UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

GRADO EN INGENIERÍA INFORMÁTICA

Propuesta y estudio de un lenguaje y entorno mínimos para la enseñanza de la programación de ordenadores.

Proposal and study of a minimum language and environment for teaching computer programming.

Realizado por
José María Pérez Bravo

Tutorizado por
Francisco José Vico Vela

Departamento
Lenguajes y Ciencias de la Computación
Universidad de Málaga

MÁLAGA, JUNIO, 2020

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

Propuesta y estudio de un lenguaje y entorno mínimos para la enseñanza de la programación de ordenadores

Proposal and study of a minimum language and environment for teaching computer programming.

Realizado por
José María Pérez Bravo

Tutorizado por
Francisco José Vico Vela

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO, 2020

Fecha defensa:

Resumen:

En los últimos años cada vez más países han aumentado sus esfuerzos por introducir la programación en sus currículos, enseñando programación en entornos de programación (IDE) adaptados, programación de robots o desarrollando algún proyecto relacionado con el mundo de la informática.

Nuestra aportación consiste en analizar varios IDE para decidir que elementos son los necesarios para enseñar programación de forma sencilla y sin redundancia. Nos centramos en un IDE en concreto: *Toolbox Academy*, desarrollado en el Dpto. de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, que se está utilizando en los sistemas educativos de las comunidades autónomas para enseñar programación en niveles preuniversitarios. Se propone también un currículo de programación dividido en unidades y un conjunto de tareas con los recursos analizados anteriormente.

Además se define un lenguaje sencillo y reducido para este IDE a partir del análisis de lenguajes muy utilizados en la actualidad como *JavaScript*, *Ruby* o *GNU Octave*: *ToyScript*.

Palabras Clave: Programación, Recursos, Lenguaje, IDE, Enseñanza.

Abstract:

In recent years more and more countries have increased their efforts to introduce programming in their curricula, by teaching programming in adapted programming environments (IDE), programming of robots or developing some project related to the world of computing.

Our contribution consists of analyze some IDEs to decide which elements are necessary to teach programming in a simple way and without redundancy. We focus on a specific IDE: *Toolbox Academy*, developed in the Department of Languages and Computer Science of the University of Malaga, which is being used in educational systems of the autonomous communities to teach programming at pre-university levels. A programming curriculum divided into units and a set of tasks with the resources previously analyzed are also proposed.

In addition, a simple and reduced language is defined for this IDE, based on the analysis of languages that are currently widely used such as *JavaScript*, *Ruby* or *GNU Octave*: *ToyScript*.

Keywords: Programming, Resources, Language, IDE, Teaching.

Índice de contenidos

CAPÍTULO 1

Introducción

1.1. Motivación

Actualmente la programación es cada vez más popular en el entorno educativo, no solo por la cantidad de nuevos puestos de trabajo que están surgiendo en relación a las TIC (Tecnologías de la Información y la Comunicación), sino también por los beneficios que aporta, como la mejora de capacidad de abstracción, el pensamiento computacional y la capacidad de resolución de problemas analíticos, demostrando además que esta mejora en la capacidad de resolver problemas analíticos ocurre para la totalidad de los alumnos, no solo para los predispuestos a especializarse en Ciencias de la computación [1].

Además de los nuevos puestos de trabajo y los beneficios cognitivos que la enseñanza de la programación aporta, la principal motivación de nuestro trabajo es la siguiente: Es necesario encontrar un conjunto de recursos mínimos donde se pueda realizar computación para no caer en la redundancia, es decir, para poder enseñar programación sin elementos que distraigan, pero sin dejar atrás ningún concepto computacional.

Esta es la principal motivación porque si no se consigue un entorno adecuado, reducido y adaptado, puede ser difícil para los estudiantes de programación aprender sobre un IDE y lenguaje cualesquiera, por la dificultad y la variedad de estructuras computacionales que nos permiten hacer la misma función y que para un estudiante pueden ser abrumadoras.

1.2. Objetivos

El principal objetivo de este trabajo es analizar a través de un IDE y un lenguaje adaptados, que recursos son los necesarios para poder enseñar programación de forma intuitiva, para que el usuario asimile los conceptos computacionales que usa a través de la resolución de tareas.

Los objetivos específicos consistirán en:

- Determinar los elementos usados hasta la actualidad.
- Estudiar estos elementos para analizar carencias.
- Proponer IDE.
- Proponer lenguaje.
- Proponer tareas con recursos mínimos.

1.3. Metodología empleada

Durante este proyecto hemos utilizado Scrum, una metodología ágil que consiste en la ejecución de iteraciones de duración fija en las que se consiguen ciertos objetivos del proyecto. Esta metodología destaca por su facilidad para la adopción de cambios y nuevos requisitos durante el desarrollo del proyecto.

Hemos complementado esta metodología con otra conocida metodología llamada kanban que consiste en organizar las tareas pendientes en un tablero, evaluando la carga de trabajo de cada una de ellas y dando mayor prioridad a las excepcionales. Esta metodología destaca por la flexibilidad que proporciona.

Para implementar estas dos metodologías hemos usado Trello, una herramienta online que permite la administración de proyectos proporcionando un tablero digital y el uso de tarjetas, que representan tareas y que se organizan en columnas de forma que podamos mover las tareas de una columna a otra.

Figura 1.1: Trello

CAPÍTULO 2

Estado de la cuestión

En este apartado analizaremos las opciones actuales para el aprendizaje de la programación. Primero haremos una breve introducción de los lenguajes más usados de forma general para centrarnos después en algunos de los lenguajes más usados para la enseñanza de la programación.

2.1. Lenguajes existentes no orientados a la enseñanza

En este apartado vamos a analizar distintas características de los lenguajes más usados en el mundo de la programación.

- **Python:** Python es un lenguaje interpretado y multiparadigma (permite programación Imperativa, Orientada a Objetos y Funcional). Respecto al tipado, estamos ante un lenguaje de tipado Dinámico y Fuerte. Es un lenguaje multiplataforma que destaca por la cantidad de frameworks que tiene y la facilidad de aprendizaje.
- **GNU Octave:** GNU Octave es un lenguaje interpretado. Respecto al tipado, estamos ante un lenguaje de tipado Dinámico y Débil.
- **Ruby:** Ruby es un lenguaje interpretado y Orientado a Objetos. Respecto al tipado, estamos ante un lenguaje de tipado Dinámico y Fuerte. Es un lenguaje multiplataforma que destaca por su sintaxis, cercana al lenguaje natural, además de ser muy

utilizado en el desarrollo web.

- **Java:** Java es un lenguaje compilado y Orientado a Objetos. Respecto al tipado, estamos ante un lenguaje de tipado Estático y Fuerte. Destaca por su robustez y seguridad.
- **C:** C es un lenguaje compilado y de paradigma imperativo. Respecto al tipado, estamos ante un lenguaje de tipado Estático y Fuerte. Destaca en la velocidad de ejecución, aunque está más alejado del lenguaje natural que otros.
- **Go:** Go es un lenguaje compilado y paradigma imperativo aunque permite objetos. Respecto al tipado, estamos ante un lenguaje de tipado Estático y Fuerte. Su principal característica es la sencillez, además de la claridad de su sintaxis.
- **JavaScript:** JavaScript es un lenguaje interpretado y Orientado a Objetos. Respecto al tipado, estamos ante un lenguaje de tipado Dinámico y Débil. Es interesante porque todos los navegadores actuales pueden interpretarlo.

2.2. Lenguajes existentes orientados a la enseñanza

En este apartado vamos a analizar los distintos lenguajes de programación orientados a la enseñanza de la misma.

2.2.1. LOGO

LOGO es un lenguaje de programación libre de alto nivel desarrollado con propósitos didácticos basado en el lenguaje Lisp, que apareció en los años 60. LOGO es un lenguaje que no tiene estándar y que posee una gran variedad de implementaciones.

El lenguaje destacó porque podía producir gráficos a través de una tortuga virtual a la que se le daba instrucciones. Además este lenguaje tiene versiones para distintos idiomas, entre ellos el español.

Hay que tener en cuenta que las instrucciones de desplazamiento pueden cambiar de una versión de LOGO a otra.

