Group 1:
William Cheng
Alexander Jacobson
Melanie Kent
Rehnoor Saini
Mattias Sebanc

2D Platformer Generator DSL Proposal

Our group will create a DSL that allows a user to design a simple 2D top-down game. Our target audience is aspiring programmers who are interested in learning the basics of game development. Our DSL will more closely resemble real code than similar block coding game DSLs like Scratch or Snap!, requiring a deeper understanding of the structure of loops and conditionals, and the importance of debugging and code formatting. Thus, we will be sure to provide clear guidelines and examples for how to write valid statements in our DSL. If all of the syntax is valid, the user will be able to play the game they designed using basic keyboard controls - up, down, left, and right.

To be more concrete, a working statement in our DSL will contain definition clauses for the square 2D arena in which the game is played, each entity in the game, behaviors for these entities, and the criteria for which the game ends. The arena definition is simple, only requiring the user to input its size. For example, the code below within the ARENA clause would create a 100 by 100-tile arena:

```
ARENA:
  size = 100
```

An entity can be the player's character, an enemy, an obstacle, or a collectible. Each entity definition allows the user to specify any number of a given list of attributes, such as the entity's name, size, starting coordinates, direction, health points, damage, and clone amount - this is the number of subsequent times an entity will clone itself once its previous clone is destroyed. Otherwise, default values will be filled in. For instance, one could write the following statement within the ENTITIES clause to define the player's character as Mario with 10 health points, a size of 2 by 2 tiles, and a starting position of 0, 0:

```
ENTITIES:
  define player Mario:
    start = 0,0
    health = 10
    size = 2
```

Non-player entities are largely pointless without any defined behaviors, which the BEHAVIOUR clause allows us to do. If the user is new to programming, they can assign very simple behaviors to an entity, such as periodically moving between two points within the arena. If they wish to be more adventurous, our DSL will allow the implementation of complex behaviors through loops, conditional statements, boolean logic, simple functions, and mutable variables. Below is an example of adding simple behaviour to the entity `Goomba`, such that it alternates between moving left and right 10 tiles:

```
BEHAVIOUR:
  Goomba:
    forever:
      face left
      move 10
      face right
      move 10
```

The `END CRITERIA clause can be used` to specify the conditions for when a game should finish. Programmers using our DSL can choose between listing one boolean condition (which when evaluated to true ends the game), or choose from one of our human-friendly boolean operators `one of` (analogous to the boolean "or" operator), and `all of` (analogous to the boolean "and" operator). These human-friendly boolean operators can also be nested, so that more complex boolean statements can be created. The example below assumes there already exists two entities, a `Mario` entity and a collectable `Coin` entity, and uses a simple `one of` statement:

```
END CRITERIA:
  one of:
    Mario health == 0
    Coin amount == 10
```

Loops are our first rich feature, the contents of which can be run either a constant number of times or until a specified condition is met. For example, a loop headed by

```
forever unless (foo == 0)
```

will continue running its contents until the mutable variable `foo` equals zero. Of course, conditional statements are not required to be in loops, and can be specified by the keyword "unless [boolean_statement]". Our next rich feature is mutable variables, which can be defined within and outside of functions and loops. Mutable variables must be initialized when they are declared. The structure for initializing a mutable variable `foo` to the value 10 is shown below:

```
var foo = 10
```

Mutable variables can also be set to mathematical operations and function calls, which will be evaluated before variable is set to the final value:

```
// the mathematical operation is evaluated before the value of bar is set
var bar = ((10 + 20) * 30)
```

```
// the function call is evaluated before the value of baz is set
var baz = some_function(20)
```

Our final rich feature is simple functions, which can help reduce duplicate code within a program. Function definitions start with the keyword `define`, followed by the name of the function and optional parameters in round brackets, separated by commas. The `return` keyword followed by the desired return value marks the end of the function definition. A very simple function that adds two numbers and returns the sum is shown below:

```
define get_sum(num):
  var sum = num + num
  return sum
```

All of our rich features can interact with each other in complex ways. Mutable variables and boolean logic let us break out of loops. Loops and mutable variables can be defined inside of functions. Mutable variables can be set to the return value of functions. Below is an example of how all of our rich features can interact with each other within the context of a complete, valid program using our DSL:

```
ENTITIES:
  define player Mario:
    start = 0,0
    health = 10
    size = 2
    direction = right
  define enemy Goomba:
    start = 1,1
    health = 8
    size = 1
    damage = 2
    direction = up
  define obstacle Box:
    start = 2,2
    size = 2
  define collectable Coin:
    start = random
    size = 1
    clone = 10

BEHAVIOUR:
  define function get_left_pos(num):
    var left_pos = num + num
   return left_pos
  define function move_goomba(left_pos, right_pos):
    forever unless (Goomba on pos Box):
      face left
      move left_pos
      face right
      move right_pos
      repeat 10 times:
        move 2
        turn right
        unless ((Goomba on pos 1,1) and (not (Goomba on pos Mario))):
          right_pos = right_pos + 2

  Goomba:
    var left_pos = get_left_pos(4)
    move_goomba(left_pos, 10)

END CRITERIA:
  all of:
    Goomba heath == 0
    one of:
      Mario health == 0
      Coin amount == 10
```

The example above is only one of limitless ways our rich features can interact with each other to create interesting 2D games, all while keeping the syntax simple enough for aspiring programmers to learn the basics of coding.