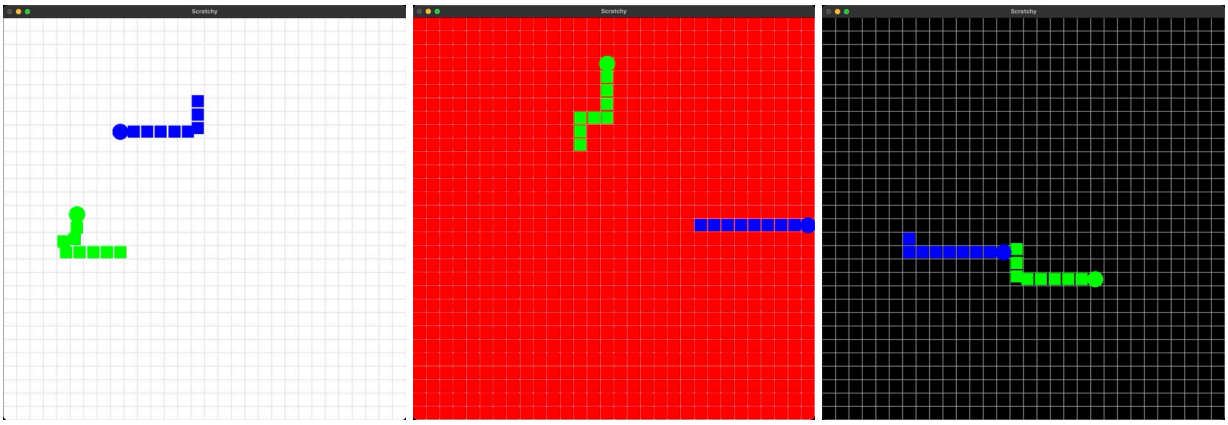
Project Description (15 Dec 2025–12 Jan, 2026)

DM580: Functional Programming, SDU



(a) Non-colliding snakes (b) Wall collision (c) Snake collision

1 Introduction

Project Goals The goals of this project are:

1. Implement a small domain-specific language, loosely inspired by the Scratch programming language. The language is embedded in Haskell, which means that the way you write programs in the domain-specific language, is by writing a Haskell program. The domain-specific language you will be implementing is summarized in Sect. 2 of this document.
2. Use the domain-specific language to implement a small game, loosely inspired by Snake, with two player-controlled snakes moving on a fixed grid (see screenshots above). The game is summarized in Sect. 3 of this document.

Project Deliverables The deliverables for this project are:

1. A .zip file containing an implementation of the project that satisfies the goals summarized above and in the rest of this document.
2. A project report that summarizes your implementation. The project report requirements are summarized in Sect. 4 of this document.

Both deliverables must be submitted via digitaleksamen.sdu.dk.

Project Prerequisites You should implement your project by building on the codebase provided in the file `scratchy.zip` on ItsLearning under 'Resources > Project'. The .zip file contains a `cabal` project with infrastructure for the Scratchy programming language that you will build. This infrastructure is summarized in Sect. 2.

2 Scratchy: A Domain-Specific Programming Language for Graphical Programming

This section summarizes the Scratchy programming language that you will implement.

2.1 Overview

The Scratchy language is given by a set of operations that affect the state of the graphical user interface (using Gloss) which renders the game state. This graphical user interface is given by a grid containing zero or more sprites. Sprites on the game grid move with a fixed speed towards a given target.

The file `app/Scratchy/Syntax.hs` contains the data type `SProg` which defines the syntax of the Scratchy embedded domain-specific language. This syntax declares operations for registering *event listeners*, setting and getting the color and targets of given sprites, creating new sprites, setting the background color of the grid, inspecting grid cells, and for timed actions. These operations are summarized below, in Sect. 2.3.