Caja 2.1 : Comandos básicos de LOGO en español

```
1
2  avanzar 100          (la tortuga avanza 100 pasos)
3  girarderecha 90      (la tortuga gira 90 grados a la derecha)
4  girarizquierda 90    (la tortuga gira 90 grados a la izquierda)
5  BP                   (borra la pantalla)
6  repite 3 [avanzar 100] (repite 3 veces lo que hay entre [ y ])
7  MT                   (muestra la tortuga)
8  OT                   (oculta la tortuga)
9  SP                   (sin pluma, no dibuja mientras camina)
10 CP                   (con pluma, dibuja mientras camina)
```

2.2.2. Scratch

Scratch es un lenguaje de programación usado en la plataforma con el mismo nombre que implementa las estructuras computacionales de manera gráfica, mediante bloques. Estos bloques se pueden encajar para construir bloques más grandes que actúan de programa.

- `when clicked`: Nos permitirá ejecutar el programa una vez hagamos click en la bandera verde.

Figura 2.1: Bloque de ejecución

- `move`: Nos permitirá mover al agente por el mapa un número de pasos. Si este número es positivo nos moveremos hacia la derecha, si es negativo nos moveremos hacia la izquierda y si es 0 nos quedaremos quietos.

Figura 2.2: Bloque de movimiento

- `turn`: Nos permitirá girar al agente un número de grados. Si este número es positivo giraremos en el sentido que indica el bloque, si es negativo nos moveremos en sentido contrario y si es 0 no giraremos.

Figura 2.3: Bloque de rotación

- `repeat`: Nos permitirá repetir un conjunto de comandos, es decir, un conjunto de bloques un número determinado de veces. Si este número es 0, no se ejecutará ningún bloque dentro del bloque `repeat`.

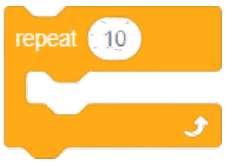

Figura 2.4: Bloque de repetición definida

- Variables: Nos permitirá realizar asignaciones a variables, además de poder usar su valor.

Figura 2.5: Bloques de operaciones con variables

- Operaciones aritméticas: Nos permitirá realizar sumas, restas, multiplicaciones y divisiones de valores numéricos.

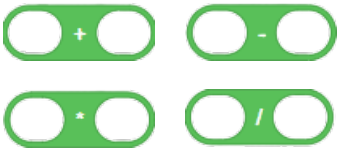

Figura 2.6: Bloques de operaciones aritméticas

- Comparaciones: Nos permitirá hacer comparaciones que podremos usar en bucles y condicionales.

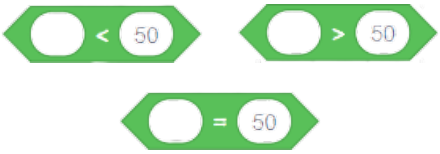

Figura 2.7: Bloques de comparaciones

- Operaciones lógicas: Nos permitirá realizar conjunciones, disyunciones y negaciones.

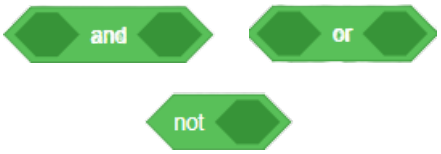

Figura 2.8: Bloques de operaciones lógicas

- `if` : Nos permitirá ejecutar un conjunto de comandos si una condición dada es verdadera.

Figura 2.9: Bloque de condición

- `if - else`: Nos permitirá ejecutar un conjunto de comandos si una condición dada es verdadera, sino, ejecuta otro conjunto de comandos dado.

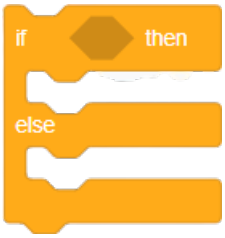

Figura 2.10: Bloque de condición

- `repeat until`: Nos permitirá repetir un conjunto de comandos hasta que una condición dada sea verdadera.

Figura 2.11: Bloque de repetición indefinida

2.2.3. Blocky

Blocky es un lenguaje de programación dedicado a enseñar programación mediante bloques. Es similar a Scratch, pero proporciona la opción de ver el código equivalente en JavaScript, Python y otros lenguajes. En concreto vamos a analizar como la plataforma Code.org implementa este lenguaje ampliamente personalizable, concretamente en su *Express Course*.

- `when run`: Nos permitirá ejecutar el programa una vez hagamos click en RUN.

Figura 2.12: Bloque de ejecución

- `move`: Nos permitirá mover al agente por el mapa hacia donde esté mirando o en sentido contrario dependiendo de la dirección, que puede ser `forward` o `backward`.

Figura 2.13: Bloques de movimiento

- `turn`: Nos permitirá girar al agente hacia la derecha o la izquierda.

Figura 2.14: Bloque de rotación

- `repeat`: Nos permitirá repetir un conjunto de comandos un número determinado de veces. Si este número es 0, no se ejecutará ningún bloque.

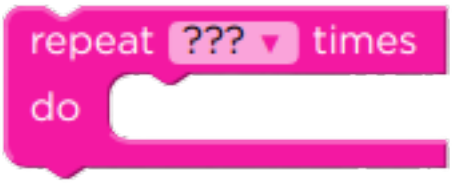

Figura 2.15: Bloque de repetición definida

- Variables: Nos permitirá realizar asignaciones a variables, además de poder usar su valor.

Figura 2.16: Bloques de operaciones con variables

- Operaciones aritméticas: Nos permitirá realizar operaciones aritméticas con valores numéricos.

Figura 2.17: Bloques de suma

- `if at` : Nos permitirá ejecutar un conjunto de comandos si el personaje se encuentra en una localización dada.

Figura 2.18: Bloque de condición

- `if at - else`: Nos permitirá ejecutar un conjunto de comandos si el personaje se encuentra en una localización dada, en otro caso ejecuta otro conjunto dado.

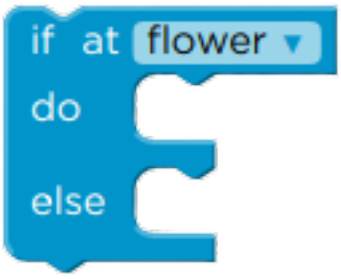

Figura 2.19: Bloque de condición

- `while`: Nos permitirá repetir un conjunto de comandos mientras que una condición dada sea verdadera.

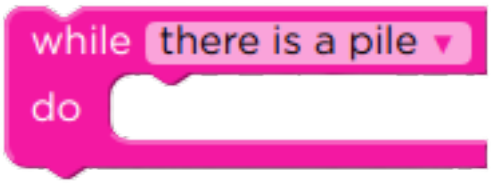

Figura 2.20: Bloque de repetición indefinida

2.2.4. Coffeescript

CoffeeScript es un lenguaje que compila a JavaScript usado principalmente en la industria del desarrollo web.

Este lenguaje puede aprenderse en plataformas como *CodeMonkey*, que introduce funciones como `step`, `turn` o `times` para poder utilizarlo sobre un mapa bidimensional y hacerlo más sencillo.

- `step`: Nos permitirá mover al agente por el mapa un número de pasos. Si este número es positivo nos moveremos hacia donde esté mirando, si es negativo nos moveremos en sentido contrario y si es 0 nos quedaremos quietos.

Caja 2.2 : `step` - Coffeescript

```
1 step 5
```

- `turn`: Nos permitirá girar al agente dependiendo de si toma el valor `left` o `right`.

Caja 2.3 : `turn` - Coffeescript

```
1 turn left
2 turn right
```

- `times`: Nos permitirá repetir un conjunto de comandos un número determinado de veces. Si este número es 0, no se ejecutará ninguna sentencia dentro del bloque.

Caja 2.4 : `times` - Coffeescript

```
1 3.times ->
2     turn right
```

- Variables: Nos permitirá realizar asignaciones a variables, además de poder usar su valor.

Caja 2.5 : Variables - Coffeescript

```
1 x = 10
2 step x
```

- Operaciones aritméticas: Nos permitirá realizar sumas, restas, multiplicaciones y divisiones de valores numéricos.

Caja 2.6 : Operaciones aritméticas - Coffeescript

```
1 x = 10
2 y = 5
3
4 z = x + y
5 x = x - y
6 z = x * y
7 x = x / y
```

- Comparaciones: Nos permitirá hacer comparaciones que podremos usar en bucles y condicionales.

Caja 2.7 : Comparaciones - Coffeescript

```
1 x = 10
2 y = 5
3
4 x == y
5 x > y
6 x < y
7 x >= y
8 x <= y
```

- Operaciones lógicas: Nos permitirá realizar conjunciones, disyunciones y negaciones.

