions
add2 = (2+) :: Int -> Int
square = \x -> x * x :: Int -> Int

-- Annotate state transformations
moveRightTyped = moveRight :: GameState -> GameState
```

Note

Type annotations are useful for:

- Documentation (explaining what types you expect)
- Catching errors early (mismatched types)
- Learning (seeing how types work)

2.6 Learning from Type Errors

Even with `:type` available, type errors are valuable learning tools:

```haskell
-- Try invalid operations to understand types
"hello" + 5          -- Error shows String vs Num mismatch
moveRight 5          -- Error shows GameState expected
initialState ++ []   -- Error shows GameState vs List

-- The error messages tell you the types!
```

Tip

Type errors are your friend! They tell you:

- What type was expected
- What type was actually provided
- Where the mismatch occurred

Use them to deepen your understanding of the type system.

2.7 Exploring Types Through Testing

Combine `:type` with testing to understand how functions work:

```haskell
-- Check the type first
:type add2
-- Output: add2 :: Int -> Int

-- Test with appropriate values
add2 5     -- Works; 5 :: Int
add2 "hi" -- Fails; String is not Int

-- For polymorphic functions
:type map
-- Output: map :: (a -> b) -> [a] -> [b]

-- Test to understand polymorphism
map add2 [1,2,3]   -- Works with Int list
map (++ "!") ["hello", "world"]  -- Works with String list
```

Note

The combination of `:type` and testing is powerful:

- Use `:type` to see what types are involved
- Test with actual values to see how they behave
- Compare expected vs actual results

3 Scratch to Haskell Mappings

3.1 Basic Mappings Table

Scratch Block	Haskell	Notes
`set my variable to 0`	`let x = 0`	Immutable binding
`change by 1`	`x + 1`	Returns new value
`my variable`	`x`	Read value
`repeat 10`	`repeatN 10 f x`	Custom function
`repeat until`	`repeatUntil cond f n x`	Custom function
`if then`	`if cond then a else b`	Standard conditional
`goto x:0 y:0`	`gotoXY 0 0 state`	Update state
`move 10 steps`	`moveSteps 10 state`	Transform state

Table 1: Scratch to Haskell Block Mappings

4 Working Without File Loading

4.1 The Copy-Paste Method

Since `:load` doesn't work, use the copy-paste method:

1. Open the `.hs` file in a text editor
2. Copy a section of code
3. Paste into the MicroHS console
4. Press Enter to evaluate
5. Repeat for each section

Tip

Work incrementally! Define one function at a time and test it before moving on.

4.2 Example: Building a Game State

4.2.1 Step 1: Define the Data Type

```
data GameState = GameState {
    xPos :: Int,
    yPos :: Int,
    score :: Int
} deriving Show
```

Copy and paste this into the console.

4.2.2 Step 2: Create Initial State

```
initialState = GameState 0 0 0
```

4.2.3 Step 3: Define Movement Functions

```
moveRight state = state { xPos = xPos state + 10 }
moveLeft state = state { xPos = xPos state - 10 }
moveUp state = state { yPos = yPos state + 10 }
moveDown state = state { yPos = yPos state - 10 }
```

4.2.4 Step 4: Test the Functions

```
initialState
moveRight initialState
moveUp (moveRight initialState)
```

5 Complete Examples

5.1 Example 1: Curried Functions

From Gattegno's pedagogy: "2 apples plus 3 apples equals 5 apples."

```
-- Define custom types
type Apple = Int
type Pen = Int

-- Create values
twoApples = 2 :: Apple
threeApples = 3 :: Apple
fiveApples = twoApples + threeApples

-- Test
fiveApples

-- Curried function
add2 = (2+)
add2 3
add2 10

-- The plus function
plus x y = x + y
plus 2 3
plus2 = plus 2
plus2 5
```

5.2 Example 2: Function Composition

```
-- Define operations
add2 = (2+)
add3 = (3+)
add5 = add3 . add2

-- Test composition
add5 0
add5 10

-- Verify commutativity
path1 = (3+) . (2+)
path2 = (2+) . (3+)
path3 = (5+)

path1 0 == path2 0 && path2 0 == path3 0
```

5.3 Example 3: Repeat Pattern

```haskell
-- Define repeat function
repeatN 0 f x = x
repeatN n f x = repeatN (n-1) f (f x)

