Trabajo fin de grado sobre la construcción de un entorno de aprendizaje pre-universitario para la programación.

Ezequiel Santamaría Navarro

16 de julio de 2016

Índice general

Resumen

Proceso de creación de un lenguaje sencillo, de un entorno de programación sin instalación y con herramientas para facilitar el desarrollo.

Capítulo 1

Introducción

1.1. Objetivo e intenciones de este trabajo

La enseñanza preuniversitaria de la programación consiste en enseñar a adolescentes y niños a poder pensar de forma lógica y abstracta. Esto conlleva un conjunto de problemas distintos al de la enseñanza a nivel universitario. Antes de la universidad no se puede asumir si los alumnos conocen o no ciertas partes de las matemáticas, si saben o no dibujo técnico, o incluso si entienden como funciona un espacio de coordenadas cartesianas.

En mi caso concreto, la primera vez que empecé con un lenguaje de programación (uno que se use de forma profesional) fue Pascal con Delphi (en 2007). Fue mi hermano mayor quien me ayudó a instalar tanto el IDE como quien me explicó algunas directrices para empezar a hacer aplicaciones muy básicas.

Durante dos años intenté hacer dos cosas muy concretas con Delphi que no conseguí y que me causaron frustración: dibujar (como en un lienzo, algunos gráficos) y conectar la aplicación a internet con otra copia de la aplicación.

La sensación que he tenido desde entonces hasta hoy es que los lenguajes de programación están orientados a ser útiles desde la perspectiva de un ingeniero, pero no de facilitar el aprendizaje.

Por ello, mi objetivo sería conseguir que el proyecto cumpla los siguientes puntos:

- El lenguaje debe estar en la medida de lo posible en castellano, para no asumir que el alumno entiende inglés.
- El lenguaje preferiblemente será tiapdo, con comprobación de tipos [13][14] y sin operaciones de memoria [12] (como los punteros de C o C++, que permiten el bypass de tipos).
- El lenguaje tendrá tipos de datos con nombres comunes. Por ejemplo, se evitará el uso de Mapa y Vector, y se usarán nombres como Relación y Lista respectivamente.

- No debe asumir automáticamente conversiones de datos. Bajo mi perspectiva, el alumno debe ser consciente de que los datos tienen un representación que se puede convertir, pero que el ordenador no es inteligente y hay que dar órdenes explícitas.
- Que no requiera instalador, importantísimo para evitar frustraciones con el software.
- Si fuera posible, integrable con la herramienta Descubre, para aprovechar el sistema de usuarios y de compartir los ejemplos que ya tiene.
- Que sea flexible para que los docentes puedan añadir funcionalidad, ampliando el conjunto de problemas, ejemplos, soluciones y aplicaciones.

Estos son los puntos fundamentales que podrían simplificar el aprendizaje del desarrollo algorítmico y la programación.

1.2. Ideas extraídas de los distintos proyectos

Las ideas que más me han gustado y que desarrollo en el siguiente capítulo son:

- Editor y ejecución (compilación transparente) en una misma web como Descubre o CodeCombat.
- Un lenguaje propio simplificado como iJava.
- Mezcla de imperativo y con objetos (aunque mayoritariamente imperativo) con una lista de subrutinas para dibujar como Processing.
- Intentar advertir de los errores de compilación con lenguaje sencillo evitando tecnicismos si es posible, como con el editor de Khan Academy
- Ofrecer un editor que ayude al alumno a escribir el código, sugiriendo nombres, autocompletando según el contexto...

Capítulo 2

Estado del arte

2.1. Entornos de aprendizaje a la programación

Investigando proyectos con puntos comunes a lo que busco, estos son los proyectos que he podido ver y me han sorprendido en algún o varios aspectos. Existen muchos más, pero estos ofrecen en conjunto lo mismo (o lo más destacable) que el resto.

2.1.1. Descubre

La herramienta Descubre, desarrollada por Juan Antonio Sánchez Laguna, Marcos Menárguez Tortosa, y Juan Antonio Martínez Navarro.

La herramienta se compone de:

- Un editor de texto, derivado de CodeMirror.
- Un lenguaje de programación propio, derivado de Java, con un las funcionalidades de processing.
- Un entorno con un lienzo para dibujar y una salida para escribir texto.
- Un repositorio de usuarios y programas que permite copiar el código de otros para hacer clones.