In summary, the `scratchy.zip` project that you are given contains the following files:

- `app/Scratchy/Syntax.hs` declares the syntax of the operations we summarize in Sect. 2.3.
- `app/Scratchy/World.hs` contains the module that declares the `World` data type, which represents the game state.
- `app/Scratchy.hs` contains the module that declares the main game loop of a Scratchy game. This module is currently configured to run the game given by the `circSprite` function, which is declared in the module `Example.CircleThatMoves`.
- `app/Example/CircleThatMoves.hs` contains a module that declares a simple Scratchy program, which creates a green circular sprite, which moves up when 'w' is pressed, and down when 's' is pressed.
- `app/Main.hs` the main file, which imports and runs the relevant `main` file from `app/Scratchy.hs`.

2.2 Missing Functionality

To implement your Scratchy programming language, you should implement the following missing functionality:

- `combine :: SProg () → SProg () → SProg ()` in file `app/Scratchy/Syntax.hs`. This function should sequentially compose two programs. That is, the program resulting from running `m1 `compose` m2` should be a program that has the same behavior as running first `m1`, and then `m2`.
- `runProg :: Game ()` in file `app/Scratchy/World.hs`. The purpose of this function is to run the main program of a `Game`. See Sect. 2.3 for a summary of how the syntax of Scratchy programs is to be interpreted.

You are allowed to make modifications to any existing file in the codebase in `scratchy.zip`, so long as you summarize what changes you made in your report.

2.3 Scratchy Operations and Their Run Semantics

This section summarizes the syntax and intended semantics of Scratchy programs.

The Pure Operation

- The operation `Pure x` has no observable effects on the grid or game state.

Sprite and Grid Actions

- `NewSprite c p k` creates a new sprite rendered as picture `p` at grid cell `c`, and applies the *continuation* `k` to an integer encoding a pointer (a *sprite pointer*, of type `SpritePtr`). Here, the continuation `k` represents the "remainder" of the program; i.e., what should happen after we have created a new sprite. Note that the `World` type (summarized below) contains a list of sprites. The type `SpritePtr` represents an index of this list.
- `SetColor s c m` sets the color of the sprite pointed to by `s` to color `c`, and subsequently runs the continuation `m`.
- `SetTarget s c m` sets cell `c` as the target of the sprite pointed to by `s`. The program `m` is the continuation of the operation.
- `GetTarget s k` gets the target cell of the sprite pointed to by `s`, and applies the continuation `k` to the resulting target cell.
- `SetBackgroundColor c m` sets the background color of the grid to color `c`. The program `m` is the continuation of the operation.
- `InspectCell c k` inspects the cell `c` on the grid, and then applies the continuation `k` to a value of type `InspectionResult`, defined as follows:

  ```
  data InspectionResult
    = HasBarrier | HasSprite | IsFree
  ```

  If the inspected cell `c` is outside of the grid, then `k` is passed the value `HasBarrier`. If the cell contains a sprite, then `k` is passed the value `HasSprite`. If the cell is within the grid, and contains no sprite, then `k` is passed the value `IsFree`.

Event Listening

- `OnKeyEvent k m m'`: This operation registers an event listener for when the key `k` is pressed. The program `m` represents the action to perform when the key is pressed; the program `m'` represents the "remainder" (continuation) of the program after the event listener has been registered. There can only be a single listener registered for each key at a time, and the most recently registered event listener takes precedence. For example, after running `OnKeyEvent k m0 (OnKeyEvent k m1 (Pure ()))`, if the key `k` is pressed, the program `m1` (*not* `m0`) must run.
- `OnTargetReached s g m`: This operation registers an event listener for when the sprite represented by the sprite pointer `s` reaches its declared target. The function `g` represents the action to perform when `s` reaches its target, while the program `m` represents the continuation of the program after the handler has been registered. There may be multiple listeners

registered for different sprites, but only a single listener for each sprite `s`. The most recently registered event listener takes precedence.

- `OnTargetUpdated s g m`: An event listener with similar behavior as `OnTargetReached`, except (1) it registers a listener for when the target of sprite `s` is updated; and (2) upon firing, the function `g` is applied to both the 'old' and 'new' target cell.
- `OnBarrierHit s g m`: An event listener with similar behavior as `OnTargetReached`, except it registers a listener for when a sprite tries to move outside of the bounds of the grid.