-- Use with game state
data State = State { x :: Int, y :: Int } deriving Show
init = State 0 0
right (State x y) = State (x+10) y

-- Repeat movement 5 times
repeatN 5 right init
```

5.4 Example 4: Repeat Until

```haskell
-- Define repeat until
repeatUntil cond f maxIter x =
    if maxIter == 0 then x
    else if cond x then x
    else repeatUntil cond f (maxIter-1) (f x)

-- Use it
data Counter = Counter { count :: Int } deriving Show
inc (Counter n) = Counter (n+1)
atTen (Counter n) = n >= 10

repeatUntil atTen inc 100 (Counter 0)
```

5.5 Example 5: Rod Arithmetic (Gattegno)

Based on Cuisenaire rods:

```haskell

-- Define rod values
data Colour = White
            | Red
            | Green
            | Purple
            | Yellow
            | DarkGreen
            | Black
            | Brown
            | Blue
            | Orange
            deriving (Show,Eq, Ord, Enum)

-- Define rod lengths (measured with White)
rodLength:: Colour -> Int
rodLength colour = 1 + fromEnum colour --- because we derive Colour
   from the Enum type class we can map the data constructors to the
   integers [0..10] using fromEnum

-- Calculate the length of a train
type Train = [Colour]
rodSum :: Train -> Int
```

```
rodSum colours = sum [rodLength c | c <- colours ] -- uses a list
    comprehension (like a set generator in mathematics)

-- Check the type signature of rodSum

-- Return the data constructor with a given length
whichRod :: Int -> Colour
whichRod 1 = White
whichRod n = succ (whichRod (n-1)) -- uses successor function from
    derived type class Ord

-- Calculate the rod equivalent in length to a train such as [Yellow,
    White] using nested function application
```

6 Advanced Patterns

6.1 Higher-Order Functions

```
-- Map: apply function to each element
map (2+) [1,2,3,4,5]
map (*3) [1,2,3,4,5]

-- Filter: keep elements matching condition
filter even [1,2,3,4,5,6]
filter (>3) [1,2,3,4,5,6]

-- Fold: combine elements
foldr (+) 0 [1,2,3,4,5]
foldr (*) 1 [1,2,3,4,5]
```

6.2 Pattern Matching

```
-- Factorial with pattern matching
ffac 0 = 1; fac n = n * fac (n-1)
fac 5

-- Fibonacci
fib 0 = 0; fib 1 = 1; fib n = fib (n-1) + fib (n-2)
fib 10

-- List sum
sumList [] = 0; sumList (x:xs) = x + sumList xs
sumList [1,2,3,4,5]
```

6.3 List Comprehensions

```
-- Generate lists
[x * 2 | x <- [1..10]]
[x | x <- [1..20], even x]
[x + y | x <- [1,2,3], y <- [10,20,30]]

-- Scratch-like: create sprite positions
[(x, y) | x <- [0,10..50], y <- [0,10..50]]
```

7 Complete Game State Example

7.1 Full Implementation

```haskell
-- Step 1: Define the game state
data GameState = GameState {
    xPos :: Int,
    yPos :: Int,
    score :: Int
} deriving Show

-- Step 2: Initial state
 initialState = GameState 0 0 0

-- Step 3: Movement operations
 moveRight state = state { xPos = xPos state + 10 }
 moveLeft state = state { xPos = xPos state - 10 }
 moveUp state = state { yPos = yPos state + 10 }
 moveDown state = state { yPos = yPos state - 10 }

-- Step 4: Score operations
 addScore n state = state { score = score state + n }
 setScore n state = state { score = n }

-- Step 5: Position operations
 gotoXY x y state = state { xPos = x, yPos = y }

-- Step 6: Utility functions
 repeatN 0 f x = x
 repeatN n f x = repeatN (n-1) f (f x)

-- Step 7: Test scenarios
initialState
moveRight initialState
repeatN 3 moveRight initialState
addScore 10 (repeatN 3 moveRight initialState)
gotoXY 50 50 initialState
```

7.2 Command Sequences