Con los siguientes puntos positivos:

- Entorno completo. Desde escribir el código hasta ver su resultado en la misma página.
- Sin instalación y sin necesidad de conocimientos sobre compilaciones.
- El lenguaje comparte sintaxis con C y Java, lo que es útil para avanzar más allá de la enseñanza.

- La página tiene una lista de tutoriales y documentación sobre la funcionalidad de processing.
- Se puede clonar con facilidad el código de otros alumnos o docentes.

Y los siguientes puntos negativos:

- El lienzo es un poco pequeño. Se limita a 320x320. Tiene una versión estirada del mismo que se ve pixelada, y otros sistemas (como por ejemplo, codecombat, Apéndice B)
- La sintaxis de iJava no es amigable para nuevos alumnos.
- La ejecución del código es síncrono. Esto reduce las posibilidades tecnológicas. Como por ejemplo, no se pueden cargar imágenes en tiempo de ejecución.
- No se puede rotar el lienzo, así que todos los dibujos deben estar alineados a los ejes de coordenadas.
- No se puede extender el lenguaje con el propio lenguaje. Es decir, no se puede escribir un módulo para que se incluya en otro código. Esta parte no es importante para el alumno, pero sí para el docente que quiera extender el lenguaje.

Acceso: `http://descubre.inf.um.es`

2.1.2. Codecombat

CodeCombat es un proyecto de código abierto para aprender a programar en base a un juego multijugador. Actualmente tiene varios problemas sencillos (que van creciendo en dificultad) para ir aprendiendo los mecanismos de los distintos lenguajes que ofrece, a la vez que vas ganando monedas para poder comprar objetos que ofrecen mayor funcionalidad (te permiten usar más métodos para resolver los distintos problemas).

Se pueden destacar los siguientes puntos del proyecto:

- Gran conjunto de problemas a resolver que van creciendo en dificultad.
- Permite seleccionar entre Python, JavaScript, Clojure, CoffeScript y Lua.
- Acabado artístico profesional, así como una gran cantidad de objetos.
- Proyecto de código abierto.
- Traducido (parcialmente) a varios idiomas, entre los cuales se encuentra el español.

Sin embargo, el objetivo de esta plataforma es aprender resolviendo problemas concretos como moverse por un laberinto, o eliminando unos enemigos, pero no crear otras aplicaciones. Aún así, es una muy buena herramienta de aprendizaje.

Como punto negativo destacaría que te obligan a programar en un lenguaje real, sin ofrecerte una alternativa amigable para nuevos alumnos. Que por defecto, Python, tiene problemas con la indentación obligatoria, que podría ser una barrera.

Se puede probar sin registro. Hay una imagen de ejemplo de la aplicación en el apéndice B de CodeCombat.

Acceso: `http://codecombat.com`

2.1.3. Scratch

Scracth es un proyecto impulsado por el MIT para el desarrollo de pequeños proyectos con dibujos, animaciones y sonidos.

Se basa en programar unos módulos encajando unas piezas que equivalen a sentencias, instrucciones o elementos de control en un lenguaje por bloques, como si fuera un juego de Lego en dos dimensiones.

El lenguaje y el editor tienen versión web y versión local, que se encuentra en paquetes como la Raspberry Pi. En la versión web, se pueden alojar proyectos para que sean de dominio público y cualquier persona puede clonarlo para alterarlo a su gusto.

De este proyecto cabe destacar la gran cantidad de usuarios, así como la interfaz y la posibilidad de no necesitar la web para hacer un proyecto. Además, los bloques que conforman el código están traducidos a varios idiomas, reduciendo la barrera de los alumnos que aprenden en castellano.

La única pega que le veo al proyecto es que la programación es por bloques, y que la interfaz, en mi opinión, está enfocado a alumnos muy jóvenes (de edades entre 6 y 12 años).

Acceso: `https://scratch.mit.edu`

2.1.4. code.org

Code.org es una web con varios tipos niveles de problemas, donde hay un lienzo representando un problema. Los alumnos (recomendado de 4 a 18 años) resuelven unos problemas que crecen progresivamente en dificultad. A diferencia de codecombat, code.org utiliza un lenguaje basado en bloques como el de Scratch.

Los problemas son visualmente muy entretenidos (como los de codecombat), aunque más enfocados a alumnos más jóvenes (desde 4 años).

Al igual que Scratch, esta web ofrece los bloques de código en castellano, y unos vídeos introductorios interesantes con subtítulos al español.