Caja 2.8 : Conjunción, Disyunción y Negación respectivamente - Coffeescript

```
1 x = 10
2 y = x == 10
3 z = false
4
5 y and z
6 y or z
7 not z
```

- `if`: Nos permitirá ejecutar un conjunto de comandos si la condición es verdadera.

Caja 2.9 : `if` - Coffeescript

```
1 x = 10
2 direction = right
3 if direction == right
4     turn direction
5     step x
```

- `if - else`: Nos permitirá ejecutar un conjunto de comandos si la condición dada es verdadera, en otro caso se ejecuta otro conjunto de comandos.

Caja 2.10 : `if-else` - Coffeescript

```
1 x = 10
2 direction = left
3 if direction == right
4     turn direction
5     step x
6 else
7     step 2 * x
```

- `until`: Nos permitirá repetir un conjunto de comandos hasta que la condición dada sea verdadera.

Caja 2.11 : Bucle `until` - Coffeescript

```
1 x = 1
2 until x > 4
3     step x
4     x = x + 2
```

- `()` `->` : Nos permitirá crear funciones, con objetivo de reutilizar código.

Caja 2.12 : Función - Coffeescript

```
1 moveif10 = (steps) ->
2     if steps == 10
3         step steps
4
5 moveif10 5
6 moveif10 10
```

2.2.5. Elementos comunes y distintivos

En esta sección analizamos las similitudes y diferencias para ver cuales son las ventajas y carencias de estos lenguajes, de forma que podamos aprovecharlos en nuestra propuesta.

Elementos comunes: Como podemos observar, todos estos lenguajes comparten unas estructuras computacionales similares.

- Bucles definidos
- Variables
- Condicionales
- Bucles indefinidos

Elementos distintivos:

- Forma de implementación

  Lenguajes que implementan la computación mediante bloques:

  - Scratch
  - Blockly

  Lenguajes que implementan la computación mediante código:

  - CoffeeScript
  - LOGO

- Instrucciones de movimiento

  Lenguajes que implementan el movimiento en términos relativos:

  - Scratch
  - CoffeeScript
  - Blockly
  - LOGO

  Ninguno de los lenguajes analizados implementan el movimiento en términos absolutos.

CAPÍTULO 3

Propuesta

Estudios recientes muestran que en un entorno controlado, con un lenguaje reducido pensado para simplificar la sintaxis y acercarlo más al lenguaje natural, alumnos de último curso de secundaria son capaces de demostrar competencias en todos los conceptos computacionales básicos como bucles, condicionales, definición de funciones, etc.[2]

No solo esto, sino que además muestran que las competencias en ciencias de la computación se adquieren independientemente del género.[3]

Después de haber analizado los lenguajes y plataformas más usados, en concreto para la enseñanza de programación, nuestra propuesta se centra en el lenguaje de programación ToyScript implementado en la plataforma ToolBox Academy.

Los motivos principales para la propuesta de este lenguaje son la utilización de elementos sencillos para facilitar la adquisición de competencias mediante la simplificación de la sintaxis y el uso de recursos computacionales reducidos. Este lenguaje no busca el conjunto mínimo, sino un conjunto de estructuras reducido que permita mantener la sencillez.

3.1. Descripción

En esta sección vamos a describir nuestra propuesta, describiendo primero detalladamente la plataforma y el lenguaje sobre los que vamos a trabajar, así como una justificación sobre ciertas decisiones que se han tomado.

3.1.1. Toolbox Academy

Toolbox Academy es una plataforma para la enseñanza de programación en el ámbito preuniversitario desarrollado por el departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga. Toolbox Academy cuenta con 3 áreas diferenciadas: Una parte donde aparecerá el enunciado de una tarea a resolver, una parte para escribir el código para resolver una tarea y una parte donde aparecerá una cuadrícula dividida en celdas. Toolbox Academy cuenta con un curso básico en **ToyScript** y otro en JavaScript.

Elementos del entorno:

- Roby: Es el personaje al que le vamos a dar ordenes mediante programas. Es el elemento que se moverá por la cuadrícula. Roby es a Toolbox Academy lo que el cabezal lector a la Máquina de Turing.

Figura 3.1: Roby

- Cuadrícula: Es el espacio por el cual Roby puede moverse. La cuadrícula es a Toolbox Academy lo que la cinta a la Máquina de Turing.

Figura 3.2: Cuadrícula

- Objeto : Estos elementos pueden ser recogidos por el agente, lo que nos justifica, entre otras estructuras computacionales, el uso de comandos de movimientos, pues necesitamos usarlos para recoger estos objetos. Si disponemos estos elementos alineados, queda justificado la estructura de bucle definido.

Figura 3.3: Objeto

- Agentes auxiliares: Estos elementos son agentes a los que Roby puede pedir información sobre cierto problema, y gracias a ellos podemos introducir el uso no solo de variables (para guardar la información que proporcionan), sino de funciones como `info` (para pedir la información que tienen).

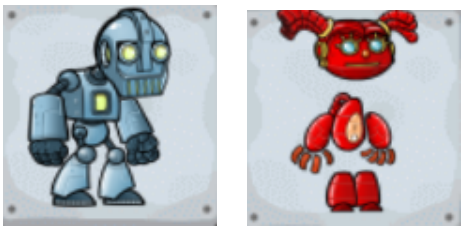

Figura 3.4: Agentes auxiliares

- Puertas: Introducimos este recurso computacional para poder justificar el uso de condicionales en nuestros programas. Ahora no se tratará de recoger tuercas como en otros casos, sino de entrar por la puerta correcta. Este recurso, junto al uso de agentes externos, son la clave para la creación de tareas con condicionales.

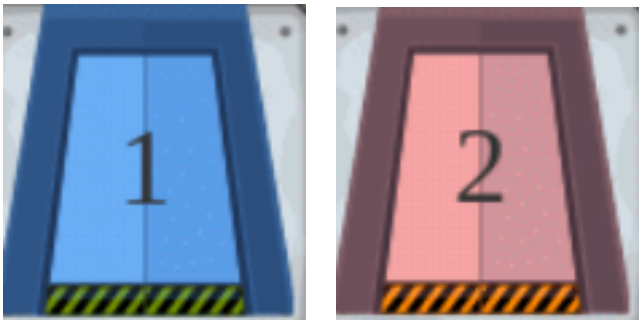

Figura 3.5: Puertas

- Cajas: Introducimos este recurso computacional para poder justificar el uso de bucles indefinidos. Ahora las tareas tratarán sobre encontrar la caja con el contenido correcto.

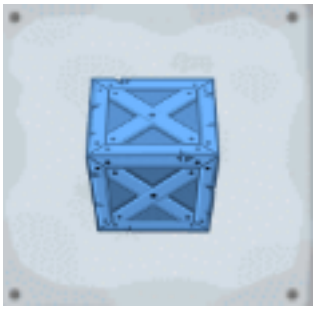

Figura 3.6: Caja

Este es el conjunto de recursos mínimos que proponemos, ya que todos ellos son necesarios para que el uso de las estructuras computacionales del lenguaje que vamos a definir tenga sentido.

3.1.2. ToyScript

ToyScript está inspirado en varios lenguajes, no solo en JavaScript. Esta inspirado en GNU Octave, pues introduce la palabra `end` para finalizar estructuras. Además introduce el bucle `repeat`, cuyo concepto existe en lenguajes como Ruby, aunque con otra sintaxis. Es muy simple, no solo sintácticamente, pues no tiene llaves, paréntesis ni puntos y comas, sino porque solo cuenta con un bucle definido (`repeat`), uno indefinido (`do-while`), y dos condicionales (`if`, `if-else`) como estructuras computacionales.

La razón de esta sencillez sintáctica es el acercamiento de este lenguaje al lenguaje natural, además de simplicidad a la hora de escribir el programa. Hay que tener en cuenta que esta propuesta está enfocada en cursos K-12 (cursos de primaria y secundaria), y que por lo tanto habrá usuarios que no estén familiarizados con el teclado, o que incluso no hayan aprendido a leer y escribir correctamente.

Por otro lado, la razón por la que contamos con tan pocas estructuras computacionales es clara, buscamos introducir los conceptos generales de la programación en este lenguaje, es decir, condicionales, bucle definido e indefinido, pero manteniendo la simpleza de tal forma que el usuario no quede abrumado. Es por esto que no existen estructuras como `for each` o `switch`.

A continuación presentaremos la sintaxis del lenguaje en notación EBNF (Extended Backus-Naur form), una notación que nos permite describir lenguajes formales a través de sus gramáticas. En nuestro caso, el axioma es $\langle code \rangle$, que representa el programa que puede estar formado por una o más instrucciones.