Timer Operation

- The operation `After d m m'` registers the program `m` to run after duration (i.e., number of game ticks) `d`.

2.4 Event Handling and Worlds

The infrastructure provided to you already invokes events in the intended way. Your main task is therefore to map the syntax of `Scratchy` programs onto sensible `Game` actions.

A `Game` is essentially a function that transforms the state of the world. The `World` type declares the following fields:

```
data World = World
  { bg         :: Color
  , sprites    :: ![Sprite]
  , keysHeld   :: ![Key]
  , prog       :: !(SProg ())         -- Current program
  , keyHdlrs   :: ![(Key, SProg ())] -- Key pressed
  , trHdlrs    :: ![( SpritePtr
                    , Cell
                      -> SProg () )] -- Target reached
  , tuHdlrs    :: ![( SpritePtr
                    , Cell
                      -> Cell
                      -> SProg () )] -- Target updated
  , bhHdlrs    :: ![( SpritePtr
                    , Cell
                      -> Cell
                      -> SProg () )] -- Barrier hit
  , timers     :: ![(Duration, SProg ())] -- Timers
  }
```

Here:

- `prog` is the game program, which is be executed to completion at each game tick.
- `keyHdlrs`,`trHdlrs`,`tuHdlrs`, and `bhHdlrs` contains listeners registered via, respectively, the operations `OnKeyEvent`, `OnTargetReached`, `OnTargetUpdated`, and `OnBarrierHit`.
- `trHdlrs` contains the target reached listeners, registered via the `OnTargetReached` operation.

- `tuHdlrs` contains the target updated listeners, registered via the `OnTargetUpdated` operation.
- `timers` contains a list of active timers.

2.5 Testing Your Implementation

To test your implementation of `combine` and `runProg`, you may wish to write and run simple test programs, such as `app/Example/CircleThatMoves.hs`.

Beyond this example, you are responsible for writing your own tests.

3 Implementing a Snakey Game in Scratchy

Once you have implemented the Scratchy domain-specific language, you should implement a small game with two snakes, one green and one blue, that move continuously on the grid.

Each snake should have seven links. The ”head” of the snake should be a circle; the ”tail” links should be squares.

The green snake should move using the keys ‘w’ (up), ‘a’ (left), ‘s’ (down), and ‘d’ (right). The blue snake should move using the keys ‘i’ (up), ‘j’ (left), ‘k’ (down), and ‘l’ (right).

The snakes should move in a snake-like manner. When a sprite turns, any linked sprite should only turn in that direction, after it has reached its designated target cell.

If one of the snakes collide with the grid barrier, the background color of the grid should turn red. If either snake collides with a sprite on the grid, the background color of the grid should turn black. When no snakes collide with barriers or sprites, the background color of the grid should be white.

You should implement your snakey game as an `SProg`; e.g.:

```
snakey :: SProg ()
snakey = undefined -- Fill this in
```

4 Project Report Requirements

Submit a brief (1-5 pages) report, documenting your project. Your report should contain sections corresponding to each subsection below.

4.1 Introduction

Assuming that your reader is familiar with the assignment, give a high-level summary of the main objectives of your project, and what you implemented.

4.2 Implementation of `combine` and `runProg`

Give a high-level summary of:

1. How you implemented `combine`.

2. How you implemented `runProg`.
3. Any changes that you made to the codebase to implement `combine` and `runProg`.

If you used particular techniques from the book, cite the relevant chapters where those techniques are described.

Summarize how you tested your implementations.

4.3 Implementation of your Snakey game

Summarize where in the codebase the reader can find your game.

Include instructions on how to run your Snakey game.

Give a high-level summary of how your implementation satisfies the requirement specification provided in Sect. 3.

Summarize how you tested your implementation.

4.4 Use of Generative AI

If you used generative AI, summarize how you used it.

If you only used Copilot for auto-completion, for example, simply state that.

If you used generative AI in any other way, describe how you used it, and summarize what prompts you gave it. If you cannot give an exhaustive list of prompts, include a few prompts that are representative to how you used generative AI.