La pega que le veo es la misma que a Scratch y Codecombat: solo ofrecen un mecanismo para resolver problemas, con una interfaz basada en bloques y con una temática enfocada a niños pequeños.

Acceso: `http://code.org`

2.1.5. Khan Academy - Aprendiendo a dibujar con Javascript

En Khan Academy hay un conjunto de tutoriales para aprender JavaScript con el objetivo de dibujar en un lienzo como el de Descubre. Tiene muchos tutoriales y son todos interactivos y subtitulados al castellano.

De aquí cabe destacar que el editor es muy interactivo. Permite la edición de casi cualquier valor con una especie de elemento visual. Además, el código se compila automáticamente (en el caso de javascript se podría evaluar directamente) y se pinta en el lienzo cada vez que el código cambia.

Acceso: Intro to Drawing

2.2. Compiladores y analizadores escritos en javascript

2.2.1. Jison - Bison para javascript

Jison es un proyecto escrito en Javascript que ofrece las herramientas similares a GNU Bison, un analizador de gramáticas libres de contexto de tipo LALR(1) [4].

Según mis cálculos (indicados más adelante) en el apartado *Definición del lenguaje ZL* es necesario al menos un analizador que avance 3 tokens antes de conocer la derivación a realizar. Así que descarto Jison ya que no creo que sea suficientemente potente para analizar ZL.

Acceso: Github de jison

2.2.2. Analizador descendente recursivo por prioridad de operadores

El analizador descendente (Top Down) por orden de prioridad de operadores (Operator Precedence) fue presentado por Vaughan Pratt en el año 73 [8]. El analizador se basa en definir unos símbolos, unas expresiones y unos operadores (indicando su prioridad de forma numérica) con funciones recursivas que indican el comportamiento a la hora de ir avanzando recursivamente.

En principio mi plan era utilizar este analizador como LL(3), pero mientras avanzaba con la implementación del análisis, ceñirse estrictamente al uso de este analizador era cada vez más difícil.

Sin embargo, este tipo de análisis ha inspirado la forma en la que he escrito el analizador sintáctico, usando reglas de derivación de forma recursiva donde en cada una de ellas se indica cómo se debe avanzar y cuál es el resultado generado.

Acceso: Texto de Douglas Crockford

2.2.3. Analizador de gramáticas de expresiones

2.3. Editores de texto para Javascript y HTML

Lo primero que he encontrado ha sido el editor Codemirror. Es el mismo editor que usa el proyecto Descubre de la facultad. No me he dedicado en buscar otros editores porque este tiene todo lo que necesito para poder preparar el entorno, y además su API está muy bien documentada[2].

Acceso: Página web oficial de codemirror.

Capítulo 3

Análisis de objetivos y metodología

Los objetivos principales han sido crear un lenguaje con palabras cercanas al lenguaje común español, un editor de texto amigable y un compilador de dicho lenguaje, buscando en la medida de lo posible el respaldo de unos usuarios a los que medía su grado de entedimiento.

He empleado una metodología que he separado en tres pasos: investigación, implementación y feedback. Con esos tres pasos he hecho varias iteraciones. En esta sección resumo la implenetación ya que más adelante se habla en detalle de las implementaciones.

El primer objetivo era definir el lenguaje. Aunque ha ido cambiando a lo largo de todo el proyecto, gran parte de lo definido al principio no ha cambiado. Aquí la investigación ha sido básicamente encontrar una forma de definir el lenguaje, en el cual me acabé decantando por la forma de BNF[3] con alguna modificación, aunque también estudié la posibilidad de usar DCG[10] pero la descarté por parecer más compleja de entender.

La implementación ha sido sencillamente escribir la sintaxis en formato BNF (en el apéndice A o el fichero sintaxis.txt), y el feedback ha sido preguntar a los usuarios sobre que estructuras y palabras les han parecido más sencillas de comprender dando a elegir entre varias opciones (las constantes booleanas verdadero/falso o cierto/falso o verdadero/mentira, entre otras).

El segundo objetivo es escribir el compilador, pero para ello necesito un editor para poder ir haciendo las pruebas, así que como objetivo intermedio se hace el editor. La investigación aquí es corta: ya conozco codemirror y además se usa en Descubre para iJava, sé que ofrece un plugin sencillo para colorear texto[5] y un plugin para autocompletar [1].