Caja 3.1 : Sintaxis de ToyScript

⟨*letter*⟩ ::= 'a'|...|'z'|'A'|...|'Z'

⟨*digit*⟩ ::= '0'|...|'9'

⟨*arithmetic*⟩ ::= '+'|'−'|'*'|'/'

⟨*relational*⟩ ::= '=='|'!='|'<'|'>'|'<='|'>='

⟨*logical*⟩ ::= '!'|'&'|'|'

⟨*integer*⟩ ::= ['−']⟨*digit*⟩{⟨*digit*⟩}

⟨*variable*⟩ ::= ⟨*letter*⟩{⟨*letter*⟩|⟨*digit*⟩}

⟨*term*⟩ ::= ⟨*integer*⟩|⟨*variable*⟩

⟨*expression*⟩ ::= ⟨*term*⟩|'('⟨*expression*⟩')'|⟨*expression*⟩⟨*arithmetic*⟩⟨*expression*⟩

⟨*condition*⟩ ::= ⟨*term*⟩⟨*relational*⟩⟨*term*⟩|⟨*term*⟩⟨*logical*⟩⟨*term*⟩|'('⟨*term*⟩')'

⟨*code*⟩ ::= ⟨*instruction*⟩{⟨*instruction*⟩}

⟨*instruction*⟩ ::= ⟨*command*⟩|⟨*assignment*⟩|⟨*repeat*⟩|⟨*if*⟩|⟨*while*⟩|⟨*info*⟩

⟨*command*⟩ ::= 'up'|'down'|'left'|'right'

⟨*info*⟩ ::= ⟨*variable*⟩'='`info`

⟨*assignment*⟩ ::= ⟨*variable*⟩'='⟨*expression*⟩

⟨*repeat*⟩ ::= `repeat`⟨*expression*⟩⟨*code*⟩'end'

⟨*if*⟩ ::= `if`⟨*condition*⟩⟨*code*⟩['`else`⟨*code*⟩]'end'

⟨*while*⟩ ::= 'do'⟨*code*⟩'`while`'⟨*condition*⟩

3.1.3. Justificación

En este apartado vamos a discutir sobre algunas decisiones que se han tomado en nuestra propuesta.

- **Eliminación de llaves y adición de `end`**: La decisión de eliminar las llaves surge por la lejanía que supone la utilización de estos elementos sintácticos con el lenguaje natural. De esta elección emana la decisión de añadir la clausula `end` para cerrar un bloque, podríamos haber usado la indentación como en otros lenguajes como Python, pero el uso de `end` está mucho más cerca del lenguaje natural que esto o una llave.
- **Eliminación de otros elementos sintácticos**: Además de lo comentado anteriormente, con objeto de acercar **ToyScript** al lenguaje natural, se eliminan otros elementos como ';' o los paréntesis, de forma que queda un código mucho más legible.
- **Utilización de comandos de movimiento absolutos**: Como podemos observar los lenguajes de programación centrados en la enseñanza analizados anteriormente, cuentan todos con sistemas de movimientos relativos. Nosotros hemos decidido usar un sistema absoluto porque es más visual a la hora de planificar las acciones a realizar, ya que con un sistema relativo es necesario conocer la rotación del personaje, además de su posición.
- **Número máximo de palabras o números**: Este es uno de los recursos más interesantes que tenemos. Gracias a él, en ciertas tareas nos aseguramos que se usen los conceptos computacionales que queremos, y no otros equivalentes. De esta forma, si queremos que se use un bucle, podremos evitar que se usen las instrucciones que irían en su interior de forma repetida.

- **Rangos de números especiales**: Este recurso, nos sirve para que el usuario no pueda diseñar programas a medida, sino que los programas escritos tengan que satisfacerse para todos los escenarios posibles, es decir, los programas deben ser generales. Podemos distinguir dos tipos de estos rangos:

  - Rango de números donde se prueban todas las combinaciones.

    Ejemplo: Si establecemos para un recurso un rango de este tipo, desde 5000 hasta 5005, entonces, el programa se probará para cada uno de estos valores en estos recursos.

  - Rango de números donde se prueban todas las permutaciones.

    Ejemplo: Si establecemos para 3 recursos un rango de este tipo, desde 100 hasta 102, entonces se probarán todas las posibles permutaciones de insertar estos valores en dichos recursos.

- **Elección de `do-while` en lugar de `while`**: La elección de `do-while` como bucle indefinido en lugar de `while` está fundamentada en el tipo de tareas propuestas para ser resueltas mediante bucle indefinido. Estas tareas normalmente consisten en explorar una serie de cajas alineadas de forma consecutiva hasta encontrar la correcta. Para ello debemos recoger información sobre cada caja y después comprobar si se cumple una condición dada. Esto se puede hacer con ambos bucles, pero es mucho más intuitivo con `do-while`, cuyo funcionamiento encaja perfectamente con el problema descrito.

Figura 3.7: Ejemplo `do-while`

Para resolver el ejemplo anterior con bucle `do-while` primero debemos movernos a la derecha, después obtener información de la caja y por último comprobar si la condición se ha cumplido.

A continuación mostramos unos ejemplos que consisten en encontrar la caja con el valor 5.

Caja 3.2 : Resolución con `do-while`

```
1 do
2    right
3    contenido_caja = info
4 while contenido_caja != 5
```

Sin embargo, para resolverlo con bucle `while`, deberemos utilizar un valor por defecto para poder realizar la primera iteración del bucle o realizar la primera iteración fuera del mismo. Esto es mucho menos intuitivo.

Caja 3.3 : Resolución con `while` - valor por defecto

```
1 contenido_caja = 0
2 while contenido_caja != 5
3   right
4   contenido_caja = info
5 end
```

Caja 3.4 : Resolución con `while` - primera iteración fuera

```
1 right
2 contenido_caja = info
3 while contenido_caja != 5
4   right
5   contenido_caja = info
6 end
```

3.2. Funcionamiento

3.2.1. Definición informal

En este apartado explicaremos el funcionamiento de cada una de las estructuras computacionales y comandos que hemos expuesto anteriormente en la sintaxis de **ToyScript**.

- `right`, `left`, `up`, `down`: Nos permitirá mover al agente una cuadrícula en la dirección que indica la instrucción.

Caja 3.5 : Movimiento - ToyScript

```
1 up
2 down
3 left
4 right
```

- `repeat`: Es un bucle definido que nos permitirá repetir un conjunto de instrucciones un número determinado de veces. Si este número es 0, no se ejecutará ninguna sentencia dentro del bucle.

Caja 3.6 : `repeat` - ToyScript

```
1 repeat 3
2   right
3   up
4   right
5 end
```

- Variables: Nos permitirá realizar asignaciones a variables, para después poder usar su valor.

Caja 3.7 : Variables - ToyScript

```
1 repeticiones = 3
2 repeat repeticiones
3   right
4 end
```

- Operaciones aritméticas: Nos permitirá realizar sumas, restas, multiplicaciones y divisiones de valores numéricos.

Caja 3.8 : Operaciones aritméticas - ToyScript

```
1 x = 10
2 y = 5
3
4 z = x + y
5 x = z - y
6 z = y * y
7 x = z / y
```

- Comparaciones: Nos permitirá hacer comparaciones que podremos usar en bucles y condicionales.

Caja 3.9 : Comparaciones - ToyScript

```
1 x = 10
2 y = 5
3
4 if x == y
5   down
6 end
7 if x > y
8   right
```

```
9  end
10 if x < y
11   up
12 end
```

- `info`: Nos permitirá recibir información del entorno, por ejemplo, para guardar en una variable la puerta por la que debemos entrar o cuántas tuercas recoger.

Caja 3.10 : Info - ToyScript

```
1 puerta = info
```

- `if`: Nos permitirá ejecutar un conjunto de comandos si se cumple una condición.

Caja 3.11 : `if` - ToyScript

```
1 puerta = info
2 right
3 if puerta == 2
4    down
5 end
```

- `if - else`: Nos permitirá ejecutar un conjunto de comandos si se cumple una condición, en otro caso se ejecuta otro conjunto de comandos.

Caja 3.12 : `if-else` - ToyScript

```
1 puerta = info
2 if puerta == 1
3    left
4 else
5    right
6 end
```

- Operaciones lógicas: Nos permitirá realizar conjunciones, disyunciones y negaciones.