```haskell
-- Execute a sequence of commands
 executeCommands [] state = state
 executeCommands (cmd:cmds) state =
    executeCommands cmds (cmd state)

-- Define command list
 commands = [moveRight, moveUp, addScore 5, moveRight]

-- Execute
executeCommands commands initialState
```

8 Troubleshooting

8.1 Common Errors and Solutions

Error	Solution
Parse error: unable to parse snippet	The command isn't supported. Use direct definitions instead of meta-commands.
Multi-line definition fails	Put everything on one line or use semicolons: `f 0 = 1; f n = n * f (n-1)`
Undefined value	Make sure to define the value with before using it.
Type mismatch	Check that your types are consistent. Add explicit type annotations if needed.

Table 2: Common Errors and Solutions

8.2 Using Type Queries for Debugging

Use `:t` to debug type-related issues:

```
-- When you get a type error, check each part
 result = moveRight (addScore 10 initialState)

-- Check each component
:type moveRight
:type addScore
:type addScore 10
:type initialState
:type addScore 10 initialState
:type moveRight (addScore 10 initialState)

-- This helps you find where types don't match
```

Tip

When debugging:

1. Break complex expressions into parts
2. Use `:type` on each part
3. Find where the type mismatch occurs
4. Fix that specific part

9 Pedagogical Notes

9.1 Key Concepts

9.1.1 Immutability

Unlike Scratch's variables, Haskell values are immutable:

- Scratch: Change the value in place
- Haskell: Create a new value with the change

```
-- Not: state.x = state.x + 10
-- But: state { xPos = xPos state + 10 }
```

9.1.2 Pure Functions

All functions are pure (no side effects):

- Same input always gives same output
- No hidden state changes
- Easier to reason about and test

9.1.3 Function Composition

Build complex operations as functions from simple functions:

```
-- Compose movements
 moveRightUp = moveUp . moveRight
moveRightUp initialState

-- Chain multiple operations
 complexMove = moveUp . moveRight . addScore 5
complexMove initialState
```

9.1.4 Nested Function Application

Apply a function to the result of another function (mathematics f(g(x)). – f $gxorf(gx)$

9.2 Threshold Concept: Functions

A function encapsulates three ideas:

1. **Operator**: The operation being performed
2. **Operand**: The input value
3. **Value**: The output result

In `(2+) 3 = 5`:

- Operator: `(2+)`
- Operand: `3`
- Value: `5`

10 Practice Exercises

10.1 Exercise 1: Square Movement

Create a function that moves in a square pattern:

```
-- Your solution here:
-- Define a function that applies:
-- right, up, left, down
-- to return to starting position
```

10.2 Exercise 2: Score Accumulator

Create a game that accumulates score based on position:

```
-- If x > 50, add 10 to score
-- If y > 50, add 5 to score
-- Write a function to check and update
```

10.3 Exercise 3: Custom Repeat

Write your own version of `repeatN` that counts up instead of down:

```
--  repeatUp f n x = ...
-- Should apply f exactly n times
```

11 Summary and Best Practices

11.1 Best Practices

1. **Work incrementally**: Define and test one function at a time
2. **Keep it simple**: Avoid complex multi-line definitions
3. **Add deriving Show**: So you can see your data structures
4. **Test frequently**: Verify each function before building on it
5. **Document types**: Add type annotations when possible

11.2 Key Takeaways

- The MicroHS console is for **interactive exploration**
- **Copy-paste** code instead of loading files
- Work with **pure functions** (no mutable state)
- Build complex operations from **simple compositions**
- Focus on **understanding concepts**, not memorising syntax

11.3 Going Further

Once comfortable with the MicroHS console:

1. Try more complex data structures
2. Explore recursive patterns
3. Learn about algebraic data types
4. Study function composition in depth
5. Move to full GHCi for advanced features

12 References

- Gattegno, C. (1970). *What We Owe Children*
- Mason, J. (2018). "How Early Is Too Early for Thinking Algebraically?" `https://bit.ly/Mason2018`
- xeus-haskell documentation
- Scratch programming language (scratch.mit.edu)