La implementación para este objetivo es relativamente sencilla, y se divide entre los ficheros **index.html** (se instancia el editor), **autocompletar.js** (se

definen las órdenes de autocompletado) y **editor.js** (se definen las órdenes de coloreado de sintaxis). Aquí el feedback que he podido obtener de los usuarios ha sido relativo a la facilidad de lectura del texto y a la facilidad del autocompletado, habiendo cambiado el algoritmo de autocompletado gracias a las indicaciones de los usuarios (de un algoritmo LCS a uno que use distancias de Levensthein).

Una vez definido lenguaje y editor, falta el último objetivo, el más complejo y quizá el más importante: el compilador. Al principio, al ver las distintas alternativas de analizadores escritos en Javascript opté por usar pegjs (en la siguiente sección se explica con más detalle), pero al final he escrito mi propio analizador. La diferencia entre escribir mi propio analizador y usar otro tampoco es tan grande ya que después de analizar la gramática libre de contexto hay que generar una tabla de símbolos, hacer comprobaciones semánticas y luego transformar a Javascript, que es la parte más compleja con diferencia.

El compilador se separa en varios ficheros, pero como son muchos entraré en más detalle sobre la implementación en la sección de la estructura del compilador para ZL. De aquí no puedo obtener mucho feedback salvo la generación de errores si el código escrito no pasa alguna de las fases de compilación, para ver si los usuarios son capaces de entender los errores y corregirlos.

3.1. Desarrollo de un entorno de programación

Al igual que Descubre o Codecombat, el principal objetivo del trabajo es crear un entorno para facilitar la programación. Este sistema debe estar compuesto de al menos un editor de texto, el compilador del lenguaje diseñado y las herramientas de entrada y salida (barras de texto para escribir, lienzos para dibujar, eventos de teclado...). Recomendablemente el entorno debe tener información sobre depuración (errores al escribir el código, información de depuración al pausar el código, etc...) como tienen también otros entornos (Descubre o Codecombat, por ejemplo).

3.1.1. Compilador del lenguaje ZL

En una primera aproximación el compilador se construyó usando PEG.js[6], que en su descripción dice ser más potente que LL(k) o LR(k). Sin embargo, no fui capaz de hacer análisis de un trozo ambiguo de ZL. El problema lo localicé al intentar analizar el siguiente trozo de pseudo-ZL (en los inicios, la sintaxis era mucho más sencilla):

```
x es Booleano
x <- falso
Si no x hacer // if (!x)
        mostrar [
                mensaje <- "x es falso"
        ]
si no hacer // else
        mostrar [
                mensaje <- "x es verdadero"
        ]
fin
```

Aquí localicé tres problemas de los cuales supe resolver 2. El primero es diferenciar 'si no **x**' de 'si no **hacer**'. Se resuelve fácilmente diferenciando gramaticalmente un nombre (por ejemplo, 'x') de ciertas palabras reservadas como 'hacer'. Con esto se fuerza un error si el analizador se encuentra con 'si' e intenta reducir/derivar 'no hacer' como una expresión, ya que para que fuese una expresión 'hacer' no debería ser una palabra reservada, sino un nombre.

El segundo problema es, precisamente, evitar el uso de la palabra **hacer** como nombre, pero se resuelve fácilmente usando expresiones regulares negativas (Javascript permite lo que se llama negative lookahead[9]).

El tercer problema, que no he sido capaz de resolver usando PEG.js, viene a ser que, si se escribe mal el código del segundo si condicional, el compilador advierte de que no se pudo resolver el primer si condicional. Por ejemplo, si en vez de escribir 'si no hacer' se escribiera 'si mo hacer' el compilador indicaría error en el primer si condicional, debido a que si no se puede reducir correctamente el segundo si no hacer, lo intenta analizar después como un si condicional, pero como 'mo hacer' no es una expresión correcta, no se puede reducir/derivar el primer si condicional, dejando ahí marcado el error.

Viendo la serie de problemas que tenía con gramáticas sencillas pero ambiguas, lo único que quedaba era o alterar los compiladores ya escritos o implementar uno por cuenta propia. Uno que ofrezca los mismos mecanismos que los otros compiladores pero que me permita romper el comportamiento por defecto (que admita cierta flexibilidad) para añadir información de error, o para resolver el problema de los múltiples si condicionales. Al final me decanté por la segunda opción, con la esperanza de que llevase menos tiempo que alterar PEG.js.