Caja 3.13 : Operadores lógicos - ToyScript

```
1 puerta = info
2 right
3 if !(puerta == 2 | puerta > 5) & puerta > 3
4    down
5 end
```

- `do-while`: Nos permitirá repetir un conjunto de comandos mientras se de una condición. Evaluará la condición después de ejecutar los comandos.

Caja 3.14 : Bucle `do-while` - ToyScript

```
1 do
2    right
3    tuercas = info
4 while tuercas != 100
```

3.2.2. Definición formal

En este apartado vamos a encargarnos de definir en primer lugar el estado de la computación de este modelo de cómputo para posteriormente poder definir formalmente la semántica del mismo. Esto es útil porque nos permite definir el comportamiento de forma muy precisa.

3.2.2.1. Definición de Estado

El estado de la computación de este modelo incluirá información sobre la posición del agente, sobre el inventario, sobre la cuadrícula y sobre las variables.

Primero definiremos las 2 clases de objetos que puede haber por el mapa. En primer lugar objetos que se pueden recoger, como pueden ser tuercas o tornillos, y en segundo lugar objetos que no se pueden recoger, por ejemplo cajas o puertas. Caracterizaremos estos objetos con un número natural ($\mathbb{N}$), que en caso de ser un objeto que se puede recoger servirá para indicar distintos tipos de objetos, y en caso de que no se pueda recoger, será un valor que al que podremos acceder desde un programa a través de `info`. Es decir, en caso de poderse recoger nos servirá para saber si es una tuerca, tornillo, ..., pero en caso de no poderse recoger, por ejemplo una caja, servirá para saber cuantos objetos contiene dicha caja.

Definimos la cuadrícula del mundo como una cinta bidimensional, infinita hacia la derecha y hacia abajo, en la que en cada cuadrícula tendremos 2 valores, uno de ellos nos indicará si el objeto se puede recoger y el otro será un valor que actuará de identificador o podrá ser usado en un programa dependiendo de si se puede o no recoger.

Por lo tanto, la cuadrícula quedará representada por una función $Cuadricula : \mathbb{N}^2 \rightarrow (\mathbb{N}, \{0, 1\})$, que dada una posición de la cuadrícula, nos devolverá el valor que contiene, y 0 en caso de no poderse recoger o 1 en caso de serlo.

Además, será necesario definir una función que nos permita vaciar una casilla de la cuadrícula, para actualizarla por ejemplo, una vez recojamos un objeto. Definimos $clearC : (Cuadricula, \mathbb{N}^2) \rightarrow Cuadricula$, que dada una Cuadrícula y una posición, nos devuelve la misma Cuadrícula con esa posición vacía. Es decir, siendo $c' = clearC(c, x, y)$:

$$c'(x', y') = \begin{cases} (0,0) & \text{si } x = x' \text{ e } y = y' \\ c(x,y) & \text{en otro caso} \end{cases}$$

Si aplicamos *clearC* a una cuadrícula c y una posición (x, y), esto nos devolverá una nueva cuadrícula (recordemos que hemos definido la cuadrícula como una función), que para una entrada (x', y') devolverá $(0, 0)$ si esa posición coincide con la usada como argumento.

Definimos la posición del agente en la cuadrícula como $posicion \in \mathbb{N}^2$, es decir, como un vector de naturales de 2 dimensiones. Siendo p la posición del agente, denotaremos como p_1 y p_2 para referirnos a la primera y segunda componente del mismo respectivamente.

Definimos el inventario del agente como $Inventario = \mathbb{N} \to \mathbb{N}$, es decir, una función que dado un natural que represente un objeto, nos devolverá la cantidad de dicho objeto que el agente posee.
Para incrementar el inventario definimos $updateI : (Inventario, \mathbb{N}) \to Inventario$, que dado un inventario y un natural que representa un tipo de objeto, se devolverá un inventario actualizado. Es decir, siendo $i' = updateI(i, n)$:

$$i'(n') = \begin{cases} i(n) + 1 & \text{si } n = n' \\ i(n') & \text{en otro caso} \end{cases}$$

Si aplicamos *updateI* a un inventario i y un objeto representado por n, se devolverá un nuevo inventario, que en caso de aplicarse a la misma entrada, nos devolverá el valor incrementado.

Por último definimos el estado de las variables como $Variables = variable \to \mathbb{Z}$, que será una función que dada una variable, nos devolverá su valor.

Para poder actualizar el estado de una variable definimos la función *updateV*.
$updateV : (Variables, variable, \mathbb{Z}) \to Variables$ que dado el estado de unas variables, una variable y un valor, devolverá un nuevo estado de variables, con el valor de la variable tratada actualizado. Es decir, siendo $v' = updateV(v, x, n)$:

$$v'(y) = \begin{cases} n & \text{si } x = y \\ v\ y & \text{si } x \neq y \end{cases}$$

Es decir, al aplicar $updateV$ a un estado de variables v, a una variable x y a un valor n, se nos devuelve un nuevo estado de variables que para la misma entrada, nos devolverá el valor actualizado.

Finalmente definimos el estado de la computación como $State = (a, b, c, d)$, $a \in Posicion$, $b \in Inventario$, $c \in Cuadricula, d \in Variables$.

En nuestra plataforma, el estado inicial estará caracterizado únicamente por la posición del agente y la cuadrícula, pues el estado inicial de las variables y el inventario será siempre el mismo.

$$inventario_0(n) = 0,\ \forall n \in \mathbb{N},\ inventario_0 \in Inventario$$

$$variable_0(x) = 0, \forall x \in variable,\ variable_0 \in Variables$$

3.2.2.2. Conceptos Previos

En este apartado vamos a encargarnos de definir funciones auxiliares y conceptos previos que serán necesarios en el siguiente apartado, donde presentaremos la semántica del modelo.

Vamos a definir la semántica de este modelo de computación mediante la utilización de semántica Denotacional. Para ello necesitamos apoyarnos en funciones cuya definición es composicional. Es decir, nuestras funciones solo podrán estar definidas en función de sus constituyentes inmediatos.

La razón por la que necesitamos este tipo de funciones es sencilla. Sobre estas estructuras podríamos realizar, si fuera necesario, demostraciones utilizando inducción estructural, por ejemplo, para demostrar que dos programas son semánticamente equivalentes, es decir, para cualquier estado inicial, obtenemos el mismo resultado.

Definiremos $\perp$ como divergencia, es por eso que a partir de ahora veremos alguna función especificada con $\hookrightarrow$ en lugar de $\rightarrow$, esto lo haremos para hablar de funciones parciales.

El objetivo será definir la función semántica, $S : (Stm, State) \hookrightarrow State$, es decir, una función que dado un programa (Stm) y un estado ($State$) nos devuelva el estado resultante.

Necesitamos una función que nos permita evaluar expresiones aritméticas y booleanas. Este no es el propósito del trabajo, por lo tanto supondremos que existen 2 funciones, $A : (Aexp, State) \rightarrow \mathbb{Z}$ y $B : (Bexp, State) \rightarrow Bool, Bool \in \{true, false\}$, siendo $Aexp$ y $Bexp$ expresiones aritméticas y booleanas respectivamente, A y B nos permitiran evaluar expresiones aritméticas y booleanas respectivamente.

Por legibilidad, cuando una función tenga un estado $\in State$ como argumento, lo pondremos separado por un espacio.

Definimos la función identidad para nuestros estados como $id : State \rightarrow State$.

$$id\ state = state$$

Definimos una función que será necesaria para poder especificar bucles y condicionales,

$cond : ((State \to Bool), (State \hookrightarrow State), (State \hookrightarrow State)) \to (State \hookrightarrow State)$.

$$cond(p, g_1, g_2)\ state = \begin{cases} g_1\ state & \text{si } p\ state = true \\ g_2\ state & \text{si } p\ state = false \end{cases} \tag{3.1}$$

Es decir, *cond* es una función que recibe un predicado y dos funciones semánticas, y devolverá una u otra dependiendo del valor del predicado para un estado dado.

Además definimos la composición de funciones semánticas como:

$$(S[\![\mathbf{S}_2]\!] \circ S[\![\mathbf{S}_1]\!])\ state = \begin{cases} state'' & \text{si } \exists\ state', S[\![\mathbf{S}_1]\!]\ state = state' \text{ y} \\ & S[\![\mathbf{S}_2]\!]\ state' = state'' \\ \\ \bot & \text{si } S[\![\mathbf{S}_1]\!]\ state = \bot \text{ o} \\ & \exists\ state', S[\![\mathbf{S}_1]\!]\ state = state' \text{ y} \\ & S[\![\mathbf{S}_2]\!]\ state' = \bot \end{cases}$$

Esta función nos devolverá el resultado de la composición, a menos que alguna de las dos funciones diverja.

Denotaremos $f^n(d)$, $f : D \to D$ y $d \in D$ como la aplicación repetida de f sobre d, n veces.

$$f^0(d) = d \tag{3.2}$$

$$f^{n+1}(d) = f(f^n(d))$$

Para poder especificar el bucle indefinido tenemos que definir la función $fix : (D \to D) \to D$, que sirve para calcular el punto fijo de F. Un punto fijo de una función f, es un punto x tal que $f(x) = x$.

$$fix(F) = \bigcup_{i \geq 0} F^i(\bot) \tag{3.3}$$

Esta función fix toma como argumento una función de un dominio a ese mismo dominio, y devuelve otro elemento de ese dominio. En concreto este dominio D sera el de las funciones semánticas, es decir, en nuestro caso $D = (State \hookrightarrow State)$.

3.2.2.3. Semántica

En este apartado nos encargaremos de definir la función semántica encargada de dado un programa y un estado, devolver el estado resultante.

$$S[\![\textbf{right}]\!](p,\ i,\ c,\ v) = \begin{cases} (p + (1,\ 0),\ updateI(i,\ n),\ clearC(c,\ p_1,\ p_2),\ v) & \text{si } b = 1, \\ & (n, b) = c(p_1,\ p_2) \\ (p + (1,\ 0),\ i,\ c,\ v) & \text{en otro caso} \end{cases}$$

$$S[\![\textbf{left}]\!](p,\ i,\ c,\ v) = \begin{cases} (p - (1,\ 0),\ updateI(i,\ n),\ clearC(c,\ p_1,\ p_2),\ v) & \text{si } b = 1, \\ & (n,\ b) = c(p_1,\ p_2), \\ & p_1 \geq 1 \\ (p - (1,\ 0),\ i,\ c,\ v) & \text{si } b = 0, \\ & (n,\ b) = c(p_1,\ p_2) \\ & \text{si } p_1 \geq 1 \\ \bot & \text{en otro caso} \end{cases}$$

$$S[\![\textbf{down}]\!](p,\ i,\ c,\ v) = \begin{cases} (p + (0,\ 1),\ updateI(i,\ n),\ clearC(c,\ p_1,\ p_2),\ v) & \text{si } b = 1, \\ & (n,\ b) = c(p_1,\ p_2) \\ (p + (0,\ 1),\ i,\ c,\ v) & \text{en otro caso} \end{cases}$$

$$S[\![\textbf{up}]\!](p,\ i,\ c,\ v) = \begin{cases} (p - (0,\ 1),\ updateI(i,\ n),\ clearC(c,\ p_1,\ p_2),\ v) & \text{si } b = 1, \\ & (n,\ b) = c(p_1,\ p_2), \\ & p_2 \geq 1 \\ (p - (0,\ 1),\ i,\ c,\ v) & \text{si } b = 0, \\ & (n,\ b) = c(p_1,\ p_2) \\ & p_2 \geq 1 \\ \bot & \text{en otro caso} \end{cases}$$

Como se puede observar, la función semántica para los comandos de movimientos nos cambia la posición del agente (si no se puede diverge), en caso de que haya algún objeto que se pueda recoger, lo recoge.

$$S[\![\textbf{variable = info}]\!](p,\ i,\ c,\ v) = (p,\ i,\ c,\ updateV(v,\ variable,\ c(p_1,\ p_2)))$$

$$S[\![\mathbf{S}_1\ \mathbf{S}_2]\!]\ state = (S[\![\mathbf{S}_2]\!]\ \circ S[\![\mathbf{S}_1]\!])\ state$$

$$S[\![\textbf{variable = expression}]\!](p,\ \ i,\ c,\ v) = (p,\ i,\ c,\ updateV(v,\ variable,\ A[\![expression]\!]\ v))$$

Aquí podemos ver como hacemos uso de la función A para expresiones aritméticas de la que hablamos anteriormente para asignar un valor a una variable.

$$S[\![\textbf{repeat expression S end}]\!]\ state = (S[\![\mathbf{S}]\!])^n\ state$$

$$S[\![\textbf{if condition S end}]\!]\ state = cond(B[\![condition]\!],\ S[\![\mathbf{S}]\!],\ id)\ state$$

$$S[\![\textbf{if condition S}_1\ \textbf{else S}_2\ \textbf{end}]\!]\ state = cond(B[\![condition]\!],\ S[\![\mathbf{S}_1]\!],\ S[\![\mathbf{S}_2]\!])\ state$$

Para los condicionales hacemos uso de la función *cond* (3.1) que nos permite elegir entre 2 funciones dependiendo de un predicado.

Podríamos pensar que la función semántica de **do S while condition** es tan sencilla como:

$$\begin{aligned} S[\![\textbf{do S while condition}]\!]\ state = cond(&B[\![condition]\!] \circ S[\![\mathbf{S}]\!], \\ &S[\![\textbf{do S while condition}]\!] \circ S[\![\mathbf{S}]\!], \\ &S[\![\mathbf{S}]\!])\ state \end{aligned}$$

Pero esta definición no es composicional, pues se apoya en si misma, por lo tanto definiremos la semántica de **do S while condition** como el menor punto fijo de F. Es decir, se busca una función f tal que $F(f)\ state = f\ state$, $\forall state \in State$.

$$S[\![\textbf{do S while condition}]\!]\ state = fix(F_{\textbf{condition, S}})\ state$$

donde:

$$F_{condition,\mathbf{S}}(f)\ state = cond(B[\![\textbf{condition}]\!] \circ S[\![\mathbf{S}]\!],\ f \circ S[\![\mathbf{S}]\!],\ S[\![\mathbf{S}]\!])\ state \tag{3.4}$$

A esta función F se le exige que sea monótona y continua, tema que no trataremos aquí. Podemos encontrar demostraciones de la monotonía y continuidad de funciones similares en otros documentos a la hora de definir la semántica del bucle **`while`**[6][7], muy similar a **`do-while`**.

Caja 3.15 : Ejemplo de Semántica `do x = x + 1 while x != 2`

Consideremos $p =$ **`do x = x + 1 while x != 2`** con $state = (p,\ i,\ c,\ v)$.

$$S[\![p]\!] = fix(F_{x!=2,x=x+1}) \textbf{ donde}$$

$$F_{x!=2,x=x+1}(f)\ state = \text{ (Según 3.4)}$$

$$= cond(B[\![\mathbf{x\ !=\ 2}]\!] \circ S[\![\mathbf{x = x + 1}]\!],\ f \circ S[\![\mathbf{x = x + 1}]\!],\ S[\![\mathbf{x = x + 1}]\!])\ state =$$

$$\text{(Según 3.1)} = \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } v(x) = 1 \\ f\ (p,\ c,\ i,\ updateV(v,\ x,\ v(x)+1)) & \text{en otro caso} \end{cases}$$

Cálculos para la aproximación del punto fijo:

Por legibilidad denotaremos $f_k\ state = F^k(\bot)\ state$.

$$f_0\ state = F^0(\bot)\ state = \bot \text{ (según 3.2)}$$

$$f_1\ (p,c,i,v) = F(f_0)\ (p,\ c,\ i,\ v) =$$

$$= \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } v(x) = 1 \\ f_0\ (p,\ c,\ i,\ updateV(v,\ x,\ v(x)+1)) = \bot & \text{en otro caso} \end{cases}$$

$$f_2\ (p,c,i,v) = F(f_1)\ (p,c,i,v) =$$

$$= \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } v(x) = 1 \\ f_1\ (p,\ c,\ i,\ updateV(v,\ x,\ v(x)+1)) & \text{en otro caso} \end{cases}$$

$$= \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } v(x) = 1 \\ f_1\ (p,\ c,\ i,\ updateV(v,\ x,\ 1)) = (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } v(x) = 0 \\ f_1\ (p,\ c,\ i,\ updateV(v,\ x,\ v(x)+1)) = \bot & \text{en otro caso} \end{cases}$$

A partir de aquí podemos conjeturar su semántica y demostrarla por inducción.

$$f_n\ (p,\ c,\ i,\ v) = \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } 2 - n \leq v(x) \leq 1 \\ \bot & \text{en otro caso} \end{cases} \tag{3.5}$$

Caso Base: n = 0

Sabemos por definición (3.2) que $f_0\ state = \bot$, si sustituimos en (3.5) obtenemos:

$$f_0(p,\ c,\ i,\ v) = \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } 2 \leq v(x) \leq 1 \\ \bot & \text{en otro caso} \end{cases}$$

Y por lo tanto $f_0\ state = \bot$, ya que no se puede dar el primer caso.