3.1.2. Editor de texto inteligente

Los puntos planteados para hacer el editor inteligente pasan básicamente por dotar a un editor la capacidad de autocompletar. El lenguaje es verboso y los datos son tipados, haciendo posible la inferencia desde el editor de las posibles y diferentes compleciones. Como se ve en la siguiente sección (Definición del lenguaje ZL), la asociación de argumentos de subrutinas es por nombre. Esto

hace que el lenguaje sea más fácil de leer sin memorizar, aunque más difícil de escribir:

```
En iJava, por ejemplo:
ellipse(100, 200, 50, 50);

En ZL:
elipse [
        centro <- {100, 200} como punto
        ancho <- 50
        alto <- 50
]
```

Para reducir la dificultad a la hora de escribir[1], es importante un buen editor que ayude a autocompletar, sugiriendo nombres y escribiendo los distintos argumentos de las subrutinas, como por ejemplo:

A izquierda, el menú contextual que aparece al escribir 'elip'. A la derecha, lo que aparece **inmediatamente después** de pulsar *enter* o seleccionar con el ratón. 'Ancho' y 'alto' se autocompletan con un número por defecto (3 es el número por defecto elegido arbitrariamente), y centro se autocompleta con el dato 'c', inferido a través de los tipos de los datos.

El funcionamiento y la implementación del autocompletado se ve más adelante en la sección **Editor inteligente - Autocompletado**.

3.2. Definición del lenguaje ZL

A la hora de definir el lenguaje ZL me centré en varios aspectos que tomé como fundamentales: que fuese cercano al lenguaje natural, y que estuviese en español[7].

Parte del léxico usado para definir el lenguaje está escogido en base a probar a personas con diferentes formaciones: sin formación informática, con nociones pero sin conocimiento de programación, ex-programadores... Las pruebas han sido, básicamente, medir su grado de entendimiento de las estructuras del lenguaje con diferentes formas.

[1] La parte del código que pone 'como punto' es un conversor que indica que {100, 200} es un punto bidimensional.

Los nombres escogidos se acercan más al lenguaje cotidiano que al lenguaje informático. Se ha preferido nombres como 'Lista' y 'Relacion' frente a 'Vector' o 'Mapa'. Se ha escogido el nombre 'Dato' frente a 'Variable'. Se usan palabras para abrir y cerrar bloques como 'Hacer' y 'Fin' en vez de llaves {}.

Por haber elegido la construcción de un lenguaje cercano al lenguaje natural aparecen repeticiones, sobretodo por usar las mismas palabras reservadas en contextos distintos: 'si' para indicar el equivalente a 'if' o 'else' (en C), 'no' para indicar negación de verdad o 'else'... Esto causa que el lenguaje sea LL(3) (para ver el lenguaje en formato BNF extendido, ver apendice A) en la siguiente regla de derivación:

```
<sicondicional> ::= "si" <_> <expresion> <_> "hacer" <_> (<sentencia> <_>)+ "fin"
| "si" <_> <expresion> <_> "hacer" <_> (<sentencia> <_>)+ <sinocondicional>
| "si" <_> <expresion> <_> "hacer" <_> (<sentencia> <_>)+ <sino>
```

A simple vista se puede ver como es LL(2), al empezar todas las reglas con "si". Después de factorizar el "si", es donde viene el problema. La primera regla es un 'if' (de C) seguido de ningún 'else' y ningún 'else if'. La segunda regla, de un 'if' seguido de al menos un 'else if'. La tercera regla, un 'if' acabado de un 'else'. El problema viene con la tercera regla, ya que:

```
<sino> ::= "si" <_> "no" <_> "hacer" <_> (<sentencia> <_>)+ "fin"
```

```
// Ejemplo de código de conflicto LL(3)
Si x hacer
        Si no y hacer
                // ...
        Fin
Si no hacer
        // ...
Fin
```

El conflicto[11] viene que hasta que no se lee 'Si no {siguiente token}' no se puede distinguir entre un 'if' anidado dentro de otro 'if' (Si no y hacer) y un 'else' (Si no hacer).

3.2.1. Descripción de la notación BNF ampliada

La notación BNF que utilizo en el Apéndice A es una notación BNF a la cual he añadido paréntesis para agrupar, y los operadores *, + y ? como operadores de repetición. Funcionan como en las expresiones regulares: * para indicar cero o más veces, + para indicar uno o más veces, y ? para indicar cero o una veces.

Los siguientes ejemplos son equivalentes en BNF:

```
BNF extendido:
a ::= b c?
x ::= (y z)*
j ::= k (l m)+

BNF:
a ::= b
    | b c
j_plus ::= l m
    | j_plus l m
j ::= k j_plus
x ::=
    | y z x
```

3.2.2. Gramática BNF del lenguaje

La gramática del lenguaje en formato BNF se puede ver en el Apéndice A. De ella, cabe destacar de la gramática las 4 expresiones y los cuatro grupos de operadores, para poder definir correctamente el orden en el cuál los operadores se reducen. En la implementación no uso esa técnica para definir el orden de las operaciones, sino que una vez construido el árbol de la expresión, cambio sus ramas para que respeten el orden de los operadores.

Otros conceptos destacables en la gramática es el uso de diferentes tipos de enumerar (en hexadecimal, decimal, o binario).

3.2.3. Descripción de la semántica

Las estructuras semánticas del lenguaje se puede clasificar en módulos, subrutinas, algoritmos, datos y configuraciones. Parte de la semántica es compleja y está enfocada para los posibles docentes y para usuarios avanzados. Los alumnos nuevos que quieran aprender pueden centrarse en escribir solo algunas subrutinas e ignorar todo el resto de las estructuras.

Para darle flexibilidad al lenguaje, el lenguaje está pensado para escribirse en módulos. Los módulos se pueden ver como unidades de código que se pueden compilar, y a su vez los módulos pueden incluir otros módulos. Un módulo puede tener configuraciones, y puede tener subrutinas.

Las subrutinas son la estructura básica de ejecución. En una subrutina se define un algoritmo, que indica el comportamiento de la subrutina, así como los datos que el algoritmo va a usar. A su vez, la subrutina puede tener modificadores que alteran el comportamiento del código sobre la subrutina. Por ejemplo, el modificador **subrutina primitiva** permite ejecutar código nativo javascript en una subrutina (ver más adelante la sección Código nativo), que está definida en ZL.

Los datos, dentro de una subrutina, se definen por un nombre, un tipo, y un ámbito: local, de entrada, de salida, de entrada y salida, o global. Cuando un dato es de tipo local, solo la subrutina que lo define conoce su valor y puede acceder a él. Los datos de ámbito de entrada se proporcionan al llamar a la subrutina, y no se pueden escribir sobre ellos. Los datos de ámbito de salida se deben proporcionar en la subrutina y son leídos (opcionalmente) al llamar a la subrutina. Los datos de tipo global tienen ámbito **dentro de un módulo**. Todas las subrutinas que definan el mismo dato (que debe, por obligación de compilación, coincidir en tipo y nombre) podrán compartir el valor, y estos datos deben inicializarse en la **subrutina inicio** de cada módulo (cada módulo inicializará sus datos globales), usándose esa subrutina como constructor.

Una configuración define constantes que pueden ser útiles para el entorno (como por ejemplo, el número de fotogramas por segundo, cuántos decimales se imprimen por defecto, el ancho y el alto del lienzo...), además de permitir incluir módulos de dos formas distintas: **integrar** o **importar**. **Integrar** un módulo equivaldría a un '#include' de C o C++, donde el código se 'copia y pega' en el lugar de la directiva. Las subrutinas del módulo integrado forman parte del entorno del módulo que lo integra. Por otra parte, **importar** un módulo equivale a un 'import' de Java. Se crea un nuevo tipo de dato (una clase si hablamos en términos Java) cuyas subrutinas pasan a ser parte de ese tipo (como si fuesen miembros, en Java).

Capítulo 4

Diseño y resolución del trabajo realizado

4.1. Estructura del compilador para ZL

4.1.1. Análisis léxico

4.1.2. Análisis sintáctico

4.1.3. Análisis semántico

4.1.4. Código nativo, implementación de las subrutinas ZL nativas

4.2. Editor inteligente - Autocompletado

4.2.1. Determinando el contexto del cursor en el editor

4.2.2. Algoritmos de similitud de nombres

4.3. Triple plataforma - Web, Electron y NodeJS

Capítulo 5

Conclusiones y vías futuras

5.1. Posibles mejoras

Bibliografía

[1] Codemirror, *Codemirror: Autocomplete demo*, https://codemirror.net/demo/complete.html, 2016.

[2] ———, *Codemirror manual*, https://codemirror.net/doc/manual.html#overview, 2016.

[3] Lars Marius Garshol, *Bnf and ebnf: What are they and how do they work?*, http://www.garshol.priv.no/download/text/bnf.html, 2008.

[4] GNU, *Bison - gnu project*, https://www.gnu.org/software/bison/, 2014.

[5] Ingo, *Codemirror (source) syntax highlighting*, http://ckeditor.com/addon/codemirror, 2015.

[6] David Majda, *Peg.js*, http://pegjs.org, 2010-2016.

[7] El mundo, *El 63 % de los alumnos españoles no comprende el inglés al finalizar la eso*, http://www.elmundo.es/elmundo/2012/06/21/espana/1340281749.html, 2012.

[8] Vaughnan R. Pratt, *Top down operator precedence*, http://dl.acm.org/citation.cfm?id=512931, 1973.

[9] Tim Vroom, *Using look-ahead and look-behind*, http://www.perlmonks.org/?node_id=518444, 2005.

[10] Wikipedia, *Definite clause grammar*, https://en.wikipedia.org/wiki/Definite_clause_grammar, 2016.

[11] ———, *Ll parser, solutions to ll(1) conflicts*, https://en.wikipedia.org/wiki/LL_parser#Solutions_to_LL.281.29_Conflicts, 2016.

[12] ———, *Memory safety*, https://en.wikipedia.org/wiki/Memory_safety, 2016.

[13] ———, *Static type checking*, https://en.wikipedia.org/wiki/Static_type-checking, 2016.

[14] ———, *Type safety*, https://en.wikipedia.org/wiki/Type_safety, 2016.

Apéndice A

Notación BNF (extendida) del lenguaje

```
; Insensible a mayúsculas y minúsculas
; Notación BNF
; <_> significa espacio en blanco (cualquier tipo de espacio en blanco), o un comentario
; Algunos de los espacios son opcionales, allá donde no haya ambiguedad:
; por ejemplo: a.b o a<-b no es ambiguo con a . b ni con a <- b
; la notación está extendida con * y con +, que tienen el mismo significado que
; en regex, así como paréntesis para agrupar y [] para opcionalidad.

; Reglas gramaticales
; nombreSimple expresión regular ([A-Za-záéíóúÁÉÍÓÚñÑ][A-Za-záéíóúÁÉÍÓÚñÑ0-9]*)
; entero expresión regular ((?:[0-1]+(?:\|2))|(?:[0-9A-Fa-f]+(?:\|16))|(?:[0-9]+(?:\|10)?))
; decimal expresión regular (\d+\.\d+)
; texto expresión regular \"([^"\\]|\\.|\\\n)*\"
; letra expresión regular \'([^"\\]|\\.|\\\n)*\'
; comentario expresión regular \/\/.*

; Nota, para las expresiones del mismo nivel,
; el arbol que se construye se reordena para evaluar primero a la izquierda.

<numero> ::= <decimal>
    | <entero>

<modulo> ::= [<configuraciones> <_>] <subrutina>*

<configuraciones> ::= "configuracion" <_> (<configuracion> <_>)* "fin"

<configuracion> ::= "importar" <_> <texto>
    | "integrar" <_> <texto>
    | <nombre> <_> "<-" <_> <numero>
    | <nombre> <_> "<-" <_> <texto>

<subrutina>  ::= <subrutinaCabecera> <_> <subrutinaCuerpo>
```

```
<subrutinaCabecera>  ::= "subrutina" (<_> <modificador>)* <_> <nombre>

<subrutinaCuerpo>  ::= <datos> <_> <algoritmo> <_> "fin"

<modificador> ::= "interna"
    | "primitiva"
    | "conversora"

<nombre> ::= <nombreSimple>

<datos> ::= "datos" <_> (<declaracion> <_>)+

<algoritmo> ::= "algoritmo" <_> (<sentencia> <_>)+

<declaracion> ::= <nombre> <_> "es" <_> <tipo> (<_> <declaracionModificador>)*

<tipo> ::= <nombre> ("(" <genericidad> ")")?

<genericidad> ::= <genericidad> "," <tipo>
    | <genericidad> "," <numero>
    | <tipo>
    | <numero>

<sentencia> ::= <asignacion>
    | <llamada>
    | <repetir>
    | <sicondicional>
    | <mientras>
    | "pausar"

<declaracionModificador> ::= "de" <_> "entrada"
    | "de" <_> "salida"
    | "global"

<asignacion> ::= <nombre> <_> "<-" <_> <expresion>
    | <lvalor> <_> "<-" <_> <expresion>

<llamada> ::= (<lvalor> <_> "." <_>) ? <nombre> <_> "[" <_> (<llamadaAsignacion> <_>)+ "]"

<repetir> ::= "repetir" <_> <expresion> <_> "veces" <_> (<sentencia> <_>)+ "fin"

<mientras> ::= "mientras" <_> <expresion> <_> "hacer" (<sentencia> <_>)+ "fin"

<sicondicional> ::= "si" <_> <expresion> <_> "hacer" <_> (<sentencia> <_>)+ "fin"
    | "si" <_> <expresion> <_> "hacer" <_> (<sentencia> <_>)+ <sinocondicional>
    | "si" <_> <expresion> <_> "hacer" <_> (<sentencia> <_>)+ <sino>

<sinocondicional> ::= "o" <_> "si" <_> <expresion> <_> "hacer" <_> (<sentencia> <_>)+ "fin"
    | "o" <_> "si" <expresion> <_> "hacer" <_> (<sentencia> <_>)+ <sinocondicional>
    | "o" <_> "si" <expresion> <_> "hacer" <_> (<sentencia> <_>)+ <sino>

<sino> ::= "si" <_> "no" <_> "hacer" <_> (<sentencia> <_>)+ "fin"

<llamadaAsignacion> ::= <expresion> <_> "->" <_> <nombre>
    | <nombre> <_> "<-" <_> <expresion>

<expresion> ::= <expresionTercera> <_> <operadorBinarioCuarto> <_> <expresion>
```

```
    | <expresionTercera>

<expresionTercera> ::= <expresionSegunda> <_> <operadorBinarioTercero> <_> <expresionTercera>
    | <expresionSegunda>

<expresionSegunda> ::= <expresionPrimera> <_> <operadorBinarioSegundo> <_> <expresionSegunda>
    | <expresionPrimera>

<expresionPrimera> ::= <operadorUnario> <_> <expresion>
    | <evaluacion> <_> <operadorBinarioPrimero> <_> <expresionPrimera>
    | <evaluacion>

<evaluacion> ::=
    | <evaluacion> <_> "como" <_> <tipo>
    | <evaluacion> <_> "(" <_> <listaAcceso> <_> ")"
    | "verdadero"
    | "falso"
    | <numero>
    | <texto>
    | <letra>
    | <nombre>
    | <listaConstructor>
    | "(" <_> <expresion> <_> ")"

<lvalor> ::= <nombre> ( <_> "(" <_> <listaAcceso> <_> ")" )+

<listaAcceso> ::= <expresion> <_> ("," <_> <expresion>)*

<listaValores> ::= <evaluacion> <_> ("," <_> <listaValores>)*

<listaConstructor> ::= "{" <_> <listaValores> <_> "}"

<operadorUnario> ::= "-"
    | "+"
    | "no"

<operadorBinarioCuarto> ::= "y"
    | "o"

<operadorBinarioTercero> ::= "<"
    | ">"
    | "<="
    | ">="
    | "="
    | "<>"

<operadorBinarioSegundo> ::= "+"
    | "-"

<operadorBinarioPrimero> ::= "*"
    | "/"
    | "%"
```

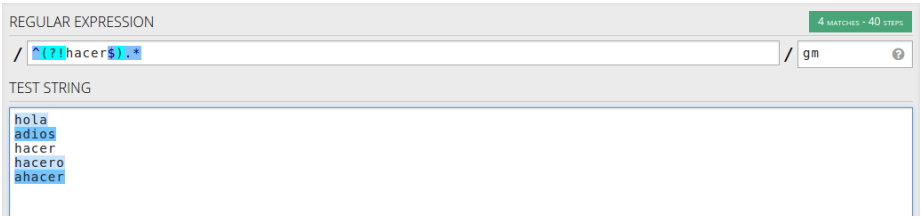
Apéndice B

CodeCombat, imagen de la aplicación

A derecha, un editor de texto con el lenguaje de programación seleccionado. A izquierda, arriba, un lienzo con la situación del puzzle a resolver, y abajo un menú con información adicional (en esta imagen está en blanco tal menú).

Apéndice C

Ejemplo de expresión regular negative lookahead



Arriba, la expresión regular con los modificadores global y multilinea. Abajo, el texto al que se le pasa la expresión regular. Como se puede ver, todo hace match salvo la palabra 'hacer'.