Caso inductivo: n = k + 1

Asumiendo $f(k)$ queremos demostrar

$$f_{k+1}(p,\ c,\ i,\ v) = \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } 1 - k \leq v(x) \leq 1 \\ \bot & \text{en otro caso} \end{cases}$$

Por definición $f_{k+1}\ state = F(f_k)\ (p, c, i, v)$

$$= \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } v(x) = 1 \\ f_k\ (p,\ c,\ i,\ updateV(v,\ x,\ v(x) + 1) & \text{en otro caso} \end{cases}$$

Por Hipótesis Inductiva tenemos que:

$$f_k\ (p,\ c,\ i,\ v) = \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } 2 - k \leq v(x) \leq 1 \\ \bot & \text{en otro caso} \end{cases}$$

Por lo tanto:

$$F(f_k)\ (p,\ c,\ i,\ v) = \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } v(x) = 1 \\ (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } 1 - k \leq v(x) \leq 0 \\ \bot & \text{en otro caso} \end{cases}$$

$$= \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } 1 - k \leq v(x) \leq 1 \\ \bot & \text{en otro caso} \end{cases}$$

Esta es la definición de f_{k+1}.

Por lo tanto, queda demostrada nuestra conjetura y

$$S[\![\textbf{do x = x + 1 while x != 2}]\!]\ state$$

$$= fix(F_{\textbf{x != 2, x = x + 1}})\ state$$

$$= \bigcup_{i \geq 0} F^i_{\text{x != 2, x = x + 1}}(\bot)$$

$$= \bigcup_{i \geq 0} f_i\ state$$

$$= \begin{cases} (p,\ c,\ i,\ updateV(v,\ x,\ 2)) & \text{si } v(x) \leq 1 \\ \bot & \text{en otro caso} \end{cases}$$

$$= cond(B[\![\textbf{x <= 1}]\!],\ (p,\ c,\ i,\ updateV(v,\ x,\ 2)),\ \bot)\ state$$

Es decir, si $x \leq 1$, la variable x acaba con valor 2, en cualquier otro caso el programa diverge.

3.3. Tareas de Entorno Mínimo

En esta sección vamos primero a proponer un currículo que recogerá los conceptos computacionales que consideramos básicos a la hora de aprender programación. Después de esto nos centraremos en proponer un modelo de tarea para cada uno de estos conceptos.

3.3.1. Currículo de programación

El currículo propuesto está organizado por unidades, cada una centrada en un concepto computacional.

Caja 3.16 : Currículo propuesto
Unidad 1: Instrucción Simple
Unidad 2: Secuencia de Instrucciones
Unidad 3: Bucle repeat con cuerpo simple
Unidad 4: Bucle repeat con cuerpo múltiple
Unidad 5: Secuencia de bucles repeat
Unidad 6: Bucles repeat anidados
Unidad 7: Variables
Unidad 8: Entrada / Salida
Unidad 9: Operadores Aritméticos
Unidad 10: Condicional If
Unidad 11: Condicional If-else
Unidad 12: Condicional If-else-if
Unidad 13: Bucle do-while
Unidad 14: Operadores Relacionales
Unidad 15: Operadores Lógicos

3.3.2. Tareas

En este apartado vamos a proponer tareas mínimas para cada unidad de forma que quede justificada la utilización de los conceptos que representan.

- Unidad 1: El objetivo de estas tareas será recoger un único objeto que estará en una de las cuatro posibles direcciones.

Figura 3.8: Tarea de instrucción simple

- Unidad 2: El objetivo de estas tareas será recoger varios objetos que estarán dispuestos por la cuadrícula.

Figura 3.9: Tarea de secuencia de instrucciones

- Unidad 3: El objetivo de estas tareas será recoger varios objetos que estarán dispuestos por la cuadrícula alineados en una dirección. Además, para asegurar el uso de bucle `repeat`, estableceremos un número de palabras y números máximo de tal forma que no se pueda resolver con instrucciones de movimiento individuales.

Figura 3.10: Tarea de `repeat` con cuerpo simple

- Unidad 4: El objetivo de estas tareas será recoger varios objetos que estarán dispuestos por la cuadrícula siguiendo algún patrón. Al igual que en el anterior, pondremos un límite de palabras y números de tal forma que solo se pueda resolver usando bucle `repeat`.

Figura 3.11: Tarea de `repeat` con cuerpo múltiple

- Unidad 5: El objetivo de estas tareas será recoger varios objetos que estarán dispuestos por la cuadrícula siguiendo varios patrones. Al igual que en el anterior, pondremos un límite de palabras y números de tal forma que solo se pueda resolver usando bucles `repeat`.

Figura 3.12: Tarea de secuencia de `repeat`

- Unidad 6: El objetivo de estas tareas será recoger varios objetos que estarán dispuestos por la cuadrícula siguiendo algún patrón que se repite. Al igual que en el anterior, pondremos un límite de palabras y números de tal forma que solo se pueda resolver usando bucles `repeat`.

Figura 3.13: Tarea de `repeat` anidados

- Unidad 7: El objetivo de estas tareas será observar como se comporta un programa dependiendo del valor de sus variables. De esta forma es sencillo comprender que el comportamiento de una variable es similar al de un valor cualquiera, con la peculiaridad de que se puede cambiar.

Figura 3.14: Tarea de Variables

Caja 3.17 : Tarea Unidad 7

```
1 tipos_tuercas = 3
2 columnas_tipo = 2
3
4 repeat tipos_tuercas * columnas_tipo
5   right
6   down
7   up
8 end
```

- Unidad 8: El objetivo de estas tareas será recoger el número de objetos que un agente auxiliar nos pida. Normalmente estarán alineados para ser resueltas usando un bucle `repeat`. Además, el agente auxiliar no devolverá un número fijo, sino que se comprobará todo un rango de valores para obligar al usuario a resolverla usando los recursos que se enseñan en dicha unidad.

Figura 3.15: Tarea de entrada/salida

- Unidad 9: El objetivo de estas tareas será recoger el número de tuercas resultante de hacer una operación aritmética con la información de dos agentes auxiliares.

Figura 3.16: Tarea de operadores aritméticos

- Unidad 10: El objetivo de estas tareas será entrar por la puerta por una de dos puertas que están alineadas. El programa se prueba para ambas puertas.

Figura 3.17: Tarea de `if`

- Unidad 11: El objetivo de estas tareas será entrar por una de dos puertas que están en direcciones distintas. El programa se prueba para ambas puertas.

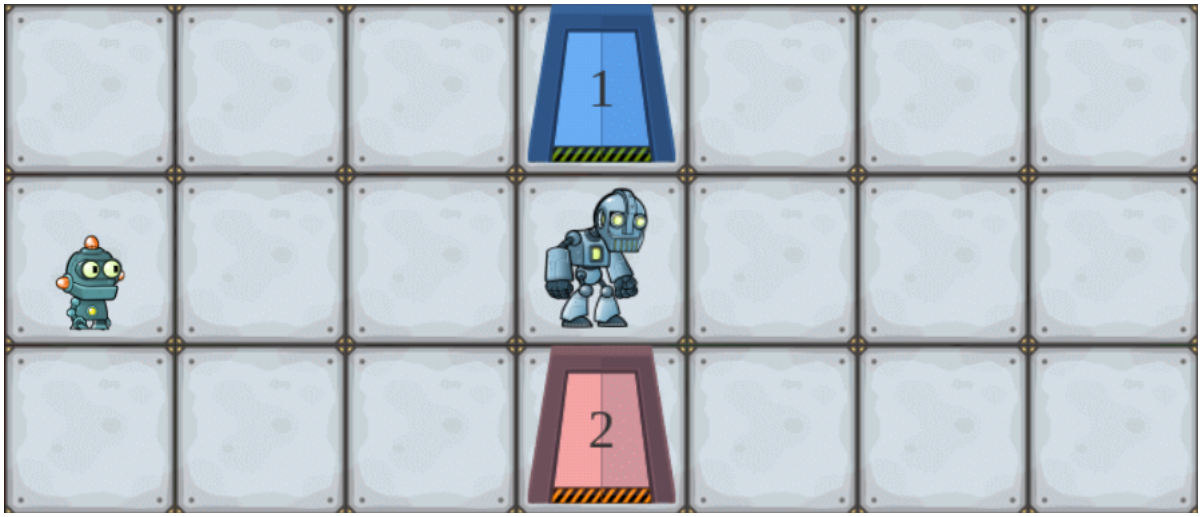

Figura 3.18: Tarea de `if` - `else`

- Unidad 12: El objetivo de estas tareas será entrar por una de varias puertas que están en cualquier disposición. El programa se prueba para todas las puertas.

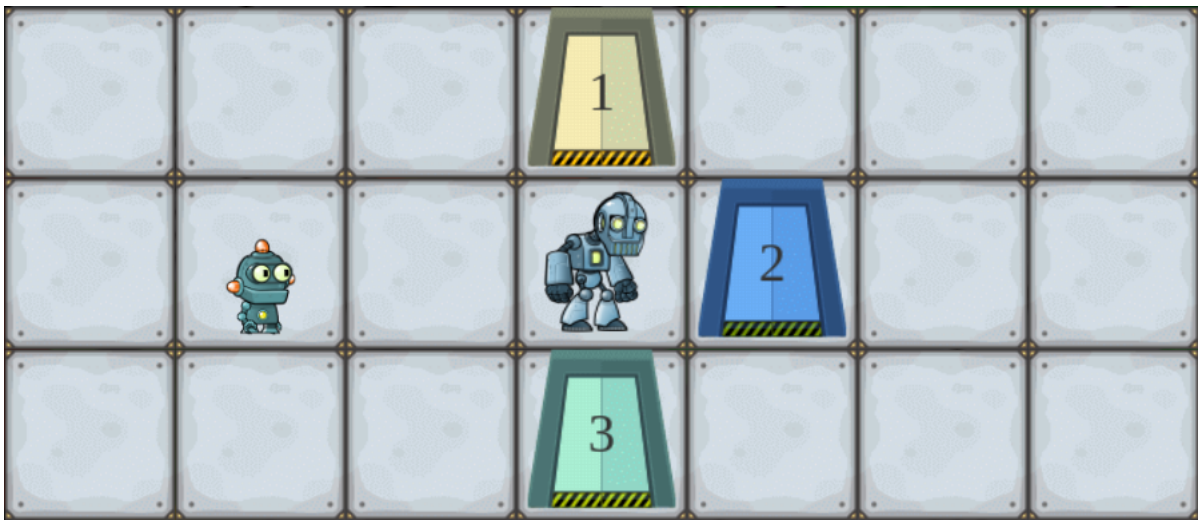

Figura 3.19: Tarea de `if` - `else` - `if`

- Unidad 13: El objetivo de estas tareas será encontrar la caja que tiene un número requerido de objetos en su interior. Solo hace uso de condiciones con == y `!=`. En estas tareas, se comprobará el programa para todas las permutaciones de los valores de las cajas.
- Unidad 14: El objetivo de estas tareas será encontrar la caja que tiene un número requerido de objetos dentro de un rango en su interior.
- Unidad 15: El objetivo de estas tareas será encontrar la caja que tiene un número requerido de objetos teniendo varias condiciones en conjunción o disyunción.

Figura 3.20: Tarea Unidades 13/14/15

CAPÍTULO 4

Conclusiones y líneas futuras

Como conclusión podemos decir que este proyecto, realizado en el marco de una beca de iniciación a la investigación, está centrado en la búsqueda de sencillez en todos los sentidos. Proponemos un modelo de computo centrado en la enseñanza que se apoya fuertemente en la visualización. Hemos cogido elementos de otros lenguajes buscando siempre reducir la dificultad en la sintaxis, ya que hay que tener en cuenta que esta herramienta será usada por personas que pueden no saber leer y escribir correctamente. Es decir, nuestro objetivo respecto a la sintaxis ha sido acercarlo al lenguaje natural y evitar paréntesis, llaves, etc, para no tener que realizar complicadas combinaciones de teclas para escribir caracteres especiales.

La sencillez es la razón por la que este lenguaje es tan reducido, de hecho, es la razón principal por la que incluimos el bucle `repeat` en lugar del bucle `for`. Usamos un sistema de movimiento absoluto, por la sencillez de planificar visualmente las acciones a realizar, sin tener en cuenta el ángulo en el que se encontrará el agente en un determinado momento.

Todas las estructuras computacionales que incluye quedan justificadas por las tareas que hay que resolver con ellas, forzando que se use el concepto computacional adecuado para completarla. Esto se puede ver especialmente en las tareas que incluyen puertas y cajas, en las que tuvimos que introducir ciertos elementos ya descritos para que desde el lado del usuario se perciba aleatoriedad. Gracias a esto nos aseguramos que no se realicen programas a medida y que los programas escritos sean generales.

Durante la realización de este proyecto hemos utilizado muchos scripts, tanto en *Bash* [8] como en *GNU Octave* [4]. Han sido usados principalmente para procesamiento de texto, pues las tareas se organizan en un formato propio, parecido a *JSON(JavaScript Object Notation)*, por lo que cualquier cambio que quisiéramos hacer en masa, se hizo usando técnicas como expresiones regulares.

Con respecto a las unidades, nos hemos centrado en que la dificultad entre ellas sea gradual. Hemos propuesto un currículo que cubra la programación básica destinada a los alumnos desde 1º de primaria hasta 2º de bachiller.

Aún queda trabajo por hacer en éste campo, por un lado, la generación automática de tareas es un campo interesante, en el que podemos utilizar técnicas como búsqueda con retroceso para generar caminos válidos de cierta longitud dentro de la cuadrícula y repartir objetos por este.

Otro tema interesante sería la utilización de este modelo para la enseñanza de otras asignaturas como Matemáticas, Física o incluso Lengua Castellana.

Finalmente esta herramienta nos abre la posibilidad de crear nuevos cursos para además de enseñar programación básica, añadir campos como Inteligencia Artificial, Programación Orientada a Objetos, Programación funcional, Algoritmia, etc. Incluso se podría estudiar la capacidad de aprendizaje de los estudiantes de los cursos K-12 para aprender sobre todos estos campos.

Bibliografía

[1] Michele Van Dyne y Jeffrey Braun. «Effectiveness of a Computational Thinking (CS0) Course on Student Analytical Skills». En: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. Atlanta, Georgia, USA: Association for Computing Machinery, 2014, págs. 133-138. ISBN: 9781450326056. DOI: 10.1145/2538862.2538956. URL: https://doi.org/10.1145/2538862.2538956.

[2] F. Vico, M. Molina, D. Orden, J. Ortiz, R. García y J. Masa. «A coding curriculum for K-12 education: the evidence-based approach». En: *EDULEARN19 Proceedings*. 11th International Conference on Education and New Learning Technologies. Palma, Spain: IATED, jul. de 2019, págs. 7102-7106. ISBN: 978-84-09-12031-4. DOI: 10.21125/edulearn.2019.1698. URL: http://dx.doi.org/10.21125/edulearn.2019.1698.

[3] F. Vico, M. Molina, D. Orden, F. Rivas-Ruiz, R. García y J. Masa. «coding skills are acquired gender-independently in the K-12 system: the Toolbox Academy experience». En: *EDULEARN19 Proceedings*. 11th International Conference on Education and New Learning Technologies. Palma, Spain: IATED, jul. de 2019, págs. 7076-7079. ISBN: 978-84-09-12031-4. DOI: 10.21125/edulearn.2019.1692. URL: http://dx.doi.org/10.21125/edulearn.2019.1692.

[4] John W. Eaton, David Bateman, Søren Hauberg y Rik Wehbring. *GNU Octave version 5.2.0 manual: a high-level interactive language for numerical computations*. 2020. URL: https://www.gnu.org/software/octave/doc/v5.2.0/.

[5] *Referencia de JavaScript*. URL: https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia.

[6] Hanne Riis Nielson y Flemming Nielson. *Semantics with Applications: An Appetizer (Undergraduate Topics in Computer Science)*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 1846286913.

[7] Roberto Bruni y Ugo Montanari. *Models of Computation*. 1st. Springer Publishing Company, Incorporated, 2017. ISBN: 3319428985.

[8] Case Western Reserve University Chet Ramey y Free Software Foundation Brian Fox. *Bash Reference Manual*. Mayo de 2019. URL: `https://www.gnu.org/software/bash/manual/bash.pdf`.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga